Machine Learning Algorithms

David Picard
École des Ponts ParisTech
david.picard@enpc.fr

February 27, 2025

Contents

Foreword

Beware, this is not a real book! This is just an export of the slides from my lectures on ML in a somewhat printable book format, to which I added a few remarks. The slides are by design mostly empty, as they are ment as a visual support for my discourse during the lecture. As such, a lot of information is missing or not structured in a way that would make sense for a book, and without the notes that every student takes during the lectures, I think it is mostly useless. So rest assured that the most informative bits in here are your handwritten notes.

Also, from a technical point of view, the slides were originally made in python notebooks. This allows me to show and run simplified code of most of the machine learning algorithms that are presented during the lectures. The main idea is also that the students can tweak the parameters and play with the algorithms to better grasp their intended behavior. Of course, a PDF file or a printed version is only but a static version of that, and I strongly encourage you to get you hand on the original notebooks.

Chapter 1

Introduction

1.1 Resources

Books:

Trevor Hastie, Robert Tibshirani, Jerome Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction.

Shai Shalev-Shwartz, Shai Ben-David, Understanding Machine Learning: From Theory to Algorithms*

Kevin P. Murphy, Probabilistic Machine Learning: An Introduction

Francis Bach, Learning Theory from First Principles**

In French: Chloé-Agathe Azencott, Introduction au Machine Learning

Other lectures at ENPC: - Deep Learning, Stat en grande dimension

This lecture uses JAX because I want to keep it investigate and look at how the algorithms work under the hood. In practice there are many high level libraries. Do not reinvent the wheel, but beware that some sell square wheels. References can be found at the end of this book: [Hastie et al., 2009], [Shalev-Shwartz and Ben-David, 2014], [Murphy, 2022], [Murphy, 2023], [Azencott, 2022].

1.2 Setup

- Coupled random variables X, y with unknown pdf $P(X, y)$, $P(X)$ or $P(y)$
- $X \in \mathcal{X}$ input domain
- $y \in \mathcal{Y}$ output domain

We want to find a function f that approximates y from X In practice X and y can be anything for which we have good reasons to believe there is a relationship. For example, X can be the location of a house and y its value, or X can be a press article and y its political orientation. Note that it is not required that the link between X and y is a physical process (ex: X is the characteristics of a steel cable and y its tensile strength), or even that this link is causal (X causes y, e.g., X is a specific gene and y an hereditary condition). We merely suggest that the two variables are *correlated* and that there exists a function f that captures that correlation to predict y from X with reasonable accuracy.

Please make sure that you understand correctly the big leap in methodology that has been made here: **we are no longer trying to explain to world by putting it into equations, we merely hypothesize that two phenomena are somehow related by a mechanism we have no interest in uncovering, but we aim at finding a function that is good enough at predicting one from the other**. There will never ever be any guarantee that our function f is correct. **The best we can do is to find a function f that has a satisfyingly low probability of error**.

1.2.1 Expected Error

- To measure to quality of the approximation: loss function $l(f(x), y)$
  - example: $l(f(X), y) = 1$ if $f(X) \neq y, 0$ else
- Find f that minimizes the average error

$$\mathbb{E}_{\sim X,y}[l(f(X), y)]$$

Recall the definition of the expectation:

$$\mathbb{E}_{\sim X,y}[l(f(X), y)] = \iint l(f(X), y)p(X, y)dXdy$$

This means that in our evaluation, we weight the error by the probability of sampling a couple (X, y), and that the higher this probability, the higher we should consider that couple.

This formulation with the expectation is called *risk*, but there could be many other ways to aggregate the values of the loss function l over all possible couples (X, y). The risk has the benefit of being very intuitive, it means that *on average* (with respect to the probability of observing (X, y)) f does a certain loss.

1.2.2 Two problems

1. $P(X, y)$ is unknown

- Complex phenomenon, no explicit model

2. Finding a minimizer may be difficult

- example: $l(f(X), y) = 1$ if $f(X) \neq y, 0$ else $\Rightarrow$ SAT problem, NP-hard

It is a little abusive to write here that the $0-1$ loss (as it is called) leads to a SAT problem in general. But it is nonetheless easy to see that if $\mathcal{X}$ is a countable set and the relationship between X and y is arbitrary, then minimizing the $0-1$ loss over that entire set require finding a function that satisfies a set of binary expressions. SAT is NP-Complete, meaning that solution can be found via brute force and verified in polynomial time, which is the hardest kind of problem that we can verify quickly.

1.3 Empirical risk minimization

Solving problem #1, $P(X, y)$ is unknown:

If P was known, we would use

$$f(x) = \arg\max_y P(y|x)$$

which is our best guess and would lead to the following error

$$P_e = \int \left(1 - \max_y P(y|x)\right) p(x)dx$$

(Bayes error) This is the lowest achievable error rate.

- If the process is deterministic, then $\max_y P(y|x) = 1$ and perfect prediction can be achieved.

- If the process is intrinsically random (*e.g.*, throw 2 dice, x is the first dice, y is the sum of the dice), then there is some irreducible error.

The Bayes error is worth keeping in mind because machine learning is, again, not about uncovering the golden equation that links X to y - which may not even exist - but to merely find a good enough way to predict y from X. Because we do not make the assumption that there is an exact deterministic causal link, it is more than likely that this irreducible error exist. And as such, the best we can ever hope to do is approaching this irreducible error. Estimate the error instead:

- Training set of examples $\mathcal{A} = \{(X_i, y_i)\}_{i \leq n}$ sampled from $P(X, y)$
- Approximate the expected error by the empirical risk

$$E(f) = \frac{1}{n} \sum_i l(f(X_i), y_i)$$

- Find f that minimizes $E(f)$

$$f^\star = \arg\min_f E(f)$$

Now we can see three of the four ingredients that define a machine learning system: - A training set - A loss function - A family of function from which we have to select the one that minimizes the loss over the training set.

A quick note: the training set has to be sampled i.i.d. from $P(X, y)$ so that the average corresponds to the empirical estimator of the expectation. This is important because is gives a good hint that increasing the size of the training set is in the limit going to attain the true error.

1.3.1 A Bad Example

Consider the function

$$f(X) = \begin{cases} y_i \text{ if } \exists X_i \in \mathcal{A} \text{ such that } X_i = X \\ 0 \text{ else} \end{cases}$$

Obviously

$$E(f) = 0$$

However, f is pretty useless at predicting anything outside of $\mathcal{A}$ This example is very important, we should always have it in mind we thinking about a machine learning problem. It is both the best optimizer of the empirical risk one can find, and the most useless predictor. It highlights the blessing and the curse of machine learning: the blessing is that we make minimal assumptions about the phenomenon and we just need some data, the curse is that we only have these data and it is very easy to be mislead by them.

1.3.2 A not-as-bad example

Points inside a random circle

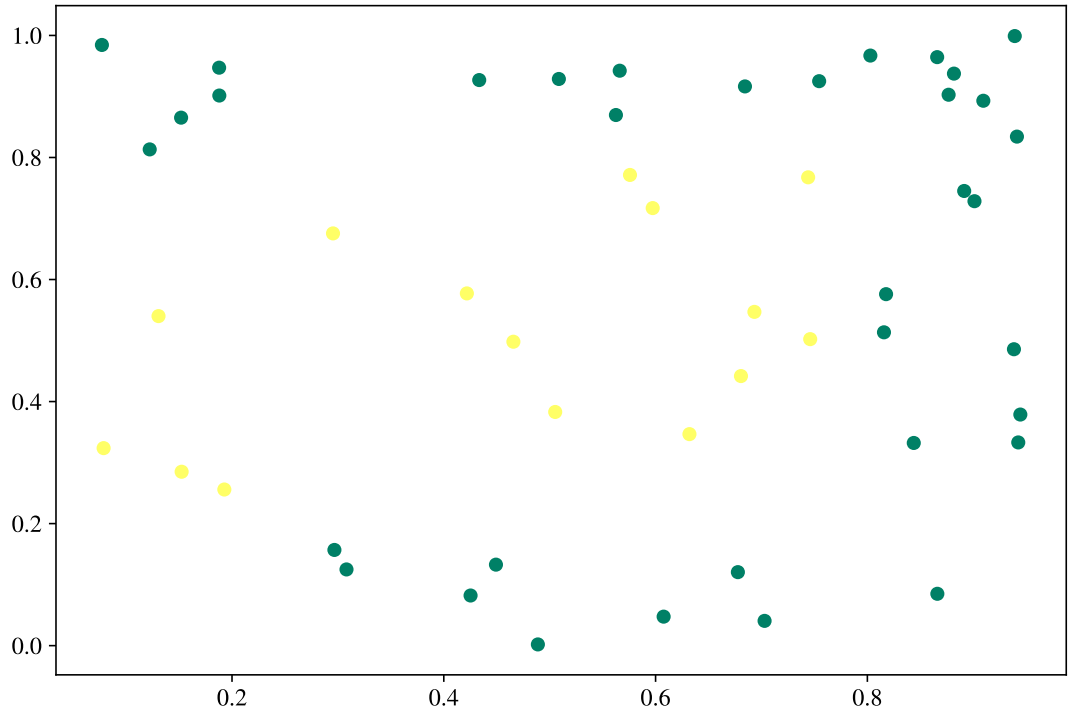
In [66]:

```
def gt(x):
    x1 = x[:,0] > 0.
    x2 = x[:,0] < 0.8
    y1 = x[:,1] > 0.2
    y2 = x[:,1] < 0.8
    return 1*(x1 * x2 * y1 * y2)
```

In [67]:

```
key = jax.random.PRNGKey(0)
key, skey = jax.random.split(key)
X = jax.random.uniform(skey, (50, 2))
y = gt(X)

plt.scatter(X[:,0], X[:,1], c=y)
```

```
<matplotlib.collections.PathCollection at 0x74d478c2da80>
```



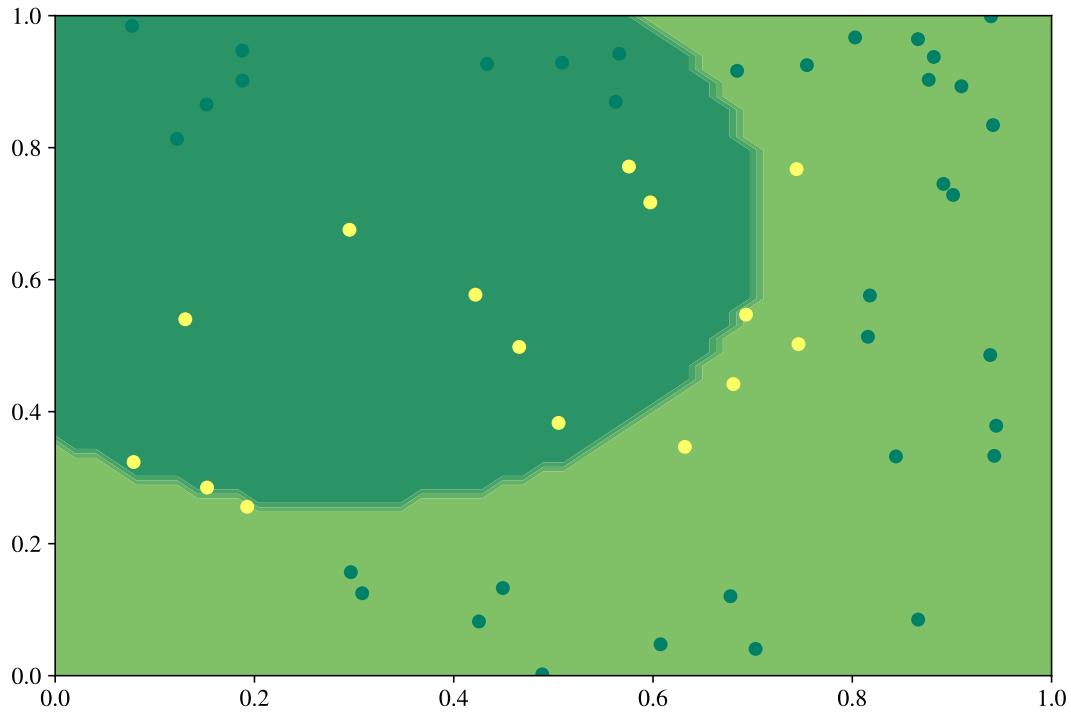
In [68]:

```
class CirclePredictor:
    def __init__(self, key):
        key, skey = jax.random.split(key)
        self.c = jax.random.uniform(key, (2,))
        self.r = jax.random.uniform(skey)
    def __call__(self, X):
        return jnp.sign(1*((X[:,0] - self.c[0])**2 + (X[:,1] - self.c[1])**2 <
self.r**2))

def loss(y_pred, y_true):
    return (1-(y_pred==y_true)).mean()
```

In [69]:

```
key, skey = jax.random.split(key)
pred = CirclePredictor(skey)

t = 50; tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
levels=jnp.linspace(-1.5, 1.5, 10)
y_pred = pred(xx).reshape(t, t)
plt.contourf(xv, yv, -y_pred, levels=levels); plt.scatter(X[:,0], X[:,1], c=y)
```

```
<matplotlib.collections.PathCollection at 0x74d478c66f50>
```

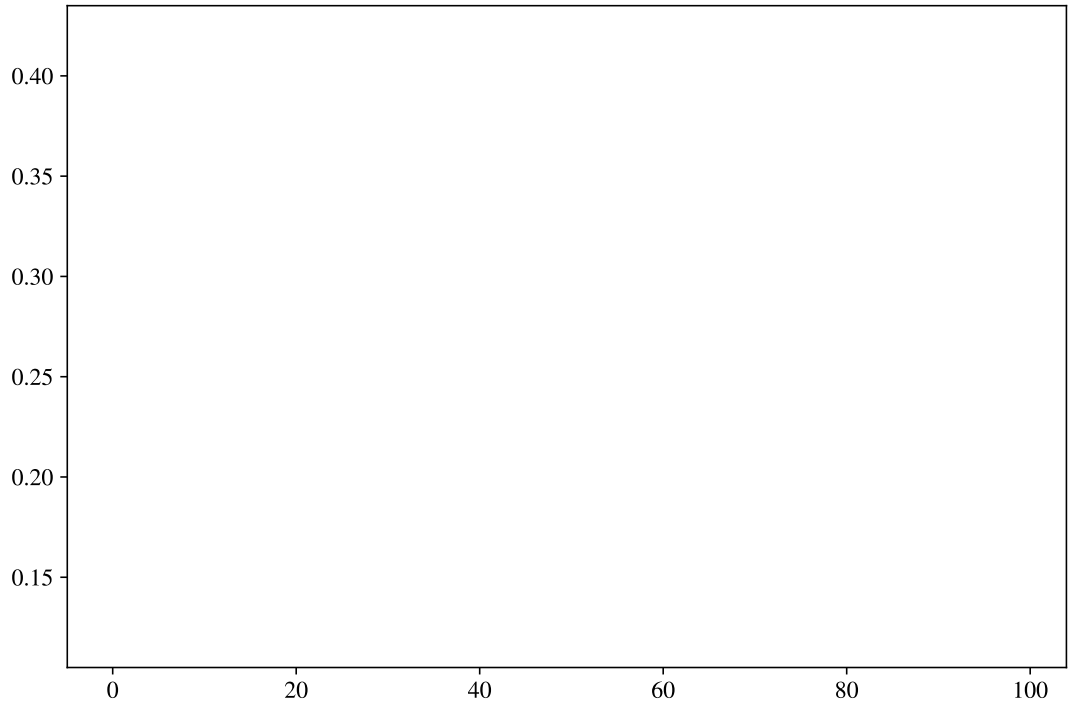


1.3.3 (Randomly) Searching for a good f

In [7]:

```
fig = plt.figure()
camera = Camera(fig)
key = jax.random.PRNGKey(7)
l_min = 20; f_best = None
le = []
for i in range(100):
    key, skey = jax.random.split(key)
    f = CirclePredictor(skey)
    l = loss(f(X), y)
    if l < l_min:
        l_min = l; f_best = f
    le.append(l_min)
    plt.plot(le, '-k'); camera.snap()
animation = camera.animate()
HTML(animation.to_html5_video())
```

```
<IPython.core.display.HTML object>
```

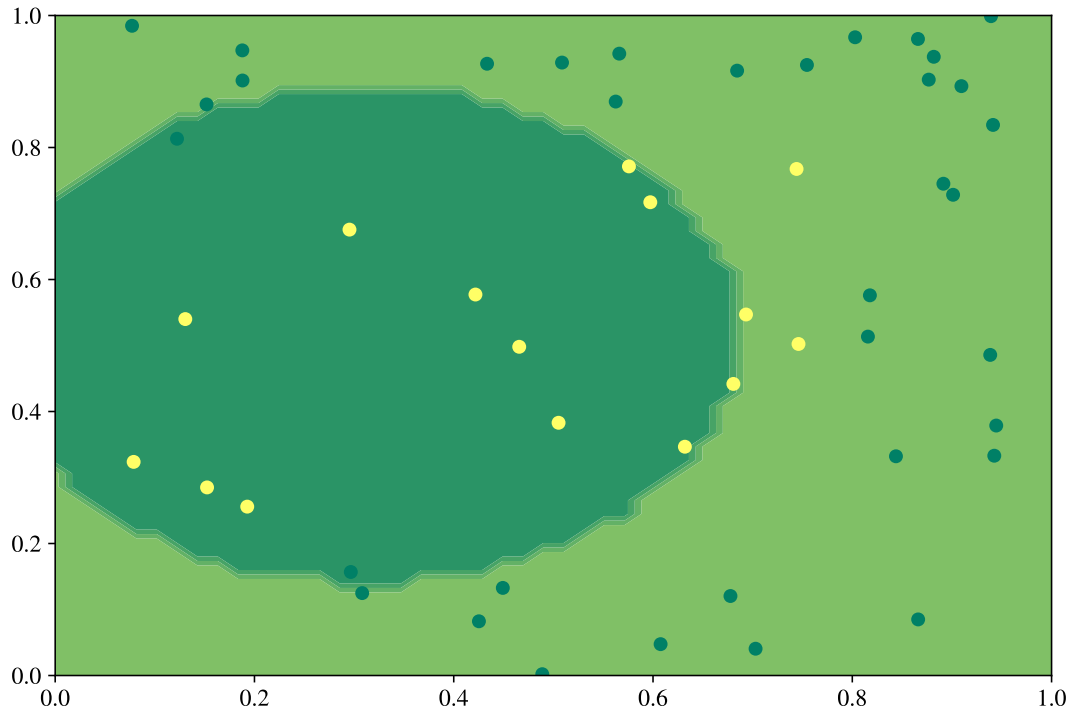


1.3.4 Are we lucky

In [8]:

```
t = 50
tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
levels=jnp.linspace(-1.5, 1.5, 10)
y_pred = f_best(xx).reshape(t, t)
plt.contourf(xv, yv, -y_pred, levels=levels)
plt.scatter(X[:,0], X[:,1], c=y)
```

```
<matplotlib.collections.PathCollection at 0x74d4d955db70>
```



1.4 Generalization

- We know f is good on $\mathcal{A}$ (at least better than other)
- We don't know if it's good on other samples

The difference between the expected risk and the empirical risk is know as the *generalization gap*

- A function that performs poorly on unseen data compared to training data is *overfitting*

How do we know if f is overfitting? - Measuring the error on $\mathcal{A}$ is not informative $\Rightarrow$ Split the examples into training and evaluation sets Generalization is what differenciates machine learning from statistics with optimization. In statistics, we are not making prediction, we are making characterization and there is no generalization. In optimization, the optimal solution is the goal, but in machine learning we know the optimal solution can be pointless as what we really want is not a low error on the samples we already have, but a low error rate on the samples we do not have yet.

In some sense, one could say that machine learning consists in optimizing the wrong problem with the wrong model, and it is indeed exactly that! But it is the best problem we can optimize and we will use it to find out the best model we can, and it is already something better than nothing.

1.4.1 Let's try

Check whether the error is the same on different sets of samples.

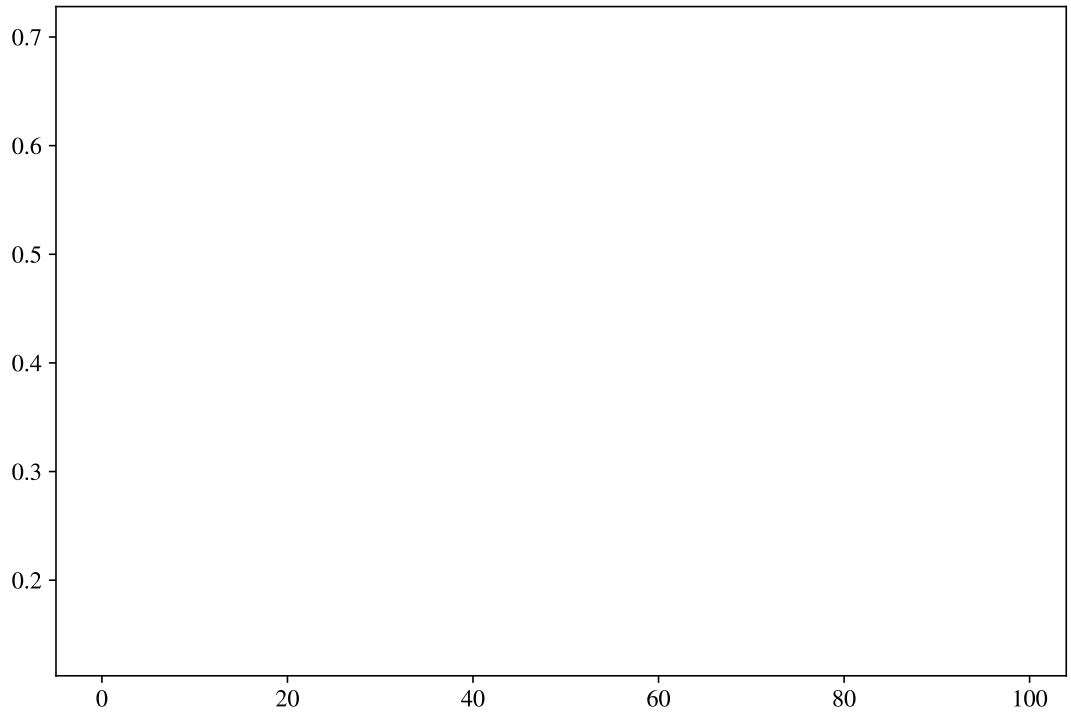
In [9]:

```
key = jax.random.PRNGKey(6)
Xt = jax.random.uniform(key, (50, 2))
yt = gt(Xt)
```

In [10]:

```
fig = plt.figure()
camera = Camera(fig)
key = jax.random.PRNGKey(1)
l_min = 20; f_best = None
le = []
lt = []
for i in range(100):
    key, skey = jax.random.split(key)
    f = CirclePredictor(skey)
    l = loss(f(X), y)
    if l < l_min:
        l_min = l; f_best = f
    le.append(l)
    lt.append(loss(f(Xt), yt))
    plt.plot(le, '-k'); plt.plot(lt, '-r'); camera.snap()
animation = camera.animate()
HTML(animation.to_html5_video())
```

```
<IPython.core.display.HTML object>
```



Both errors are correlated (the contrary would be very worrying), but some extreme values may be very different. We are at the risk of selecting an f because we were lucky.

1.5 k-NN: A Better learning machine

k nearest neighbor: prediction is a vote among k nearest elements of the training set

Example $1-NN$:

$$f(x) = y_i \text{ s.t. } i = \arg\min_{x_j \in \mathcal{A}} \|x - x_j\|^2$$

- Memorizes the entire training set
- Does o empirical error on $\mathcal{A}$

k-NN is one of the oldest algorithm of machine learning (see [Cover and Hart, 1967] and [Pelillo, 2014]), and certainly one of the most intuitive. Interestingly, it took much longer to formalize than more complicated methods (such as quadratic discriminant analysis). The key aspect of k-NN is that it is solely based on data: there is no model, nothing that tries to put the phenomenon in equation. The only assumption that we implicitely make is that the phenomenon has a topological property, *id est*, that nearby X in the sense of the chosen distance have similar y. In practice, this is often the case. Or to put it differently, if it were not the case and there is not regularity in the way X and y are related, then our chances of finding a good predictor are incredibly slim.

The training procedure of a k-NN is very simple, we just have to memorize the entire training set. This of course becomes increasingly costly as the training set size grows, but in practical terms, storage is not a very challenging issue. Prediction is more problematic as it requires find the nearest neighbors which has a complexity linear with the size of the training set. There are approximate search tools that have become very effective, but it is nonetheless a burden that limits in practice the applicability of k-NN to large datasets.

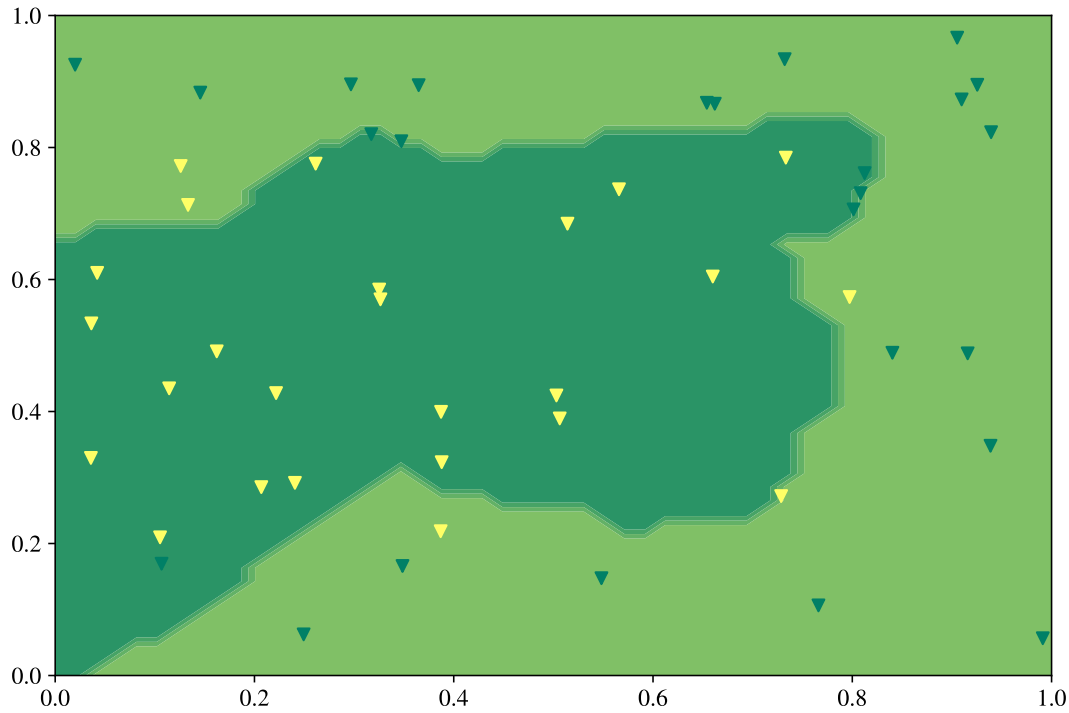
In [70]:

```
class FirstNearestNeighbor:
    def __init__(self, X, y):
        self.X = X
        self.y = y
    def __call__(self, x):
        dist = ((self.X[None,:,:] - x[:,None,:])**2).sum(axis=2) # broadcast to B x n x
dim
        index = jnp.argmin(dist, axis=1)
        return self.y[index]
```

In [74]:

```
nn = FirstNearestNeighbor(X, y)
t = 50
tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
levels=jnp.linspace(-1.5, 1.5, 10)
y_pred = nn(xx).reshape(t, t)
plt.contourf(xv, yv, -y_pred, levels=levels)
#plt.scatter(X[:,0], X[:,1], c=y)
plt.scatter(Xt[:,0], Xt[:,1], marker='v', c=yt)
```

```
<matplotlib.collections.PathCollection at 0x74d478c87850>
```



1.5.1 Generalization bound

What is the expected error of 1-NN in the limit?

Theorem [Cover and Hart, 1967]: Let X be a metric space. Let p_1 and p_2 be such that with probability 1, x is either 1) a continuity point of p_1 and p_2, or 2) a point on non-zero probability measure. Then the NN risk R (probability of error) has the bounds:

$$R^\star \leq R \leq 2R^\star(1 - R^\star)$$

With $R^\star$ the Bayes error (irreducible error) $R^\star = E[\min_j \sum_i p_i(x)L(i,j)]$.

Proof Lemma: Let x'_n denote the nearest neighbor of x in the set $\{x_0, \ldots, x_n\}$, then $x'_n \to x$ with probability one (continuity + point measure).

- Pointwise error: $r(x, x'_n) = P[y = 1|x]P[y' = 2|x'_n] + P[y = 2|x]P[y' = 1|x'_n] = p_1(x)p_2(x'_n) + p_2(x)p_1(x'_n)$
- By lemma: $r(x, x'_n) \to 2p_1(x)p_2(x) = 2p_1(x)(1 - p_1(x))$
- Bayes pointwise error: $r^\star(x) = \min\{p_1(x), 1 - p_1(x)\}$ (lowest probability or error)
- $r(x) \leq 2r^\star(x)(1 - r^\star(x))$ and take the expectation over x: $R = E[2r^\star(x)(1 - r^\star(x))]$
- $R = E[r^\star(x)] + E[r^\star(x)(1 - 2r^\star(x))] \geq E[r^\star(x)] = R^\star$
- $R = 2R^\star(1 - R^\star) - 2\text{Var}(r^\star(x)) \leq 2R^\star(1 - R^\star)$

1.5.2 What is the effect of k?

We can increase the value k to take more than the first neighbor into account when making prediction.

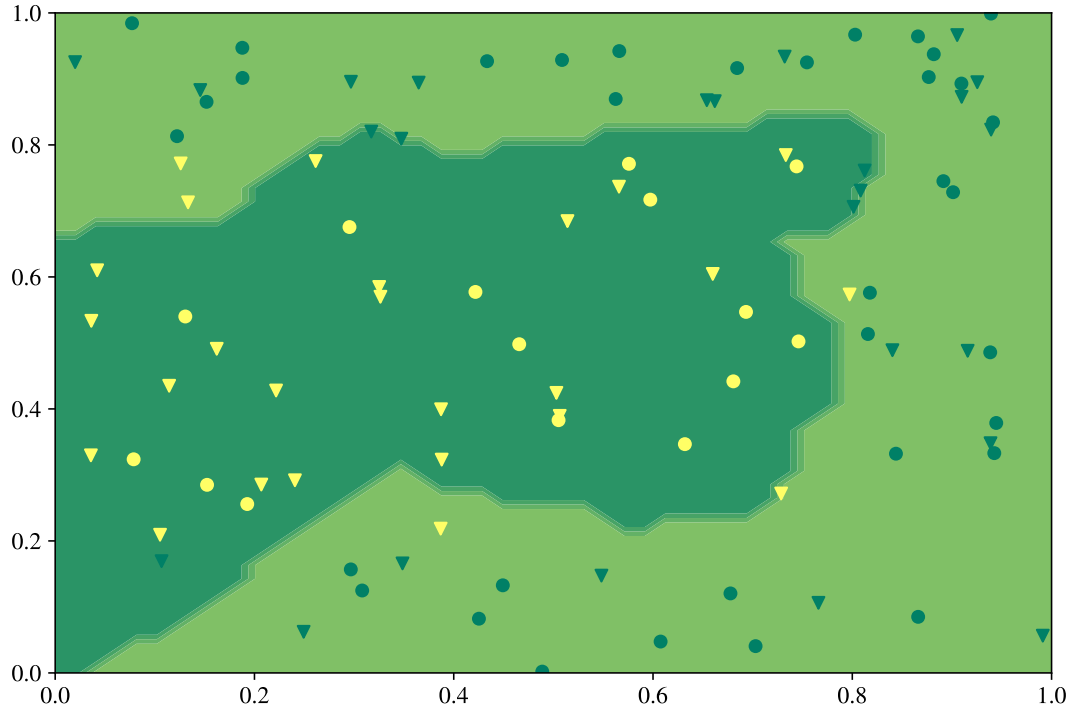
In [13]:

```
class KNearestNeighbor:
    def __init__(self, X, y, k=1):
        self.X = X
        self.y = y
        self.k = k
    def __call__(self, x):
        dist = ((self.X[None,:,:] - x[:,None,:])**2).sum(axis=2) # broadcast to B x n x
dim
        indices = jnp.argsort(dist, axis=1)
        yp = 1*((self.y[indices[:,0:self.k]]).sum(axis=1) > self.k//2)
        return yp
```

In [14]:

```
nn = KNearestNeighbor(X, y, k=1)
t = 50
tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
levels=jnp.linspace(-1.5, 1.5, 10)
y_pred = nn(xx).reshape(t, t)
plt.contourf(xv, yv, -y_pred, levels=levels)
plt.scatter(X[:,0], X[:,1], c=y), plt.scatter(Xt[:,0], Xt[:,1], marker='v', c=yt)
```

```
(<matplotlib.collections.PathCollection at 0x74d4d85133a0>,
 <matplotlib.collections.PathCollection at 0x74d4d853abf0>)
```

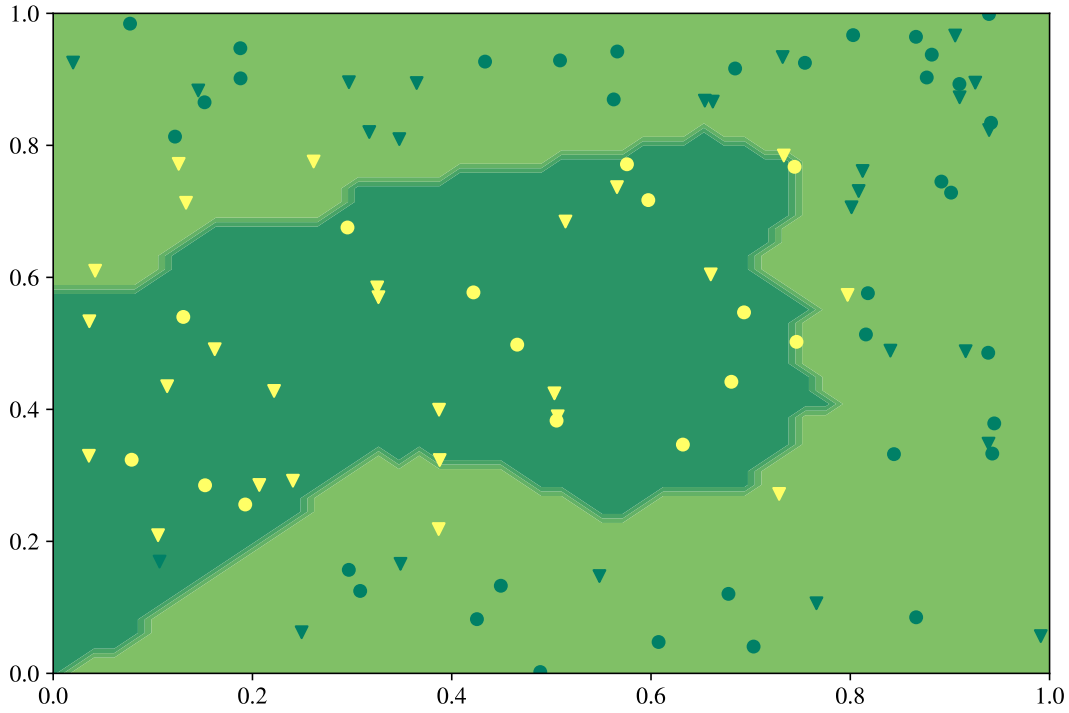


In [15]:

```
nn = KNearestNeighbor(X, y, k=2)
t = 50
tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
```

```
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
levels=jnp.linspace(-1.5, 1.5, 10)
y_pred = nn(xx).reshape(t, t)
plt.contourf(xv, yv, -y_pred, levels=levels)
plt.scatter(X[:,0], X[:,1], c=y), plt.scatter(Xt[:,0], Xt[:,1], marker='v', c=yt)
```

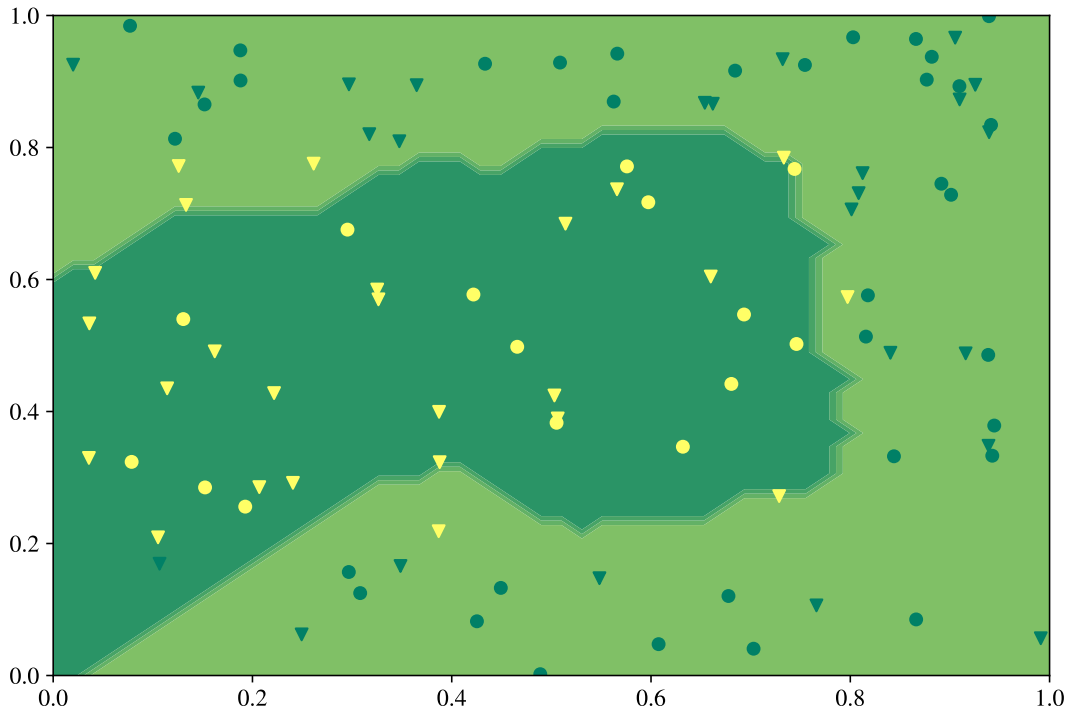
```
(<matplotlib.collections.PathCollection at 0x74d4d8aacc10>,
 <matplotlib.collections.PathCollection at 0x74d4d8aea890>)
```



In [16]:

```
nn = KNearestNeighbor(X, y, k=3)
t = 50
tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
levels=jnp.linspace(-1.5, 1.5, 10)
y_pred = nn(xx).reshape(t, t)
plt.contourf(xv, yv, -y_pred, levels=levels)
plt.scatter(X[:,0], X[:,1], c=y), plt.scatter(Xt[:,0], Xt[:,1], marker='v', c=yt)
```

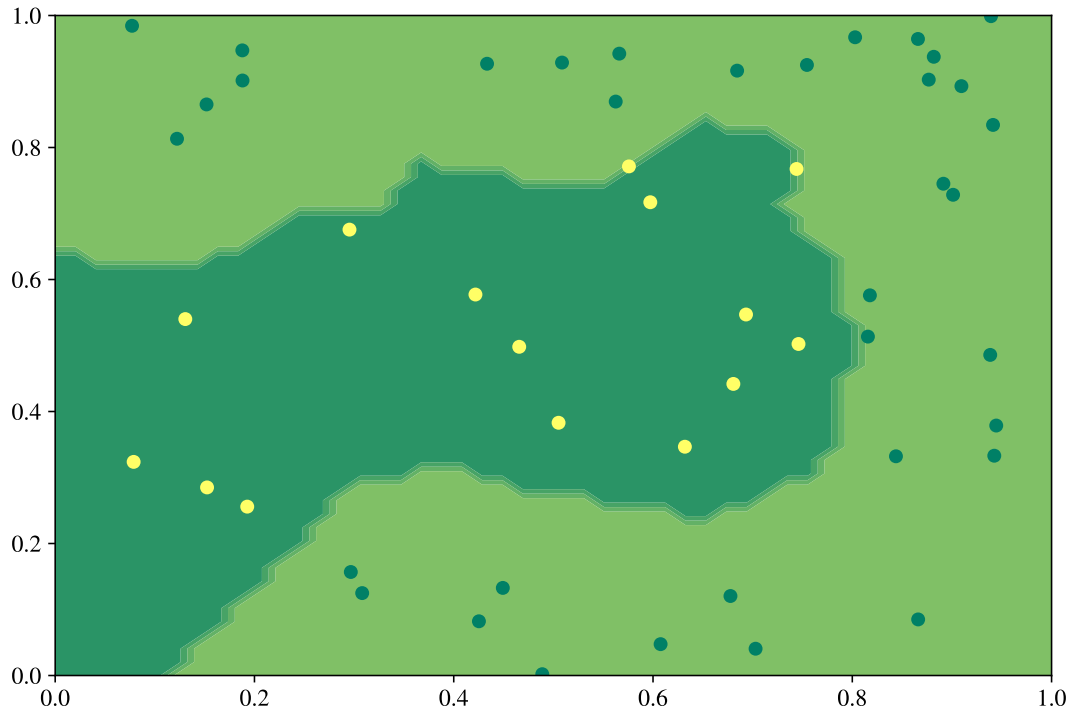
```
(<matplotlib.collections.PathCollection at 0x74d4d9f0bd90>,
 <matplotlib.collections.PathCollection at 0x74d4d9f34df0>)
```



In [17]:

```
nn = KNearestNeighbor(X, y, k=5)
t = 50
tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
levels=jnp.linspace(-1.5, 1.5, 10)
y_pred = nn(xx).reshape(t, t)
plt.contourf(xv, yv, -y_pred, levels=levels)
plt.scatter(X[:,0], X[:,1], c=y)#, plt.scatter(Xt[:,0], Xt[:,1], marker='v', c=yt)
```

```
<matplotlib.collections.PathCollection at 0x74d4d85dd870>
```



1.5.3 Model selection

How do we select k? - They all do 0 error on $\mathcal{A}$

- We can split $\mathcal{A}$ in 2:
  - One for training each k-NN: *training* set
  - One for evaluating each k-NN: *validation* set

Since the validation set is used to select a model, it cannot be used to give us an idea of the expected risk Standard 3-split procedure: *train, validation, test*

- Train on *train*
- Perform model selection on *validation*
- Evaluate on *test*

This is maybe the most important bit of methodology there is about machine learning in general. Because we are using data and only that for everything that we do, we have to be very careful to not use them twice or we inevitably get a biased result. We usually need data three times: - During training to build our function f (in the case of k-NN, we just memorize it). - During model selection to select the best predictor among several algorithmic choices. - During evaluation to estimate how good our selected model will perform in the wild.

It is of the utmost importance that one of these steps does not contaminate the others. For example, if training examples leak into the validation set, then it can change the outcome of model selection leading to selecting the wrong model. Similarly, if training examples leak into the test set (or *vice versa*), then we will over-estimate the performances of the model.

We show below the error rates for various values of k, knowing that we can only look at the validation for selecting the best k, and then look at the test to estimate how it will perform in reality.

In [18]:

```
key = jax.random.PRNGKey(33)
Xv = jax.random.uniform(key, (50, 2))
yv = gt(Xt)
```

In [19]:

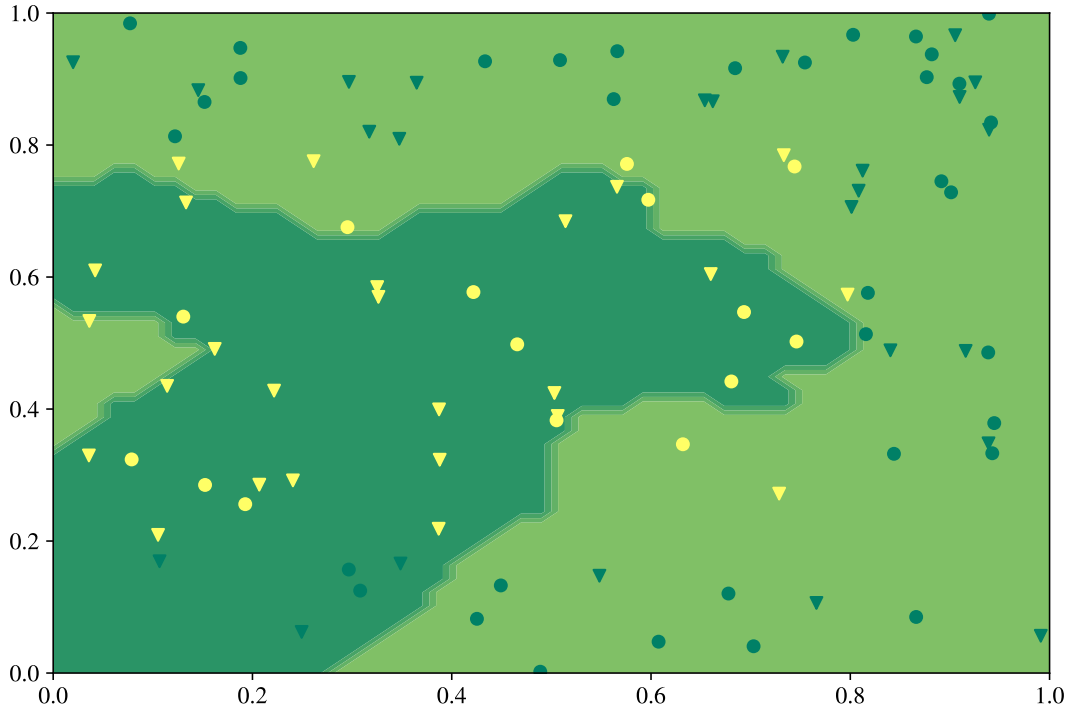
```
lv = []; lt = []; lr = []
for k in range(1,30):
    nn = KNearestNeighbor(X, y, k)
    lv.append(loss(nn(Xv), yv))
    lt.append(loss(nn(Xt), yt))
    lr.append(loss(nn(X), y))
plt.plot(lv, '-k'), plt.plot(lt, '-r'), plt.plot(lr, '-g')
```

```
([<matplotlib.lines.Line2D at 0x74d4b9155f30>],
 [<matplotlib.lines.Line2D at 0x74d4b9156740>],
 [<matplotlib.lines.Line2D at 0x74d4b9156f50>])
```

In [20]:

```
nn = KNearestNeighbor(X, y, k=17)
t = 50
tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
levels=jnp.linspace(-1.5, 1.5, 10)
y_pred = nn(xx).reshape(t, t)
plt.contourf(xv, yv, -y_pred, levels=levels)
plt.scatter(X[:,0], X[:,1], c=y), plt.scatter(Xt[:,0], Xt[:,1], marker='v', c=yt)
```

```
(<matplotlib.collections.PathCollection at 0x74d4e40c3580>,
 <matplotlib.collections.PathCollection at 0x74d4b91f3ac0>)
```



1.6 Statistical fluke?

Knowing that f does ϵ expected error, what is the probability that f has an empirical error of η or less on a dataset of size n?

- Probability that f does *exactly* m error over n samples

$$\binom{n}{m}\epsilon^m(1-\epsilon)^{n-m}$$

- Probability that f does m or less error over n samples

$$\sum_{k=1}^{m}\binom{n}{k}\epsilon^k(1-\epsilon)^{n-k}$$

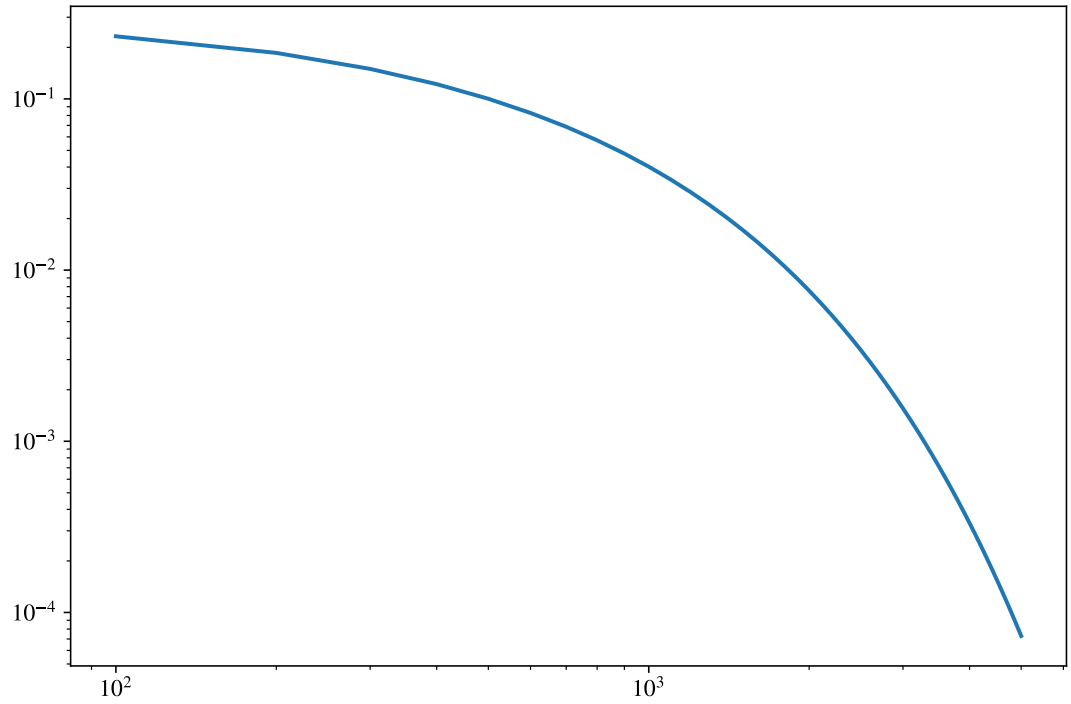
For η observe error rate

$$\sum_{k=1}^{\lfloor \eta n \rfloor}\binom{n}{k}\epsilon^k(1-\epsilon)^{n-k}$$

In [64]:

```
def Pn_of_eta_given_eps(n, eta, eps):
    p = 0
    for k in range(int(eta*n)):
        p += scipy.special.comb(n, k) * eps**k * (1-eps)**(n-k)
    return p
x = range(100, 5100, 100)
p = [Pn_of_eta_given_eps(i, 0.03, 0.04) for i in x]
plt.loglog(x, p)
```

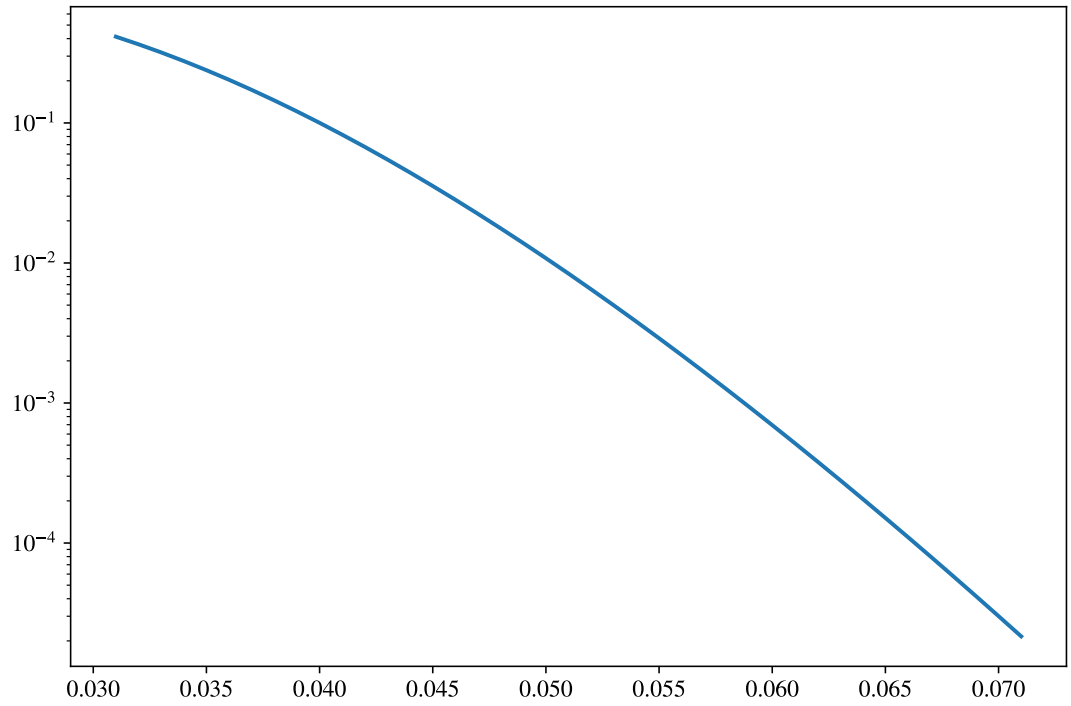
```
[<matplotlib.lines.Line2D at 0x74d480008250>]
```



In [59]:

```
x = 0.031+0.001*jnp.arange(0, 41, 1)
p = [Pn_of_eta_given_eps(500, 0.03, i) for i in x]
plt.semilogy(x, p)
```

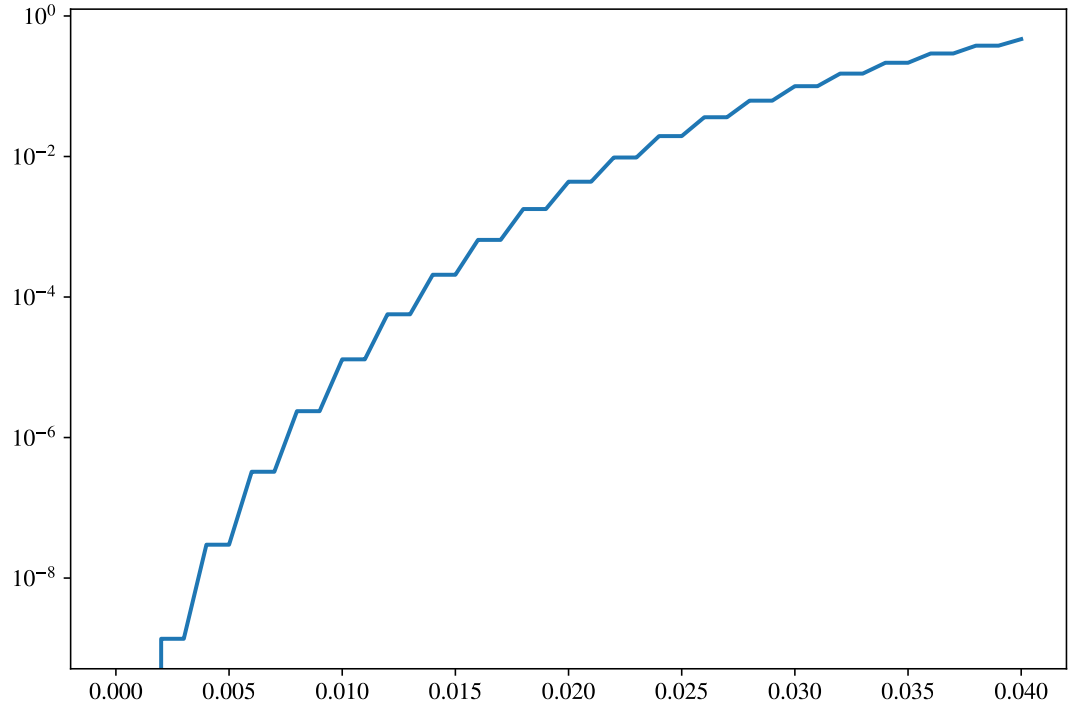
```
[<matplotlib.lines.Line2D at 0x74d478649900>]
```



In [62]:

```
x = 0.001*jnp.arange(0, 41, 1)
p = [Pn_of_eta_given_eps(500, i, 0.04) for i in x]
plt.semilogy(x, p)
```

```
[<matplotlib.lines.Line2D at 0x74d4790bcac0>]
```



Although these small curves may not seem impressive, they are a reminder that there is always a chance that we get a specific training set that over- or under- estimates the actual error rate.

In a recent technical report, I showed that deep learning methods are sensitive to the choice of the pseudorandom number generator initialization (see [Picard, 2021]). Obvisouly, changing the random seed should not affect the results significantly, and if they do, it is only because we have sampled a specific training set that happens to be better for the model obtained with that particular seed. So instead of sampling training set, sampling random seeds achieves the same effect. In that report, I argue that most of the changes that are made during the development of a machine learning system have the same effect: they are inconsequential with respect to the problem and yet they may improve the results just because they correspond to sampling a new training set that may be a lucky one.

We should always keep that in mind. As D. Knuth put it in *The Art of Computer Programming*, *"premature optimization is the root of all evil"*, and the corresponding statement in machine learning would be that premature optimization of the error rate without proper validation and model selection is the root of all evil.

1.6.1 Cross-validation

Split the data into several training-validation sets and average the error

- Random split: perform r random splits of x% training $(1-x)$% validation (typically 80/20)

- K-fold: split in k subsets and perform k permutations $k-1$ sets for training, 1 set for validation

Select model that has lowest average validation error and evaluate on test

- Variance gives an idea of the relevance of the selection process

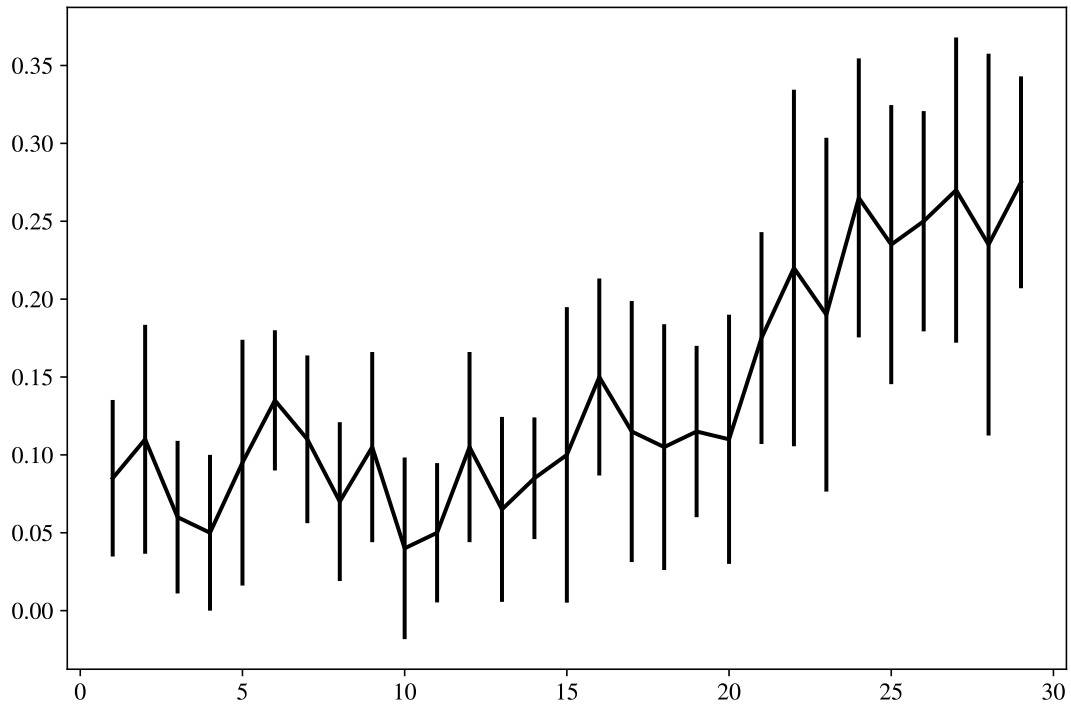
Cross validation is the proper way of developping a machine learning system. It takes into account the variability due to sampling a specific training set, and it gives a confidence associated to the error rate.

In [23]:
```
key = jax.random.PRNGKey(4) # chosen by a fair dice roll
X = jax.random.uniform(key, (100, 2))
y = gt(X)
```

In [24]:
```
def randomSplit(key, X, y, train_part=0.8):
    n = X.shape[0]
    n_train = int(train_part*n); n_test = n - n_train
    p = jax.random.permutation(key, n)
    X_train = X[p[0:n_train], :]; y_train = y[p[0:n_train]]
    X_val = X[p[n_train:],:] ; y_val = y[p[n_train:]]
    return X_train, y_train, X_val, y_val
```

In [25]:
```
key = jax.random.PRNGKey(32)
l = []
for k in range(1, 30):
    lk = []
    for s in range(10):
        key, skey = jax.random.split(key)
        X_train, y_train, X_val, y_val = randomSplit(skey, X, y)
        nn = KNearestNeighbor(X_train, y_train, k=k)
        lk.append(loss(nn(X_val), y_val))
    l.append(lk)
l = jnp.asarray(l)
plt.errorbar(range(1,30), l.mean(axis=1), l.std(axis=1), fmt='-k')
```

```
<ErrorbarContainer object of 3 artists>
```



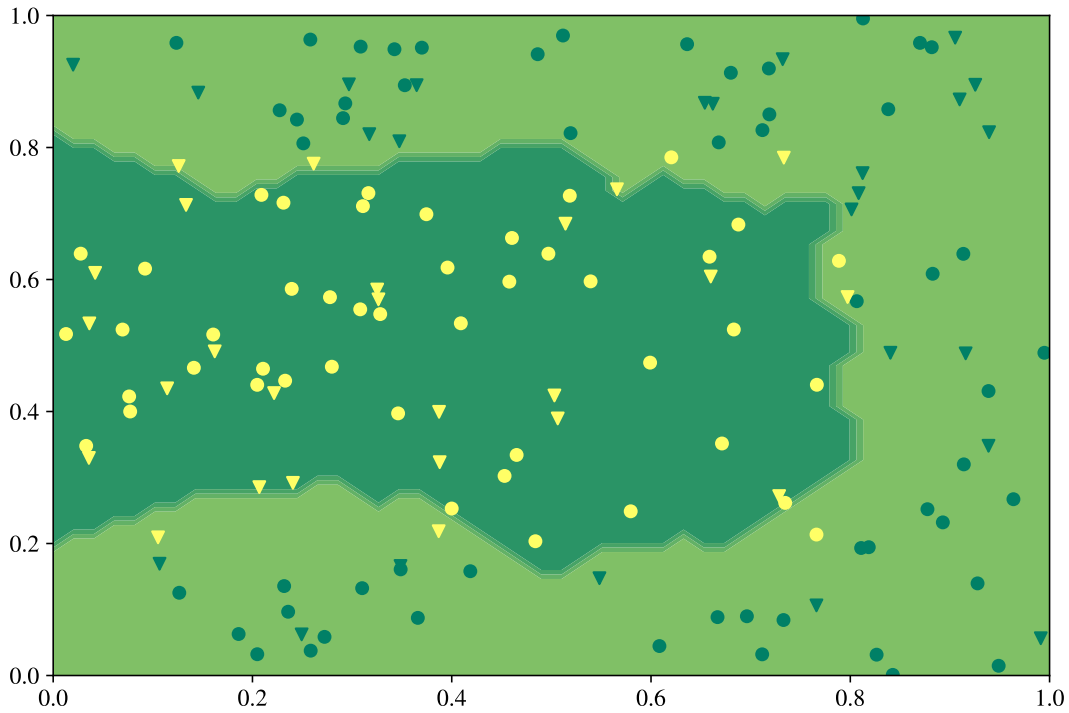
1.6.2 Full training

Once hyperparameters are selected, train on full training set, eval on test

In [26]:

```
nn = KNearestNeighbor(X, y, k=4)
t = 50
tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
levels=jnp.linspace(-1.5, 1.5, 10)
y_pred = nn(xx).reshape(t, t)
plt.contourf(xv, yv, -y_pred, levels=levels)
plt.scatter(X[:,0], X[:,1], c=y), plt.scatter(Xt[:,0], Xt[:,1], marker='v', c=yt)
```

```
(<matplotlib.collections.PathCollection at 0x77e707ae90c0>,
 <matplotlib.collections.PathCollection at 0x77e707b30850>)
```



1.6.3 Conclusion on validation in ERM

- Low training error does not imply generalization (*e.g.*, k-NN)
- A single run training/validation can just be lucky
- Model selection using cross-validation
- Final performance evaluation on a test set

1.7 Finding f is hard

Solving problem #2, finding a good f is hard:

- 0-1 loss is difficult to optimize, alternatives?

The main idea is to find a proxy for the 0-1 loss that is easier to optimize, for example something that leads to a well known optimization problem. Idealy, we want the proxy to upper-bound the 0-1 loss because then we known that the true error rate (the 0-1 loss) is below the attained value, but in practice it is not always required because even if we cannot easily optimize the 0-1 loss, we can however measure it and make sure that the solution to the optimization problem is indeed good in terms of error rate even without theoretical guarantees.

1.7.1 Regression

- $\mathcal{Y}$ is continuous

$$MSE : (y - f(X))^2, \quad MAE : |y - f(x)|$$

- Vector case: any norm of $y - f(X)$

This is probably the most used form of approximation of the 0-1 loss, but it makes one very important hidden assumption in that it considers some structure for $\mathcal{Y}$. Indeed, by assuming that we can use some metric to measure the error, we implicitly say that some targets y are closer together, which may not always be the case.

1.7.2 Classification

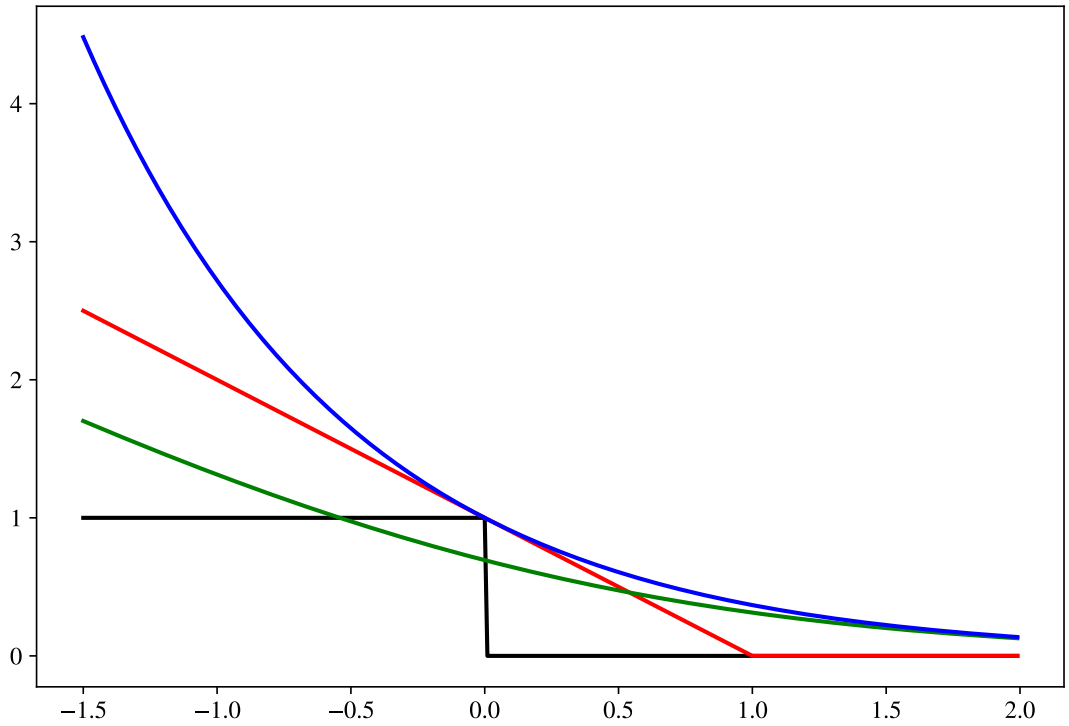
- $\mathcal{Y}$ is categorical $\rightarrow$ continuous relaxation, then decision with $\text{sign}(f(X))$
- Simple binary case: $\mathcal{Y} = \{-1; 1\}$
  - hinge loss: $\max(0, 1 - yf(X))$
  - log loss: $\log(1 + e^{-yf(X)})$
  - exp loss: $e^{-yf(X)}$

Classification is the other big category of learning and in contrast to regression it does not make an assumption about $\mathcal{Y}$. However, since elements in $\mathcal{Y}$ cannot be ordered, we have to design loss functions that will introduce an ordering such that we can perform the optimization. We show below the shape of popular loss functions.

In [27]:

```
t = jnp.arange(-1.5, 2, 0.01)
plt.plot(t, 1-(jnp.sign(t)==1), '-k')
plt.plot(t, jnp.maximum(0, 1 - t), '-r')
plt.plot(t, jnp.log(1+jnp.exp(-t)), '-g')
plt.plot(t, jnp.exp(-t), '-b')
```

```
[<matplotlib.lines.Line2D at 0x77e70799cb20>]
```



1.7.3 Turning ERM into an optimization problem

Ellipse classifier with parameter c_1, c_2, a, b:

$$f(X) = 1 - (a(X_1 - c_1)^2 + b(X_2 - c_2)^2)$$

In matrix form

$$f(X) = 1 - (X - C)^T A(X - C)$$

Using MSE

$$\min_{A,C} \sum_x (y - 1 + (x - C)^T A(x - C))^2$$

- Use optimization formulation to get closed form solution (*e.g.*, KKT)
- Use optimization techniques to get approximate solution (*e.g.*, interior points, cutting planes)
- Use gradient descent (it always gets you a better solution than random)

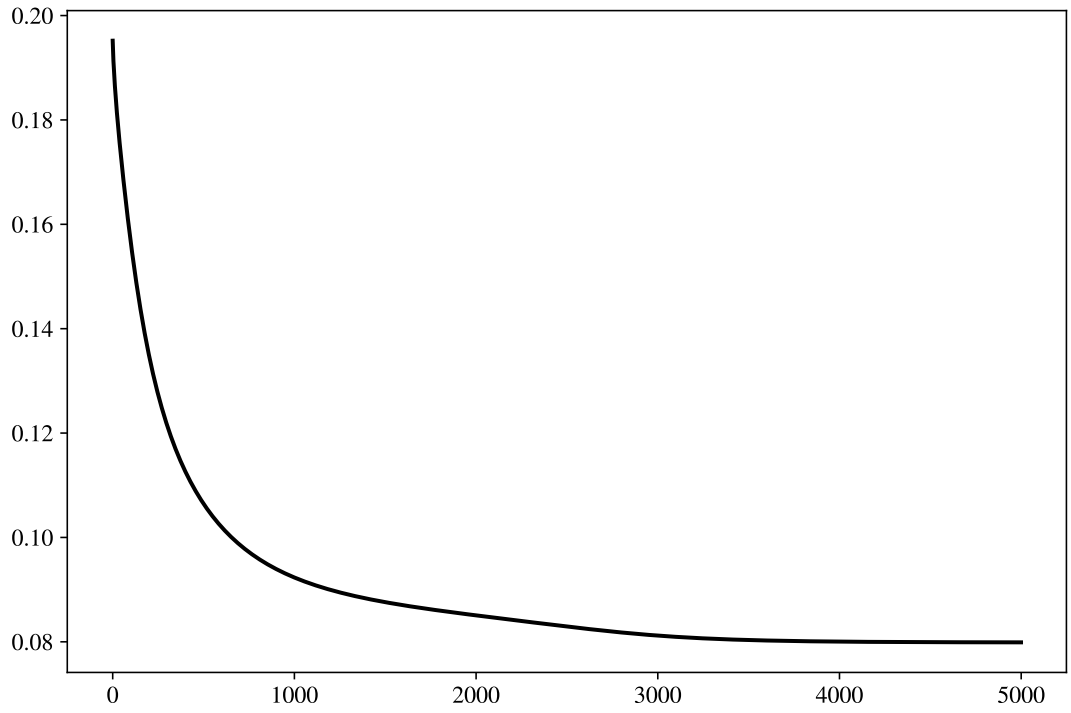
In [28]:

```
def mse(y_hat, y):
    return ((y-y_hat)**2).mean()

def circle(x, a, c):
    xc = x - c[None, :] # broadcast to n x 2
    return 1 - (a*xc**2).sum(1) # sum on axis=1

def loss(a, c, x, y):
    y_hat = circle(x, a, c)
    return mse(y_hat, y)

@jax.jit
def update(a, c, x, y):
    da, dc = jax.grad(loss, argnums=(0,1))(a, c, x, y)
    return a - 0.1 * da, c - 0.1 * dc
```

In [29]:

```
key = jax.random.PRNGKey(32)
key, skey = jax.random.split(key)
c = jax.random.uniform(key, (2,))
a = jnp.ones(2)
l = []
for t in range(5000):
    a, c = update(a, c, X, y)
    l.append(loss(a, c, X, y))
plt.plot(l, '-k')
```

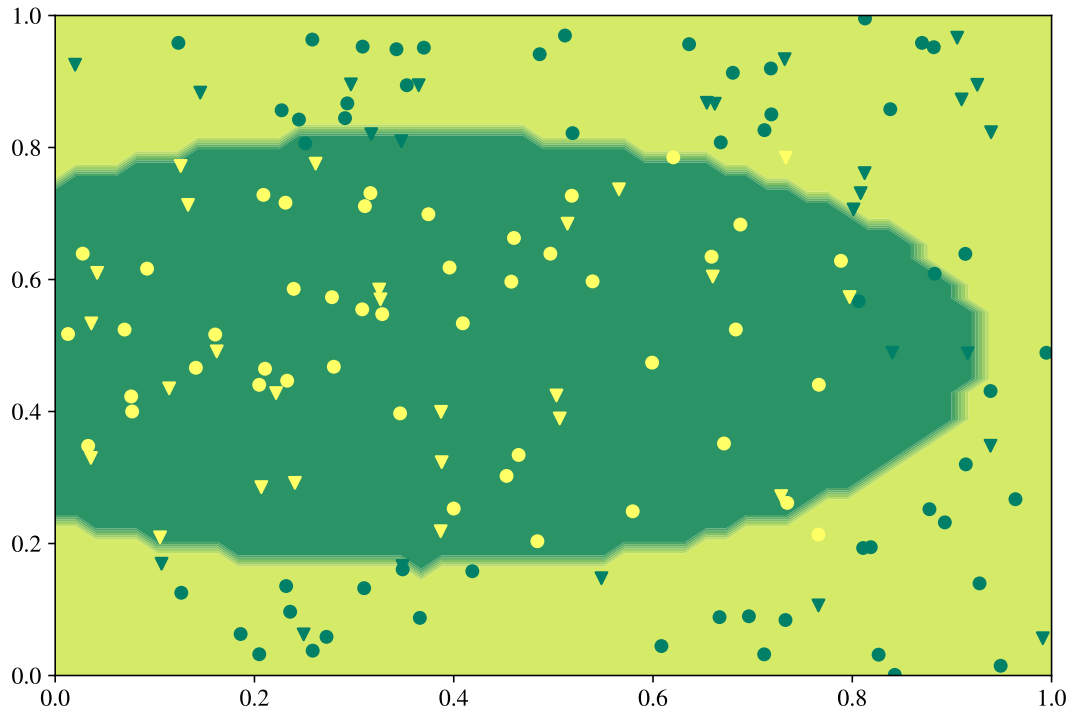
```
[<matplotlib.lines.Line2D at 0x77e7078cf280>]
```



In [30]:

```
t = 50
tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
levels=jnp.linspace(-1.5, 1.5, 10)
y_pred = jnp.sign(circle(xx, a, c)-0.5).reshape(t, t)
plt.contourf(xv, yv, -y_pred, levels=levels)
plt.scatter(X[:,0], X[:,1], c=y), plt.scatter(Xt[:,0], Xt[:,1], marker='v', c=yt)
```

```
(<matplotlib.collections.PathCollection at 0x77e707740d30>,
 <matplotlib.collections.PathCollection at 0x77e707741180>)
```



Rectangle → switch from ℓ_2 to ℓ_∞ norm

$$f(X) = 1 - \max(a(X_1 - c_1)^2, b(X_2 - c_2)^2)$$

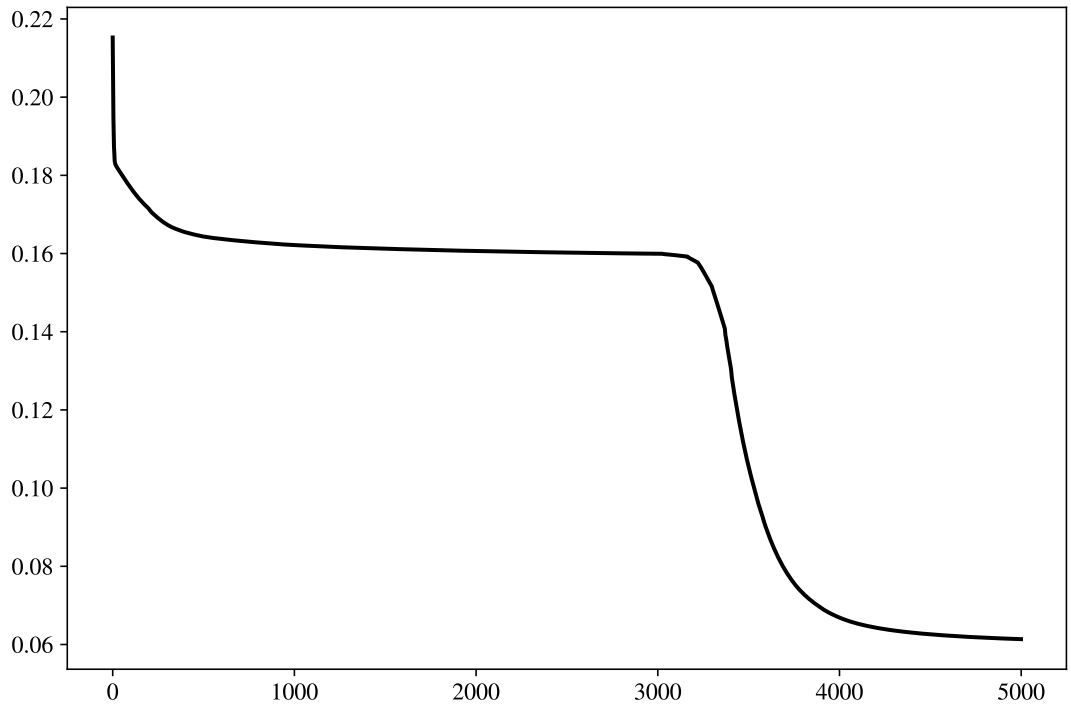
In [31]:

```
def square(x, a, c):
    xc = x - c[None, :] # broadcast to n x 2
    return 1 - (a*xc**2).max(1) # inf norm

def loss(a, c, x, y):
    y_hat = square(x, a, c)
    return mse(y_hat, y)

@jax.jit
def update(a, c, x, y):
    da, dc = jax.grad(loss, argnums=(0,1))(a, c, x, y)
    return a - 0.1 * da, c - 0.1 * dc
```

In [32]:

```
key = jax.random.PRNGKey(32)
key, skey = jax.random.split(key)
c = jax.random.uniform(key, (2,))
a = jnp.ones(2)
l = []
for t in range(5000):
    a, c = update(a, c, X, y)
    l.append(loss(a, c, X, y))
plt.plot(l, '-k')
```

```
[<matplotlib.lines.Line2D at 0x77e7078c1450>]
```

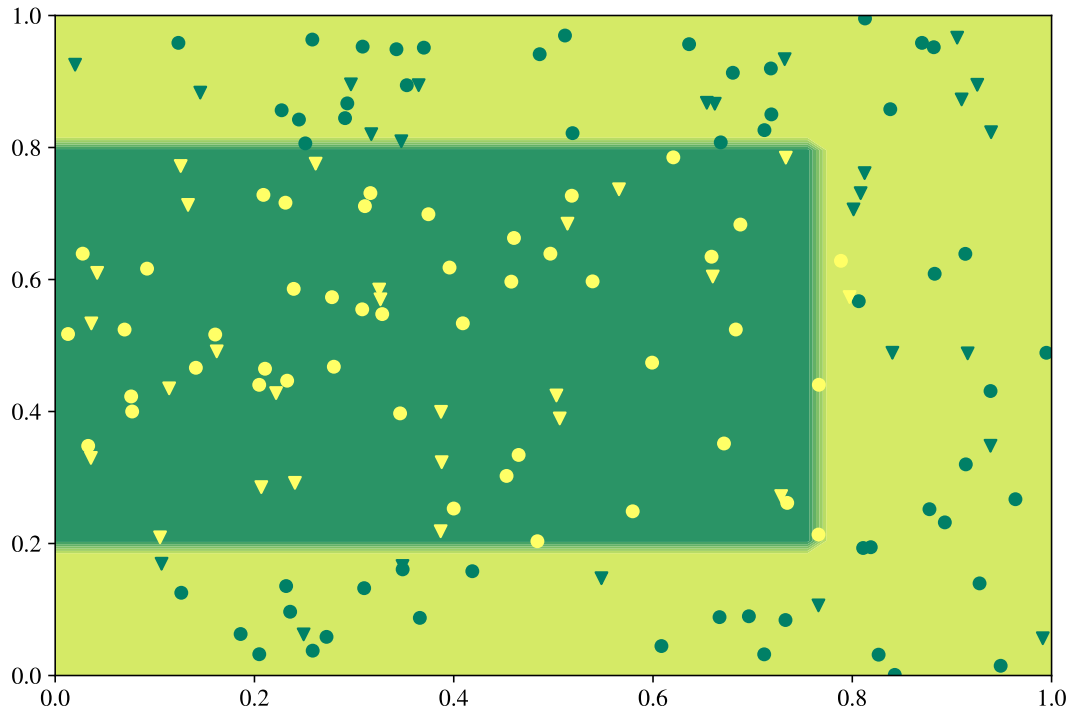


A quick note on that staircase shaped curve: this is often the case with bad problems. In this case, it is because of the infinite norm which has gradient only on the worst offending coordinate, so we can only optimize the axes one at a time meaning that the error due to the others axes will remain until they become the highest. There is nothing wrong with that, but remember that everytime the is a max in a gradient descent, it means that the optimization will be sequential with respect to the coordinates and thus is likely to be very slow.

In [33]:

```
t = 50
tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
levels=jnp.linspace(-1.5, 1.5, 10)
y_pred = jnp.sign(square(xx, a, c)-0.5).reshape(t, t)
plt.contourf(xv, yv, -y_pred, levels=levels)
plt.scatter(X[:,0], X[:,1], c=y), plt.scatter(Xt[:,0], Xt[:,1], marker='v', c=yt)
```

```
(<matplotlib.collections.PathCollection at 0x77e707d07730>,
 <matplotlib.collections.PathCollection at 0x77e707a5b790>)
```



1.8 Exercise

- Perform cross validation on the square classifier to set the number of optimization steps
- Plot train and val losses over time with errorbars

In [34]:

```
def RandomSplitCV(key, X, y, cls_func, max_steps=10000):
    # get a random 80% split of X,y
    # optimize a,c using cls_func for max_steps
    # keep track of training loss and validation loss
    return l_train, l_val
```

In [ ]:

```
# perform 10 random split CV
key = jax.random.PRNGKey(67)
# l_train = jax.random.uniform(key, (10000, 10)); l_val = 0.2*l_train

# plot train and val loss
x = jnp.arange(10000)
l_mean = l_train.mean(axis=1); l_std = l_train.std(1)
plt.plot(x, l_mean, '-b'); plt.fill_between(x, l_mean, l_mean-l_std, l_mean+l_std,
color='b', alpha=0.5)
l_mean = l_val.mean(axis=1); l_std = l_val.std(1)
plt.plot(x, l_mean, '-r'); plt.fill_between(x, l_mean, l_mean-l_std, l_mean+l_std,
color='r', alpha=0.5)
```

1.9 Conclusion on ML and optimization

- Optimizing the 0-1 loss is really hard

- Regression is easy to set (continuous target, continuous f, standard optimization problem)
- Classification: relax to continuous f, find proxy loss (*e.g.*, hinge, logistic)

- Any classification problem can be cast as a regression by arbitrarily mapping $\mathcal{Y}$ to $\mathbb{R}$

- Regression is harder to train than classification (harder to generalize)
- Any regression problem can be transformed into a classification problem by quantizing $\mathcal{Y}$ (but you lose the topology of $\mathcal{Y}$)

1.9.1 ML taxonomy

- Supervised vs Unsupervised
  - Supervised: y is known, effective, difficult to have data
  - Unsupervised: y is unknown, difficult problem, easy to obtain data
  - Semi-supervised: mix of both
  - Reinforcement learning: supervised but only after k decision steps
- Online vs Batch:
  - Batch: train once on all data
  - Online: train on stream of data, then freeze the model
  - Continuous learning: train on steam, never freeze the model
- Passive vs Active:
  - Passive: all training data are i.i.d.
  - Active: training data obtained via a selection process to be more efficient
- Shallow vs Deep
  - Shallow learning: handcrafted/engineered features + ML based decision
  - Deep learning: train both feature extractor and decision

1.10 Lecture 1's take home

- ERM principle
- *train/val/test* mantra, cross-validation
- ML is optimizing parameters to fit data
- Taxonomy: supervised/unsupervised, classification/regression
- Our first learning algorithm: k-NN!

Chapter 2

Linear models

2.1 Linear Regression

2.1.1 Scalar input, scalar output

- Input space: $x \in \mathbb{R}$
- Output space: $y \in \mathbb{R}$
- Linear model: $f(x) = ax$

$$\min_a \mathbb{E}_x[(y - ax)^2]$$

Training set $\mathcal{A} = \{(x_i, y_i)\}_{i \leq n}$, minimize the empirical risk

$$\min_a \frac{1}{n} \sum_i (y_i - ax_i)^2$$

This is the simplest case possible. You have a set of pairs of scalar values and want to know the best coefficient to transform one into the other. Now that it is framed as *"minimizing the empirical risk"*, it called be called AI and sold at a higher price. You're welcome. Closed form solution: - vectorize: $\mathbf{x} = [x_i]$, $\mathbf{y} = [y_i]$

$$\min_a \frac{1}{n} \|\mathbf{y} - a\mathbf{x}\|^2$$

- Stationary condition

$$\frac{\partial}{\partial a} \frac{1}{n} \|\mathbf{y} - a\mathbf{x}\|^2 = 0 = 2a\|\mathbf{x}\|^2 - 2\langle \mathbf{y}, \mathbf{x} \rangle$$

$$a = \frac{\mathbf{y}^\top \mathbf{x}}{\|\mathbf{x}\|^2}$$

2.1.2 Linear regression - Vector input, scalar output

- Input space: $\mathbf{x} \in \mathbb{R}^d$
- Output space: $y \in \mathbb{R}$
- Linear model: $f(x) = \mathbf{a}^\top \mathbf{x}$

$$\min_{\mathbf{a}} \mathbb{E}_x[(y - \mathbf{a}^\top \mathbf{x})^2]$$

Training set $\mathcal{A} = \{(\mathbf{x}_i, y_i)\}_{i \leq n}$, minimize the empirical risk

$$\min_a \frac{1}{n} \sum_i (y_i - \mathbf{a}^\top \mathbf{x}_i)^2$$

The next version is slightly more practical, as we now want to predict a scalar value from a vector. Closed form solution - Matrix form: $\mathbf{X} = [\mathbf{x}_i]$, $\mathbf{y} = [y_i]$

$$\min_{\mathbf{a}} \frac{1}{n} \|\mathbf{y} - \mathbf{X}^\top \mathbf{a}\|^2$$

- Stationary condition

$$\frac{\partial}{\partial \mathbf{a}} \frac{1}{n} \|\mathbf{y} - \mathbf{X}^\top \mathbf{a}\|^2 = 0 = -2\mathbf{X}^\top \mathbf{y} + 2\mathbf{X}\mathbf{X}^\top \mathbf{a}$$

$$\mathbf{a} = (\mathbf{X}\mathbf{X}^\top)^{-1} \mathbf{X}^\top \mathbf{y}$$

Pseudo-inverse Notice that it is required that $\mathbf{X}\mathbf{X}^\top$ be invertible. For this, it is required that $\mathbf{X}$ has rank d, and thus if we have less training samples than dimension, we cannot use the pseu-do inverse technique. At least, not in this fashion. Instead, we have to resort to the dual approach using the Gram matrix.

2.1.3 Vector input, bis

- SVD: $\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^\top$

$$\mathbf{y} = \mathbf{V}\mathbf{S}\mathbf{U}^\top \mathbf{a}$$

$$\mathbf{a} = \mathbf{U}\mathbf{S}^{-1}\mathbf{V}^\top \mathbf{y}$$

Easy solution by projecting into the eigen space of $\mathbf{X}$, $\mathbf{a}$ is in the eigenspace of $\mathbf{X}$

2.1.4 Karhunen-Loève theorem

Let $\mathbf{x}$ be a stochastic process with covariance matrix $\sum_{\mathbf{x}}$ then

$$\mathbf{x}_i = \sum_k^p z_{k,i} \mathbf{e}_k$$

with $\mathbf{e_k}$ the eigenvectors of $\sum_{\mathbf{x}}$.

- Samples $\mathbf{x}_i$ exist in the space spaned by the eigenvectors of the covariance matrix (hence PCA)
- if $span(\mathbf{x}) < d$, some dimensions are useless (noisy)
- Strong influence on the pseudo-inverse solution ($\mathbf{S}^{-1}$) $\Rightarrow$ remove directions with small eigenvalues

We mention here the Karhunen-Loève theorem to prepare for future results known a representer theorems. The key idea is that since samples exist in a space that is defined by the eigenvectors of their covariance matrix, it is intuitive that only these eigenvector play a role in finding the solution to our regression problem. In essence, why should a direction that never happens in the data matter for the problem? The answer is it shouldn't and this is why the eigenspace of the data appears in the solution.

In [2]:

```
key = jax.random.PRNGKey(0)
key, skey = jax.random.split(key)
x = jax.random.uniform(skey, (50, 1))
X = jnp.concatenate((x, -5*x), axis=1)
a = jnp.array([2, 0])
y = jnp.matmul(X, a)

U, S, V = jnp.linalg.svd(X.T, full_matrices=False)
print('eigenvalues: {}'.format(S))
Vty = jnp.matmul(V, y)
SinvVty = jnp.matmul(jnp.diag(1./S), Vty)
a_hat = jnp.matmul(U, SinvVty)
print('a_hat: {} a: {}'.format(a_hat, a))
```

```
eigenvalues: [2.0413006e+01 3.0459864e-07]
a_hat: [-2.8013153  -0.96026266] a: [2 0]
```

2.1.5 Linear regression, bias case

Adding a constant to the model is equivalent to the vector case

$$\min_{\mathbf{a},b} \frac{1}{n} \|\mathbf{y} - \mathbf{X}^\top \mathbf{a} - \mathbf{1}^\top b\|^2$$

- concatenate b to $\mathbf{a}$ and 1 to $\mathbf{X}$

$$\min_{\mathbf{a},b} \frac{1}{n} \|\mathbf{y} - [\mathbf{X}; \mathbf{1}]^\top [\mathbf{a}; b]\|^2$$

2.1.6 Linear Regression, Vector input, vector output

- Input space: $\mathbf{x} \in \mathbb{R}^d$
- Output space: $\mathbf{y} \in \mathbb{R}^p$
- Linear model: $f(x) = \mathbf{A}^\top \mathbf{x}$

$$\min_{\mathbf{a}} \mathbb{E}_x[\|\mathbf{y} - \mathbf{A}^\top \mathbf{x}\|^2]$$

Training set $\mathcal{A} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i \leq n}$, minimize the empirical risk

$$\min_a \frac{1}{n} \sum_i \|\mathbf{y}_i - \mathbf{A}^\top \mathbf{x}_i\|^2$$

Equivalent to p scalar output cases stacked together This is ultimately the real practical application of linear regression: predict several values altogether. However, it is rather unsatisfying since it turns out to be equivalent to performing independent scalar regression in parallel. This is because we never put any constraints on the columns of $\mathbf{A}$ to be correlated.

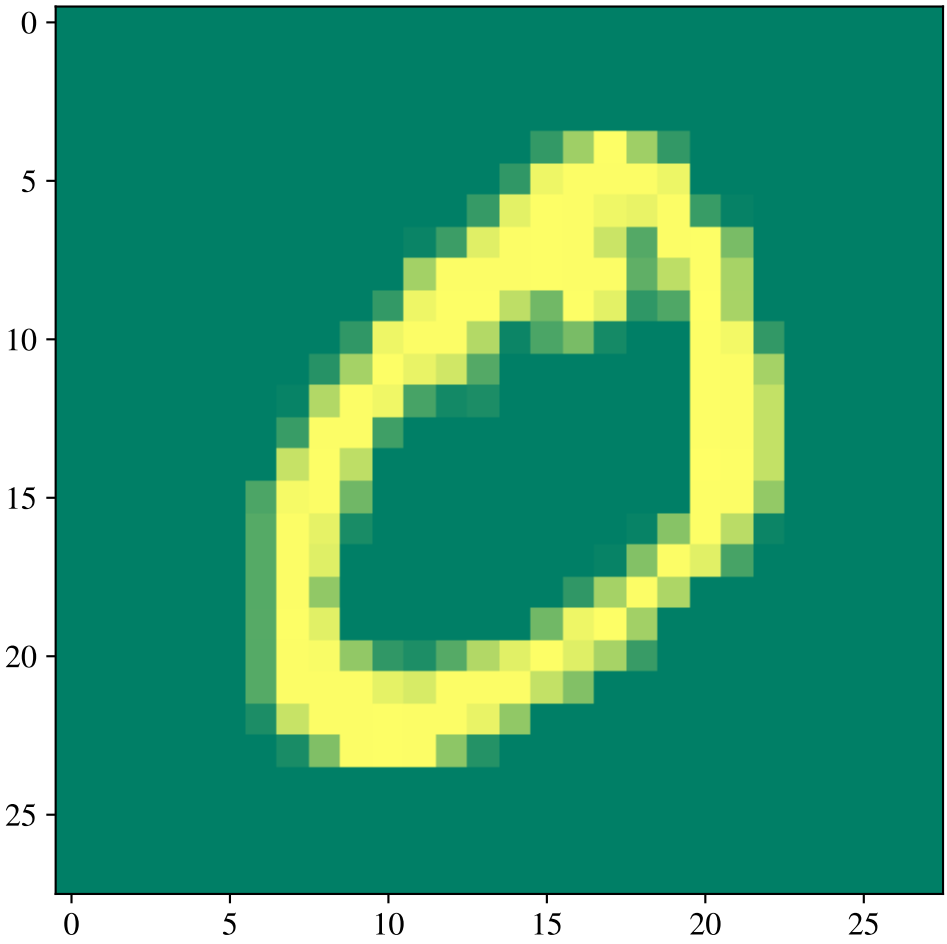
2.1.7 Let's try with MNIST

Regress 0 and 1

In [3]:

```
data = np.load('mnist.npz')
X = data['X_train_bin']
y = data['y_train_bin']
plt.imshow(X[0,:].reshape(28,28))
print(y[0])
```

```
0
```



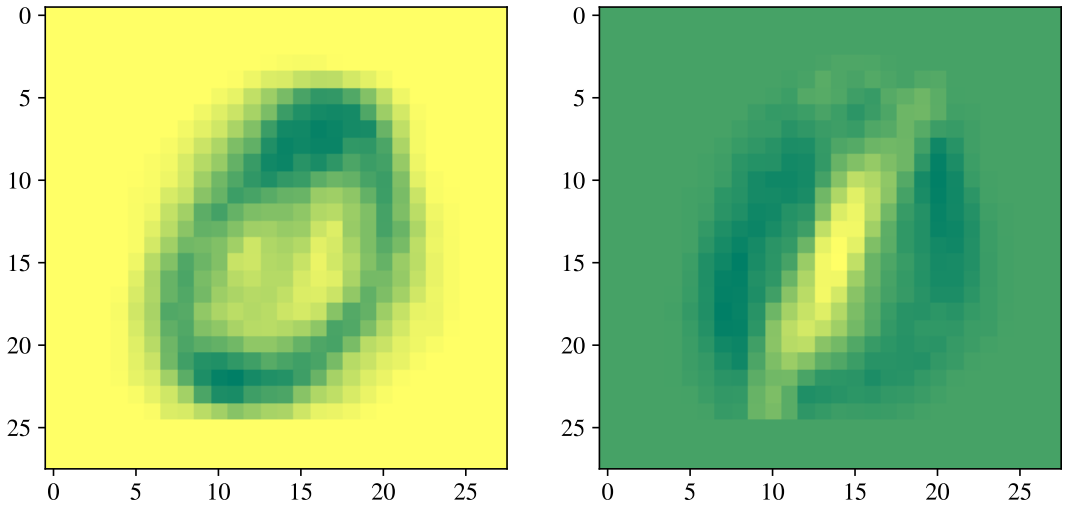
In [4]:

```
U, S, V = jnp.linalg.svd(X.T, full_matrices=False)
print('eigenvalues: {}'.format(S))
plt.subplot(1,2,1)
plt.imshow(U[:,0].reshape((28,28)))
plt.subplot(1,2,2)
plt.imshow(U[:,1].reshape((28,28)))
```

```
eigenvalues: [36.45615   17.209862  12.030047  11.947479   9.36883    7.8752723
  6.7997932  6.3316774  5.729243   5.4004116  5.1866603  4.988159
```

```
  4.8420706  4.1217637  3.9173453  3.5896347  3.2801106  3.1127822
  3.0160408  2.7964358  2.6677098  2.543037   2.2888196  2.1948047
  2.14488    1.8337901  1.0252038]
```

```
<matplotlib.image.AxesImage at 0x71008d42c040>
```

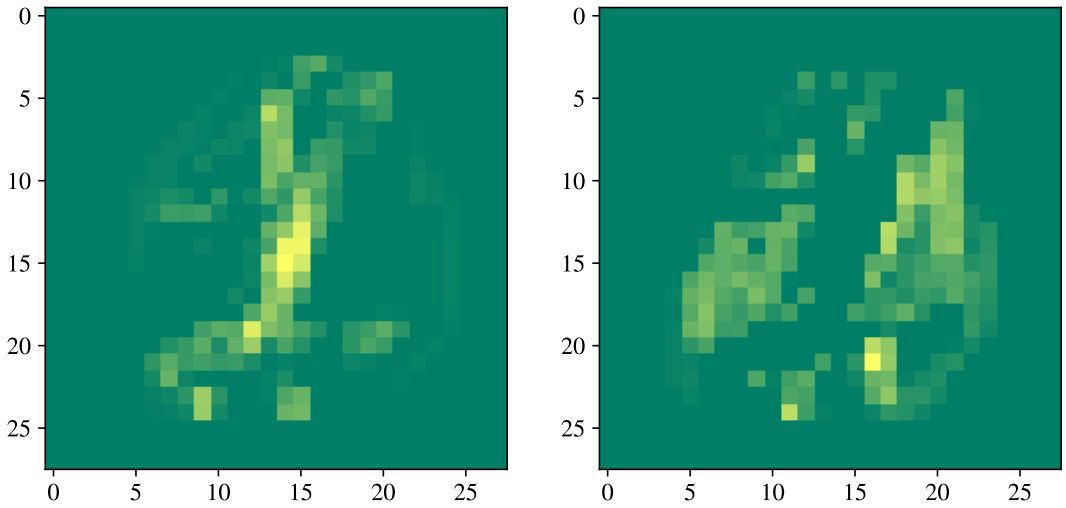


If we interpret our predictor as an image, positive pixels tend to push the prediction towards class 1, whereas negative pixels tend to push the prediction toward class 0. Let us display the images of pixels of the same sign.

In [5]:

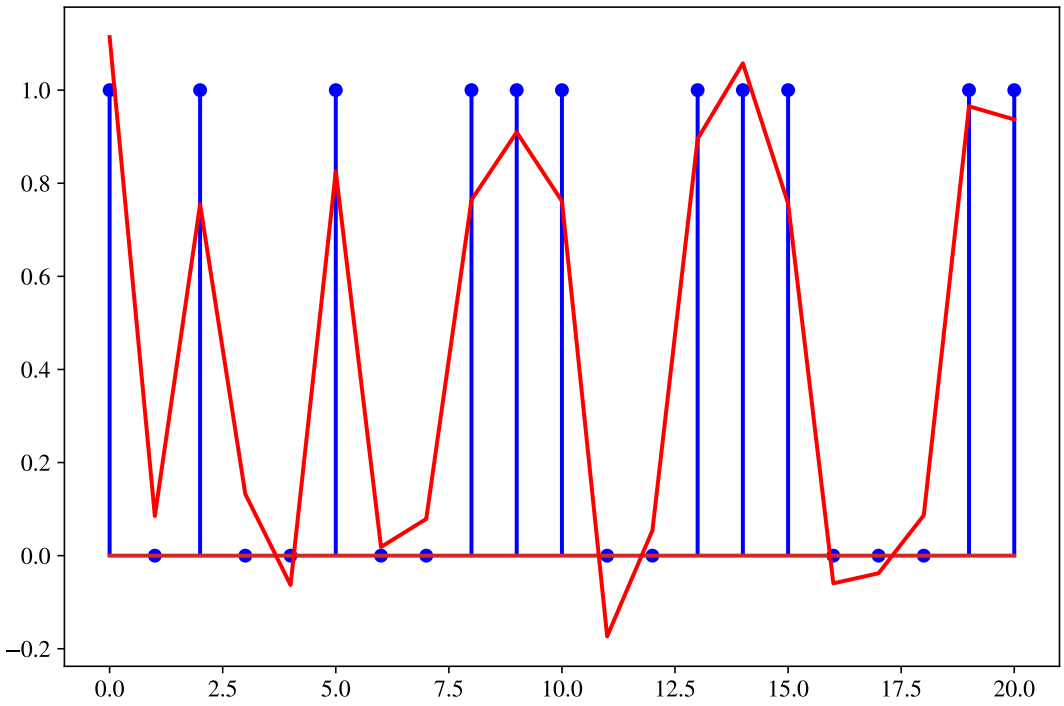
```
Vty = jnp.matmul(V, y)
SinvVty = jnp.matmul(jnp.diag(1./S), Vty)
a_hat = jnp.matmul(U, SinvVty)

plt.subplot(1,2,1)
plt.imshow(jnp.maximum(a_hat, 0).reshape((28, 28)))
plt.subplot(1,2,2)
plt.imshow(jnp.maximum(-a_hat, 0).reshape((28, 28)))
```

```
<matplotlib.image.AxesImage at 0x71008d3b8160>
```



In [6]:
```
X_val = data['X_val_bin']
y_val = data['y_val_bin']
y_hat = jnp.matmul(X_val, a_hat)
plt.stem(range(len(y_val)), y_val, '-b')
plt.plot(range(len(y_val)), y_hat, '-r')
```

```
[<matplotlib.lines.Line2D at 0x71008c7ea170>]
```



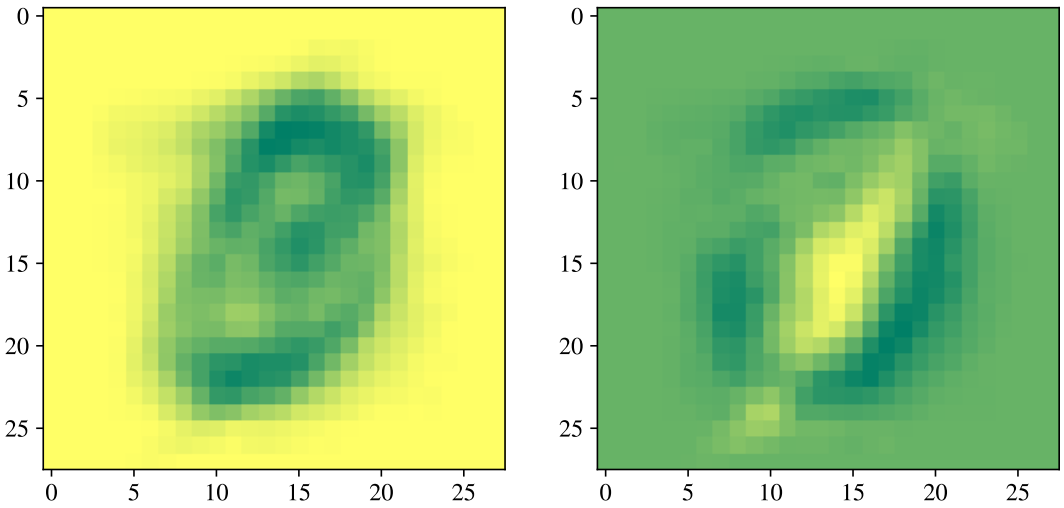
This is actually not bad! Remember that this is on a validation set, and fixing a threshold at 0.5 we can even reach 100% accuracy on this very small validation set.

2.1.8 MNIST, regress 0-9

In [7]:
```
X = data['X_train']
y = data['y_train']

U, S, V = jnp.linalg.svd(X.T, full_matrices=False)
print('eigenvalues: {}'.format(S[0:10]))
plt.subplot(1,2,1)
plt.imshow(U[:,0].reshape((28,28)))
plt.subplot(1,2,2)
plt.imshow(U[:,1].reshape((28,28)))
```

```
eigenvalues: [61.84758  22.601597 19.816174 18.98699  17.21556  15.531815 14.318835
 13.584824 12.348789 11.739741]
```

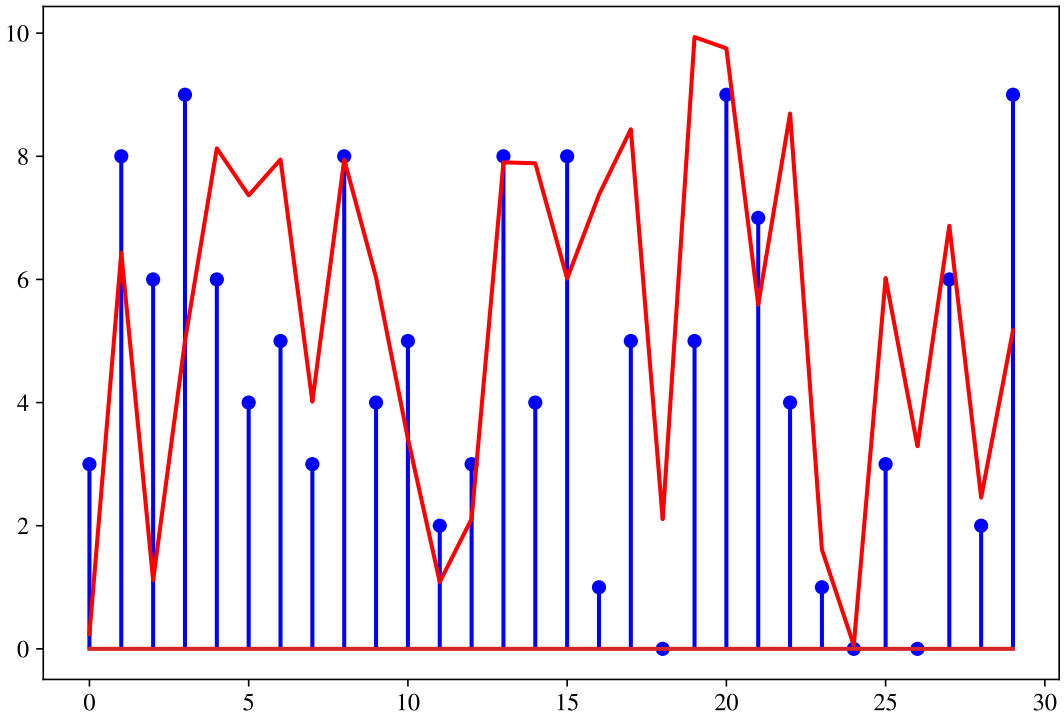
```
<matplotlib.image.AxesImage at 0x71008c432140>
```



In [8]:

```
Vty = jnp.matmul(V, y)
SinvVty = jnp.matmul(jnp.diag(1./S), Vty)
a_hat = jnp.matmul(U, SinvVty)

X_val = data['X_val'][0:30,...]
y_val = data['y_val'][0:30]
y_hat = jnp.matmul(X_val, a_hat)
plt.stem(range(len(y_val)), y_val, '-b')
plt.plot(range(len(y_val)), y_hat, '-r')
```

```
[<matplotlib.lines.Line2D at 0x71008c4e6950>]
```



Output space not adapted (artificial topology) Here we are doing much worse. Because we

are predicting the classes as a real value between 0 and 9, we have a success if and only if we are predicting within 0.5 of the target value. The annoying aspect with theis approach is that to us, as an optical character recognition task, a failure is a failure, no matter what was the digit that was predicted. That the function predicts a 3 or an 8 instead of a 2 is exactly the same because it is the wrong character, but the regression loss considers those errors to be different. It values predicting a 3 for a 2 better than predicting an 8 because we introduced an arbitrary sense of distance between the classes.

This could be aleviated by making sure all classes are equidistant with respect to the norm that is used for the regression.

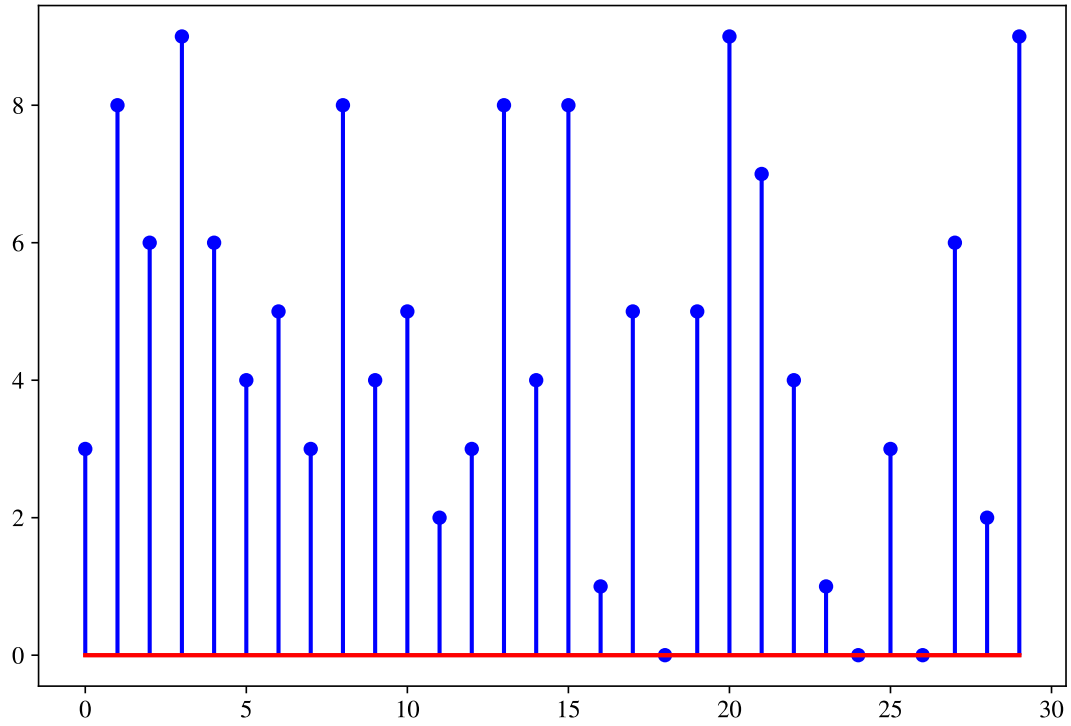
2.1.9 MNIST regress 0-9 as one-hot

- Exercise: train a linear regression for each class (one versus all)

In [9]:

```
y = jax.nn.one_hot(y, 10)

a = []
for k in range(10):
    #
    #
    a_k = jnp.zeros(784)
    a.append(a_k)
a = jnp.array(a)

y_hat = jnp.argmax(jnp.matmul(X_val, a.T), axis=1)
plt.stem(range(len(y_val)), y_val, '-b')
plt.plot(range(len(y_val)), y_hat, '-r')
```

```
[<matplotlib.lines.Line2D at 0x71008d5725f0>]
```



2.2 Non-linear case

2.2.1 Polynomial regression

What if the relation between x and y is not linear?

- Map $\phi : x \mapsto [x, x^2, x^3, \ldots, x^p]$

$$\min_{\mathbf{a}} \mathbb{E}_x[(y - \phi(x)^\top \mathbf{a})^2]$$

In [10]:

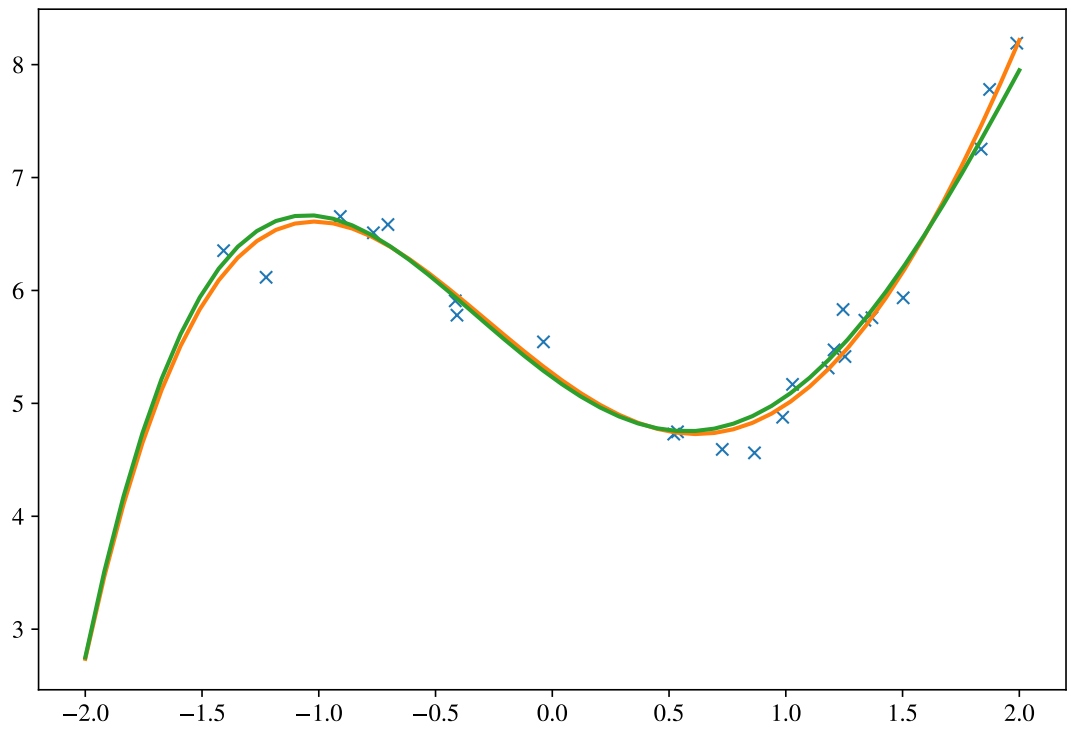
```
a = [-0.2, 0.7, 0.83, -1.5, 5.23]
p = np.poly1d(a)
x = np.random.rand(24)*4-2
y = p(x) + 0.2*np.random.randn(24)
```

In [11]:

```
X = jnp.stack([jnp.ones((len(x))), x, x**2, x**3, x**4], axis=1)
U, S, V = jnp.linalg.svd(X.T, full_matrices=False)
Vty = jnp.matmul(V, y)
SinvVty = jnp.matmul(jnp.diag(1./S), Vty)
a_hat = jnp.matmul(U, SinvVty)
print(a_hat)
pp = np.poly1d(a_hat[::-1])

t = np.linspace(-2, 2, 50)
plt.plot(x, y, 'x')
plt.plot(t, pp(t))
plt.plot(t, p(t))
```

```
[ 5.266477   -1.538329    0.6909166   0.7270644  -0.15955621]
```

```
[<matplotlib.lines.Line2D at 0x7100685f9570>]
```



2.2.2 Periodic signals

Map $\phi : x \mapsto [\sin(f_0 x), \sin(2f_0 x), \ldots, \sin(pf_0 x)]$

In [12]:

```
a = np.array([0.7, 0.83, -1.5])
x = np.random.rand(48)*16-8
X = jnp.array([ jnp.sin(x), jnp.sin(2*x), jnp.sin(3*x)])
y = jnp.matmul(a, X) + 0.3*np.random.randn(48)
```

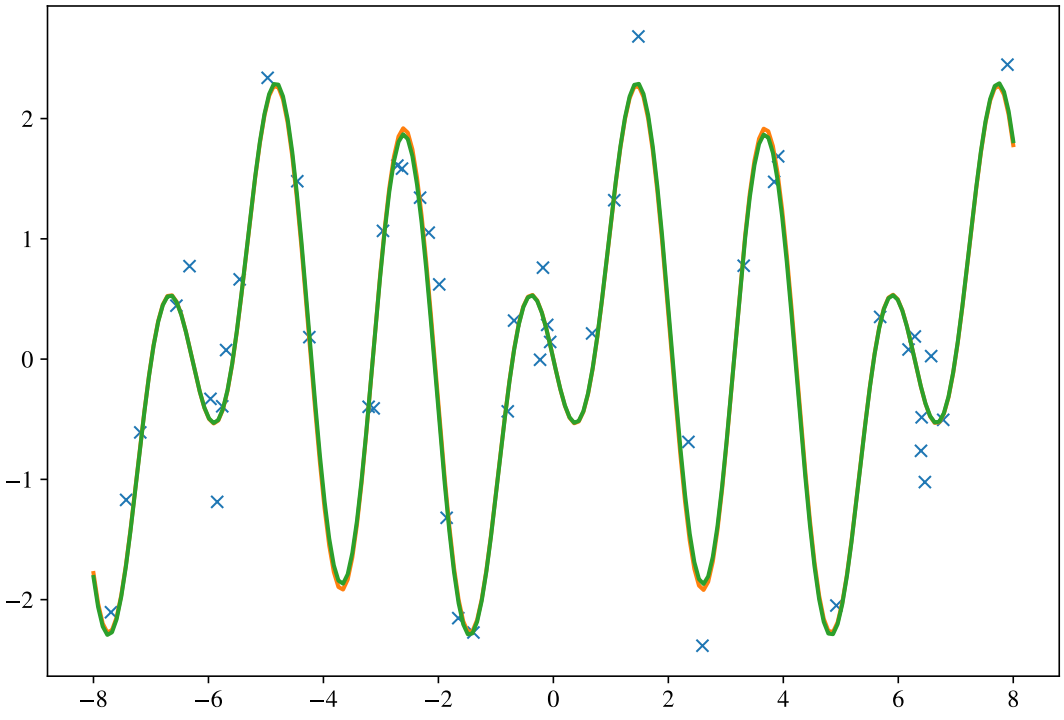
In [13]:

```
X = jnp.array([jnp.sin(x), jnp.sin(2*x), jnp.sin(3*x)])
ap = jnp.matmul(jnp.matmul(jnp.linalg.inv(jnp.matmul(X, X.transpose())), X), y)
Yp = jnp.matmul(ap, X)
```

In [14]:

```
t = np.linspace(-8, 8, 200)
T = np.array([np.sin(t), np.sin(2*t), np.sin(3*t)])
plt.plot(x, y, 'x')
plt.plot(t, np.matmul(ap, T))
plt.plot(t, np.matmul(a, T))
```

```
[<matplotlib.lines.Line2D at 0x710069721420>]
```



2.2.3 Overcomplete models

What if $p < \hat{p}$ (model has greater capacity than data) Let use fit that curve with increasingly higher polynomial degrees.

In [15]:

```
def sin_approx(x, y, p):
    Xp = jnp.sin(jnp.matmul(x.reshape(-1, 1),
1+jnp.arange(p).reshape(1,p))).transpose()
    ap = jnp.matmul(jnp.matmul(jnp.linalg.inv(jnp.matmul(Xp, Xp.transpose())), Xp), y)
    Yp = jnp.matmul(ap, Xp)
    return ap, Xp, Yp
```

In [16]:

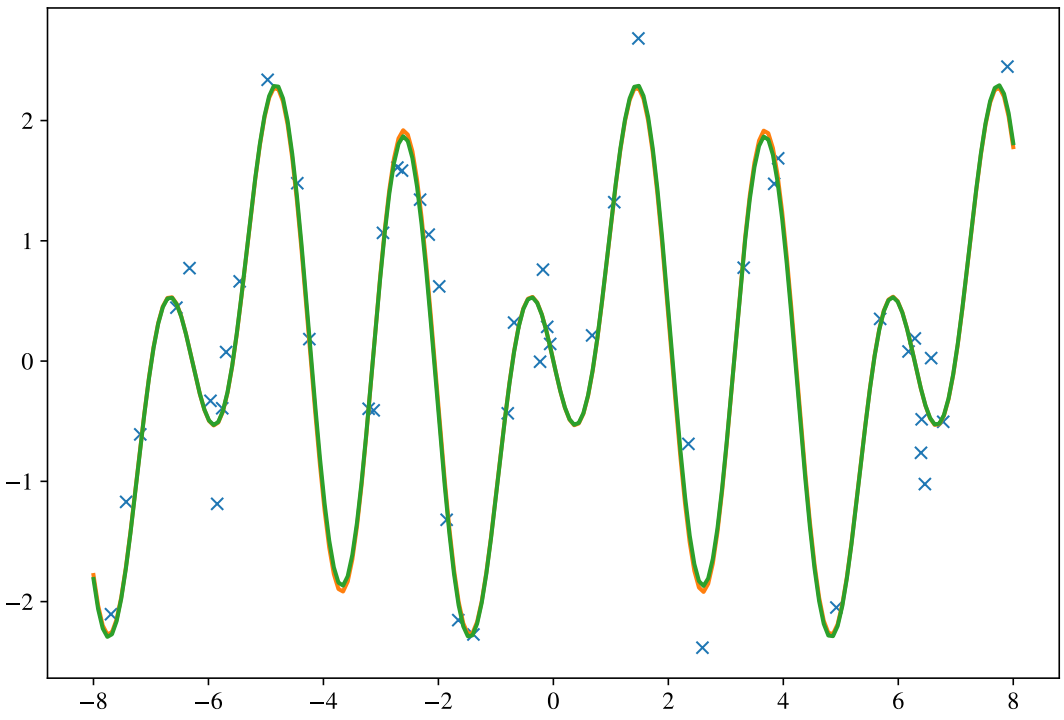
```
p = 3
ap, Xp, Yp = sin_approx(x, y, p)
print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [ 0.66503567  0.8555399  -1.5098379 ]
```

In [17]:

```
t = jnp.linspace(-8, 8, 200)
T = jnp.sin(np.matmul(t.reshape(-1, 1), 1+jnp.arange(p).reshape(1,p))).transpose()
plt.plot(x, y, 'x')
plt.plot(t, jnp.matmul(ap, T[:len(ap), :]))
plt.plot(t, jnp.matmul(a, T[:len(a), :]))
print('MSE: {}'.format(((y - Yp)**2).mean()))
```

```
MSE: 0.13037486374378204
```



In [18]:

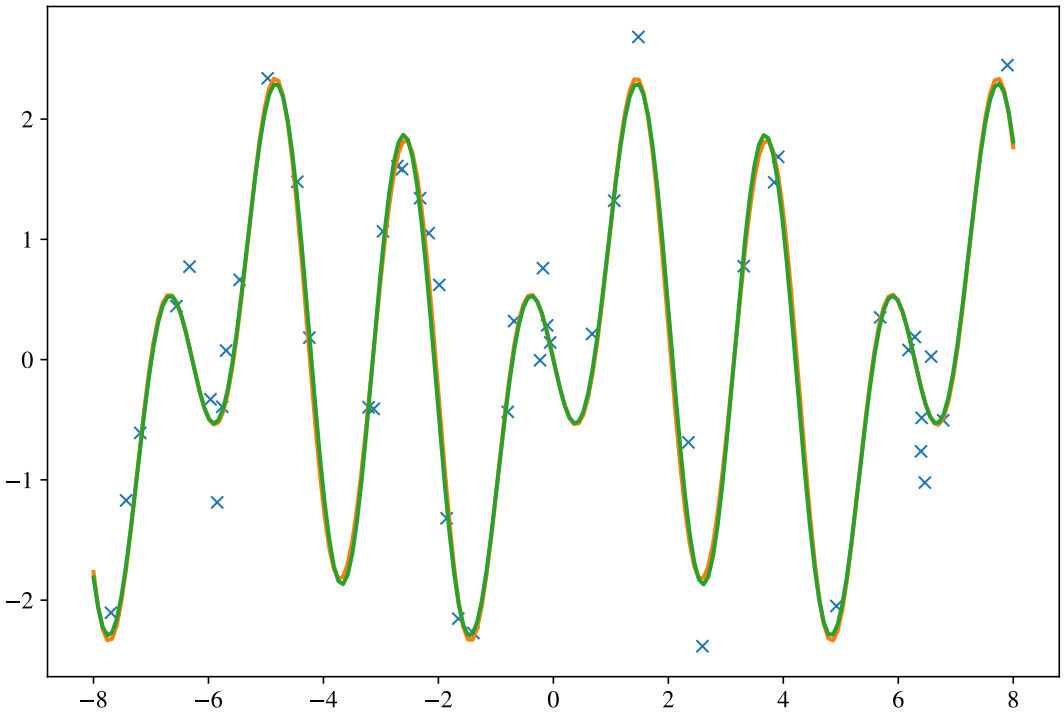
```
p = 5
ap, Xp, Yp = sin_approx(x, y, p)
print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [ 0.66392684  0.8552013  -1.4939932  -0.07257754  0.05675733]
```

In [19]:

```
t = jnp.linspace(-8, 8, 200)
T = jnp.sin(np.matmul(t.reshape(-1, 1), 1+jnp.arange(p).reshape(1,p))).transpose()
plt.plot(x, y, 'x')
plt.plot(t, jnp.matmul(ap, T[:len(ap), :]))
plt.plot(t, jnp.matmul(a, T[:len(a), :]))
print('MSE: {}'.format(((y - Yp)**2).mean()))
```

```
MSE: 0.1274968534708023
```



In [20]:

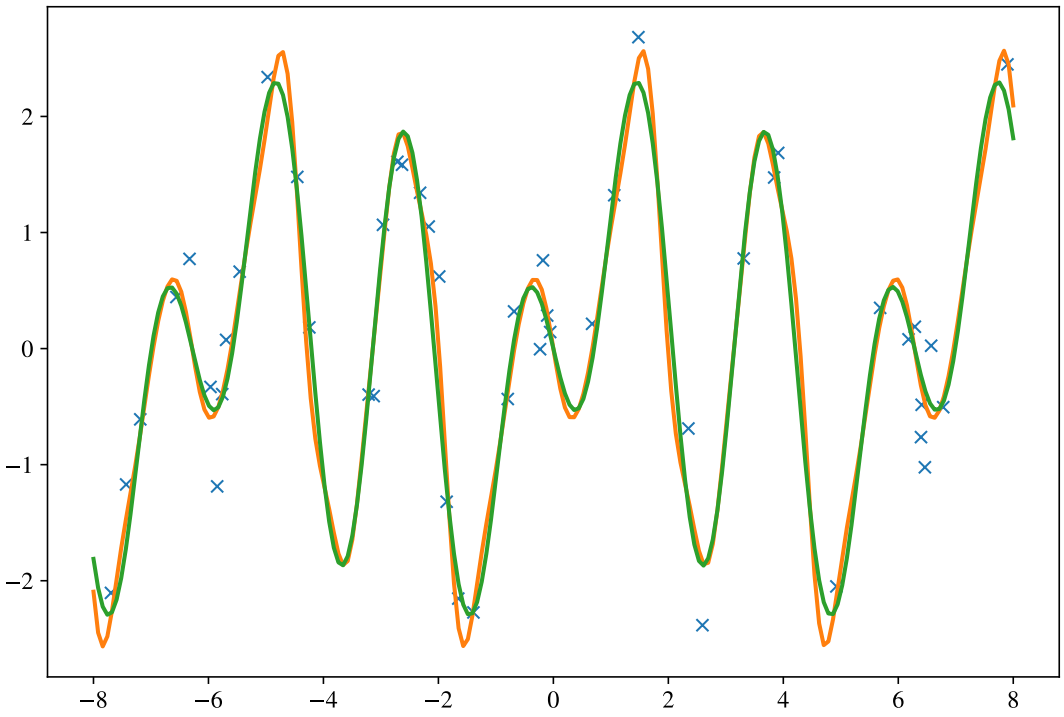
```
p = 10
ap, Xp, Yp = sin_approx(x, y, p)
print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [ 0.64930737  0.86065483 -1.5159405  -0.0419542   0.10985056
-0.04247902
 -0.1989938   0.04065548  0.08720873 -0.07313146]
```

In [21]:

```
t = jnp.linspace(-8, 8, 200)
T = jnp.sin(np.matmul(t.reshape(-1, 1), 1+jnp.arange(p).reshape(1,p))).transpose()
plt.plot(x, y, 'x')
plt.plot(t, jnp.matmul(ap, T[:len(ap), :]))
plt.plot(t, jnp.matmul(a, T[:len(a), :]))
print('MSE: {}'.format(((y - Yp)**2).mean()))
```

```
MSE: 0.1047743558883667
```



In [22]:

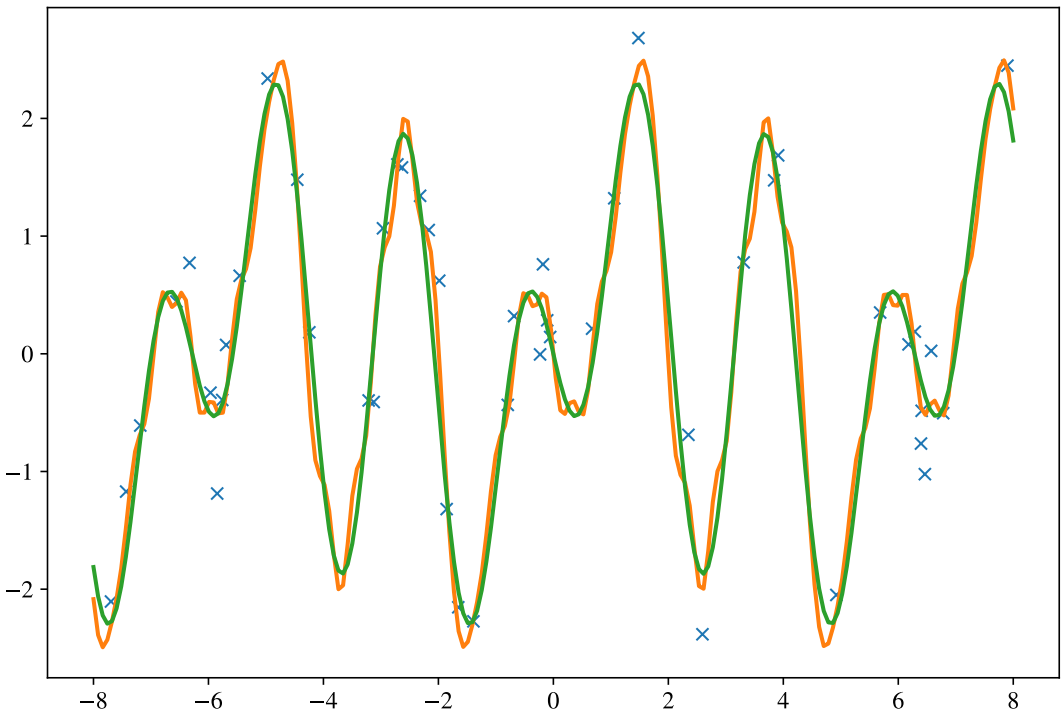
```
p = 15
ap, Xp, Yp = sin_approx(x, y, p)
print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [ 0.64816654  0.85237145 -1.4848707  -0.06554112  0.17367777
-0.08139829
 -0.15539262  0.0064044   0.08781601 -0.0583418   0.01320645  0.03894603
 -0.13762946  0.01072791 -0.09005098]
```

In [23]:

```
t = jnp.linspace(-8, 8, 200)
T = jnp.sin(np.matmul(t.reshape(-1, 1), 1+jnp.arange(p).reshape(1,p))).transpose()
plt.plot(x, y, 'x')
plt.plot(t, jnp.matmul(ap, T[:len(ap), :]))
plt.plot(t, jnp.matmul(a, T[:len(a), :]))
print('MSE: {}'.format(((y - Yp)**2).mean()))
```

```
MSE: 0.09392114728689194
```



In [24]:

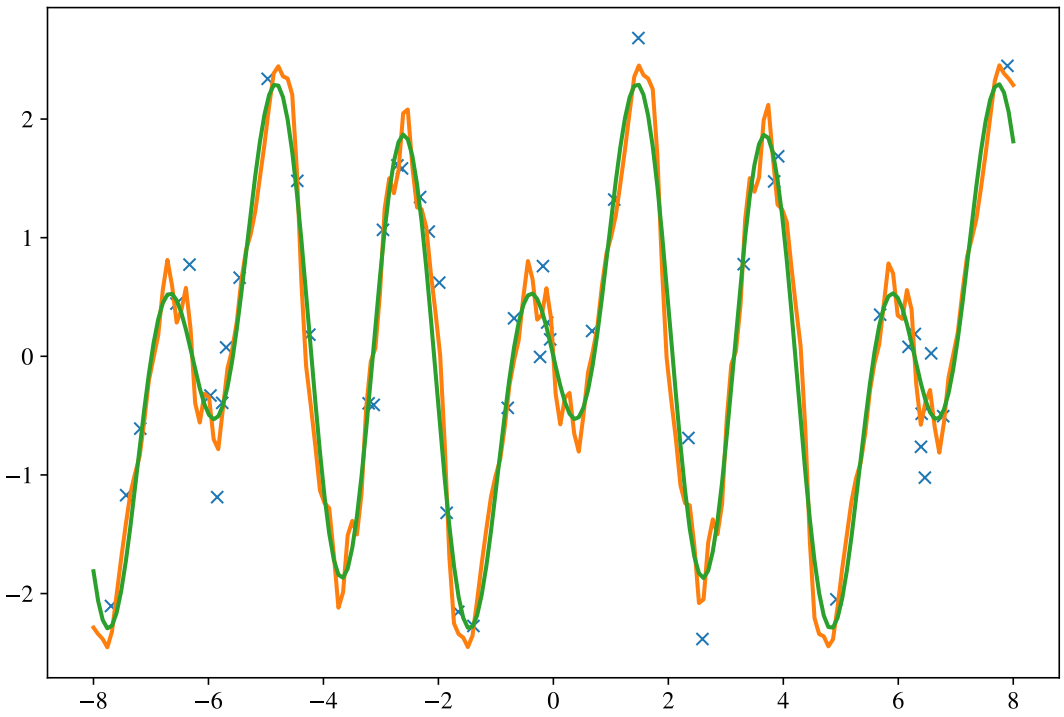
```
p = 20
ap, Xp, Yp = sin_approx(x, y, p)
print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [ 0.63273454  0.862865   -1.5327747  -0.04082306  0.12400526
-0.07723609
 -0.18338399  0.05933774  0.09869707 -0.05381919  0.01258338  0.04874885
 -0.09009814 -0.00779995  0.00236045 -0.04848589 -0.05025677 -0.11085568
  0.04835366 -0.12392354]
```

In [25]:

```
t = jnp.linspace(-8, 8, 200)
T = jnp.sin(np.matmul(t.reshape(-1, 1), 1+jnp.arange(p).reshape(1,p))).transpose()
plt.plot(x, y, 'x')
plt.plot(t, jnp.matmul(ap, T[:len(ap), :]))
plt.plot(t, jnp.matmul(a, T[:len(a), :]))
print('MSE: {}'.format(((y - Yp)**2).mean()))
```

```
MSE: 0.08044012635946274
```



In [26]:

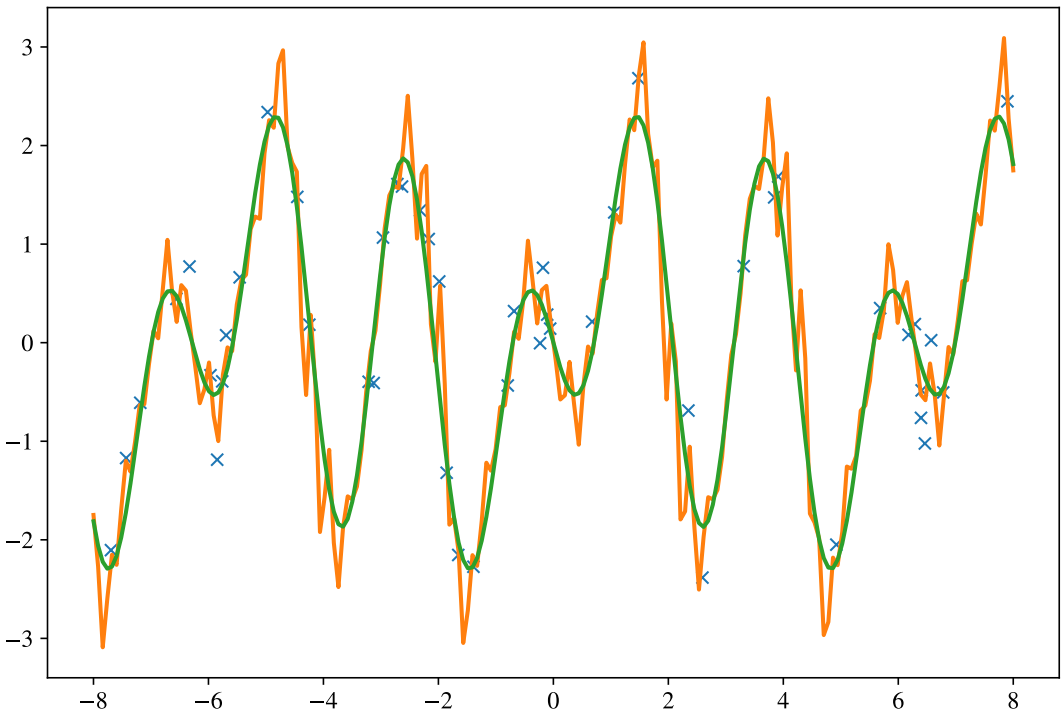
```
p = 25
ap, Xp, Yp = sin_approx(x, y, p)
print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [ 0.5857791   0.9295168  -1.5959327  -0.06027614  0.15495446
-0.07315889
 -0.18192858 -0.00906502  0.12587665  0.02122447 -0.05915882  0.0195908
  0.00481635 -0.02463253 -0.0524892  -0.01131299 -0.07204011 -0.1264672
  0.09676984 -0.20841293  0.02010447  0.1596179  -0.20179838  0.03811657
  0.21566837]
```

In [27]:

```
t = jnp.linspace(-8, 8, 200)
T = jnp.sin(np.matmul(t.reshape(-1, 1), 1+jnp.arange(p).reshape(1,p))).transpose()
plt.plot(x, y, 'x')
plt.plot(t, jnp.matmul(ap, T[:len(ap), :]))
plt.plot(t, jnp.matmul(a, T[:len(a), :]))
print('MSE: {}'.format(((y - Yp)**2).mean()))
```

```
MSE: 0.054160039871931076
```



In [28]:

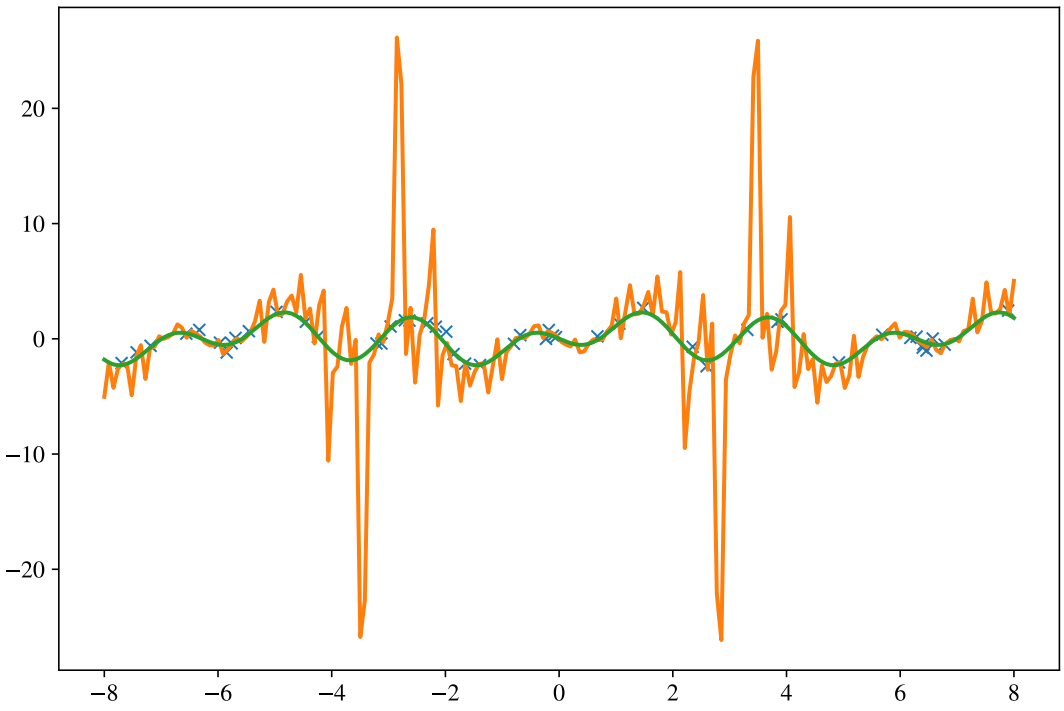
```
p = 35
ap, Xp, Yp = sin_approx(x, y, p)
print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [ 0.60171056  2.1561666  -3.7234256   1.9323449  -1.881022
1.7465707
 -1.9733595   2.061804   -1.430026    0.31073868  0.800243   -1.7518697
  2.4190798  -2.4179287   1.7261914  -0.81368876  0.62834394 -1.0664698
  0.6718048  -0.6353989  -0.20793903  1.5484711  -1.6906972   0.984274
 -0.906407    0.850715   -0.3213622   0.83473617 -0.9671983   0.1878281
  0.6671101  -1.3488506   1.6334653  -0.616805   -0.2670184 ]
```

In [29]:

```
t = jnp.linspace(-8, 8, 200)
T = jnp.sin(np.matmul(t.reshape(-1, 1), 1+jnp.arange(p).reshape(1,p))).transpose()
plt.plot(x, y, 'x')
plt.plot(t, jnp.matmul(ap, T[:len(ap), :]))
plt.plot(t, jnp.matmul(a, T[:len(a), :]))
print('MSE: {}'.format(((y - Yp)**2).mean()))
```

```
MSE: 0.02646658569574356
```



At some point, we are just fitting the noise. Remember that we can pass a polynomial exactly through all the training examples as soon as its degree is higher than the size of the training set. Because we are totally blind between the training examples, our function can take any value there and it will not be penalized by the loss. This is obvisouly a problem that can only be mitigated by making sure the function is also "observed" between the training examples. There are two ways of doing so.

2.2.4 Train/validation

The first way is by performing validation. By having a set of example that is reserved for model selection, we measure the function outside of the training set and penalize functions that do not behave properly in that aspect. However, a validation set is finite and it still leaves blindspots.

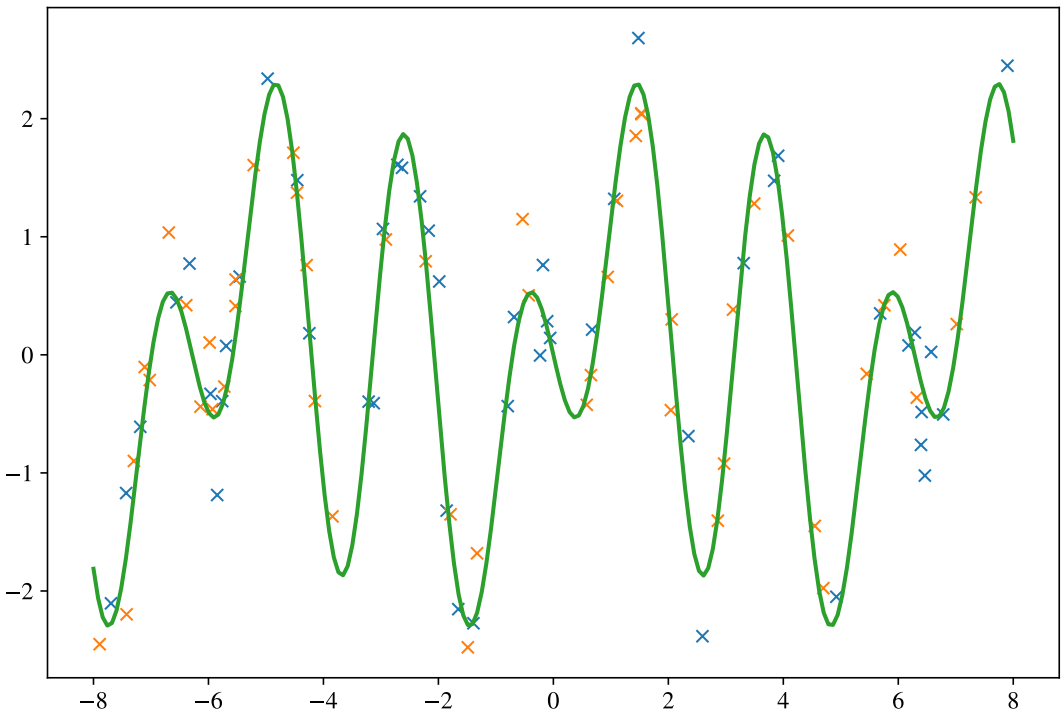
In [30]:

```
x_val = np.random.rand(48)*16-8
X_val = jnp.array([ jnp.sin(x_val), jnp.sin(2*x_val), jnp.sin(3*x_val)])
y_val = jnp.matmul(a, X_val) + 0.3*np.random.randn(48)
```

In [31]:

```
plt.plot(x, y, 'x')
plt.plot(x_val, y_val, 'x')
plt.plot(t, jnp.matmul(a, T[:len(a), :]))
```

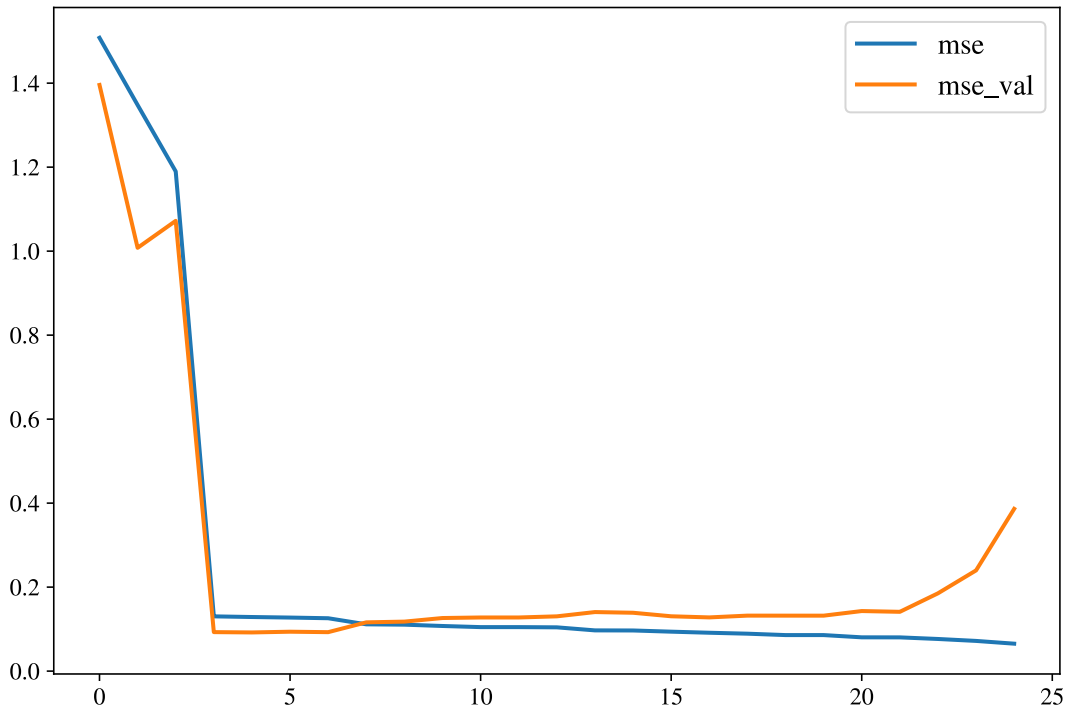
```
[<matplotlib.lines.Line2D at 0x710068e3fc40>]
```



In [32]:

```
mse = []
mse_val = []
for p in range(25):
    ap, Xp, Yp = sin_approx(x, y, p)
    _, Xp_val, _ = sin_approx(x_val, y_val, p)
    Yp_val = jnp.matmul(ap, Xp_val)
    mse.append(((y - Yp)**2).mean()), mse_val.append(((y_val - Yp_val)**2).mean())
plt.plot(mse, label='mse'); plt.plot(mse_val, label='mse_val')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x710055d96e30>
```



2.3 Regularization

The better way of handling blindspots in the loss is to somehow "observe" implicitly the function everywhere. This can be done by adding a structural cost that penalizes function that take values outside of the training samples that are radically different from the ones on the training examples. One easy way of doing so it to penalize functions that use all the parameters of their family, for example a high degree polynomial that has non-zero coefficient on all powers. Such function is the most complex of its family and should be attained only in rare cases where that capacity is required. Noisy observation:

$$y = \mathbf{a}^\top \mathbf{x} + \varepsilon, \varepsilon \sim \mathcal{N}(0, \sigma)$$

Assume $\|\mathbf{a}\|_0 < d$ (not all input dimensions are used), can we force $\hat{\mathbf{a}}$ to be also sparse?

$$\min_{\mathbf{a}} \mathbb{E}_x[(y - \mathbf{x}^\top \mathbf{a})^2] + \Omega(\mathbf{a})$$

With $\Omega(\mathbf{a})$ a *regularizer* that increases cost for more complex $\mathbf{a}$

2.3.1 LASSO

Least Absolute Shrinkage and Selection Operator [Tibshirani, 1996]

$$\min_{\mathbf{a}} \frac{1}{n} \sum_i (y_i - \mathbf{x}_i^\top \mathbf{a})^2 + \lambda \|\mathbf{a}\|_1$$

Optimize using gradient descent

$$\mathbf{a} \leftarrow \mathbf{a} - \eta \left[\frac{-2}{n} \sum_i (y_i - \mathbf{x}_i^\top \mathbf{a}) + \lambda \text{sign}(\mathbf{a}) \right]$$

The LASSO is interesting in that it is not exactly the cost that we would like (the better cost would be the ℓ_0 norm that counts non zero components), but the one that has a similar behavior while being easier to optimize. There are countless specialized algorithms for optimizing the LASSO, but in practice and for large enough training sets, a gradient descent with a carefully chosen learning rate is good enough.

The main appeal for the LASSO is that enforcing a sparse model allows the practitioner to see which variables are in play. It has thus an explanatory value.

In [33]:

```
def sin_pred(a, X):
    return jnp.matmul(a, X)

def sin_lasso(a, X, y, lam):
    yp = sin_pred(a, X)
    return ((y - yp)**2).mean() + lam*jnp.abs(a).sum()

@jax.jit
def update(a, X, y, lam):
    da = jax.grad(sin_lasso, argnums=0)(a, X, y, lam)
    return a - 0.05*da
```

In [34]:

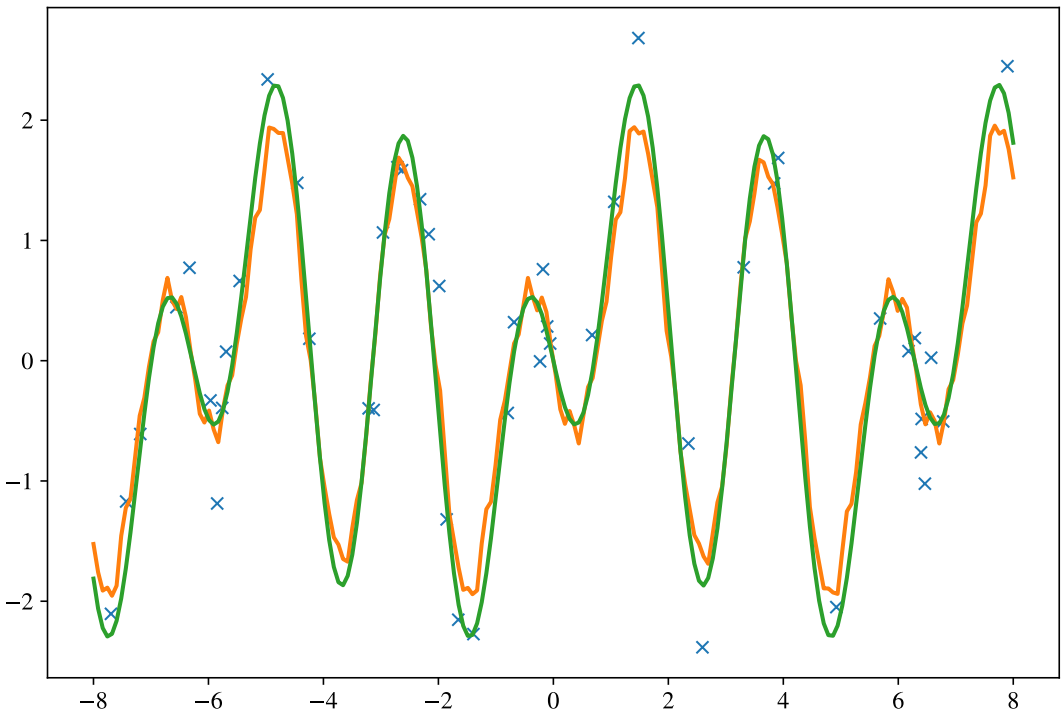
```
p=25
Xp = jnp.sin(jnp.matmul(x.reshape(-1, 1), 1+jnp.arange(p).reshape(1,p))).transpose()
ap = jnp.zeros(p)
for i in range(100):
    ap = update(ap, Xp, y, 0.1)
print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [ 5.0244969e-01  6.7708075e-01 -1.3217005e+00 -5.5618468e-03
  2.1797190e-03 -2.4922101e-03 -7.4531697e-02 -6.1854455e-03
  1.1975669e-03 -3.2022649e-03 -3.3327036e-03  3.0571646e-03
 -1.7762976e-02 -5.9072860e-03 -4.4773789e-03 -6.0488996e-03
 -6.4997919e-02 -6.8982323e-03  1.4052609e-03 -7.1133333e-03
  3.2059646e-03 -7.8694215e-03 -4.8849126e-03  1.2485289e-02
  6.9161236e-02]
```

In [35]:

```
t = jnp.linspace(-8, 8, 200)
T = jnp.sin(np.matmul(t.reshape(-1, 1), 1+jnp.arange(p).reshape(1,p))).transpose()
plt.plot(x, y, 'x')
plt.plot(t, jnp.matmul(ap, T[:len(ap), :]))
plt.plot(t, jnp.matmul(a, T[:len(a), :]))
print('MSE: {}'.format(((y - Yp)**2).mean()))
```

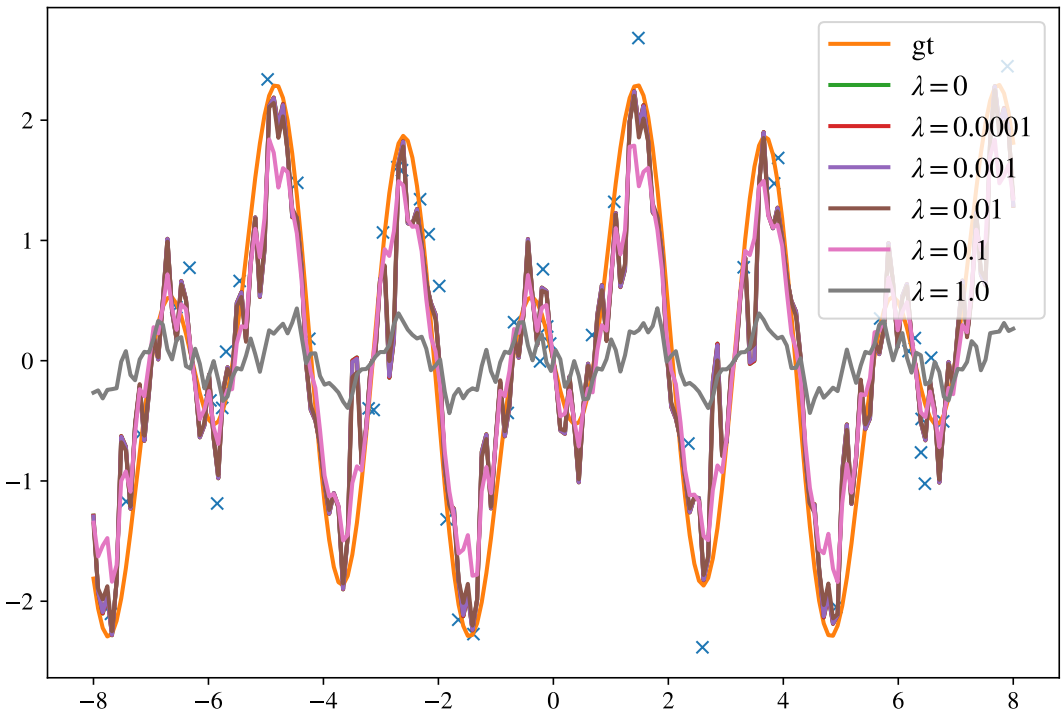
```
MSE: 0.06530888378620148
```



In [36]:

```
Xp = jnp.sin(jnp.matmul(x.reshape(-1, 1), 1+jnp.arange(p).reshape(1,p))).transpose()
plt.plot(x, y, 'x')
plt.plot(t, jnp.matmul(a, T[:len(a), :]), label='gt')
for lam in [0, 0.0001, 0.001, 0.01, 0.1, 1.0]:
    ap = jnp.zeros(p)
    for i in range(50):
        ap = update(ap, Xp, y, lam)
    plt.plot(t, jnp.matmul(ap, T[:len(ap), :]), label='$\lambda={}$'.format(lam))
plt.legend()
```

```
<matplotlib.legend.Legend at 0x710055c4b220>
```



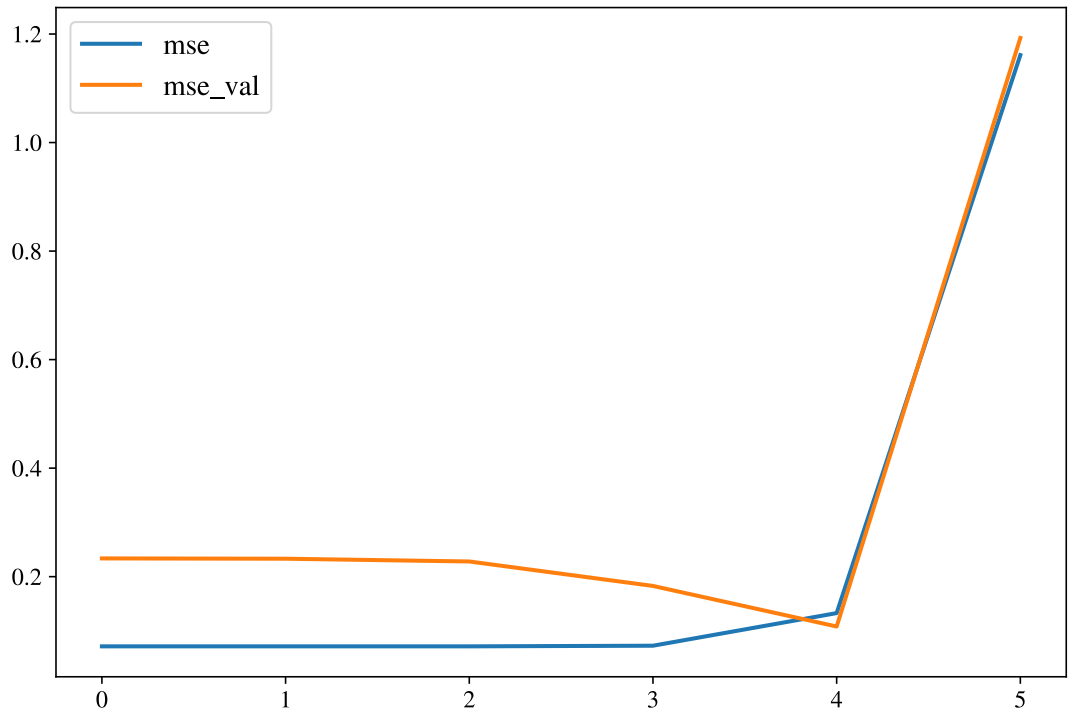
In [37]:

```
def lasso_approx(Xp, y, lam):
    ap = jnp.zeros(len(Xp[:,0]))
    for i in range(100):
        ap = update(ap, Xp, y, lam)
    Yp = jnp.matmul(ap, Xp)
    return ap, Xp, Yp
```

In [38]:

```
Xp = jnp.sin(jnp.matmul(x.reshape(-1, 1), 1+jnp.arange(p).reshape(1,p))).transpose()
Xp_val = jnp.sin(jnp.matmul(x_val.reshape(-1, 1),
1+jnp.arange(p).reshape(1,p))).transpose()
mse = []
mse_val = []
for lam in [0, 0.0001, 0.001, 0.01, 0.1, 1.0]:
    ap, Xp, Yp = lasso_approx(Xp, y, lam)
    Yp_val = jnp.matmul(ap, Xp_val)
    mse.append(((y - Yp)**2).mean()), mse_val.append(((y_val - Yp_val)**2).mean())
plt.plot(mse, label='mse'); plt.plot(mse_val, label='mse_val')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x710055c5d7b0>
```



We exchanged one hyper-parameter (the degree) for another one (λ), and so we have no other choice but to use a validation set to select its value. However, we still gained something: we separated the idea of observing the function outside of the training set from selecting the better model. In the case of selecting the degree directly with the validation set, both where done on the same data, whereas with the regularization, we "observe" the function everywhere thanks to our regularizer without using any additional data and with select the correct hyper-parameter with the validation set.

2.3.2 Analysis

Project $\mathbf{X}$ into its eigenspace:

$$\begin{aligned}\min_{\mathbf{a}} \frac{1}{n}\|\mathbf{y} - \mathbf{X}^\top \mathbf{U}\mathbf{a}\|^2 + \lambda\|\mathbf{a}\|_1 \\ = \frac{1}{n}\|\mathbf{y} - \mathbf{V}\mathbf{S}\mathbf{a}\|^2 + \lambda\|\mathbf{a}\|_1\end{aligned}$$

Stationary condition:

$$\begin{aligned}\frac{\partial}{\partial \mathbf{a}} = 0 = -2\mathbf{S}\mathbf{V}^\top\mathbf{y} + 2\mathbf{S}^2\mathbf{a} + \lambda\text{sign}(\mathbf{a}) \\ \mathbf{a} = \mathbf{S}^{-1}\mathbf{V}^\top\mathbf{y} - \mathbf{S}^{-2}\frac{\lambda\text{sign}(\mathbf{a})}{2}\end{aligned}$$

Let $\tilde{\mathbf{a}} = \mathbf{S}^{-1}\mathbf{V}^\top\mathbf{y}$

Note that $\text{sign}(\mathbf{a}) = \text{sign}(\tilde{\mathbf{a}}) = \frac{\tilde{\mathbf{a}}}{|\tilde{\mathbf{a}}|}$

$$\mathbf{a} = \tilde{\mathbf{a}}\left(1 - \frac{\lambda\mathbf{S}^{-2}}{2|\tilde{\mathbf{a}}|}\right)$$

Case > 0, $\text{sign}(\mathbf{a}) = \text{sign}(\tilde{\mathbf{a}}) = 1$

$$\mathbf{a}_i = \underbrace{\tilde{\mathbf{a}}_i}_{>0}\left(1 - \frac{\lambda \mathbf{S}_i^{-2}}{2|\tilde{\mathbf{a}}_i|}\right) > 0$$

$$\mathbf{a}_i = \tilde{\mathbf{a}}_i \max\left(0, 1 - \frac{\lambda \mathbf{S}_i^{-2}}{2|\tilde{\mathbf{a}}_i|}\right)$$

Case < 0, $\text{sign}(\mathbf{a}) = \text{sign}(\tilde{\mathbf{a}}) = -1$

$$\mathbf{a}_i = \underbrace{\tilde{\mathbf{a}}_i}_{<0}\left(1 - \frac{\lambda \mathbf{S}^{-2}}{2|\tilde{\mathbf{a}}_i|}\right) < 0$$

$$\mathbf{a}_i = \tilde{\mathbf{a}}_i \max\left(0, 1 - \frac{\lambda \mathbf{S}^{-2}}{2|\tilde{\mathbf{a}}_i|}\right)$$

Soft thresholding:

$$\mathbf{a} = \tilde{\mathbf{a}} \max\left(0, 1 - \frac{\lambda \mathbf{S}^{-2}}{2|\tilde{\mathbf{a}}|}\right)$$

λ removes components that would change the sign of the solution $\rightarrow$ Sparse solution
Remember: analysis only valid in eigenspace

2.3.3 Conditioning

In eigenspace, pseudo-inverse solution:

$$\mathbf{a} = \mathbf{U}\mathbf{S}^{-1}\mathbf{V}^\top \mathbf{y}$$

What if $\mathbf{S}$ has small eigenvalues ($\text{span}(\mathbf{X}) < d$)? How to prevent solution to focus on the noise? Avoid large values in the solution:

$$\min_{\mathbf{a}} \frac{1}{n}\sum_i (y_i - \mathbf{x}_i^\top \mathbf{a})^2 + \lambda \|\mathbf{a}\|^2$$

Tikonov regularization (ridge regression) Tikonov or ℓ_2 regularization is the most popular regularization scheme. It shows up everywhere. The intuition is very simple: you do not want some parameters of the model to blow up because of the optimization, so you penalize large values. Simple, but effective.

2.3.4 Analysis

$$\frac{1}{n}\|\mathbf{y} - \mathbf{X}^\top \mathbf{a}\|^2 + \lambda \|\mathbf{a}\|^2$$

Stationary condition:

$$\frac{\partial}{\partial \mathbf{a}} = 0 = \frac{2}{n}(-\mathbf{X}^\top \mathbf{y} + \mathbf{X}\mathbf{X}^\top \mathbf{a}) + 2\lambda \mathbf{a}$$

$$\mathbf{a} = (\mathbf{X}\mathbf{X}^\top + n\lambda \mathbf{I})^{-1}\mathbf{X}^\top \mathbf{y}$$

Offsetting all eigenvalues in the covariance matrix by λ Adding entries on the diagonal on the covariance matrix (hence the name *ridge*) has two effect: First, it makes sure the entire

space is spanned by the data making the inversion possible. Second, it compresses the importance of the larger dimensions, making the orginal data look more and more like a sphere. A similar effect could be obtained by a whitened PCA (reduce dimension and standardize to have variance 1 everywhere).

2.3.5 Elastic net

Add both regularization

$$\min_{\mathbf{a}} \frac{1}{n} \sum_i (y_i - \mathbf{x}_i^\top \mathbf{a})^2 + \lambda_1 \|\mathbf{a}\|_1 + \lambda_2 \|\mathbf{a}\|_2^2$$

- λ_1 controls sparsity
- λ_2 controls sensitivity to noisy components

Optimize using gradient descent You may want to do both because they have complementary properties. ℓ_1 produces a sparse solution that is explanatory, ℓ_2 prevents some parameters to blow up.

2.4 Other loss functions

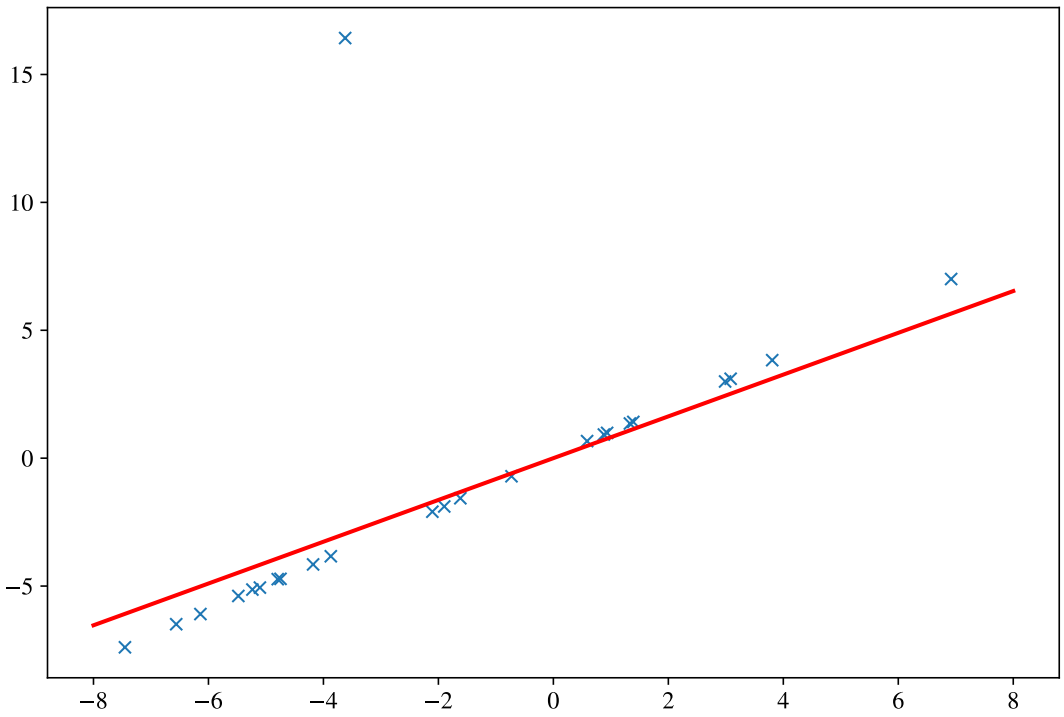
ℓ_2 is sensitive to outliers

In [39]:

```
x = np.random.rand(24)*16-8
y = x + 0.1*np.random.rand(24)
y[0] += 20

a = jnp.dot(x, y)/jnp.dot(x, x)
print(a)
t = jnp.linspace(-8, 8, 10)
plt.plot(x, y, 'x')
plt.plot(t, a*t, '-r')
```

```
0.8166666
```

```
[<matplotlib.lines.Line2D at 0x710055a97f40>]
```



2.4.1 MAE

Mean absolute error (or ℓ_1 error)

$$\min_{\mathbf{a}} \mathbb{E}[|y_i - \mathbf{a}^\top \mathbf{x}_i|]$$

Vector case

$$\min_{\mathbf{a}} \mathbb{E}[\|\mathbf{y}_i - \mathbf{A}^\top \mathbf{x}_i\|_1]$$

No close form solution, gradient descent (subderivative $\nabla \|0\|_1 = 0$)

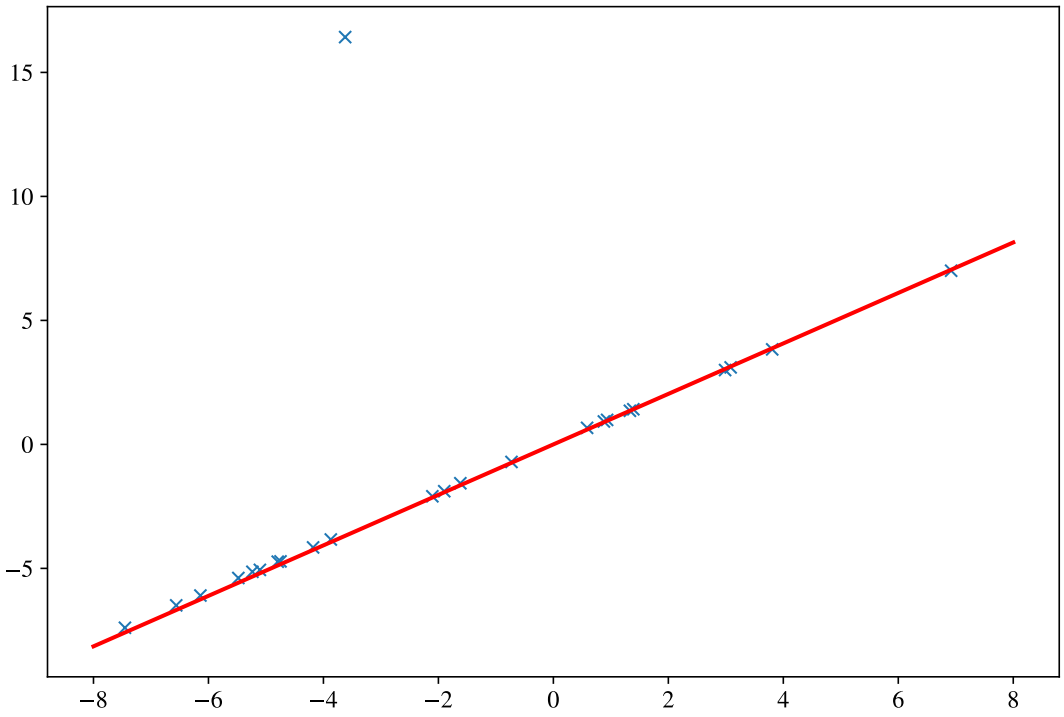
robust regression

In [40]:

```
def l1(a, x, y):
    return jnp.abs(y - a*x).mean()

@jax.jit
def update(a, x, y):
    da = jax.grad(l1, argnums=0)(a, x, y)
    return a - 0.02*da

a = 0.
for i in range(100):
    a = update(a, x, y)
print(a)
t = jnp.linspace(-8, 8, 10)
plt.plot(x, y, 'x')
plt.plot(t, a*t, '-r')
```

```
1.0179636
```

```
[<matplotlib.lines.Line2D at 0x710055934760>]
```



Robust regression is important for tasks where details matter. For example, in everything image related, using ℓ_2 leads to blury images because small errors are not takens into account enough, whereas ℓ_1 usually produces sharper results.

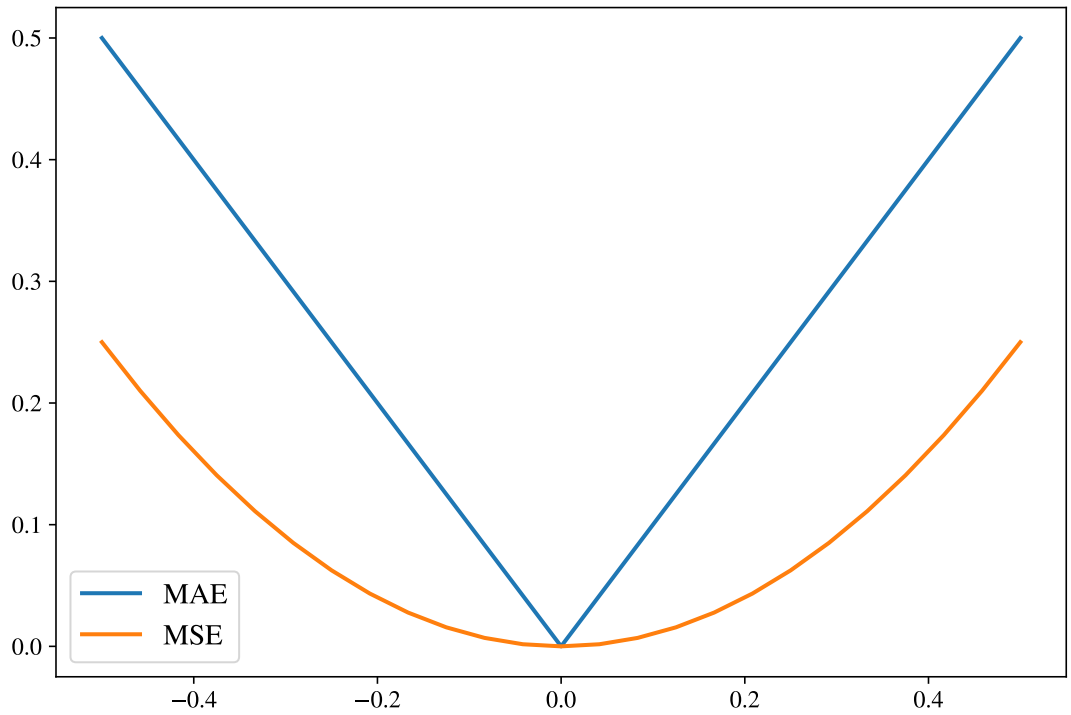
2.5 Sensitivity to small errors

- ℓ_2: derivative falls quickly to zero
- ℓ_1: constant derivative

In [41]:

```
t = jnp.linspace(-0.5, 0.5, 25)
plt.plot(t, jnp.abs(t), label='MAE')
plt.plot(t, t**2, label='MSE')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x710055993c70>
```



The caveat is that although ℓ_2 can be longer to optimize because of the smaller and smaller gradient, ℓ_1 can be trickier because of jittering (when the solution oscillates between different sides of the optimal point).

2.5.1 Do both?

- Penalize large errors: ℓ_2
- Penalize small errors (assuming no outliers): ℓ_1

$$\min_{\mathbf{A}} \mathbb{E}[\|\mathbf{y} - \mathbf{A}^\top \mathbf{x}\|^2 + \lambda \|\mathbf{y} - \mathbf{A}^\top \mathbf{x}\|_1]$$

Optimize using gradient descent

2.5.2 Full model

- Ridge regularization (noisy components)
- Sparsity regularization (overcomplete model)
- Large errors penalization
- Small errors penalization

$$\min_{\mathbf{A}} \mathbb{E}[\|\mathbf{y} - \mathbf{A}^\top \mathbf{x}\|^2 + \lambda \|\mathbf{y} - \mathbf{A}^\top \mathbf{x}\|_1] + \lambda_2 \|\mathbf{A}\|_F^2 + \lambda_1 \|\mathbf{A}\|_1$$

- Optimize using gradient descent (surprise!)
- 3 hyper-parameters: use proper cross validation (“*With four parameters I can fit an elephant, and with five I can make him wiggle his trunk*”, J. Von Neumann)

Using proper cross-validation is essential. People tend to vastly under-estimate the potential for overfitting with only 3 hyper-parameters. Here, all of them have a global impact on the solution, they are more potent in their effect than adding one to the degree of the polynomial.

2.6 Dictionary learning

Unsupervised learning: target space is a new representation of the input space

- Input: $\mathbf{X} \in \mathbb{R}^{d \times n}$
- Model: Dictionary $\mathbf{D} \in \mathbb{D}^{d \times p}$
- Output: Factors $\mathbf{A} \in \mathbb{R}^{p \times n}$

$$\min_{\mathbf{D},\mathbf{A}} \|\mathbf{X} - \mathbf{D}\mathbf{A}\|_F^2$$

If $p < d$, then $\mathbf{D}$ are the p leading left singular vectors of $\mathbf{X}$ and the factors $\mathbf{A}$ are the combination of the corresponding singular values with the right singular vectors If $p > d$, we have an *overcomplete* dictionary, which means we can afford to not use all entries to reconstruct a sample

$$\min_{\mathbf{a}} \|\mathbf{x} - \mathbf{D}\mathbf{a}\|^2 + \lambda \|\mathbf{a}\|_0$$

Sparse coding Alternate update:

- Fix $\mathbf{D}$, update $\mathbf{A}$
  - Difficult problem, relax to $\|\mathbf{a}\|_1$ or use iterative thresholding methods
- Fix $\mathbf{A}$, update $\mathbf{D}$

$$\mathbf{D} = \mathbf{X}(\mathbf{A}^\top \mathbf{A})^{-1}$$

2.6.1 K-SVD

Update one atom at a time (see [Aharon et al., 2006])

- $\mathbf{d}_k$: atom k of the dictionary
- $\mathbf{a}^k \in \mathbb{R}^n$: factors corresponding to atom k
- $\bar{\mathbf{D}}_k = [\mathbf{d}_i]_{i \neq k} \in \mathbb{R}^{d \times p-1}$: reduced dictionary without atom $\mathbf{d}_k$
- $\bar{\mathbf{A}}^k = [\mathbf{a}_i]_{i \neq k} \in \mathbb{R}^{p-1 \times n}$: factors corresponding to the reduced dictionary

$$\min_{\mathbf{D},\mathbf{A}} \|\mathbf{X} - \mathbf{D}\mathbf{A}\|_F^2 = \|\mathbf{X} - \bar{\mathbf{D}}_k \bar{\mathbf{A}}^k - \mathbf{d}_k \mathbf{a}^k\|_F^2$$

$$\mathbf{E}_k = \mathbf{X} - \bar{\mathbf{D}}_k \bar{\mathbf{A}}^k$$

Iterative updates:

$$\min_{\mathbf{d}_k, \mathbf{a}^k} \|\mathbf{E}_k - \mathbf{d}_k \mathbf{a}^k\|_F^2$$

- SVD of $\mathbf{E}_k = \mathbf{U}\mathbf{S}\mathbf{V}^\top$
- Rank 1 approximation: $\mathbf{E}_k \approx \mathbf{u}_1 s_1 \mathbf{v}_1^\top$
- Get hard thresholding selection matrix: $\Omega_k \in \{0,1\}^{n \times n'}$, that select n' samples that are coded by atom k (ex: highest absolute values of $\mathbf{v}_1$)
- Compute reduced problem for selected samples:

$$\min_{\mathbf{d}_k, \mathbf{a}^k} \|\mathbf{E}\Omega_k - \mathbf{d}_k \mathbf{a}^k \Omega_k\|_F^2$$

- Update $\mathbf{d}_k$ and $\mathbf{a}^k$ using rank-1 approximation of $\mathbf{E}_k\Omega_k \approx \mathbf{u}s\mathbf{v}^\top$

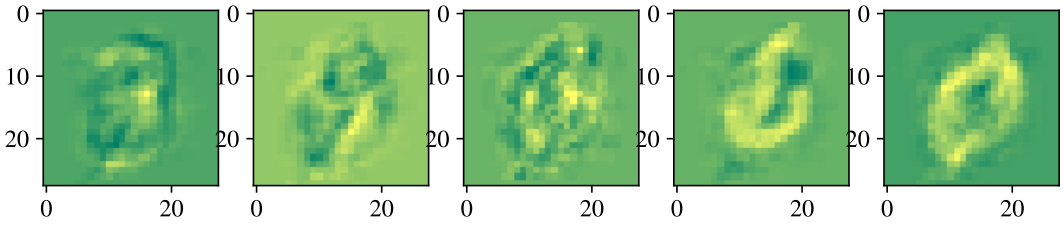
In [42]:

```
X = jnp.transpose(data['X_train'])
y = data['y_train']

D = np.random.rand(784, 64)
A = np.random.rand(64, 100)

for e in range(50):
    D = jnp.matmul(jnp.matmul(X, A.T), jnp.linalg.inv(jnp.matmul(A, A.T)))
    A = jnp.matmul(jnp.linalg.inv(jnp.matmul(D.T, D)), jnp.matmul(D.T, X))
    S = jnp.sign(A)
    I = jnp.argsort(jnp.abs(A), axis=0)[-33, :]
    A = S * jnp.clip(jnp.abs(A) - jnp.abs(A[I, jnp.arange(100)]), a_min=0)
```

In [43]:

```
plt.subplot(1,5,1)
plt.imshow(D[:,0].reshape(28, 28))
plt.subplot(1,5,2)
plt.imshow(D[:,1].reshape(28, 28))
plt.subplot(1,5,3)
plt.imshow(D[:,2].reshape(28, 28))
plt.subplot(1,5,4)
plt.imshow(D[:,3].reshape(28, 28))
plt.subplot(1,5,5)
plt.imshow(D[:,4].reshape(28, 28))
```

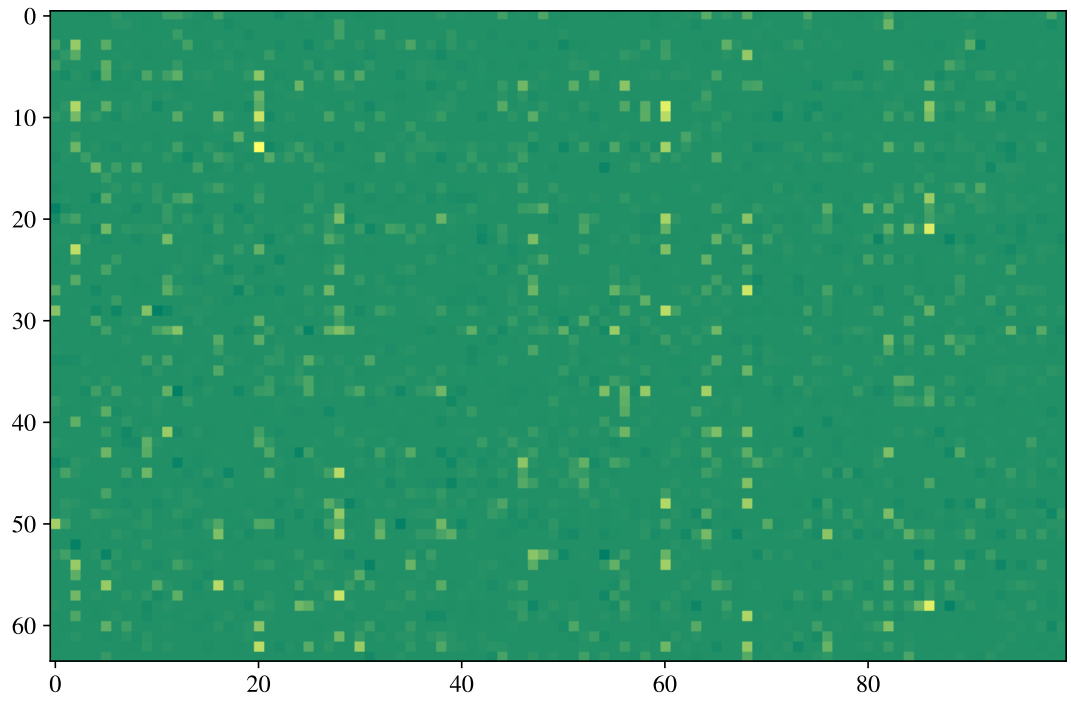
```
<matplotlib.image.AxesImage at 0x7100554fcfd0>
```



In [44]:

```
plt.imshow(A)
```

```
<matplotlib.image.AxesImage at 0x7100553ede10>
```



2.6.2 MNIST

Exercise: Try a linear regression (vector output) using **A** instead of **X**

In [ ]:

2.6.3 Why?

- **A** may provide a better alternative to **X** for doing learning a predictor
- **D** may provide insights (modes of **X**)

Dictionary learning was once a very hot topic because usually the original data is not in a good format to perform linear prediction on it. It is a way of obtaining a better representation, without requiring labels. It has since been mostly replaced by deep representation learning (especially self-supervised learning) but may make a comeback in very specific cases. Relation to k-means

$$\min_{\mathbf{D},\mathbf{A}} \|\mathbf{X} - \mathbf{D}\mathbf{A}\|_F^2 \quad \text{s.t. } \forall i, \|\mathbf{a}_i\|_0 = 1$$

- Only a single atom selected per sample - Alternate optimization:

$$\mathbf{d}_k = \frac{\mathbf{X}\mathbf{a}^{k\top}}{\|\mathbf{a}^k\|_1}$$

$$\mathbf{a}_i = [\mathbf{1}_{m=n}]_m, n = \text{argmin}_k \|\mathbf{d}_k - \mathbf{x}_i\|$$

2.7 Linear Model (regression), take home

- MSE often leads to closed form solution
- MSE to penalize large errors, MAE to penalize small errors
- MAE robust to outliers

- Sensitivity to condition number: ℓ_2 regularization
- Sparse model: ℓ_1 regularization

- Dictionary learning
  - Find better representation with a linear model

- Non linear relation: explicit non-linear mapping + linear model

Chapter 3

Support Vector Machines and Kernels

3.1 Binary Linear Classification

Let us tackle the simplest classification problem there is by considering a linear predictor that has to distinguish d dimensional vectors into 2 classes.

- Input $\mathbf{x} \in \mathbb{R}^d$
- Output $y \in \{-1; 1\}$

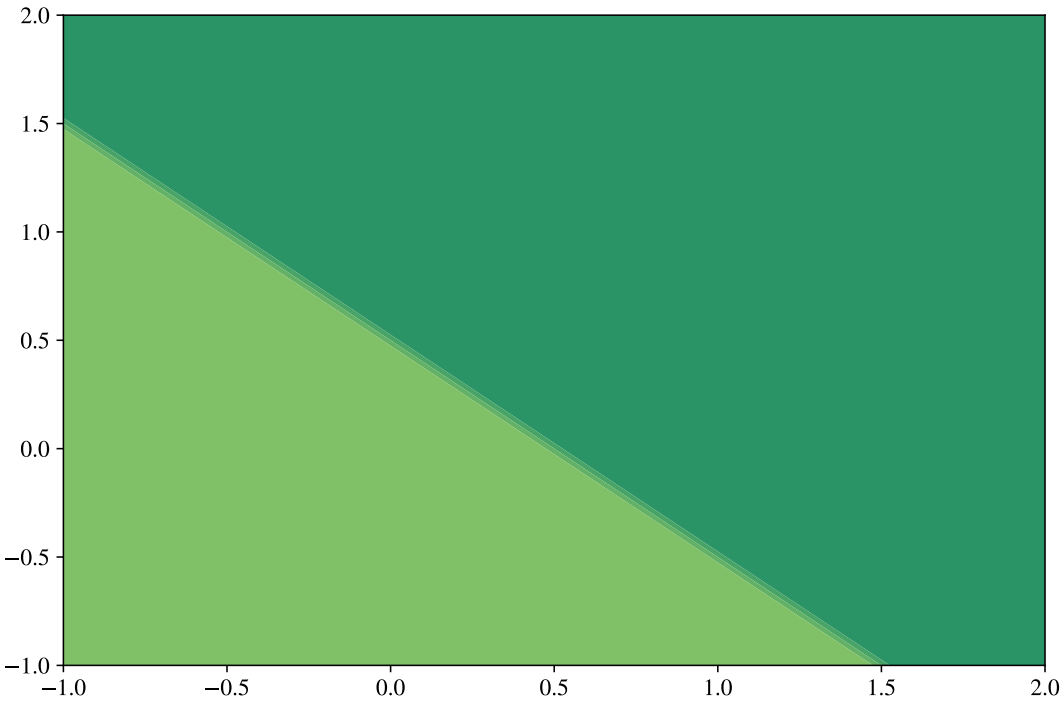
Linear prediction function

$$f(\mathbf{x}) = \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle + b)$$

Defines a hyperplane - Normal vector $\mathbf{w}$ - Bias (offset) b

In [2]:

```
w = jnp.ones(2)
b = -0.5

t = 40; tx = jnp.linspace(-1, 2, t); ty = jnp.linspace(-1, 2, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
levels=jnp.linspace(-1.5, 1.5, 10)
y_pred = (1.*(jnp.matmul(xx, w)+b > 0)).reshape(t, t)
plt.contourf(xv, yv, -y_pred, levels=levels);
```



This linear classifier can be obtained by optimizing a classification loss over a set of training pairs.

3.1.1 ERM

Hinge loss:

$$l(y, f(\mathbf{x})) = \max(0, 1 - yf(\mathbf{x}))$$

Given training set $\mathcal{A} = \{(\mathbf{x}, y)\}$, minimize the empirical risk:

$$\min_{\mathbf{w},b} \frac{1}{n} \sum_i max(0, 1 - y_i(\langle \mathbf{w}, \mathbf{x} \rangle + b))$$

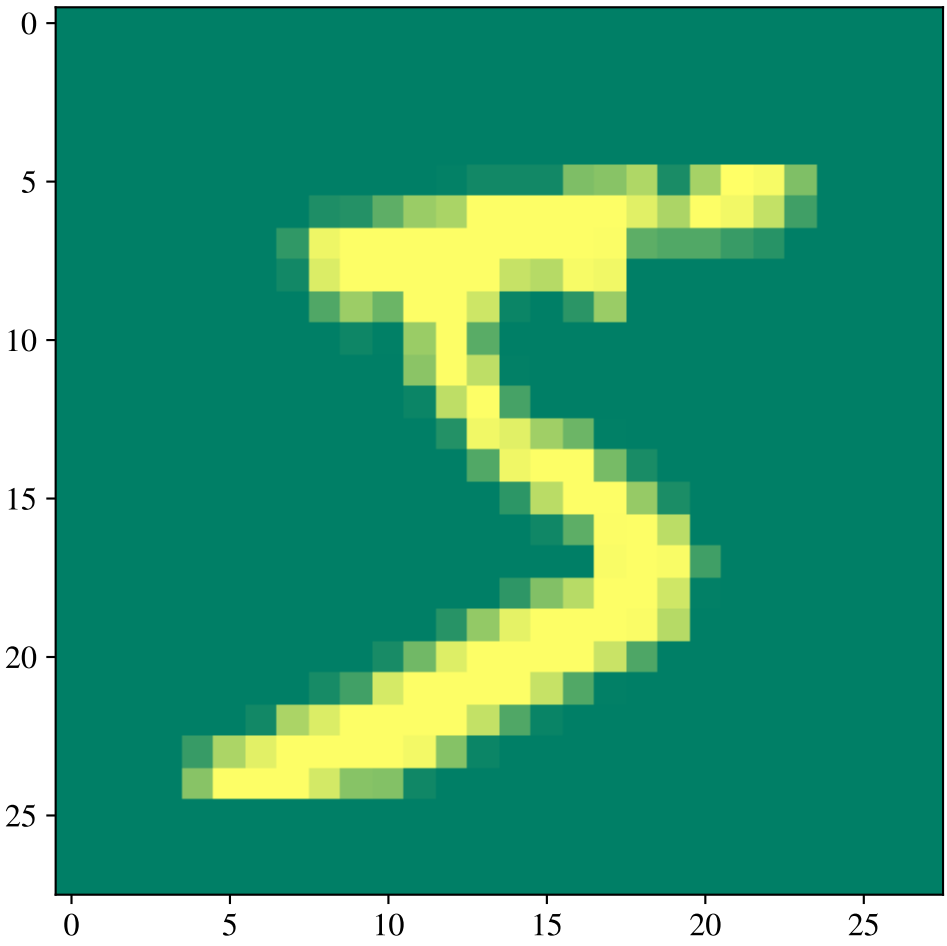
Convex problem (sum of convex) easy optimization by gradient descent For large training sets, stochastic gradient descent works great

3.1.2 MNIST

In [3]:

```
# Load the dataset
data = np.load('mnist.npz')
X = data['X_train']
y = data['y_train']
plt.imshow(X[0,:].reshape(28,28))
print(y[0])
```

```
5
```



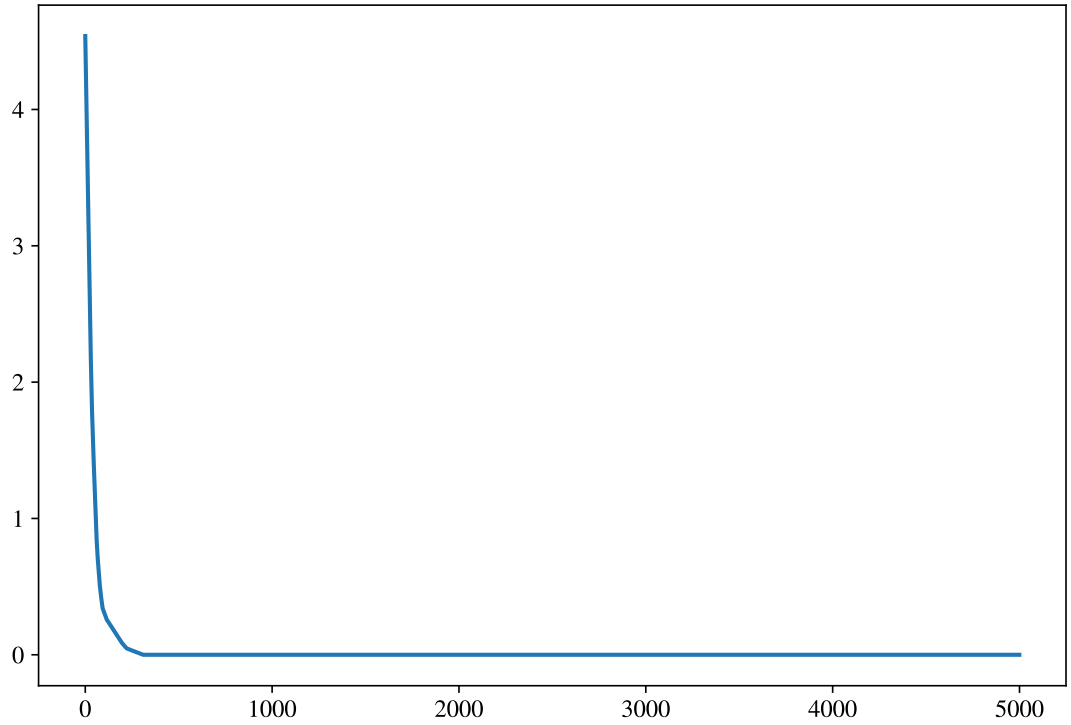
In [4]:

```
X = data['X_train_bin']
y = data['y_train_bin']*2-1

def func(w, b, x):
    return jnp.matmul(x, w) + b

def hinge(w, b, x, y):
    return jax.nn.relu(1 - y * func(w, b, x)).mean()

@jax.jit
def update(w, b, x, y):
    dw, db = jax.grad(hinge, argnums=(0,1))(w, b, x, y)
    return w - 0.01*dw, b - 0.01*db
```

In [5]:

```
w = np.random.randn(784)
b = 0.

loss = []
for t in range(5000):
    loss.append(hinge(w, b, X, y))
    w, b = update(w, b, X, y)
plt.plot(loss)
```

```
[<matplotlib.lines.Line2D at 0x787ed4287e50>]
```



In [6]:

```
def accuracy(y_pred, y_true):
    return jnp.sign(y_true*y_pred).mean()

y_pred = func(w, b, X)
print('accuracy: {}'.format(accuracy(y_pred, y)))
```

```
accuracy: 1.0
```

In [7]:

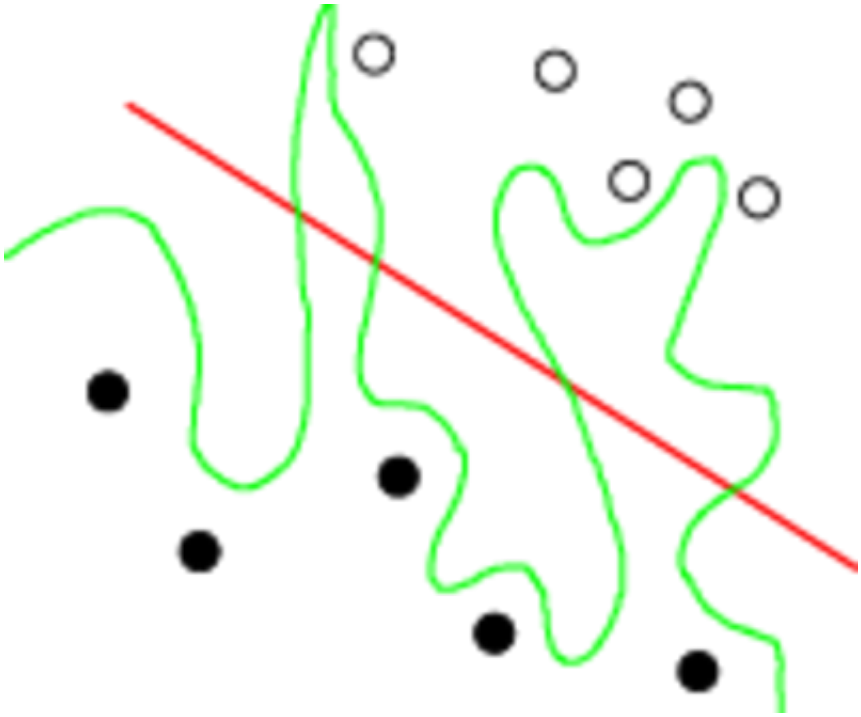
```
X_val = data['X_val_bin']
y_val = data['y_val_bin']*2-1

y_pred = func(w, b, X_val)
print('validation accuracy: {}'.format(accuracy(y_pred, y_val)))
```

```
validation accuracy: 0.9047619104385376
```

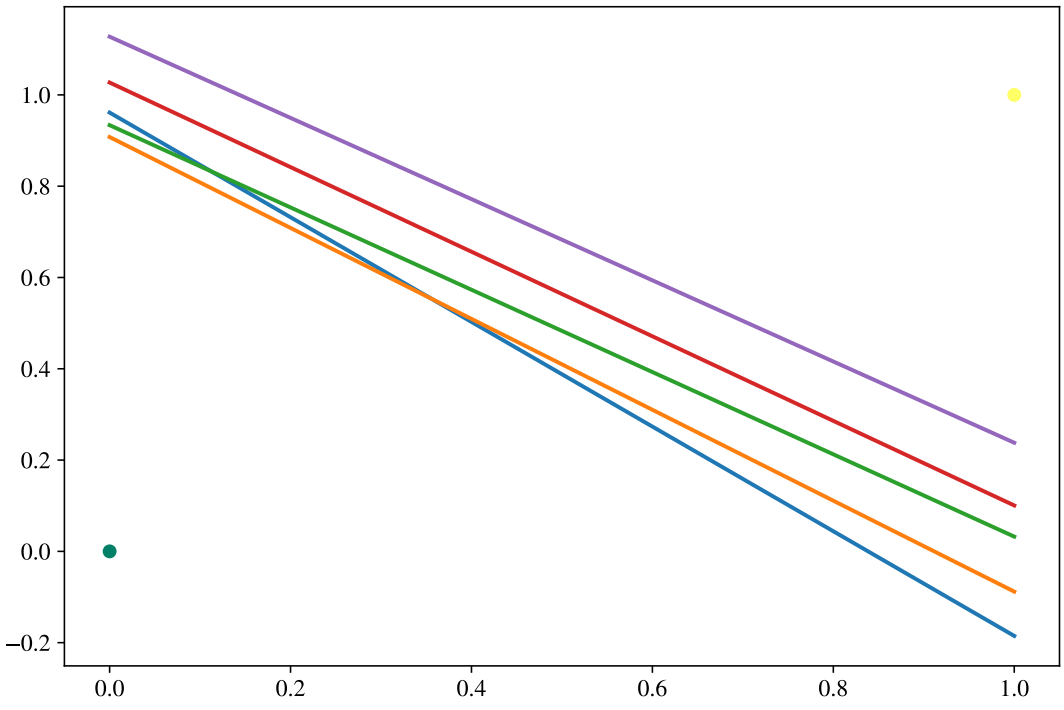
3.1.3 Equivalent solutions

In [8]:

```
w = jnp.array([-1, 1]) + 0.1*np.random.randn(5, 2)

plt.scatter([0, 1], [0, 1], c=[0, 1])
for i in range(5):
    plt.plot([0, 1], [w[i,1], w[i,0]+w[i,1]])
```

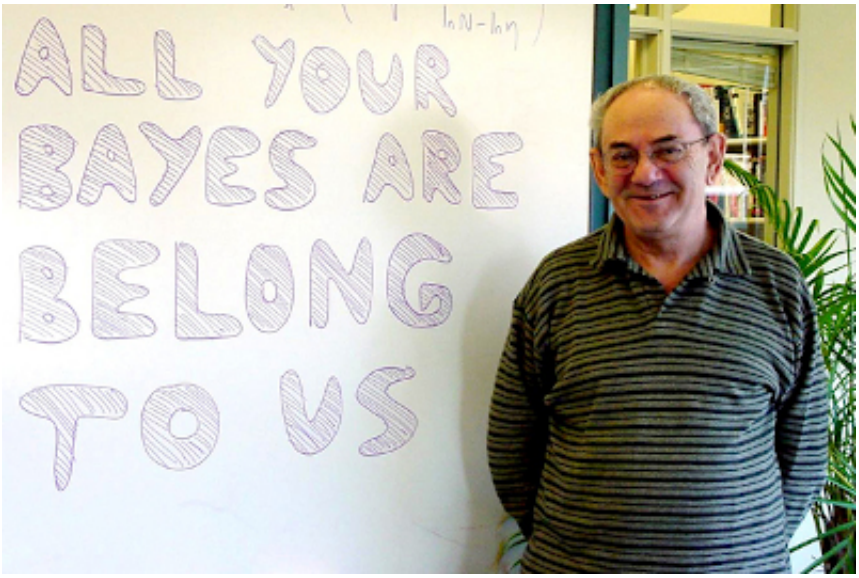


complexity.pdf.png



3.1.4 Complexity impacts generalization

Similarly to how we used regularization in regression to avoid having our function take absurd values outside of the training samples, we here want to enforce some regularity in our function to prevent having boundaries between classes with a complexe geometry that



vapnik

cannot be infered from the training data. The key idea is that the boundary should not be more *complex* than the data that was used to create it in the first place. Doing differently would be equivalent to hallucinating structure where there was none that we could observe. Of course, it all depends on the definition of *complex*, but this leads us to a great design choice which is to prefer *less complex* solutions when possible.

3.1.5 Structural Risk Minimization

The Structural Risk Minimization principle defines a trade-off between the quality of the approximation of the given data and the complexity of the approximating function (see [Vapnik, 1995])

3.1.6 SRM selection principle

Given a family of functions that all have $R_e = 0$ and that can be split into subsets S_k ordered by their complexity h_k

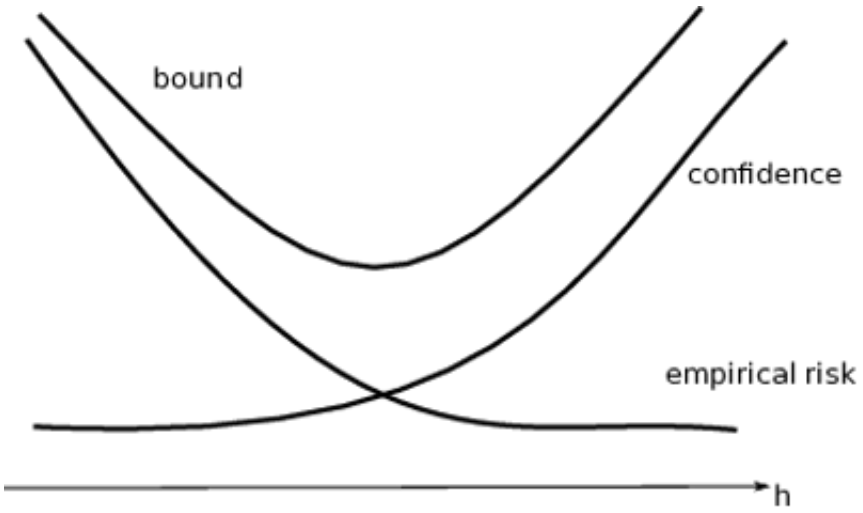
$$S_0 \subset S_1 \subset \cdots \subset S_N$$

$$h_0 \leq h_1 \leq \cdots \leq h_N$$

We choose the functions with the lowest complexity This is a machine learning of Occam's razor: among all possible explanations, the simplest is the best. Instead of explanations, we want to approximate a results, so among all functions that make similar approximation errors, we chose the simplest as measured by their complexity h_k.

3.1.7 Measuring complexity - VC Dimension

The VC Dimension of a set of indicator functions $Q(z, \alpha), \alpha \in \Lambda$, is the maximum number h of vectors $z_1, \ldots, z_h$ that can be separated into 2 classes in all 2^h possible ways using functions from the set. The VC dimension is a measure of complexity, and the nice thing about it is that it is solely defined in terms of data and not in terms of structural properties, etc. It is also very intuitive: the higher the number of points that could be perfectly classified however



struct

their labels, the more complex the function. The main drawback of VC is that it is very hard to estimate for functions that are not super simple. Also, it is a very crude measurement of the complexity.

3.1.8 Exercises

- What is the VC Dimension of linear functions in the 2D plane?
- What is the VC Dimension of axis-aligned rectangles in the 2D plane?

3.1.9 Risk Bound

True risk is bounded by a combination of empirical risk and structural risk depending on h (full results in [Vapnik, 1995])

$$R(\alpha) \leq R_e(\alpha) + F(h)$$

3.1.10 Large margin

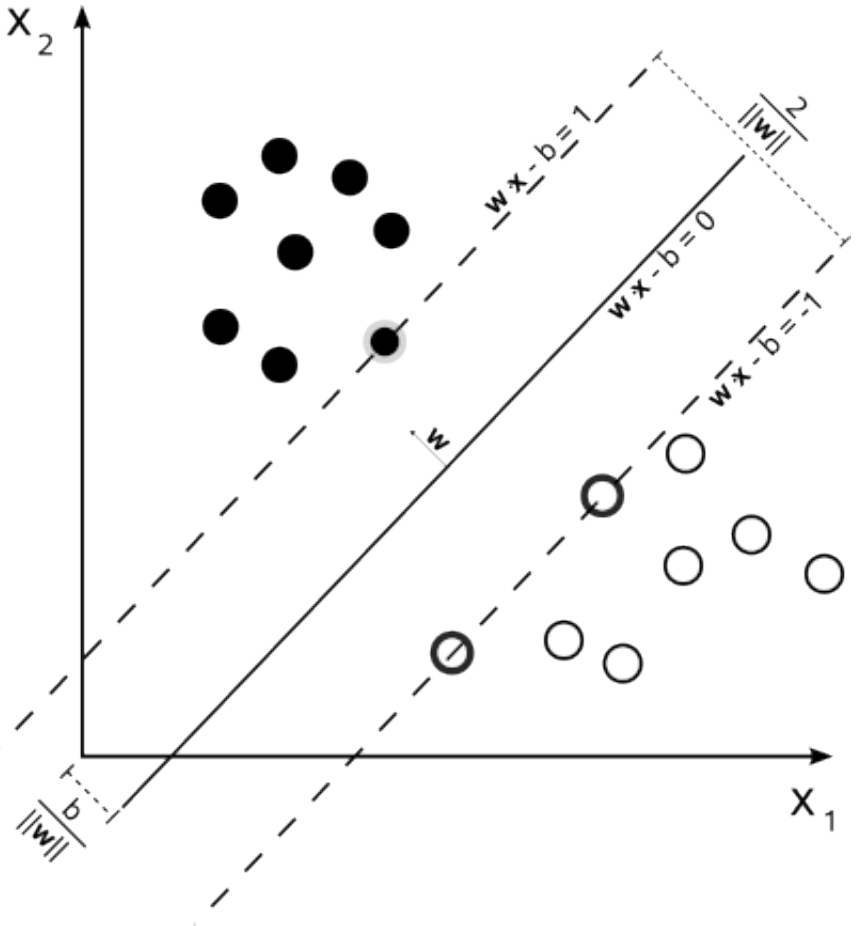
Let $\mathbf{w}$ be the separating hyperplane with margin Δ :

$$y = \begin{cases} 1 & \text{if } \mathbf{w}^\top x - b \geq \Delta, \\ -1 & \text{if } \mathbf{w}^\top x - b \leq -\Delta. \end{cases} \tag{3.1}$$

Theorem (Vapnik 1995): Given a training set $\mathcal{A} = \{(\mathbf{x}_i \in \mathbb{R}^d, y_i)\}$ such that $\|\mathbf{x}_i\| \leq R$ and that can be separated by an hyperplane with margin Δ,

$$h \leq \min\left(\frac{R^2}{\Delta^2}, d\right) + 1 \tag{3.2}$$

$\Rightarrow$ We have to maximize the margin



margin

3.1.11 ℓ_2 norm

$\Rightarrow$ we have to minimize $||w||^2$

This is fairly intuitive: we want the decision boundary to be as far as possible from the training samples because near these training samples, we assume the class is more likely to be that of the nearby sample, and thus not changing.

3.2 Support Vector Machines

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{s.t.} \quad & \forall i, y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1 \end{aligned}$$

Of all the hyperplane that perfectly classify the training sample, select the one with minimal norm, see [Boser et al., 1992].

3.2.1 Soft Margin

In practice, define a soft margin

$$\begin{aligned} \min_{\mathbf{w},b,\zeta_i} \quad & \frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{n}\sum_i \zeta_i \\ \text{s.t.} \quad & \forall i, y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1 - \zeta_i \end{aligned}$$

Solve the equivalent problem using stochastic gradient descent

$$\min_{\mathbf{w},b} \quad \frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{n}\sum_i \max(0, 1 - y_i(\mathbf{w}^\top \mathbf{x} + b))$$

Strongly convex problem The soft margin problem was introduced in [Cortes and Vapnik, 1995]. It is the most common and practical way to formulate the SVM problem as it is unlikely that our training set can be linearly separated to begin with.

3.2.2 MNIST Cont.

In [9]:

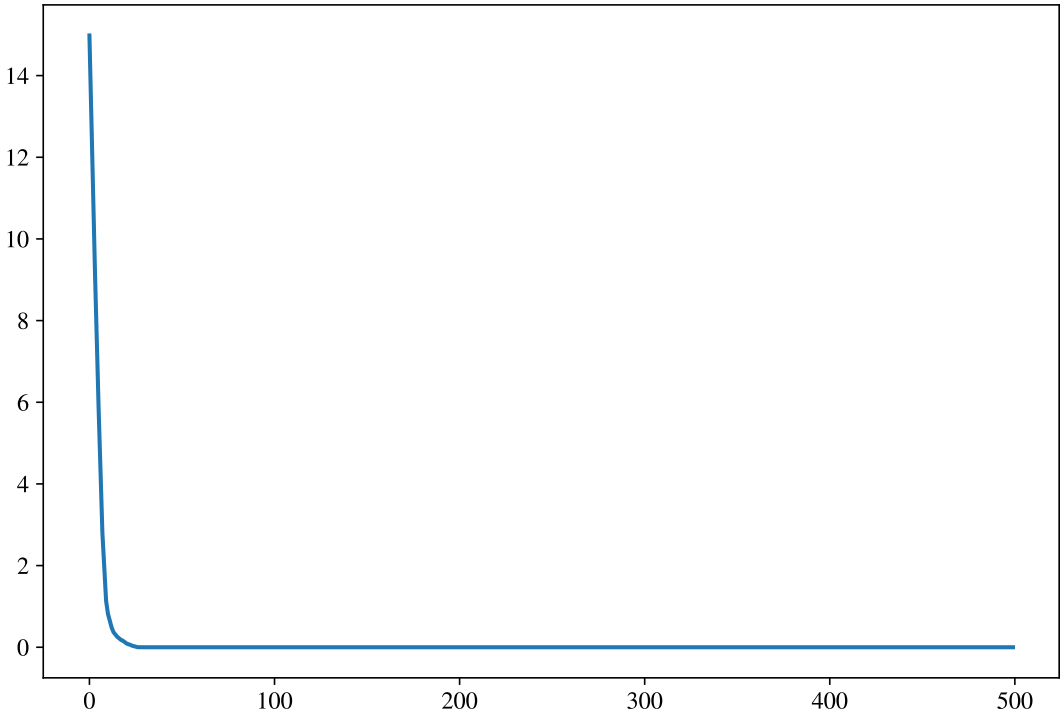
```
def func(w, b, x):
    return jnp.matmul(x, w) + b

def hinge(w, b, x, y):
    return jax.nn.relu(1 - y * func(w, b, x)).mean()

def loss(w, b, x, y):
    return 0.0001*(w*w).sum() + hinge(w, b, x, y)

@jax.jit
def update(w, b, x, y):
    dw, db = jax.grad(loss, argnums=(0,1))(w, b, x, y)
    return w - 0.1*dw, b - 0.01*db
```

In [10]:

```
w = np.random.randn(784)
b = 0.

l = []
for t in range(500):
```

```
    l.append(hinge(w, b, X, y))
    w, b = update(w, b, X, y)
plt.plot(l)
```

```
[<matplotlib.lines.Line2D at 0x787edc1ee410>]
```



In [11]:

```
def accuracy(y_pred, y_true):
    return jnp.sign(y_true*y_pred).mean()

y_pred = func(w, b, X)
print('accuracy: {}'.format(accuracy(y_pred, y)))
```

```
accuracy: 1.0
```

In [12]:

```
y_pred = func(w, b, X_val)
print('validation accuracy: {}'.format(accuracy(y_pred, y_val)))
```

```
validation accuracy: 0.4285714328289032
```

3.2.3 Multiple classes

2 types of approaches for handling M classes

- One versus All: M classifiers, take the argmax
- One versus One: $M(M-1)/2$ classifiers, majority vote

3.2.4 MNIST

One versus all

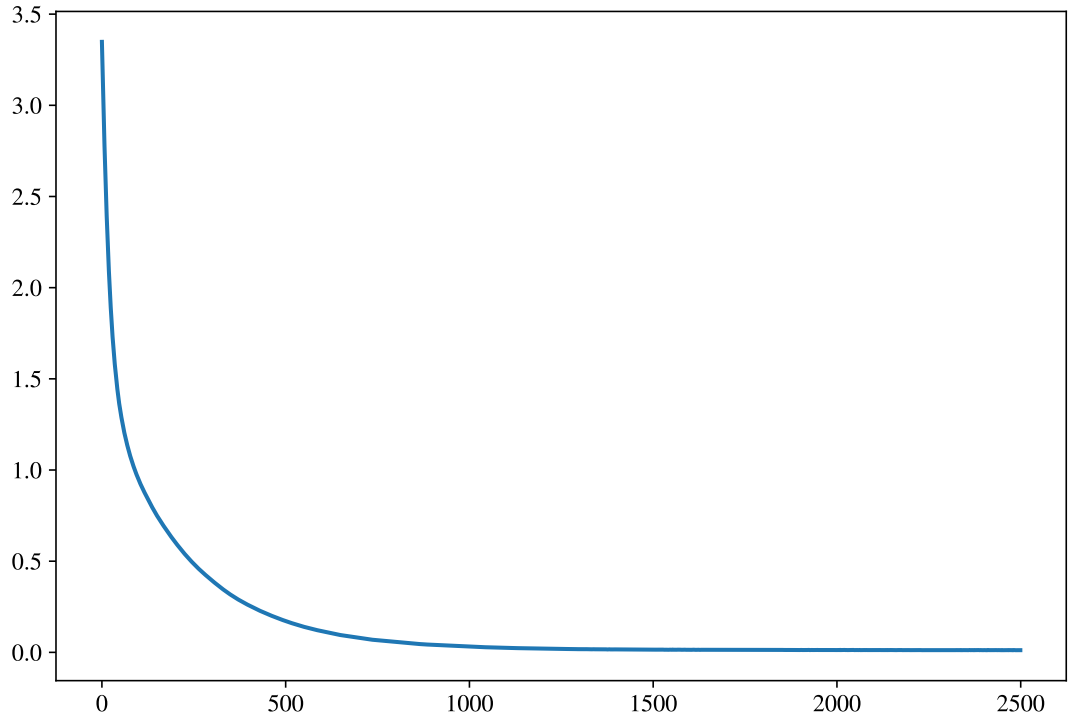
In [13]:

```
X = data['X_train']
y = jax.nn.one_hot(data['y_train'], 10)*2 - 1

def func(w, b, x):
    return jnp.matmul(x, w) + b

def hinge(w, b, x, y):
    return jax.nn.relu(1 - y * func(w, b, x)).mean()

def loss(w, b, x, y):
    return 0.01*(w*w).sum() + hinge(w, b, x, y)

@jax.jit
def update(w, b, x, y):
    dw, db = jax.grad(loss, argnums=(0,1))(w, b, x, y)
    return w - 0.1*dw, b - 0.1*db
```

In [14]:

```
w = np.random.randn(784, 10)
b = jnp.zeros(10)

l = []
for t in range(2500):
    l.append(hinge(w, b, X, y))
    w, b = update(w, b, X, y)
plt.plot(l)
```

```
[<matplotlib.lines.Line2D at 0x787ed47dc400>]
```



In [15]:

```
def accuracy(y_pred, y_true):
    return (1.*(jnp.argmax(y_true, axis=1) == jnp.argmax(y_pred, axis=1))).mean()

y_pred = func(w, b, X)
print('accuracy: {}'.format(accuracy(y_pred, y)))
```

```
accuracy: 1.0
```

In [16]:

```
X_val = data['X_val']
y_val = jax.nn.one_hot(data['y_val'], 10)*2 - 1

y_pred = func(w, b, X_val)
print('validation accuracy: {}'.format(accuracy(y_pred, y_val)))
```

```
validation accuracy: 0.6499999761581421
```

3.2.5 Dual Problem

Back to the hard margin:

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{s.t.} \quad & \forall i, y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1 \end{aligned}$$

Compute the Lagrangian:

$$\begin{aligned} \mathcal{L}(\mathbf{w}, b, \alpha) \quad & = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_i \alpha_i (y_i(\mathbf{w}^\top \mathbf{x_i} + b) - 1) \\ \text{s.t.} \quad & \forall i, \alpha_i \geq 0 \end{aligned}$$

α_i are the Lagrange multipliers for the augmented problem

3.2.6 KKT Conditions

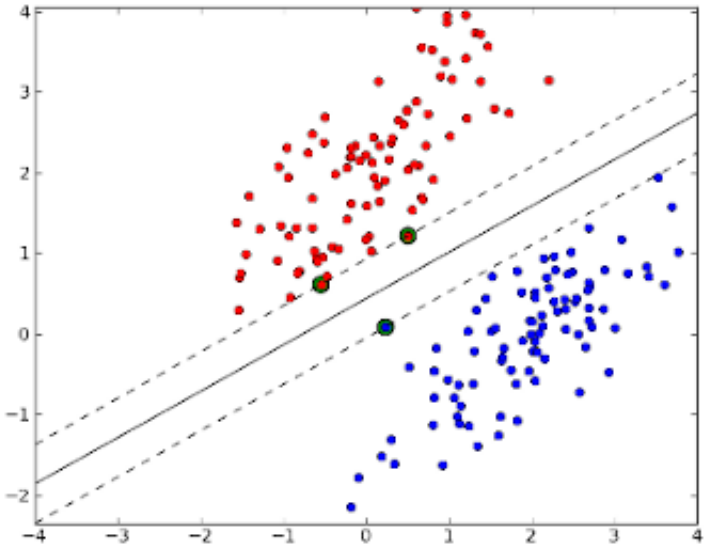
Karush-Kuhn-Tucker optimal conditions (well known in optimization):

- Stationarity: $\frac{\partial \mathcal{L}}{\partial \mathbf{w}^\star} = \mathbf{0}$
- Primal feasibility: $\forall i, y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1$
- Dual feasibility: $\forall i, \alpha_i \geq 0$
- Complementary slackness: $\forall i, \alpha_i(y_i(\mathbf{w}^\top \mathbf{x_i} + b) - 1) = 0$

3.2.7 Support vectors

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{w}^\star} &= \mathbf{w}^\star - \sum_i \alpha_i y_i \mathbf{x}_i \\ \mathbf{w}^\star - \sum_i \alpha_i y_i \mathbf{x}_i &= 0 \\ \mathbf{w}^\star &= \sum_i \alpha_i y_i \mathbf{x}_i \end{aligned}$$

$\mathbf{w}^\star$ is a linear combination of the training samples



sv

3.2.8 Representer theorem

Theorem (See [Schölkopf and Smola, 2002]): Let a training set $\mathcal{A} = \{(\mathbf{x}_i, y_i)\}$, an arbitrary error measuring function $l(\cdot, \cdot)$ and a stricly increasing function g, then any minimizer of the empirical risk

$$\mathbf{w}^\star = \text{argmin}_{\mathbf{w}} \frac{1}{n} \sum_i l(y_i, \langle \mathbf{w}, \mathbf{x}_i \rangle) + g(\|\mathbf{w}\|)$$

has a decomposition of the form

$$\mathbf{w}^\star = \sum_i \alpha_i \mathbf{x}_i$$

(The training set is a spanning set of the solution space)

3.2.9 Support Vectors cont.

Complementary slackness:

$$\forall i, \alpha_i (y_i (\mathbf{w}^\top \mathbf{x_i} + b) - 1) = 0$$

Which means

$$\mathbf{w}^\top \mathbf{x_i} + b \neq y_i \Rightarrow \alpha_i = 0$$

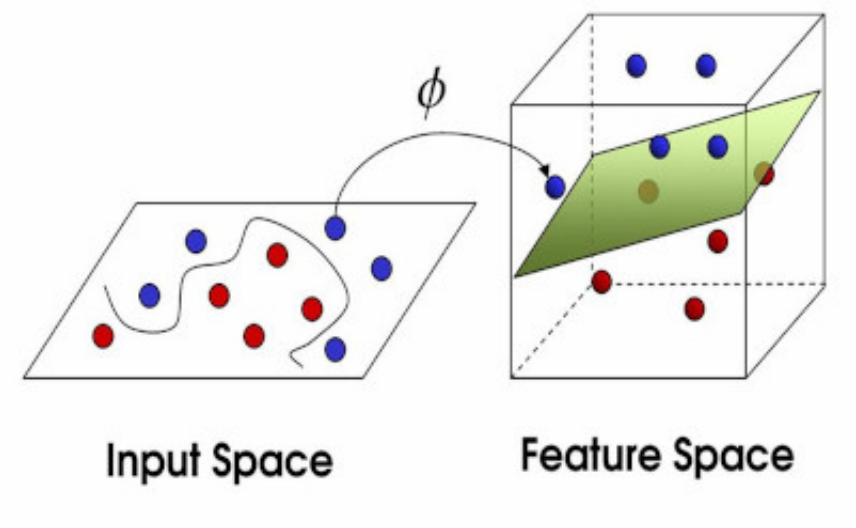
$\mathbf{w}^\star$ is a combination of the samples that are **on** the margin

3.2.10 Dual problem

Solving the dual problem:

$$\max_{\alpha} \inf_{\mathbf{w}, b} \mathcal{L}(\mathbf{w}, b, \alpha)$$

Since $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$



map

$$\inf_{\mathbf{w},b} \mathcal{L}(\mathbf{w}, b, \alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^\top \mathbf{x}_j$$

$$\max_{\alpha} D(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^\top \mathbf{x}_j$$

3.3 Kernels

- Remark that $D(\alpha)$ does not depend on $\mathbf{x}$ but only on $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$
- We can map $\mathbf{x}$ to a higher dimensional space using a non-linear mapping $\phi(\mathbf{x})$ (increase h)
- **Kernel trick**: we do not need to explicit ϕ, only $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$

$k(\mathbf{x}_i, \mathbf{x}_j)$ is called a **kernel** and defines the **similarity** between $\mathbf{x}_i$ and $\mathbf{x}_j$

3.3.1 Kernel map

3.3.2 Kernel SVM

Dual problem:

$$\max_{\alpha} D(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j)$$

Or in matrix form:

$$\max_{\alpha} D(\alpha) = \alpha^\top \mathbf{1} - \frac{1}{2} (\alpha \circ \mathbf{y})^\top \mathbf{K} (\alpha \circ \mathbf{y})$$

With $\circ$ the Hadamard (element wise) product, and $\mathbf{K}$ the Gram matrix of the kernel The kernelized version was already in the original [Boser et al., 1992] paper, although no good software were available at the time to efficiently solve the corresponding quadratic problem.

3.3.3 Kernels

- Explicit: $k(x_i, x_j) = \langle \phi(x_i)\phi(x_j) \rangle$
- Implicit: Symmetric positive definite function ($\forall \alpha_i, \forall \alpha_j, \sum_{ij} \alpha_i \alpha_j k(x_i, x_j) \geq 0$) is a kernel

Examples:

- Linear: $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \mathbf{x}_i, \mathbf{x}_j \rangle$
- Polynomial: $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \mathbf{x}_i, \mathbf{x}_j \rangle^p$ or $k(\mathbf{x}_i, \mathbf{x}_j) = (1 + \langle \mathbf{x}_i, \mathbf{x}_j \rangle)^p$
- Gaussian: $k(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2}$

3.3.4 Exercise

Show that the polynomial kernel and the Gaussian kernel are indeed kernels

3.3.5 Soft margin

Problem non linearly separable in the mapped space

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i \zeta_i \\ \text{s.t.} \quad & \forall i, y_i(\mathbf{w}^\top \mathbf{x} + b) \geq 1 - \zeta_i \\ & \forall i, \zeta_i \geq 0 \end{aligned}$$

Compute Lagrangian:

$$\begin{aligned} \mathcal{L} =& \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i \zeta_i - \sum_i \mu_i \zeta_i - \sum_i \alpha_i (y_i(\mathbf{w}^\top \mathbf{x}_i + b) - 1 + \zeta_i) \\ \text{s.t.} & \forall i, \alpha_i \geq 0, \mu_i \geq 0 \end{aligned}$$

3.3.6 KKT

Stationarity:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \mathbf{w} - \sum_i \alpha_i y_i \mathbf{x}_i$$

$\Rightarrow \mathbf{w}^\star = \sum_i \alpha_i y_i \mathbf{x}_i$

$$\frac{\partial \mathcal{L}}{\partial \zeta_i} = C - \mu_i - \alpha_i$$

$\Rightarrow C = \alpha_i + \mu_i \Rightarrow 0 \leq \alpha_i \leq C$

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_i \alpha_i y_i \Rightarrow \sum_i \alpha_i y_i = 0$$

3.3.7 Kernel SVM

$$\begin{aligned} \max_{\alpha} D(\alpha) = & \quad \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x_j}) \\ \text{s.t.} & \quad \forall i, 0 \leq \alpha_i \leq C, \sum_i \alpha_i y_i = 0 \end{aligned}$$

Similar to the hard margin problem but with upper bound on the Lagrange multipliers (a training sample can not contribute more than C)

Decision function:

$$f(\mathbf{x}) = \sum_i \alpha_i y_i k(\mathbf{x}, \mathbf{x_i}) + b$$

bias b is annoying because of $\sum \alpha_i y_i = 0$

3.3.8 K-SVM algorithm (SDCA)

Stochastic Dual Coordinate Ascent [Shalev-Shwartz and Zhang, 2013]

1. Initialize $\alpha = \mathbf{0}$
2. Pick random sample $\mathbf{x}_i$
3. Compute $z_i = y_i \sum_j \alpha_j y_j k(\mathbf{x}_i, \mathbf{x_j})$
4. Update $\alpha_i \leftarrow \alpha_i + (1 - z_i)/k(\mathbf{x}_i, \mathbf{x}_i)$
5. Clip $\alpha_i \leftarrow \max(0, \min(C, \alpha_i))$
6. Goto 2

Second order ascent using diagonal Hessian approximation

3.3.9 Toy test

In [17]:

```
def GaussKernel(x1, x2, gamma=10.0):
    return jnp.exp(-gamma*( jnp.linalg.norm(x1, axis=-1, keepdims=True)**2 +
jnp.linalg.norm(x2, axis=-1, keepdims=True).T**2 - 2*jnp.dot(x1, x2.T)))
```

In [18]:

```
def SDCAupdate(i, alpha, x, y, K, C=1.0, eps=1e-7):
    y_pred = jnp.dot(K, alpha)
    err = 1 - y[i]*y_pred[i]
    if jnp.abs(err) < eps:
        return alpha[i]
    da = err/K[i,i]
    ai = y[i] * jnp.maximum(0, jnp.minimum(C, da+y[i]*alpha[i]))
    return ai
```

In [19]:

```
def f_pred(x, alpha, x_train, gamma=10.0):
    K = GaussKernel(x, x_train, gamma)
    return jnp.dot(K, alpha)
```

In [20]:

```
def f_true(x):
    if x <= 0.25: return -1.
    if x < 0.25 and x <= 0.5: return 1.
    if x > 0.5 and x <= 0.75: return -1.
    return 1.
```

In [21]:

```
n = 20
gamma = 100.0
```

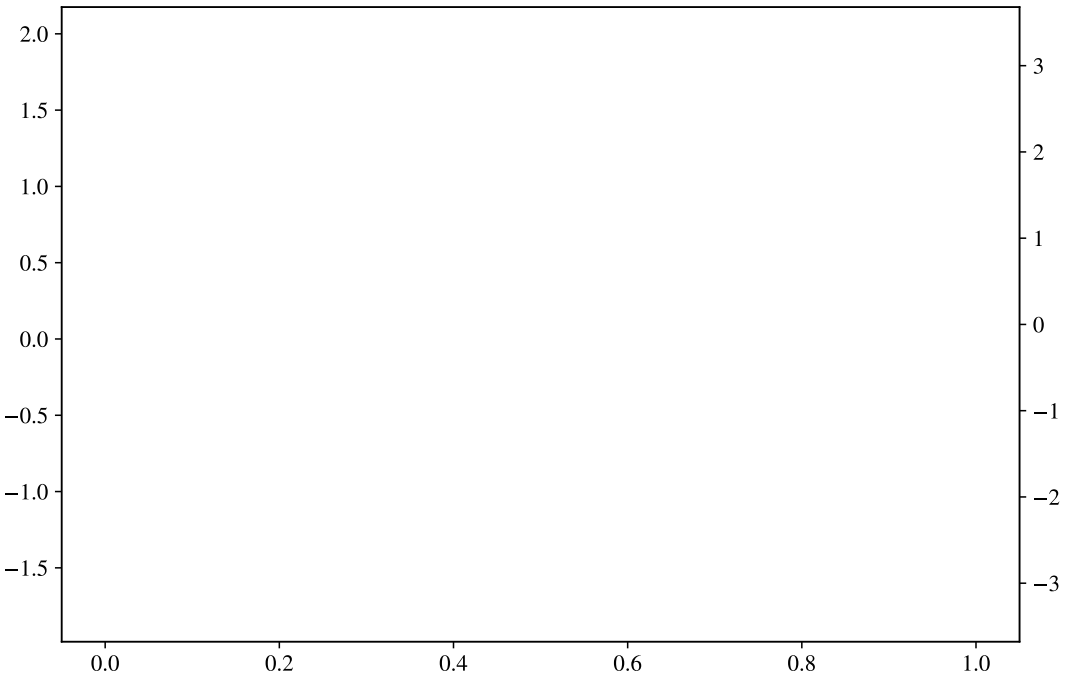
In [22]:

```
key = jax.random.PRNGKey(42)
x = jax.random.uniform(key, (n,1))
y = [f_true(xi) for xi in x]
```

In [23]:

```
alpha = jnp.zeros((n,1))
K = GaussKernel(x,x, gamma)

fig, ax1 = plt.subplots()
ax2 = ax1.twinx()
camera = Camera(fig)
t = jnp.arange(51)/50.

for e in range(10):
    r = jax.random.permutation(key, n)
    for i in r:
        ai = SDCAupdate(i, alpha, x, y, K, C=100.)
        alpha = alpha.at[i].set(ai)
        ax1.plot(t, [f_true(i) for i in t], '-k')
        y_pred = f_pred(t[:,None], alpha, x, gamma)
        ax1.plot(t, y_pred, '-r')
        ax2.stem(x, alpha, basefmt=" ")
        camera.snap()

animation = camera.animate()
HTML(animation.to_html5_video())
```

```
<IPython.core.display.HTML object>
```



3.3.10 Exercise

Knowing that the VC dimension of a linear classifier in $\mathbb{R}^d$ is $d+1$, - What is the VC dimension of a kernel SVM using $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \mathbf{x}_i, \mathbf{x}_j \rangle^2$?

- What is the VC dimension of a kernel SVM using $k(\mathbf{x}_i, \mathbf{x}_j) = exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2)$?

3.3.11 Reproducing Kernel Hilbert Space

A RKHS is a space $\mathcal{H}$ of functions $f : \mathcal{X} \to \mathbb{R}$ for which the pointwise evaluation corresponds to the dot product with specific functions

$$f(x) = \langle f, \phi_x \rangle$$

Since $\phi_x \in \mathcal{H}$, we have

$$\phi_x(y) = \langle \phi_x, \phi_y \rangle = k(x, y)$$

The theory of RKHS can be a bit overwhelming, especially the original material, but a very good overview can be found in [Rakotomamonjy et al., 2005].

3.3.12 Representer theorem

Theorem (simplified from [Schölkopf and Smola, 2002]): Let a training set $\mathcal{A} = \{(\mathbf{x}_i, y_i)\}$, $\mathcal{H}$ a Hilbert space of function associated with reproducing kernel k, an arbitrary error measuring function $l(\cdot, \cdot)$ and a stricly increasing function g, then any minimizer $f \in \mathcal{H}$ of the empirical risk

$$f^\star = \text{argmin}_{f \in \mathcal{H}} \frac{1}{n} \sum_i l(y_i, f(\mathbf{x}_i)) + g(\|\mathbf{f}\|_{\mathcal{H}})$$

has a decomposition of the form

$$f^\star = \sum_i \alpha_i k(\mathbf{x}_i, \cdot)$$

(The training set is a spanning set of the solution space)

3.3.13 Kernel approximation

Find an explicit mapping that approximate the kernel

$$k(\mathbf{x}_i, \mathbf{x}_j) \approx \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$$

Map the training set

$$\forall i, \bar{\mathbf{x}}_i = \phi(\mathbf{x_i})$$

Train a linear SVM on mapped samples

$$\min_{\mathbf{w}, b} \quad \frac{\lambda}{2} \|\mathbf{w}\|^2 + \frac{1}{n} \sum_i \max(0, 1 - y_i(\mathbf{w}^\top \bar{\mathbf{x}} + b))$$

Prediction function

$$f(\mathbf{x}) = \langle \mathbf{w}, \phi(\mathbf{x}) \rangle + b$$

3.3.14 Nyström approximation [Williams and Seeger, 2000]

Kernel matrix of the training set

$$\mathbf{K} = [k(\mathbf{x}_i, \mathbf{x}_j)]_{ij}$$

Low rank approximation

$$\mathbf{K} = \mathbf{U}\mathbf{L}\mathbf{U}^\top$$

Non linear projection

$$\phi(\mathbf{x}) = \mathbf{L}_m^{-1/2}\mathbf{U}_m^\top \mathbf{K}(x), \quad \mathbf{K}(x) = [k(\mathbf{x}_i, \mathbf{x})]_i$$

Limits the VC dimension to m There are many more kernel approximation methods, such as Random Fourier Features (see [Yang et al., 2012]) or Taylor expansion based (Random MacLaurin), but in practice Nyström is very effective and always a good start.

3.3.15 MNIST

In [24]:

```
def SquareKernel(X1, X2):
    return jnp.matmul(X1, X2.T)**2
```

In [25]:

```
X = data['X_train']
X = X/jnp.linalg.norm(X, axis=1)[:,None]
y = jax.nn.one_hot(data['y_train'], 10)*2 - 1
K = SquareKernel(X, X)
L, U = jnp.linalg.eigh(K)
L = L[-64:]
U = U[:, -64:]
P = jnp.sqrt(1./L)[:, None]*U.T
```

In [26]:

```
X_bar = jnp.matmul(P, K).T
```

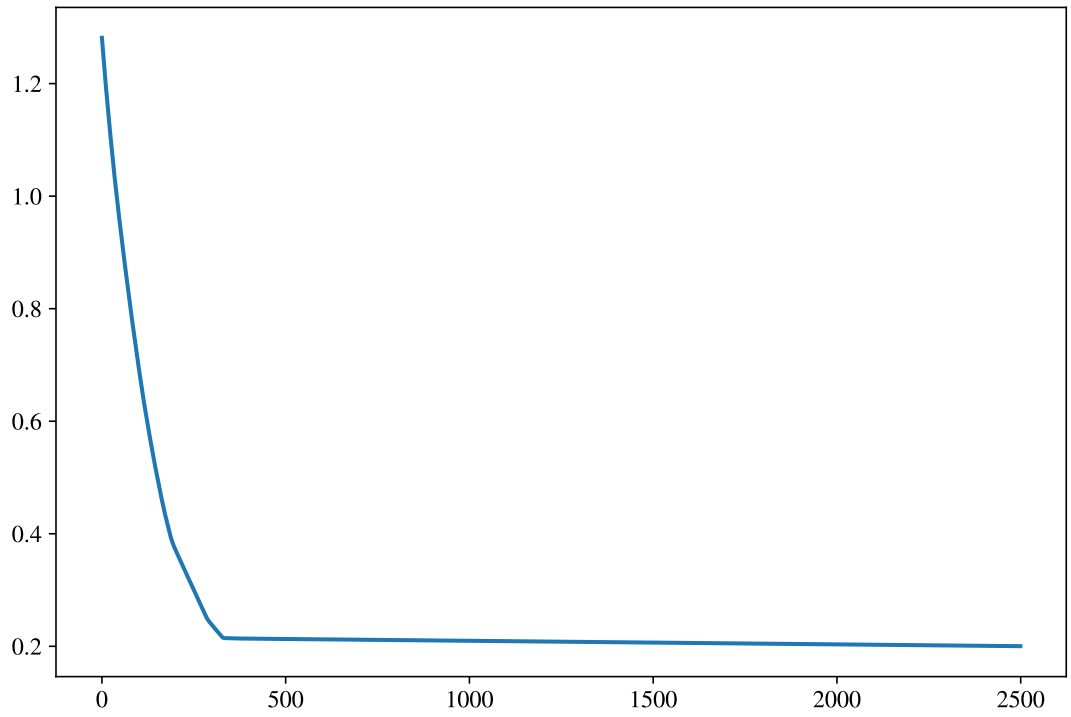
In [27]:

```
def func(w, b, x):
    return jnp.matmul(x, w) + b

def hinge(w, b, x, y):
    return jax.nn.relu(1 - y * func(w, b, x)).mean()

def loss(w, b, x, y):
    return 0.1*(w*w).sum() + hinge(w, b, x, y)

@jax.jit
def update(w, b, x, y):
    dw, db = jax.grad(loss, argnums=(0, 1))(w, b, x, y)
    return w - 0.1*dw, b - 0.1*db
```

In [28]:

```
w = np.random.randn(64, 10)
b = np.random.randn(10)

l = []

for t in range(2500):
    l.append(hinge(w, b, X_bar, y))
    w, b = update(w, b, X_bar, y)
plt.plot(l)
```

```
[<matplotlib.lines.Line2D at 0x787e88102a10>]
```



In [29]:

```
def accuracy(y_pred, y_true):
    return (1.*(jnp.argmax(y_true, axis=1) == jnp.argmax(y_pred, axis=1))).mean()

y_pred = func(w, b, X_bar)
print('accuracy: {}'.format(accuracy(y_pred, y)))
```

```
accuracy: 0.8899999856948853
```

In [30]:

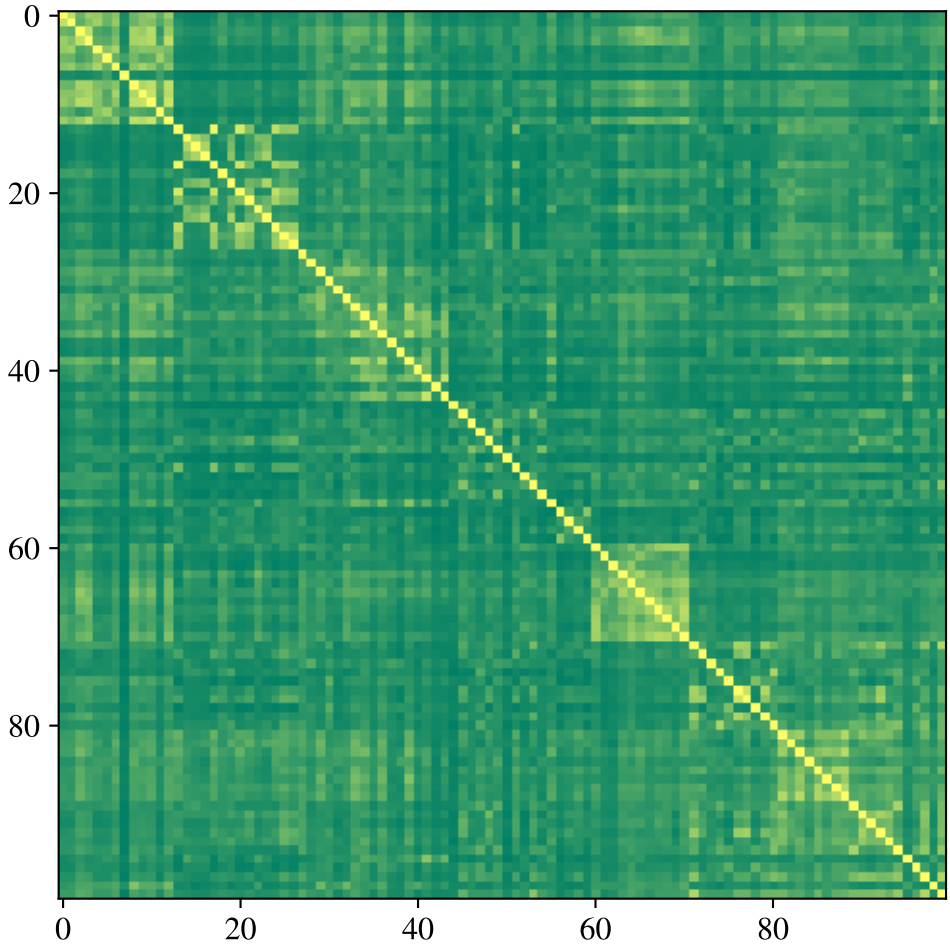
```
X_val = data['X_val']
X_val = X_val/jnp.linalg.norm(X_val, axis=1)[:,None]
y_val = jax.nn.one_hot(data['y_val'], 10)*2 - 1

K_val = SquareKernel(X, X_val)
X_valbar = jnp.matmul(P, K_val).T

y_pred = func(w, b, X_valbar)
print('validation accuracy: {}'.format(accuracy(y_pred, y_val)))
```

```
validation accuracy: 0.5299999713897705
```

In [31]:

```
Xt = X[jnp.argsort(data['y_train']), :]
Xt = Xt/jnp.linalg.norm(Xt, axis=1)[:,None]
Kt = SquareKernel(Xt, Xt)
plt.imshow(Kt)
```

```
<matplotlib.image.AxesImage at 0x787e68644e20>
```



3.3.16 Multiple Kernel Learning

Combination kernel

$$k(\mathbf{x}_i, \mathbf{x}_j) = \sum_m \beta_m k_m(\mathbf{x}_i, \mathbf{x}_j), \beta_m \geq 0$$

MKL problem:

$$\begin{aligned} \min_{\beta} \max_{\alpha} & \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \sum_m \beta_m k_m(\mathbf{x}_i, \mathbf{x_j}) \\ \text{s.t.} \quad & \forall i, 0 \leq \alpha_i \leq C \\ & \forall m, \beta_m \geq 0 \\ & \Omega(\beta) = 1 \end{aligned}$$

Norm constraint Ω: - $\Omega(\beta) = \|\beta\|_1$: Joint classification and feature selection - $\Omega(\beta) = \|\beta\|_2$: Joint classification and feature combination Multiple Kernel learning is an intellectually very satisfying approach, but it has limited interest in practice. While the idea of selecting both the features and the classifier at the same time is appealing, the question remains why not going the extra mile and doing deep learning where everything is learned together. So appart from setups where data is scares and there is good reason for performing handcrafted kernel selection/combination, MKL is no longer used.

3.3.17 Alternate optimization

1. Optimize α until optimal (e.g., SDCA)
2. Gradient descent step on β
3. Projection of β onto the constraint $\Omega(\beta) = 1$

3.3.18 Kernel ridge regression

Kernel function

$$k(\mathbf{x}_1, \mathbf{x}_2) = \langle \phi(\mathbf{x}_1), \phi(\mathbf{x}_2) \rangle$$

Ridge regression problem

$$\min_{\mathbf{w}} \frac{\lambda}{2} \|\mathbf{w}\|^2 + \frac{1}{n} \sum_i (y_i - \langle \mathbf{w}, \phi(\mathbf{x}_i) \rangle)^2$$

Representer theorem

$$\mathbf{w} = \sum_i \alpha_i \phi(\mathbf{x}_i)$$

Pseudo-inverse solution

$$\mathbf{w} = (\phi(\mathbf{X})\phi(\mathbf{X})^\top + \lambda I)^{-1} \phi(\mathbf{X})\mathbf{y}$$

Identity trick

$$(P^{-1} + BTR^{-1}B)^{-1} BTR^{-1} = PBT(BPBT + R)^{-1}$$

with

- $P^{-1} = \lambda I$
- $R = I$
- $B = \phi(\mathbf{X})$

we get

$$\mathbf{w} = \phi(\mathbf{X})(\mathbf{K} + \lambda I)^{-1}\mathbf{y}$$

Thus

$$\alpha = (\mathbf{K} + \lambda I)^{-1})\mathbf{y}$$

$$f(\mathbf{x}) = \sum_i \alpha_i k(\mathbf{x_i}, \mathbf{x})$$

3.4 SVM and kernel methods, take home

Linear binary classification:

- hinge loss
- Gradient descent or stochastic gradient descent
- many equivalent predictor

Linear SVM

- SRM: Structural risk Minimization (Occam's razor)
- VC Dimension: $d + 1$ for linear predictors
- ℓ_2 regularization of the predictor

Kernel SVM - Solve non-linearly separable problem with non-linear mapping - Implicit mapping $k(x_1, x_2) = \langle \phi(x_1), \phi(x_2) \rangle$ Multiclass

- One versus all
- One versus One with voting strategy

Kernel methods

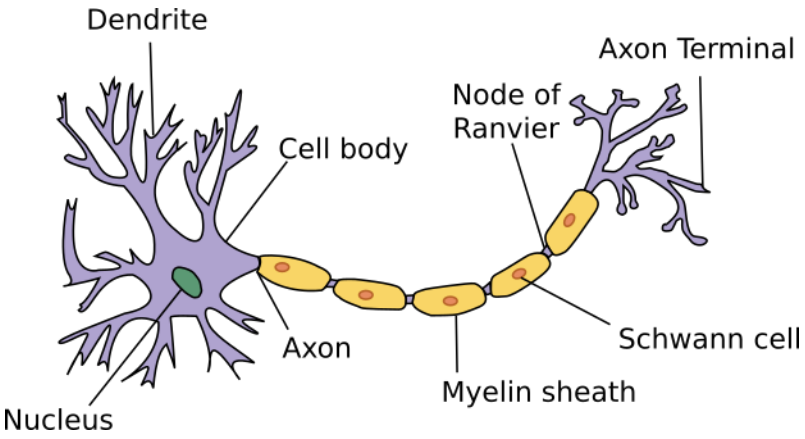
- Representer theorem: solution is in span$(\mathcal{A})$
- Combination is as large as the training set
- Support vectors for hinge loss
- Approximate kernel methods

Chapter 4

Neural Networks

4.1 Natural neuron

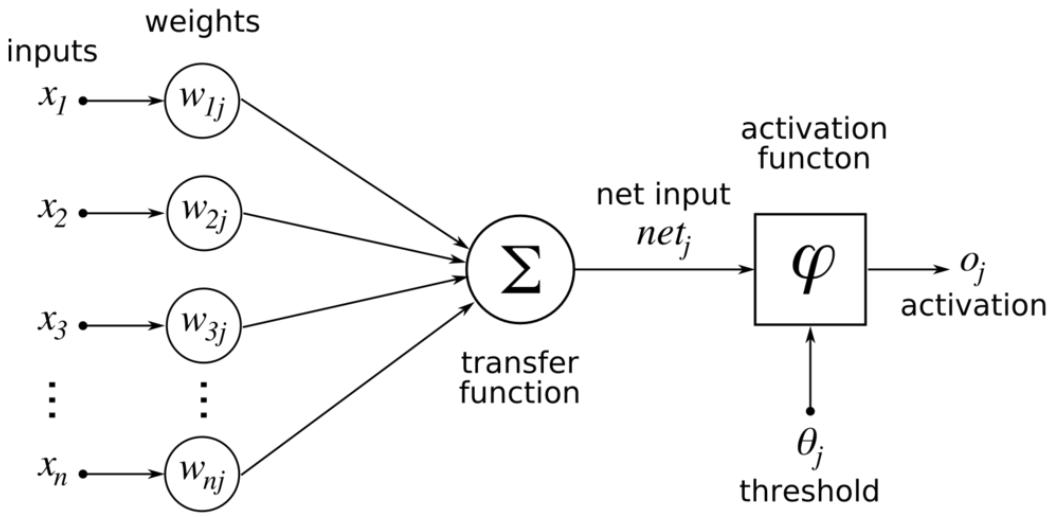
In [2]:
```
Image('Neuron.png', width=400)
```



4.2 Artificial Neuron (McCulloch & Pitts)

$$f(\mathbf{x}) = \varphi(\langle \mathbf{w}, \mathbf{x} \rangle + \theta)$$

In [3]:
```
Image('a_neuron.png', width=400)
```



4.2.1 Activation functions

Linear: $\varphi(x) = x$

Rectified Linear: $\varphi(x) = \max(0, x)$

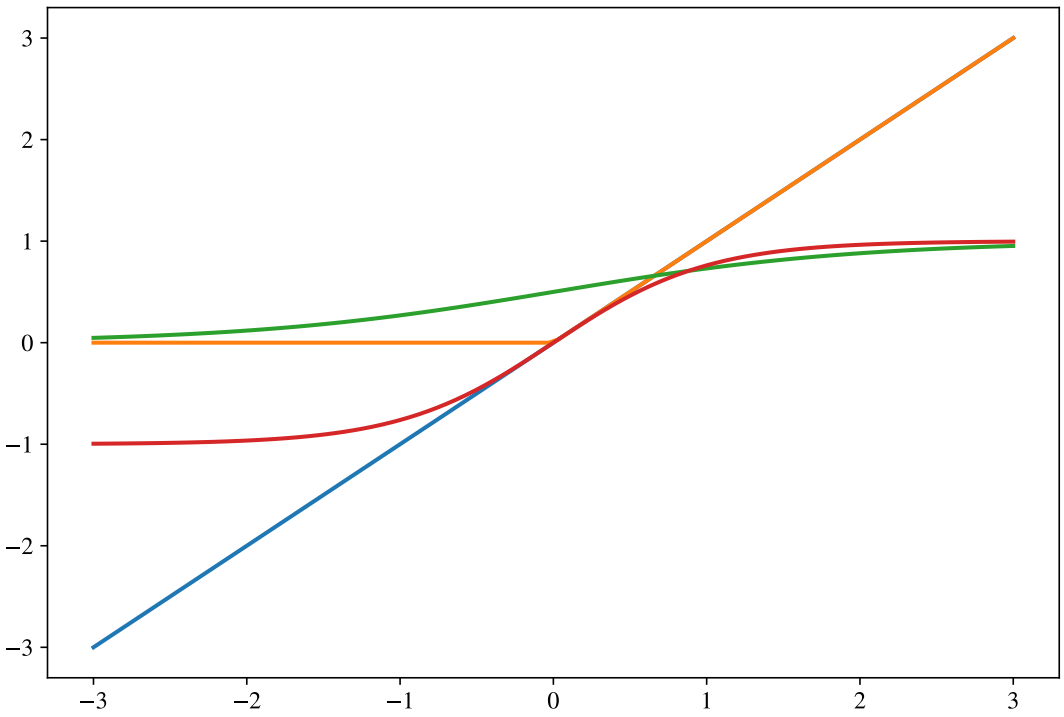
Sigmoid: $\varphi(x) = \frac{1}{1+e^{-x}}$

Hyperbolic tangent $\varphi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

In [4]:

```
x = jnp.linspace(-3, 3, 100)
plt.plot(x, x, label='linear')
plt.plot(x, jax.nn.relu(x), label='relu')
plt.plot(x, jax.nn.sigmoid(x), label='sigmoid')
plt.plot(x, jnp.tanh(x), label='tanh')
```

```
[<matplotlib.lines.Line2D at 0x77e488785f60>]
```



4.2.2 Training

Stochastic gradient descent on mini-batches from $\mathcal{A} = \{(\mathbf{x}, y)\}$, with loss function l

1. Draw random samples batch $B = \{(\mathbf{x}_i, y_i)\} \subset \mathcal{A}$
2. Compute gradient estimator

$$\delta = \frac{1}{|B|} \sum_{(\mathbf{x}_i, y_i) \in B} \frac{\partial l(y_i, f(\mathbf{x}_i)}{\partial \mathbf{w}}$$

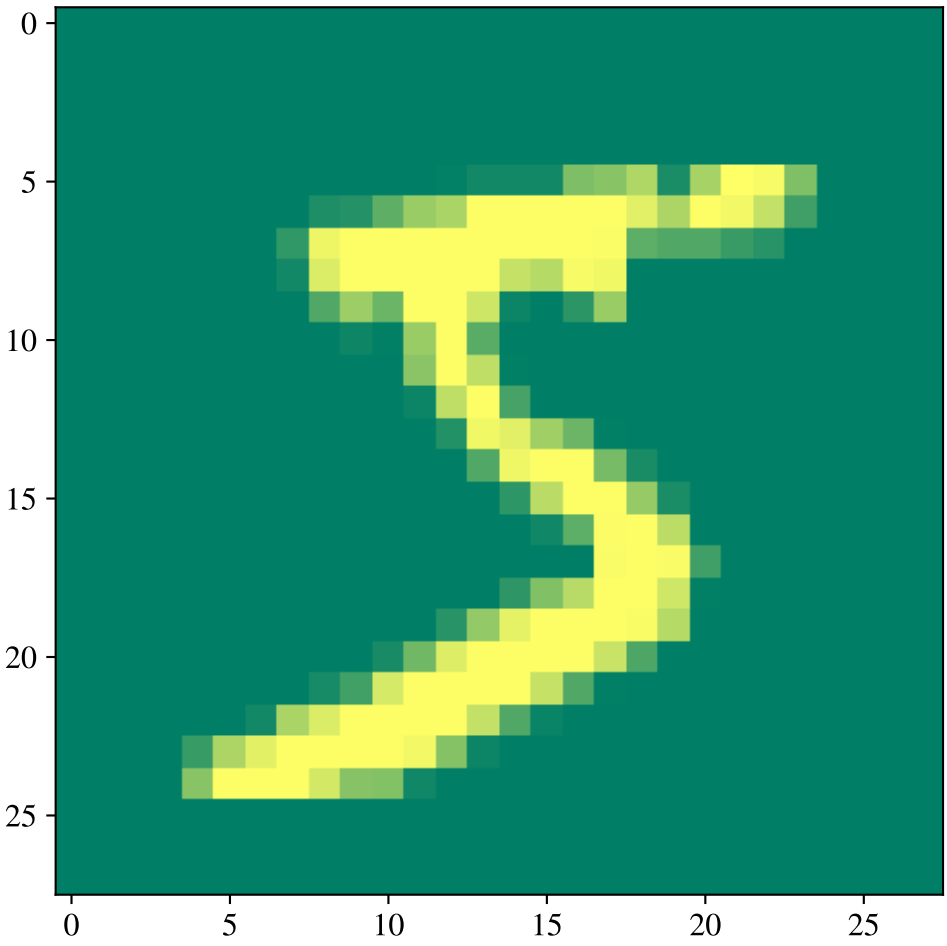
3. Apply gradient descent $\mathbf{w} \leftarrow \mathbf{w} - \eta\delta$

4.2.3 Small example

In [5]:

```
# Load the dataset
data = np.load('mnist.npz')
X = data['X_train']
y = data['y_train']
plt.imshow(X[0,:].reshape(28,28))
print(y[0])
```

```
5
```



Binary cross-entropy loss

Using sigmoid activation, $f(\mathbf{x})$ is a probability

Minimize the cross-entropy (push output to either 0 or 1)

$$\mathcal{L} = -y \log f(\mathbf{x}) - (1-y) \log(1 - f(\mathbf{x}))$$

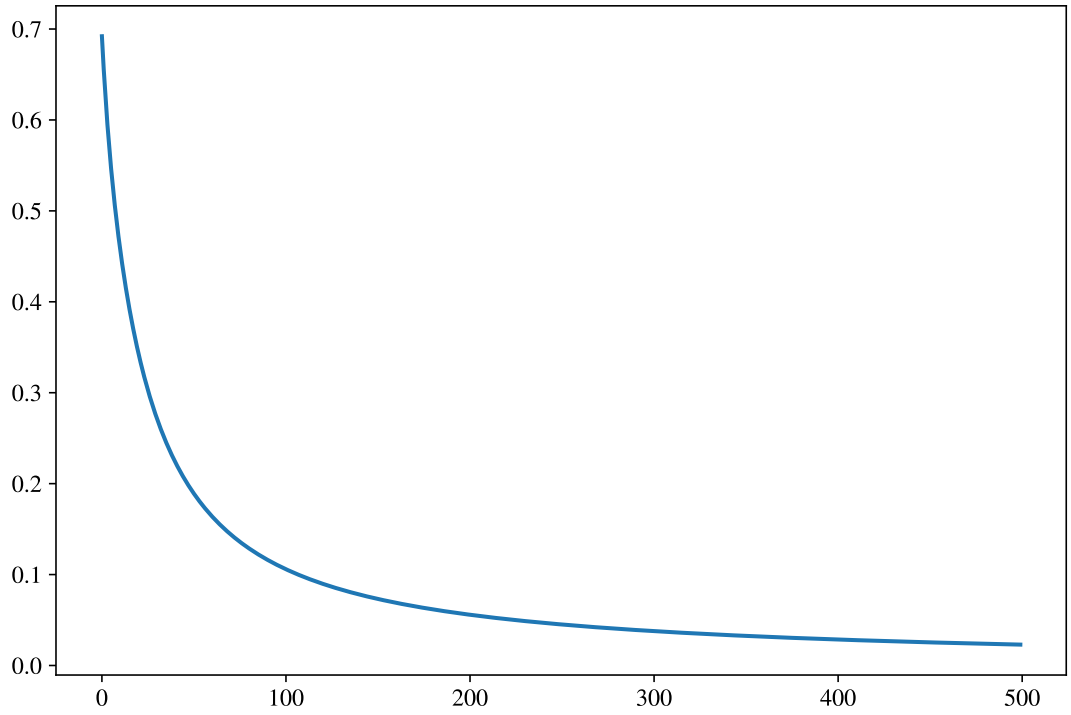
In [6]:

```
X = data['X_train_bin']
y = data['y_train_bin']

def func(w, b, x):
    return jax.nn.sigmoid(jnp.matmul(x, w) + b)

def xe(w, b, x, y):
    fx = func(w, b, x)
    return (-y*jnp.log(fx)-(1-y)*jnp.log(1-fx)).mean()

@jax.jit
def update(w, b, x, y):
    dw, db = jax.grad(xe, argnums=(0,1))(w, b, x, y)
    return w - 0.01*dw, b - 0.01*db
```

In [7]:

```
w = np.random.randn(784)/784
b = 0.

loss = []
for t in range(500):
    loss.append(xe(w, b, X, y))
    w, b = update(w, b, X, y)
plt.plot(loss)
```

```
[<matplotlib.lines.Line2D at 0x77e4885c63e0>]
```



In [8]:

```
def accuracy(y_pred, y_true):
    return (y_true==y_pred).mean()

y_pred = (func(w, b, X)>0.5)*1.
print('accuracy: {}'.format(accuracy(y_pred, y)))
```

```
accuracy: 1.0
```

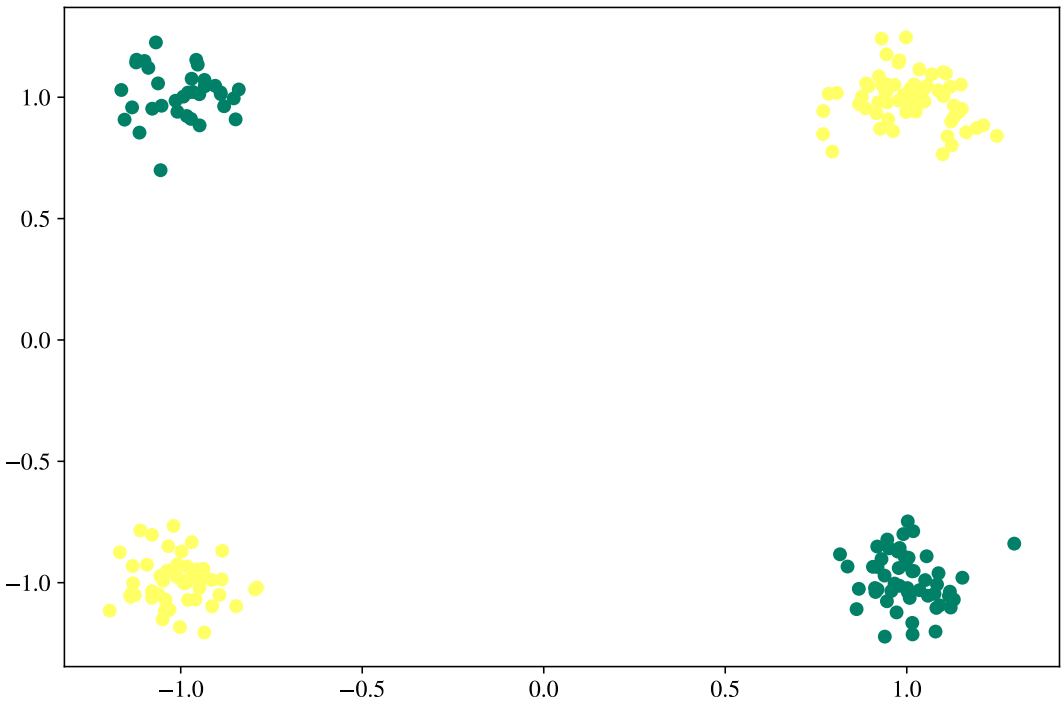
Non linearly separable problems

What about XOR?

In [9]:

```
Xor = jnp.sign(np.random.randn(200, 2)) + 0.1*np.random.randn(200,2)
yor = 1.*((Xor[:,0]*Xor[:,1])>0)

plt.scatter(Xor[:,0], Xor[:,1], c=yor)
```

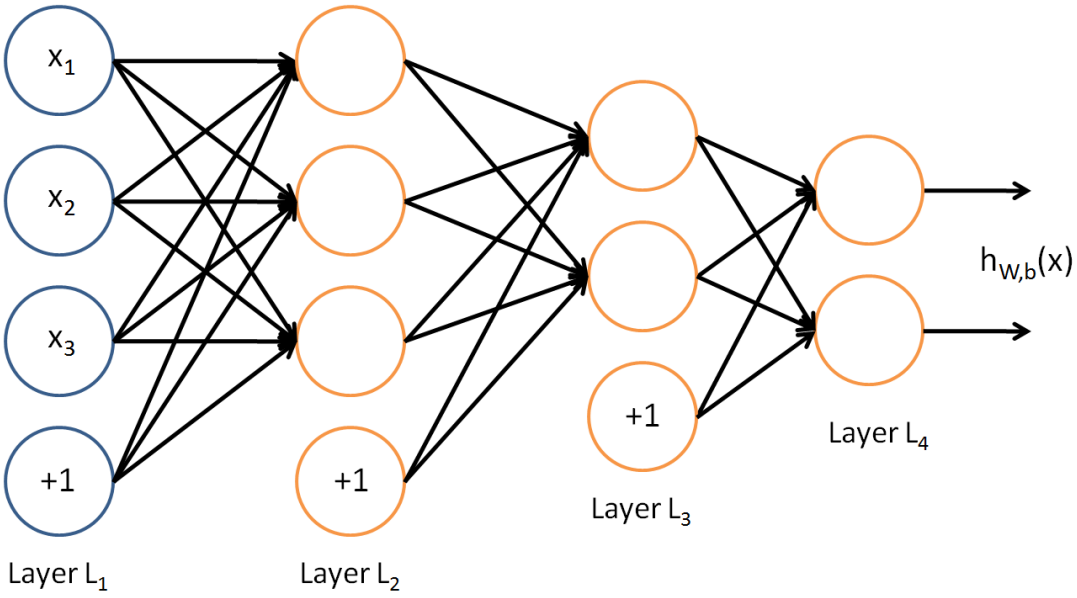
```
<matplotlib.collections.PathCollection at 0x77e48848b280>
```



Multiple layer

In [10]:

```
Image('mlp.png', width=400)
```



4.3 Multiple Layer Perceptron

Set layer i as the function that maps to $\mathbb{R}^d$ by stacking neurons

$$f_i(\mathbf{x}) = [\sigma(\mathbf{W}_{ij}^\top \mathbf{x} + \theta_{ij})]_{j \leq d}$$

With - $\mathbf{W}_{ij}, \theta_{ij}$ the weights and bias of neuron j at layer i - σ the activation function
Create a network by composing L layers

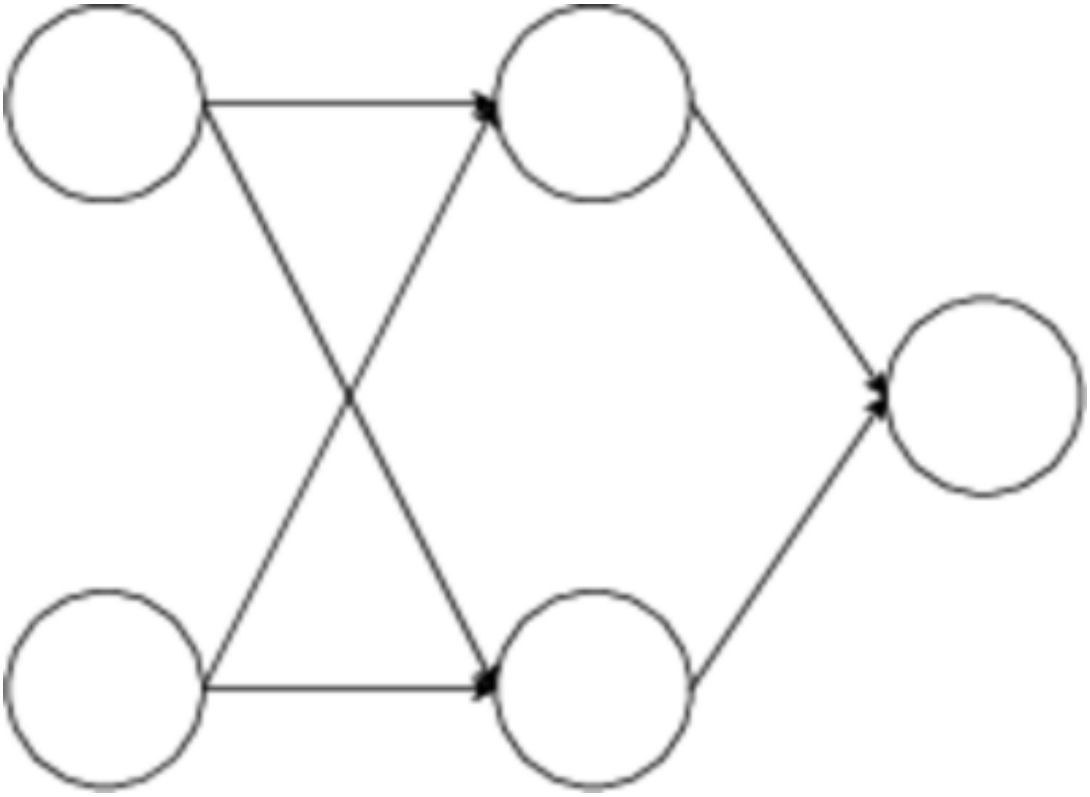
$$F(\mathbf{x}) = f_L \circ \cdots \circ f_1(\mathbf{x})$$

4.3.1 XOR - Exercise

Find all weights for 1 hidden layer of width 2
Use Relu activation for the hidden layer and sign for the output layer

In [11]:

```
Image('2_xor.png', width=400)
```



In [ ]:

```
```

4.3.2 Training

ERM principle

$$\min_{\{\mathbf{w}_i\}_i} \mathbb{E}_{(\mathbf{x},y)} \left[l(y, F(\mathbf{x})) \right]$$

Gradient descent

$$\forall i, \mathbf{w}_i \leftarrow \mathbf{w}_i - \eta \mathbb{E}_{(\mathbf{x},y)} \left[\frac{\partial l(y, F(\mathbf{x}))}{\partial \mathbf{w}_i} \right]$$

Monte-Carlo estimation with mini-batch strategy

$$\mathbb{E}_{(\mathbf{x},y)} \left[\frac{\partial l(y, F(\mathbf{x}))}{\partial \mathbf{w}_i} \right] \approx \frac{1}{N} \sum_n \frac{\partial l(y_n, F(\mathbf{x}_n))}{\partial \mathbf{w}_i}$$

4.3.3 Backpropagation

- Denote $\mathbf{x}_k = f_k \circ \cdots \circ f_1(\mathbf{x})$ the k-th intermediate output
- Denote $g_k(\mathbf{x}_k) = f_L \circ \cdots \circ f_{k+1}(\mathbf{x}_k)$ the output computed from $\mathbf{x}_k$
- Remark $\forall k, F_k = g_k(\sigma(\mathbf{w}_k^\top \mathbf{x}_{k-1} + \theta_k))$

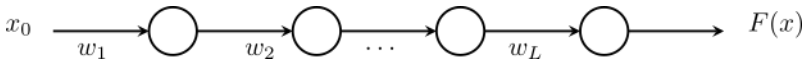
Chain rule (Leibnitz notation)

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z} \frac{\partial z}{\partial x}$$

4.3.4 Backpropagation

Single neuron chain

In [12]:
```
Image('neural_chain.png', width=600)
```



$$\frac{\partial l(y, F(\mathbf{x}))}{\partial w_k} = \frac{\partial l(y, F(\mathbf{x}))}{\partial F(\mathbf{x})} \frac{\partial F(\mathbf{x})}{\partial w_k}$$
$$= l'(y, F(\mathbf{x})) \frac{\partial \sigma(w_L \mathbf{x}_{L-1} + \theta_L)}{\partial w_k}$$
$$= l'(y, F(\mathbf{x})) \sigma'(\mathbf{x}_L) \frac{\partial w_L \mathbf{x}_{L-1} + \theta_L}{\partial w_k}$$
$$= l'(y, F(\mathbf{x})) \sigma'(\mathbf{x}_L) w_L \frac{\partial \mathbf{x}_{L-1}}{\partial w_k}$$
$$\frac{\partial \mathbf{x}_{L-1}}{\partial w_k} = \frac{\partial \sigma(w_{L-1} \mathbf{x}_{L-2} + \theta_{L-1})}{\partial w_k}$$
$$= \sigma'(\mathbf{x}_{L-1}) w_{L-1} \frac{\partial \mathbf{x}_{L-2}}{\partial w_k}$$

Recursion

$$\frac{\partial \mathbf{x}_{k+t+1}}{\partial w_k} = \sigma'(\mathbf{x}_{k+t+1}) w_{k+t+1} \frac{\partial \mathbf{x}_{k+t}}{\partial w_k}$$
$$\frac{\partial \mathbf{x}_k}{\partial w_k} = \sigma'(\mathbf{x}_k) \mathbf{x}_k$$

$$\frac{\partial l(y, F(\mathbf{x}))}{\partial w_k} = l'(y, F(\mathbf{x})) \prod_{t=k+1}^{L} \sigma'(\mathbf{x}_t) w_t \sigma'(\mathbf{x}_k) \mathbf{x}_k$$

Note:

- if $w_t \ll 1$: vanishing gradients
- if $w_t \gg 1$: exploding gradients
- if $\sigma'(\cdot) \approx 0$: vanishing gradient (sigmoid, tanh, but not relu)

Fully connected network

Recursion

$$\begin{aligned} \delta_L &= l(y, F(\mathbf{x}))\sigma'(\mathbf{x}_L) \\ \delta_k &= \mathbf{w}_{k+1}(\sigma'(\mathbf{x}_{k+1}) \circ \delta_{k+1}) \\ \frac{\partial l(y, F(\mathbf{x}))}{\partial \mathbf{w}_k} &= \delta_k \circ \mathbf{x}_k \end{aligned}$$

4.3.5 Algorithm

Forward pass

- Compute and store $\forall k, \mathbf{x}_k$

Backward pass

- Compute $l'(y, F(\mathbf{x}))$
- $\forall k$, compute δ_k
- Update $\mathbf{w}_k$ using $l'(y, F(\mathbf{x})), \delta_k, \mathbf{x}_k$

In practice, ML libraries have auto-grad features (pytorch, tensorflow, jax, etc) for certain operators that you compose to build F

4.3.6 XOR with MLP

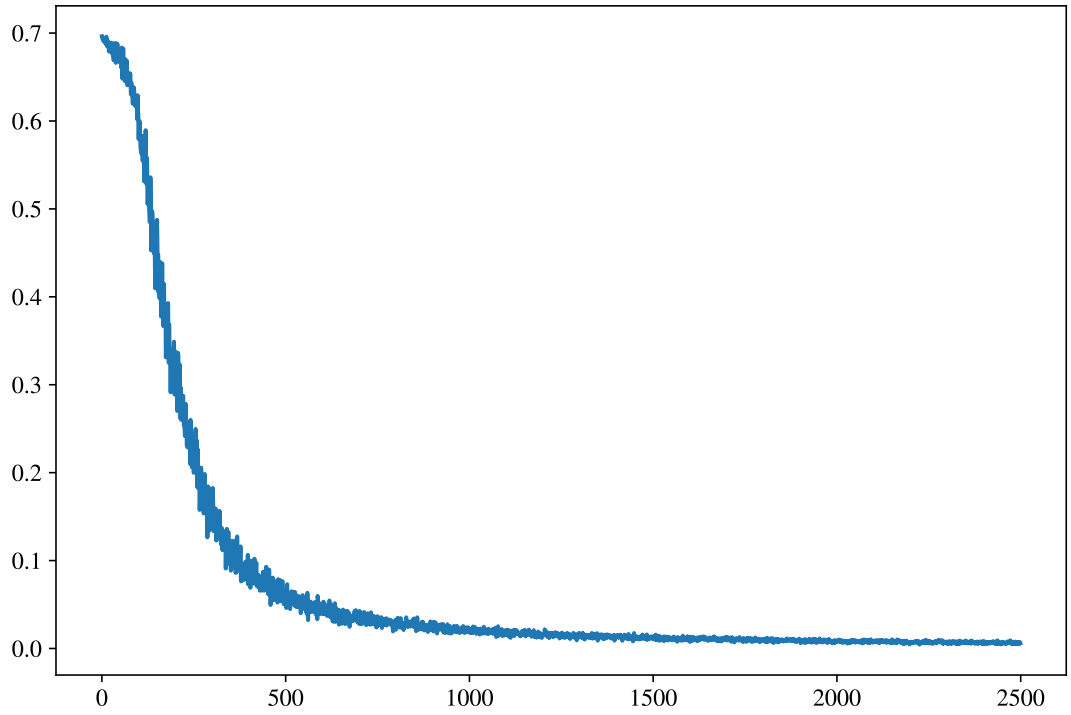
In [13]:

```
def l1(w1, b1, x):
    return jax.nn.relu(jnp.matmul(x, w1) + b1)

def func(w1, w2, b1, b2, x):
    x1 = l1(w1, b1, x)
    x2 = jax.nn.sigmoid(jnp.matmul(x1, w2) + b2)
    return x2

def xe(w1, w2, b1, b2, x, y):
    fx = func(w1, w2, b1, b2, x)
    return (-y*jnp.log(fx)-(1-y)*jnp.log(1-fx)).mean()

@jax.jit
def update(w1, w2, b1, b2, x, y, eta=0.1):
    dw1, dw2, db1, db2 = jax.grad(xe, argnums=(0,1,2,3))(w1, w2, b1, b2, x, y)
    return w1 - eta*dw1, w2 - eta*dw2, b1 - eta*db1, b2 - eta*db2
```

In [14]:

```
w1 = np.random.randn(2, 4)/10
w2 = np.random.randn(4)/10
b1 = np.random.randn(4)/10
b2 = np.random.randn(1)/10

loss = []
for t in range(2500):
    ind = np.random.choice(len(Xor), size=64, replace=False)
    loss.append(xe(w1, w2, b1, b2, Xor[ind, :], yor[ind]))
    w1, w2, b1, b2 = update(w1, w2, b1, b2, Xor[ind, :], yor[ind], eta=0.1)
plt.plot(loss)
```

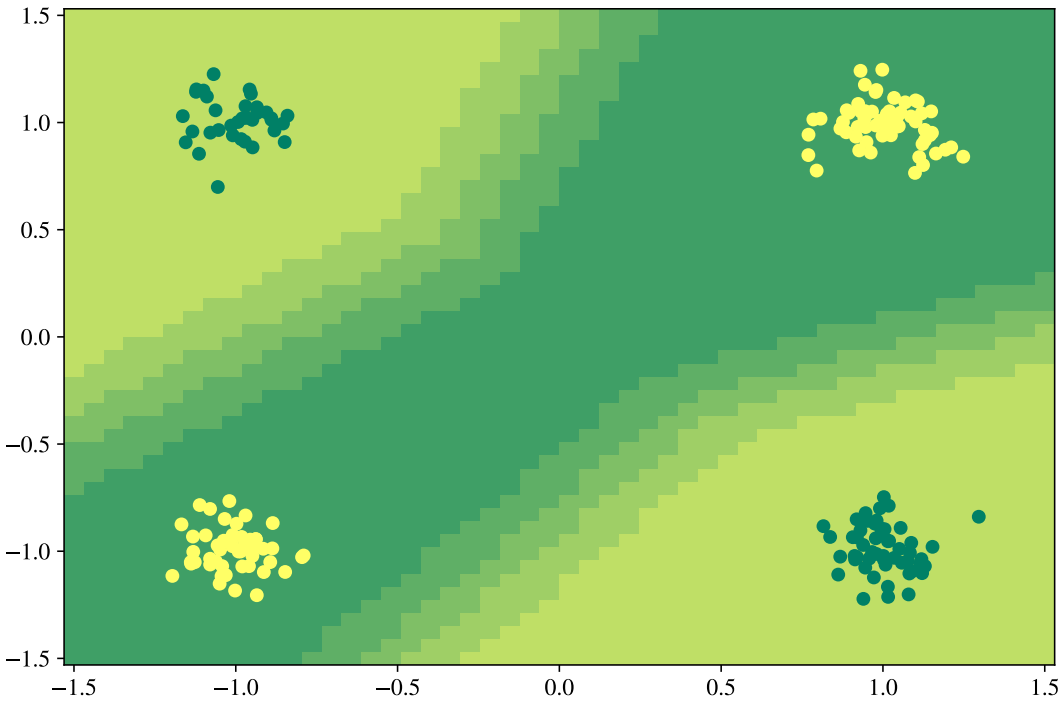
```
[<matplotlib.lines.Line2D at 0x77e46cd9bb20>]
```



In [15]:

```
t = 50; tx = jnp.linspace(-1.5, 1.5, t)
xv, yv = jnp.meshgrid(tx, tx, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(func(w1, w2, b1, b2, xx)).reshape(t, t)
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .5, 10)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred, shading='nearest', norm=norm);
plt.scatter(Xor[:,0], Xor[:,1], c=yor)
```

```
<matplotlib.collections.PathCollection at 0x77e46cc3fd90>
```

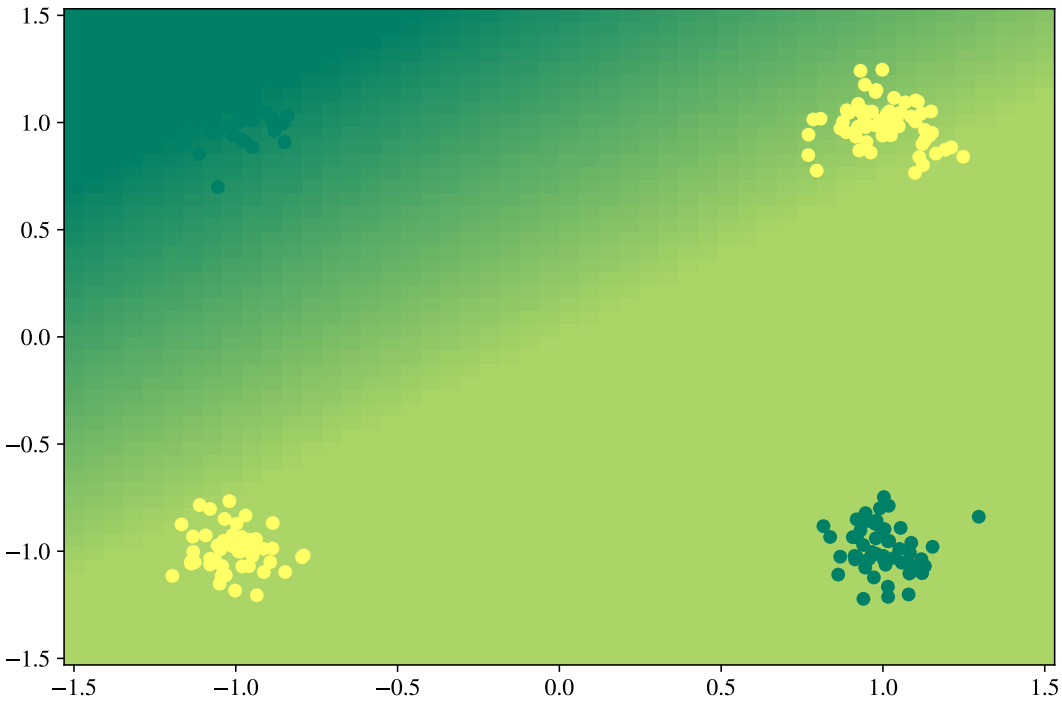


Intermediate layers

In [16]:

```
t = 50; tx = jnp.linspace(-1.5, 1.5, t);
xv, yv = jnp.meshgrid(tx, tx, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(l1(w1, b1, xx)).reshape(t, t, 4)
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-4., 2., 100)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred[:,:,0], shading='nearest', norm=norm);
plt.scatter(Xor[:,0], Xor[:,1], c=yor)
```

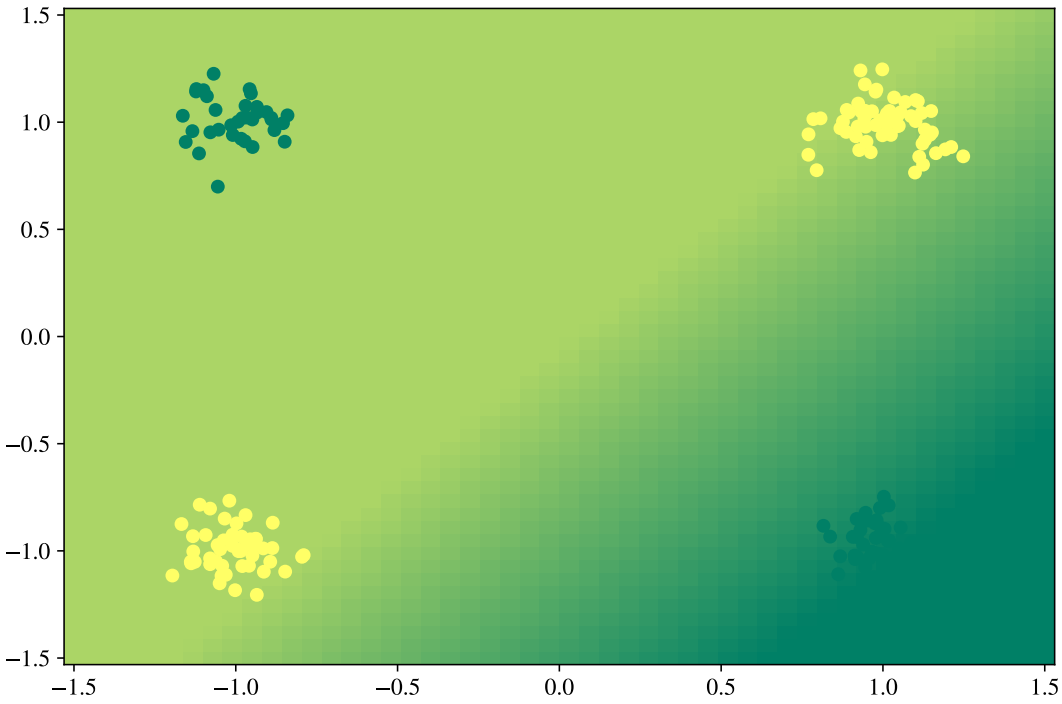
```
<matplotlib.collections.PathCollection at 0x77e4883af400>
```



In [17]:

```
levels=jnp.linspace(-4., 2., 100)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred[:,:,1], shading='nearest', norm=norm);
plt.scatter(Xor[:,0], Xor[:,1], c=yor)
```

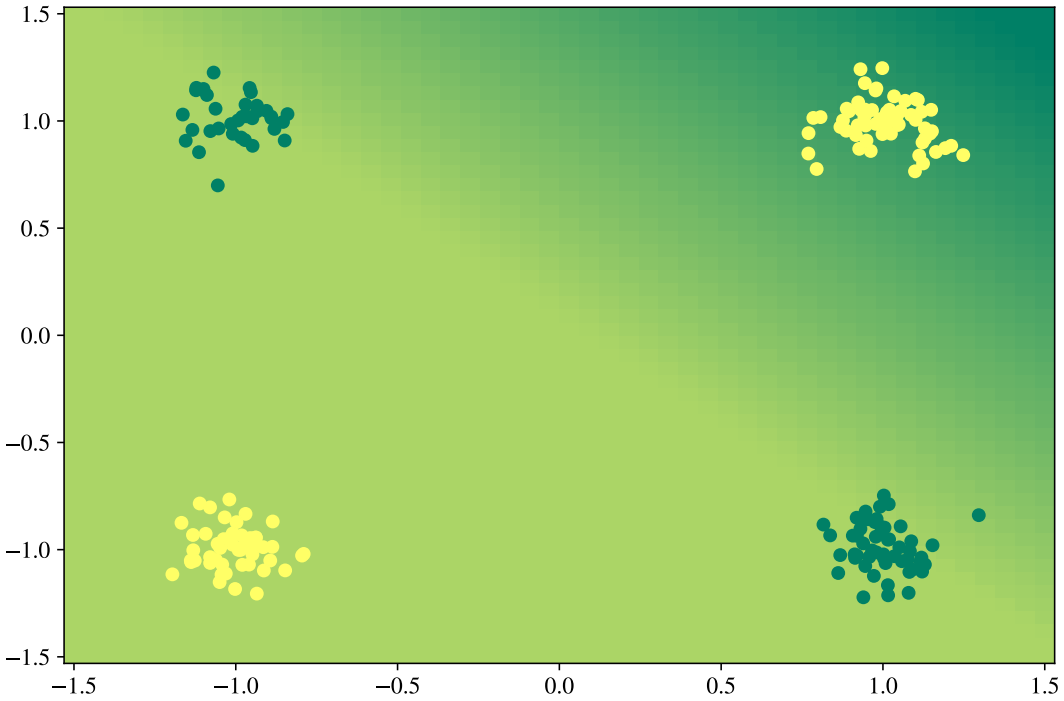
```
<matplotlib.collections.PathCollection at 0x77e46ca05c30>
```



In [18]:

```
levels=jnp.linspace(-4., 2., 100)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred[:,:,2], shading='nearest', norm=norm);
plt.scatter(Xor[:,0], Xor[:,1], c=yor)
```

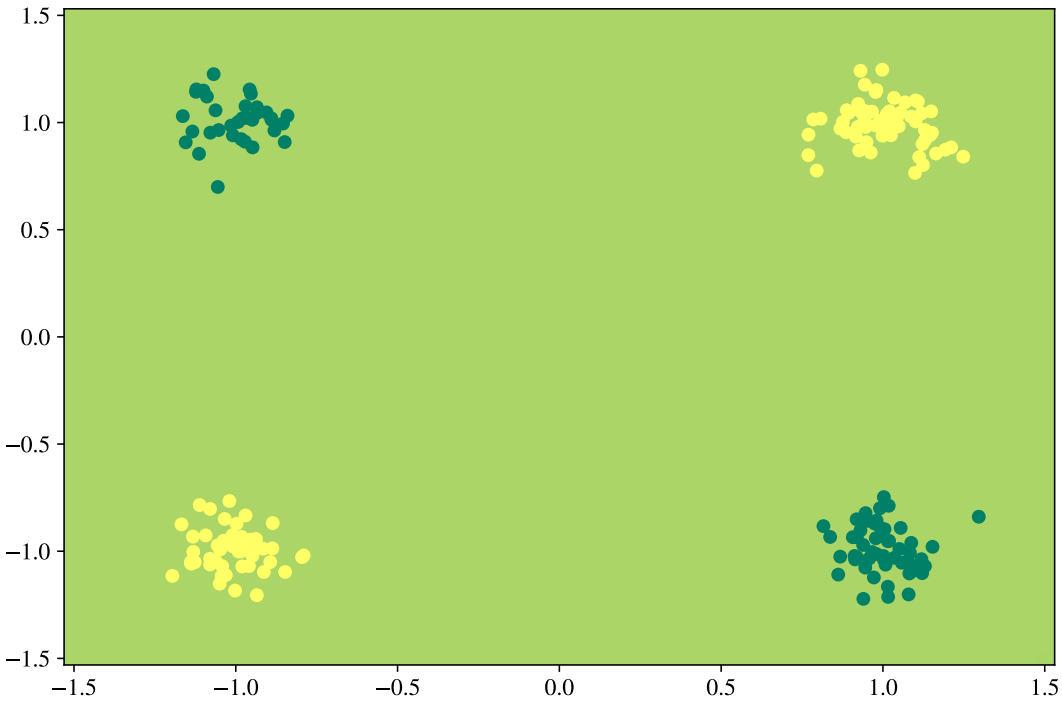
```
<matplotlib.collections.PathCollection at 0x77e46c880bb0>
```



In [19]:

```
levels=jnp.linspace(-4., 2., 100)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred[:,:,3], shading='nearest', norm=norm);
plt.scatter(Xor[:,0], Xor[:,1], c=yor)
```

```
<matplotlib.collections.PathCollection at 0x77e46c8f3940>
```



4.3.7 Neural networks losses

Classification

- Independent classes, binary crossentropy with sigmoid activation
- Exclusive classes, categorical crossentropy with softmax activation

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$
$$l(y, F(x))) = -\sum_i y_i \log F(\mathbf{x})[i]$$

Regression

- Usual ℓ_2, ℓ_1 losses

4.4 Neural networks capacity

Consider that a feed forward neural network is an acyclic directed graph (V, E)

Theorem: $\forall n$, there exists a graph V, E of depth 2, such that $\mathcal{H}(V, E, \text{sign})$ contains all function from $\{\pm 1\}^n$ to $\{\pm 1\}$. A neural network with a single hidden layer and the sign activation function can approximate any binary function over binary vectors

4.4.1 Proof

- Consider a network with a single hidden layer of 2^n neurons

- Let $\{u_i\}_{1 \leq k \leq n}$ be the set of k input vector that have label 1
- Remark that $\forall i, \langle u_i, u_i \rangle = n$ and $\forall x, \forall i, x \neq u_i \Leftrightarrow \langle x, u_i \rangle \leq n-2$ (minimum 1 bit difference)
- Set k neurons to $h_i(x) = \text{sign}(u_i^\top x - n + 1)$, we have $h_i(x) = 1$ iff $x = u_i$ and -1 else
- Set the output to

$$F(x) = \text{sign}(\sum_i h_i(x) + k - 1)$$

4.4.2 Neural network capacity

Theorem (see [Hornik et al., 1989] and [Cybenko, 1989]): $\forall n$, let $s(n)$ be the minimal integer such that there exist a graph (V, E) with $|V| = n$ such that the hypothesis class $\mathcal{H}(V, E, \text{sign})$ contains all the function to $\{0,1\}^n$ to $\{0,1\}$. Then, $s(n)$ is exponential in n. Similar results hold for the sigmoid function. 1 hidden layer MLPs can approximate any function but require an exponential number of neurons.

4.4.3 VC dimension

Theorem: The VC dimension of $\mathcal{H}(V, E, \text{sign})$ is $\mathcal{O}(|E| \log |E|)$ The capacity of neural networks is more defined by their connectivity than by their number of neurons. Deeper networks have higher capacity.

4.4.4 Effect of width

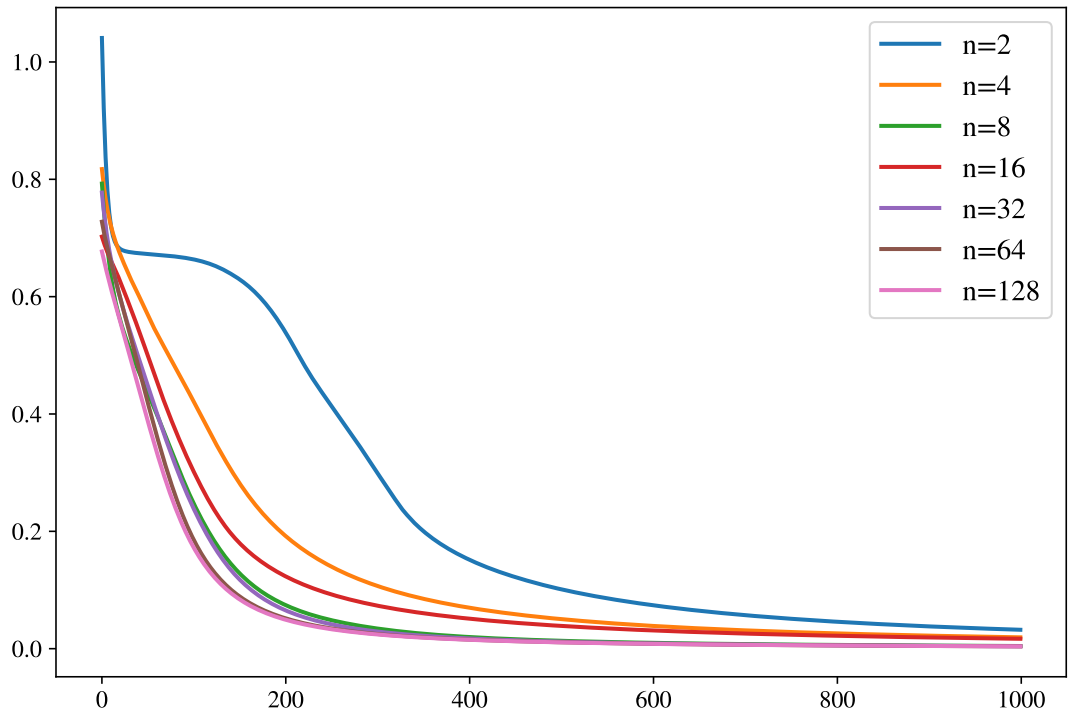
In [20]:

```
def train_nn(n):
    w1 = np.random.randn(2, n)/(jnp.sqrt(n))
    w2 = np.random.randn(n)/(jnp.sqrt(n))
    b1 = np.random.randn(n)/(jnp.sqrt(n))
    b2 = np.random.randn(1)/(jnp.sqrt(n))

    loss = []
    for t in range(1000):
        loss.append(xe(w1, w2, b1, b2, Xor, yor))
        w1, w2, b1, b2 = update(w1, w2, b1, b2, Xor, yor, eta=0.1)
    return loss
```

In [21]:

```
for n in range(1, 8):
    plt.plot(train_nn(2**n), label='n={}'.format(2**n))
plt.legend()
```

```
<matplotlib.legend.Legend at 0x77e46c4a68f0>
```



Larger NN, easier to train?

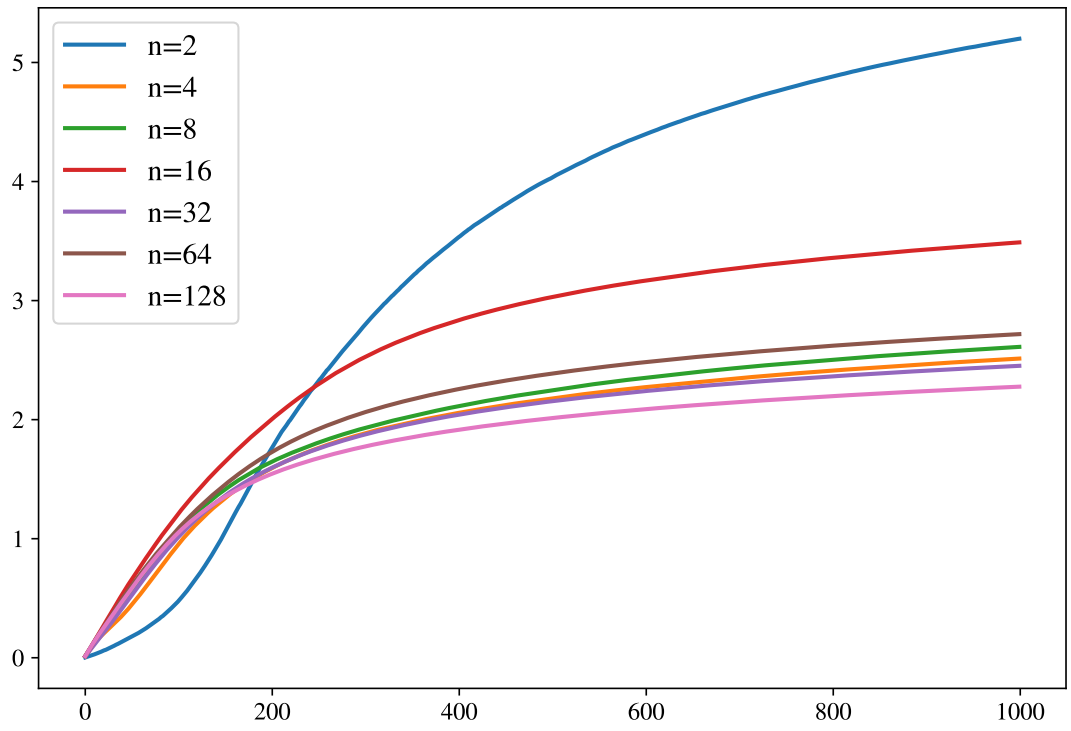
In [22]:

```
def train_nn2(n):
    w1 = np.random.randn(2, n)/(jnp.sqrt(n))
    w2 = np.random.randn(n)/(jnp.sqrt(n))
    b1 = np.random.randn(n)/(jnp.sqrt(n))
    b2 = np.random.randn(1)/(jnp.sqrt(n))

    w = w1
    w_change = []
    for t in range(1000):
        ind = np.random.choice(len(Xor), size=128, replace=False)
        w1, w2, b1, b2 = update(w1, w2, b1, b2, Xor[ind, :], yor[ind], eta=0.1)
        w_change.append(jnp.linalg.norm(w1 - w)/jnp.linalg.norm(w))
    return w_change
```

In [23]:

```
for n in range(1, 8):
    plt.plot(train_nn2(2**n), label='n={}'.format(2**n))
plt.legend()
```

```
<matplotlib.legend.Legend at 0x77e46c741bd0>
```



Larger networks tend to change less than smaller networks?

4.4.5 Neural Tangent Kernel [Jacot et al., 2018]

Consider the loss function as a function of $\mathbf{w}$ instead of x

$$l_{\mathbf{x}}(\mathbf{w}) = l(y, F_{\mathbf{w}}(\mathbf{x}))$$

Approximate it using its Taylor expansion

$$l_{\mathbf{x}}(\mathbf{w}) \approx l_{\mathbf{x}}(\mathbf{w}_0) + \nabla l_{\mathbf{x}}(\mathbf{w}_0)^\top (\mathbf{w} - \mathbf{w}_0)$$

Corresponds to a linear model with the non-linear mapping

$$\phi(\mathbf{x}) = \nabla l_{\mathbf{x}}(\mathbf{w}_0)$$

Defining a kernel This approximation tends to be better when the width grows
Some weights are highly influencial ($|\sum_i \phi(\mathbf{x}_i)[j]| \gg 0$)
Blind spots in the kernel ($\sum_i \phi(\mathbf{x}_i)[j] \approx 0$) mean that some components are barely updated
Very large model are easier to train because only few neurons need to be changed to (over?)fit the training data

4.4.6 Lottery ticket hypothesis

The Lottery Ticket Hypothesis [Malach et al., 2020]

A randomly-initialized, dense neural network contains a subnet-work that is initialized such that—when trained in isolation—it can match the test accuracy of theoriginal network after training for at most the same number of iterations.

- Pruning a randomly initialized network can yield a good predictor (verified empirically).
- In practice, pruning is difficult, better train the full model.

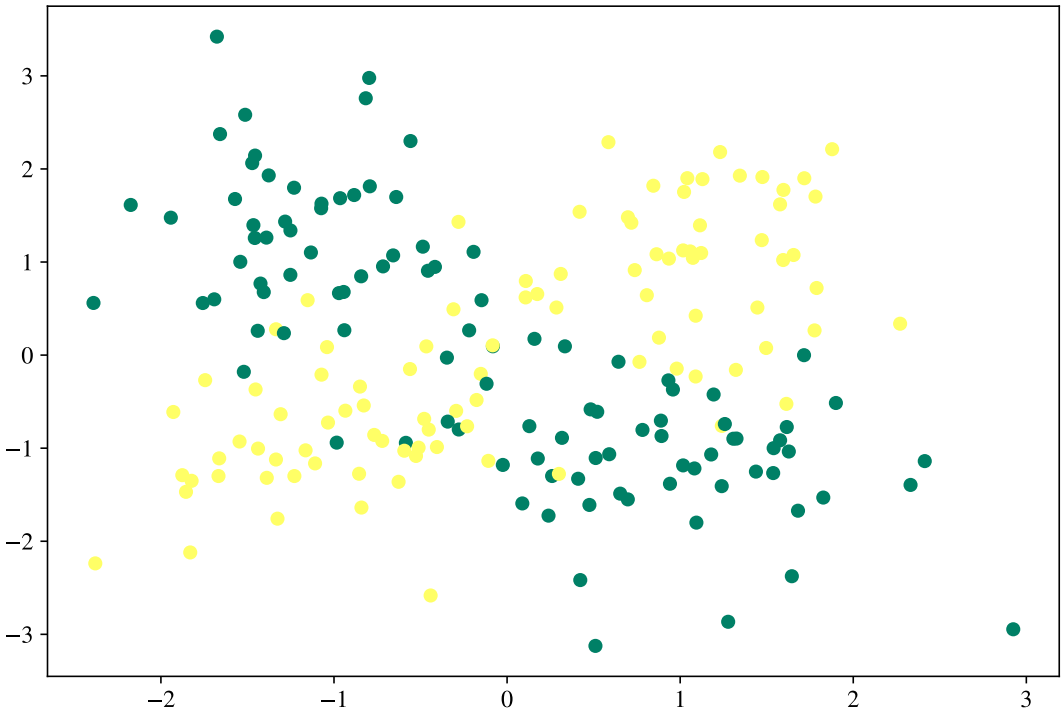
4.4.7 Double Descent [Belkin et al., 2019]

Do larger networks systematically overfit?

In [24]:

```
Xor_tr = jnp.sign(np.random.randn(200, 2))
yor_tr = 1.*((Xor_tr[:,0]*Xor_tr[:,1])>0)
Xor_tr += 0.7*np.random.randn(200,2)
Xor_te = jnp.sign(np.random.randn(200, 2))
yor_te = 1.*((Xor_te[:,0]*Xor_te[:,1])>0)
Xor_te += 0.7*np.random.randn(200,2)

plt.scatter(Xor_tr[:,0], Xor_tr[:,1], c=yor_tr)
```

```
<matplotlib.collections.PathCollection at 0x77e46c04ed40>
```



In [25]:

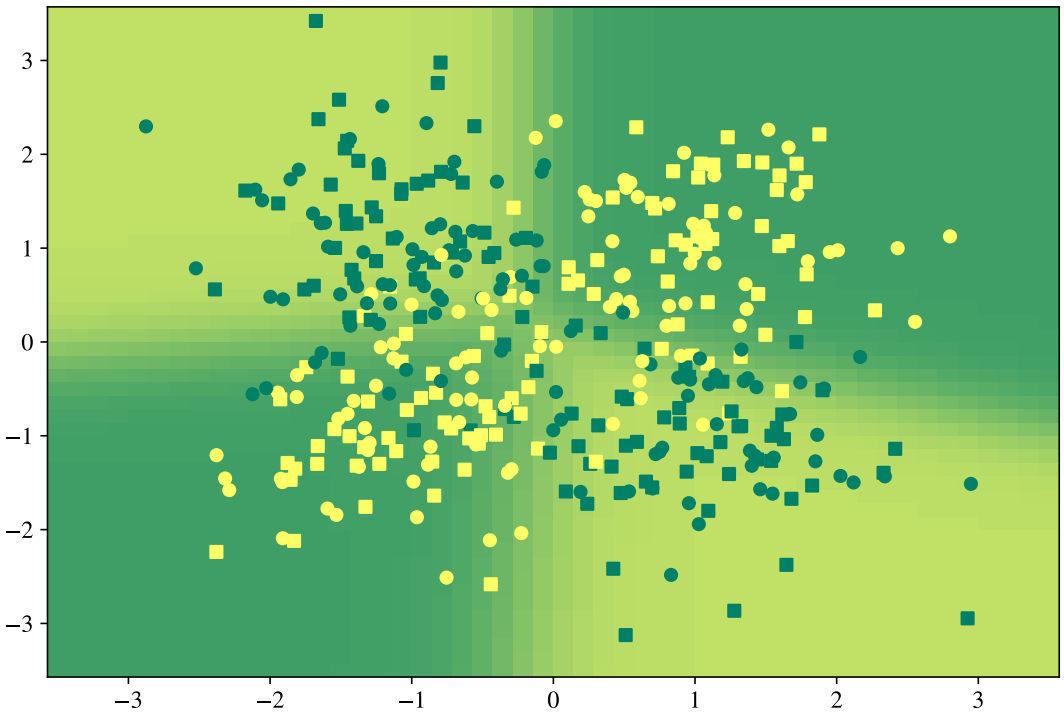
```
n = 10000
w1 = np.random.randn(2, n)/(jnp.sqrt(n))
w2 = np.random.randn(n)/(jnp.sqrt(n))
b1 = np.random.randn(n)/(jnp.sqrt(n))
b2 = np.random.randn(1)/(jnp.sqrt(n))

loss = 0
for t in range(1000):
    loss = xe(w1, w2, b1, b2, Xor_te, yor_te)
    w1, w2, b1, b2 = update(w1, w2, b1, b2, Xor_tr, yor_tr, eta=0.1)
```

In [26]:

```
t = 50; tx = jnp.linspace(-3.5, 3.5, t)
xv, yv = jnp.meshgrid(tx, tx, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(func(w1, w2, b1, b2, xx)).reshape(t, t)
```

```
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .5, 100)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred, shading='nearest', norm=norm);
plt.scatter(Xor_tr[:,0], Xor_tr[:,1], c=yor_tr, marker='s')
plt.scatter(Xor_te[:,0], Xor_te[:,1], c=yor_te)
```

```
<matplotlib.collections.PathCollection at 0x77e46cd7ea70>
```



Double descent phenomenon

Overfitting solution exist, but are difficult to attain with SGD from well initialized network

4.5 Metric Learning

So far, we use either the *natural* distance on $\mathcal{X}$ or the one induced by the choice of a kernel Can we just *learn* the distance? Train a neural network $\phi(\cdot)$ such that

- Related samples have short distances
- Unrelated samples have larger distances

Contrastive loss

Linear model

$$\phi(\mathbf{x}) = \mathbf{P}\mathbf{x}$$

Define *Positive* and *Negative* sets $\mathcal{P}(\mathbf{x}), \mathcal{N}(\mathbf{x})$ for each example $\mathbf{x}$

$$\min_{\mathbf{P}} \sum_{\mathbf{x}} \sum_{\mathbf{x}_p \in \mathcal{P}(\mathbf{x})} \|\mathbf{P}\mathbf{x} - \mathbf{P}\mathbf{x}_p\|^2 - \lambda \sum_{\mathbf{x}_n \in \mathcal{N}(\mathbf{x})} \|\mathbf{P}\mathbf{x} - \mathbf{P}\mathbf{x}_n\|^2$$

In practice, we don't want to put negative examples at an infinite distance

$$\min_{\mathbf{P}} \sum_{\mathbf{x}} \sum_{\mathbf{x}_p \in \mathcal{P}(\mathbf{x})} \|\mathbf{P}\mathbf{x} - \mathbf{P}\mathbf{x}_p\|^2 + \lambda \sum_{\mathbf{x}_p \in \mathcal{N}(\mathbf{x})} \max(0, \beta - \|\mathbf{P}\mathbf{x} - \mathbf{P}\mathbf{x}_n\|^2)$$

Push negative example until they are above margin β Similar argument for the *positive* set with margin

$$\max(0, \|\mathbf{P}\mathbf{x} - \mathbf{P}\mathbf{x}_p\|^2 - \alpha)$$

4.5.1 Large Margin Nearest Neighbor [Weinberger and Saul, 2009]

Learn a distance that *enhances* a nearest neighbor classifier

- Define *positives* as elements of the k nearest neighbors with the same label as $\mathbf{x}$

$$\mathcal{P}(\mathbf{x}) = \{\mathbf{x}_c \in k\text{NN}(\mathbf{x}) | y_c = y\}$$

- Define *negatives* as elements of the k nearest neighbors with different labels as $\mathbf{x}$

$$\mathcal{N}(\mathbf{x}) = \{\mathbf{x}_c \in k\text{NN}(\mathbf{x}) | y_c \neq y\}$$

4.6 Neural Networks , take home

- Artificial neuron: linear combination with pointwise non-linearity
- Layer: stacked neurons
- MLP: Layer composition

Training

- Backpropagation: Automatic differenciation
- Stochastic gradient descent with mini-batch
- Vanishing/exploding gradient (saturating non-linearity)
- Non-convex optimization problem, but very effective in practice

Capacity

- Universal approximation theorem
- Capacity depending on connectivity (rather than size)

NTK

- Well initialized large networks tend to behave linearly during optimization
- Some neurons will not be updated
- $\exists$ a good subnetwork in the random initialization - Lottery ticket

Double Descent

- Extremely large models can avoid overfitting
- Double descent phenomenon (overfitting difficult to reach with SGD from good init)

Chapter 5

Decision Trees and ensembling methods

5.1 Region based classification

1. Memorizing all regions is cumbersome
   - R 1 = {(a 1 , b 1 , c 1 , d 1), y 1 }
   - R 2 = {(a 2 , b 2 , c 2 , d 2), y 2 }
   - ...
2. Classification requires to check all regions

```
for region Ri in all regions:
  if x ∈ Ri:
    return yi
```

5.1.1 Tree based equivalent

5.1.2 Tree representation

5.1.3 Decision Tree

A decision tree is a hierarchical classifier with a tree structure where each node partitions the feature space along a specified component [Breiman et al., 1984].

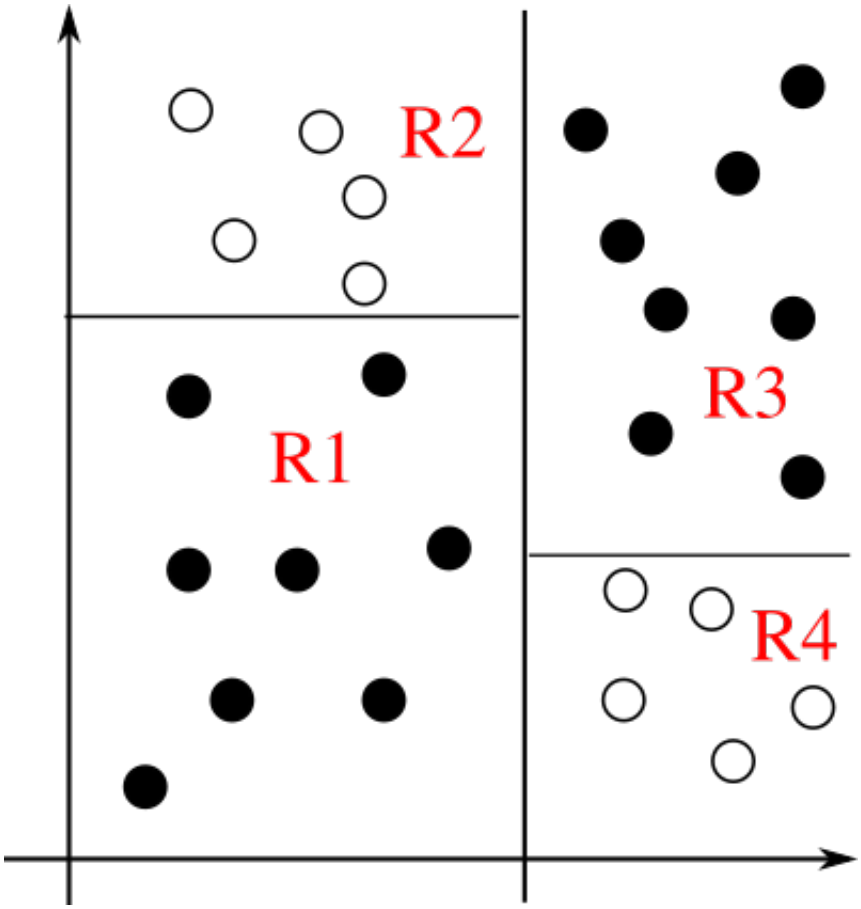
Leo Breiman (1928 - 2005)

5.1.4 Growing the tree

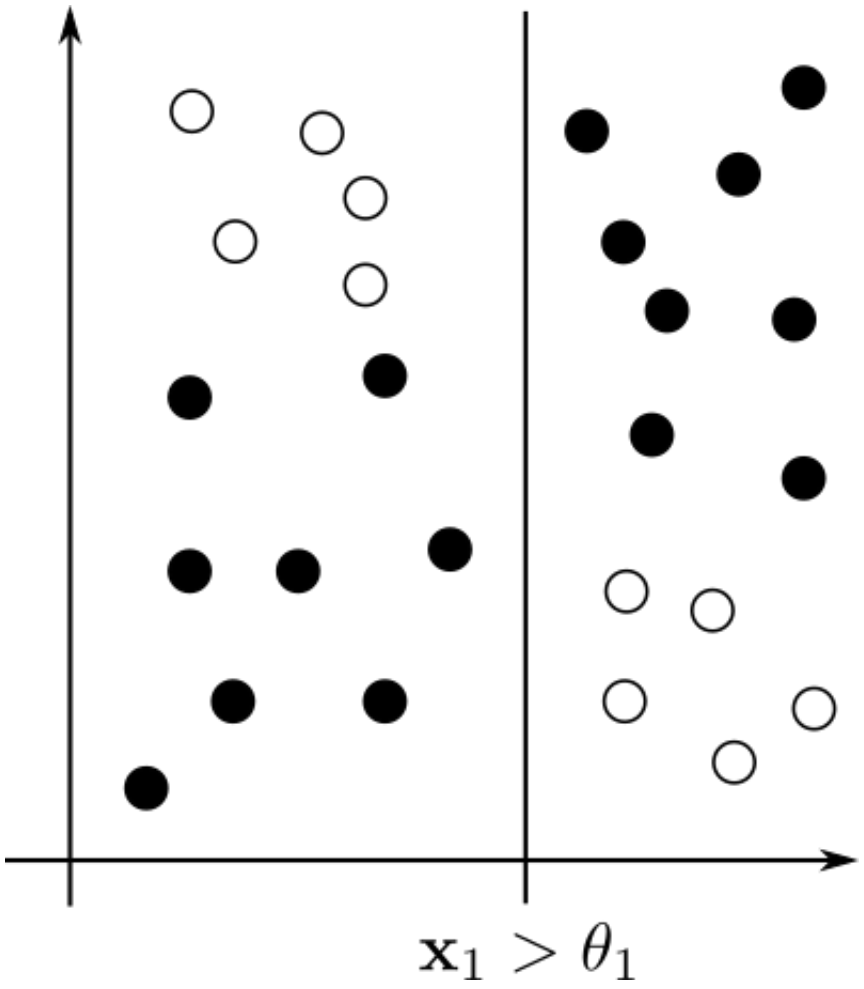
1. Tree($\mathcal{S} = \{\mathbf{x}_i, y_i\}$):
2. if $|\mathcal{S}| < T$:
3. return Leaf($\text{argmax}_c \sum_i 1_{y_i=c}$)
4. $d^\star, \theta^\star = \text{argmax}_{d,\theta} \text{Gain}(\mathcal{S}, d, \theta)$
5. $T_1 = \text{Tree}(\{\mathbf{x}_i, y_i\} \in \mathcal{S} | \mathbf{x}_i[d^\star] < \theta^\star)$
6. $T_2 = \text{Tree}(\{\mathbf{x}_i, y_i\} \in \mathcal{S} | \mathbf{x}_i[d^\star] \geq \theta^\star)$
7. return Node($d^\star, \theta^\star, T_1, T_2$)

5.1.5 Gain measure

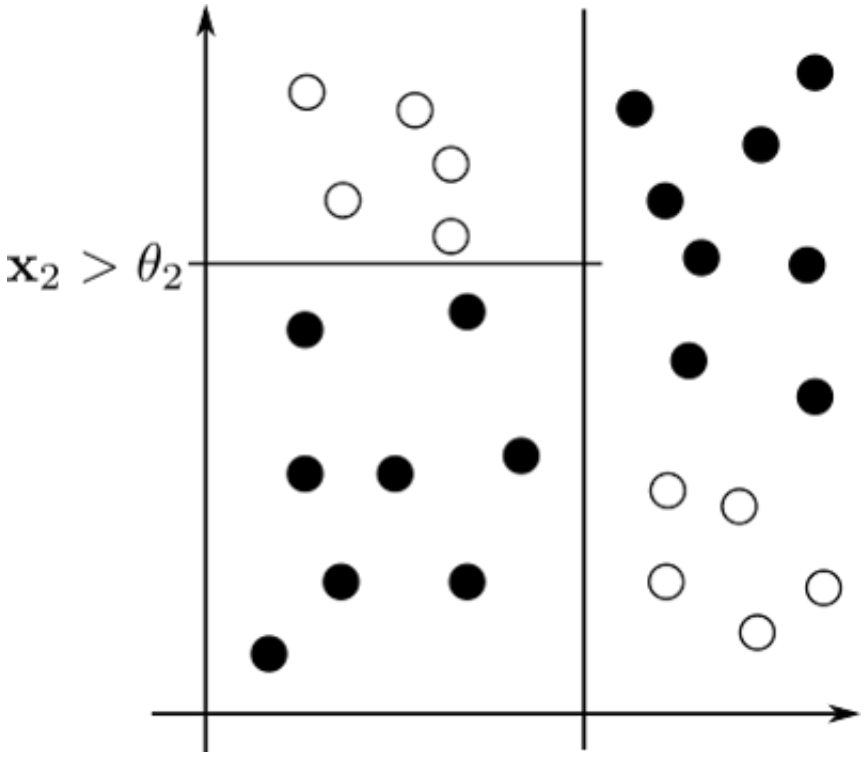
Proportion of class k in $\mathcal{S}$



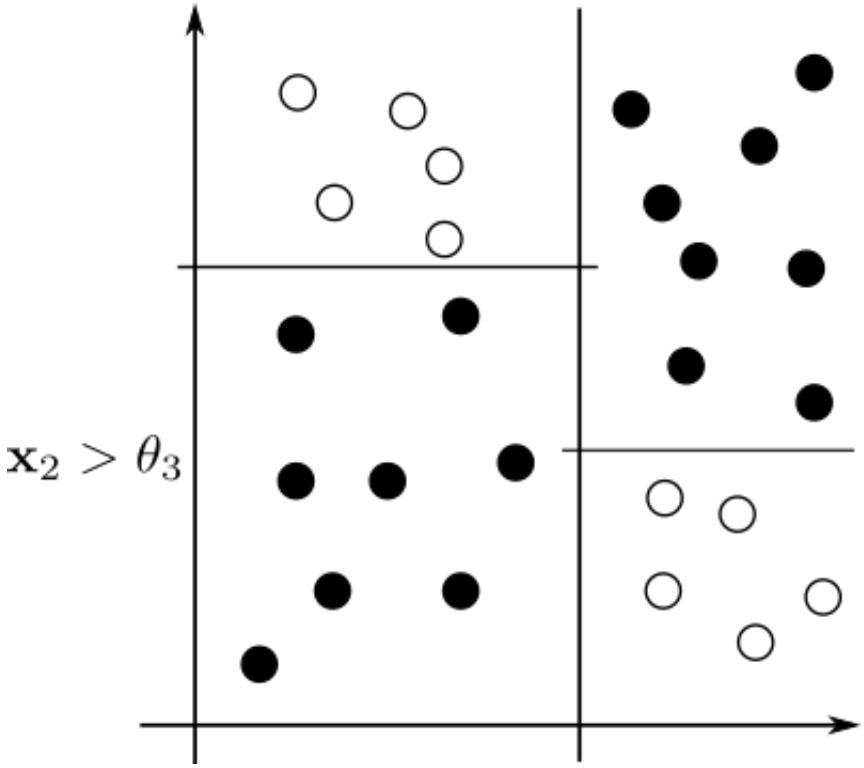
region



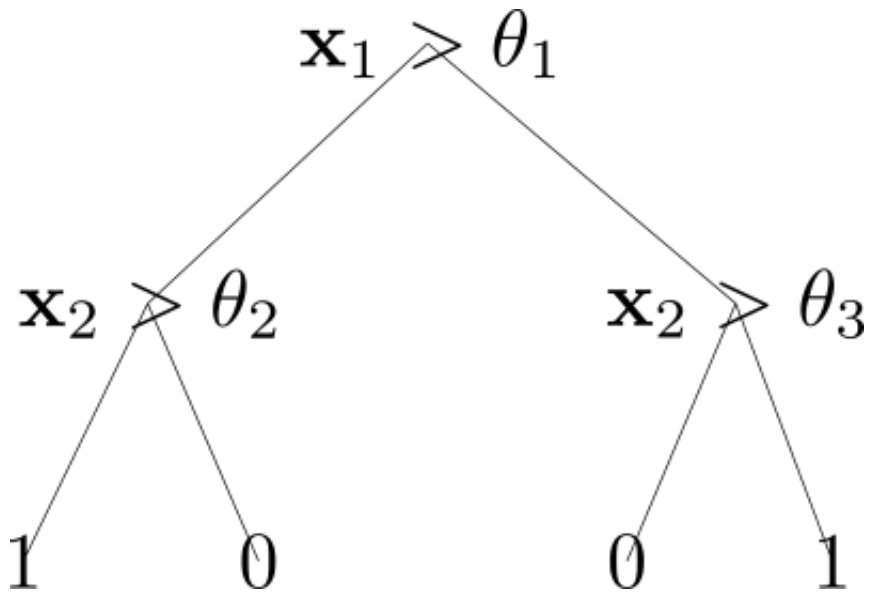
tree1



tree2



tree3



dt

leo

$$p_k(\mathcal{S}) = \frac{1}{|\mathcal{S}|} \sum_{y_i=k} 1$$

Prediction for $\mathcal{S}$

$$f(\mathcal{S}) = \text{argmax}_k p_k(\mathcal{S})$$

0-1 loss

$$C(\mathcal{S}) = \frac{1}{N} \sum_i (1 - \delta(y_i, f(\mathbf{x}_i)) = 1 - p_{f(\mathcal{S})}(\mathcal{S})$$

How much did the error decrease with the split on component d at threshold θ that leads to subsets $\mathcal{S}_1$ and $\mathcal{S}_2$:

$$\text{Gain}(\mathcal{S}, d, \theta) = C(\mathcal{S}) - \left[\frac{N_1}{N} C(\mathcal{S}_1) + \frac{N_2}{N} C(\mathcal{S}_2) \right]$$

Choose d, θ with maximal gain

5.1.6 Information Gain

Other popular gain measures:

- Entropy

$$C(\mathcal{S}) = -\sum_k p_k(\mathcal{S}) \log p_k(\mathcal{S})$$

- Gini index

$$C(\mathcal{S}) = -\sum_k p_k(\mathcal{S})(1 - p_k(\mathcal{S}))$$

5.1.7 Small example

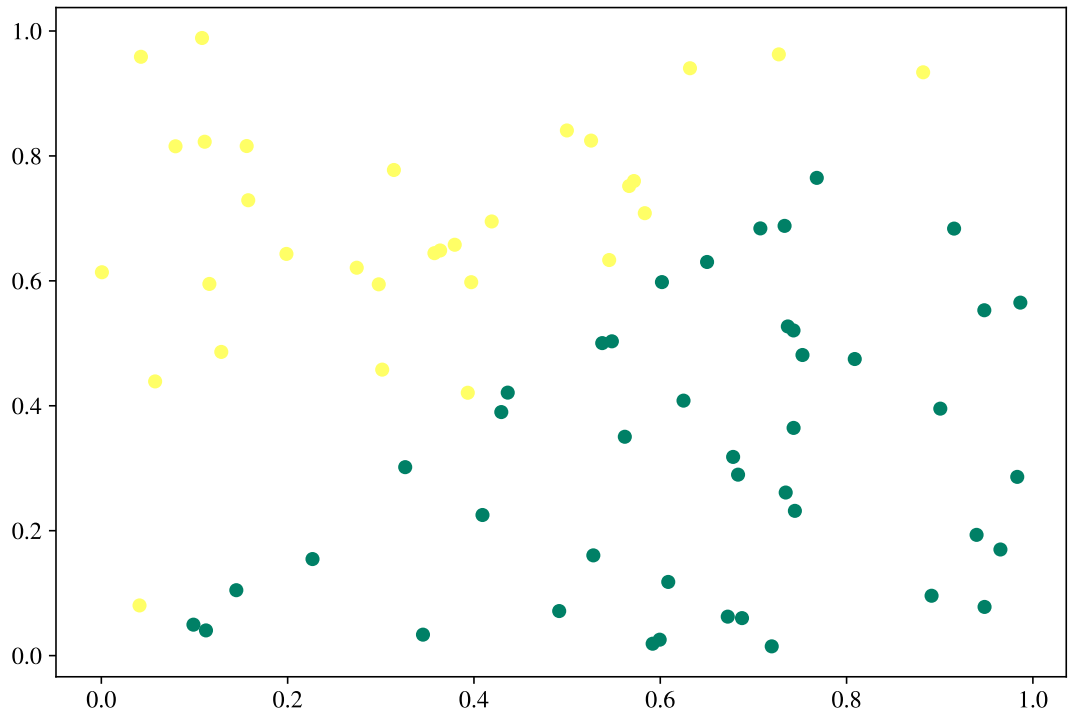
In [8]:

```
X = np.random.rand(75, 2)
y = 1.*(X[:,1] > X[:,0])
```

In [9]:

```
plt.scatter(X[:,0], X[:,1], c=y)
```

```
<matplotlib.collections.PathCollection at 0x7b530f18ffa0>
```



In [10]:

```
def entropyGain(X, y, d, theta):
    if len(y) <= 1:
        return 0.
    p = y.mean()
    e = jax.scipy.special.entr(p)
    l = 1.*(X[:,d] < theta)
    p1 = (y * l).sum()/(l.sum()+1e-12)
    e1 = jax.scipy.special.entr(p1)
    r = 1-l
    p2 = (y * r).sum()/(r.sum()+1e-12)
    e2 = jax.scipy.special.entr(p2)
    return e - (l.sum()*e1 + r.sum()*e2)/len(y)
```

In [11]:

```
def findBestTheta(X, y, d, gain=entropyGain):
    n = len(y)
    best_g = -1.
    theta = None
    xx = jnp.sort(X[:,d])-1e-7
    for t in xx:
        g = gain(X, y, d, t)
        if g > best_g:
            best_g = g
            theta = t
    if theta == None:
        print('theta faillure!!')
    return theta, best_g
```

In [12]:

```
def findBestDTheta(X, y, gain=entropyGain):
    best_d = None
    theta = None
```

```
    best_g = -1
    for d in range(X.shape[1]):
        t, g = findBestTheta(X, y, d, gain)
        if g > best_g:
            best_d = d
            theta = t
            best_g = g
    if best_d is None:
        print('D failure!!')
    return best_d, theta
```

In [13]:

```
class BinaryClassificationTree():
    def __init__(self, X, y, gain=entropyGain, min_size=1):
        p = y.mean()
        if len(y) <= min_size or jax.scipy.special.entr(p) == 0.:
            self.label = 1.*(p>=0.5)
        else:
            self.label = None
            self.d, self.theta = findBestDTheta(X, y, gain)
            ind = 1.*(X[:,self.d] < self.theta)
            if ind.sum() == 0 or ind.sum() == len(y):
                print('single split !!! {} {} {}'.format(ind, y, X))
            ind1 = ind.nonzero()
            X1 = X[ind1]
            y1 = y[ind1]
            ind2 = (1-ind).nonzero()
            X2 = X[ind2]
            y2 = y[ind2]
            self.T1 = BinaryClassificationTree(X1, y1, gain=gain, min_size=min_size)
            self.T2 = BinaryClassificationTree(X2, y2, gain=gain, min_size=min_size)
    def __call__(self, X):
        if self.label is not None:
            return self.label * jnp.ones(len(X))
        return jnp.concatenate([ self.T1([x]) if x[self.d] < self.theta else
self.T2([x]) for x in X])
```

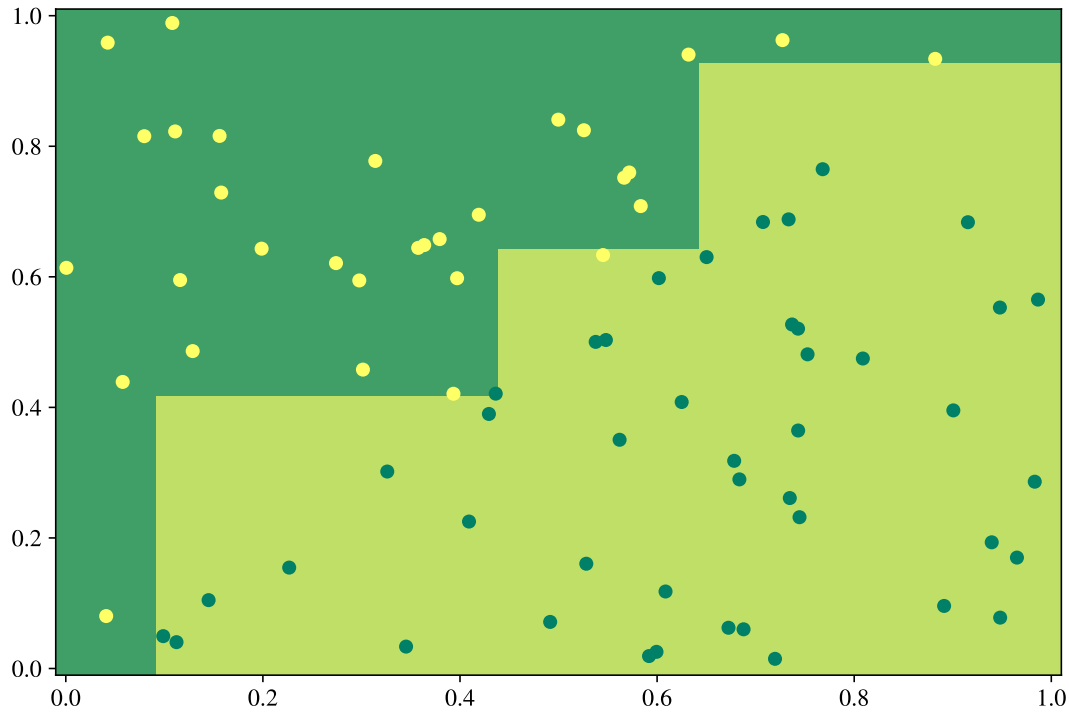
In [14]:

```
T = BinaryClassificationTree(X, y)
```

In [15]:

```
t = 50; tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(T(xx)).reshape(t, t)
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .5, 10)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred, shading='nearest', norm=norm);
plt.scatter(X[:,0], X[:,1], c=y)
```

```
<matplotlib.collections.PathCollection at 0x7b52d4222f80>
```



5.1.8 Decision Trees

- Interpretable
- Fast
- Handle categorical data

But

- Poor accuracy
- Unstable
- Need a lot of examples
- Finding the optimal tree is hard, growing is greedy

5.1.9 Unstable

In [10]:

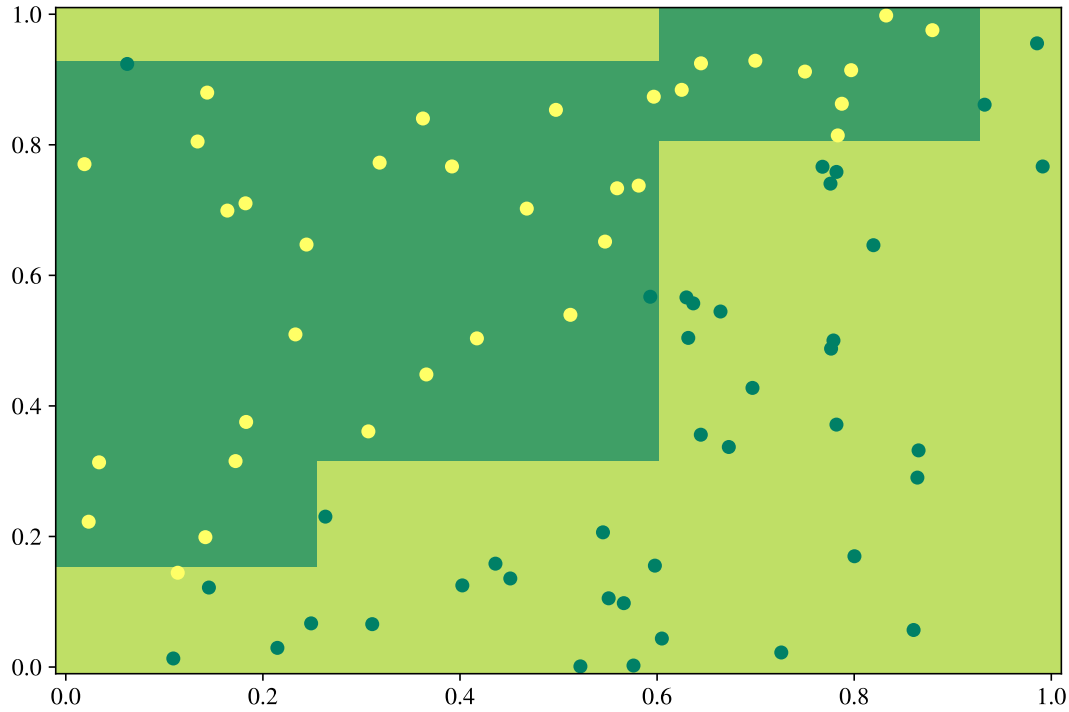
```
y[6] = 1 - y[6]
T = BinaryClassificationTree(X, y)
```

In [11]:

```
t = 50; tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(T(xx)).reshape(t, t)
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .5, 10)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
```

```
plt.pcolormesh(xv, yv, -y_pred, shading='nearest', norm=norm);
plt.scatter(X[:,0], X[:,1], c=y)
```

```
<matplotlib.collections.PathCollection at 0x7fb5d6f7c310>
```



5.1.10 Generalization

Theorem: For a tree of n nodes in dimension d and for m samples, we have with probability δ

$$R \leq R_e + \sqrt{\frac{(n+1)\log_2(d+3) + \log_2(2/\delta)}{2m}}$$

Exercise: What is the VC dimension of decision tree over $\{0,1\}^d$?

5.2 Random Forest

Overcome DT instabilities by averaging B randomized trees [Breiman, 2001]

- Randomized training set $\mathcal{A}_b \subset \mathcal{A}$
- Randomized components $\S \in \mathcal{X}_b \subset \mathcal{X}$

Final decision by majority vote: $f(\mathbf{x}) = \text{argmax}_d \left[\sum_b f_b(\S)\right]_d$

- Average value for regression

5.2.1 Limiting overfitting

Ensemble of classifier $h_1, \ldots, h_K$, define margin function

$$mg(\mathbf{x}, y) = \text{avg}_k \mathbb{1}[h_k(\mathbf{x}) = y] - \max_{j \neq y} \text{avg}_k \mathbb{1}[h_k(\mathbf{x}) = j]$$

(difference between true class vote and max false class vote)

Generalization error

$$R = \mathbb{P}[mg(\mathbf{x}, y) < 0]$$

Random forest: classifier drawn i.i.d. from a distribution of parameters Θ

Theorem [Breiman, 2001]: As the number of trees increases, for almost surely all sequences $\Theta_1, \ldots$, the generalization error R converges to

$$\mathbb{P}\left[\mathbb{P}_\Theta[h_\theta(\mathbf{x}) = y] - \max_{j \neq y} \mathbb{P}_\Theta[h_\theta(\mathbf{x}) = j] < 0\right]$$

R does not increase as the number of trees grows, limiting overfitting

In [12]:

```
class RandomForest():
    def __init__(self, X, y, nb_tree=25, p=0.5):
        self.trees = []
        n = len(y)
        k = int(p*n)
        for b in range(nb_tree):
            i = np.random.permutation(n)
            Xb = X[i[0:k], ...]
            yb = y[i[0:k]]
            DT = BinaryClassificationTree(Xb, yb)
            self.trees.append(DT)
    def __call__(self, X):
        y = []
        for DT in self.trees:
            y.append(DT(X))
        return 1.*(jnp.array(y).mean(axis=0))
```

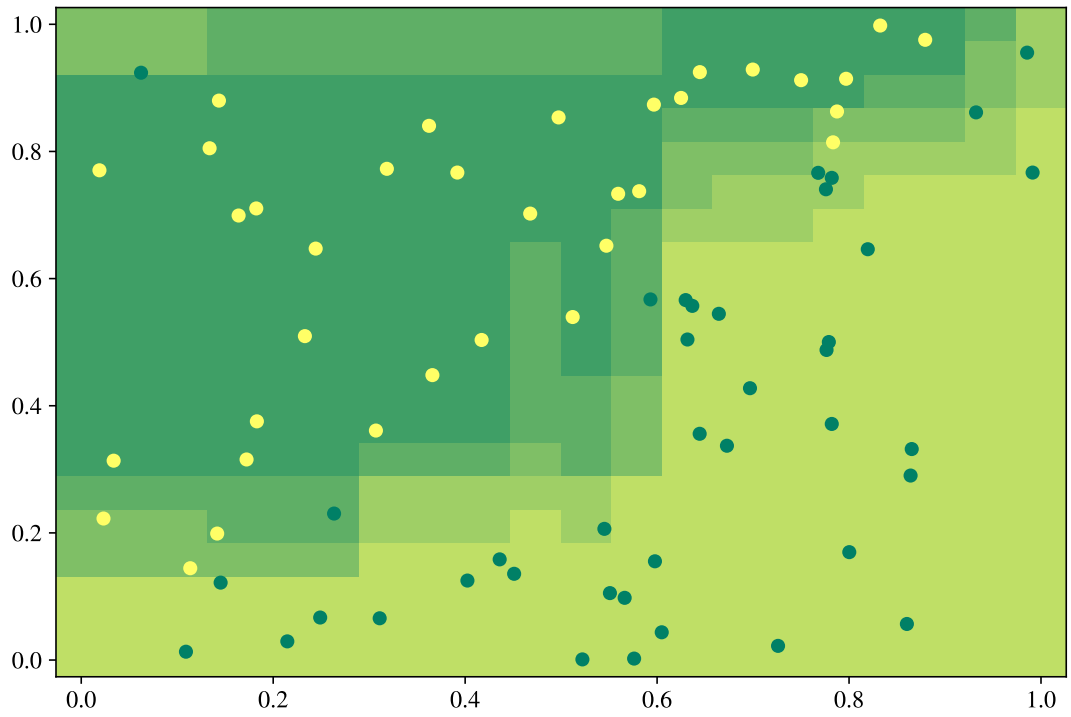
In [13]:

```
T = RandomForest(X, y)
```

In [14]:

```
t = 20; tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(T(xx)).reshape(t, t)
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .5, 10)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred, shading='nearest', norm=norm);
plt.scatter(X[:,0], X[:,1], c=y)
```

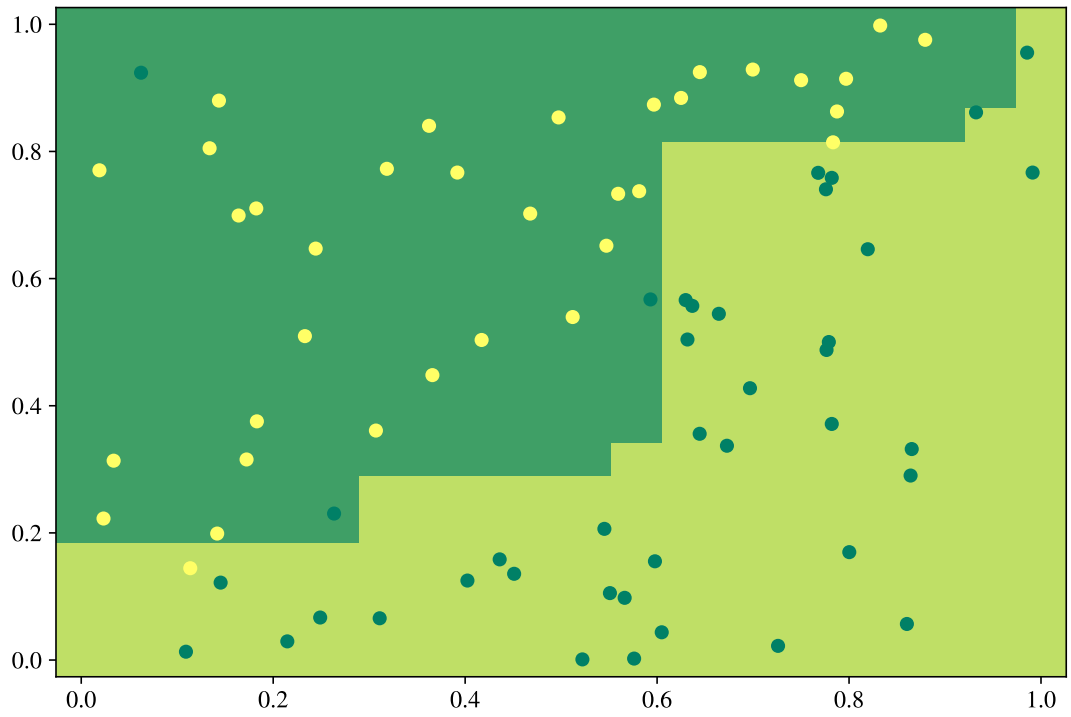
```
<matplotlib.collections.PathCollection at 0x7fb5ac716ce0>
```



In [15]:

```
t = 20; tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(T(xx)).reshape(t, t)
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .5, 10)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -1.*(y_pred>0.5), shading='nearest', norm=norm);
plt.scatter(X[:,0], X[:,1], c=y)
```

```
<matplotlib.collections.PathCollection at 0x7fb5ac7b5870>
```



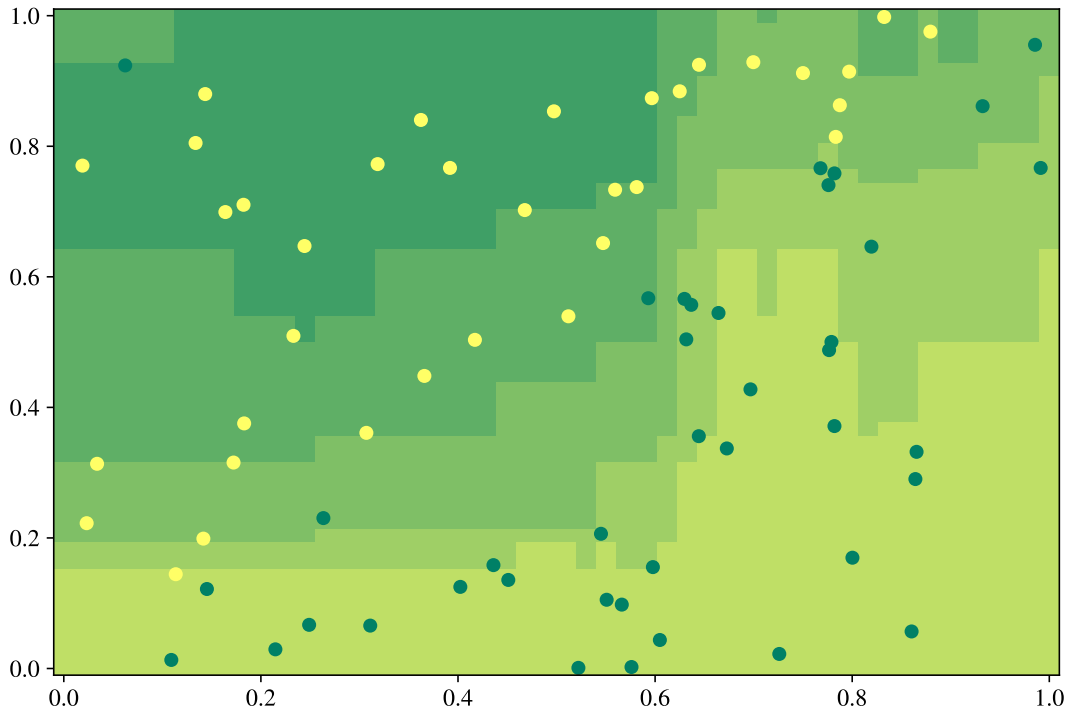
In [16]:

```
T = RandomForest(X, y, nb_tree=100, p=0.2)
```

In [17]:

```
t = 50; tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(T(xx)).reshape(t, t)
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .5, 10)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred, shading='nearest', norm=norm);
plt.scatter(X[:,0], X[:,1], c=y)
```

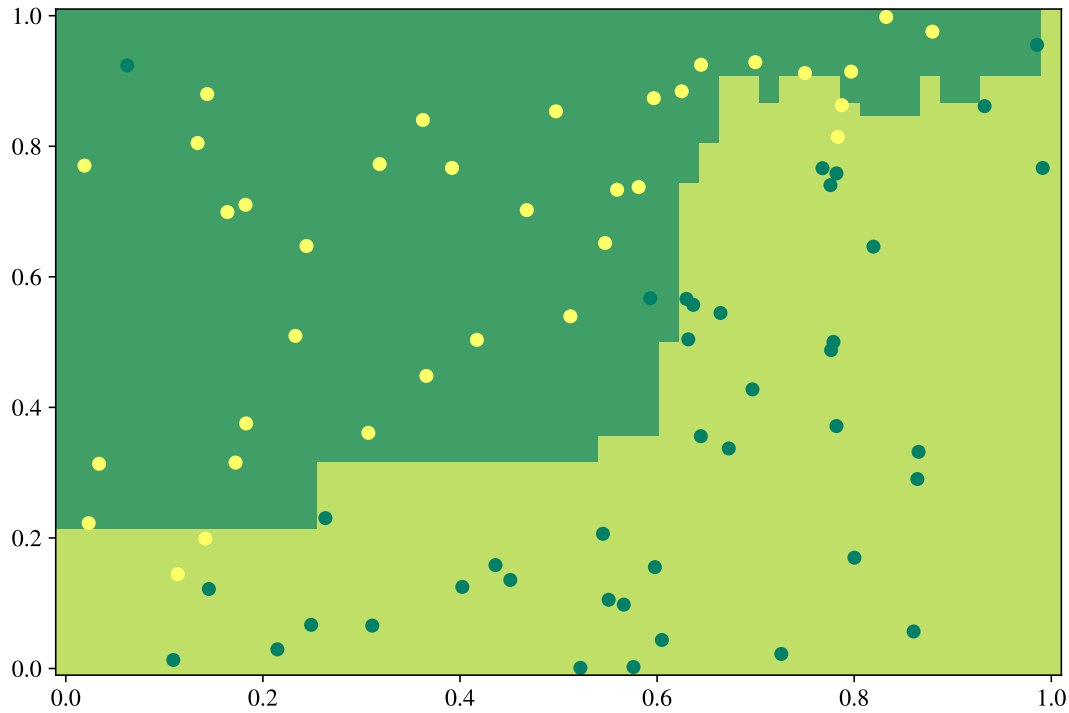
```
<matplotlib.collections.PathCollection at 0x7fb5ac6463b0>
```



In [18]:

```
t = 50; tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(T(xx)).reshape(t, t)
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .5, 10)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -1.*(y_pred>0.5), shading='nearest', norm=norm);
plt.scatter(X[:,0], X[:,1], c=y)
```

```
<matplotlib.collections.PathCollection at 0x7fb5aff8f3a0>
```



5.2.2 Reducing the variance

What is the variance of the average of B random variables, each of variance σ^2, that are correlated by ρ? Correlation

$$\frac{1}{\sigma^2}\mathbb{E}\left[(x_i-\mu)(x_j-\mu)\right]=\rho\geq 0$$

$$\mathbb{E}[x_i^2]=\sigma^2+m^2$$
$$\mathbb{E}[x_ix_j]=\rho\sigma^2+m^2$$

$$Var\left(\frac{\sum_i x_i}{B}\right)=\frac{1}{B^2}Var\left(\sum_i x_i\right)$$

$$=\frac{1}{B^2}\left(\mathbb{E}\left[\left(\sum_i x_i\right)^2\right]-\mathbb{E}\left[\sum_i x_i\right]^2\right)$$

$$=\frac{1}{B^2}\left(\sum_{i,j}\mathbb{E}\left[x_ix_j\right]-\left(\sum_i\mathbb{E}\left[x_i\right]\right)^2\right)$$

$$=\frac{1}{B^2}\left(B(\sigma^2+m^2)+(B^2-B)(\rho\sigma^2+m^2)-B^2m^2\right)$$

$$Var\left(\frac{\sum_i x_i}{B}\right)=\frac{1}{B^2}\left(B(\sigma^2+m^2)+(B^2-B)(\rho\sigma^2+m^2)-B^2m^2\right)$$

$$=\frac{\sigma^2+m^2}{B}+\rho\sigma^2+m^2-\frac{\rho\sigma^2+m^2}{B}-m^2$$

$$= \frac{\sigma^2}{B} + \rho\sigma^2 - \frac{\rho\sigma^2}{B}$$
$$= \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$$
$$\leq \sigma^2 \text{ iff } \rho < 1 \text{ and } B > 1$$

5.3 Ensemble learning

ERM principle subject to bias-variance trade-off

- Simple model: low estimation error, large approximation error
- Complex model: high estimation error, low approximation error

Ensemble idea: aggregate many simple models

- Each model has low estimation error
- Aggregation has low approximation error

5.3.1 Bagging

Assume M **independent** predictors $h_m(\mathbf{x})$

- Trained on different features (*e.g.*, colors and textures)
- Trained on different samples (*e.g.*, different images)

Bagging aggregate

$$h(\mathbf{x}) = \sum_m h_m(\mathbf{x})$$

Corresponds to a voting strategy

5.3.2 Exemple

Random axis, select optimal threshold

In [19]:

```
class RandomAxisClassifier():
    def __init__(self, X, y, d):
        self.d = d
        self.theta, _ = findBestTheta(X, y, d)
        i = jnp.where(X[:,d]<self.theta)
        self.y = (y[i].mean()>0.5)
    def __call__(self, X):
        return (X[:,self.d]<self.theta) if self.y else 1-(X[:,self.d]<self.theta)
```

In [20]:

```
class BaggingClassifier():
    def __init__(self, X, y, nb_cls, p=0.5):
        self.cls = []
        n = len(y)
        k = int(p*n)
        for b in range(nb_cls):
            i = np.random.permutation(n)
```

```
            Xb = X[i[0:k], ...]
            yb = y[i[0:k]]
            self.cls.append(RandomAxisClassifier(Xb, yb, np.random.randint(2)))
    def __call__(self, X):
        y = []
        for c in self.cls:
            y.append(c(X))
        return jnp.array(y).mean(axis=0)
```

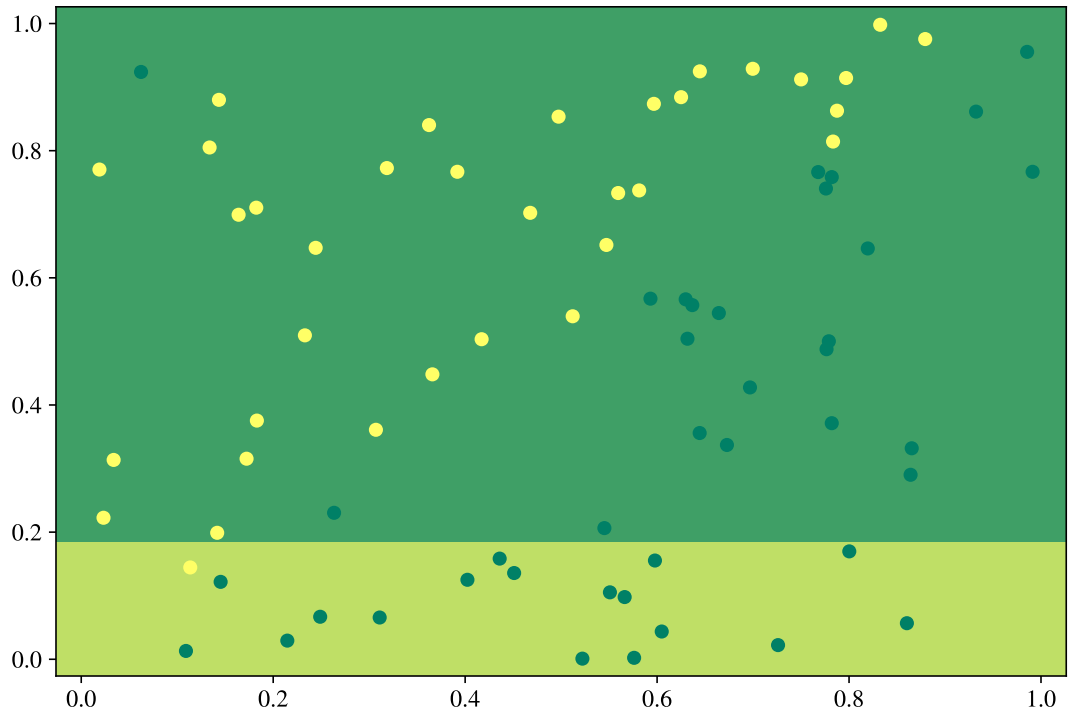
In [21]:

```
T = BaggingClassifier(X, y, 1)
```

In [22]:

```
t = 20; tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(T(xx)).reshape(t, t)
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .5, 10)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred, shading='nearest', norm=norm);
plt.scatter(X[:,0], X[:,1], c=y)
```

```
<matplotlib.collections.PathCollection at 0x7fb5affa1330>
```



In [23]:

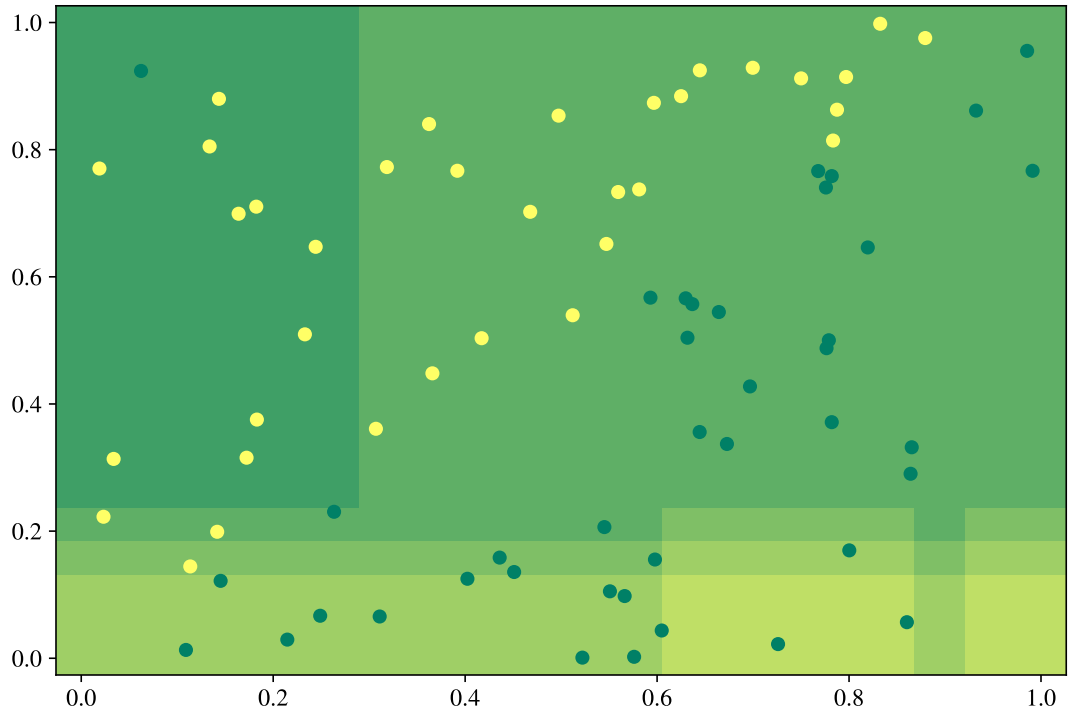
```
T = BaggingClassifier(X, y, 10)
```

In [24]:

```
t = 20; tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(T(xx)).reshape(t, t)
```

```
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .5, 10)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred, shading='nearest', norm=norm);
plt.scatter(X[:,0], X[:,1], c=y)
```

```
<matplotlib.collections.PathCollection at 0x7fb5afd81b10>
```



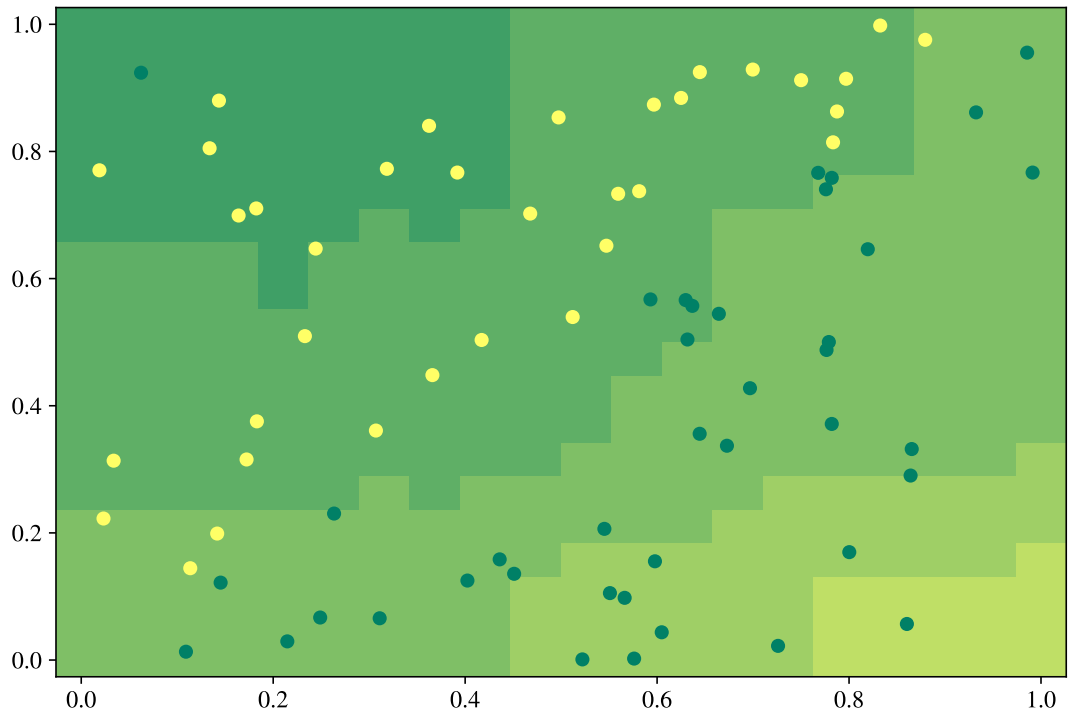
In [25]:

```
T = BaggingClassifier(X, y, 100, p=0.2)
```

In [26]:

```
t = 20; tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(T(xx)).reshape(t, t)
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .5, 10)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred, shading='nearest', norm=norm);
plt.scatter(X[:,0], X[:,1], c=y)
```

```
<matplotlib.collections.PathCollection at 0x7fb5afe03640>
```



5.4 Boosting

In bagging, each predictor as the same weight

- Some decisions are redundant
- Some decisions are bad and we know they are bad

Can we define weights that better reflect each classifier strong points?

5.4.1 Adaboost [Freund and Schapire, 1996]

Key ideas:

- Weight each sample $\mathbf{x}_i$ with w_i
- Train a classifier g with weighted error $w_i I(y_i \neq g(\mathbf{x}_i))$
- Update weights such that samples with high error have higher weights
- Train new classifier f_m with updated weighted error
- Combine both classifier $g \leftarrow g + \beta f_m$
- Iterate until combined classifier is good enough

5.4.2 Exponential loss function

$$L(y, f(\mathbf{x})) = e^{-yf(x)}$$

Given a classifier f_{m-1}, we want to add a new classifier that reduces the error
We have to solve

$$\beta_m, G_m = \arg\min_{\beta, G} \sum_i \exp[-y_i(f_{m-1}(\mathbf{x}_i) + \beta G(\mathbf{x}_i))]$$

5.4.3 Independent updates

$$\beta_m, G_m = \arg\min_{\beta,G} \sum_i \exp[-y_i(f_{m-1}(\mathbf{x}_i) + \beta G(\mathbf{x}_i))]$$

Can be rewritten as

$$\beta_m, G_m = \arg\min_{\beta,G} \sum_i w_i \exp[-y_i \beta G(\mathbf{x}_i)]$$

with

$$w_i = \exp[-y_i f_{m-1}(\mathbf{x}_i)]$$

5.4.4 Solving for G

Remark that given $\beta > 0$

$$\arg\min_{G} \sum_i w_i \exp[-y_i \beta G(\mathbf{x}_i)]$$

is obtained by

$$\arg\min_{G} \sum_i w_i I(y_i \neq G(\mathbf{x}_i))$$

because

$$\sum_i w_i \exp[-y_i \beta G(\mathbf{x}_i)] = e^{-\beta} \sum_{y_i = G(\mathbf{x}_i)} w_i + e^{\beta} \sum_{y_i \neq G(\mathbf{x}_i)} w_i$$

5.4.5 Solving for β

Given G, we have to solve

$$\arg\min_{\beta} \sum_i w_i \exp[-y_i \beta G(\mathbf{x}_i)]$$

Remark that

$$\sum_i w_i \exp[-y_i \beta G(\mathbf{x}_i)]$$
$$= \left(e^{\beta} - e^{-\beta}\right) \sum_i w_i I(y_i \neq G(\mathbf{x}_i) + e^{-\beta} \sum_i w_i$$

Thus

$$\beta = \frac{1}{2} \log \frac{1 - err_m}{err_m}, \quad err_m = \frac{\sum_i w_i I(y_i \neq G(\mathbf{x}_i)}{\sum_i w_i}$$

5.4.6 Adaboost

1. Initialize $\forall i, w_i = 1/N$
2. For $m = 1 \ldots M$
3. Fit classifier $G_m(\mathbf{x})$ to training sample using w_i
4. Compute

$$err_m = \frac{\sum_i w_i I(y_i \neq G_m(\mathbf{x}_i))}{\sum_i w_i}$$

5. Compute $\beta_m = \log((1 - err_m)/err_m)$
6. Set $\forall i, w_i \leftarrow w_i e^{\beta_m I(y_i \neq G_m(\mathbf{x}_i))}$
7. Output $G(\mathbf{x}) = \sum_m \beta_m G_m(\mathbf{x})$

In [16]:

```
def weightedError(w, y_pred, y_true):
    return (w * (y_pred != y_true)).sum()/w.sum()
```

In [17]:

```
def weightedFindBestTheta(w, X, y, d):
    n = len(y)
    err = w.sum()+1
    theta = None
    p = None
    xx = jnp.sort(X[:,d])-1e-7
    for t in xx:
        e = weightedError(w, 1.*(X[:,d] < t), y)
        if e < err:
            err = e
            theta = t
            p = True
        e = weightedError(w, 1. - (X[:,d] < t), y)
        if e < err:
            err = e
            theta = t
            p = False
    if theta == None:
        print('theta faillure!!')
    return theta, p, err
```

In [18]:

```
class WeightedRandomAxisClassifier():
    def __init__(self, w, X, y, d):
        self.d = d
        self.theta, self.y, self.err = weightedFindBestTheta(w, X, y, d)
    def __call__(self, X):
        return 1.*(X[:,self.d]<self.theta) if self.y else 1.-(X[:,self.d]<self.theta)
```

In [19]:

```
class AdaBoost():
    def __init__(self, X, y, nb_cls=5):
        n = len(y)
        w = jnp.ones(n)/n
        self.beta = []
        self.cls = []
        for b in range(nb_cls):
            err_b = []
            cls_b = []
            for d in range(X.shape[1]):
                c = WeightedRandomAxisClassifier(w, X, y, d)
                cls_b.append(c)
                err_b.append(c.err)
            a = jnp.argmin(jnp.array(err_b))
            c = cls_b[a]
            e = c.err
            b = jnp.log((1-e)/e)
            w = w * jnp.exp(b * (c(X)!=y))
            self.beta.append(b)
            self.cls.append(c)
    def __call__(self, X):
        y = []
        for i, c in enumerate(self.cls):
            y.append(self.beta[i] * c(X))
        return jnp.array(y).mean(axis=0)
```

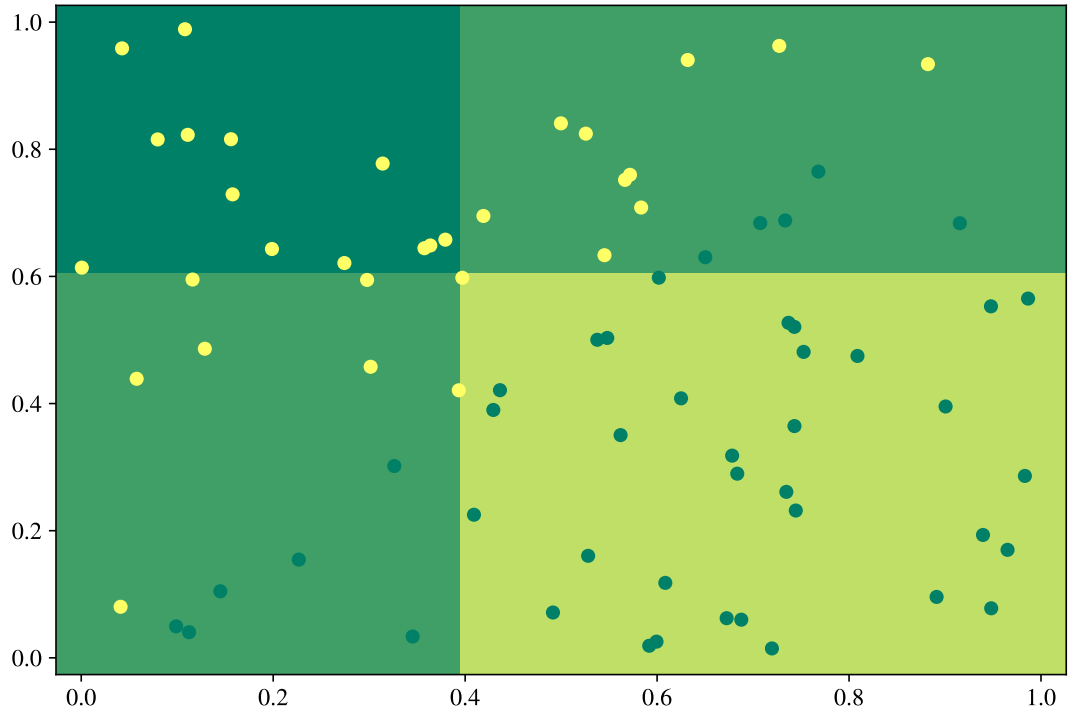
In [20]:

```
T = AdaBoost(X, y, 2)
```

In [21]:

```
t = 20; tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(T(xx)).reshape(t, t)
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .5, 10)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred, shading='nearest', norm=norm);
plt.scatter(X[:,0], X[:,1], c=(y>0.5))
```

```
<matplotlib.collections.PathCollection at 0x7b52bb7818a0>
```



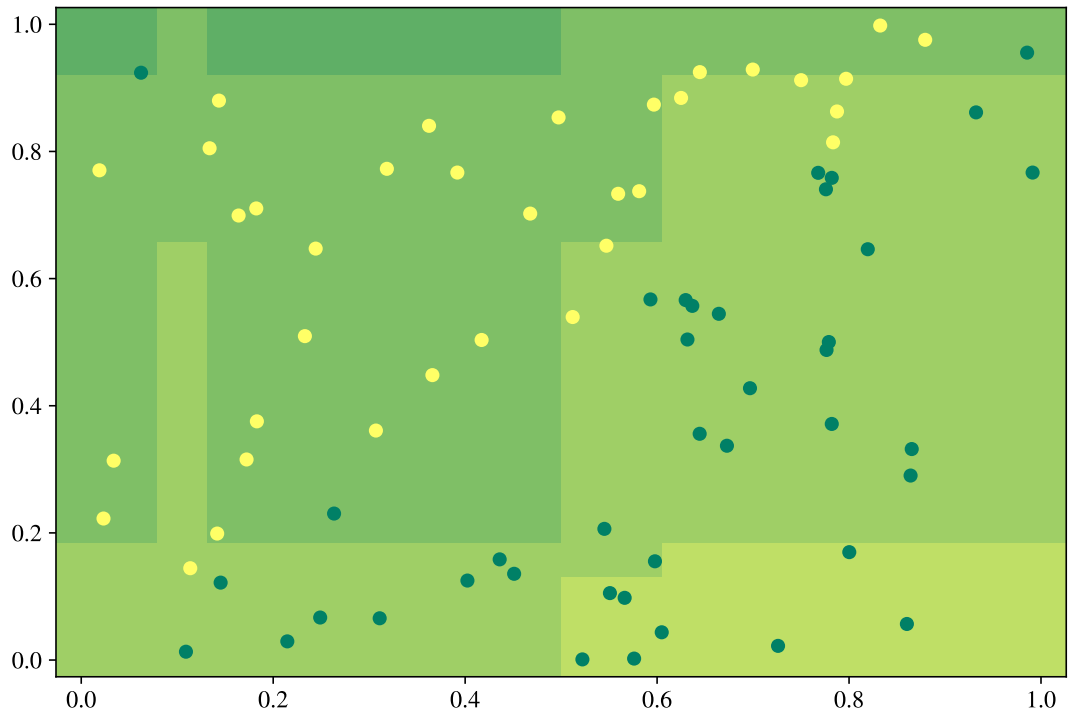
In [33]:

```
T = AdaBoost(X, y, 10)
```

In [34]:

```
t = 20; tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(T(xx)).reshape(t, t)
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .5, 10)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred, shading='nearest', norm=norm);
plt.scatter(X[:,0], X[:,1], c=y)
```

```
<matplotlib.collections.PathCollection at 0x7fb5af9adea0>
```



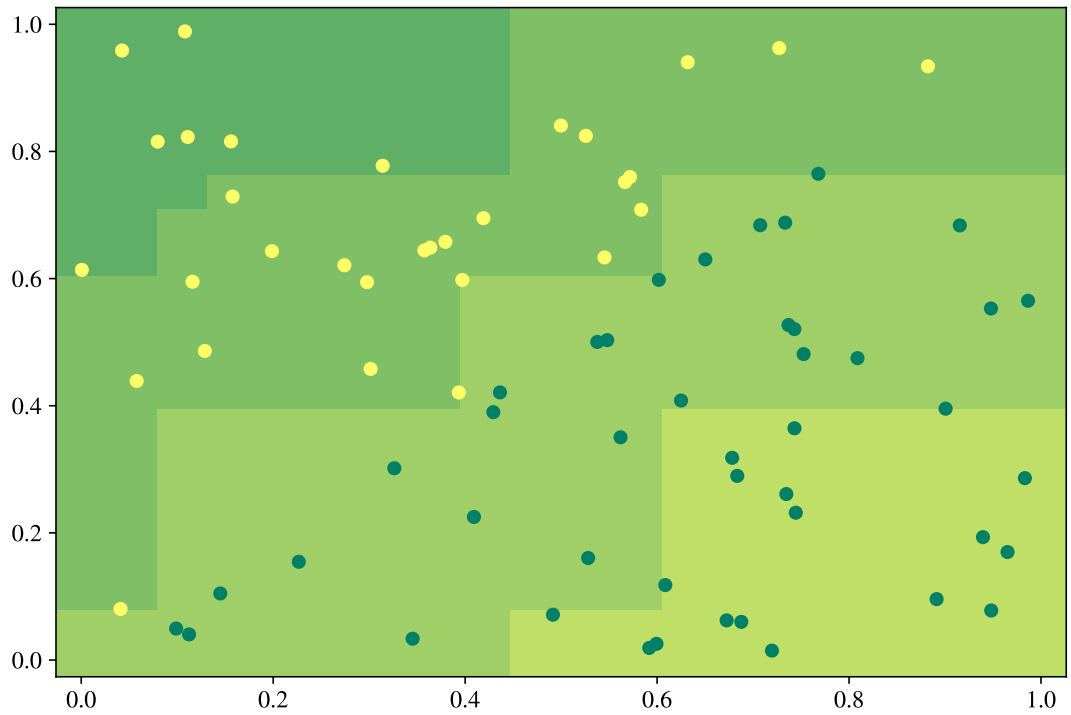
In [22]:

```
T = AdaBoost(X, y, 50)
```

In [24]:

```
t = 20; tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(T(xx)).reshape(t, t)
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .5, 10)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred, shading='nearest', norm=norm);
plt.scatter(X[:,0], X[:,1], c=(y>0.5))
```

```
<matplotlib.collections.PathCollection at 0x7b52bb7f1930>
```



5.4.7 Gradient Tree Boosting

Given clasiffier $\hat{y}_i = \sum_k f_k(x_i)$, consider objective

$$\mathcal{L} = \sum_i l(y_i, \hat{y}_i) + \sum_k^T \Omega(f_k), \quad \Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2$$

Second order approx

$$\mathcal{L}^t \approx \sum_i [l(y_i, \hat{y}_i) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i] + \Omega(f_t)$$

$g_i = \partial_{\hat{y}_i} l(y_i, \hat{y}_i)$, $g_i = \partial^2_{\hat{y}_i} l(y_i, \hat{y}_i)$ Remove constant term and define $I_j = \{i | q(x_i) = j\}$ the set of samples in leaf j

$$\tilde{\mathcal{L}} = \sum_j^T \left[w_j \sum_{i \in I_j} g_i + \frac{w_j^2}{2} (\lambda + \sum_{i \in I_j} h_i) \right] + \gamma T$$

Optimal weight

$$w_j^\star = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}$$

Optimal objective value

$$\mathcal{L}^\star = -\frac{1}{2} \sum_j^T \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T$$

Provides a criterion for splitting nodes: - Letting $I = I_L \cup I_R$

$$\mathcal{L}_{\text{split}} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma$$

- if < 0, do not split
- can be < 0 because of γ (regularisation)

- In practice, randomizing features and samples allows very large trees
- taking a sub-optimal weight ($\eta w_i^\star$) allows more room for subsequent splits.

Very efficient implementations (e.g., XGBoost, [Chen and Guestrin, 2016])

5.4.8 Remarks

Classifiers have to be weak

- A perfect classifier yields $\beta_m = \infty$ breaking subsequent classifiers
- Example: Kernel SVM with sufficiently tight Gaussian kernel

Classifiers have to be slightly better than random

- Worst than random gets an negative weight (opposite classifier)
- Example: single feature classifier

Boosted Decision Trees are an excellent first try in most cases

5.4.9 Exercise

Show that

$$f^\star(\mathbf{x}) = \arg\min_f \mathbb{E}_{\sim(\mathbf{x},y)}\left[e^{-yf(\mathbf{x})}\right] = \frac{1}{2}\log\frac{P[y=1|\mathbf{x}]}{P[y=-1|\mathbf{x}]}$$

5.5 Decision Trees and Ensemble Learning, take home

Decision Trees

- Simple
- Fast
- Explainable (domain experts understand the decision process)
- Handle categorical data (or even mixed)

But

- Overfit, unstable
- Require massive amount of data

Random forest

- Simple, Fast, handle categorical data

- Stable
- No longer explainable

Ensemble

- Bagging: simple solution, good idea to reduce bias
- Boosting: optimized combination

Large literature and many libraries on boosting

- Bossted trees: very good default classifier in many cases (see [Grinsztajn et al., 2022])

In [ ]:

Chapter 6

Time Series

Prediction is difficult, especially if it's about the future,

Niels Bohr

- Specific case of $y = f(x)$ where y are x are temporally related
- Often used in a setup that breaks classical ML assumption (i.i.d., stationarity)

6.1 General setup

- Stochastic process $X_t, 0 \leq t \leq T$
- Observe $\{X_t\}_{1 \leq t \leq \tau}$ for $\tau < T$
- Predict $\{X_t\}_{\tau < t \leq T}$

Simplest case: predict next value X_T from $X_{t<T}$

6.1.1 Training set

Machine Learning relies on i.i.d. training/test data, but

- Obviously $P(X_T|X_{t<T}) \neq P(X_T)$ else nothing to observe
- Training set has to consist of *many* independant sequences (i.d.), else no generalization

Typical cases:

- $X_t = f(X_{t-1}, ..., X_0)$ autoregressive process
- $X_t = f(X_{t-1})$, Markov process
- $Y_t = f(X_t), X_t = g(X_{t-1})$, Hidden Markov model (observe Y, deduce X)
- $Y_t = f(X_t, ..., X_0, Y_{t-1}, ..., Y_0)$, Infinite impulse response system (control theory)

6.2 Autoregressive processes

Setup:

$$X_n = f(X_{n-1}, ...X_0)$$

Assume linear model with additive noise:

$$X_n = \sum_{k=1}^{P} a_k X_{n-k} + \epsilon_n, \epsilon_n \sim \mathcal{N}(0, \sigma_\epsilon)$$

Assume stationarity:

- $\forall n, E[X_n] = E[X_0] = \mu$, constant mean (=0 for simplicity)
- $\forall n, k, E[X_n X_k] = E[X_{n-k} X_0]$, autocorrelation does not depend on t, only $n - k$
- $\forall n, E[X_n^2] < \infty$, bounded variance

Then, one long sequence split into chunks is as good as many small sequences

6.2.1 Fitting the model

Multiplying by X_{n-m} and taking the expectation:

$$\gamma_X(m) = \sum_{k=1}^{P} a_k \gamma_X(m-k) + \gamma_{X,\epsilon}(k)$$

with $\gamma_X(k) = E[X_n X_{n-k}]$ the autocorrelation and $\gamma_{X,\epsilon}(k) = E[X_n \epsilon_{n-k}]$ the cross-correlation
Note that $\gamma_{X,\epsilon}(k) = 0$ except for $\gamma_{X,\epsilon}(0) = \sigma_\epsilon^2$ (ϵ_t are independant)

We have a system of $P + 1$ equations (called Yule-Walker equations):

$$\gamma_X(0) = \sum_{k=1}^{P} a_k \gamma_X(k) + \sigma_\epsilon^2 \gamma_X(m) = \sum_{k=1}^{P} a_k \gamma_X(m-k)$$

In matrix form:

$$\Gamma_X \mathbf{a} = \mathbf{p} \qquad \gamma_X(0) = \mathbf{a}^\top \gamma_X + \sigma_\epsilon^2$$

with Γ_X the covariance matrix (symmetric, ≥ 0)
Which corresponds to a least square problem. Solution:

$$\mathbf{a} = \Gamma_X^{-1} \mathbf{p} \qquad \sigma_\epsilon = \gamma_X(0) - \mathbf{p}^\top \Gamma_X^{-1} \mathbf{p}$$

- $\mathbf{a}$ corresponds to frequencies associated with a linear filter
- Start from pure noise $\rightarrow$ get a process that resembles the observations
- n-step ahead prediction: use $X_{\leq n}$ to predict X_{n+1}, then add to the sequence and predict X_{n+2}
- AR models can be used online: the more you observe, the more you can refine the future
- Limited to stationary processes (rare for interesting problems)
- For generic processes X, AR modeling is the best linear model in least square, and σ_ϵ is the error in modeling

6.2.2 Example

Trying with the periodique function from the chapter on Linear Models.

In [2]:

```
a = np.array([0.7, 0.83, -1.5])
x = np.linspace(-10, 5, 75)
X = np.array([ np.sin(x), np.sin(2*x), np.sin(3*x)])
y = np.matmul(a, X) + 0.1*np.random.randn(75)
```

In [3]:

```
def fit_ar_p(y, P):
    N = len(y)-P-1
    T = np.array([y[i:i+P+1] for i in range(N)]) # set of P sized sequences
    T = T.T @ T / N # covariance matrix
    G = T[0:P, 0:P]
    p = T[P,0:P]
    a_hat = np.linalg.inv(G)@p.T
    s_hat = T[P,P] - p@np.linalg.inv(G)@p.T
    return a_hat, s_hat
```

In [4]:

```
P = 1
a_hat, s_hat = fit_ar_p(y, P)
```

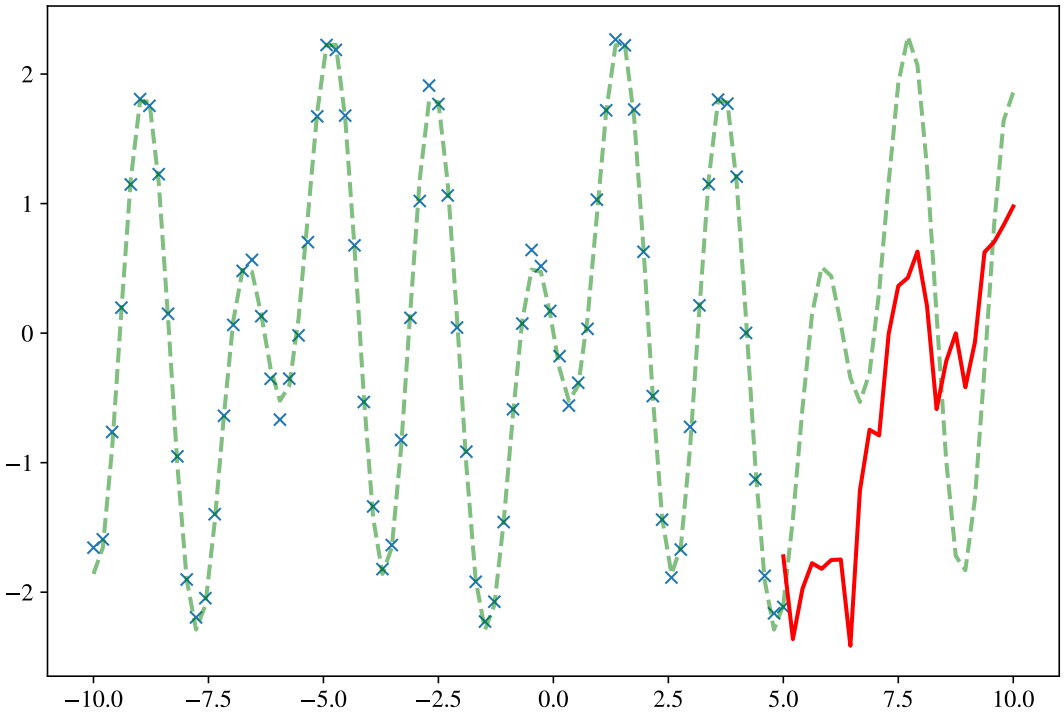
In [5]:

```
# AR prediction
y_hat = y[-P:]
for i in range(25):
    next = np.array([np.dot(y_hat[-P:], a_hat) + s_hat*np.random.randn()])
    y_hat = np.concatenate([y_hat, next])
```

In [6]:

```
x2 = np.linspace(5, 10, 25)
plt.plot(x, y, 'x')
plt.plot(x, np.matmul(a, X), 'g--', alpha=0.5)
plt.plot(x2, np.matmul(a, np.array([ np.sin(x2), np.sin(2*x2), np.sin(3*x2)])), 'g--',
alpha=0.5)
plt.plot(x2, y_hat[P:], 'r-')
```

```
[<matplotlib.lines.Line2D at 0x7eb5600cefe0>]
```



In [7]:

```
P = 10
a_hat, s_hat = fit_ar_p(y, P)
```

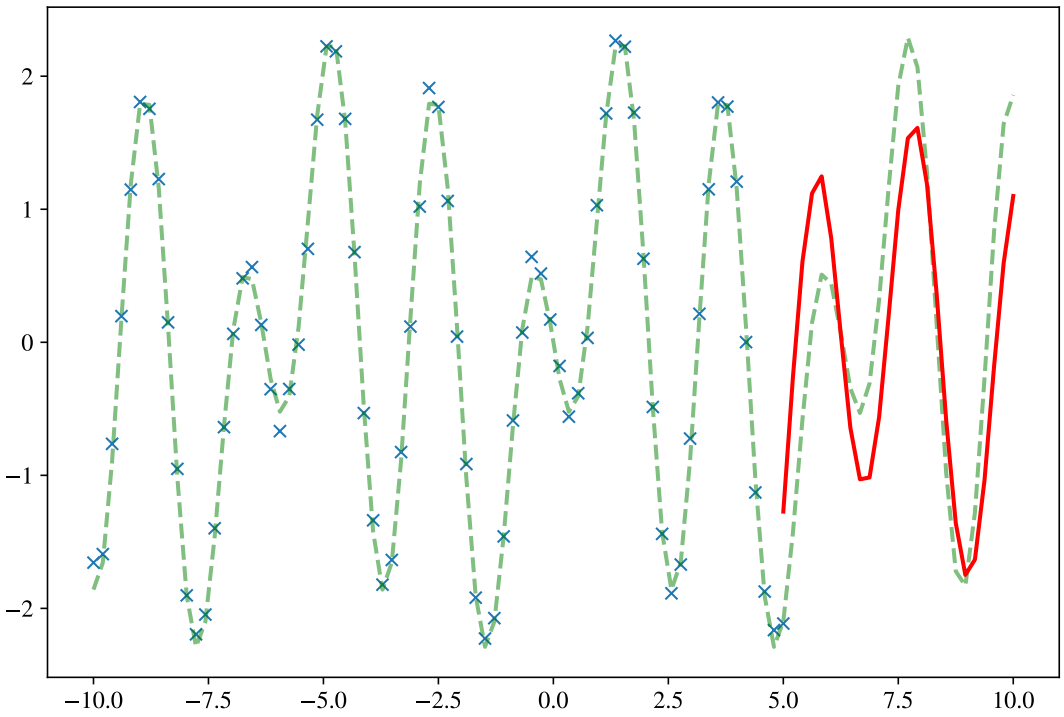
In [8]:

```
# AR prediction
y_hat = y[-P:]
for i in range(25):
    next = np.array([np.dot(y_hat[-P:], a_hat) + s_hat*np.random.randn()])
    y_hat = np.concatenate([y_hat, next])
```

In [9]:

```
x2 = np.linspace(5, 10, 25)
plt.plot(x, y, 'x')
plt.plot(x, np.matmul(a, X), 'g--', alpha=0.5)
plt.plot(x2, np.matmul(a, np.array([ np.sin(x2), np.sin(2*x2), np.sin(3*x2)])), 'g--',
alpha=0.5)
plt.plot(x2, y_hat[P:], 'r-')
```

```
[<matplotlib.lines.Line2D at 0x7eb557fb7700>]
```



In [10]:

```
P = 25
a_hat, s_hat = fit_ar_p(y, P)
```

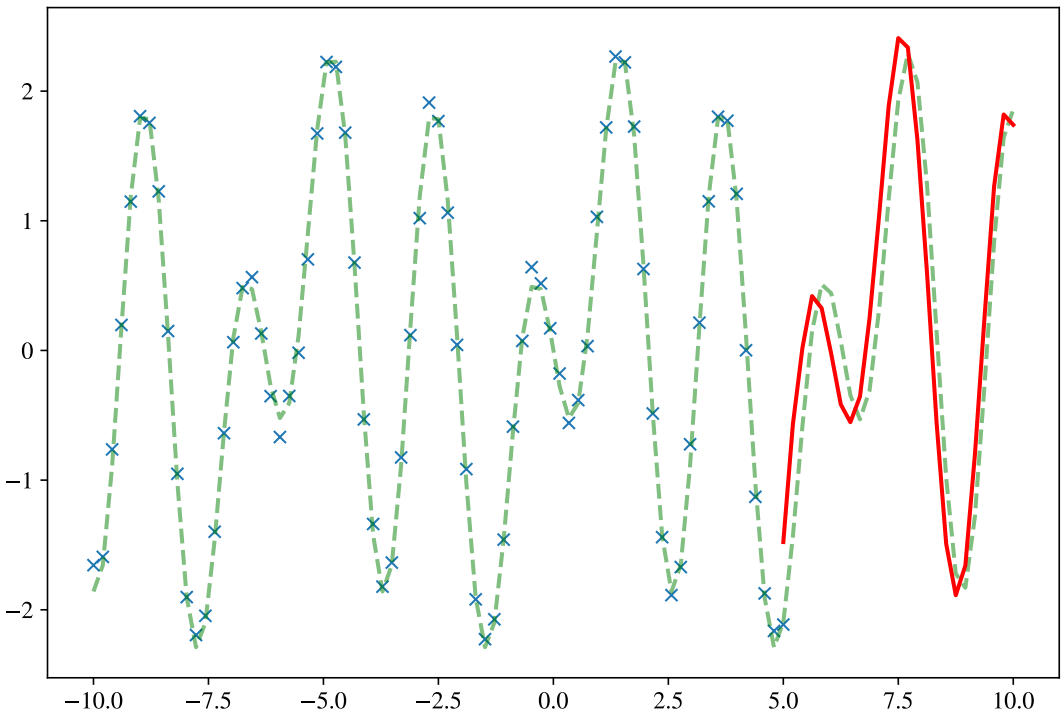
In [11]:

```
# AR prediction
y_hat = y[-P:]
for i in range(25):
    next = np.array([np.dot(y_hat[-P:], a_hat) + s_hat*np.random.randn()])
    y_hat = np.concatenate([y_hat, next])
```

In [12]:

```
x2 = np.linspace(5, 10, 25)
plt.plot(x, y, 'x')
plt.plot(x, np.matmul(a, X), 'g--', alpha=0.5)
plt.plot(x2, np.matmul(a, np.array([ np.sin(x2), np.sin(2*x2), np.sin(3*x2)])), 'g--',
alpha=0.5)
plt.plot(x2, y_hat[P:], 'r-')
```

```
[<matplotlib.lines.Line2D at 0x7eb5555534f0>]
```



6.3 Markov Models

6.3.1 Markov chains

Consider discrete process $X_n \in \mathcal{C}, |\mathcal{C}| = M$ states

- Markov property: $P(X_n|X_{n-1}, ..., X_0) = P(X_n|X_{n-1})$
- Limited memory equivalence:

$$P(X_n|X_{n-1}, ...X_0) = P(X_n|X_{n-1}, ..., X_{n-l}) \Leftrightarrow P(Y_n|Y_{n-1}, ..., Y_0) = P(Y_n|Y_{n-1}), Y_n = X_n...X_{n-k}$$

Use the observations to estimate the transition matrix:

$$\mathbf{P} = [P(X_n = j|X_{n-1} = i)]_{i,j}$$

- Empirical estimation
- Each row is ≥ 0 and sums to 1 (row stochastic matrix)
- Given a state distribution π, the next state distribution is given by $\pi\mathbf{P}$

Stationary distribution π given by

$$\pi = \pi\mathbf{P}$$

- Corresponds to iterating the process, $\mathbf{P}^k, k \to \infty$
- π is a left eigenvector of $\mathbf{P}$
- Convergence in λ_1/λ_2 (iterated power of $\mathbf{P}$
- See Perron-Frobenius theorem for nice properties of such processes

6.3.2 Hidden Markov models

What if X is not observable, but Y is?

- Model $P(Y_n|X_n)$ (emission probability)
- Model $P(X_n|X_{n-1})$ (transition probability)

Y_n can be either discrete or continous.
See the good (but old) tutorial by [Rabiner, 1989]

Three problems

Consider HMM model $\lambda(A, B, \pi)$ with transition matrix A, emission vector B and initial distribution π

1. Given $Y_0, ..., Y_n$, how to efficiently compute $P(Y|\lambda)$? (observation probability of Y)
2. Given $Y_0, ..., Y_n$, how to estimate the best $X_0, ..., X_n$ in the sense of λ? (explication for Y)
3. Given $Y_0, ..., Y_n$, how to adjust λ to maximize the lieklihood of Y? (learning the HMM)

Probability estimation

Probability of an observed sequence $Y = Y_0, ..., Y_n$:

- $P(Y) = \sum_X P(Y|X)P(X)$
- sum over all possible hidden state sequences $X = X_0, ..., X_n$

Done using dynamic programming Forward Algorithm:

$$\alpha(X_t) = P(Y_{\leq t}, X_t)$$

Using the chain rule, following recusrion:

$$\alpha(X_t) = P(Y_t|X_t) \sum_{X_{t-1}} P(X_t|X_{t-1})\alpha(X_{t-1})$$

- $P(Y_t|X_t) = B_{Y_t}$ the corresponding row of the emission matrix B
- $P(X_t|X_{t-1})$ the corresponding cell of the transition matrix A
- Initialize with $\alpha(X_0)$ from $P(X_0)$

Best hidden states explanation

Viterbi algorithm [Viterbi, 1967]

- Forward pass: compute the state that has max probability given past
- Backward pass: starting from the maximum proba last state, iterate previous

Complexity in $T \times M^2$

6.3.3 Fitting HMM

Expectation Maximization (Baum-Welsh algorithm [Baum et al., 1970]):

- Estimate probabilities of X and Y given model
- Update transition and emission matrices according to maximize the estimated probabilities

Estimation

- Forward/backward variables:

$$\alpha_i(t) = P(Y_{\leq t}, X_t = i) = \sum_j \alpha_j(t-1) a_{j,i} b_i$$

$$\beta_i(t) = P(Y_{>t}, X_t = i) = \sum_j \beta_j(t+1) a_{i,j} b_j$$

- Intermediate variables:

$$\mathcal{L} = P(Y) = \sum_i \alpha_i(T)$$

$$\gamma_i(t) = P(X_t = i | Y_{\leq t}) = \frac{\alpha_i(t)\beta_i(t)}{\mathcal{L}}$$

$$\xi_{i,j}(t) = P(X_t = i, X_{t+1} = j | Y_{>t}) = \frac{\alpha_i(t) a_{i,j} b_j \beta_j(t+1)}{\mathcal{L}}$$

Maximization

$$\pi = \gamma(1)$$

$$a_{i,j} = \frac{\sum_t \xi_{i,j}(t)}{\sum_t \gamma_i(t)}$$

$$b_i = \frac{\sum_t [\gamma_i(t)]_{=Y_t}}{\sum_t \gamma_i(t)}$$

6.4 Gaussian Processes

Distribution over functions defined by the Gaussian distribution

- Set of observation $X = \{X_0, ..., X_n\}$
- Random function f sampled from a distribution such that

$$f(X) = \mathcal{N}(m(X), k(X, X))$$

with $m(X)$ a function providing the means for X and $k(X, X)$ providing the covariances for X

6.4.1 Choice of kernel

- $k(x, x')$ has to be positive definite (to lead to a feasible covariance matrix)
- Models how different points influence each other

Popular choice: Gaussian kernel

$$k(x_1, x_2) = \exp\left(\frac{\|x_1 - x_2\|^2}{2\sigma^2}\right)$$

with σ carefully chosen

6.4.2 Inference

Predict $Y_2 = f(X_2)$ depending on observed data X_1, Y_1

Assume that Y_1 and Y_2 come from the same multivariate Gaussian:

$$\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \mathcal{N}\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11}, \Sigma_{12} \\ \Sigma_{21}, \Sigma_{22} \end{bmatrix}\right)$$

With $\Sigma_{ij} = k(X_i, X_j)$ Getting the conditional probability

$$P(Y_2|Y_1, X_1, X_2) = \mathcal{N}(\mu_{2|1}, \Sigma_{2|1})$$

with

$$\mu_{2|1} = \mu_2 + \Sigma_{21}\Sigma_{11}^{-1}(Y_1 - \mu_1) = \Sigma_{21}\Sigma_{11}^{-1}(Y_1 - \mu_1)$$

(assume $\mu_2 = 0$)

$$\Sigma_{2|1} = \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}$$

6.4.3 Handling noise

$$f(X2) = Y_2 + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma)$$

Model in the kernel:

$$\Sigma_{11} = k(X, X) + \sigma I$$

6.4.4 Example

Trying with the periodique function from the chapter on Linear Models.

In [13]:

```
a = np.array([0.7, 0.83, -1.5])
x = np.random.rand(48)*16-8
X = jnp.array([ jnp.sin(x), jnp.sin(2*x), jnp.sin(3*x)])
y = jnp.matmul(a, X) + 0.2*np.random.randn(48)
```

In [14]:

```
def GaussKernel(x1, x2, gamma=5):
    x1 = jnp.expand_dims(x1, 1)
    x2 = jnp.expand_dims(x2, 1)
    return jnp.exp(-gamma*( jnp.linalg.norm(x1, axis=-1, keepdims=True)**2 +
jnp.linalg.norm(x2, axis=-1, keepdims=True).T**2 - 2*jnp.dot(x1, x2.T)))
```

In [15]:

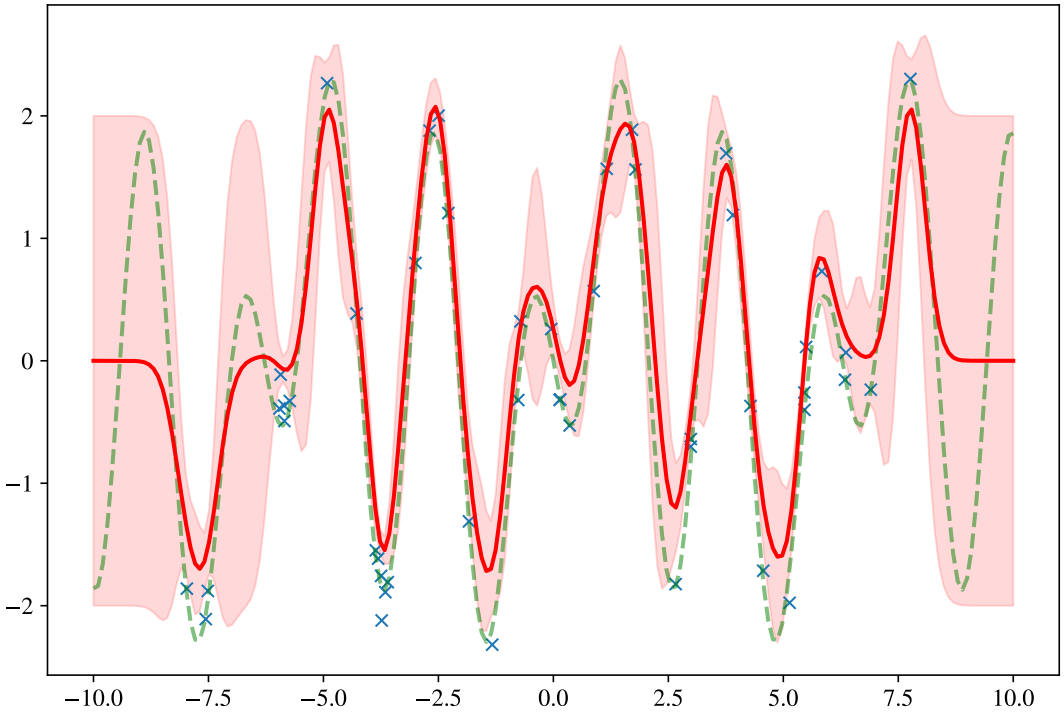
```
def sample_gp(X2, X1, Y1, kernel, noise=0.25):
    S11 = kernel(X1, X1) + noise * jnp.eye(len(X1))
    S22 = kernel(X2, X2)
    S12 = kernel(X1, X2)
    S21 = S12.T

    mu1 = jnp.mean(X1)
    S11inv = jnp.linalg.inv(S11)

    mu2to1 = S21 @ S11inv @ (Y1 - mu1)
    S2to1 = S22 - S21 @ S11inv @ S12

    return mu2to1, jnp.diag(S2to1) # mean = best prediction, var = range of functions
```

In [16]:

```
x2 = np.linspace(-10, 10, 200)
X2 = np.array([np.sin(x2), np.sin(2*x2), np.sin(3*x2)])
y2, s2 = sample_gp(x2, x, y, GaussKernel)
```

In [17]:

```
plt.plot(x, y, 'x')
plt.plot(x2, np.matmul(a, X2), 'g--', alpha=0.5)
plt.plot(x2, y2, 'r-')
plt.fill_between(x2, y2-2*s2, y2+2*s2, color='red', alpha=0.15)
plt.show()
```



6.4.5 Exercice

Try to fit the previous example using periodic kernels

$$k(x_1, x_2) = \exp\left(-\gamma \sin^2\left(\frac{\|x_1 - x_2\|}{p}\pi\right)\right)$$

6.5 Recurrent neural networks

Use a neural network to model the transition from X_n to X_{n+1}

$$X_{n+1} = f_\theta(X_n)$$

In practice, keep information in a hidden state:

$$X_{n+1}, H_{n+1} = f_\theta(X_n, H_n)$$

With H_n hidden state (initialized to 0)

6.5.1 Training an RNN

- Offset the observed sequence:

$$Y_0, ..., Y_n = X_1, ..., X_{n+1}$$

- Unroll the forward pass:

$$f_\theta(X_0, 0), ..., f_\theta(X_n, H_n)$$

- SGD with teacher forcing:

$$\min_\theta \sum_i \|Y_i - f_\theta(X_i, H_i)\|^2$$

- Sequential cost during training, costly for very long sequences
- Inference is almost always out-of-distribution

Vanishing information

Linear model:

$$H_{n+1} = f(X_n, H_n) = W_h H_n + W_x X_n$$

Full unrolling:

$$H_{n+1} = W_h^n W_x X_0 + g(X_1, ..., X_n)$$

Depending on the spectrum of W_h, X_0 could have totally vanished from H

6.5.2 LSTM

Preventing vanishing information [Hochreiter and Schmidhuber, 1997]

- Hidden state can store finite amount of information
- Decide when to let information in
- Decide when to forget information

- input gate: $i_n = \sigma_g(W_{i,x} x_n + W_{h,x} h_n)$
- forget gate: $f_n = \sigma_g(W_{f,x} x_n + W_{h,x} h_n)$

- cell input: $\tilde{c}_n = \sigma_h(X_{c,x}x_n + W_{c,h}h_n)$
- cell: $c_n = f_n \odot c_{n-1} + i_n \odot \tilde{c}_n$
- output gate: $o_n = \sigma_g(W_{o,x}x_n + W_{o,h}h_n)$
- output: $h_n = o_n \odot \sigma_h(c_n)$

σ_g sigmoid, σ_h hyperbolic tangent Prediction:

- Stack several layers (for layer L, forward hidden state such that $X_n^L = H_n^{L-1}$)
- Transform last layer hidden state H_n into the next input X_{n+1}

LSTM are notoriously hard to train

- Sigmoids saturate $\rightarrow$ carefull initialization and learning rate
- Need unrolling and sequential processing during training
- Causal problem: need to guess what to put in the cell before knowing if it's useful

6.6 Autoregressive transformers

RNN have 2 problems:

- Sequential training is slow and costly
- Inference is tricky (need to memorize patterns before knowing if it's useful)

Transformers solve both problem:

- Training can be done in parallel
- Memorizes all the input sequence

(Softmax) Attention

- Input sequence X (sequence of tokens in columns)
- Projection W_q, W_k, W_v

Queries, Keys, Values:

$$Q = W_q X, K = W_k X, V = W_v X$$

Attention matrix (Gram matrix + softmax):

$$A = \sigma(Q^\top K)$$

Attention operation:

$$Y = X + AV$$

Aggregates information from the sequence into each token

Transformer layer

Residual attention and small MLP (typically 1 hidden layer)

$$X^{L+1} = X^L + \sigma(Q^\top K)V + \mathrm{FFW}(X)$$

- Attention: mixes information between tokens
- MLP: processes each token

Masked attention

Transorfmers are intented for set (permutation equivariant) $\rightarrow$ masked attention

- Causal sequence: token X_t can only attend tokens $X_{\leq t}$
- Mask: binary, lower triangular matrix M

$$A = \sigma(Q^\top K) \odot M^\top$$

Universal approximation theorem

Transformers are universal sequence to sequence approximator [Yun et al., 2020]

Define the distance d_p as:

$$d_p(f, g) = \left(\int \|f(X) - g(X)\|_p^p dX \right)^{1/p}$$

Theorem: Let $1 \leq p \leq \infty$ and $\epsilon > 0$, then for any given $f \in \mathcal{F}$ the set of continuous functions that map a compact domain in $\mathbb{R}^{d \times n}$ to $\mathbb{R}^{d \times n}$, there exists a Polymorpher g with learned positional encoding such that $d_p(f, g) \leq \epsilon$.

Proof: Attention can map sequences to unique value, MLP can map unique input values to unique outputs.

Quadratic cost

Computing attention is $\mathcal{O}(n^2)$ with sequence length n

Fast approximation:

- Low rank approx: LinFormer [Wang et al., 2020]
- Hashing or nearest neighbor based: [Kitaev et al., 2020]
- Quadratic approx with closed form linearization: PoM [Picard and Dufour, 2024]

6.7 Time series, take home

AR:

- Linear model
- Online update
- Stationarity hypothesis

HMM:

- Models transition and emission probability
- Easier in discrete case (categorical distributions)
- Fitting the model is complex

RNN:

- Non-linear model (high capacity)
- Teacher forcing for training (OOD)
- Vanishing information

LSTM:

- Decision to add or remove from the hidden state
- Difficult to train
- Causality pb (decide what to keep before it is useful)

Transformers:

- State of the art for long term dependencies (eg, LLM)
- Memorizes all the past
- Attention cost quadratic $\rightarrow$ fast approximation (Linformer, Reformer, PoM, etc)

Chapter 7

Clustering

7.1 Unsupervised learning

What if we don't have labels?

- Training set $\mathcal{A} = \{\mathbf{x}\}$

What if we don't even have any idea about the structure of $\mathcal{Y}$?

- Density estimation (how likely is $\mathbf{x}$)
- Clustering (partition $\mathcal{X}$ into exclusive classes)

7.1.1 Density estimation

Given a training set of samples $\mathcal{A} = \{\mathbf{x}\}$, we want to model a probability density function $f(\mathbf{x})$ corresponding to the distribution D such that $\mathbf{x} \sim D$

- f has to be a pdf:
  - $\forall \mathbf{x} \in \mathcal{X}, f(\mathbf{x}) \geq 0$
  - $\int_{\mathcal{X}} f(\mathbf{x}) d\mathbf{x} = 1$

Useful for anomaly detection, e.g., if $f(\mathbf{x}) \leq \theta$ (Also called *Out of Distribution* detection)

- Simple pdf model, ex

$$f(\mathbf{x}) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{\|\mathbf{x}-\mu\|^2}{2\sigma^2}}$$

We have observations, they are likely by definition
Maximizing the probability of $\mathcal{A}$

$$\max_f Pr[\mathcal{A}] = \prod_{\mathbf{x}_i} f(\mathbf{x}_i)$$

Equivalently, maximizing the log probability

$$\max_f \log Pr[\mathcal{A}] = \sum_{\mathbf{x}_i} \log f(\mathbf{x}_i)$$

For the Gaussian

$$\sum_{\mathbf{x}_i} \log f(\mathbf{x}_i) = \sum_{\mathbf{x}_i} -\|\mathbf{x}_i - \mu\|^2 / 2\sigma^2 - n \log \sqrt{2\pi}\sigma$$

The maximimzation is equivalent to

$$\min_{\mu} \sum_{\mathbf{x}_i} \|\mathbf{x}_i - \mu\|^2$$

Which is attained for

$$\begin{aligned} \frac{\partial \sum_{\mathbf{x}_i} \|\mathbf{x}_i - \mu\|^2}{\partial \mu} &= 0 \\ 2n\mu - 2\sum_{\mathbf{x}_i} \mathbf{x}_i &= 0 \\ \mu &= \frac{1}{n} \sum_{\mathbf{x}_i} \mathbf{x}_i \end{aligned}$$

For σ

$$\min_{\sigma} \sum_{\mathbf{x}_i} \|\mathbf{x}_i - \mu\|^2 / 2\sigma^2 + n \log \sqrt{2\pi}\sigma$$

$$\begin{aligned} \frac{\partial \sum_{\mathbf{x}_i} \|\mathbf{x}_i - \mu\|^2 / 2\sigma^2 + n \log \sqrt{2\pi}\sigma}{\partial \sigma} &= 0 \\ -\frac{\sum_{\mathbf{x}_i} \|\mathbf{x}_i - \mu\|^2}{\sigma^3} + \frac{n}{\sigma} &= 0 \\ \sigma^2 &= \frac{\sum_{\mathbf{x}_i} \|\mathbf{x}_i - \mu\|^2}{n} \end{aligned}$$

7.1.2 Expectation-Maximization

Some pdf models do not lead to closed form solution (case of mixture models) Alternate optimisation scheme

Initialize model parameters Γ_0

- Expectation step: Assumming parameters Γ_t, compute *expected* likelihood (or log-likelihood) of f w.r.t. $\mathcal{A}$
- Maximization step: Maximize this expected likelihood of f w.r.t. Γ

7.1.3 Gaussian Mixture model

$$f(\mathbf{x}) = \sum_k \pi_k e^{(\mathbf{x}-\mu_k)^\top \Gamma_k (\mathbf{x}-\mu_k)}$$

π_k is the weight (population) , μ_k the mean and Γ_k is the inverse covariance matrix of component k An observation $\mathbf{x}$ belongs to component k with likelihood $f_k(\mathbf{x})$ E step

$$E[\log f(\mathbf{x}_i)] = \sum_k Pr[k|\mathbf{x}_i] \log f_k(\mathbf{x}_i)]$$

With

$$Pr[k|\mathbf{x}_i] = h_i^k = \frac{\pi_k e^{(\mathbf{x}_i-\mu_k)^\top \Gamma_k (\mathbf{x}_i-\mu_k)}}{\sum_j \pi_j e^{(\mathbf{x}_i-\mu_j)^\top \Gamma_j (\mathbf{x}_i-\mu_j)}}$$

M step:

$$\max_{\pi,\mu,\Gamma} E[\log f(\mathbf{x})] = \sum_i \sum_k h_i^k \log f_k(\mathbf{x}_i)$$

$$\pi_k = \frac{\sum_i h_i^k}{n}$$

$$\mu_k = \frac{\sum_i h_i^k \mathbf{x}_i}{\sum_i h_i^k}$$

$$\Gamma_k^{-1} = \frac{\sum_i h_i^k (\mathbf{x}_i - \mu_k)(\mathbf{x}_i - \mu_k)^\top}{\sum_i h_i^k}$$

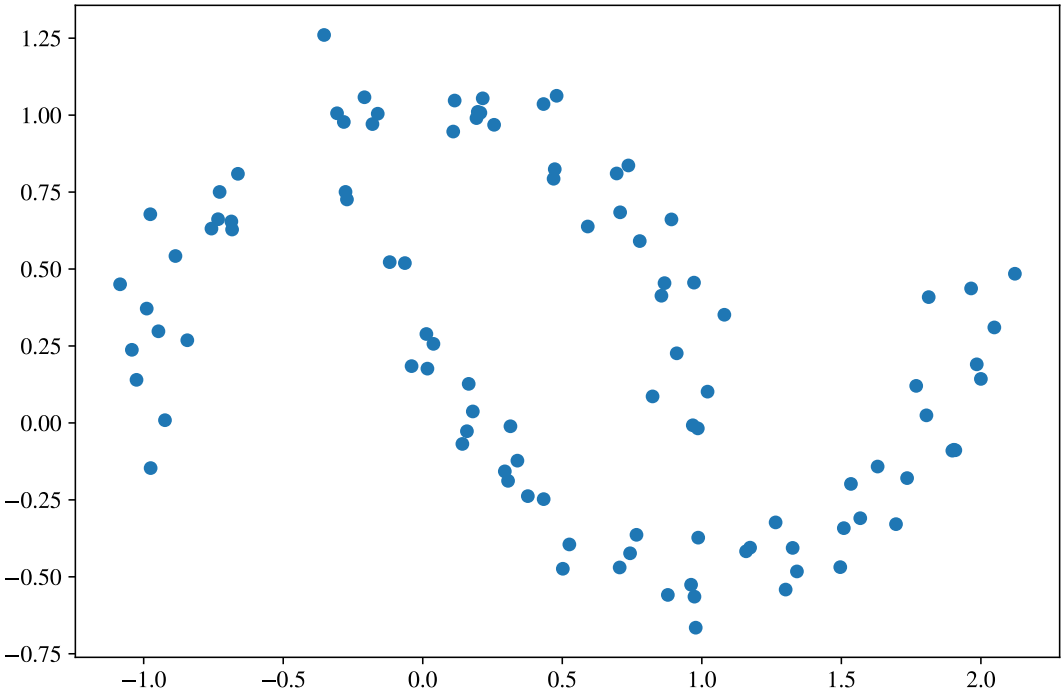
In [2]:

```
from sklearn.datasets import make_moons
```

In [3]:

```
X, y = make_moons(100, noise=0.1)
plt.scatter(X[:,0], X[:,1])
```

```
<matplotlib.collections.PathCollection at 0x73ac154a5090>
```



In [4]:

```
def Estep(X, w, mu, s2): # nx5, 5, 5x2, 5x2
    xmu = X[:, None, :] - mu[None, :, :] # n x 5 x 2
    s2xm = xmu/(1e-7+s2[None,:,:]) # n x 5 x 2
    xmsxm = w * jnp.exp(-(xmu * s2xm).sum(2)) # n x 5
    return xmsxm # n x 5

def Mstep(X, h): # nx2, nx5
    h = h / h.sum(axis=1, keepdims=True) # normalize likelihood
    w = h.mean(axis=0, keepdims=True) # n x 5 -> 1x5
    m = (h[:,:,None]*X[:,None,:]).sum(axis=0) / h.sum(axis=0)[:,None] # 5 x 2
    xm = X[:,None,:] - m[None,:,:] # n x 5 x 2
    s = (h[:,:,None]*xm*xm).sum(axis=0) / h.sum(axis=0)[:,None] # 5 x 2
    return w, m, s
```

In [5]:

```
np.random.seed(4)
w = np.ones((1, 5))/5.
m = 0.5*np.random.randn(5, 2)+0.5
s = 0.5*jnp.ones((5,2))
```

In [6]:

```
def plot_density(X, w, m, s):
    t = 50; tx = jnp.linspace(-2, 3, t); ty = jnp.linspace(-2, 2, t)
    xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
    xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
    y_pred = jnp.array(Estep(xx,w,m,s)).sum(axis=1).reshape(t, t)
    plt.contourf(xv, yv, y_pred, cmap='coolwarm')
    cs = plt.contour(xv, yv, y_pred, colors='k')
    plt.clabel(cs, inline=True)
    plt.scatter(X[:,0], X[:,1], c=0*jnp.ones(len(X)))
    plt.scatter(m[:,0], m[:,1], c=1+jnp.arange(len(m)), marker='^',
s=150*(1+s.sum(axis=1)), edgecolors='k')
```

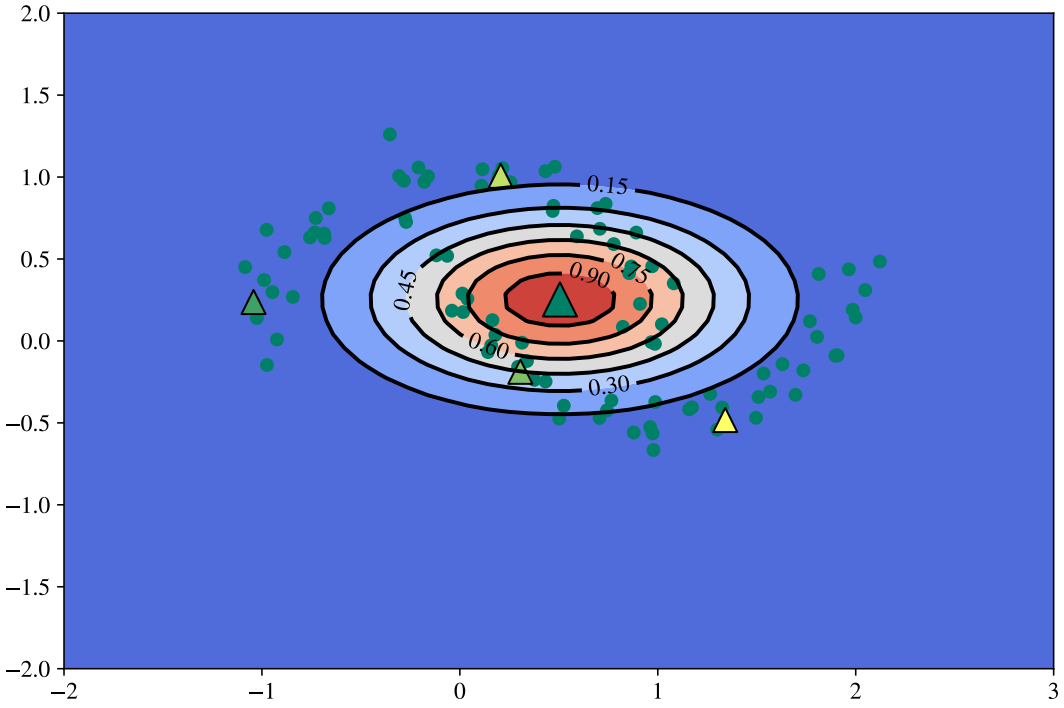
In [7]:

```
fig = plt.figure(dpi=150)
plt.axis('off')
camera = Camera(fig)
for i in range(30):
    plot_density(X, w, m, s)
    h = Estep(X, w, m, s)
    w, m, s = Mstep(X, h)
    plt.axis([-2, 3, -2, 2])
    camera.snap()

animation = camera.animate()
HTML(animation.to_html5_video())
```

```
<IPython.core.display.HTML object>
```

In [8]:

```
plot_density(X, w, m, s)
```



7.1.4 One class SVM

Given a training set $\mathcal{A} = \{\mathbf{x}_i\}$ and a kernel $k(\cdot,\cdot) = \langle \phi(\cdot), \phi(\cdot) \rangle$, we want to find a classifier of high density regions:

$$\min_{\mathbf{w},\xi,\rho} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i \xi_i - \rho$$

$$\text{s.t. } \forall i, \langle \mathbf{w}, \phi(\mathbf{x}_i) \rangle \geq \rho - \xi_i$$

$$\forall i, \xi_i \geq 0$$

Using KKT:

$$\mathbf{w} = \sum_i \alpha_i \phi(\mathbf{x}_i)$$

$$\sum_i \alpha_i = 1$$

$$0 \leq \alpha_i \leq C$$

Dual problem:

$$\max_{\alpha} \frac{1}{2} \sum_{ij} \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j)$$

$$\text{s.t. } \forall i, 0 \leq \alpha_i \leq C$$

$$\sum_i \alpha_i = 1$$

Solved using any QP solver (or projected coordinate ascent) Recovering ρ
KKT, complementary slackness:

$$\forall i, \lambda_i (\rho - \xi_i - \sum_{ij} \alpha_j k(\mathbf{x}_i, \mathbf{x}_j)) = 0$$

if $\alpha_i \neq 0$ and $\alpha_i \neq C$

$$\xi_i = 0$$

and

$$\lambda_i \neq 0$$

Thus

$$\rho = \sum_j \alpha_j k(\mathbf{x}_i, \mathbf{x}_j)$$

In [9]:

```
def gauss_kernel(X1, X2, gamma=5.):
    D = (X1[:,None,:] - X2[None,:,:])**2
    return jnp.exp(-gamma*D.sum(axis=2))
```

In [10]:

```
def loss(alpha, X):
    K = gauss_kernel(X, X)
    return 0.5 * (alpha[None,:] @ (K @ alpha[:,None])).squeeze()
```

In [11]:

```
@jax.jit
def update(alpha, X, C = 1., eta=0.1):
    da = jax.grad(loss, argnums=0)(alpha, X)
    a = jnp.clip(alpha + eta*da, 0, C)
    return a/a.sum()
```

In [12]:

```
t = 50; tx = jnp.linspace(-2, 3, t); ty = jnp.linspace(-2, 2, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
```

In [13]:

```
def plot_density(X, xx, y_pred):
    plt.contourf(xv, yv, y_pred, cmap='coolwarm')
    cs = plt.contour(xv, yv, y_pred, colors='k')
    plt.clabel(cs, inline=True)
    plt.scatter(X[:,0], X[:,1], c=jnp.zeros(100))
```

In [14]:

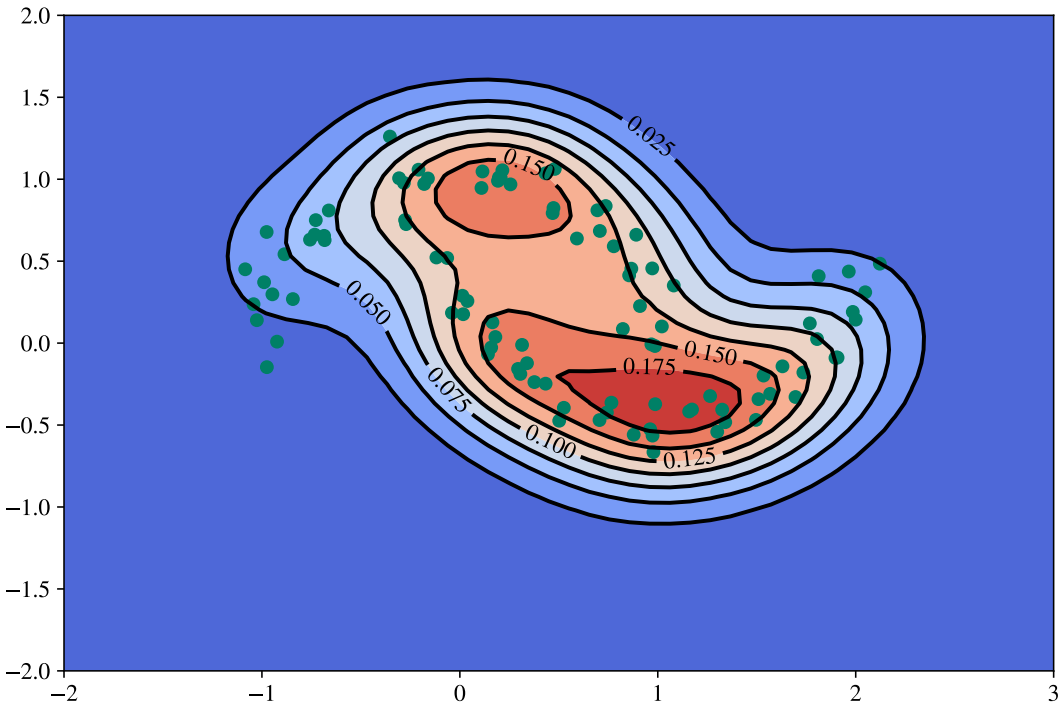
```
fig = plt.figure(dpi=150); plt.axis('off'); camera = Camera(fig)
alpha = np.ones(100)/100; l = []
for i in range(200):
    y_pred = gauss_kernel(xx, X)@alpha[:,None]
    y_pred = jnp.array(y_pred).reshape(t, t)
    plot_density(X, xx, y_pred)
```

```
    camera.snap()
    alpha = update(alpha, X, C=0.02, eta=.04)
    l.append(loss(alpha, X))
animation = camera.animate()
HTML(animation.to_html5_video())
```

```
<IPython.core.display.HTML object>
```

In [15]:

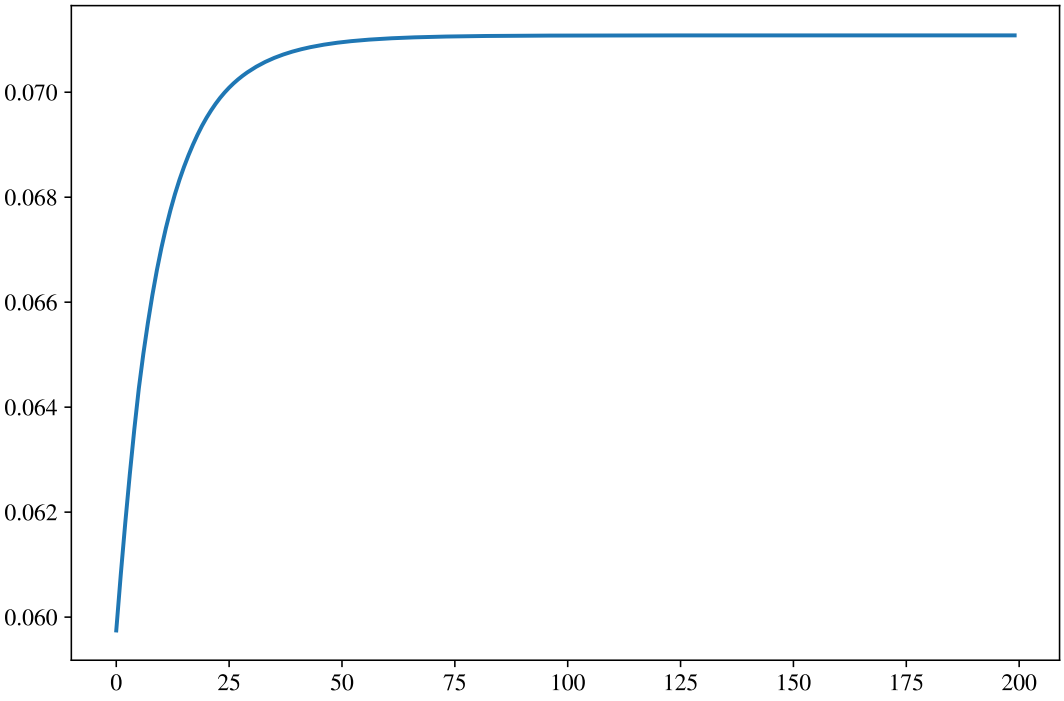
```
plot_density(X, xx, y_pred)
```



In [16]:

```
plt.plot(l)
```

```
[<matplotlib.lines.Line2D at 0x73abe03fe3e0>]
```



7.1.5 Exercice

Code a GMM and run it on MNIST, plot the means as images.

7.2 Clustering

If we have a mixture model, could we use the likelihood of each component as a categorization prediction? Yes, but not the objective function (i.e., no competition between classes)

Better use a dedicated algorithm

7.2.1 k-means

Categorization by approximation

Define M partitions C_k of the training set $\mathcal{A}$ and their associated predictor μ_k

$$\min_{C_k,\mu_k} \sum_k \sum_{\mathbf{x}\in C_k} \|\mathbf{x}-\mu_k\|^2$$

Alternate stepest descent between C_k and μ_k

Stepest descent on C_k

$$\min_{C_k,\cup C_k=\mathcal{A}} \sum_k \sum_{\mathbf{x}\in C_k} \|\mathbf{x}-\mu_k\|^2$$

Attained with nearest nearbor assignment

$$C_k = \{\mathbf{x}\in\mathcal{A} | \mu_k = \text{argmin}_c \|\mathbf{x}-\mu_c\|^2\}$$

Corresponds to an E step in EM with

$$h_i^k = \mathbb{1}_{[\mu_k=\text{argmin}_c\|\mathbf{x}-\mu_c\|^2]}$$

Stepest descent on μ_k

$$\min_{\mu_k} \sum_{\mathbf{x}\in C_k} \|\mathbf{x}-\mu_k\|^2$$

Attained for the barycenter

$$\mu_k = \frac{1}{|C_k|}\sum_{\mathbf{x}\in C_k}\mathbf{x}$$

Corresponds to an M step in EM

7.2.2 Kernel k-means

Using a kernel $k(\mathbf{x}_i,\mathbf{x}_j) = \langle\phi(\mathbf{x}_i),\phi(\mathbf{x}_j)\rangle$

$$\min_{C_k,\mu_k} \sum_k \sum_{\mathbf{x}\in C_k} \|\phi(\mathbf{x})-\mu_k\|^2$$

Representer theorem

$$\mu_k = \sum_i \alpha_i \phi(\mathbf{x}_i)$$

$$\|\phi(\mathbf{x})-\mu_k\|^2 = k(\mathbf{x},\mathbf{x}) + \sum_{ij}\alpha_i\alpha_j k(\mathbf{x}_i,\mathbf{x}_j) - 2\sum_i \alpha_i k(\mathbf{x},\mathbf{x_i})$$

C_k step unchanged μ_k step, note that

$$\mu_k = \frac{1}{|C_k|} \sum_{\mathbf{x} \in C_k} \phi(\mathbf{x})$$

Thus

$$\alpha_i = \begin{cases} 1/|C_k| \text{if } \mathbf{x} \in C_k \\ 0, \text{else} \end{cases}$$

In [17]:

```
def Estep(X, mu):
    D = ((X[:,None,:] - mu[None,:,:])**2).sum(axis=2) # n x M
    return D.argmin(axis=1)
def Mstep(X, h):
    h = jax.nn.one_hot(h, num_classes=25)
    mu = (X[:,None,:]*h[:,:,None]).sum(axis=0)/(1e-5+h[:,:,None].sum(axis=0))
    return mu
```

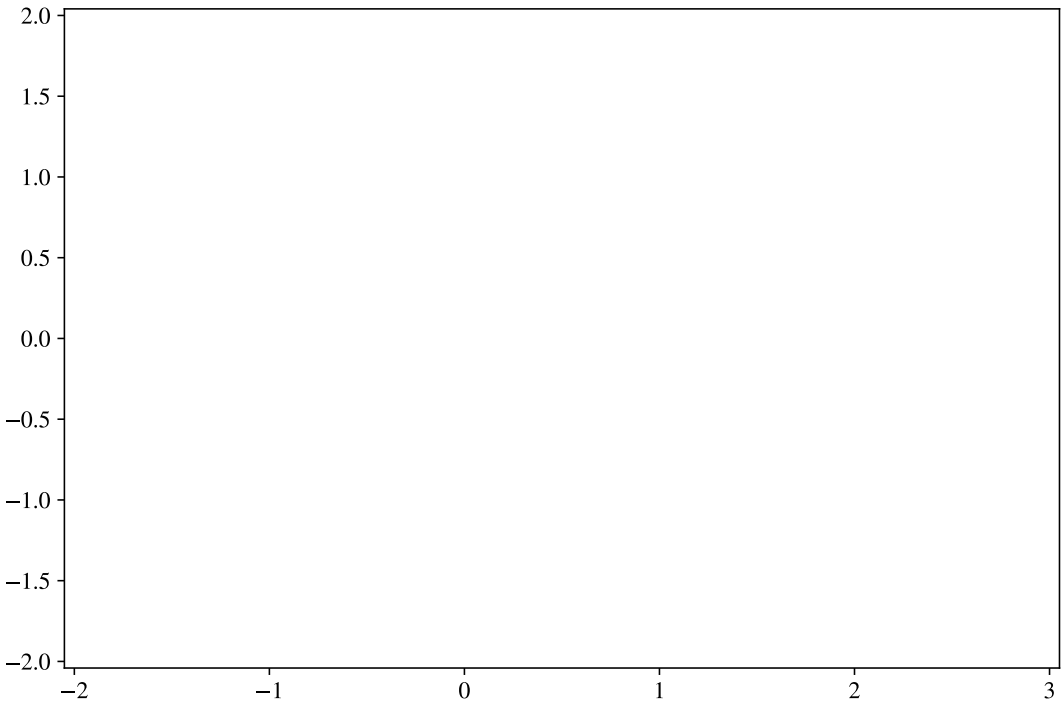
In [18]:

```
def plot_km(X, xv, yv, y_pred, mu):
    plt.pcolormesh(xv, yv, y_pred, shading='auto', aa=True)
    plt.scatter(mu[:,0], mu[:,1], c=np.arange(25), marker='^', s=150, edgecolors='k')
    plt.scatter(X[:,0], X[:,1], c=h, edgecolors='k')
```

In [19]:

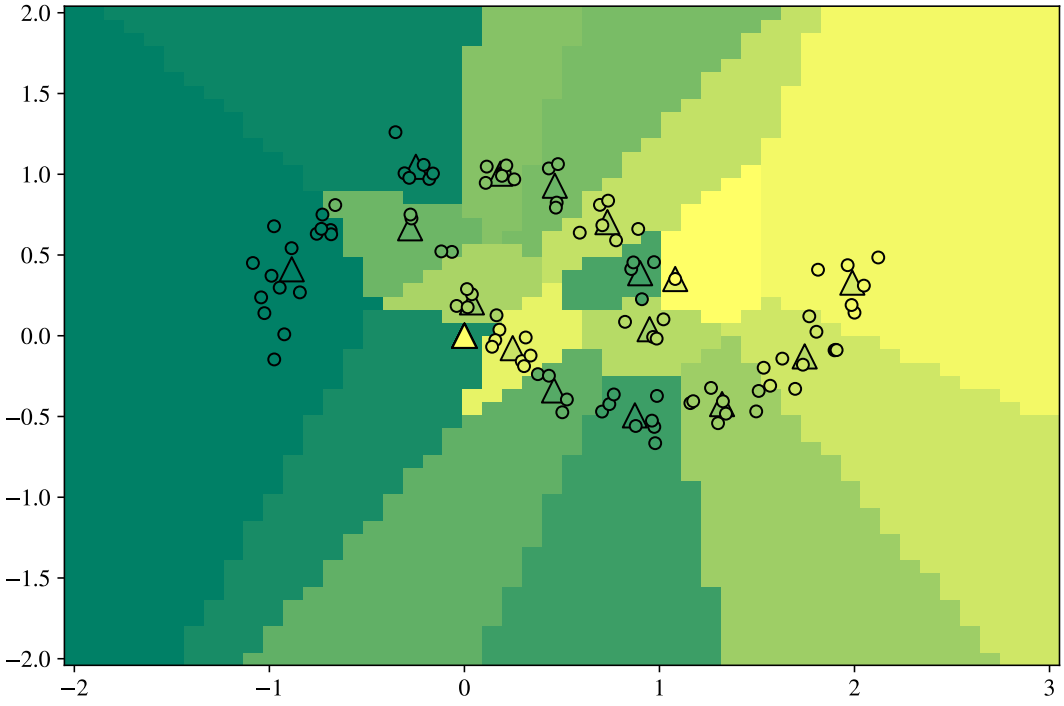
```
fig = plt.figure(dpi=150)
camera = Camera(fig)
mu = 2*np.random.random((25, 2))-0.5
for e in range(20):
    h = Estep(X, mu)
    y_pred = Estep(xx, mu)+1
    y_pred = jnp.array(y_pred).reshape(t, t)
    plot_km(X, xv, yv, y_pred, mu)
    camera.snap()
    mu = Mstep(X, h)
    plot_km(X, xv, yv, y_pred, mu)
    camera.snap()
animation = camera.animate(interval=700)
HTML(animation.to_html5_video())
```

```
<IPython.core.display.HTML object>
```



In [20]:

```
plot_km(X, xv, yv, y_pred, mu)
```



7.2.3 Exercise

Code k-Means and run it on MNIST, plot the centroids as images

Chapter 8

Introduction to PAC Learning

8.1 Probably Approximately Correct

Can we obtain formal guaranties in Learning? Formal model of learnability [Valiant, 1984]

- Domain set $\mathcal{X}$: observations with distribution $\mathcal{D}$
- Label set $\mathcal{Y}$: target of prediction
- Concept $f : \mathcal{X} \to \mathcal{Y}$, data generation process
- Hypothesis $h : \mathcal{X} \to \mathcal{Y}$, prediction function
- Hypothesis class $\mathcal{H} = \{h\}$, set of hypotheses
- Error $L_{\mathcal{D},f}(h) = Pr_{x\sim\mathcal{D}}[h(x) \neq f(x)]$

8.1.1 Empirical Risk Minimization

- $\mathcal{D}$ is unknown, and so is $L_{\mathcal{D},f}(h)$
- Training set $\mathcal{A} = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ sampled i.i.d from $\mathcal{D}$ and labeled with f
- Empirical risk: $L_{\mathcal{A}}(h) = \frac{1}{m}\sum_i [h(x_i) \neq y_i]$

ERM principle:

$$h_{\mathcal{A}} = \text{argmin}_{h\in\mathcal{H}} L_{\mathcal{A}}(h)$$

Realizability assumption

Definition (Realizability assumption): There exists $h^\star \in \mathcal{H}$ such that $L_{\mathcal{D},f}(h^\star) = 0$ Remark that with probability 1 over a random set $\mathcal{A} = \{(x_i, y_i)\}, x_i \sim \mathcal{D}$ and $y_i = f(x_i)$, we have $L_{\mathcal{A}}(h^\star) = 0$

8.1.2 ERM Failures?

Given the realizability assumption, what is the probability that the ERM fails? Bounding the generalization risk

$$Pr_{\mathcal{A}}[L_{\mathcal{D},f}(h_{\mathcal{A}}) > \epsilon] \leq \delta$$

Probably (δ) Approximately (ϵ) Correct

8.1.3 Generalization bound

- Bad hypotheses set

$$\mathcal{H}_B = \{h \in \mathcal{H} | L_{\mathcal{D},f}(h) > \epsilon\}$$

- Misleading training sets

$$M = \{\mathcal{A} | \exists h \in \mathcal{H}_B, L_{\mathcal{A}}(h) = 0\}$$

Set of training sets that appear to be good ($L_{\mathcal{A}}(h_{\mathcal{A}}) = 0$) but are bad in reality ($L_{\mathcal{D},f}(h_{\mathcal{A}}) > \epsilon$)

We want to know the probability that the ERM fails because we have sampled a misleading dataset Let us construct M

- We follow the ERM
- We have the realizability assumption
- Misleading training sets achieve zero empirical risk for bad classifiers

$$M = \cup_{h \in \mathcal{H}_B} \{\mathcal{A} | L_{\mathcal{A}}(h) = 0\}$$

Remark that

$$\{\mathcal{A} | L_{\mathcal{D},f}(h_{\mathcal{A}}) > \epsilon\} \subseteq M$$

(a training set that leads to ϵ generalization error is necessarily a misleading training set beacuse of realizability) Combining the two using an union bound

$$Pr_{\mathcal{A}}[\mathcal{A} | L_{\mathcal{D},f}(h_{\mathcal{A}}) > \epsilon] \leq \sum_{h \in \mathcal{H}_B} Pr_{\mathcal{A}}[\mathcal{A} | L_{\mathcal{A}}(h) = 0]$$

Elements of $\mathcal{A}$ are sampled i.i.d

$$Pr_{\mathcal{A}}[\mathcal{A} | h \in \mathcal{H}_B, L_{\mathcal{A}}(h) = 0] = \prod_{i=1}^{m} Pr[h(x_i) = f(x_i)]$$

Since $h \in \mathcal{H}_B$, $Pr[h(x_i) = f(x_i)] \leq 1 - \epsilon$, thus

$$Pr_{\mathcal{A}}[\mathcal{A} | h \in \mathcal{H}_B, L_{\mathcal{A}}(h) = 0] \leq (1 - \epsilon)^m \leq e^{-\epsilon m}$$
$$Pr_{\mathcal{A}}[\mathcal{A} | L_{\mathcal{D},f}(h_{\mathcal{A}}) > \epsilon] \leq |\mathcal{H}_B| e^{-\epsilon m} \leq |\mathcal{H}| e^{-\epsilon m}$$

Theorem Let $\mathcal{H}$ be a finite hypothesis class, $\delta \in [0, 1]$, $\epsilon > 0$ and m such that

$$m \geq \frac{\log(|\mathcal{H}|/\delta)}{\epsilon}$$

Then, for any labeling function f, and on any distribution $\mathcal{D}$ for which the realizability assumption holds, we have for every ERM hypothesis $h_{\mathcal{A}}$

$$Pr_{\mathcal{A}}[L_{\mathcal{D},f}(h_{\mathcal{A}}) \leq \epsilon] \geq 1 - \delta$$

8.1.4 PAC Learning

Definition (PAC Learnability): A hypothesis class $\mathcal{H}$ is PAC learnable if $\exists m_{\mathcal{H}} : [0,1]^2 \to \mathbb{N}$ and a learning algorithm such that $\forall \epsilon, \delta \in [0,1]^2, \forall \mathcal{D}$ over $\mathcal{X}, \forall f : \mathcal{X} \to \{0,1\}$, if the realizability assumption holds w.r.t. $\mathcal{H}, \mathcal{D}, f$, then running the algorithm on $m \geq m_{\mathcal{H}}(\epsilon, \delta)$ i.i.d. samples generated by $\mathcal{D}$ and labeled by f, the algorithm returns h such that with probability at least $1 - \delta$, $L_{\mathcal{D},f}(h) \leq \epsilon$

Theorem Every finite hypothesis class is PAC learnable with sample complexity

$$m_{\mathcal{H}}(\epsilon, \delta) \leq \left\lceil \frac{\log(|\mathcal{H}|/\delta)}{\epsilon} \right\rceil$$

8.1.5 Fundamental theorem of PAC Learning

Theorem Let $\mathcal{H}$ be a hypothesis class over $\mathcal{X} \to \{0,1\}$ and using the 0-1 loss function, then the following are equivalent

1. $\mathcal{H}$ is PAC learnable
2. Any ERM rule is a successful PAC learner for $\mathcal{H}$
3. $\mathcal{H}$ has finite VC dimension

8.1.6 No Free Lunch Theorem

Theorem Let A be any learning algorithm for the task of binary classification with the 0-1 loss over $\mathcal{X}$, Let $m < |\mathcal{X}|/2$ be a training set size. Then there exists a distribution over $\mathcal{X}$ and a concept f such that:

1. $\exists h : \mathcal{X} \to \{0,1\}, L_{\mathcal{D},f}(h) = 0$ (there is a good hypothesis)
2. With probability at least $1/7$ over the choice of $\mathcal{A} \sim \mathcal{D}$, we have $L_{\mathcal{D},f}(A(\mathcal{A})) \geq 1/8$ (the algorithm does not find it)

No universal learner

8.2 Clustering, Metric learning and PAC, take home

Unsupervised learning

- Density estimation: how likely is an example
- Clustering: partitioning $\mathcal{X}$ into arbitrarily chosen classes
- EM is a powerful algorithm for DE and clustering, but sensitive to init
- k-means shows up frequently as a basic tool (many improved version: splitting init, codeword shifting, etc)
- k-means is an excellent init for GMM

Metric learning

- ML algorithm dependent on the natural distance on $\mathcal{X}$

- can be improved by using a better kernel (metric in the induced space)
- Learning the metric can turn hard learning problem into easy ones
- kNN with metric learning often has a good complexity/accuracy trade-off

PAC

- Formal study of learnability without specifying the data distribution or the type of hypothesis
- Finite classes are learnable
- But bounds have unrealistic number of samples
- No universal learner: some algorithms are better at some problems than others

Bibliography

[Aharon et al., 2006] Aharon, M., Elad, M., and Bruckstein, A. (2006). K-svd: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Transactions on Signal Processing*, 54(11):4311–4322.

[Azencott, 2022] Azencott, C.-A. (2022). *Introduction au Machine Learning-2e éd.* Dunod.

[Baum et al., 1970] Baum, L. E., Petrie, T., Soules, G., and Weiss, N. (1970). A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The annals of mathematical statistics*, 41(1):164–171.

[Belkin et al., 2019] Belkin, M., Hsu, D., Ma, S., and Mandal, S. (2019). Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854.

[Boser et al., 1992] Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152.

[Breiman, 2001] Breiman, L. (2001). Random forests. *Machine learning*, 45:5–32.

[Breiman et al., 1984] Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). Classification and regression trees.

[Chen and Guestrin, 2016] Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794.

[Cortes and Vapnik, 1995] Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20:273–297.

[Cover and Hart, 1967] Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27.

[Cybenko, 1989] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

[Freund and Schapire, 1996] Freund, Y. and Schapire, R. E. (1996). Experiments with a new boosting algorithm. In *Machine Learning, Proceedings of the Thirteenth International Conference (ICML '96), Bari, Italy, July 3-6, 1996*, pages 148–156.

[Grinsztajn et al., 2022] Grinsztajn, L., Oyallon, E., and Varoquaux, G. (2022). Why do tree-based models still outperform deep learning on typical tabular data? *Advances in Neural Information Processing Systems*, 35:507–520.

[Hastie et al., 2009] Hastie, T., Tibshirani, R., Friedman, J. H., and Friedman, J. H. (2009). *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Hornik et al., 1989] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

[Jacot et al., 2018] Jacot, A., Gabriel, F., and Hongler, C. (2018). Neural tangent kernel: Convergence and generalization in neural networks. *Advances in neural information processing systems*, 31.

[Kitaev et al., 2020] Kitaev, N., Kaiser, L., and Levskaya, A. (2020). Reformer: The efficient transformer. In *International Conference on Learning Representations*.

[Malach et al., 2020] Malach, E., Yehudai, G., Shalev-Schwartz, S., and Shamir, O. (2020). Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*, pages 6682–6691. PMLR.

[Murphy, 2022] Murphy, K. P. (2022). *Probabilistic machine learning: an introduction*. MIT press.

[Murphy, 2023] Murphy, K. P. (2023). *Probabilistic machine learning: Advanced topics*. MIT press.

[Pelillo, 2014] Pelillo, M. (2014). Alhazen and the nearest neighbor rule. *Pattern Recognition Letters*, 38:34–37.

[Picard, 2021] Picard, D. (2021). Torch. manual_seed (3407) is all you need: On the influence of random seeds in deep learning architectures for computer vision. *arXiv preprint arXiv:2109.08203*.

[Picard and Dufour, 2024] Picard, D. and Dufour, N. (2024). Pom: Efficient image and video generation with the polynomial mixer.

[Rabiner, 1989] Rabiner, L. (1989). A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286.

[Rakotomamonjy et al., 2005] Rakotomamonjy, A., Canu, S., and Smola, A. (2005). Frames, reproducing kernels, regularization and learning. *Journal of Machine Learning Research*, 6(9).

[Schölkopf and Smola, 2002] Schölkopf, B. and Smola, A. J. (2002). *Learning with kernels: support vector machines, regularization, optimization, and beyond*.

[Shalev-Shwartz and Ben-David, 2014] Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding machine learning: From theory to algorithms*. Cambridge university press.

[Shalev-Shwartz and Zhang, 2013] Shalev-Shwartz, S. and Zhang, T. (2013). Stochastic dual coordinate ascent methods for regularized loss minimization. *Journal of Machine Learning Research*, 14(1).

[Tibshirani, 1996] Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 58(1):267–288.

[Valiant, 1984] Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142.

[Vapnik, 1995] Vapnik, V. N. (1995). *The nature of statistical learning theory*. Springer-Verlag.

[Viterbi, 1967] Viterbi, A. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269.

[Wang et al., 2020] Wang, S., Li, B. Z., Khabsa, M., Fang, H., and Ma, H. (2020). Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*.

[Weinberger and Saul, 2009] Weinberger, K. Q. and Saul, L. K. (2009). Distance metric learning for large margin nearest neighbor classification. *Journal of machine learning research*, 10(2).

[Williams and Seeger, 2000] Williams, C. and Seeger, M. (2000). Using the nyström method to speed up kernel machines. *Advances in neural information processing systems*, 13.

[Yang et al., 2012] Yang, T., Li, Y.-F., Mahdavi, M., Jin, R., and Zhou, Z.-H. (2012). Nyström method vs random fourier features: A theoretical and empirical comparison. *Advances in neural information processing systems*, 25.

[Yun et al., 2020] Yun, C., Bhojanapalli, S., Rawat, A. S., Reddi, S., and Kumar, S. (2020). Are transformers universal approximators of sequence-to-sequence functions? In *International Conference on Learning Representations*.