Machine Learning Algorithms

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

February 21, 2022

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. 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 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Also, from a technical point of view, the slides were originally made in python notebooks.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

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 low level 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...

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

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

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

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

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.

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 $E(f)$

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

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 $\mathcal{A}$

1.3.2 A not-as-bad example

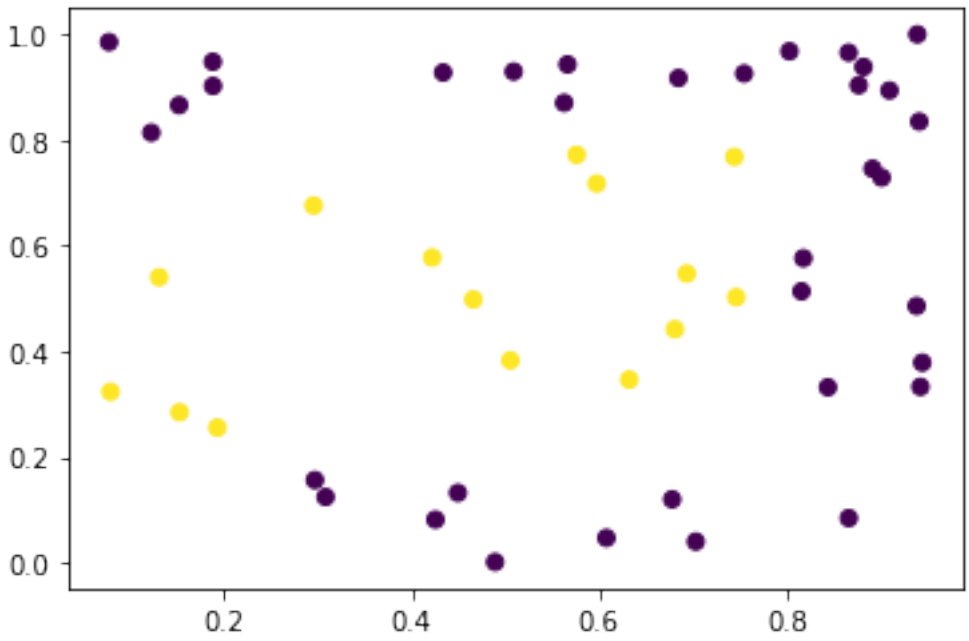
Points inside a random circle

```
In [2]: 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 [3]: 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)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and
rerun for more info.)
```

```
<matplotlib.collections.PathCollection at 0x7f09a5de3290>
```

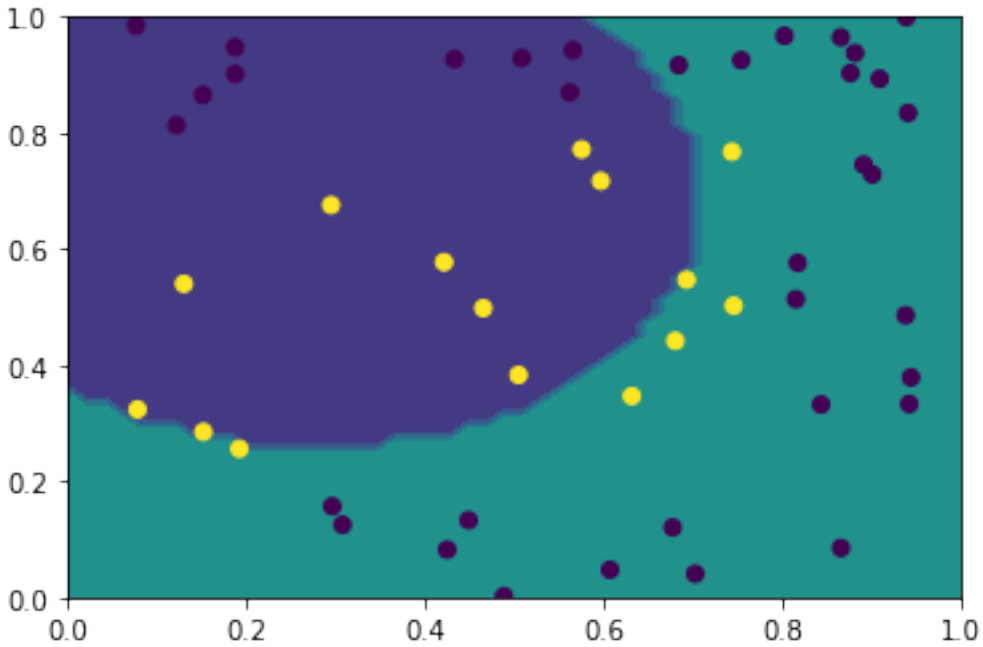



```
In [4]: class CirclePredictor:
            def __init__(self, key):
                key, skey = jax.random.split(key)
                self.c = jax.random.uniform(key, (2,))
                self.r = jax.random.uniform(skey)
            def __call__(self, X):
                return jnp.sign(1*((X[:,0] - 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 [5]:
```
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 0x7f09a03479d0>
```

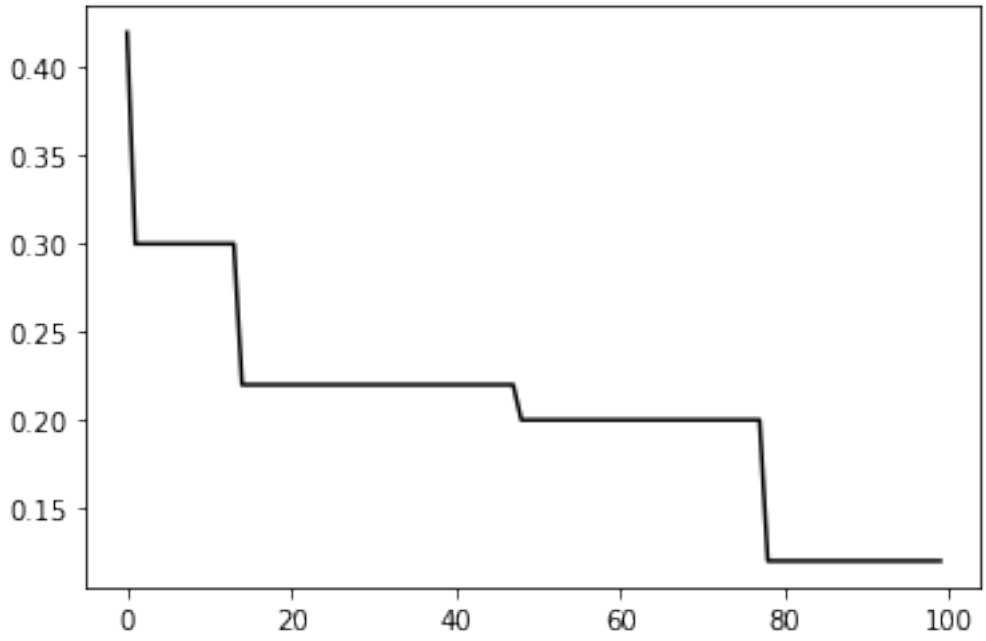


1.3.3 (Randomly) Searching for a good f

In [6]:
```
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())
```

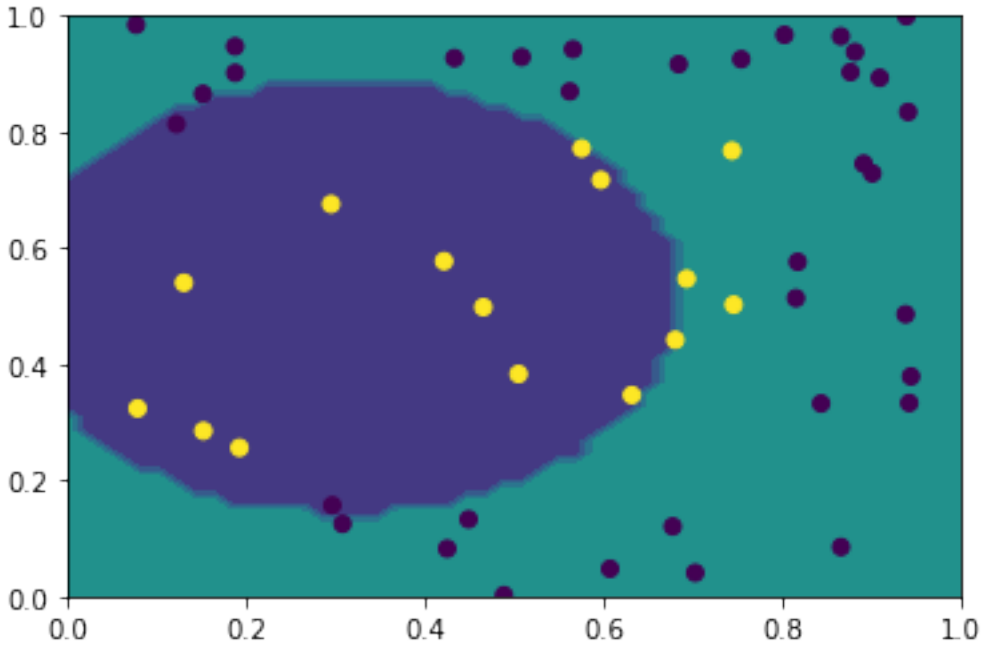
```
<IPython.core.display.HTML object>
```



1.3.4 Are we lucky

```
In [7]: 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 0x7f09a033a910>
```



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*

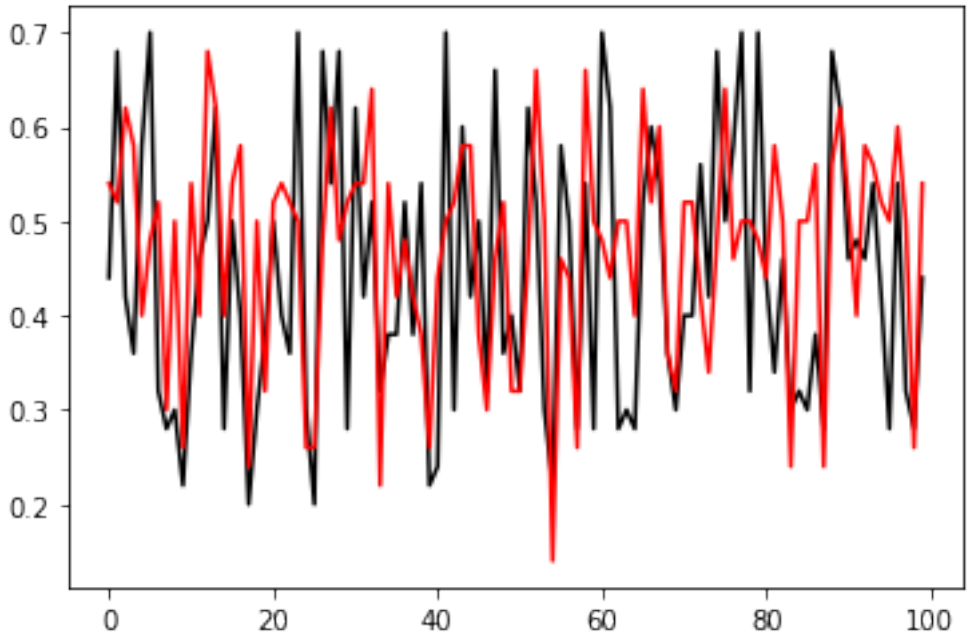
How do we know if f is overfitting? - Measuring the error on $\mathcal{A}$ is not informative $\Rightarrow$ Split the examples into training and evaluation sets

1.4.1 Let's try

```
In [8]: key = jax.random.PRNGKey(6)
        Xt = jax.random.uniform(key, (50, 2))
        yt = gt(Xt)

In [9]: 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 $1 - NN$:

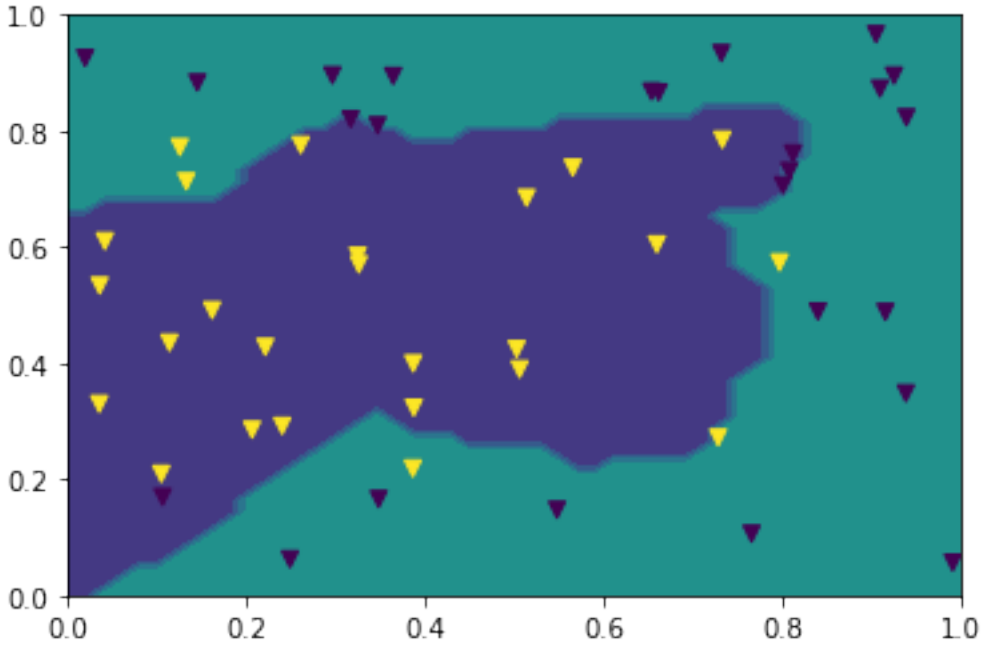
$$f(x) = y_i \text{ s.t. } i = \arg\min_{x_j \in \mathcal{A}} \|x - x_j\|^2$$

- Memorizes the entire training set
- Does 0 empirical error on $\mathcal{A}$

```
In [10]: 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
         dim
                 index = jnp.argmin(dist, axis=1)
                 return y[index]
```

```
In [11]: 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 0x7f09e1c05910>
```



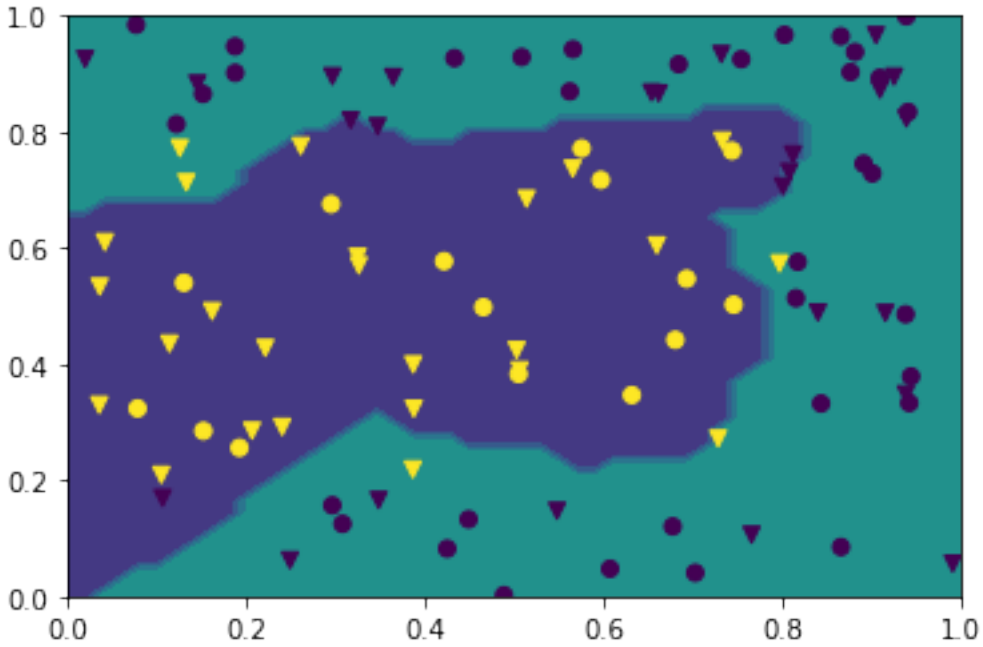
1.5.1 What is the effect of k?

```
In [12]: 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
         dim
                 indices = jnp.argsort(dist, axis=1)
                 yp = 1*((self.y[indices[:,0:self.k]]).sum(axis=1) > self.k//2)
                 return yp
```

```
In [13]: nn = KNearestNeighbor(X, y, k=1)
         t = 50
         tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
```

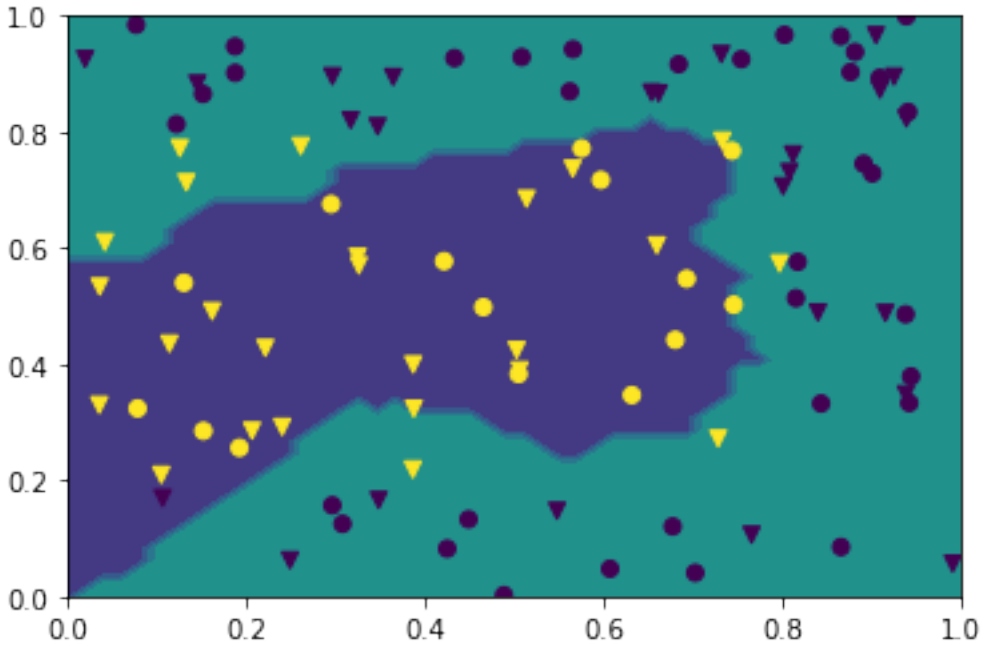
```
        xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
        xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
        levels=jnp.linspace(-1.5, 1.5, 10)
        y_pred = nn(xx).reshape(t, t)
        plt.contourf(xv, yv, -y_pred, levels=levels)
        plt.scatter(X[:,0],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

```
(<matplotlib.collections.PathCollection at 0x7f09e1ff8a10>,
 <matplotlib.collections.PathCollection at 0x7f09e1ff8b50>)
```



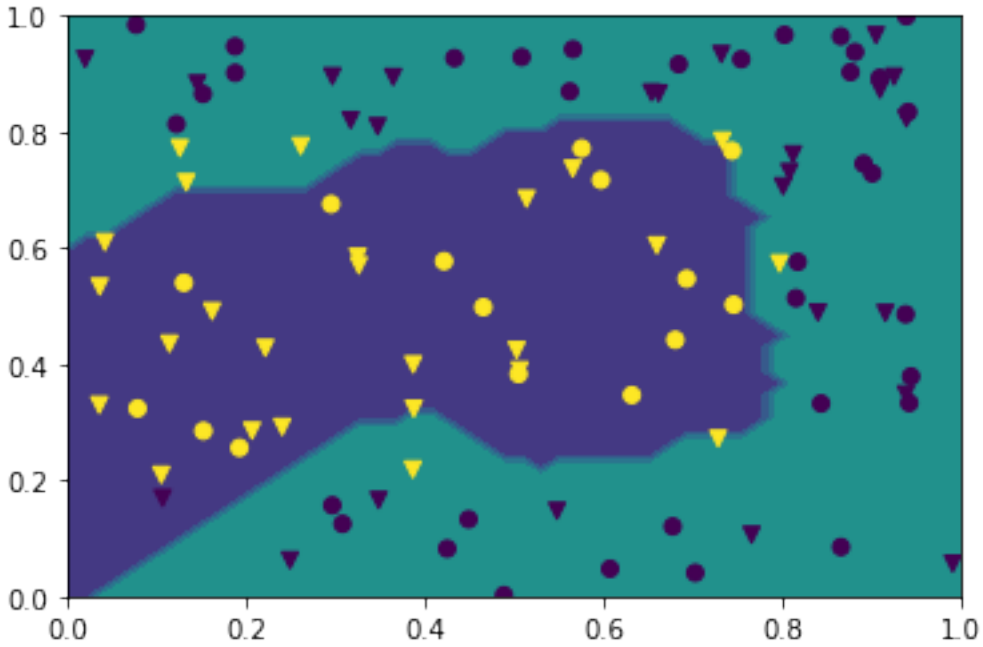

```
In [14]: nn = KNearestNeighbor(X, y, k=2)
        t = 50
        tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
        xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
        xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
        levels=jnp.linspace(-1.5, 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 0x7f09e1ff8f90>,
 <matplotlib.collections.PathCollection at 0x7f0988784a10>)
```



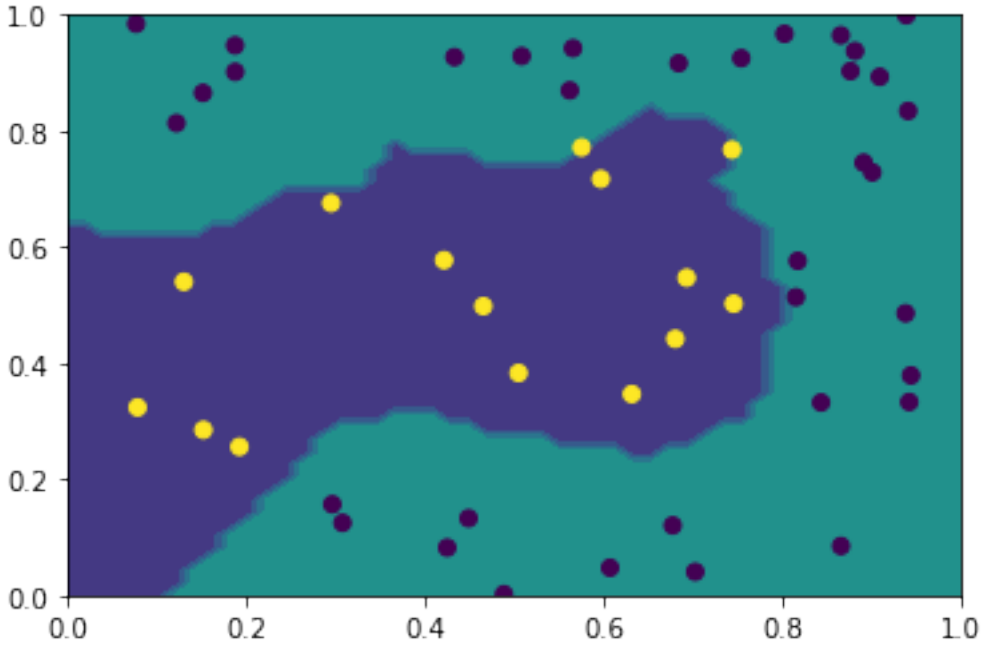

```
In [15]: nn = KNearestNeighbor(X, y, k=3)
         t = 50
         tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
         xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
         xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
         levels=jnp.linspace(-1.5, 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 0x7f09e1ff4d90>,
 <matplotlib.collections.PathCollection at 0x7f098878a350>)
```



In [16]:
```
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 0x7f09887f6b10>
```



1.5.2 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

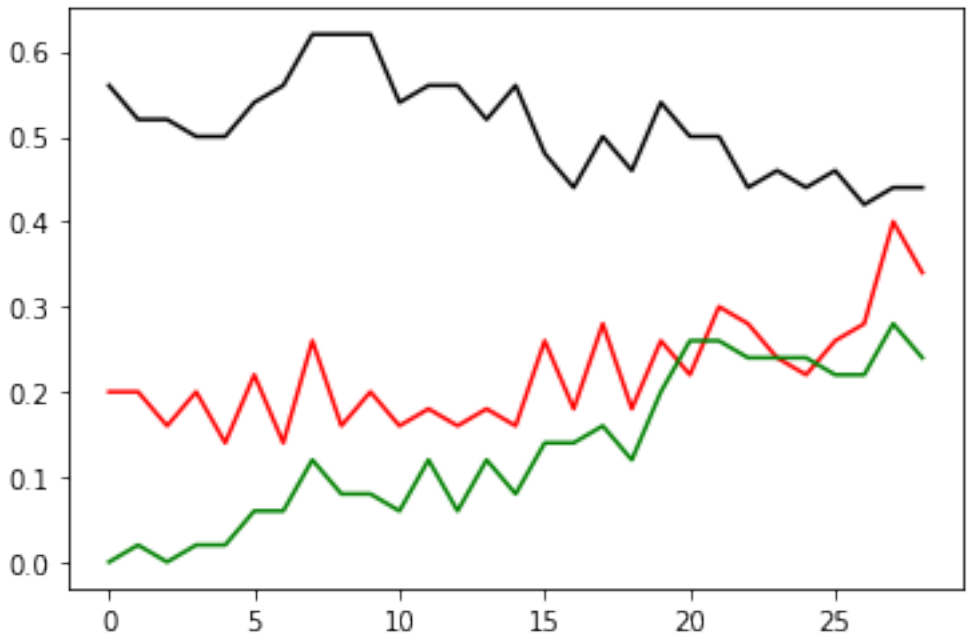
Since the validation set is used to select a model, it cannot be used to give us an idea of the expected risk Standard 3-split procedure: *train, validation, test*

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

```
In [17]: key = jax.random.PRNGKey(33)
         Xv = jax.random.uniform(key, (50, 2))
         yv = gt(Xt)
```

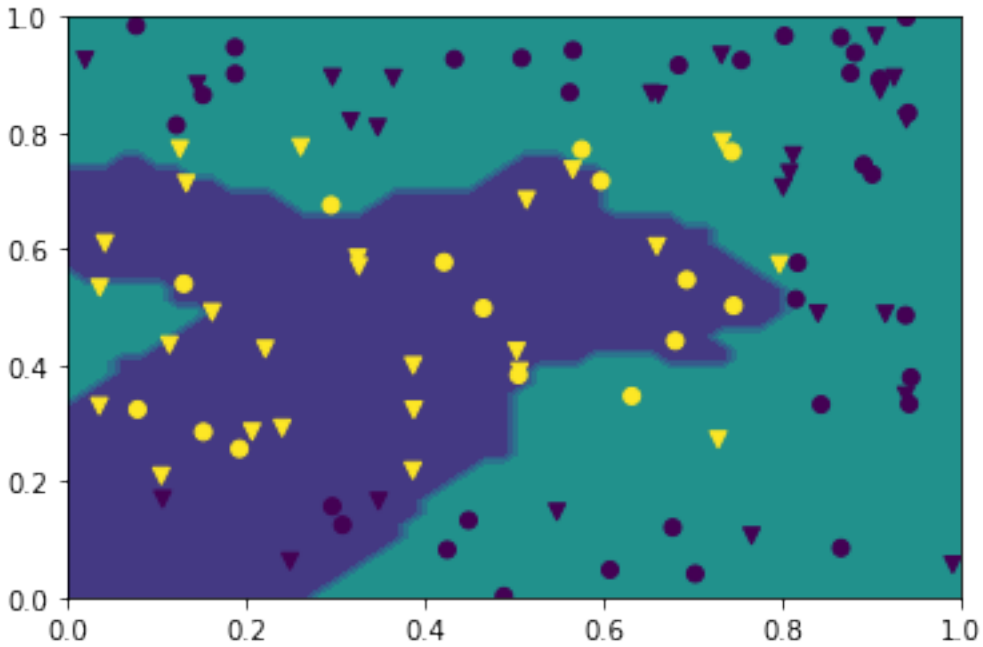
```
In [18]: lv = []; lt = []; lr = []
         for k in range(1,30):
             nn = KNearestNeighbor(X, y, k)
             lv.append(loss(nn(Xv), yv))
             lt.append(loss(nn(Xt), yt))
             lr.append(loss(nn(X), y))
         plt.plot(lv, '-k'), plt.plot(lt, '-r'), plt.plot(lr, '-g')
```

```
([<matplotlib.lines.Line2D at 0x7f09a0454f50>],
 [<matplotlib.lines.Line2D at 0x7f09e1829610>],
 [<matplotlib.lines.Line2D at 0x7f09e1829a50>])
```



```
In [19]: 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 0x7f09a0021850>,
 <matplotlib.collections.PathCollection at 0x7f09e17e9550>)
```



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

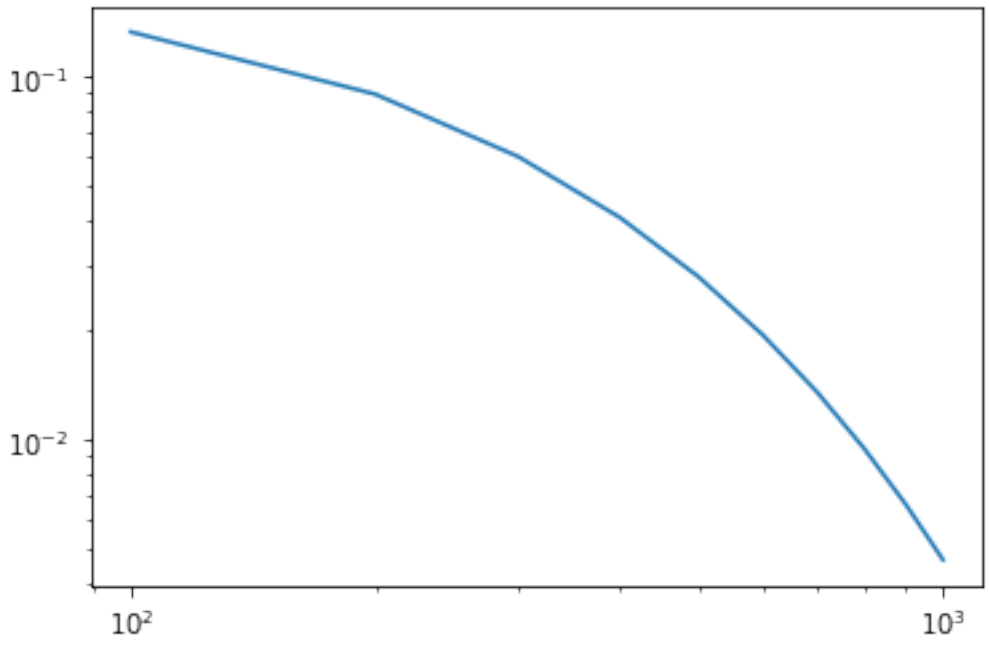
$$\sum_{k=1}^{m}\binom{n}{k}\epsilon^k(1-\epsilon)^{n-k}$$

For η observe error rate

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

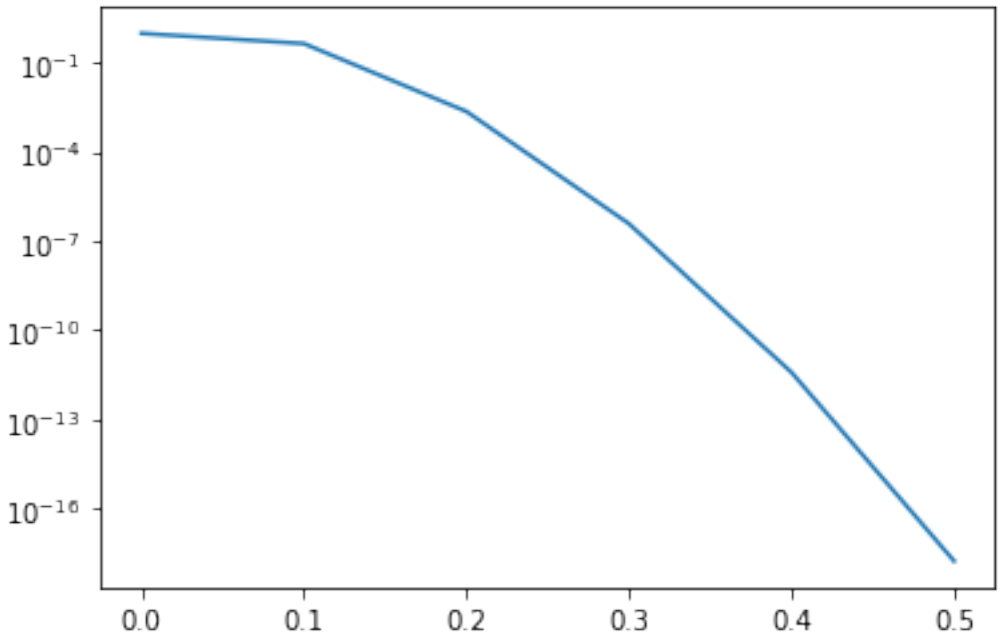
```
In [20]: def Pn_of_eta_given_eps(n, eta, eps):
             p = 0
             for k in range(int(eta*n)):
                 p += scipy.special.comb(n, k) * eps**k * (1-eps)**(n-k)
             return p
         x = range(100, 1100, 100)
         p = [Pn_of_eta_given_eps(i, 0.01, 0.02) for i in x]
         plt.loglog(x, p)
```

```
[<matplotlib.lines.Line2D at 0x7f09e1719810>]
```



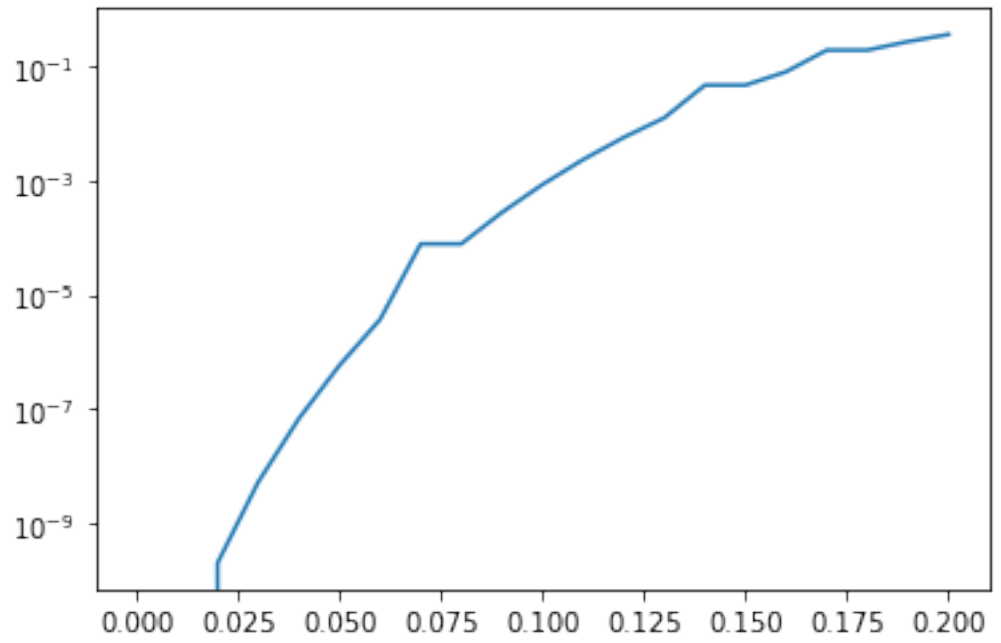

```
In [21]: x = 0.1*jnp.arange(0, 6, 1)
         p = [Pn_of_eta_given_eps(100, 0.1, i) for i in x]
         plt.semilogy(x, p)
```

```
[<matplotlib.lines.Line2D at 0x7f09e1fe3190>]
```



In [22]:
```
x = 0.01*jnp.arange(0, 21, 1)
p = [Pn_of_eta_given_eps(100, i, 0.2) for i in x]
plt.semilogy(x, p)
```

```
[<matplotlib.lines.Line2D at 0x7f09e14a1310>]
```



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

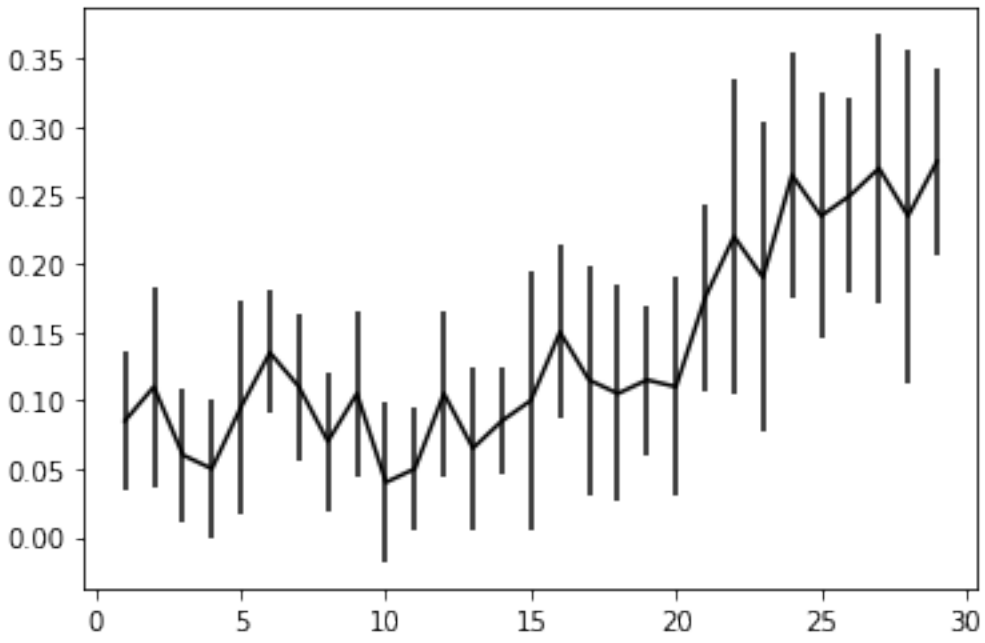
- Variance gives an idea of the relevance of the selection process

```
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>
```

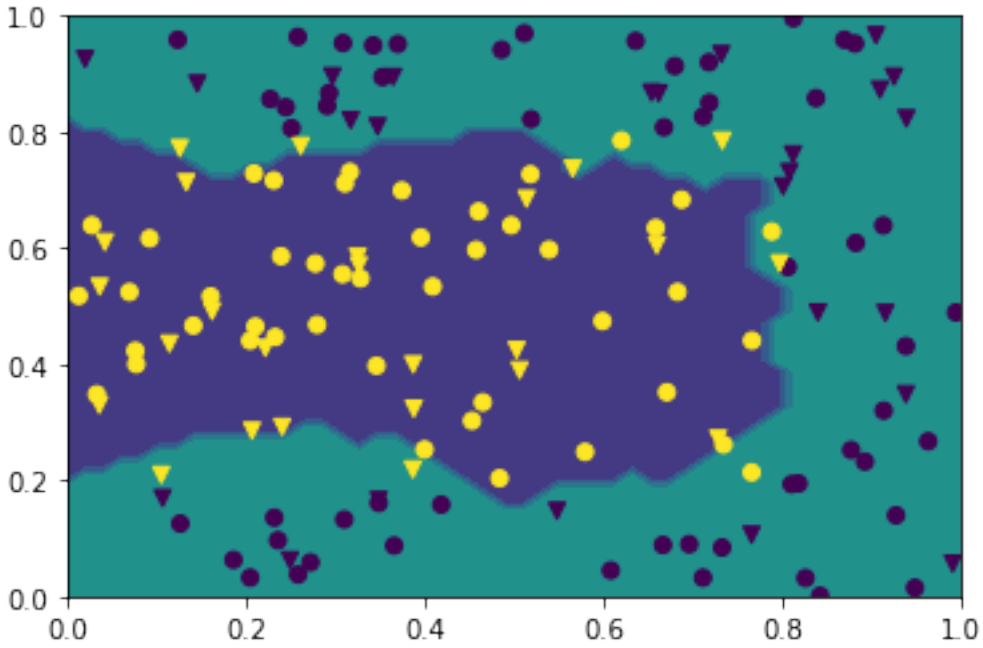


1.6.2 Full training

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 0x7f09e0fca9d0>,
 <matplotlib.collections.PathCollection at 0x7f09e13e8e50>)
```



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?

1.7.1 Regression

- $\mathcal{Y}$ is continuous

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

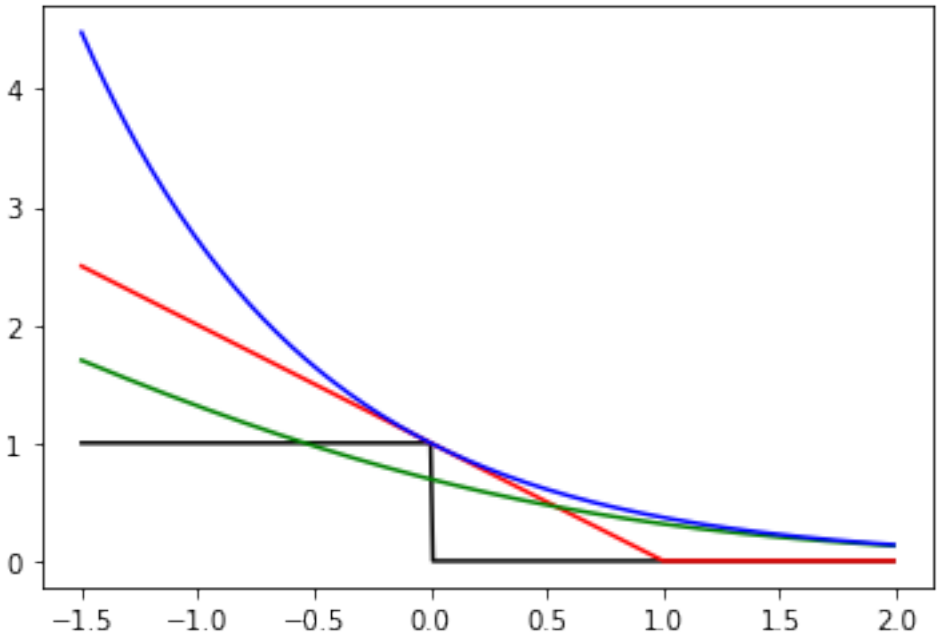
- Vector case: any norm of $y - f(X)$

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: $e^{-yf(X)}$

```
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 0x7f09e0f59510>]
```



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

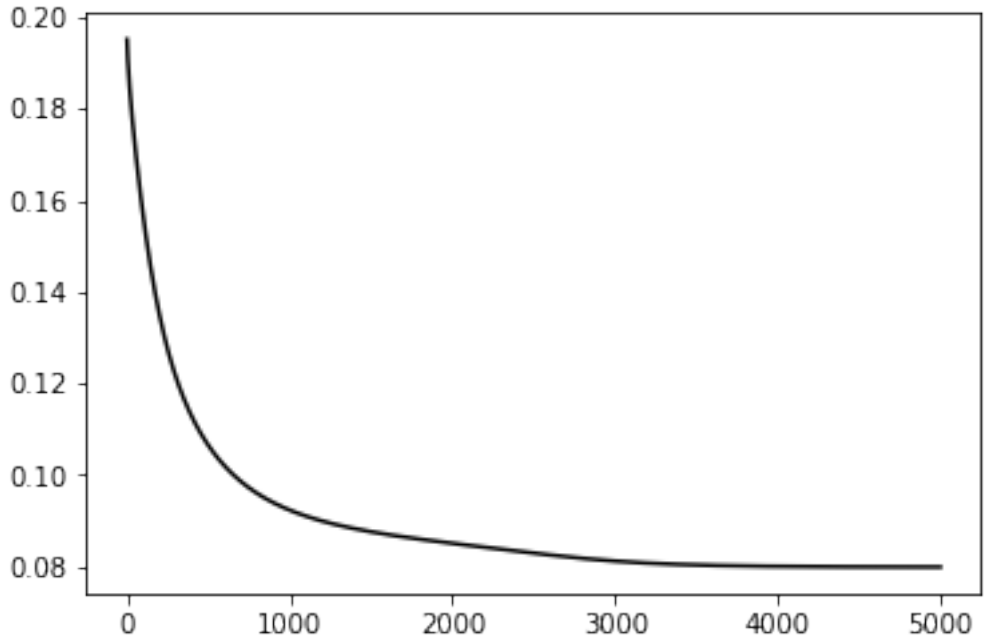
$$\min_{A,C} \sum_{x} (y - 1 + (x - C)^T A (x - C))^2$$

- 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
```

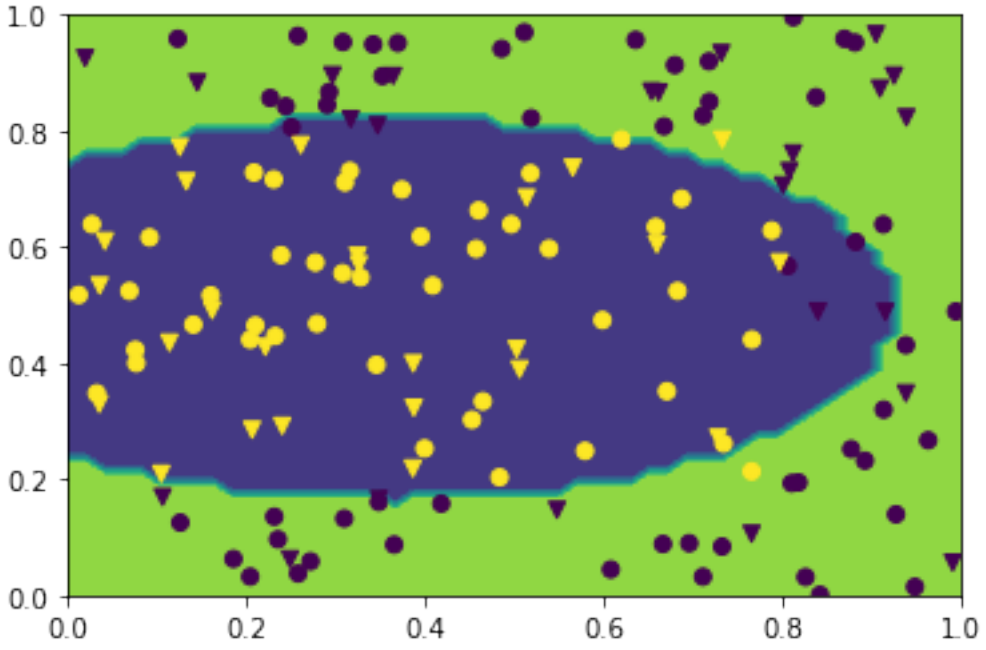
In [29]:
```
key = jax.random.PRNGKey(32)
key, skey = jax.random.split(key)
c = jax.random.uniform(key, (2,))
a = jnp.ones(2)
l = []
for t in range(5000):
    a, c = update(a, c, X, y)
    l.append(loss(a, c, X, y))
plt.plot(l, '-k')
```

```
[<matplotlib.lines.Line2D at 0x7f09e1908090>]
```



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 0x7f09e13b4150>,
 <matplotlib.collections.PathCollection at 0x7f09e0cfd310>)
```



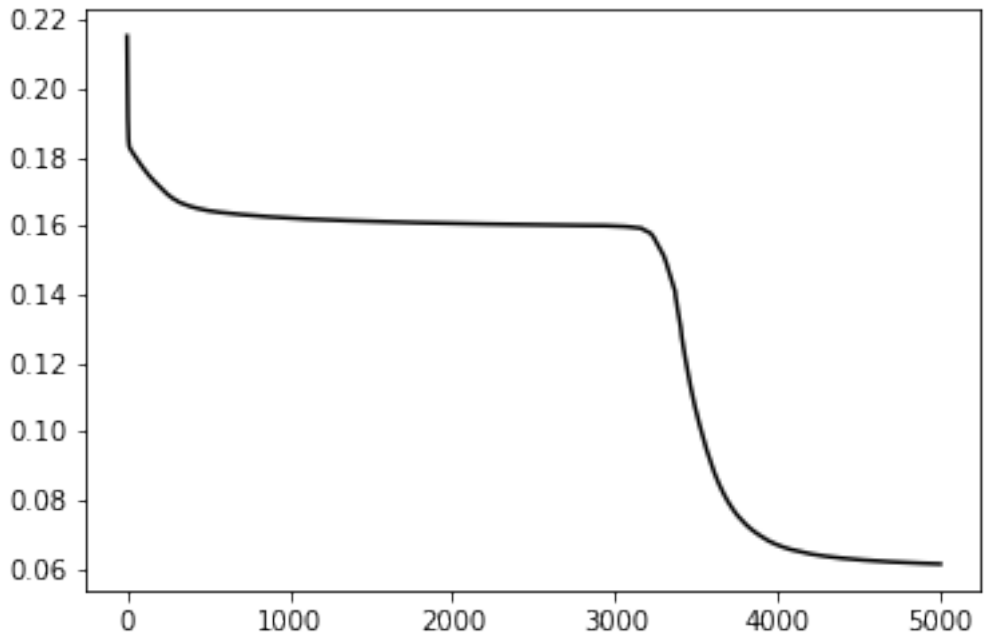
Rectangle → switch from ℓ_2 to ℓ_∞ norm

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

```
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
```

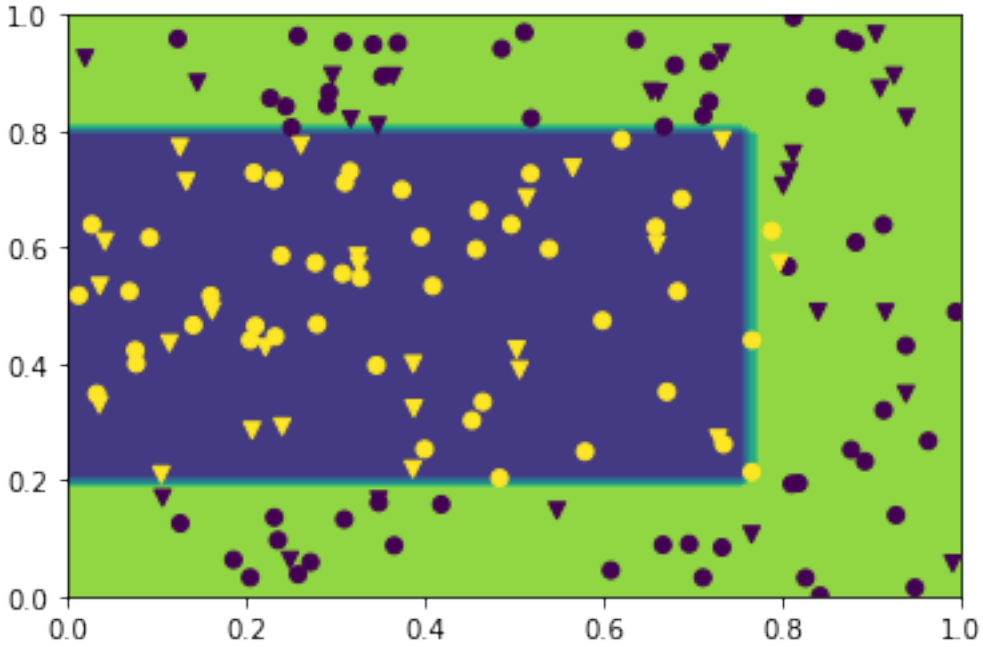
```
In [32]: key = jax.random.PRNGKey(32)
         key, skey = jax.random.split(key)
         c = jax.random.uniform(key, (2,))
         a = jnp.ones(2)
         l = []
         for t in range(5000):
             a, c = update(a, c, X, y)
             l.append(loss(a, c, X, y))
         plt.plot(l, '-k')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

```
[<matplotlib.lines.Line2D at 0x7f09e0f17c50>]
```



```
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 0x7f09e136de50>,
 <matplotlib.collections.PathCollection at 0x7f09e136d350>)
```



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

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

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

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

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

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

```
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))
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and
rerun for more info.)
```

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

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

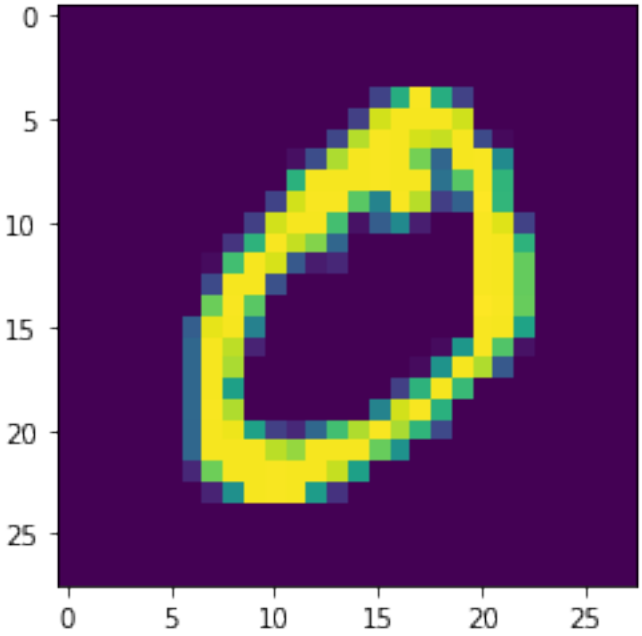
Equivalent to p scalar output cases stacked together

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])
```

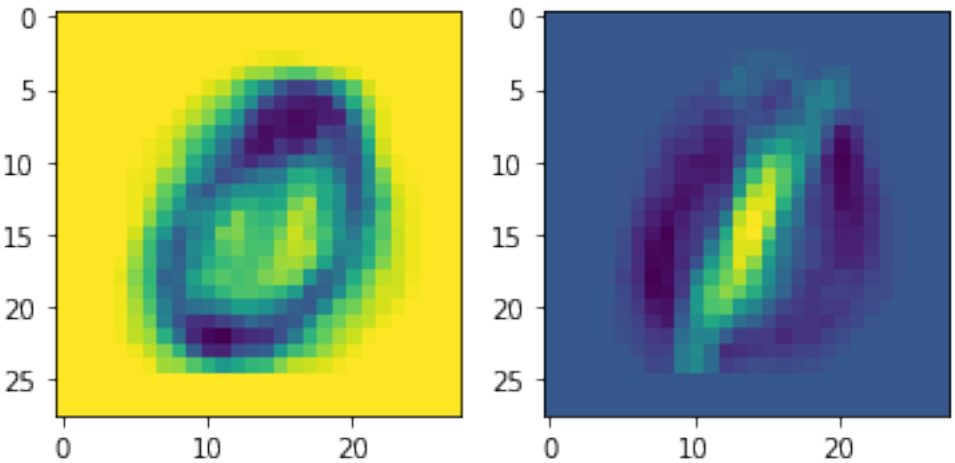
```
0
```




```
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]
```

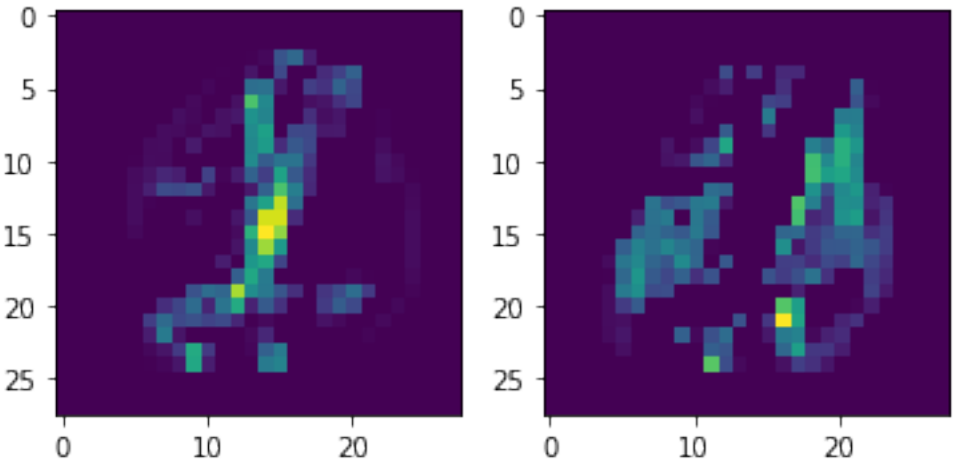
```
<matplotlib.image.AxesImage at 0x7f19a0082290>
```



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.

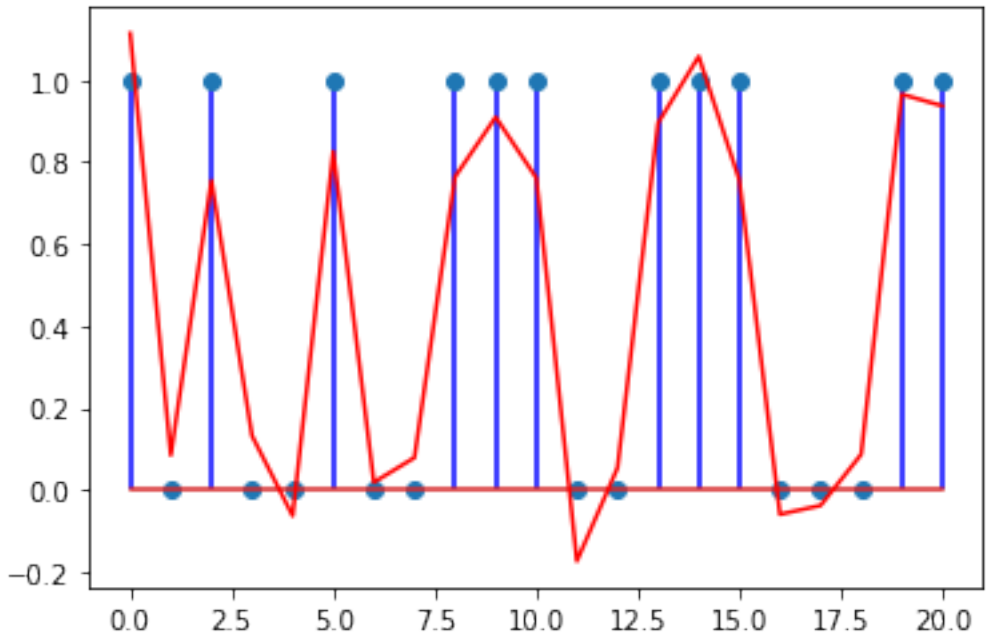
```
In [5]: Vty = jnp.matmul(V, y)
        SinvVty = jnp.matmul(jnp.diag(1./S), Vty)
        a_hat = jnp.matmul(U, SinvVty)

        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 0x7f198c728b10>
```




```
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 0x7f19a01e8b10>]
```

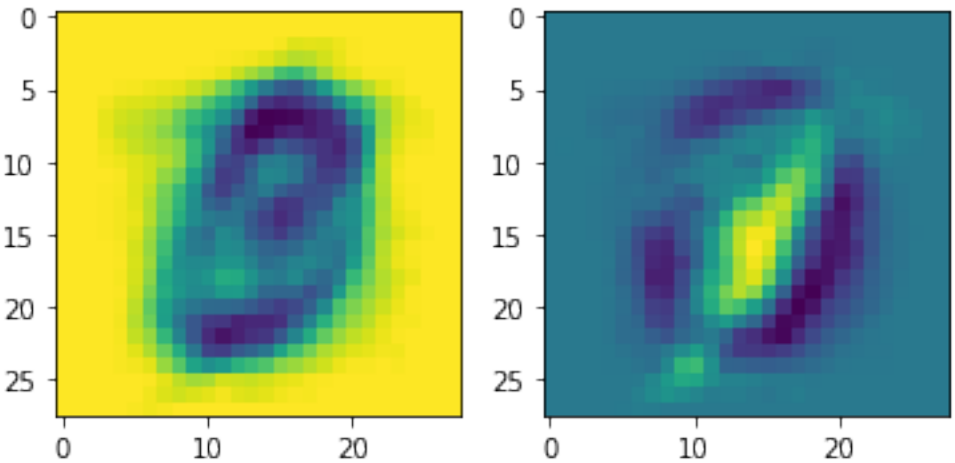


2.1.8 MNIST, regress 0-9

```
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)))
```

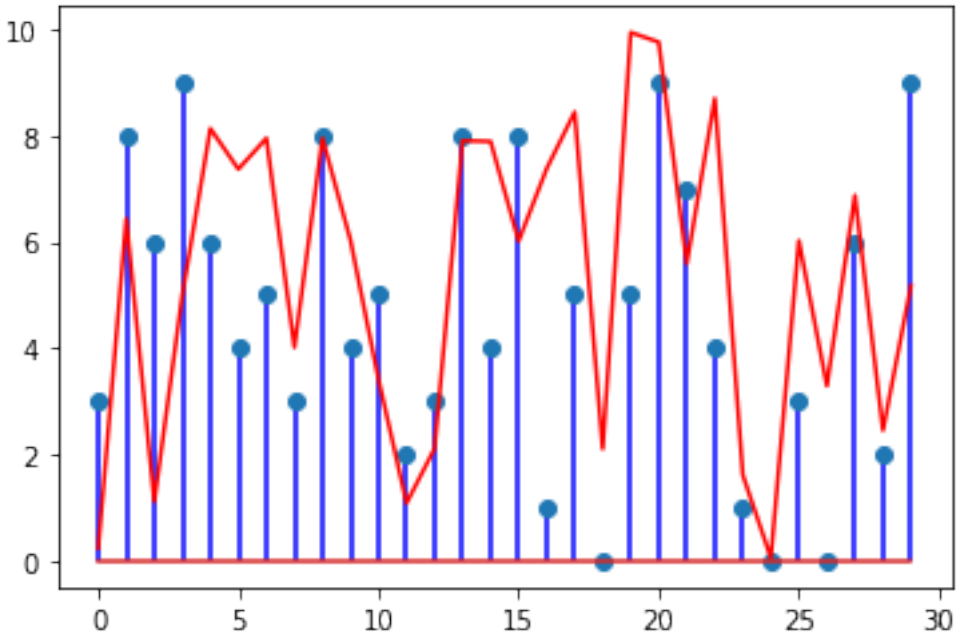
```
eigenvalues: [61.84758  22.601597 19.816174 18.98699  17.21556  15.531815 14.318835
 13.584824 12.348789 11.739741]
```

```
<matplotlib.image.AxesImage at 0x7f198c6107d0>
```



```
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 0x7f198c554510>]
```



Output space not adapted (artificial topology)

2.1.9 MNIST regress 0-9 as one-hot

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

```
In [ ]: 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')
```

2.2 Non-linear case

2.2.1 Polynomial regression

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

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

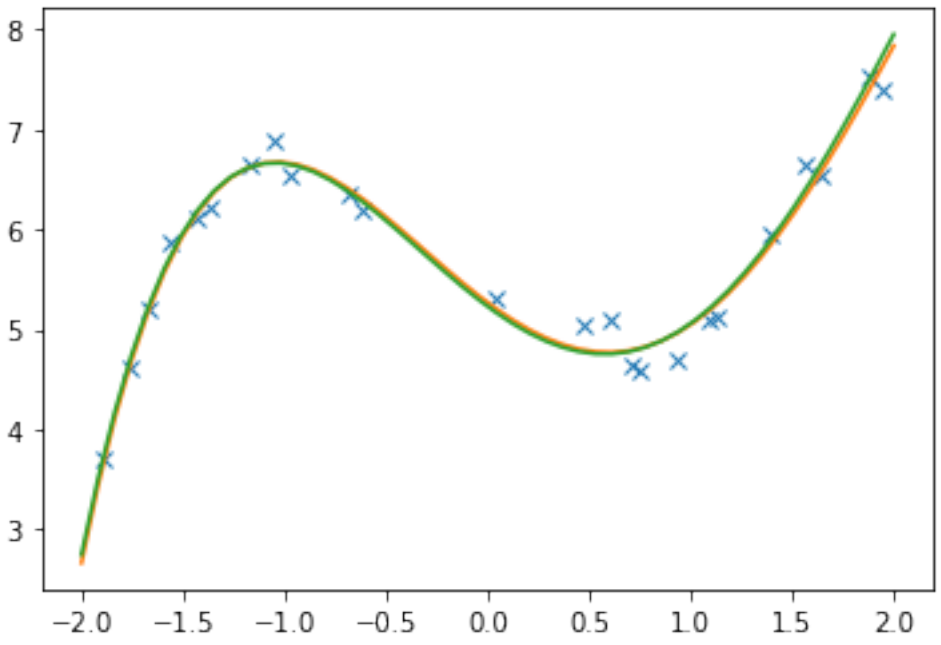
$$\min_{\mathbf{a}} \mathbb{E}_x[(y - \phi(x)^\top \mathbf{a})^2]$$

```
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))
```

```
[ 5.2697163  -1.5135039   0.7879974   0.70156187 -0.19815296]
```

```
[<matplotlib.lines.Line2D at 0x7f19846b1fd0>]
```

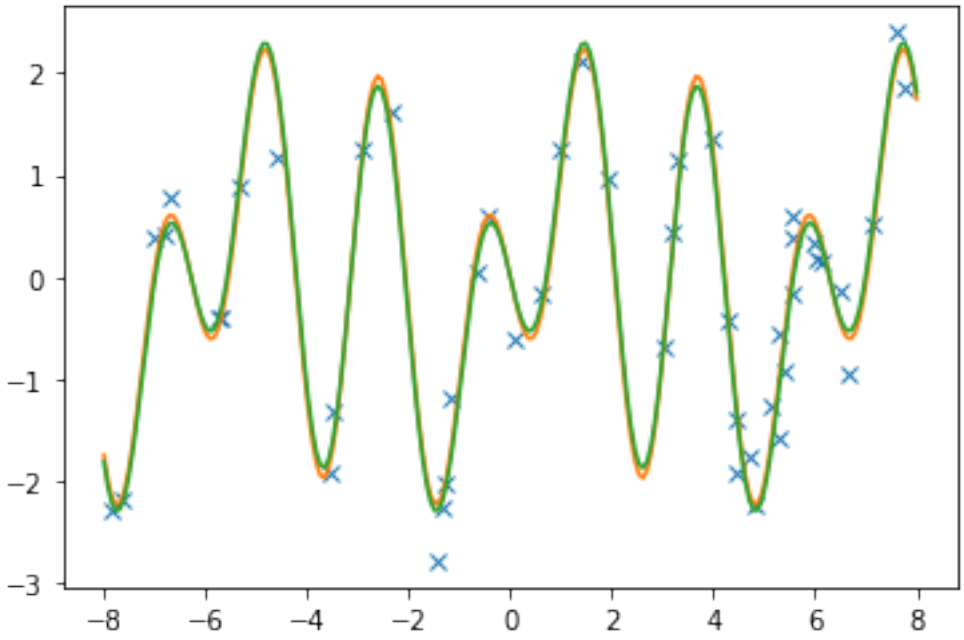


2.2.2 Periodic signals

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 0x7f198457be10>]
```



2.2.3 Overcomplete models

What if $p < \hat{p}$ (model has greater capacity than data)

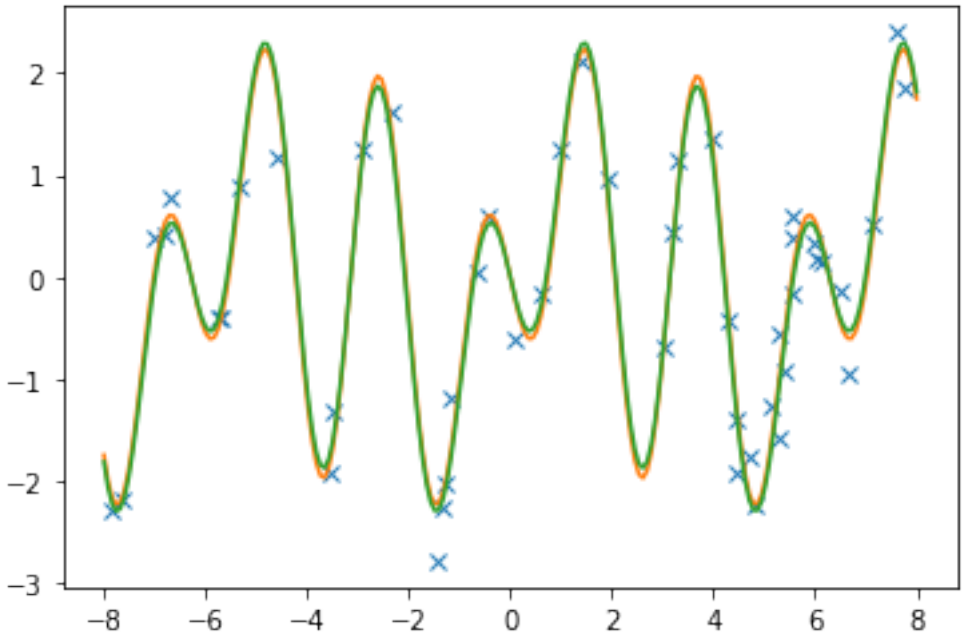
```
In [15]: def sin_approx(x, y, p):
             Xp = jnp.sin(jnp.matmul(x.reshape(-1, 1), 1+jnp.arange(p).reshape(1,p))).transpo
             ap = jnp.matmul(jnp.matmul(jnp.linalg.inv(jnp.matmul(Xp, Xp.transpose())), Xp),
             Yp = jnp.matmul(ap, Xp)
             return ap, Xp, Yp
```

```
In [16]: p = 3
         ap, Xp, Yp = sin_approx(x, y, p)
         print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [ 0.59392786  0.83312976 -1.5430541 ]
```

```
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: 0.09781882911920547
```

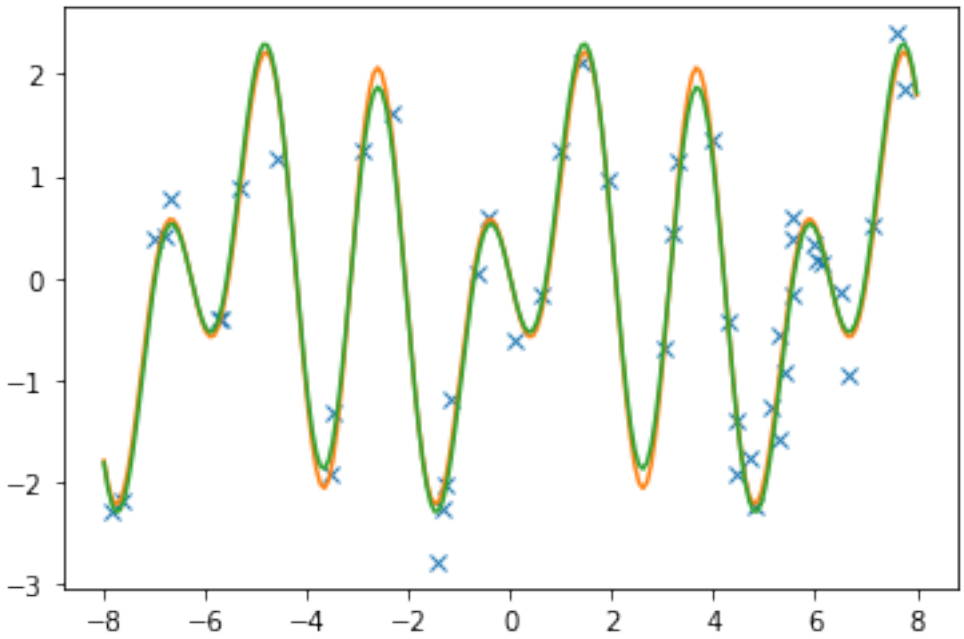



```
In [18]: p = 5
         ap, Xp, Yp = sin_approx(x, y, p)
         print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [ 0.59637094  0.8385289  -1.5672264   0.06469631 -0.01800793]
```

```
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: 0.0961289182305336
```

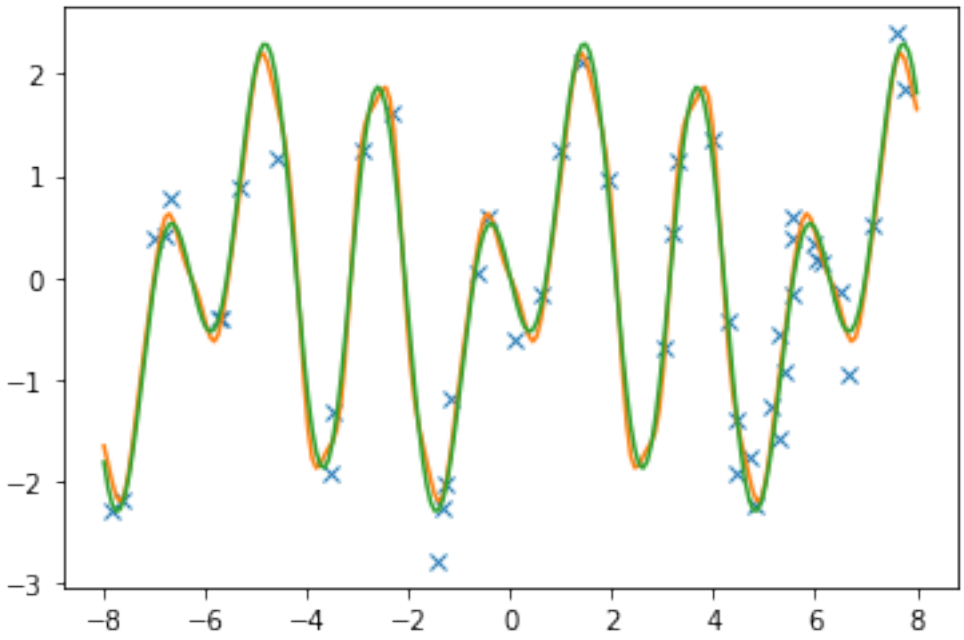



```
In [20]: p = 10
         ap, Xp, Yp = sin_approx(x, y, p)
         print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [ 0.6073679   0.81426585 -1.5253923   0.03802376 -0.00418478
-0.01558349
  0.06264809  0.05295399 -0.09149553  0.11939779]
```

```
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: 0.08592929691076279
```

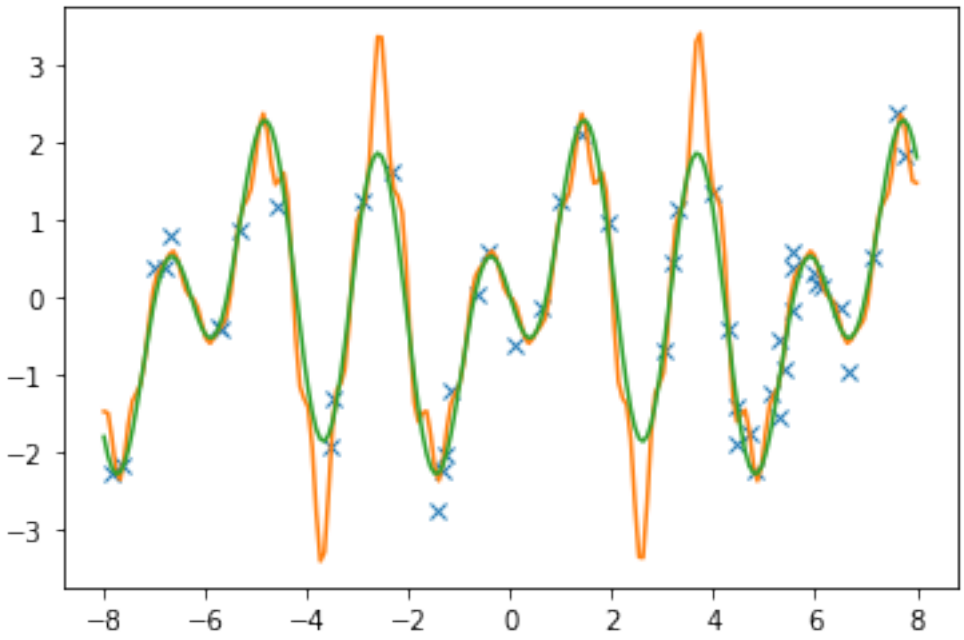



```
In [22]: p = 15
         ap, Xp, Yp = sin_approx(x, y, p)
         print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [ 0.35529602  1.1142317  -1.7437214   0.15871115  0.00858292
-0.10777945
  0.15600105 -0.08507273  0.08278456 -0.04437378  0.08144233 -0.01490425
 -0.21221659  0.28036714 -0.02712816]
```

```
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: 0.06189465522766113
```

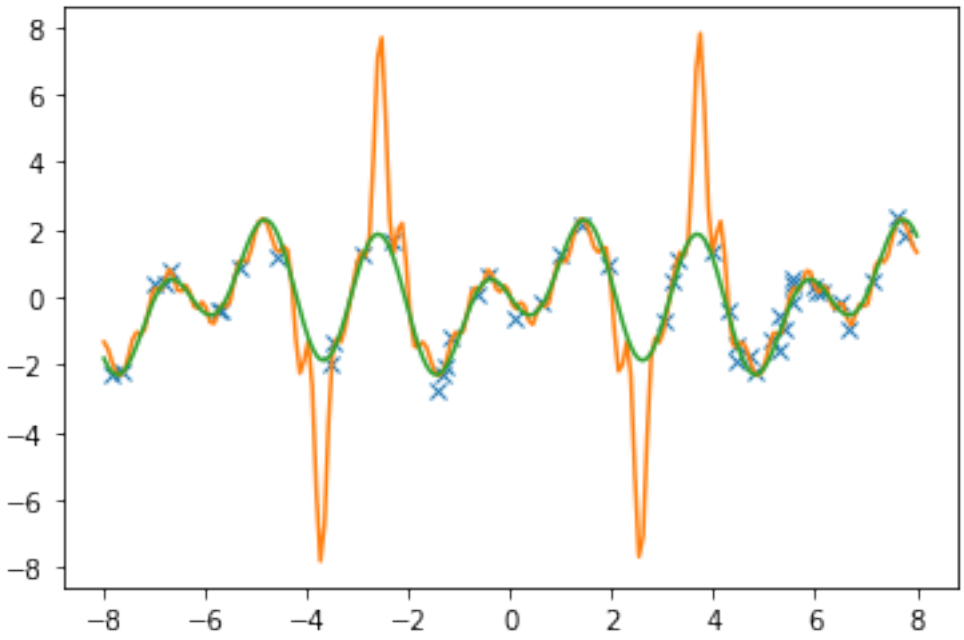



```
In [24]: p = 20
         ap, Xp, Yp = sin_approx(x, y, p)
         print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [-0.11323678  1.7993791  -2.3644838   0.49984902 -0.01194978
-0.32451046
  0.5129152  -0.50142884  0.45345783 -0.24447411  0.10667838  0.20195109
 -0.6100314   0.6498178  -0.25845143 -0.03708184  0.16033    -0.29417005
  0.03206168 -0.0902726 ]
```

```
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: 0.04660849645733833
```

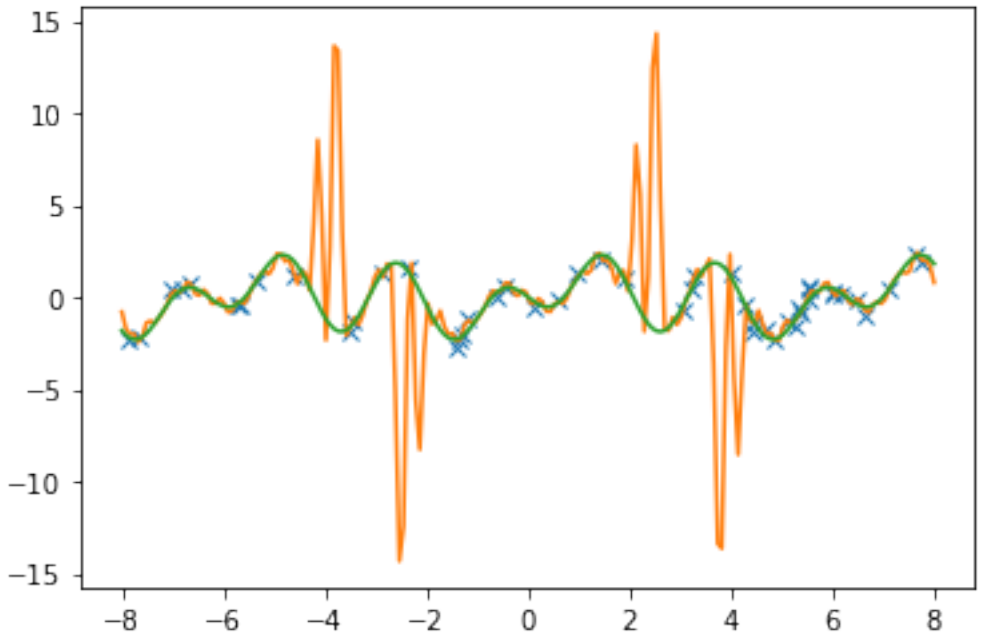



```
In [26]: p = 25
         ap, Xp, Yp = sin_approx(x, y, p)
         print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [ 2.483035   -1.9207842   0.57731056 -0.44159245 -0.9491972
1.4420291
 -1.1824136   0.8650079  -0.61695516  0.45492303  0.19773307 -1.0832504
  1.5003977  -1.1892037   0.3630464   0.81794167 -1.6101568   1.2754178
 -0.51291    -0.5356204   0.87538815 -0.7491144   0.10084951  0.21558303
 -0.11464696]
```

```
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: 0.0363682359457016
```

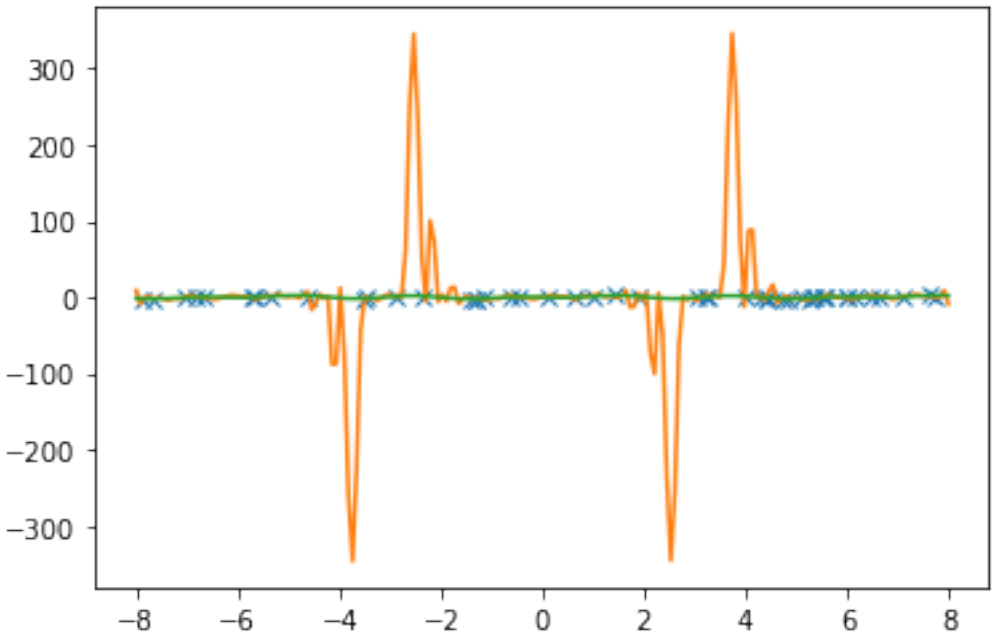



```
In [28]: p = 35
         ap, Xp, Yp = sin_approx(x, y, p)
         print(a, ap)
```

```
[ 0.7   0.83 -1.5 ] [-34.420845    53.54271    -47.404167    24.04839      2.5520477
 -22.580738    33.575417   -30.581444    22.487469    -8.539528
  -5.8411484   17.944695   -26.375664    26.720001   -15.869003
  -2.5119934   19.008682   -22.40055     14.070158    -1.2852402
  -7.173237     8.027071    -5.567108     0.89237213   2.7684917
  -2.0045357   -2.4827309    5.055525    -2.6962426   -2.1028605
   4.513747    -2.6814532   -0.19490051   0.98603344  -0.61041975]
```

```
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: 2.7143747806549072
```

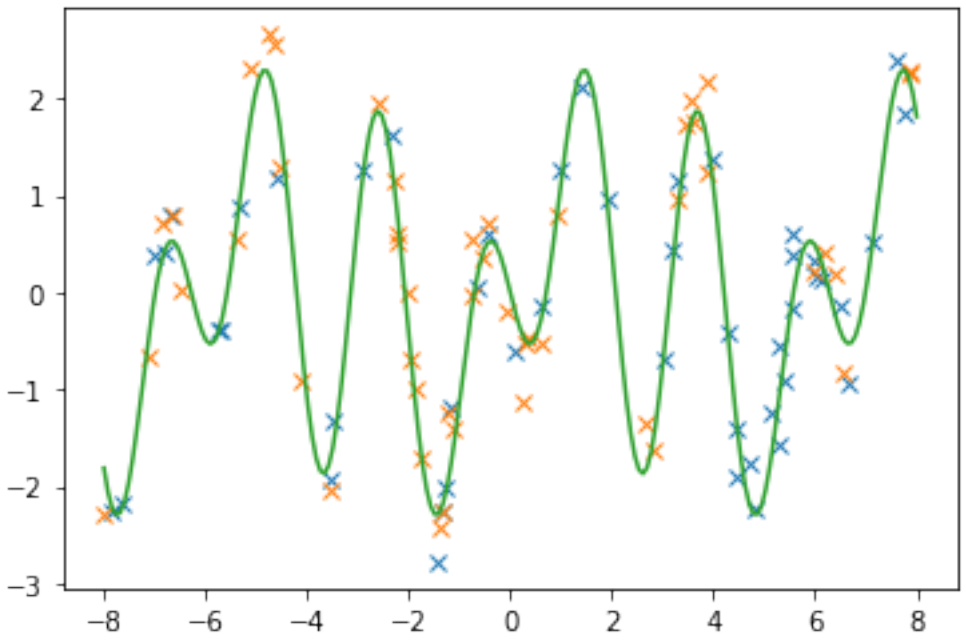


2.2.4 Train/validation

```
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)
```

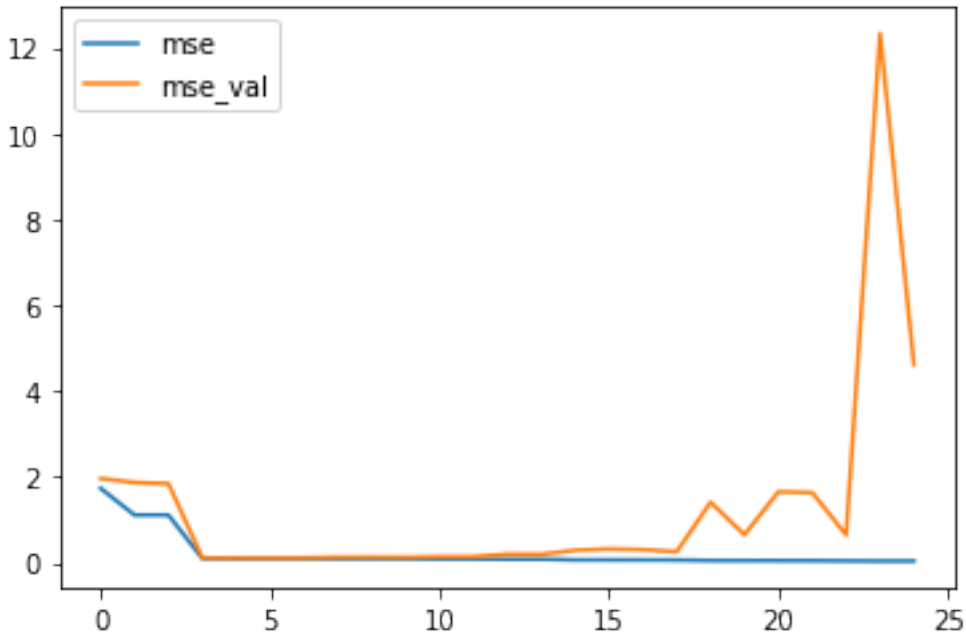
```
In [31]: plt.plot(x, y, 'x')
         plt.plot(x_val, y_val, 'x')
         plt.plot(t, jnp.matmul(a, T[:len(a), :]))
```

```
[<matplotlib.lines.Line2D at 0x7f194c73a3d0>]
```



```
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 0x7f19a3b17310>
```



2.3 Regularization

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

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

Optimize using gradient descent

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

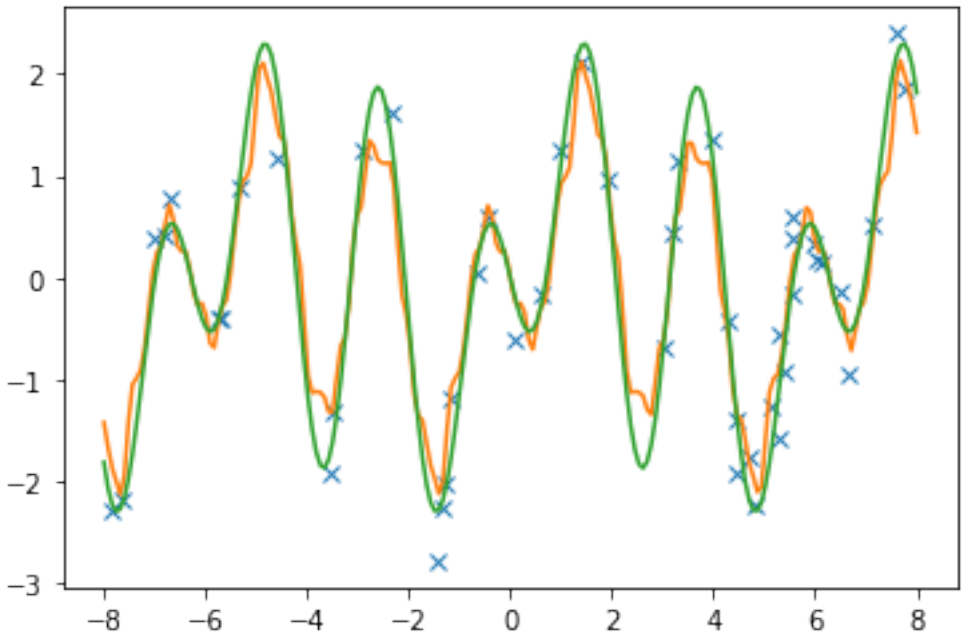
```
In [33]: def sin_pred(a, X):
             return jnp.matmul(a, X)

         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 ] [ 6.5387887e-01  4.9198857e-01 -1.2269275e+00  1.6643824e-03
 -5.4124887e-03  2.7261022e-03 -4.6199327e-03  1.8823075e-03
 -1.3611598e-03  1.0498903e-01  6.7312829e-04 -6.6337660e-02
  1.5109220e-03  6.7850342e-03  6.5390445e-02 -2.7749939e-03
 -8.5329272e-02  3.3036065e-03 -6.3027009e-02 -5.3070008e-04
 -4.4770658e-04 -5.5702450e-03 -4.1388847e-02  4.2398345e-02
  5.0350791e-03]
```

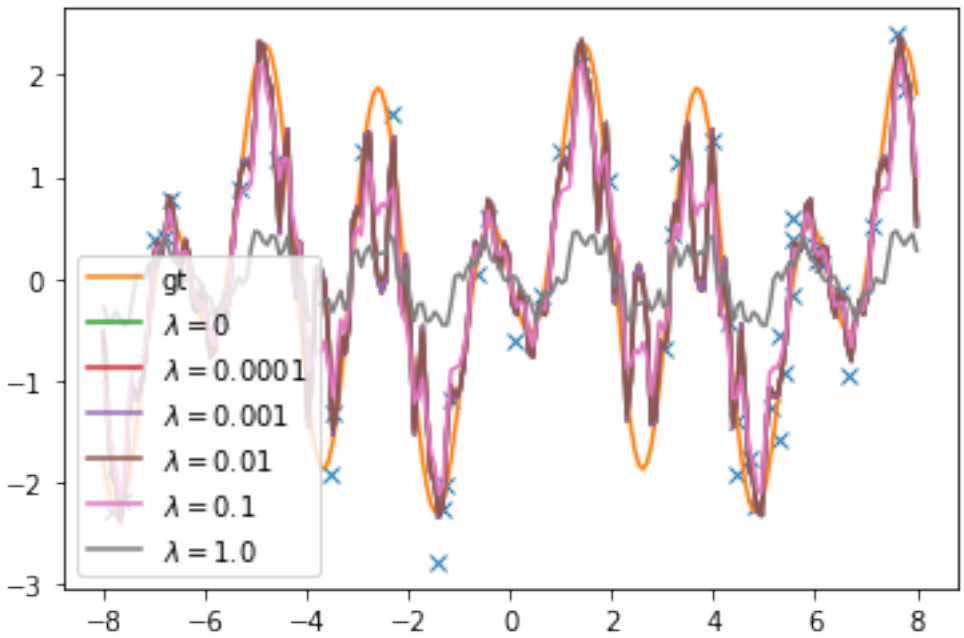
```
In [35]: t = jnp.linspace(-8, 8, 200)
         T = jnp.sin(np.matmul(t.reshape(-1, 1), 1+jnp.arange(p).reshape(1,p))).transpose()
         plt.plot(x, y, 'x')
         plt.plot(t, jnp.matmul(ap, T[:len(ap), :]))
         plt.plot(t, jnp.matmul(a, T[:len(a), :]))
         print('MSE: {}'.format(((y - Yp)**2).mean()))
```

```
MSE: 0.037019111216068 27
```




```
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()
```

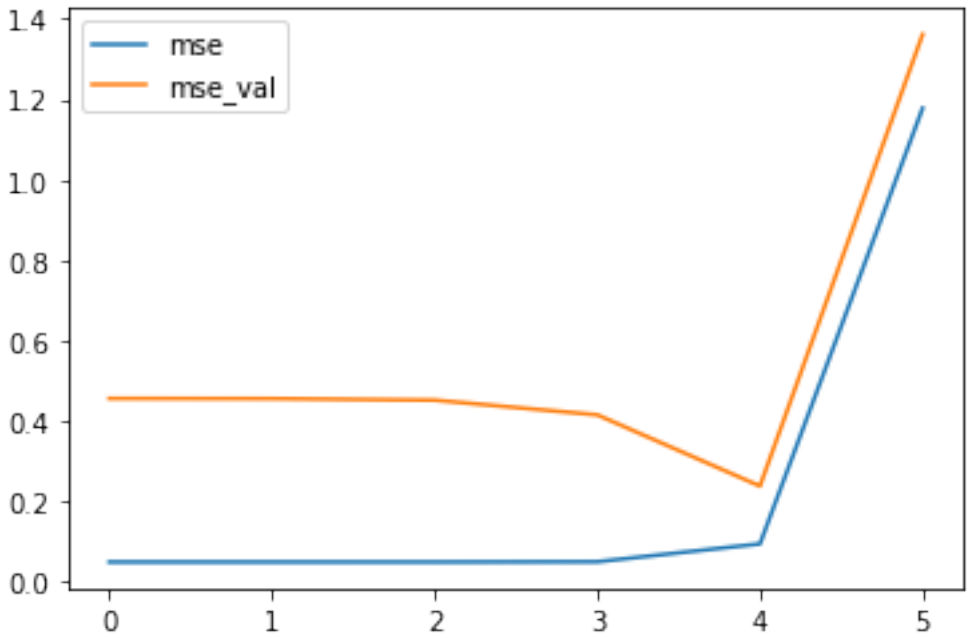
```
<matplotlib.legend.Legend at 0x7f19842dc2d0>
```



```
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 0x7f1928662990>
```



2.3.2 Analysis

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

$$\min_{\mathbf{a}} \frac{1}{n}\|\mathbf{y} - \mathbf{X}^\top \mathbf{U}\mathbf{a}\|^2 + \lambda\|\mathbf{a}\|_1$$

$$= \frac{1}{n}\|\mathbf{y} - \mathbf{VSa}\|^2 + \lambda\|\mathbf{a}\|_1$$

Stationary condition:

$$\frac{\partial}{\partial \mathbf{a}} = 0 = -2\mathbf{SV}^\top\mathbf{y} + 2\mathbf{S}^2\mathbf{a} + \lambda \text{sign}(\mathbf{a})$$

$$\mathbf{a} = \mathbf{S}^{-1}\mathbf{V}^\top\mathbf{y} - \mathbf{S}^{-2}\frac{\lambda \text{sign}(\mathbf{a})}{2}$$

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:

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

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)

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

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

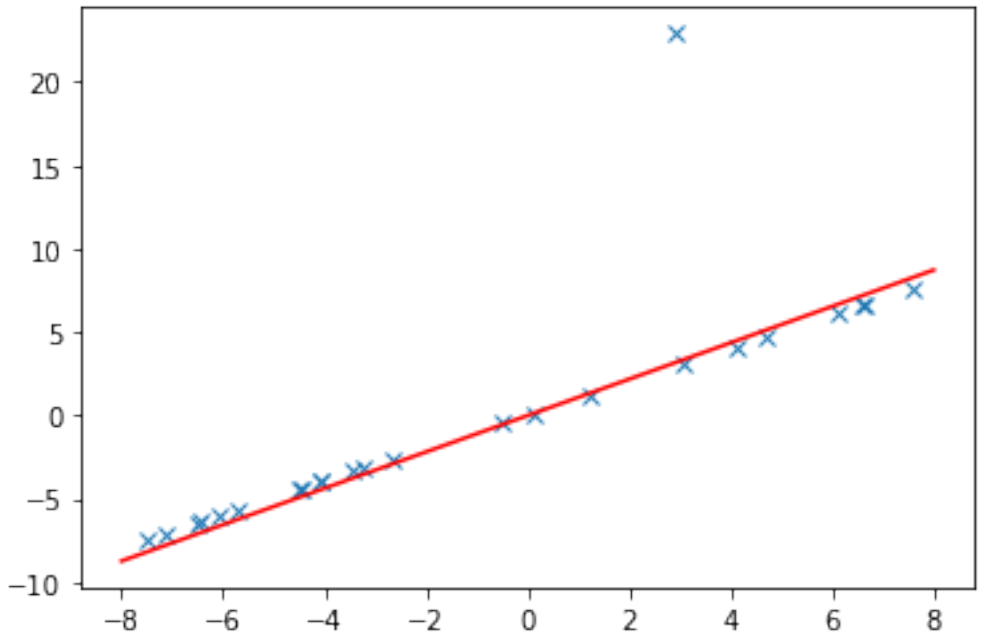
2.4 Other loss functions

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')
```

```
1.0905238
```

```
[<matplotlib.lines.Line2D at 0x7f198467fd50>]
```



2.4.1 MAE

Mean average error (or ℓ_1 error)

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

Vector case

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

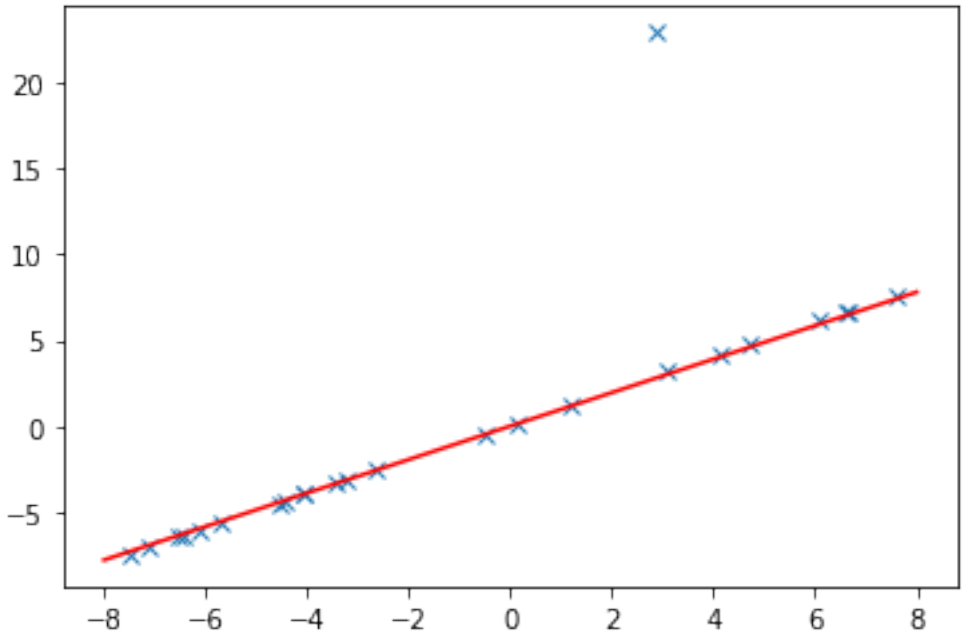
No close form solution, gradient descent (subderivative $\nabla \|0\|_1 = 0$)

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')
```

```
0.974505
```

```
[<matplotlib.lines.Line2D at 0x7f1984355790>]
```

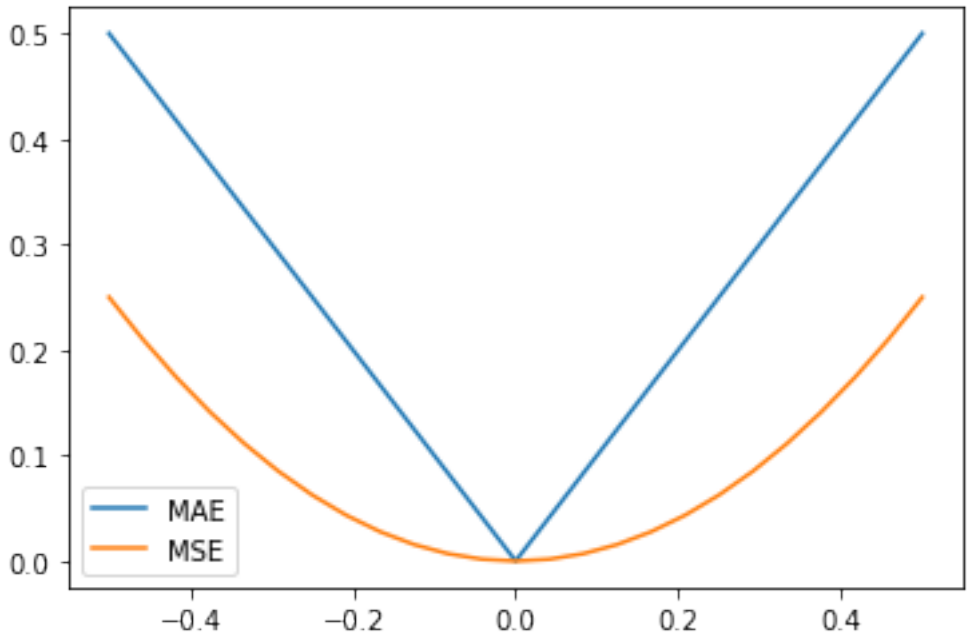


2.5 Sensitivity to small errors

- 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 0x7f1928500110>
```



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)

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

- $\mathbf{d}_k$: 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:

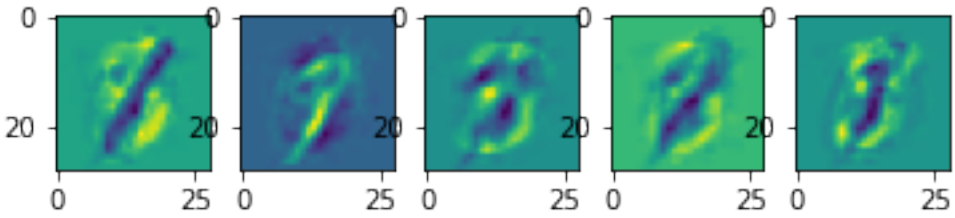
$$\min_{\mathbf{d}_k, \mathbf{a}^k} \|\mathbf{E}\Omega_k - \mathbf{d}_k \mathbf{a}^k \Omega_k\|_F^2$$

- 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 [43]:
```
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)
```

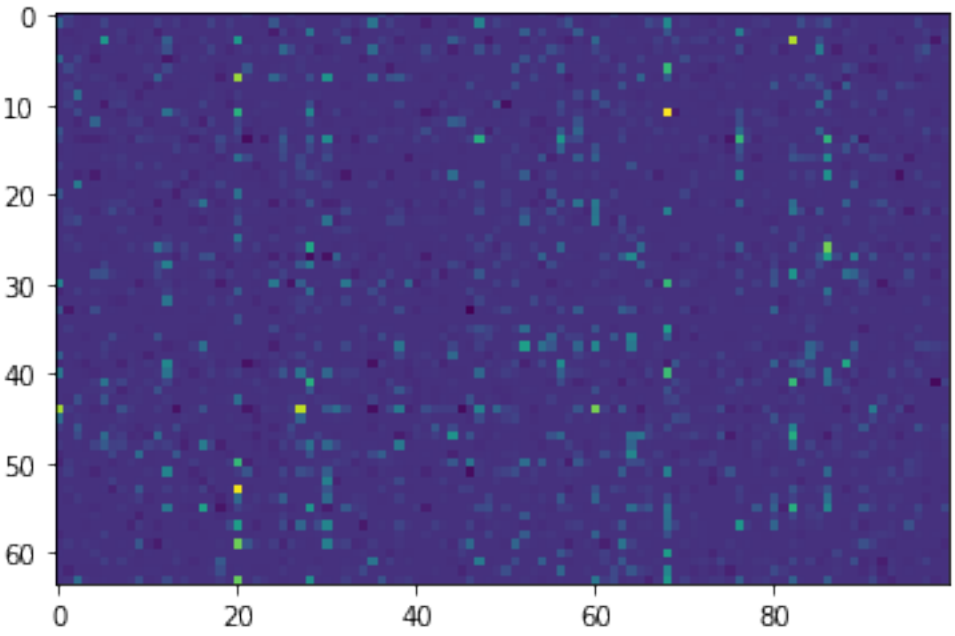
In [44]:
```
plt.subplot(1,5,1)
plt.imshow(D[:,0].reshape(28, 28))
plt.subplot(1,5,2)
plt.imshow(D[:,1].reshape(28, 28))
plt.subplot(1,5,3)
plt.imshow(D[:,2].reshape(28, 28))
plt.subplot(1,5,4)
plt.imshow(D[:,3].reshape(28, 28))
plt.subplot(1,5,5)
plt.imshow(D[:,4].reshape(28, 28))
```

```
<matplotlib.image.AxesImage at 0x7f19282cff10>
```



In [45]:
```
plt.imshow(A)
```

```
<matplotlib.image.AxesImage at 0x7f19281f1e10>
```



2.6.2 MNIST

Exercise: Try a linear regression (vector output) using $\mathbf{A}$ instead of $\mathbf{X}$

In [ ]:

2.6.3 Why?

- $\mathbf{A}$ may provide a better alternative to $\mathbf{X}$ for doing learning a predictor
- $\mathbf{D}$ may provide insights (modes of $\mathbf{X}$)

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

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

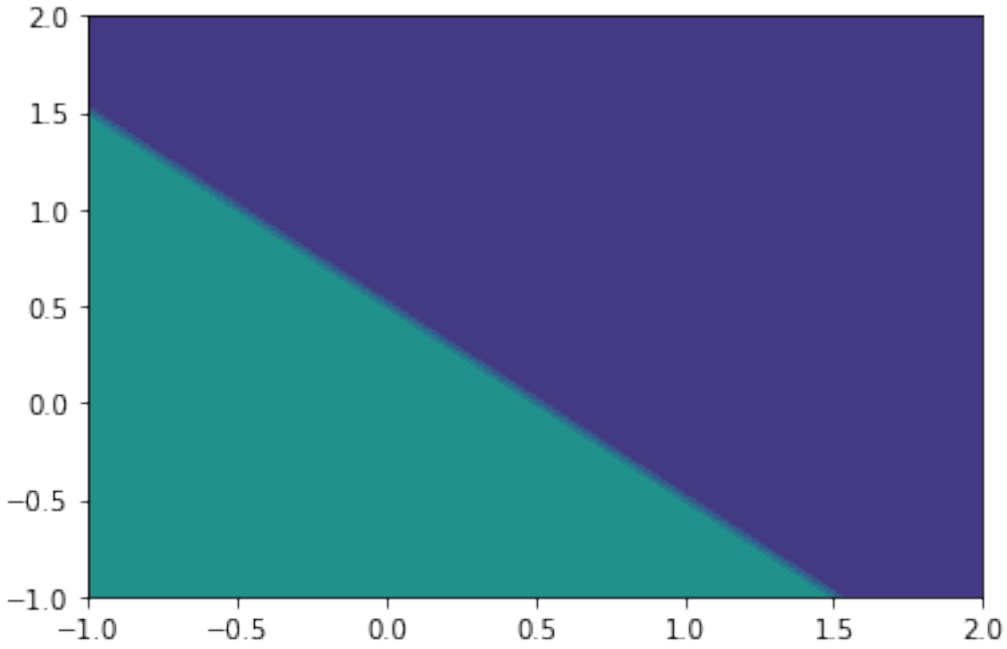
Linear prediction function

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

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

```
In [3]: 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);
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and
rerun for more info.)
```



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:

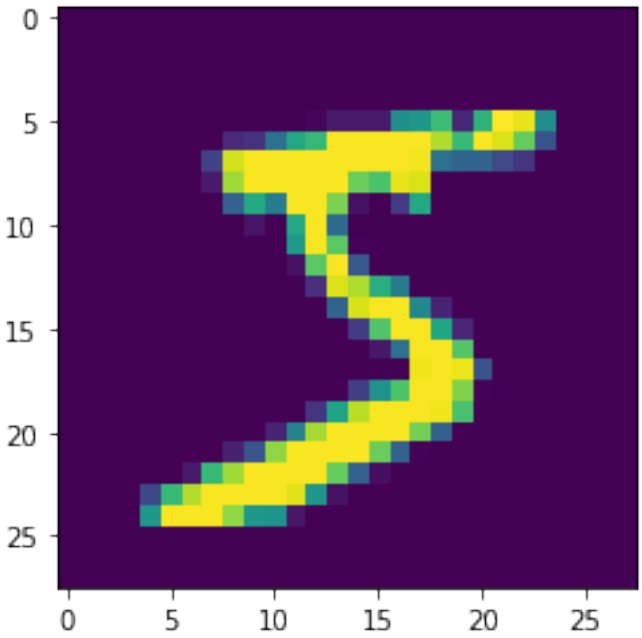
$$\min_{\mathbf{w},b} \frac{1}{n} \sum_i max(0, 1 - y_i(\langle \mathbf{w}, \mathbf{x} \rangle + b))$$

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

3.1.2 MNIST

```
In [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])
```

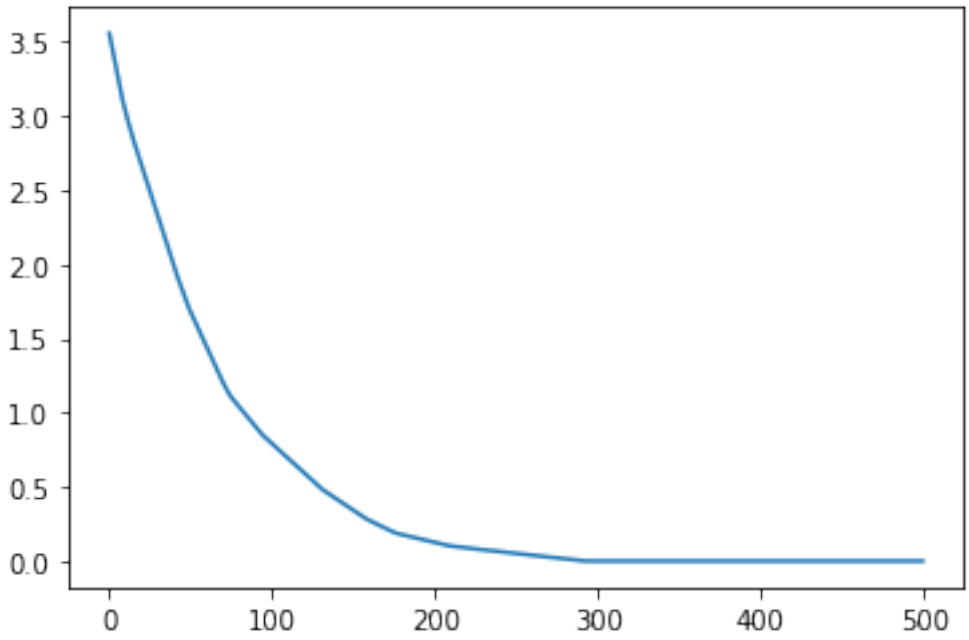
```
5
```




```
In [9]: 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 [10]: w = np.random.randn(784)
         b = 0.

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

```
[<matplotlib.lines.Line2D at 0x7f5cc401d5d0>]
```




```
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)))
```

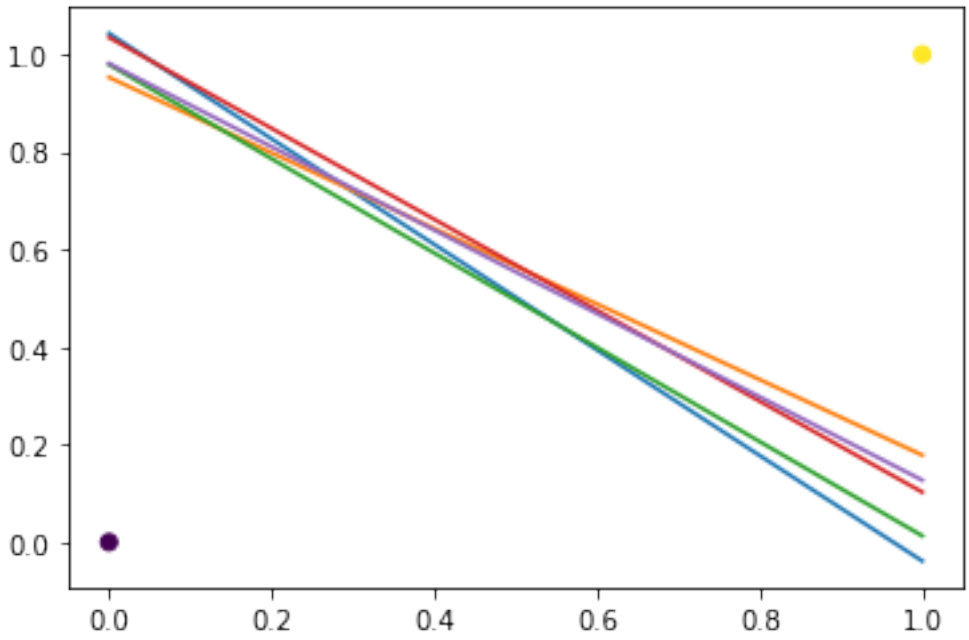
```
accuracy: 1.0
```

```
In [12]: 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
```

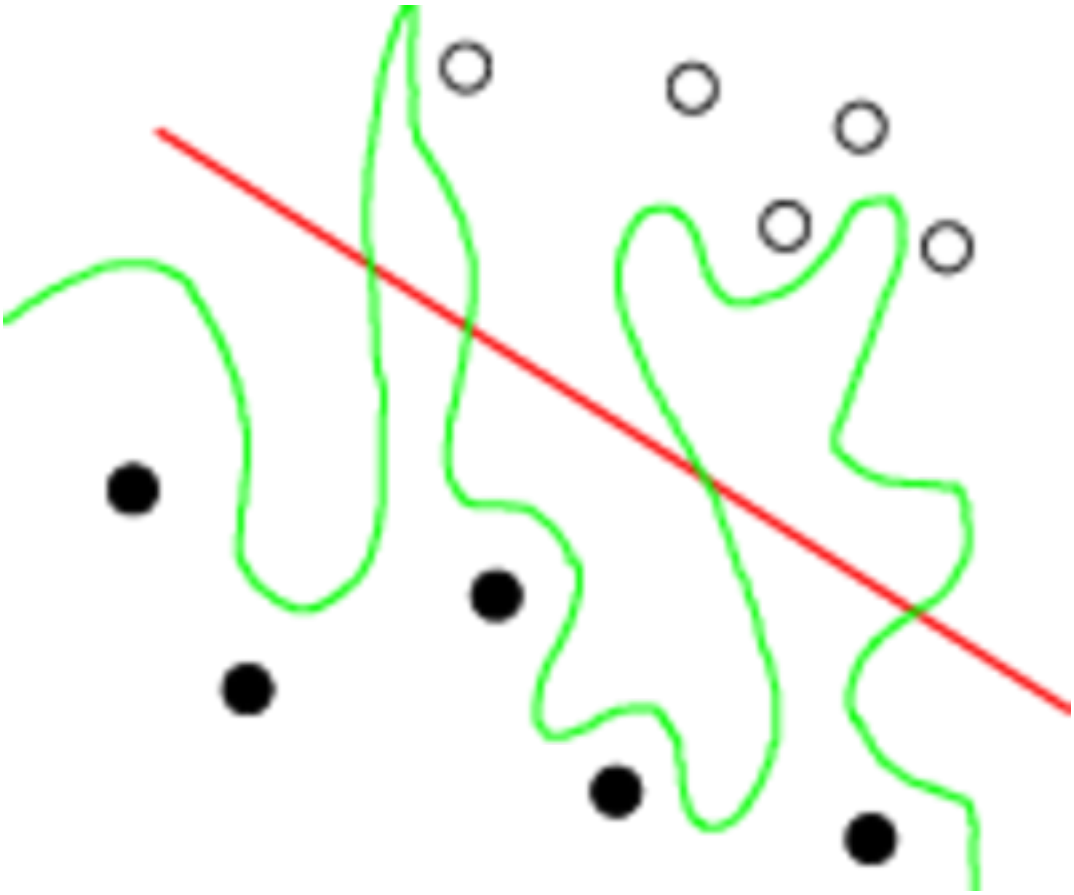
3.1.3 Equivalent solutions

```
In [13]: 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]])
```



3.1.4 Complexity impacts generalization

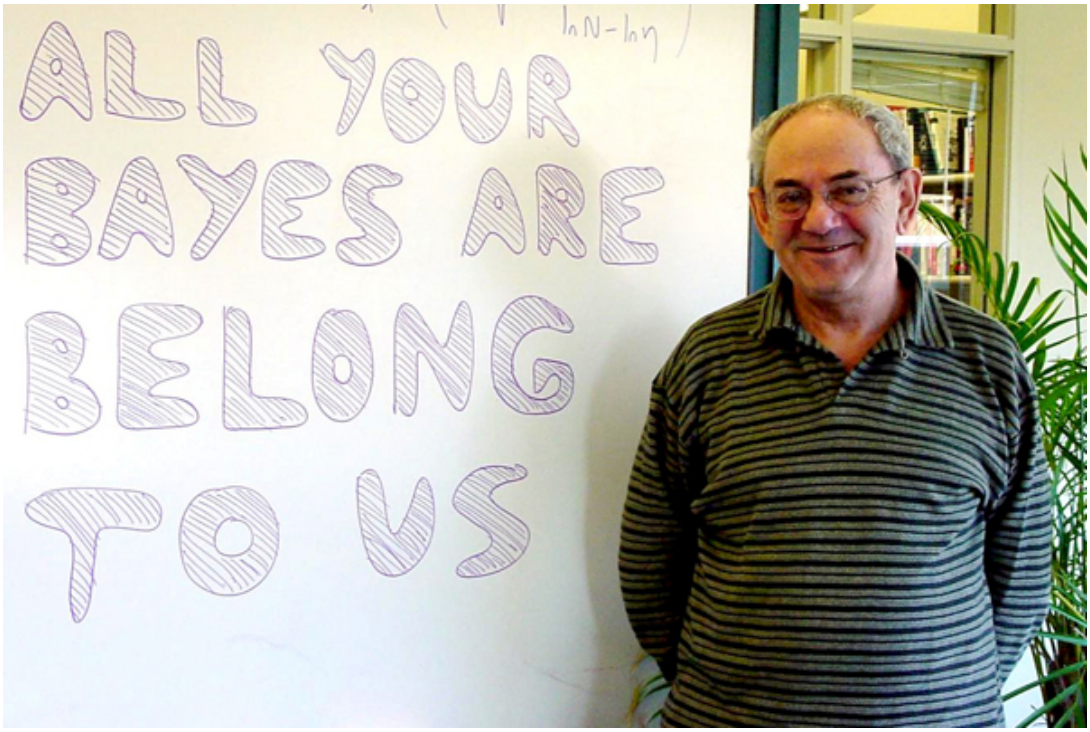
In [16]: Image('complexity.pdf.png', width=400)



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 (Vladimir N. Vapnik)

```
In [18]: Image('vapnik.jpg', width=400)
```



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

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.

3.1.8 Exercises

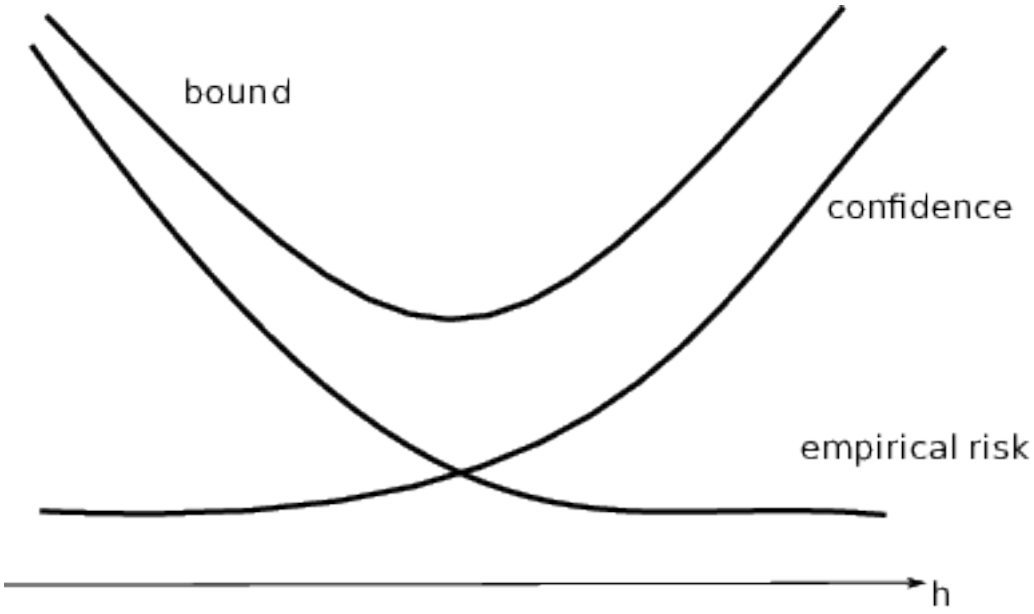
- What is the VC Dimension of linear functions in the 2D plane?
- What is the VC Dimension of axis-aligned rectangles in the 2D plane?

3.1.9 Risk Bound

True risk is bounded by a combination of empirical risk and structural risk depending on h

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

```
In [19]: Image('struct.pdf.png', width=400)
```



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,

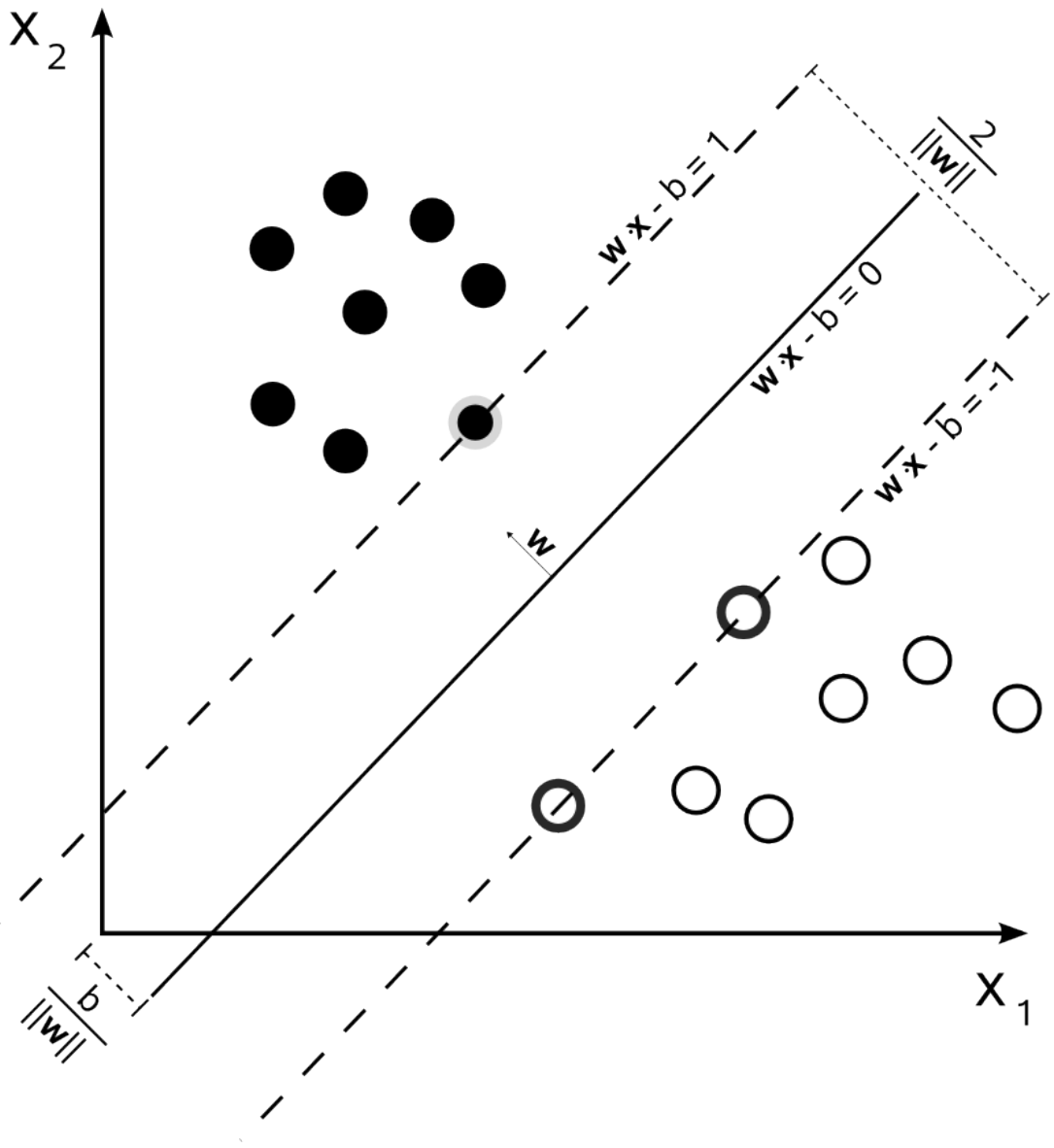
$$h \leq \min\left(\frac{R^2}{\Delta^2}, d\right) + 1 \tag{3.2}$$

$\Rightarrow$ We have to maximize the margin

3.1.11 ℓ_2 norm

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

```
In [21]: Image('margin.png', width=400)
```



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

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

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

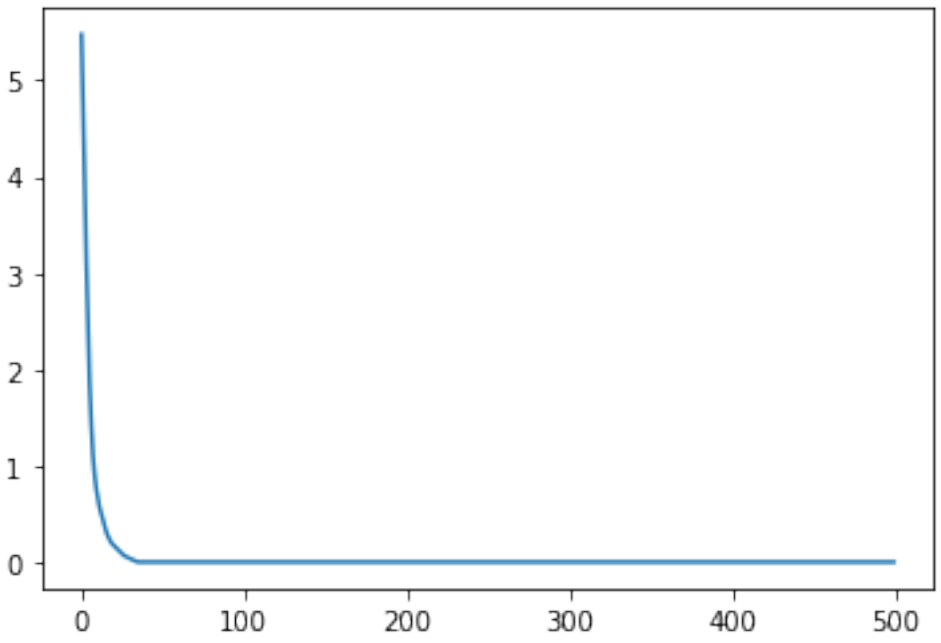
Strongly convex problem

3.2.2 MNIST Cont.

```
In [26]: 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 0.001*(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 [27]: 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 0x7f5cb46a8e10>]
```



```
In [28]: 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 [29]: y_pred = func(w, b, X_val)
         print('validation accuracy: {}'.format(accuracy(y_pred, y_val)))
```

```
validation accuracy: 1.0
```

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

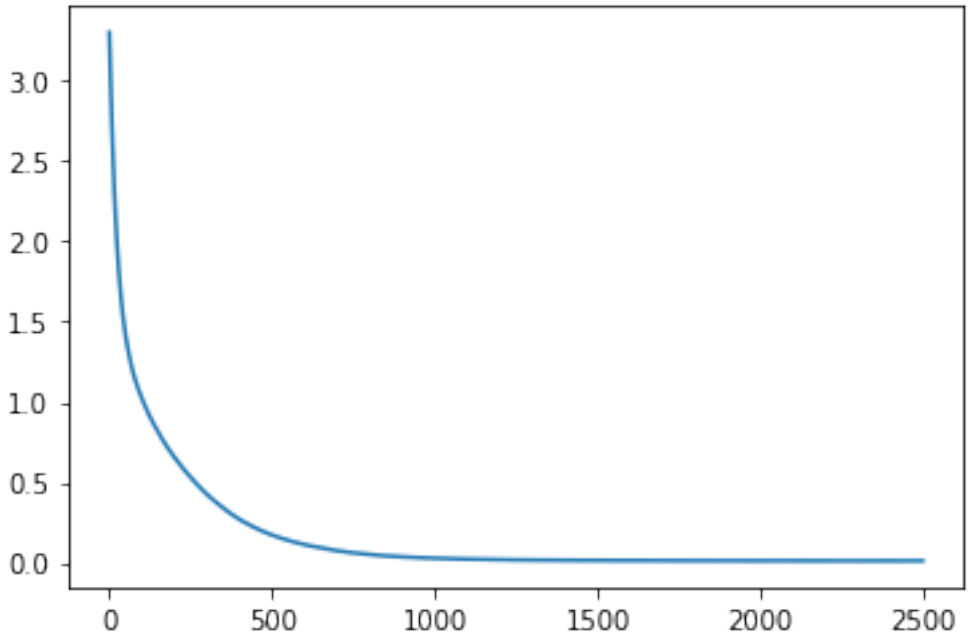
3.2.4 MNIST

One versus all

```
In [44]: 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 [45]: 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 0x7f5cc41b2910>]
```




```
In [46]: 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 [47]: 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: 0.6700000166893005
```

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:

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

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

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

$$\mathbf{w}^\star - \sum_i \alpha_i y_i \mathbf{x}_i = 0$$
$$\mathbf{w}^\star = \sum_i \alpha_i y_i \mathbf{x}_i$$

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

3.2.8 Representer theorem

Theorem (Schölkopf et al.): 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:

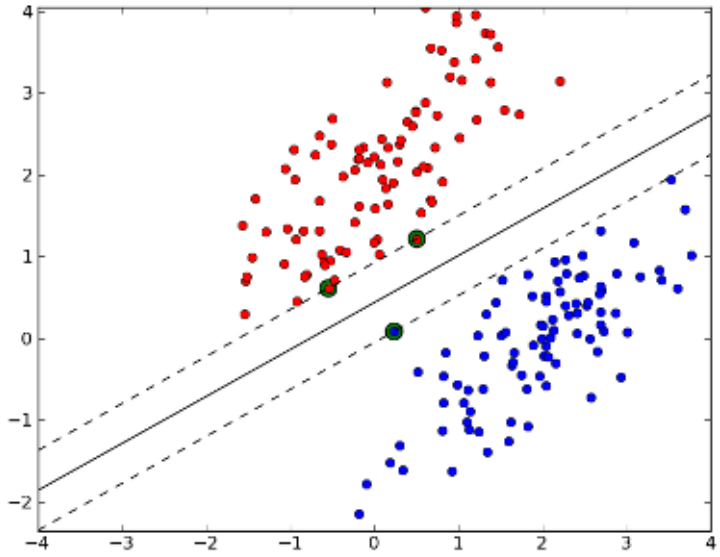
$$\forall i, \alpha_i(y_i(\mathbf{w}^\top \mathbf{x_i} + b) - 1) = 0$$

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

```
In [48]: Image('sv.png', width=400)
```



3.2.10 Dual problem

Solving the dual problem:

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

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

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

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

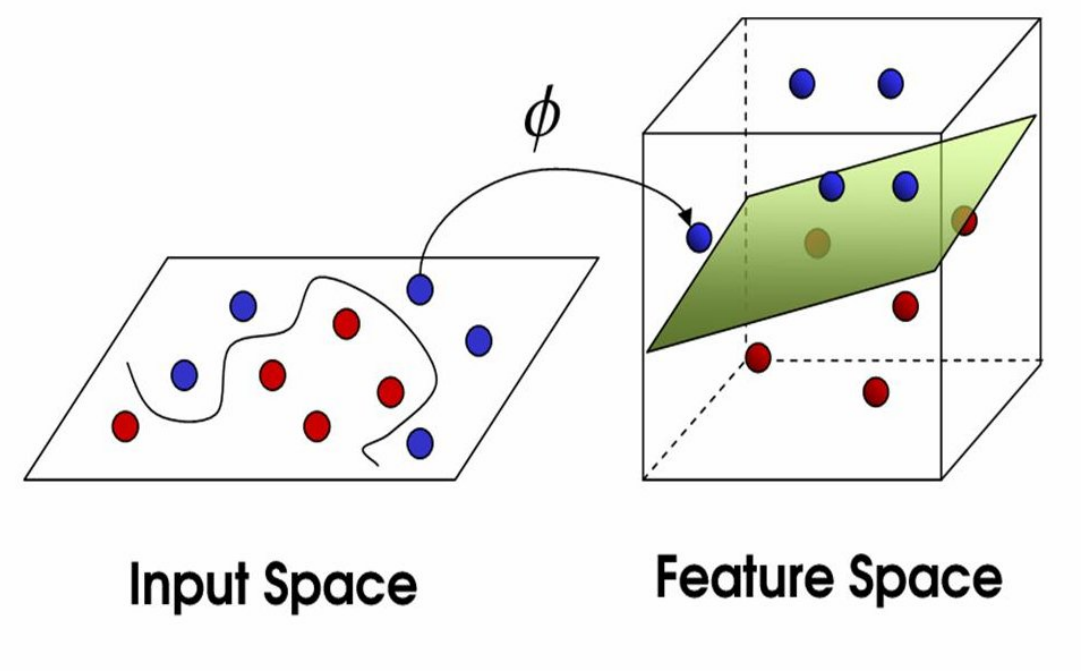
3.3 Kernels

- 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

```
In [49]: Image('map.jpg', width=400)
```



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

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

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

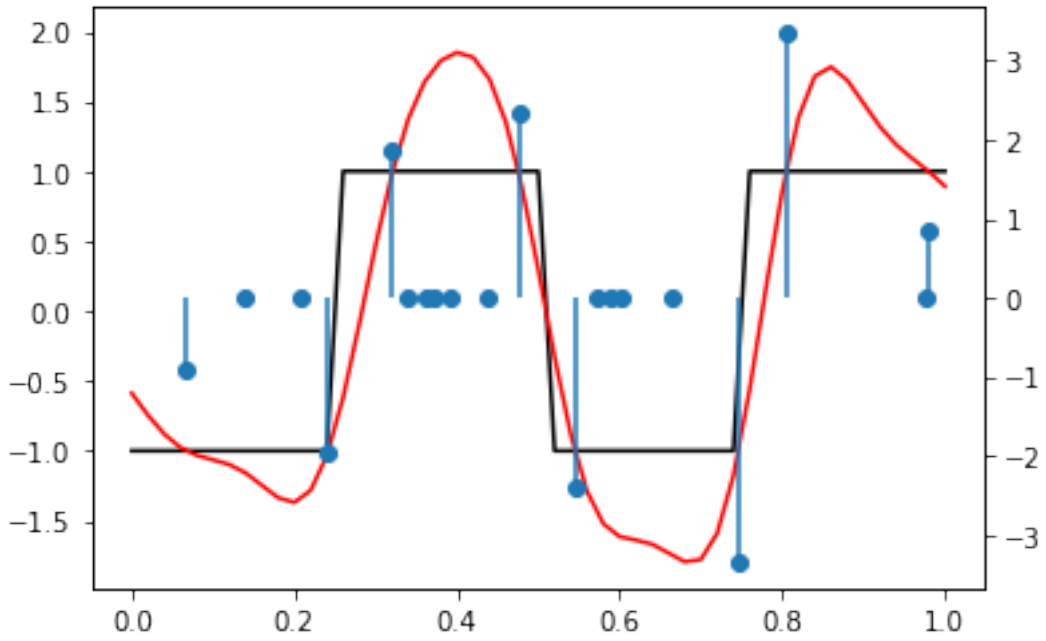
Second order ascent using diagonal Hessian approximation

3.3.9 Toy test

```
In [50]: 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 [51]: 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 [52]: def f_pred(x, alpha, x_train, gamma=10.0):
             K = GaussKernel(x, x_train, gamma)
             return jnp.dot(K, alpha)

In [53]: 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 [54]: n = 20
         gamma = 100.0

In [55]: key = jax.random.PRNGKey(42)
         x = jax.random.uniform(key, (n,1))
         y = [f_true(xi) for xi in x]

In [56]: 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 = jax.ops.index_update(alpha, i, 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=" ", use_line_collection=True)
                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

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

3.3.12 Representer theorem

Theorem (Schölkopf et al.): 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

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

3.3.15 MNIST

In [57]:
```
def SquareKernel(X1, X2):
    return jnp.matmul(X1, X2.T)**2
```

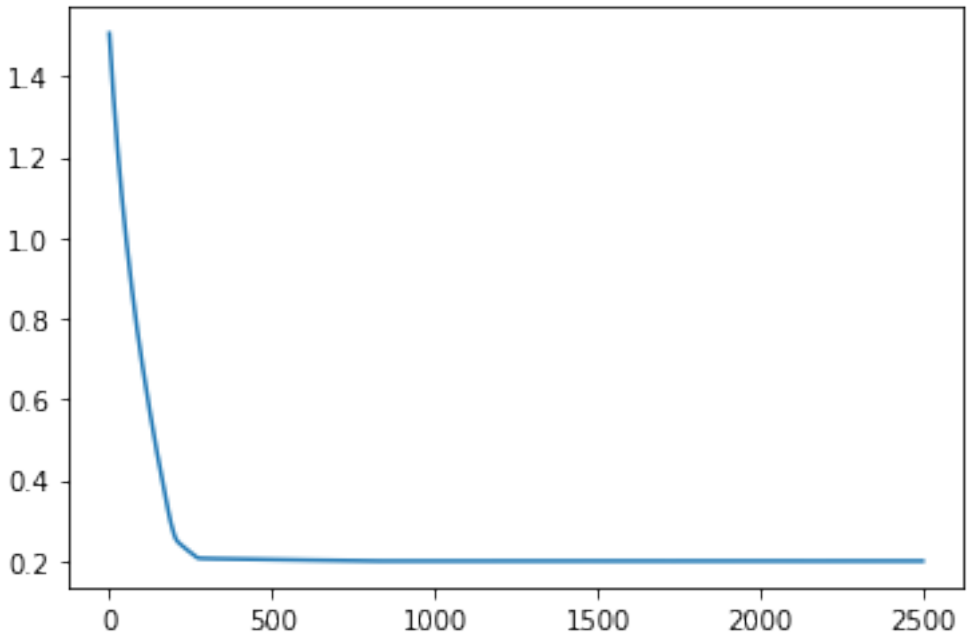
In [58]:
```
X = data['X_train']
X = X/jnp.linalg.norm(X, axis=1)[:,None]
y = jax.nn.one_hot(data['y_train'], 10)*2 - 1
K = SquareKernel(X, X)
L, U = jnp.linalg.eigh(K)
L = L[-64:]
U = U[:, -64:]
P = jnp.sqrt(1./L)[:, None]*U.T
```

In [59]:
```
X_bar = jnp.matmul(P, K).T
```

In [60]:
```
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 [61]:
```
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 0x7f5d05e735d0>]
```




```
In [62]: 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)))
```

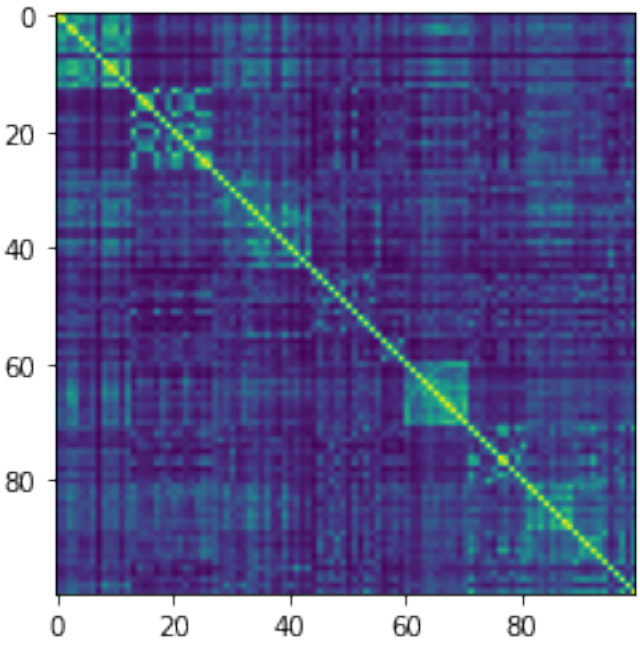
```
accuracy: 1.0
```

```
In [63]: 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: 0.6100000143051147
```

```
In [65]: 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 0x7f5d04d5a990>
```



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

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

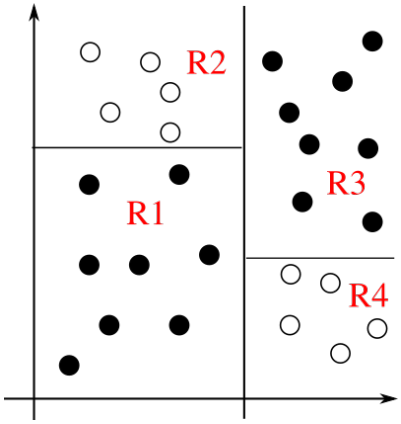
- Representer theorem: solution is in span($\mathcal{A}$)
- Combination is as large as the training set
- Support vectors for hinge loss
- Approximate kernel methods

Chapter 4

Decision Trees and ensembling methods

4.1 Region based classification

```
In [17]: Image(filename='region.png')
```

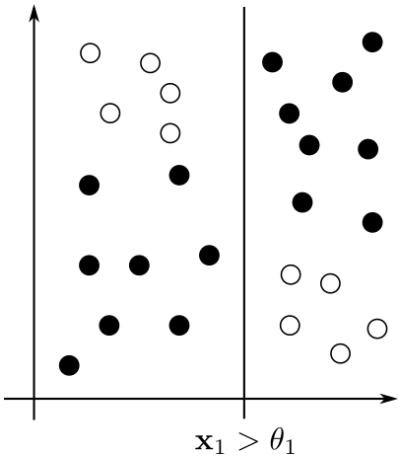


1. Memorizing all regions is cumbersome
   - R 1 = {(a 1 , b 1 , c 1 , d 1), y 1 }
   - R 2 = {(a 2 , b 2 , c 2 , d 2), y 2 }
   - ...
2. Classification requires to check all regions

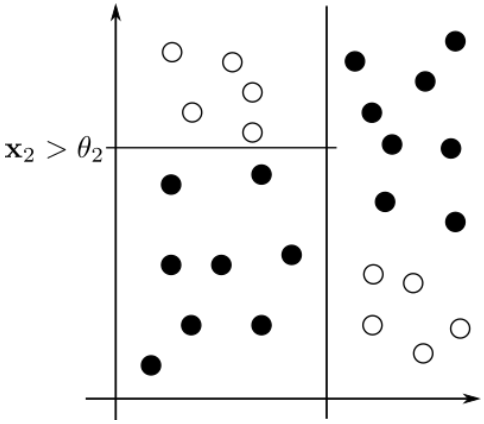
```
for region Ri in all regions:
  if x ∈ Ri:
    return yi
```

4.1.1 Tree based equivalent

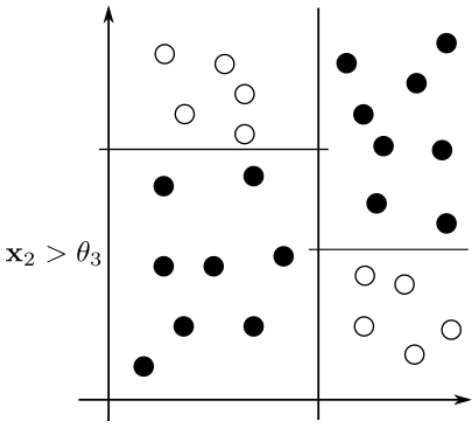
```
In [18]: Image(filename="tree1.png", width=400)
```



In [19]: Image(filename="tree2.png", width=400)

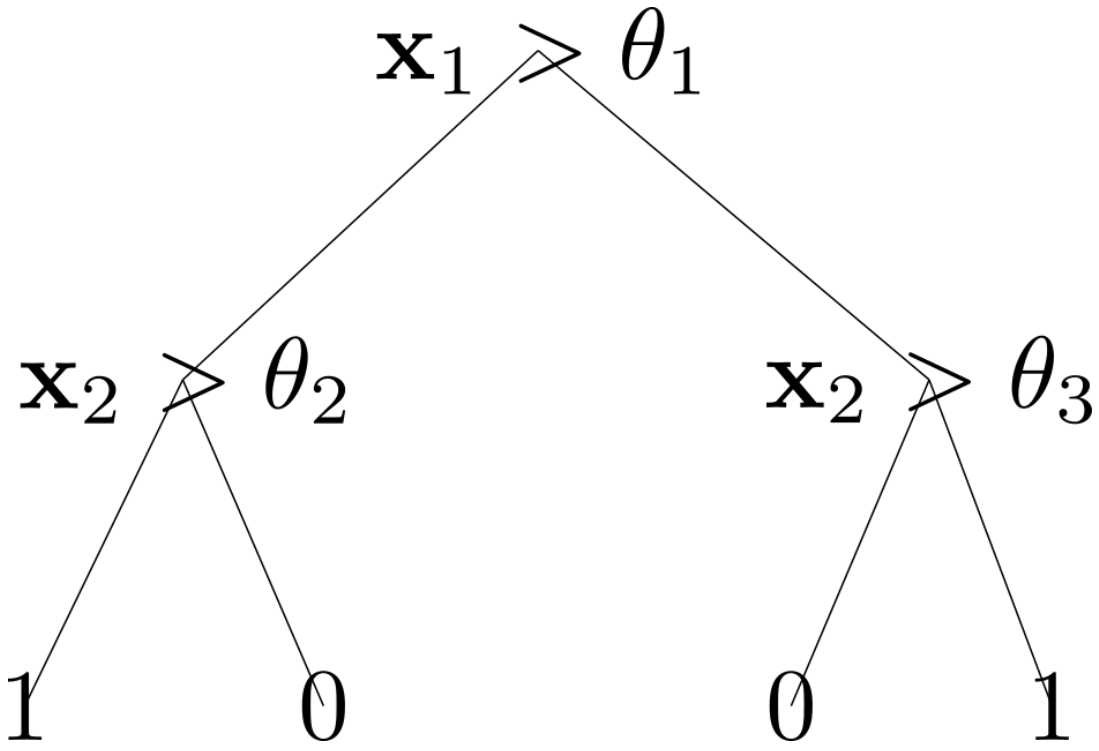


In [20]: Image(filename="tree3.png", width=400)



4.1.2 Tree representation

```
In [22]: Image(filename="dt.png", width=400)
```



4.1.3 Decision Tree

A decision tree is a hierarchical classifier with a tree structure where each node partitions the feature space along a specified component.

Leo Breiman (1928 - 2005)

```
In [24]: Image(filename="leo.jpg", width=400)
```

4.1.4 Growing the tree

1. Tree($\mathcal{S} = \{\mathbf{x}_i, y_i\}$):
2. if $|\mathcal{S}| < T$:
3. return Leaf($\text{argmax}_c \sum_i y_i$)
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($d^\star, \theta^\star, T_1, T_2$)

4.1.5 Gain measure

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

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

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

4.1.6 Information Gain

Other popular gain measures:

- Entropy

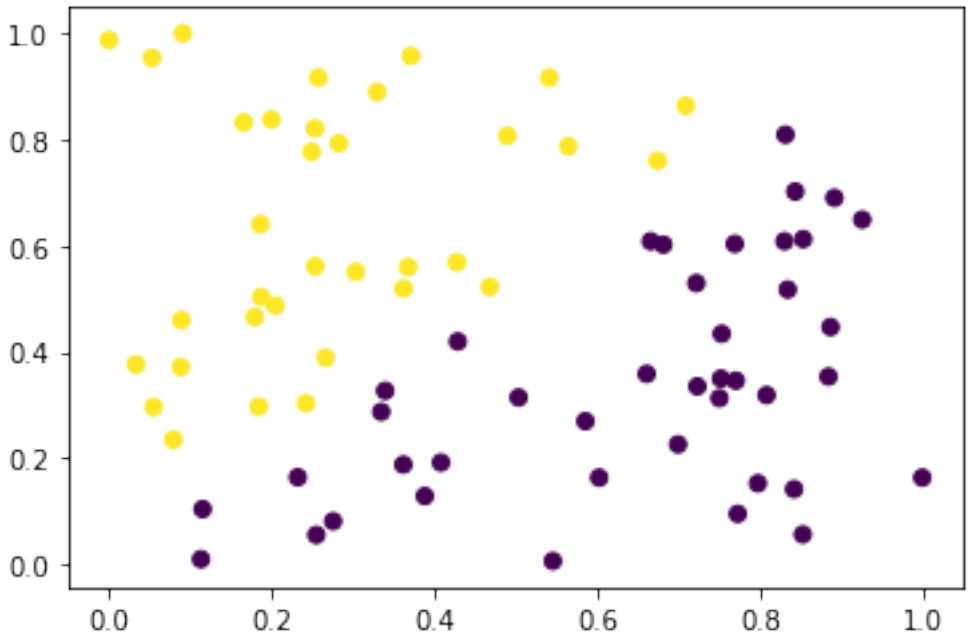
$$C(\mathcal{S}) = -\sum_k p_k(\mathcal{S}) \log p_k(\mathcal{S})$$

- Gini index

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

4.1.7 Small example

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

In [3]: plt.scatter(X[:,0], X[:,1], c=y)

<matplotlib.collections.PathCollection at 0x7fb6994a1890>
```




```
In [4]: 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 [5]: 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 [6]: 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 [7]: 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])
```

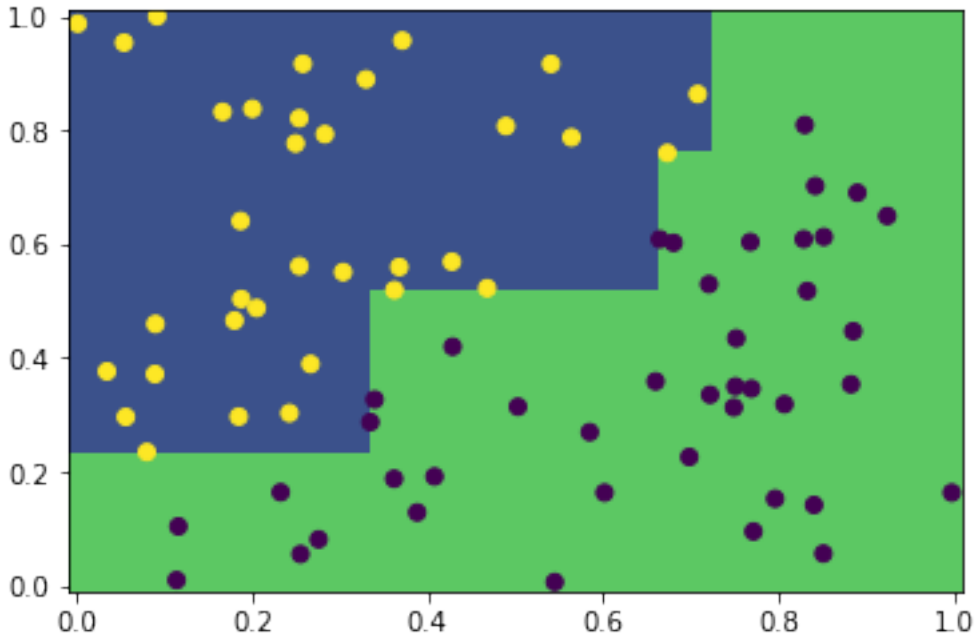
```
In [8]: T = BinaryClassificationTree(X, y)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and
rerun for more info.)
```

```
In [9]: 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 0x7fb694204250>
```



4.1.8 Decision Trees

- Interpretable
- Fast
- Handle categorical data

But

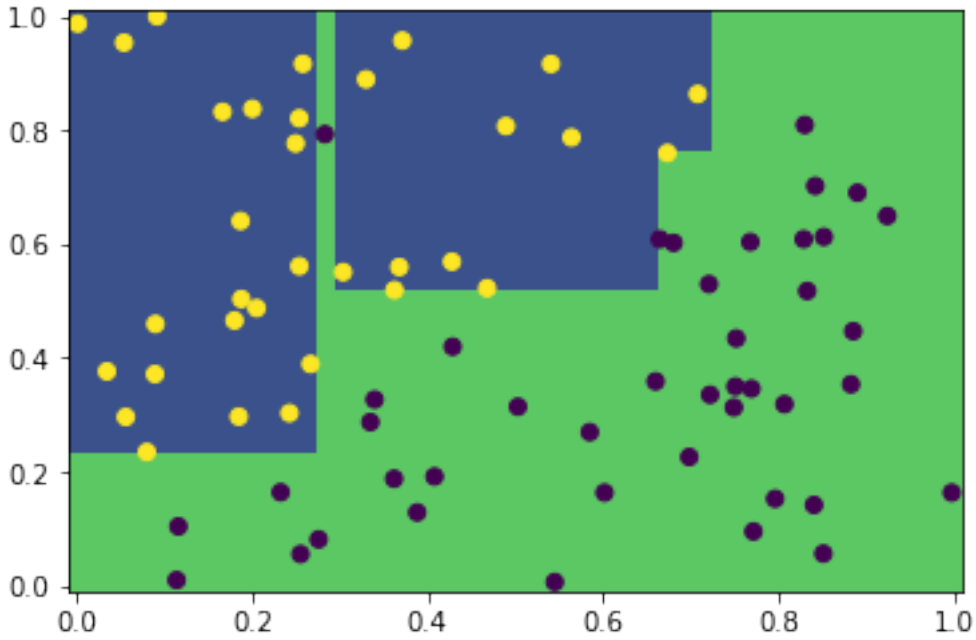
- Poor accuracy
- Unstable
- Need a lot of examples
- Finding the optimal tree is hard, growing is greedy

4.1.9 Unstable

```
In [17]: y[6] = 1 - y[6]
         T = BinaryClassificationTree(X, y)
```

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

```
<matplotlib.collections.PathCollection at 0x7fb6941195d0>
```



4.1.10 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 $\{0,1\}^d$?

4.2 Random Forest

Overcome DT instabilities by averaging B randomized trees

- 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

4.2.1 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

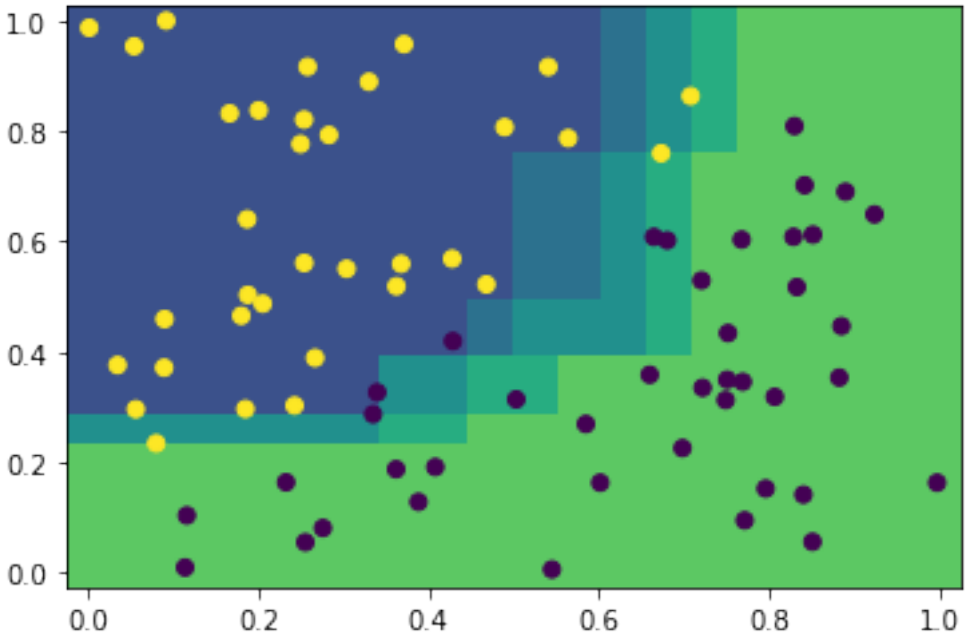
Theorem (Breiman): As the number of trees increases, for almost surely all sequences $\Theta_1, \ldots$, the generalization error R converges to

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

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

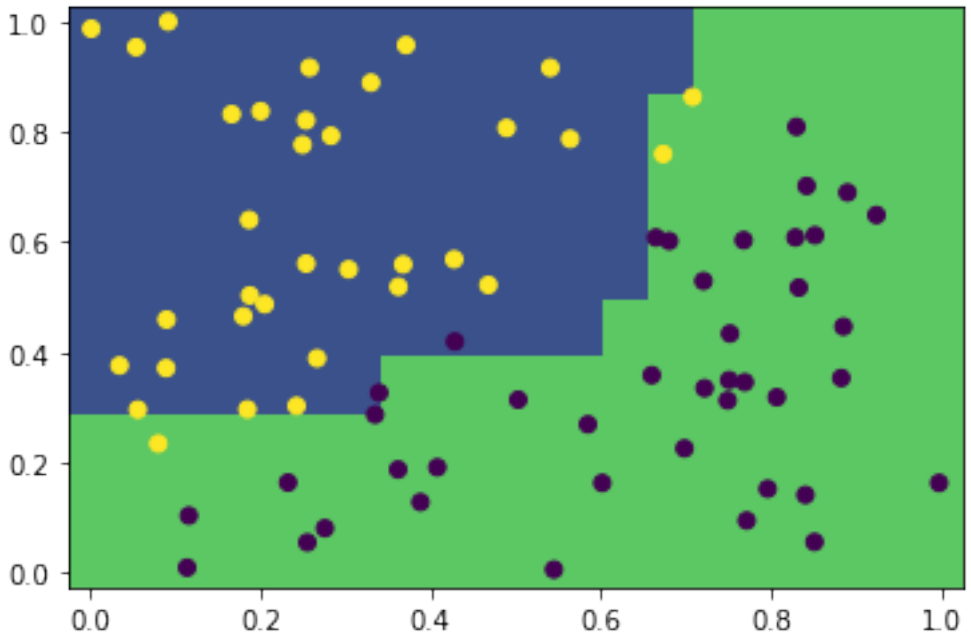
```
In [38]: class RandomForest():
             def __init__(self, X, y, nb_tree=25, p=0.5):
                 self.trees = []
                 n = len(y)
                 k = int(p*n)
                 for b in range(nb_tree):
                     i = np.random.permutation(n)
                     Xb = 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 [39]: T = RandomForest(X, y)

In [41]: 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 0x7fb65049fad0>
```




```
In [42]: 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)
```

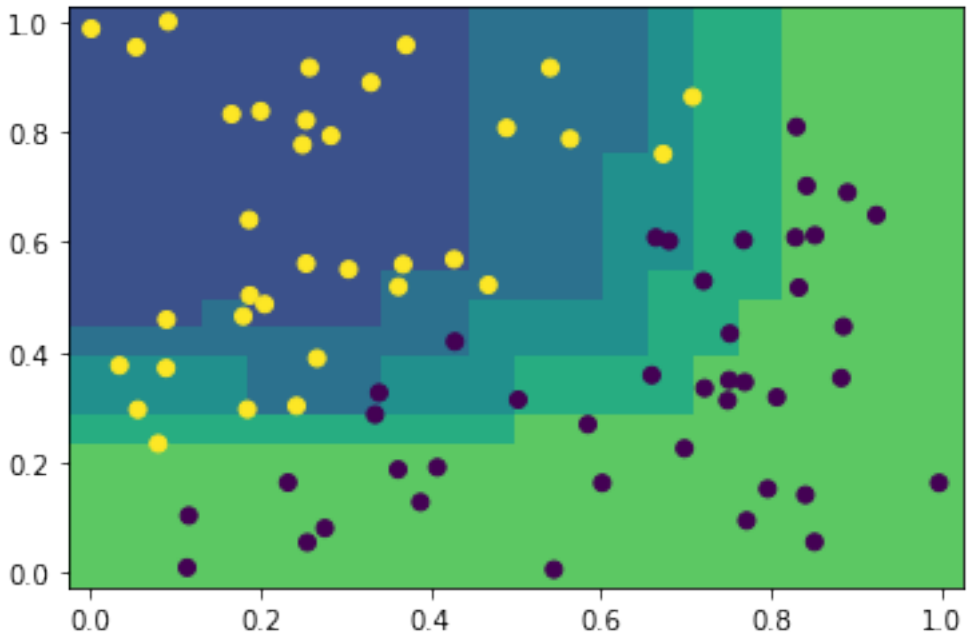
```
<matplotlib.collections.PathCollection at 0x7fb6402955d0>
```



In [43]:
```
T = RandomForest(X, y, nb_tree=100, p=0.2)
```

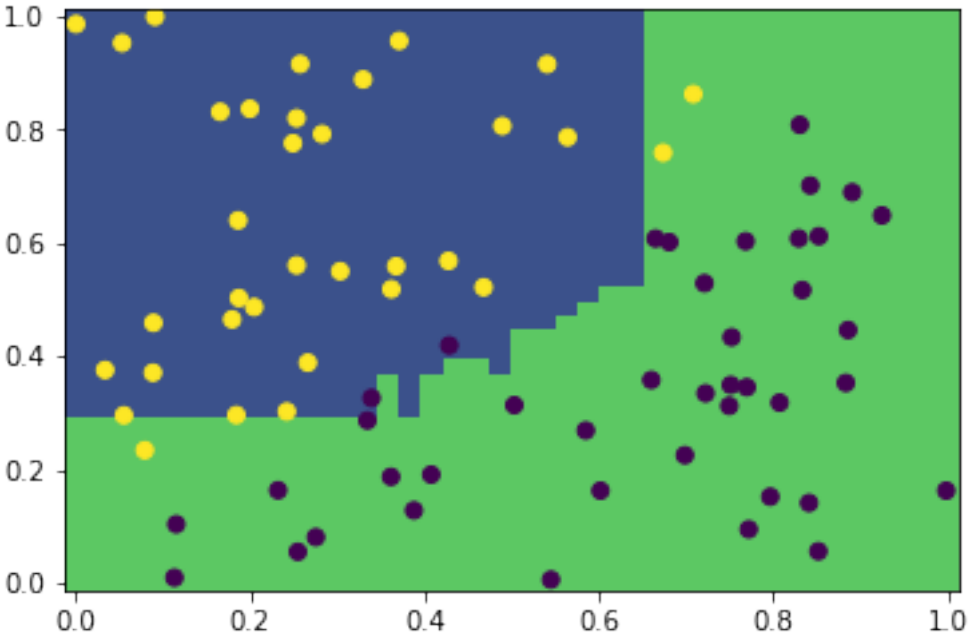
In [44]:
```
t = 50; tx = jnp.linspace(0, 1, t); ty = jnp.linspace(0, 1, t)
xv, yv = jnp.meshgrid(tx, ty, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
y_pred = jnp.array(T(xx)).reshape(t, t)
cmap = plt.get_cmap('PiYG')
levels=jnp.linspace(-1.5, .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 0x7fb64022a450>
```




```
In [48]: 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 0x7fb6501a0050>
```



4.2.2 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

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

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

$$Var\left(\frac{\sum_i x_i}{B}\right)=\frac{1}{B^2}Var\left(\sum_i x_i\right)$$
$$=\frac{1}{B^2}\left(\mathbb{E}\left[\left(\sum_i x_i\right)^2\right]-\mathbb{E}\left[\sum_i x_i\right]^2\right)$$
$$=\frac{1}{B^2}\left(\sum_{i,j}\mathbb{E}\left[x_ix_j\right]-\left(\sum_i\mathbb{E}\left[x_i\right]\right)^2\right)$$
$$=\frac{1}{B^2}\left(B(\sigma^2+m^2)+(B^2-B)(\rho\sigma^2+m^2)-B^2m^2\right)$$
$$Var\left(\frac{\sum_i x_i}{B}\right)=\frac{1}{B^2}\left(B(\sigma^2+m^2)+(B^2-B)(\rho\sigma^2+m^2)-B^2m^2\right)$$
$$=\frac{\sigma^2+m^2}{B}+\rho\sigma^2+m^2-\frac{\rho\sigma^2+m^2}{B}-m^2$$
$$=\frac{\sigma^2}{B}+\rho\sigma^2-\frac{\rho\sigma^2}{B}$$
$$=\rho\sigma^2+\frac{1-\rho}{B}\sigma^2$$
$$\leq\sigma^2 \text{ iff } \rho<1 \text{ and } B>1$$

4.3 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

4.3.1 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

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

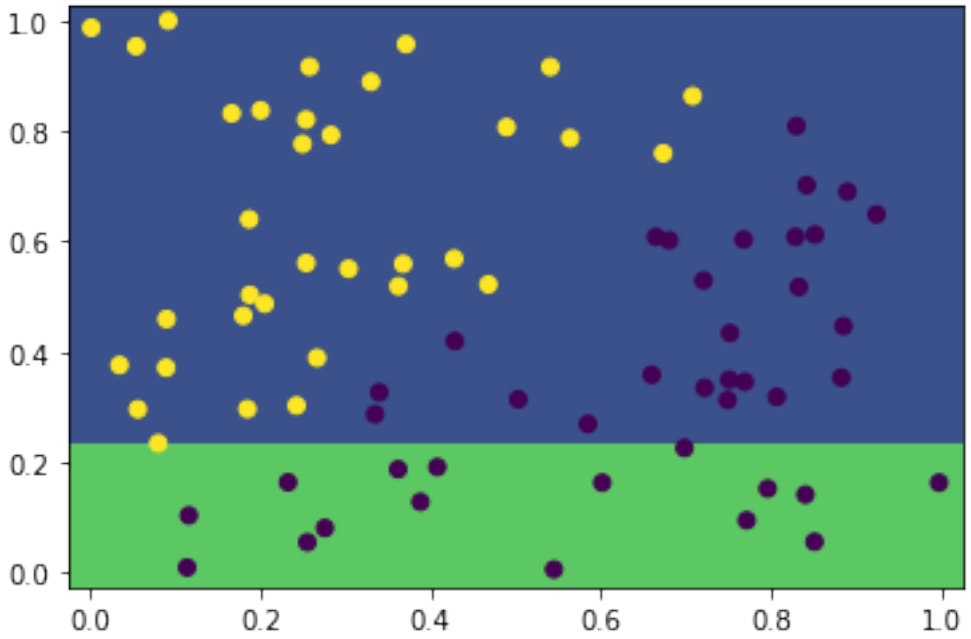
Corresponds to a voting strategy

4.3.2 Exemple

Random axis, select optimal threshold

```
In [103]: 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 [104]: 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 [105]: T = BaggingClassifier(X, y, 1)

In [106]: 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)
```

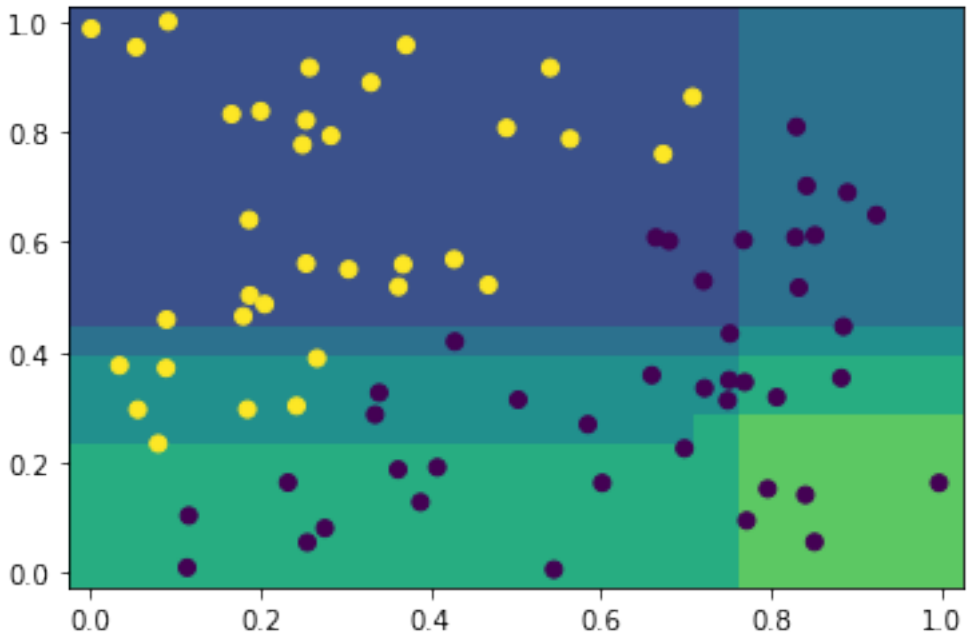
```
<matplotlib.collections.PathCollection at 0x7fb64005ae90>
```




```
In [107]: T = BaggingClassifier(X, y, 10)
```

```
In [108]: 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)
```

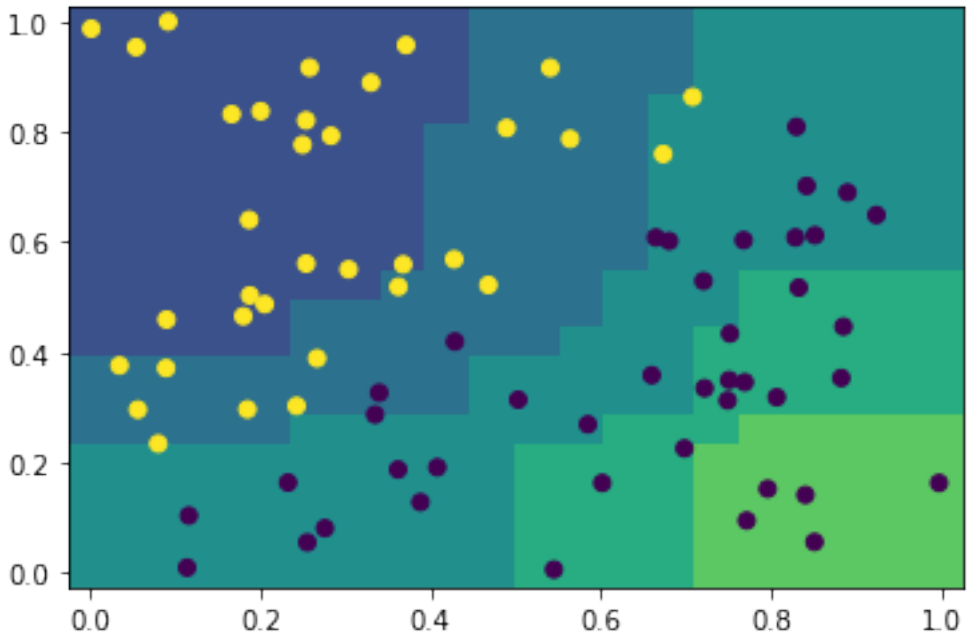
```
<matplotlib.collections.PathCollection at 0x7fb60872c990>
```



In [113]:
```
T = BaggingClassifier(X, y, 100, p=0.2)
```

In [114]:
```
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 0x7fb6084f7810>
```



4.4 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?

4.4.1 Adaboost

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

4.4.2 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

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

4.4.3 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

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

4.4.4 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

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

4.4.5 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

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

Thus

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

4.4.6 Adaboost

1. Initialize $\forall i, w_i = 1/N$
2. For $m = 1 \dots M$
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 $G(\mathbf{x}) = \sum_m \beta_m G_m(\mathbf{x})$

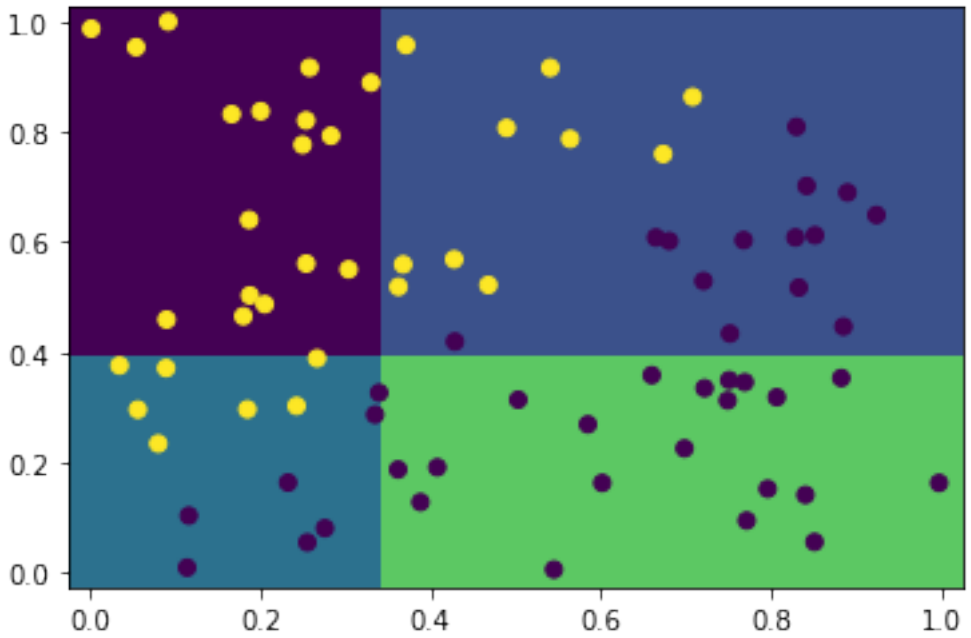
```
In [226]: def weightedError(w, y_pred, y_true):
              return (w * (y_pred != y_true)).sum()/w.sum()

In [227]: 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 [228]: 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.thet

In [236]: 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 [239]:
```
T = AdaBoost(X, y, 2)
```

In [240]:
```
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)
```

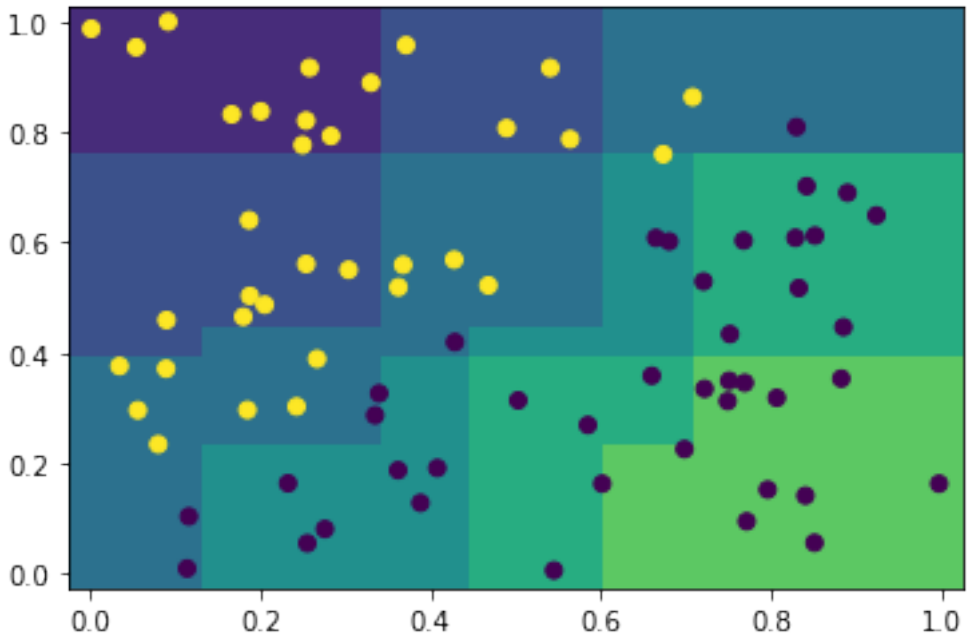
```
<matplotlib.collections.PathCollection at 0x7fb5f8560910>
```



In [241]:
```
T = AdaBoost(X, y, 10)
```

In [242]:
```
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)
```

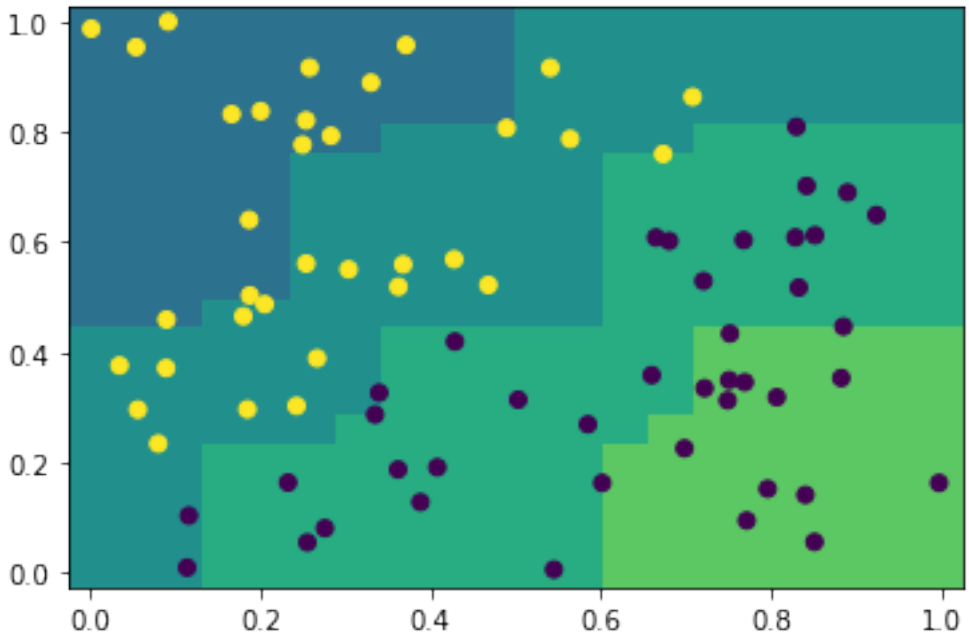
```
<matplotlib.collections.PathCollection at 0x7fb5f860a5d0>
```



In [243]:
```
T = AdaBoost(X, y, 50)
```

In [244]:
```
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 0x7fb5f8511d50>
```



4.4.7 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

4.4.8 Exercise

Show that

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

4.5 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

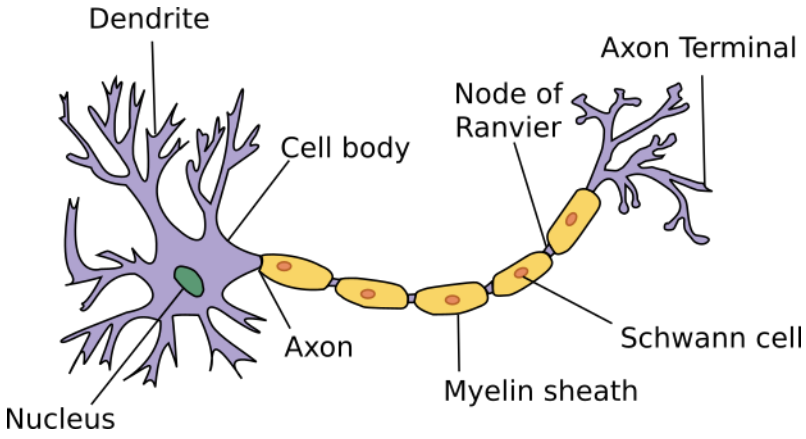
- Bossted trees: very good default classifier in many cases

Chapter 5

Neural Networks

5.1 Natural neuron

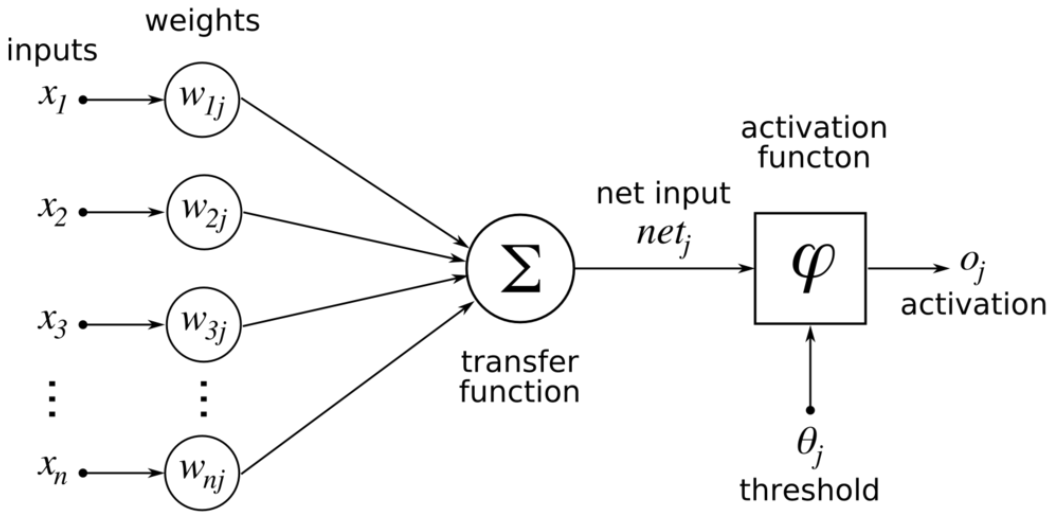
```
In [2]: Image('Neuron.png', width=400)
```



5.2 Artificial Neuron (McCulloch & Pitts)

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

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



5.2.1 Activation functions

Linear: $\varphi(x) = x$

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

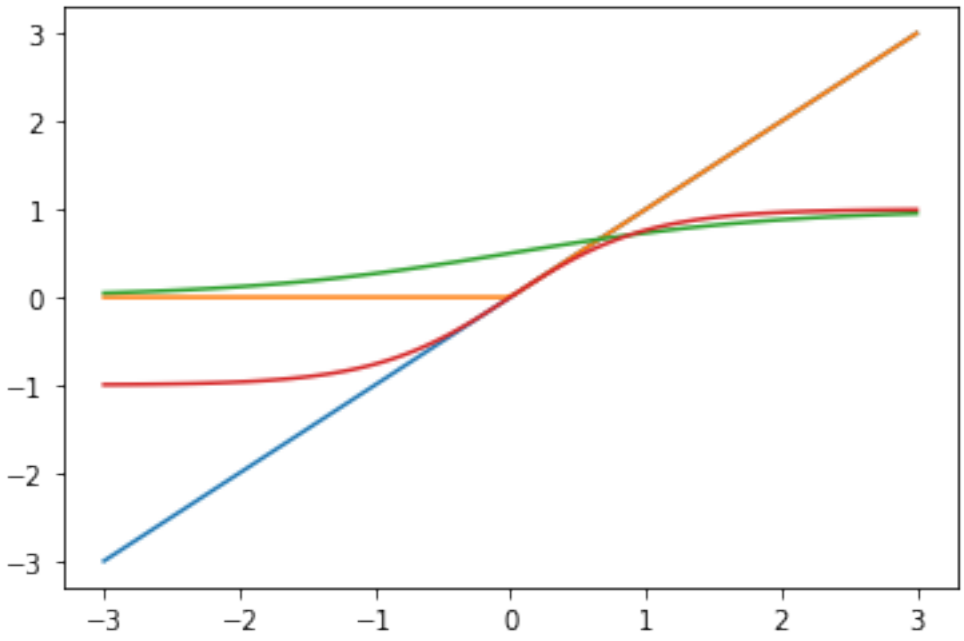
Sigmoid: $\varphi(x) = \frac{1}{1+e^{-x}}$

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')
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and
rerun for more info.)
```

```
[<matplotlib.lines.Line2D at 0x7fd5f428e110>]
```



5.2.2 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

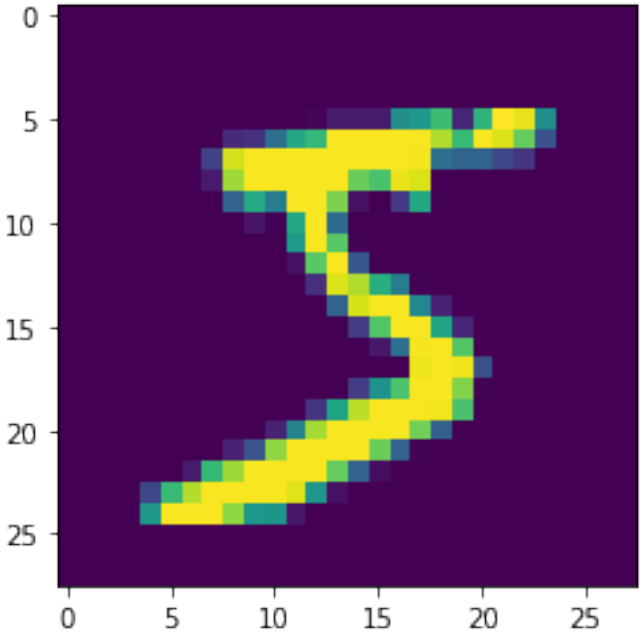
$$\delta = \frac{1}{|B|} \sum_{(\mathbf{x}_i, y_i) \in B} \frac{\partial l(y_i, f(\mathbf{x}_i)}{\partial \mathbf{w}}$$

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

5.2.3 Small example

```
In [7]: # 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)

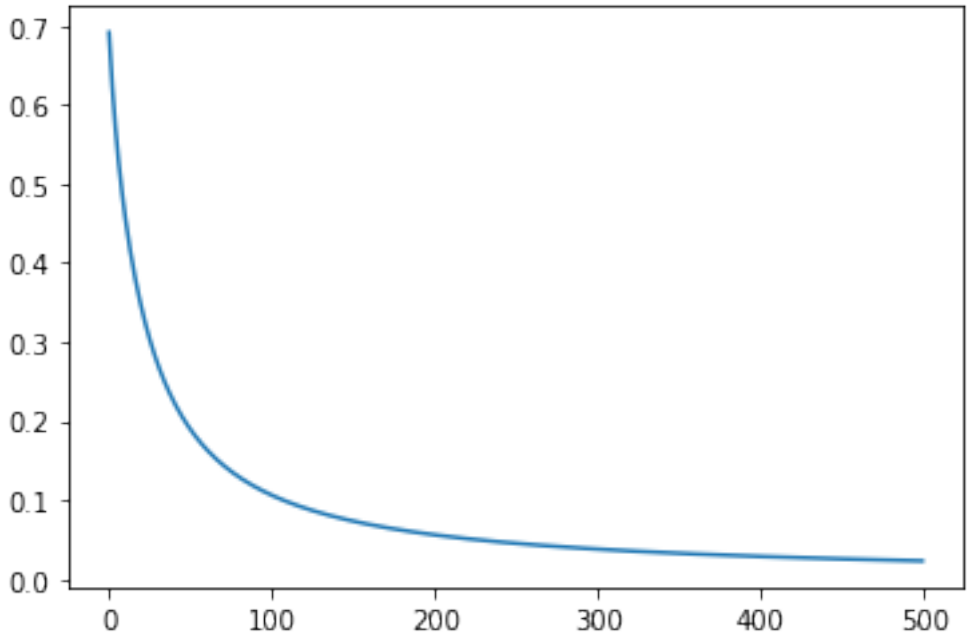
$$\mathcal{L} = -y \log f(\mathbf{x}) - (1-y) \log(1 - f(\mathbf{x}))$$

```
In [8]: 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 [9]:
```
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 0x7fd5ec740990>]
```



In [10]:
```
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)))
```

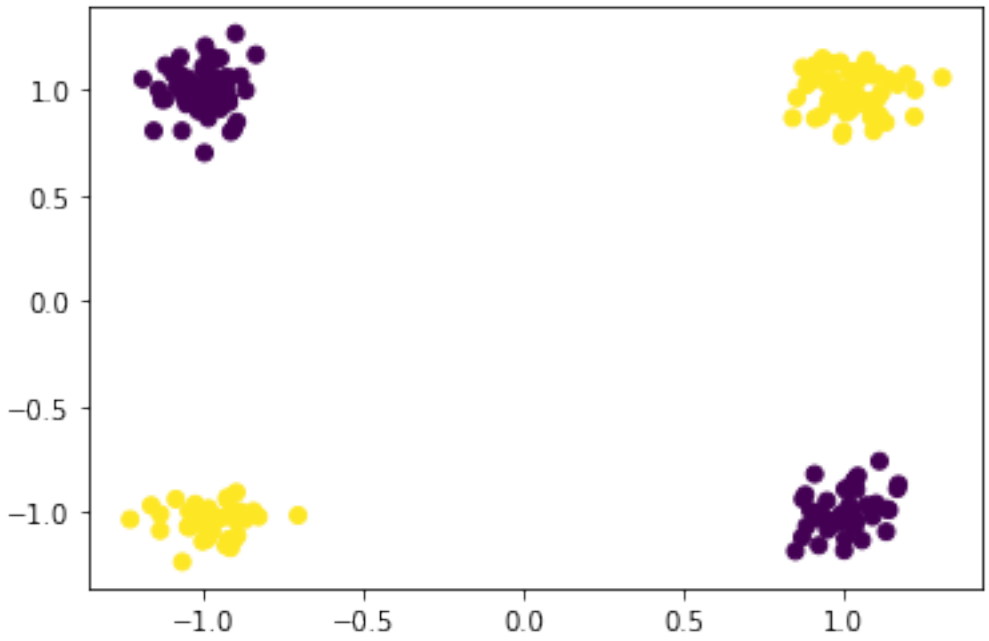
```
accuracy: 1.0
```

Non linearly separable problems

What about XOR?

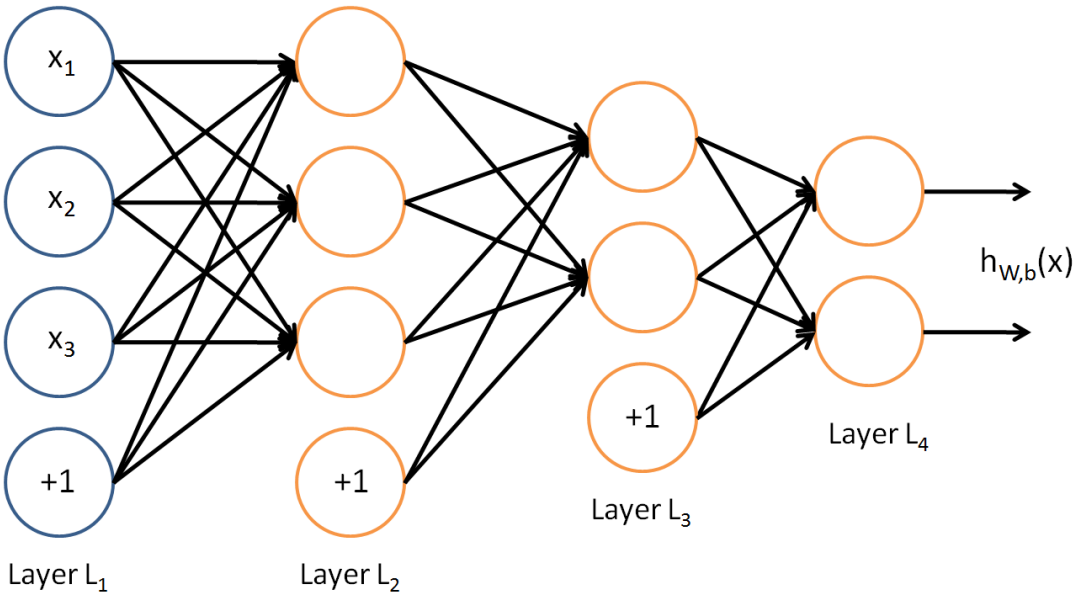
In [11]:
```
Xor = jnp.sign(np.random.randn(200, 2)) + 0.1*np.random.randn(200,2)
yor = 1.*((Xor[:,0]*Xor[:,1])>0)

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

```
<matplotlib.collections.PathCollection at 0x7fd5ec6bfe10>
```



Multiple layer

```
In [12]: Image('mlp.png', width=400)
```



5.3 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

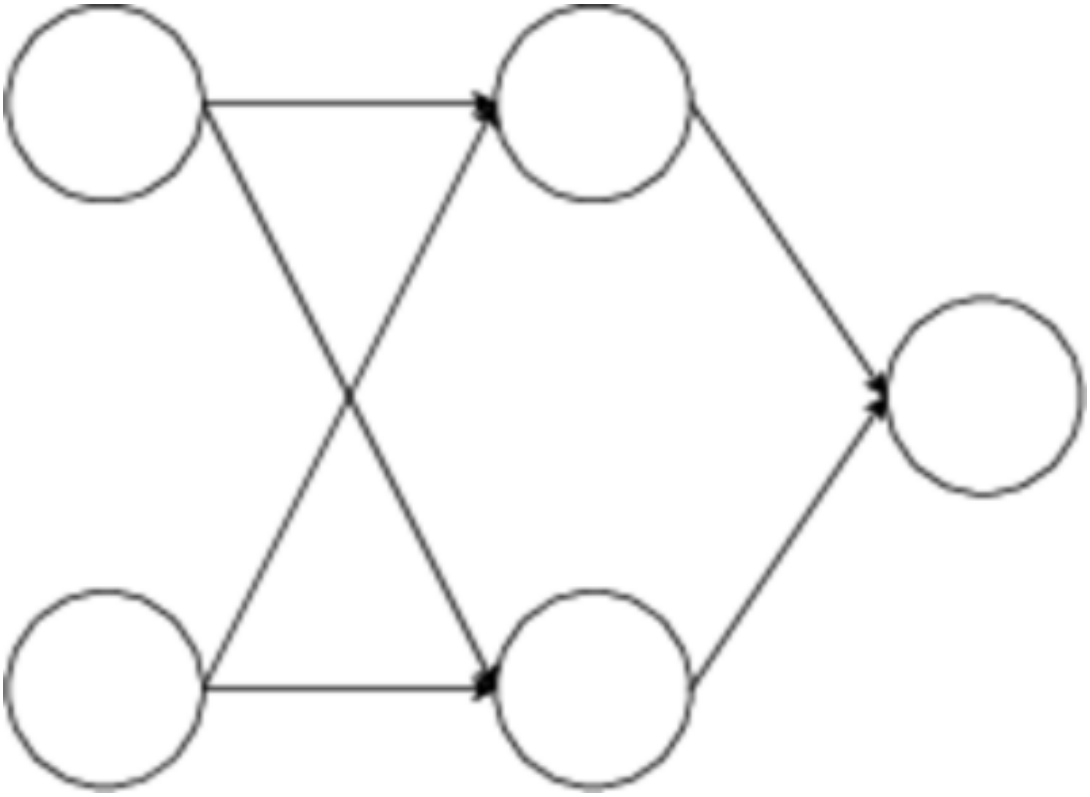
$$F(\mathbf{x}) = f_L \circ \cdots \circ f_1(\mathbf{x})$$

5.3.1 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 [14]: Image('2_xor.png', width=400)
```



5.3.2 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

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

5.3.3 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 $\forall k, F_k = g_k(\sigma(\mathbf{w}_k^\top \mathbf{x}_{k-1} + \theta_k))$

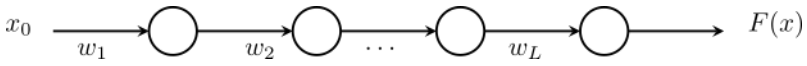
Chain rule (Leibnitz notation)

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

5.3.4 Backpropagation

Single neuron chain

```
In [15]: Image('neural_chain.png', width=600)
```



$$\frac{\partial l(y,F(\mathbf{x}))}{\partial w_k} = \frac{\partial l(y,F(\mathbf{x}))}{\partial F(\mathbf{x})}\frac{\partial F(\mathbf{x})}{\partial w_k}$$

$$= l'(y,F(\mathbf{x}))\frac{\partial \sigma(w_L \mathbf{x}_{L-1} + \theta_L)}{\partial w_k} \tag{5.1}$$

$$= l'(y,F(\mathbf{x}))\sigma'(\mathbf{x}_L)\frac{\partial w_L \mathbf{x}_{L-1} + \theta_L}{\partial w_k} \tag{5.2}$$

$$= l'(y,F(\mathbf{x}))\sigma'(\mathbf{x}_L) w_L \frac{\partial \mathbf{x}_{L-1}}{\partial w_k} \tag{5.3}$$

$$\frac{\partial \mathbf{x}_{L-1}}{\partial w_k} = \frac{\partial \sigma(w_{L-1}\mathbf{x}_{L-2} + \theta_{L-1})}{\partial w_k} \tag{5.4}$$

$$= \sigma'(\mathbf{x}_{L-1}) w_{L-1}\frac{\partial \mathbf{x}_{L-2}}{\partial w_k} \tag{5.5}$$

Recursion

$$\frac{\partial \mathbf{x}_{k+t+1}}{\partial w_k} = \sigma'(\mathbf{x}_{k+t+1}) w_{k+t+1}\frac{\partial \mathbf{x}_{k+t}}{\partial w_k} \tag{5.6}$$

$$\frac{\partial \mathbf{x}_k}{\partial w_k} = \sigma'(\mathbf{x}_k)\mathbf{x}_k \tag{5.7}$$

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

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

$$\delta_L = l(y, F(\mathbf{x}))\sigma'(\mathbf{x}_L) \tag{5.8}$$

$$\delta_k = \mathbf{w}_{k+1}(\sigma'(\mathbf{x}_{k+1}) \circ \delta_{k+1}) \tag{5.9}$$

$$\frac{\partial l(y, F(\mathbf{x}))}{\partial \mathbf{w}_k} = \delta_k \circ \mathbf{x}_k \tag{5.10}$$

5.3.5 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 $l'(y, F(\mathbf{x})), \delta_k, \mathbf{x}_k$

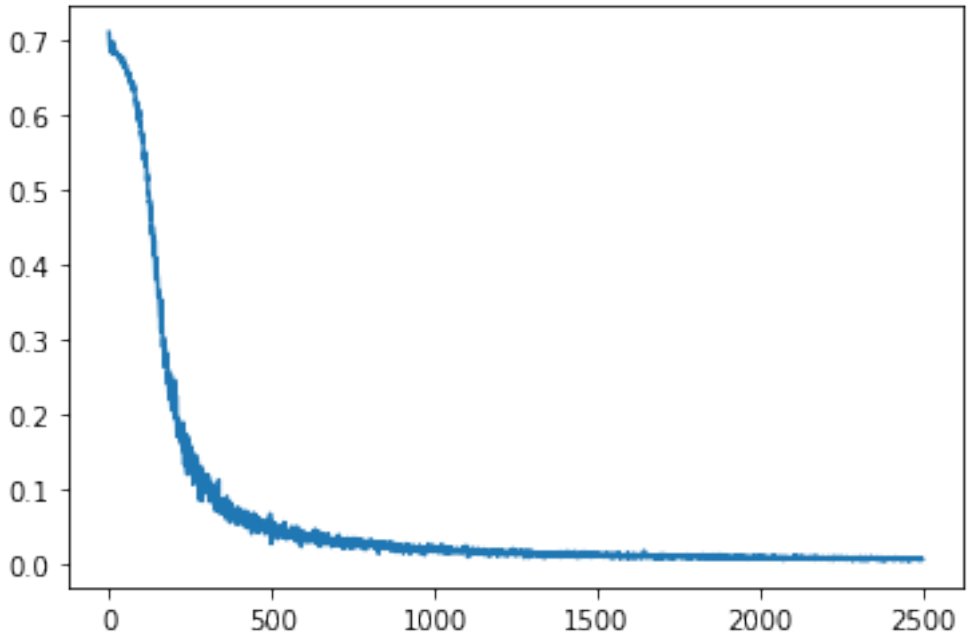
In practice, ML libraries have auto-grad features (pytorch, tensorflow, jax, etc) for certain operators that you compose to build F

5.3.6 XOR with MLP

```
In [16]: 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
```

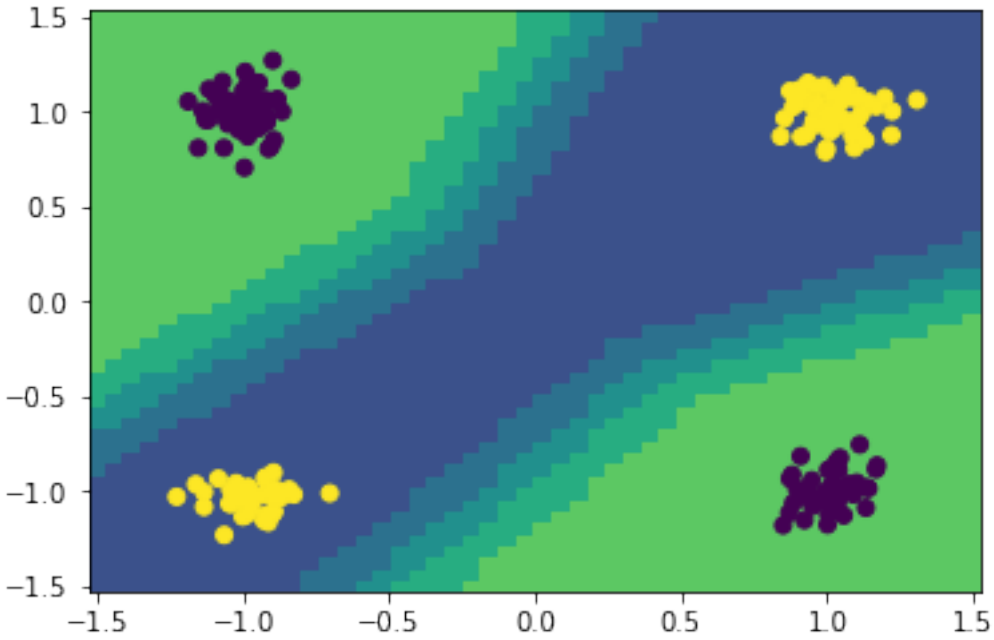
In [17]:
```
w1 = np.random.randn(2, 4)/10
w2 = np.random.randn(4)/10
b1 = np.random.randn(4)/10
b2 = np.random.randn(1)/10

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 0x7fd5ec0ae210>]
```



In [18]:
```
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)
```

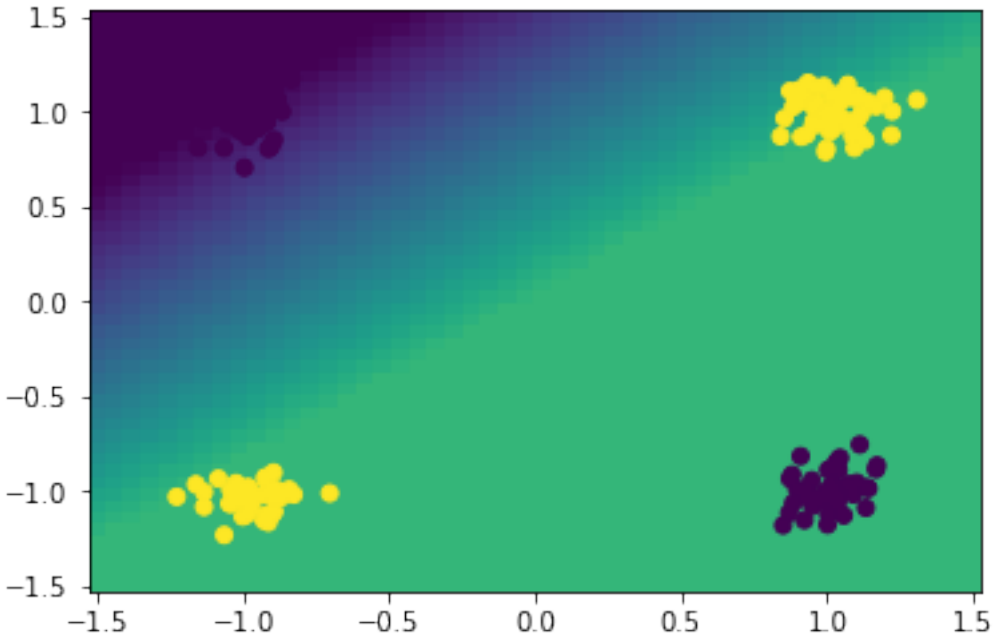
```
<matplotlib.collections.PathCollection at 0x7fd5cc7f0b90>
```



Intermediate layers

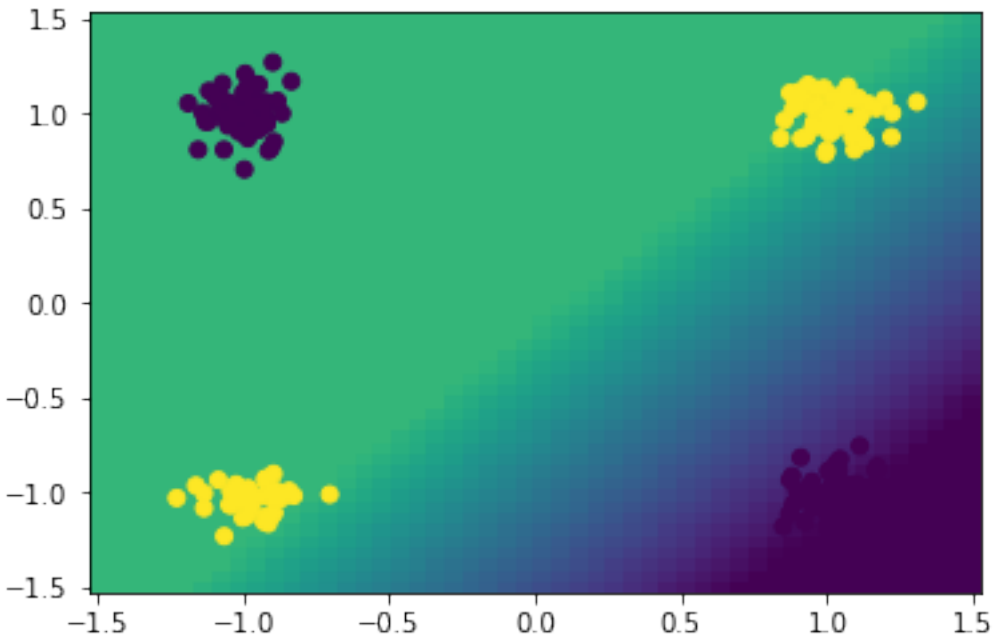
```
In [19]: t = 50; tx = jnp.linspace(-1.5, 1.5, t);
         xv, yv = jnp.meshgrid(tx, tx, sparse=True); xv = xv.squeeze(); yv = yv.squeeze()
         xx = jnp.array([[xx, yy] for yy in yv for xx in xv])
         y_pred = jnp.array(l1(w1, b1, xx)).reshape(t, t, 4)
         cmap = plt.get_cmap('PiYG')
         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 0x7fd5cc718b50>
```



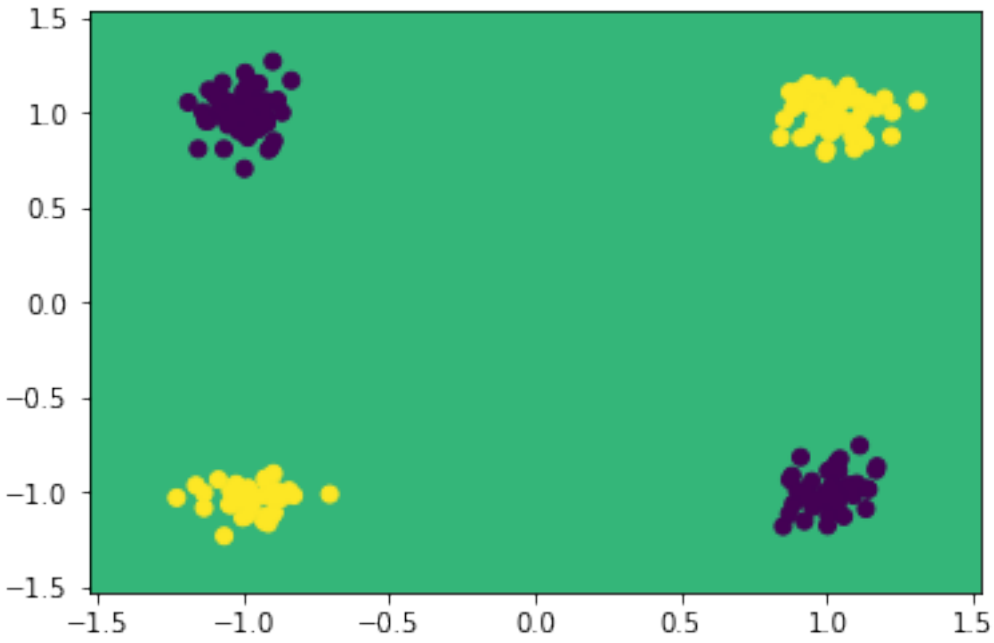
In [20]:
```
levels=jnp.linspace(-4., 2., 100)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred[:,:,1], shading='nearest', norm=norm);
plt.scatter(Xor[:,0], Xor[:,1], c=yor)
```

```
<matplotlib.collections.PathCollection at 0x7fd5cc683e10>
```



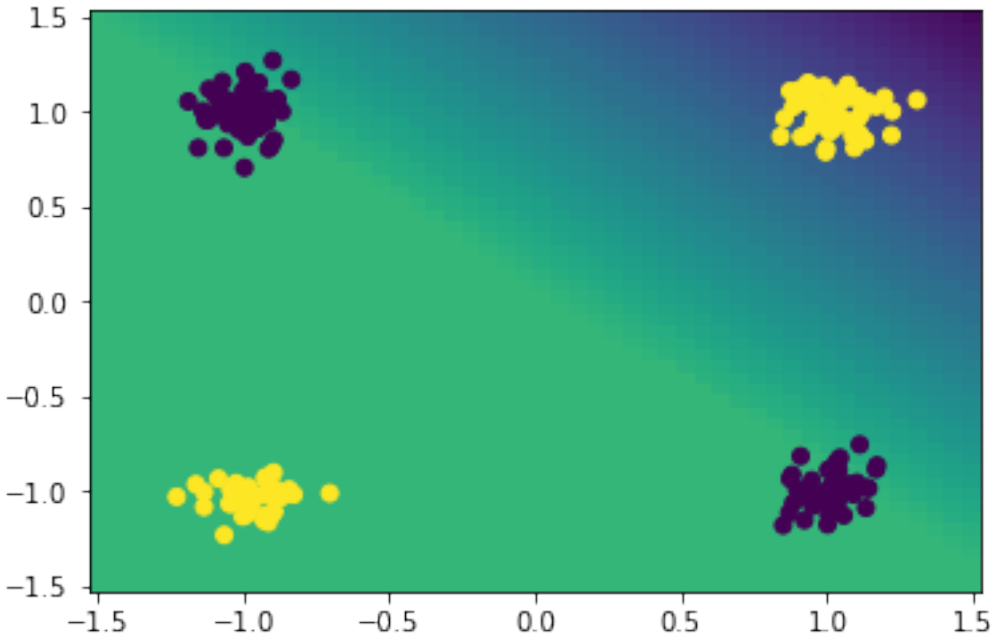
In [21]:
```
levels=jnp.linspace(-4., 2., 100)
norm = matplotlib.colors.BoundaryNorm(levels, ncolors=cmap.N, clip=True)
plt.pcolormesh(xv, yv, -y_pred[:,:,2], shading='nearest', norm=norm);
plt.scatter(Xor[:,0], Xor[:,1], c=yor)
```

```
<matplotlib.collections.PathCollection at 0x7fd5cc5fa350>
```



In [22]:
```
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 0x7fd5ec74e490>
```



5.3.7 Neural networks losses

Classification

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

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{5.11}$$

$$l(y, F(x))) = -\sum_i y_i \log F(\mathbf{x})[i] \tag{5.12}$$

Regression

- Usual ℓ_2, ℓ_1 losses

5.4 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

5.4.1 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

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

5.4.2 Neural network capacity

Theorem (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.

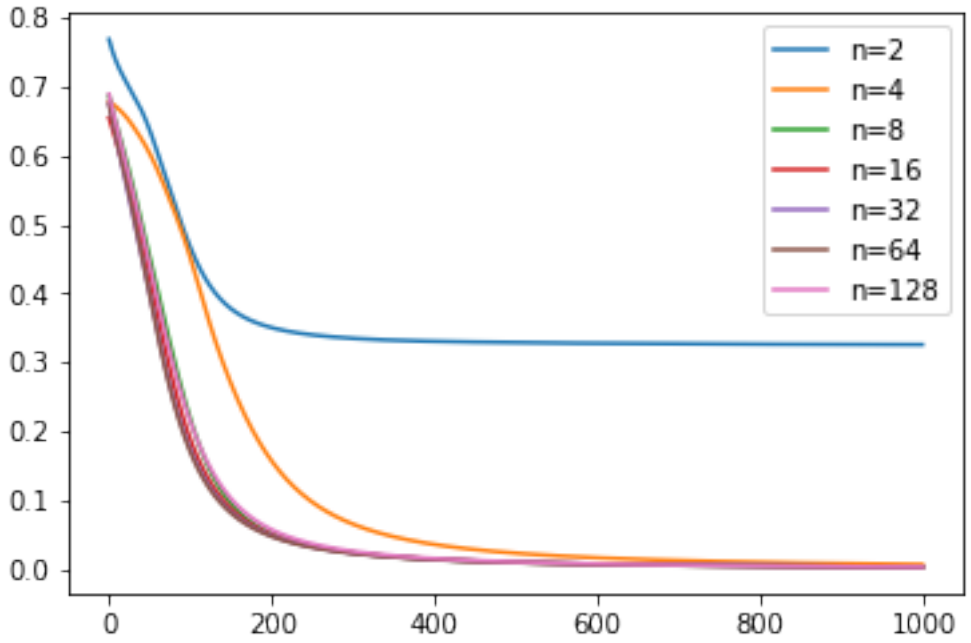
5.4.3 VC dimension

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.

5.4.4 Effect of width

```
In [23]: 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 [24]: for n in range(1, 8):
             plt.plot(train_nn(2**n), label='n={}'.format(2**n))
         plt.legend()

<matplotlib.legend.Legend at 0x7fd5ec6506d0>
```

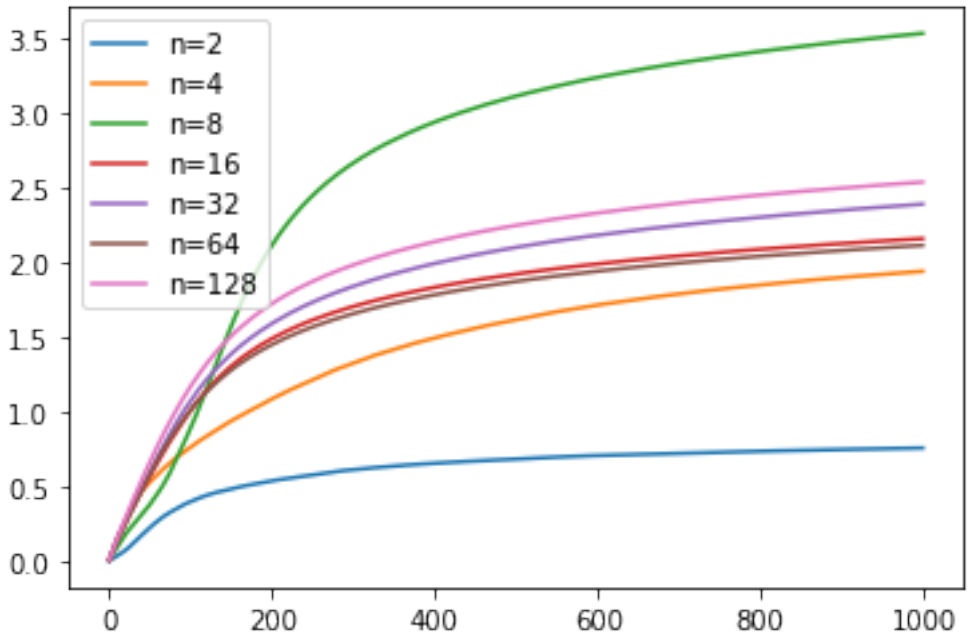


Larger NN, easier to train?

```
In [25]: 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 [26]: for n in range(1, 8):
             plt.plot(train_nn2(2**n), label='n={}'.format(2**n))
         plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fd5f404d150>
```



Larger networks tend to change less than smaller networks?

5.4.5 Neural Tangent Kernel

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

5.4.6 Lottery ticket hypothesis

The Lottery Ticket Hypothesis 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.

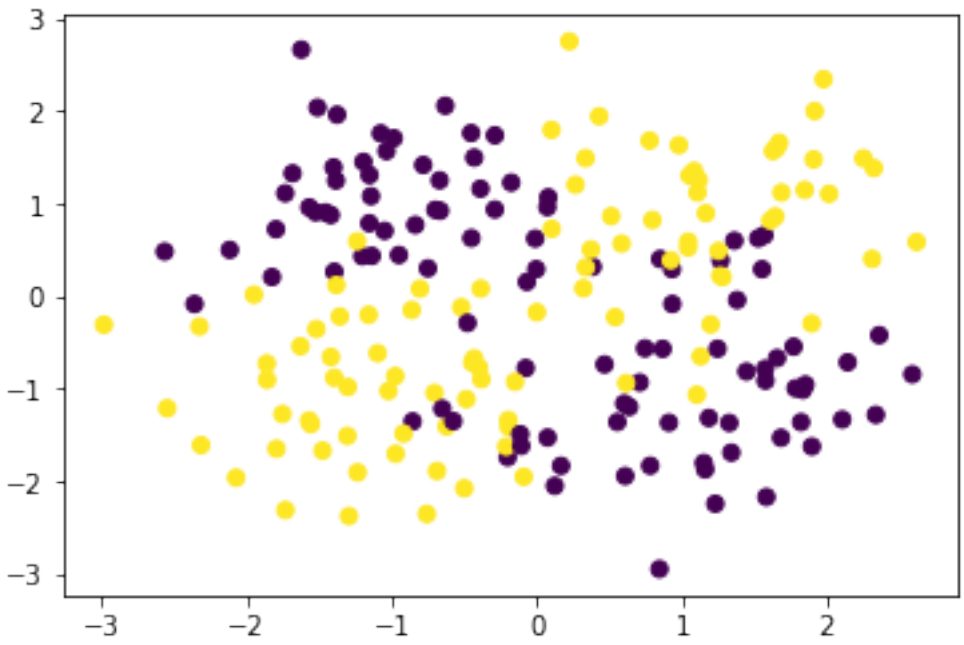
- Pruning a randomly initialized network can yield a good predictor (verified empirically).
- In practice, pruning is difficult, better train the full model.

5.4.7 Double Descent

Do larger networks systematically overfit?

```
In [27]: 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 0x7fd5cc3e7f50>
```

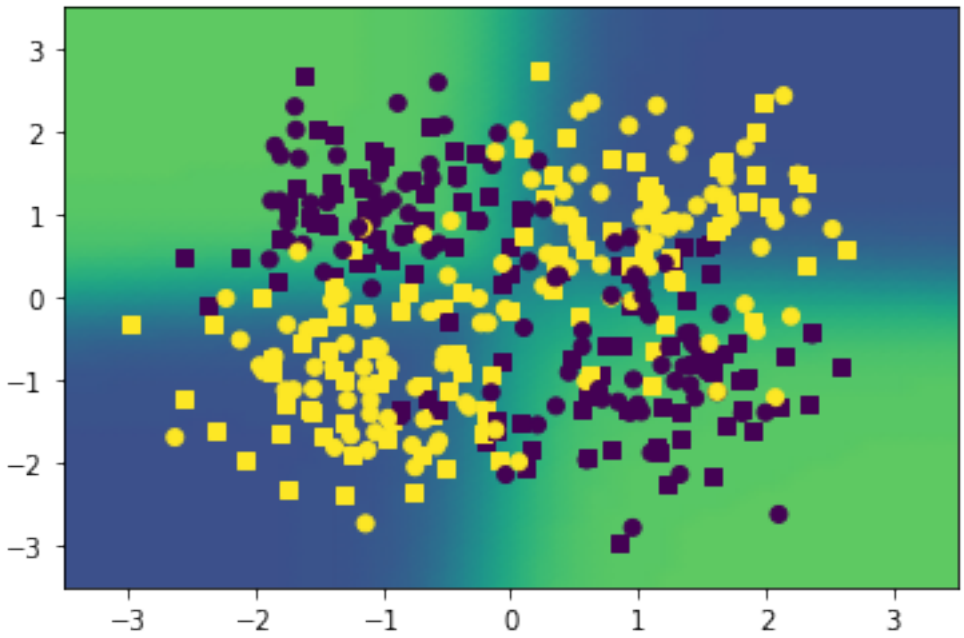



```
In [28]: n = 10000
         w1 = np.random.randn(2, n)/(jnp.sqrt(n))
         w2 = np.random.randn(n)/(jnp.sqrt(n))
         b1 = np.random.randn(n)/(jnp.sqrt(n))
         b2 = np.random.randn(1)/(jnp.sqrt(n))

         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 [29]: 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='gouraud', 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 0x7fd5cc4982d0>
```



Double descent phenomenon

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

5.5 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 6

Clustering, Metric Learning and PAC Theory

6.1 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)

6.1.1 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

$$\max_f Pr[\mathcal{A}] = \sum_{\mathbf{x}_i} \log f(\mathbf{x}_i)$$

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

$$\frac{\partial \sum_{\mathbf{x}_i} \|\mathbf{x}_i - \mu\|^2}{\partial \mu} = 0 \tag{6.1}$$

$$2n\mu - 2\sum_{\mathbf{x}_i} \mathbf{x}_i = 0 \tag{6.2}$$

$$\mu = \frac{1}{n}\sum_{\mathbf{x}_i} \mathbf{x}_i \tag{6.3}$$

For σ

$$\min_{\sigma} \sum_{\mathbf{x}_i} \|\mathbf{x}_i - \mu\|^2/2\sigma^2 + n\log\sqrt{2\pi}\sigma$$

$$\frac{\partial \sum_{\mathbf{x}_i} \|\mathbf{x}_i - \mu\|^2/2\sigma^2 + n\log\sqrt{2\pi}\sigma}{\partial \sigma} = 0 \tag{6.4}$$

$$-\frac{\sum_{\mathbf{x}_i} \|\mathbf{x}_i - \mu\|^2}{\sigma^3} + \frac{n}{\sigma} = 0 \tag{6.5}$$

$$\sigma^2 = \frac{\sum_{\mathbf{x}_i} \|\mathbf{x}_i - \mu\|^2}{n} \tag{6.6}$$

6.1.2 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

6.1.3 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:

$$\max_{\pi,\mu,\Gamma} E[\log f(\mathbf{x})] = \sum_i \sum_k h_i^k \log f_k(\mathbf{x}_i)$$

$$\pi_k = \frac{\sum_i h_i^k}{n} \tag{6.7}$$

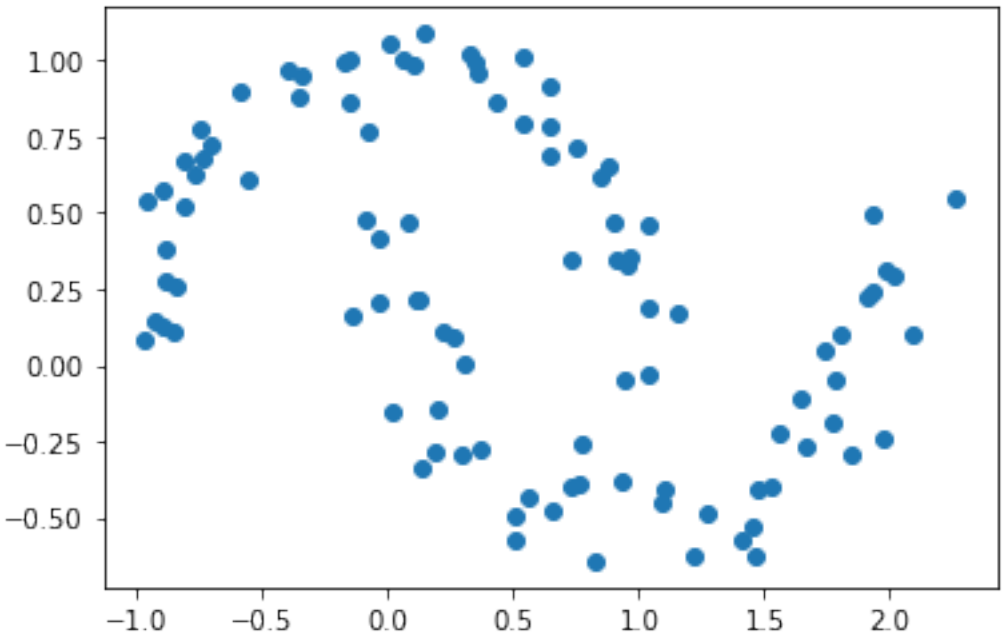
$$\mu_k = \frac{\sum_i h_i^k \mathbf{x}_i}{\sum_i h_i^k} \tag{6.8}$$

$$\Gamma_k^{-1} = \frac{\sum_i h_i^k (\mathbf{x}_i - \mu_k)(\mathbf{x}_i - \mu_k)^\top}{\sum_i h_i^k} \tag{6.9}$$

```
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 0x7f57d071d190>
```




```
In [166]: def Estep(X, w, mu, s):
              xmu = X[:, None, :] - mu[None, :, :] # n x 5 x 2
              sxm = xmu/(1e-7+s[None,:,:]) # n x 5 x 2
              xmsxm = w * jnp.exp(-(xmu * sxm).sum(2)) # n x 5
              return xmsxm

          def Mstep(X, h):
```

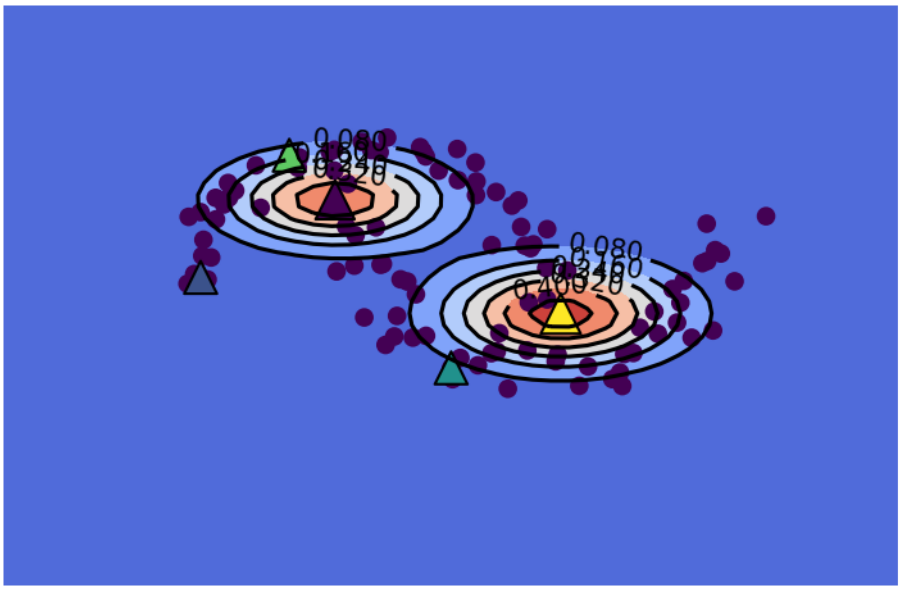
```
    w = h.mean(axis=0)
    m = (h[:,:,None]*X[:,None,:]).sum(axis=0) / h.sum(axis=0)[:,None] # 5 x 2
    xm = X[:,None,:] - m[None,:,:] # n x 5 x 2
    s = (h[:,:,None]*xm*xm).sum(axis=0) / h.sum(axis=0)[:,None] # 5 x 2
    return w, m, s
```

```
In [199]: np.random.seed(4)
          w = np.ones(5)/5.
          m = 0.5*np.random.randn(5, 2)+0.5
          s = 0.5*jnp.ones((5,2))
```

```
In [211]: 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 [201]: fig = plt.figure(dpi=150)
          plt.axis('off')
          camera = Camera(fig)
          for i in range(50):
              plot_density(X, w, m, s)
              h = Estep(X, w, m, s)
              h = h / (1e-4+h.sum(axis=1, keepdims=True))
              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>
```



6.1.4 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:

$$\mathbf{w} = \sum_i \alpha_i \phi(\mathbf{x}_i) \tag{6.10}$$

$$\sum_i \alpha_i = 1 \tag{6.11}$$

$$0 \leq \alpha_i \leq C \tag{6.12}$$

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 [279]: def gauss_kernel(X1, X2, gamma=5.):
              D = (X1[:,None,:] - X2[None,:,:])**2
              return jnp.exp(-gamma*D.sum(axis=2))
```

```
In [280]: def loss(alpha, X):
              K = gauss_kernel(X, X)
              return 0.5 * (alpha[None,:] @ (K @ alpha[:,None])).squeeze()
```

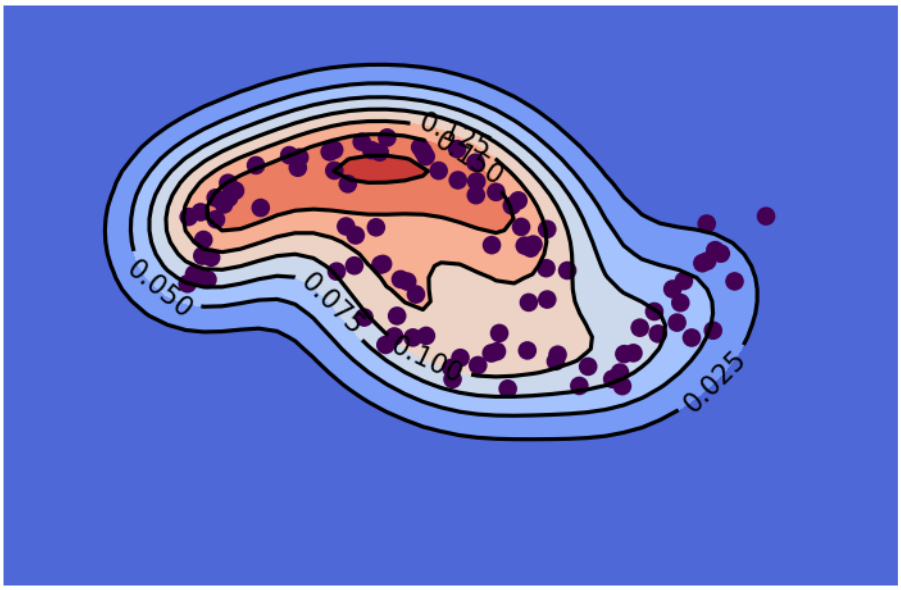
```
In [293]: @jax.jit
          def update(alpha, X, C = 1., eta=0.1):
              da = jax.grad(loss, argnums=0)(alpha, X)
              a = jnp.clip(alpha + eta*da, 0, C)
              return a/a.sum()
```

```
In [294]: 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 [308]: 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))
```

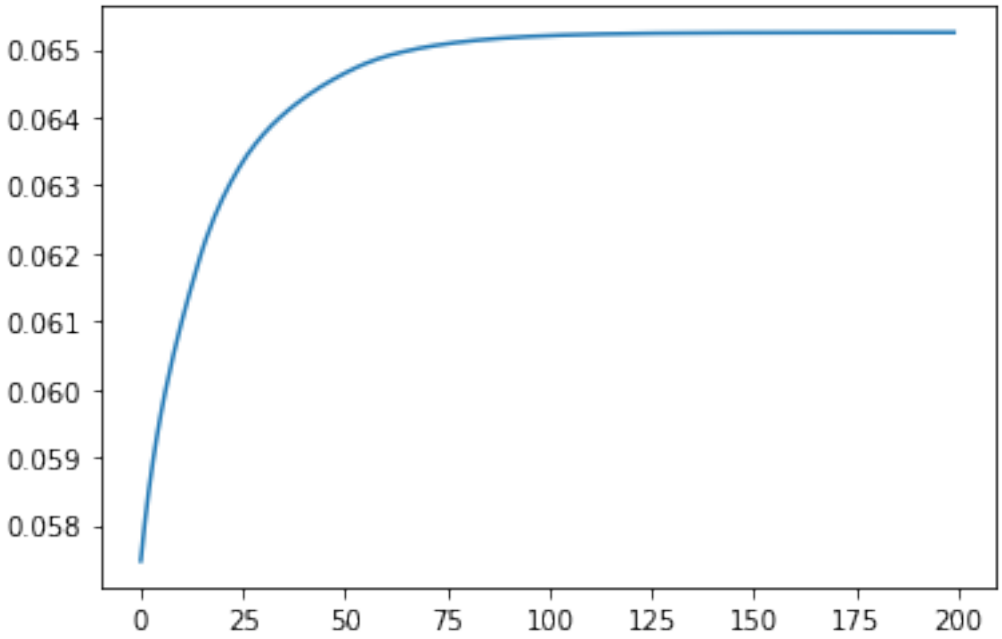
```
In [311]: fig = plt.figure(dpi=150); plt.axis('off'); camera = Camera(fig)
          alpha = np.ones(100)/100; l = []
          for i in range(200):
              y_pred = gauss_kernel(xx, X)@alpha[:,None]
              y_pred = jnp.array(y_pred).reshape(t, t)
              plot_density(X, xx, y_pred)
              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 [312]: plt.plot(l)
```

```
[<matplotlib.lines.Line2D at 0x7f56df650ed0>]
```



6.2 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

6.2.1 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

6.2.2 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

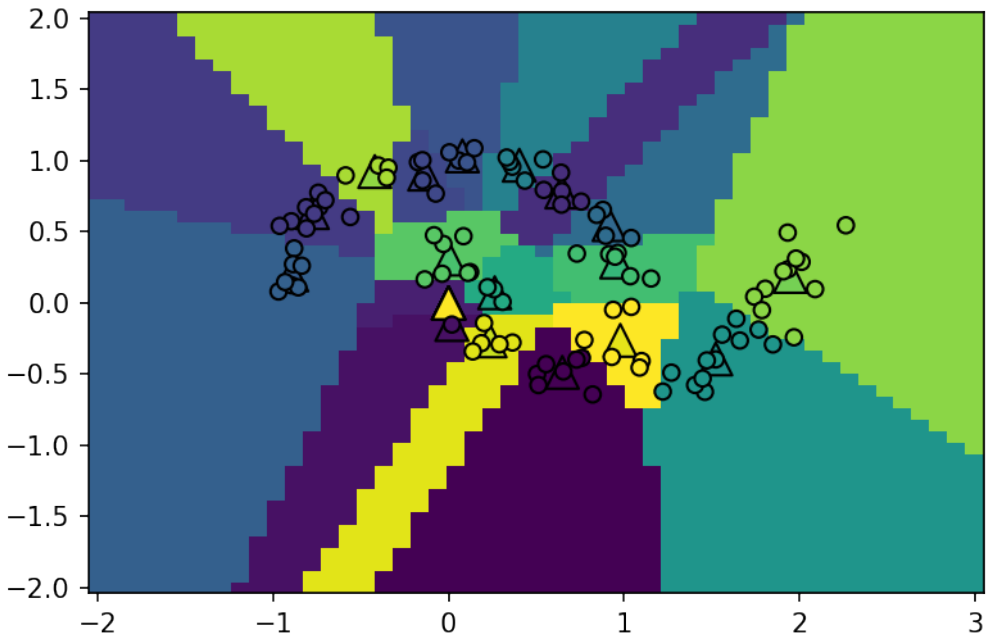
$$\alpha_i = \begin{cases} 1/|C_k| \text{if } \mathbf{x}\in C_k \\ 0, \text{else} \end{cases} \tag{6.13}$$

```
In [313]: 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 [315]: 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 [316]: 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>
```



6.3 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? Find transform $\phi(\cdot)$ 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 $\mathbf{x}$

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

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

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

6.3.1 Large Margin Nearest Neighbor

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 $\mathbf{x}$

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

6.4 Probably Approximately Correct

Can we obtain formal guaranties in Learning? Formal model of learnability

- 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 $L_{\mathcal{D},f}(h) = Pr_{x \sim \mathcal{D}}[h(x) \neq f(x)]$

6.4.1 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 $L_{\mathcal{A}}(h^\star) = 0$

6.4.2 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

6.4.3 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 $h_{\mathcal{A}}$

$$Pr_{\mathcal{A}}[L_{\mathcal{D},f}(h_{\mathcal{A}}) \leq \epsilon] \geq 1 - \delta$$

6.4.4 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

$$m_{\mathcal{H}}(\epsilon, \delta) \leq \left\lceil \frac{\log(|\mathcal{H}|/\delta)}{\epsilon} \right\rceil$$

6.4.5 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

6.4.6 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

6.5 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