Header styling inspired by CS 70: https://www.eecs70.org/

1 Introduction

1.1 Examples of Machine Learning

- The things listed here are just here to provide a sense of what machine learning can be useful for. It's not a complete list (obviously), and we will spend time throughout the semester learning about some of these things.

  i. Recognizing digits in images – this is something that people have spent a lot of time trying to fine tune historically. Nowadays we have it figured out, but it took a surprising amount of time to get this right.

  ii. Identifying a tumor in an x-ray

  iii. Classifying email as spam

  iv. Diagnosing a disease from symptoms

  v. Predicting the price of a stock 6 months into the future.

  vi. Predicting the 3D structure of all atoms in a protein from it's amino acid structure.

  There are also many types of learning problems in machine learning:

  – *Classification Problems*: where the output is a label from a finite set.

  – *Regression Problems*: the output is real-valued

  – *Ranking Problems*: the output only ranks examples relative to one another.

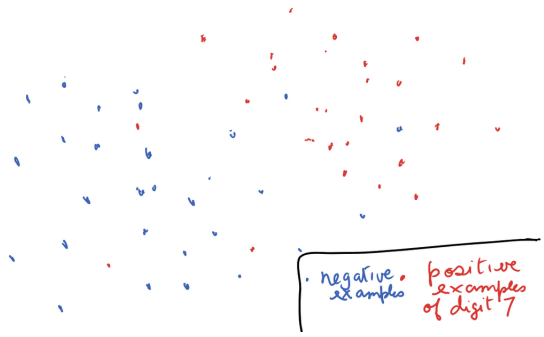
  – *Unsupervised Problems*: models like Chat-GPT, are used for generative problems.

1.2 MNIST

- This is a training set of 60,000+ examples, which was a very popular dataset to grade ML model performance. Each digit is a 28x28 pixel of a grey level image, and the goal is to have a machine identify what each number is.
- In essence, you can think of each input as a matrix with a real valued number in it.
- Nowadays, we've managed to get the error rate down to somewhere around 0.5%.

1.3 Machine Learning Approach

- Essentially, the thing we are training is what's called "feature vectors", which train a classifier, done in what's called "training time".
- At training time, the classifier then takes in new data on an untrained dataset, and classifies them (test time). This test dataset is usually a dataset in which we know the correct answer, which helps us evaluate how well the model behaves.
- Finally, we deploy this model for its intended use. Shockingly, this is the one that's easiest to forget.
- In the case of training images, we turn the 2x2 matrix of image intensities into a vector instead, and classify the vector (this is the same process as before, no data is being destroyed here).
- Suppose we're training a model, and our dataset looks like:



Let's say that the red dots are "positive" examples of the digit 7, and the blue dots represent "negative examples" (not a 7). Now, if we get a new point, how do we classify it?

- One way we can do this is via *nearest neighbors*, which basically finds the point closest to the new data point, and classifies it as such. The idea here is that the similarity between the new point is highest with its nearest neighbor, so the probability it classifies in the same way is also the highest.
- There is also the idea of a linear classifier, where we draw a line $\vec{w} \cdot \vec{x} + b = 0$, which gives us a way to classify new point based on where they fall relative to that boundary.
- Sometimes, linear classifiers aren't good enough. If our distribution is less skewed, then sometimes a single line isn't very effective. Sometimes, we can even have multiple boundaries, making the entire thing highly nonlinear.

1.3.1 Small aside on training error

- Is it good to have zero training error? Yes, but only if you have a LOT of data. Companies like Google, Microsoft. etc. work with incredibly large datasets, in which case for them zero training error is a target, but if we're working with a smaller dataset (which will be the case at most companies except the largest ones), having some training error may actually be a good thing.
- Having zero training error means that the model has basically memorized the training data, so there's absolutely no way that this model can be good elsewhere. why is this?

1.4 How to make nonlinear decision boundaries?

- Nearest neighbors gives us a nonlinear decision boundary, and it gets pretty good once we start considering multiple neighbors.
- One other way we can do this is to use Neural networks to basically make better feature vectors directly from the initial ones. At the end, it generates a linear decision boundary. This is called the "latent representation". Then, we can build a classifier on top of this decision boundary. In essence, this is what a linear classifier does.
- Suppose all the positive examples lie inside a disk. Then, there's no possible way for a linear classifier to work well on the initial data, but the hope is that we can use machine learning to generate another feature space where a linear classifier *would* work.

  For the disk example, we could take a boundary $x_1^2 + x_2^2 = c$, then the classification is linear with respect to x_1^2 and x_2^2! Therefore, we can perform a linear classification here. In our case, our features could be:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \end{bmatrix}$$

  Note that this doesn't change our data at all, it just modifies it in a way so that we can perform linear regression on it. Here, we would try to find a linear classifier in this 5-dimensional space.

1.5 Neural Networks

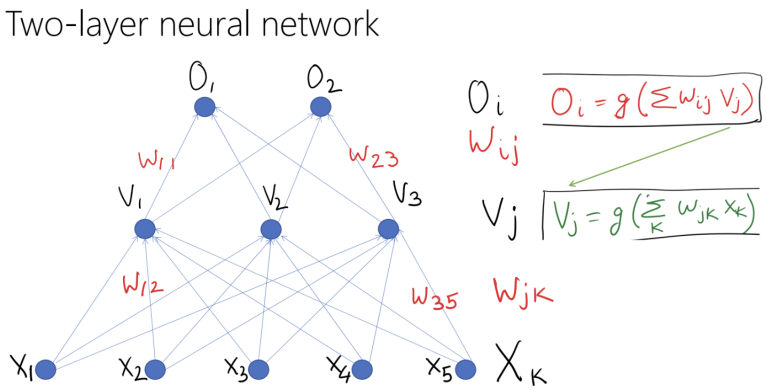
- All they are is just composing simple logistic regression functions over and over again.
- For a single layer, single output neural network, it would take each input, and have its own weight w_i associated with them. Then, we can output something along the lines of:

$$v_2 = \sum_k w_k x_k$$

  In other words, we can think of this as an inner product between our input and the weight vector.
- Then, what we will do is to introduce some nonlinearity into the equation, so that we can get something meaningful out of the neural network. If we just keep repeating this with linear functions, then we're obviously going to get something linear as output, which doesn't help us.
- In linear regression, one function we may choose to use is the sigmoid function, written as:

$$g(z) = \frac{1}{1+e^{-z}}$$

  and we output $v_2 = g(\sum_k w_k x_k)$. Now that we pass the dot product through a nonlinear function, then we now get something more meaningful.
- Sometimes, these functions $g(z)$ are called *activation functions*. We can put these in between layers as well.
- Now we can push this to multiple layers, where we have a layer in between our input and output:



  Notice how the v_i gets pushed through our activation function, and so does our o_i. Because of the way information is passed from layer to layer, this is sometimes called a *feed-forward* neural network. If we put it all together, we get:

$$o_i = g\left(\sum_j w_{ij} g\left(\sum_k w_{jk} x_k\right)\right)$$

1.6 Training a Neural Network

- At the end of the day, the goal is to find a w, such that o_i is as close as possible to the true label (or true output). If this probability is low, then we have a bad setting of the parameters.
- To do this, we can define a loss function $\mathcal{L}(w)$, and compute $\nabla_w \mathcal{L}$. Then, we update w by:

$$w_{\text{new}} = w_{\text{old}} - \eta \nabla_w \mathcal{L}$$

  The value of η is called a **hyperparameter** which we set prior to running the training procedure. This process is known as **gradient descent**. More complicated optimization procedures do exist, but they're basically never used in practical machine learning.
- Now, the question is, what kind of loss function $\mathcal{L}$ should we use?

- A good choice for a loss function is the likelihood (also known as cross-entropy):

$$\mathcal{L} = -\sum_{\text{input data}} (y_i \ln o_i + (1 - y_i) \ln(1 - o_i)$$

  It turns out, using this as our loss function actually reduces the problem directly to logistic regression!

2 Maximum Likelihood Estimation

- Recap of last time: we train a neural network by defining a loss function $\mathcal{L}(w)$, and continuously update our value of w via gradient descent.
- In the equation for $\mathcal{L}$ that we gave, y_i represents the true label of the data point, and o_i represents the output of our model.

  If o_i matches exactly the desired output y_i, then the loss function $\mathcal{L} = 0$. We only get a positive $\mathcal{L}$ if there is a mismatch between y_i and o_i.
- For a multi-layer neural network, the same thing applies. We compute the gradient with respect too all weights across *all* layers. The difference between multi-layer NNs and single layered ones is that the loss function $\mathcal{L}$ is no longer convex, so you might end up at a global optimum instead of a local optimum.
- Generally, computing this gradient is quadratic in the size of w, but the back propagation algorithm (a DP-based alg) that allows us to do this in linear time instead. This algorithm works for any number of layers.

2.1 Optimal Classifier

- What is considered the "best" classifier? Theoretically, what is the "best" classifier that could be achieved?

  Here, it's the classifier that minimizes the probability of misclassification, or in other words it maximizes the probability that you classify correctly. This is called the *Bayes Classifier*. This classifier may not have zero training error.

  However, this is not practical in practice, since we need to know the probability distribution of the dataset. It's a circular issue almost, since what we want to do anyways is to discover this distribution by looking at our data.

2.2 Error, Validation and Cross-Validation

- There are two main kinds of error that we really care about:

  Training Error: we train a classifier to minimize the training set error. We want to minimize the probability of success on this dataset.

  shouldn't we want to *maximize* the probability of success?

  Test set error: This is the error from your classifier on a dataset that you didn't use to train the model, but a dataset in which you do know the true answers for. Part of the game in machine learning is figuring out whether you're fooling yourself by making your model seem better than it actually is. This test set is also called the validation set.
- So given a dataset, how do we partition the data in such a way that we don't run into these issues? One way to do so is through k-fold cross validation. The way it works is we split our dataset up into k equal chunks, and use $k-1$ of these to train the model, while the last chunk we use to test/validate the model.

2.3 Maximum Likelihood Estimation (MLE)

- Remember that we want to maximize the likelihood that our model classifies things properly, so evidently, MLE is useful here. In terms of our loss, we would like to minimize the loss function.
- In supervised learning, we're going to work with a training dataset $D = \{(x_i, y_i)\}_{i=1}^{N}$. x_i is a d-dimensional vector (in the case of our images, it would be a 28×1 vector, and y_i is some value dependent on the problem. If we have a

classification, then usually y_i is constrained to two values, -1 and 1, but in regression then y_i can be any number, so $y_i \in \mathbb{R}$ in that case.

The only difference between classification and regression is the kind of output that we want, whether y_i is a discrete or continuous variable.

- In unsupervised learning, we lose the notion of y_i and instead work only with a dataset $D = \{x_i\}_{i=1}^N$. In essence, we no longer know the right answer, but we do have a bunch of data. In some sense it almost feels like a prediction problem, but the model is basically trained to learn what actually comes next. This is what Chat-GPT does.
- We also have a *Model class*, written as:

$$f(x \mid w, b) = w^\top x + b$$

the model class basically defines the kind of function you are looking for. This is also something that we (the creator) have to define for the model. For now, we can think of ML as picking out the best possible model function out of this family – so picking out an optimal (w, b) in our case – but eventually we'll also get into the notion of picking a *distribution* of models instead.

- Our learning objective is now to compute:

$$\underset{w,b}{\operatorname{argmin}} \sum_{i=1}^{N} L\big(y_i, f(x_i \mid w, b)\big)$$

- This only allows us to study statistical models. It means that we treat the thing we are trying to estimate as a RV, and we want to know the probability distribution of that RV. This phrasing allows us to cast many problems as statistical models, which is what makes ML so powerful.

2.3.1 Quick aside on RVs

- A random variable (RV) is a function $x \to \mathbb{R}$. Discrete RVs have a probability mass function (PMF), and continuous RVs have a probability density function (PDF) instead. In terms of normalization, the sum of all probabilities in the PMF is 1, and the analogous constraint for PDFs is that the integral over the domain is 1.
- Just some RVs as examples:
  - Bernoulli RV: $P(\text{heads}) = p$, $P(\text{tails}) = 1 - p$.
  - Binomial RV: We model n coin tosses, and count the number of heads k:

$$P(x = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

  - Poisson RV: model the number of mutations k, given that it has a mean mutation rate λ over a fixed time interval:

$$P(x = k) = \frac{e^{-\lambda} \lambda^k}{k!}$$

  - Gaussian: a continuous RV:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right)$$

- So far, we've only looked at univariate distributions. With multivariate distributions, the space of outcomes is a vector instead of a scalar. The mean becomes a vector also, and the variance now becomes a matrix of covariances. We will learn more about this next lecture.

2.4 MLE Setup

- Given a dataset $D = \{x_i\}_{i=1}^N$, where each $x_i \in \mathbb{R}^d$, and assume a set of distributions on $\mathbb{R}^d$ parametrized by θ, denoted as $p_\theta(x)$. In the case of a Gaussian, then θ could be the mean and the variance.
- The key assumption we will make is that D contains sample that belongs to one of these distributions in our family:

  $$x_i \sim p_{\hat{\theta}}(x)$$

  This assumption also requires that each x_i is distributed i.i.d. from each other.

  Out goal is to "learn" or estimate the value of θ which "pins down" the distribution from which the data came. We will define this optimal θ to be θ_{MLE}. That is, $\theta_{\text{MLE}} = \text{argmax}_{\theta \in \Theta}\, p(D \mid \theta)$.
- Note that we have $p(D \mid \theta)$ instead of the other way around. Then because D is i.i.d, then we have the nice simplification that $p(\{x_i\}_{i=1}^N \mid \theta) = \prod_{i=1}^n p(x_i \mid \theta)$. This lets us break down x_i away from a summation into a product.
- Is there a unique value of θ that satisfies MLE? This is not always guaranteed! There may be infinitely many solutions that give you the maximum likelihood, but there are techniques that allow us to deal with this.

2.5 MLE Properteis

- *Consistency*: if the data were truly generated by our hypothesis family, then as we get more data we're guaranteed that maximizing the likelihood is equivalent to finding the true parameters.
- *Statistically efficient*: Given some fixed amount of data, MLE is efficient in getting the information it needs out of it.
- The value of $p(D \mid \theta_{\text{MLE}}$ is invariant to reparametrization. Basically, this means that the value of p doesn't depend on the way we look at the problem (or how we parametrized it). Even though the returned parameters may change in value, the likelihood does not.

  Suppose you have a dataset generated from $N(\mu, \sigma)$, then if you were to draw from $N(2+\mu, \sigma)$ instead, then the model would find optimality at $\mu' = \mu - 2$, but the likelihood doesn't change.
- MLE can still yield a parameter estimate even when the data were not generated from that family. Usually it's a Gaussian, but we don't know that for sure obviously. It doesn't matter how bad your hypothesis class is, it will return you a value.

2.6 Univariate Gaussian Example

- Assume the data is generated from $X \sim N(x \mid \mu, \sigma^2)$. Now, our goal is to find $\text{argmax}_{\theta \in \Theta}\, p(D \mid \mu, \sigma^2)$.
- The first step is always to write down the likelihood function:

  $$p(D \mid \theta) = p(x_1, \ldots, x_N \mid \mu, \sigma^2) = \prod_{i=1}^N p(x_i \mid \mu, \sigma^2)$$

  the product is usually hard to work with, so we'll work with the log likelihood instead:

  $$\log p(D \mid \theta) = \sum_{i=1}^N \log p(x_i \mid \mu, \sigma^2)$$

  and because the log function is monotonic, then the value of θ_{MLE} doesn't change when we make this change to the log likelihood.
- To solve this maximization problem, we look for stationary points in the likelihood, so points where the derivative with respect to μ and σ are zero. We then also need to confirm that we have a maximum point, which we can do by computing the Hessian. In the Gaussian case, the Hessian is diagonal, so we can just look at the parameters separately.

For the mean:

$$\begin{aligned}\frac{\partial}{\partial\mu}\sum_{i=1}^{N}\log p(x_i\mid\mu,\sigma^2) &= \sum_i \frac{\partial}{\partial\mu}\log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{1}{2\sigma^2}(x_i-\mu)^2\right)\right]\\ &= \sum_i \frac{\partial}{\partial\mu}\left[-\log\left(\sqrt{2\pi\sigma^2}\right)-\frac{1}{2\sigma^2}(x_i-\mu)^2\right]\\ &= \sum_i\left[0+\frac{1}{\sigma^2}(x_i-\mu)\right]\end{aligned}$$

We now set this to zero, which gives us $\sum_i x_i = \sum_i \mu$. So this tells us that $\sum_i x_i = N\mu$, or rather $\mu = \frac{\sum_i x_i}{N}$, as expected.

- You can do the same thing with the variance, the math is a little more complicated but σ^2 does work out. Just take the derivative with respect to σ, set it to zero, and you will find:

$$\sigma^2_{\text{MLE}} = \frac{1}{N}\sum_{i=1}^{N}(x_i-\mu)^2 = \sigma$$

This is a biased estimate, since the MLE parameter σ_{MLE} we estimated depends on μ, which is something we already estimated. As a result, we need to change the denominator from a $\frac{1}{N}$ to a $\frac{1}{N-1}$ to account for this loss of a degree of freedom.

Truthfully, I don't really understand why changing N to $N-1$ corrects for this.

- One drawback of MLE is that it only yields a point estimate of our parameter and nothing else. The value of θ that maximizes our likelihood may be a very sharp spike, which is generally unwanted since it means that the model is *very* sensitive to changes in the data. In that same vein, a peak which is shorter but doesn't vary as much over a wide range of σ may be a better choice. The former case is much more likely during overfitting, or basically when you try to fit a 50th order polynomial to a function that is represented more by a 4th order.

3 Multivariate Gaussians

3.1 MLE for Multinomial Distribution

- Now suppose we want to do this for the multinomial distribution. Consider flipping a six-sided die, and we want to know the probability of each result. That is, θ is now a 6-dimensional vector where θ_i is the probability that the value i is rolled.
- Given this, we can write $P(X=k\mid\theta)=\theta_k$.
- Now, since one side is always face up, then we know that $\sum_k \theta_k = 1$, by law of total probability.
- Now, we can write the likelihood:

$$P(D\mid\theta) = P(x_1,\ldots,x_N\mid\theta) = \prod_{i=1}^{N} P(x_i\mid\theta) = \prod_{i=1}^{N}\prod_{k=1}^{6}\theta_k^{I[x_i=k]}$$

the exponent is an indicator variable that tracks whether roll i is equal to k. Otherwise, the exponent evaluates to $\theta_k^0 = 1$. This is just a weird notational thing that will be helpful later. We can then simplify this one more time:

$$P(D\mid\theta) = \prod_{k=1}^{N}\theta_k^{\sum_i^N I[x_i=k]} = \prod_{k=1}^{6}\theta_k^{n_k}$$

this last step works because exponents add when we multiply them. Now, we take the derivative of this with respect to θ_k. Now, our MLE becomes:

$$\theta_{\text{MLE}} = \underset{\theta\in\Theta}{\text{argmax}}\log p(D\mid\theta) = \underset{\theta\in\{\Theta\mid 1=\sum_k\theta_k\}}{\text{argmax}}\sum_{k=1}^{6}\log\theta_k^{n_k}$$

- Now to go ahead and calculate this optimum point, we will use the method of lagrange multipliers, because we have a constraint in the problem. We look for $J(\theta, \lambda) = \log p(D \mid \theta) + \lambda(1 - \sum_k \theta_k)$.

  The rationale for Lagrange multipliers is to take the function we are trying to optimize, and add a λ term to it, thereby making it "augmented". We now look for stationary points with respect to θ and λ. It's really just there so that we can make sure that our parameters match the constraints. It's a way for us to force $\sum_k \theta_k = 1$, since otherwise $J(\theta, \lambda)$ will increase in value. The expression for J is:

$$J(\theta, \lambda) = \log p(D \mid \theta) + \lambda\left(1 - \sum_k \theta_k\right) = \sum_{k=1}^{6} \log \theta_k^{n_k} + \lambda\left(1 - \sum_k \theta_k\right)$$

- Now we take derivatives with respect to λ and θ. Taking this with respect to λ:

$$\frac{\partial J}{\partial \lambda} = 0 \implies 1 = \sum_k \theta_k$$

  so we just get our constraint back, not very surprising.

- Now taking the derivative with respect to θ_k:

$$\frac{\partial J}{\partial \theta_k} = \frac{\partial}{\partial \theta_k} \sum_{k=1}^{6} \log \theta_k^{n_k} - \frac{\partial}{\partial \theta_k} \lambda \theta_k = \frac{n_k}{\theta_k} - \lambda = 0 \implies \theta_k = \frac{n_k}{\lambda}$$

  Plugging this into the first equation, we get:

$$1 = \sum_k \theta_k = \sum_k \frac{n_k}{\lambda} \implies \lambda = \sum_k n_k = N$$

  All together, we have $\theta_k = \frac{n_k}{N}$. It is also nice (and important) to check that θ_k is valid, since it ranges between 0 and 1.

  Review Lagrange multipliers

- So far, we've been able to use pen and paper to determine what the optimal parameters are. This is not the case in general, as many times we will have optimization problems that have no closed form solution. To solve this, we need something called *iterative optimization*.

3.2 Multivariate Gaussians

- Recall the univariate Gaussian:

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right)$$

  The Multivariate Gaussian is just an extension of this, where $x \in \mathbb{R}^d$, $\mu \in \mathbb{R}^d$, and $\Sigma \in \mathbb{R}^{d \times d}$:

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x - \mu)^\top \Sigma^{-1}(x - \mu)\right)$$

  Despite it looking more complicated, a lot of the things we do with the univariate Gaussian work the same in the multivariate case. μ used to be a scalar, but now it's a $1 \times d$ vector, and the variance σ is now this large covariance matrix Σ. Σ is a PSD (positive semidefinite) matrix, and its inverse is sometimes called the precision matrix.

- Why a lecture on MVGs? They are extremely central to modern day ML! For instance:
  - Classification problems: generative vs. discriminative (we will see this in a later lecture)
  - Unsupervised models: Principal Components Analysis & autoencoders (in a later lecture)
  - Advanced Topics: Gaussian Process Regression (beyond the scope of this class)

  Even in the case where they aren't useful, because of the central limit theorem many datasets we deal with will naturally be normal, and therefore we will need to learn how to deal with MVGs. They're also really nice to work with (analytically tractable).

- As a teaser, MVGs allow us to reduce dimensions, and it's used to derive the steps for Principal Component Analysis (PCA)

3.3 Univariate Gaussians

- To start, let's analyze the case where we have two normal distributions, say, the height and weight of humans. By CLT with genetics, we can argue that this should be distributed normally, so we have:

$$\text{height} = X_h \sim N(\mu_h, \sigma_h^2) \quad \text{weight} = X_w \sim N(\mu_w, \sigma_w^2)$$

- Now, suppose we want to write the *joint distribution*, how would we write this down? One way we can do this is to plot the distribution, and ask questions about it.
  - If we computed the mean of the distribution, what would that look like? Well, since the height and weight are different parameters, we may write it as a vector $u = [\mu_h, \mu_w]$, for the different means.
  - How would we describe the spread of the distribution? The covariance matrix Σ gives us a way to talk about the spread of the distribution when plotted.

  Ideally, we'd want to use something of the form:

$$p([x_h, x_w]) = N(x_h; \mu_h, \sigma_h^2) N(x_w; \mu_w, \sigma_w^2)$$

  At what point have we established that it should even be a product of two Gaussians?

  But because height and weight are not immediately independent, we cannot write it in this form. But, in essence we would like to find a way to transform the plot so that we *can* write it in this way.

- The trick is so "rotate" the distribution so that the correlation between the the two parameters is zero, then we can write it as a product. To compute this rotation, we want to multiply by some orthonormal matrix Q such that:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = Q \begin{bmatrix} x_h \\ x_w \end{bmatrix}$$

  keep this in mind; we will return to this later.

3.4 Understanding the Covariance Matrix

- Suppose we have two independent Gaussian RVs that are 1D, so we have $X \sim p(x) = N(\mu_1, \sigma_1^2)$ and $Y \sim p(y) = N(\mu_2, \sigma_2^2)$. Then, we can write the joint distribution:

$$p([x, y]) = \frac{1}{\sqrt{2\pi\sigma_1^2}} \exp\left[-\frac{1}{2\sigma_1^2}(x-\mu_1)^2\right] \frac{1}{\sqrt{2\pi\sigma_2^2}} \exp\left[-\frac{1}{2\sigma_2^2}(y-\mu_2)^2\right]$$

- We can then write this in matrix form:

$$P(x, y) = \frac{1}{2\pi\sqrt{\sigma_1^2\sigma_2^2}} \exp\left[-\frac{1}{2}\begin{bmatrix} x-\mu_1 & y-\mu_2 \end{bmatrix} \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}^{-1} \begin{bmatrix} x-\mu_1 \\ y-\mu_2 \end{bmatrix}\right]$$

  The intuition for having the *inverse* covariance matrix in the exponent is that in a univariate Gaussian, the exponent has a $\frac{1}{\sigma^2}$ term, so to generalize that we have a $\frac{1}{\sigma_i^2}$ term, which is achieved using the inverse covariance matrix.

  Why does this also work for non-diagonal matrices?

3.5 Review of Expectations, Variance, Covariance

- Recall that an expectation $E(X)$ of a random variable X is defined as $\sum xP(x)$ for discrete and $\int xP(x)dx$ for continuous random variables.

  There's also the principle of linearity:

$$E\left(\sum_i \alpha_i x_i\right) = \sum_i \alpha_i E(x_i)$$

this is a guaranteed properly regardless of whether x_i are independent. For products, if x_i are independent, then we have:

$$E\left(\prod_{i=1}^{n} x_i\right) = \prod_{i=1}^{n} E(x_i)$$

- The Variance of a RV with mean $\mu = E(X)$, then the variance is defined as $\text{Var}(X) = E[(X-\mu)^2]$. Expanding htis we get the following properties:
  - $\text{Var}(x) = E(X^2) - \mu^2$
  - $\text{Var}(aX+b) = a^2\,\text{Var}(X)$
  - If $x_1, \ldots, x_n$ are independent and $\alpha_1, \ldots, \alpha_n$ are constants:

$$\text{Var}\left(\sum_i \alpha_i x_i\right) = \sum \alpha_i^2 \text{Var}(x_i)$$

- The covariance of two random variables is defined as:

$$\text{Cov}(X,Y) = E\left((X - E(X))(Y - E(Y))\right) = E\left((x-\mu_x)(Y-\mu_y)\right) = E(XY) - E(X)E(Y)$$

  We also have $\text{Cov}(X,X) = \text{Var}(X)$, and if two RVs are independent, then $\text{Cov}(X,Y) = 0$.

- The correlation is a related parameter, defined as:

$$\text{Corr}(X,Y) = \frac{\text{Cov}(X,Y)}{\sqrt{\text{Var}(X)\,\text{Var}(Y)}}$$

  It's useful to think of it as a "normalized covariance", since the correlation is restricted over the domain $[-1,1]$. -1 is considered the minimum correlation, and 1 is the max correlation, and 0 is no correlation.

3.6 Return to MVG

- So now we can go back to the covariance matrix, which is a big matrix whose (i,j)-th entry is defined as:

$$\Sigma_{ij} = \text{Cov}(X_i, X_j)$$

  From this definition, it's clear that the covariance matrix is always symmetric, since $\text{Cov}(X,Y) = \text{Cov}(Y,X)$. For Gaussian RVs, it's true that if $\text{Cov}(X,Y) = 0$ iff X, Y are independent. **This is not always true, the general case is an "if" case only!**

- Now we return to the MVG that we had earlier. We worked through the "baby" case of independent RVs, and move to the general case where the covariance matrix is not always diagonal.

  At some fundamental level, the precision matrix is really a descriptor of which way the axes in the scatterplot of the data is pointing, and we will see how to quantify this by decomposing Σ^{-1} into rotation matrices and such.

- Recall the general form for MVGs:

$$p(x;\mu,\Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|}\exp\left(-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\right)$$

  the only term we will concern ourselves right now is the quadratic term in the exponent. Intuitively, the quadratic term basically maps out the "level sets" of the probability distribution, or effectively which values of x give us $p(x)$ being constant. This is because the equation $x^\top \Sigma^{-1} x = c$ basically defines a d-dimensional ellipse (the case is quite easy to see with two dimensions) , with which gives us an "orientation" of the Gaussian.

  The key takeaway is that the quadratic term defines level sets on the MVG, which are in general ellipses. In particular, this description comes from the elements of the covariance matrix.

- As an aside, there is a way to transform a general MVG into one whose points lie in a circle – this process is called "sphering" the MVGs. This has applications in PCA, advanced linear regressions, etc.

3.7 Diagonalizing a Matrix

- The spectral theorem tells us that if we have a symmetric matrix A, then we can write $A = QDQ^\top$, where Q is orthonormal and D is diagonal matrix with real eigenvalues.

  Since the covariance matrix is symmetric, we can diagonalize it too! Moreover, finding the diagonalization is exactly what we need in order to "axis align" the MVG, since a remember that a diagonal covariance matrix is equivalent to axis alignment.

- There's also another property of MVGs that is important for diagonalization called the **affine property**, which basically states that if X is a set of independent RVs, then if $Y = AX + \mu \sim N(\mu, \Sigma)$. In particular, the covariance matrix of Y is given by $AA^\top$.

  We can also go the other way around: if $Y \sim N(\mu, \Sigma)$, then $\Sigma^{-1/2}(Y - \mu) \sim N(0, I)$. This is analogous to the that you can always transform a Gaussian to one with zero mean and unit variance.

4 Linear Regression

- Today we will make use of MLE that we've been learning, and also our knowledge about Gaussians.

4.1 Regression

- It's a form of supervised learning, where we have data pairs $D = \{(x_i, y_i)\}$ where x_i may be discrete or continuous. x_i may also be scalar/vector, but y_i is always a scalar.
- Formally, we want to find $p(y \mid x)$, or essentially the conditional pdf for the data points y given the values x.
- We usually think about getting a "point" prediction of a model which is just given by $\hat{y} = E_y[p(Y \mid X = x)]$. We will explore this for a while, then back out of this and look at the other forms later.
- When you usually think about regression you probably think of a regression line – one way to think about the regression line is that it also gives us a probability for every given value of x, and the best fit line is the one that maximizes this probability.
- What are some strategies we could use to estimate $p(y \mid x)$?

  1. **Generative:** Estimate $p(x, y \mid \theta)$, then we can use the fitted model to compute $p(y \mid x, \hat{\theta})$.

     Basically, we can use what we built earlier with MLEs to calculate $p(x, y \mid \theta)$, then we can use that to compute $p(y \mid x)$.

     In particular, we can compute:

     $$p(y \mid x, \theta) = \frac{p(y, x \mid \hat{\theta})}{p(x, \hat{\theta})} = \frac{p(y, x \mid \hat{\theta})}{\int_y (y, x \mid \hat{\theta})\, dy}$$

     For MLEs, don't we have $p(y, x \mid \theta)$ (or rather, we find the θ that optimizes the data given the parameter, which is not what we're doing here when we write $p(x, y \mid \theta)$?

  2. **Discriminative:** We can assume the inputs to be fixed, and only the output is a random variable. That is, we try to directly model $p(y \mid x, \hat{\theta})$.

     The advantage of this approach versus the previous one is that we don't treat x as an RV. In some cases, if we *absolutely* don't understand the distribution of x, it may be a bad idea make assumptions about it (e.g. treat it as a Gaussian RV). This is a deep question and we won't go into this further.

4.2 Linear Regression

- This takes the discriminative approach to generate the prediction. IN particular, the predictions will be a linear function of the parameters – this is what the "linear" in linear regression refers to. For instance:

  $$\hat{y} = E_y[p(y \mid x)] = w^\top x + w_0$$

w is the parameters, x is the input features, and w_0 is the "bias". This is considered linear, since y is linear in terms of x.

- One thing we will do is that instead of a bias, we will make an extra features that always equals 1, and instead use $x' = [x, 1]$. This allows us to basically keep track of the bias without having to write it out explicitly – we can also write $\hat{y} = w^\top x'$ instead.
- Linear models are also very useful! Instead of just computing $\hat{y} = w^\top x$, we can also *augment* our feature space, and also track, say, a quadratic term. So, we can also have $\hat{y} = w^\top [x, x^2]$ for instance, and in fact we can augment by any polynomial. This allows our linear regression to "fit" to any polynomial.

  This process of taking the original input space and "expanding" this out in terms of predetermined functions is called a *basis expansion*. Suppose you had $x \in \mathbb{R}^2 = [x_1, x_2]$, then we can basis expand quadratically:

  $$[1, x_1, x_2, x_1 x_2, x_1^2, x_2^2] \in \mathbb{R}^6$$

  this can all come from x! For $d = 1$, then the polynomial expansions is the set $[1, x, x^2, \ldots, x^n]$.
- These are predetermined functions that we always know ahead of time and plug x into, so this gives us a lot of freedom in what we want to fit to. As a notational thing, we will write $\hat{y} = w^\top \Phi(x)$ instead, where $\Phi(x)$ is the basis expansion.

  Note that $\Phi(x)$ can be *literally anything*! We can pick anything from just the standard polynomial basis to even the Fourier basis for our regression. It's all up to us.
- As a sneak peek, we can actually *learn* the basis functions, but we won't go into this today.
- For the rest of this lecture, we will just assume $\hat{y} = w^\top x$ for now.

4.3 Specific Linear Regression

- So far, we know that $\hat{y} = w^\top x$. What should we use for $p(y \mid x)$?
- For linear regression, the assumption we will use is that for each x, there is a Gaussian distribution for y, that shares the same variance. So, $p(y \mid x) = N(y \mid w^\top x, \sigma^2)$. This is equivalent to writing $Y = w^\top x + \epsilon$, where $\epsilon \sim N(0, \sigma^2)$ is a Gaussian.

  I don't like the notation here, instead I think it may be better to write $p(y \mid x) = N(w^\top x, \sigma^2)$, the given symbol makes no sense in a distribution definition.
- We can then rearrange this to $Y - w^\top x = \epsilon \sim N(0, \sigma^2)$. ϵ is also sometimes called the *residuals*, which we assume to be normally distributed with equal variance.

  As an aside, there are other distributions called heavy-tail distributions, which favor values along the tails much more than the Gaussian. This is just to show that the residuals can realistically be any distribution, we will stick to Gaussians in this lecture.
- Now we fit the parameters w. We will use MLE. Here, $\theta_{\text{MLE}} = (w_{\text{MLE}}, \sigma^2_{\text{MLE}})$. This is very similar to what we did two weeks ago. We will take the log-likelihood, so this is equal to:

  $$\theta_{\text{MLE}} = \underset{w,\sigma^2}{\text{argmax}} \sum_{i=1}^{n} \log p(y_i \mid x_i, \theta) = \underset{w,\sigma^2}{\text{argmax}} \sum_{i=1}^{n} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - w^\top x_i)^2$$

  This last line comes from the assumption that each y_i can be modeled by a normal distribution with mean $w^\top x_i$ and variance σ^2.
- Note that here the estimation of w doesn't depend on σ, so maximizing w is the same as finding:

  $$\underset{w}{\text{argmin}} \sum_{i=1}^{n} (y_i - w^\top x_i)^2 = \underset{w}{\text{argmin}} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

  this is the same least-squares function that you've been taught in previous classes.

- Now to actually find w. If we vectorize this equation, we can write this as $y - Aw$, where A is called the "design matrix" and represents the vector of inputs x. With this in mind, we can then rewrite the loss as $\text{argmin}_w \|y - Aw\|_2^2$.

  The loss can then be written as:

  $$\mathcal{L} = \|y - Aw\|_2^2 = y^\top y - 2w^\top A^\top y + w^\top A^\top Aw$$

  and we want to set $\frac{\partial L}{\partial w} = 0$. Computing the derivative (using vector calculus, review this), we have:

  $$\nabla_w \mathcal{L} = -A^\top y + A^\top Aw \implies w = (A^\top A)^{-1} A^\top y$$

  Then, we just need to check that this is a critical point, which we can do by looking at the Hessian. The hessian matrix is $\nabla_w^2 \mathcal{L} = 2A^\top A$, and we require that this is positive definite (PD) in order for it to be minimized. This is achieved when A is full rank.

- The σ^2_{MLE} for this is really just the exact same thing, and we will end up getting

  $$\sigma^2 = \frac{1}{n} \sum_i (y_i - w^\top x)^2$$

  exactly as we expect.

- There is also a geometric viewpoint of this, but this is much more rigid, and overall seems to be a "weaker" interpretation of least squares than the statistical approach.

5 Linear Regression II

- Recall from last time we talked about Gaussian Linear Regression, where we wanted to estimate $\hat{y} = -\mathbb{E}_Y[p(y \mid x)] = w^\top x$. We assumed that the residuals were Gaussian distributed, and from this math we were able to recover the standard equations of linear regression.

  We wrote out $\mathcal{L}_w = (Y - Aw)^\top (y - Aw)$, and then set $\frac{\partial \mathcal{L}}{\partial w} = 0$ to solve for w^*. From this, we obtained $w_{\text{MLE}} = (A^\top A)^{-1} A^\top y$, and $\sigma^2_{\text{MLE}} = \frac{1}{n} \sum_i (y_i - w^\top x)^2$

- However, one major problem in this approach is that $A^\top A$ may not be invertible, which is the case when the features are not linearly independent. For instance, if you have more features than data points, then this will be the case. When it's not invertible, then there are infinitely many good equations for w_{MLE}, and this is called an *underdetermined system*.

- There are two ways for us to fix this:

  1. Remove features (via some type of heuristic) in the problem until the problem is linearly independent. This is called "feature selection". This is an entire area of ML that people are actively researching.
  2. Add constraints to the system to "tighten the system". This is called "regularization". In some sense, we add constraints to limit the number of degrees of freedom from the system.

     Here, we keep the same number of parameters, but add constraints on the system so that we remove some of the freedom offered by these features.

5.1 Intuition for infinite solutions

- To illustrate this, suppose you have two features that are linearly dependent, so $\alpha x_1 = x_2$.

- Suppose now that you found one $\hat{w}$. Then, for any training data point, we have $\hat{y} = x^\top \hat{w}$ to be our point prediction. We will now show that there are infinitely many $\hat{w}$ that give us the same prediction. If we expand this out:

  $$\hat{y} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} \hat{w}_1 \\ \hat{w}_2 \end{bmatrix} = \hat{w}_1 x_1 + \hat{w}_2 (\alpha x_1) = (\hat{w}_1 + \hat{w}_2 \alpha + \beta - \beta) x_1 = (\hat{w}_1 + \beta) x_1 + (\hat{w}_2 \alpha - \beta) \frac{x_2}{\alpha} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \underbrace{\begin{bmatrix} \hat{w}_1 + \beta \\ \frac{\hat{w}_2 \alpha - \beta}{\alpha} \end{bmatrix}}_{\hat{w}'}$$

  and this gives us a different value of $\hat{w} = \hat{w}'$ that also gives the same prediction $\hat{y}$. As a result, there are an infinite number of "optimal" $\hat{w}$ that gives the same prediction.

- The one that the Moore-Penrose chooses is the one that has the smallest norm. Why might we want to choose this one?

  The reason we may want to choose this is because usually, solutions with small norms affect our model in a more "uniform" way, where it is not greatly influenced by small perturbations to our input (imagine adding a little amount δ to x, and checking how our model reacts to it). Keeping the norm small is one way we can ensure that this principle is maintained.

- What about the case where our design matrix is full rank? Is there any reason we may want to put a constraint on $\|w\|$?

5.2 L2 Regularized Linear Regression

- We will learn two systems, today, L2 and L1 regularization. These just refer to the different norms that we will use to choose our optimal $\hat{w}$.

- In L2 regression, our loss function is:

$$\mathcal{L} = (y - Aw)^\top (y - Aw) + \lambda \|w\|_2^2$$

  This is also sometimes called "ridge" regression, they mean the same thing.

- In te Bayesian model approach, what we will do is establish a *prior distribution* on the parameters, $p(\theta)$. Then, we want to compute the posterior distribution, $p(\theta \mid D)$.

  The $p(\theta)$ is something that we can establish ahead of time (without looking at the data), and then with this we can calculate $p(\theta \mid D)$.

- Then, the predictive distribution is given by:

$$p(y \mid x) = \int_\theta p(y \mid x, \theta) p(\theta \mid D)\, d\theta$$

  This is generally done using Bayes rule:

$$p(\theta \mid D) = \frac{p(D \mid \theta) p(\theta)}{p(D)}$$

  This is exceedingly dfficult to do in practice! This is not really something that we will touch on further due to its difficulty.

- We will be "lazy", so instead of using normalized Bayesian probability (as described here), we will just compute $\text{argmax}_\theta\, p(\theta \mid D)$, and that will be our optimal θ. This is called maximum *a posteriori* (MAP) estimation.

  Conceptually, the way we solve this is the exact same as MLE, except we now multiply by a $p(\theta)$ term. This selects favorably for some probability models over others; you can equivalently think of MLE as the case where there is no preference for any particular model, since $p(\theta) = 1$.

5.3 MAP Estimation

- Again, now we are calculating $\theta_{\text{MAP}} = \text{argmax}_\theta\, p(\theta \mid D)$. By Bayes' rule, we have:

$$\operatorname*{argmax}_\theta \frac{p(D \mid \theta) p(\theta)}{p(D)} = \operatorname*{argmax}_\theta p(D \mid \theta) p(\theta)$$

  recall that $p(\theta)$ is our "prior" density for θ, and $p(D \mid \theta)$ is the posterior distribution.

- The prior $p(w)$ that corresponds to L2 regression is $p(w) = N(w; 0, \lambda I)$. So, using this, let's try to find w_{MAP}. First, we take the logarithm, so we have:

$$\begin{aligned}
w_{\text{MAP}} &= \operatorname*{argmax}_w \log p(D \mid w) p(w) \\
&= \operatorname*{argmax}_w \log p(D \mid w) + \log N(w; 0, \lambda I) \\
&= \operatorname*{argmax}_w \sum_{i=1}^{N} \log N(y_i \mid w^\top x_i, \sigma^2) + \log N(w \mid 0, \lambda I) \\
&= \operatorname*{argmin}_w \frac{1}{2\sigma^2} (y - Aw)^\top (y - Aw) - \sum_{i=1}^{d} \log \left[\frac{1}{\sqrt{2\pi\lambda}} \exp\left(-\frac{w_i^2}{2\lambda} \right) \right]
\end{aligned}$$

Note that we haven't taken the logarithm of the second expressino yet so that's why the signs don't match yet. Then, after more simplification we get:

$$w_{\text{MAP}} = \underset{w}{\text{argmin}} (y - Aw)^\top (y - Aw) + \lambda' \| w \|_2^2$$

for $\lambda' = \frac{\sigma^2}{\lambda}$.

- You will almost always benefit from doing this as opposed to ordinary linear regression – this approach helps a lot with numerical stability as well as guaranteeing uniqueness.
- The MAP solution just involves taking the partial derivative and set it to zero, which we've already done on the homework. Here, we have:

$$\nabla_w \mathcal{L}_{\text{MAP}} = -2A^\top y + 2A^\top Aw + 2\lambda I w \implies w^* = (A^\top A + \lambda I)^{-1} A^\top y$$

  if $\lambda > 0$, then we can always guarantee that the matrix $A^\top A + \lambda I$ is invertible.
- The effect of λ is also important here. As you increase λ, the norm of w^* gets increasingly smaller. Practically, how should we set λ?

  yeah how do you do this? the slides don't really go into detail about this.

5.4 Lasso regression

- This is using the L1 norm, which means that:

$$w_{L_1} = \underset{w}{\text{argmin}} (y - Aw)^\top (y - Aw) + \lambda \| w \|_1$$

  This penalty tends to choose sparse w over other ones, because the penalty favors points on the axes compared to other points. L1 regression is more likely to hit a contour on one of the axes, and thus it has a higher chance of setting some parameters to exactly zero.

  By contrast, the L2 penalty looks like a ball, so there is no such bias.
- There is also a way for us to derive L1 regression, by using a Laplacian prior:

$$p(w) = \exp(-\lambda' \| w \|_1)$$

- There are some advantages to Lasso over ridge. Because of Lasso's tendency to sparsity, sometimes it is best to just turn one of the parameters off, which is sometimes favorable. On the other hand, when you have two parameters that are highly correlated, then Lasso is bad because in choosing an optimum it might just turn one completely off.
- People have also combined these two together, into a regression called "elastic net regression". This is a middleground between the two regression techniques, but we won't go into the details here.

6 Classification: Generative and Discriminative

- Classification is a fundamentally different problem than a regression problem. IN a regression problem, we usually deal with a dataset $y_i \in \mathbb{R}$, but in regression, the output is not a real number but rather a label:

$$\mathcal{X} \to \mathcal{Y} \in \{0, \ldots, K-1\}$$

- The labels can be binary, or also perhaps more complicated. The MNIST dataset is not binary, since we have images and we want to classify them into the digits 0-9, for example.
- In general, the problem is phrased like this: given a dataset $\{(x_i, y_i)\}_{i=1}^n$, we are interested in *learning* the classification

$$f_\theta : \mathcal{X} \to \{0, \ldots K-1\}$$

6.1 Bayes' Rule

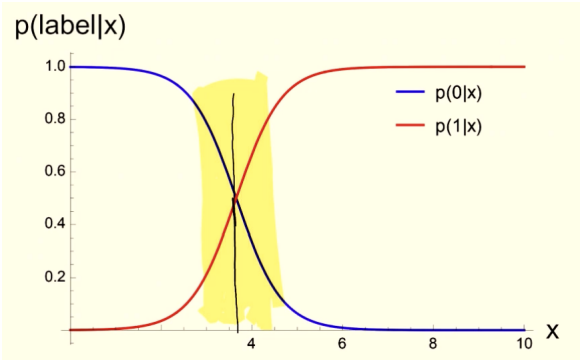
- In the context of classification, it's used to transform class-conditional probabilities into posterior probabilities. In particular,

$$p(0 \mid x) = \frac{p(x \mid 0)p(0)}{p(x)}$$
$$p(1 \mid x) = \frac{p(x \mid 1)p(1)}{p(x)}$$

  The variables $p(x \mid 0), p(x \mid 1)$ are called the *class-conditional probabitlities*. The value of $p(x)$ is determined using the law of total probability:

$$p(x) = p(x \mid 0)p(0) + p(x \mid 1)p(1)$$

- The hard part is finding the right class-conditional probability – we always have to ask ourselves, what kind of distribution do we expect? The answer to this is usually motivated by our prior knowledge about the population; maybe half of the population we expect to tend one way, so our distribution should reflect that.
- If we map the posterior probabilities, we'll get a graph like this:



  But what should we do in the region highlighted in yellow? This is a gray-area, where we aren't 100% certain that we should classify one way or the other.

  – One thing we can do here is look at the intersection between these two curves, call it x^*, and basically classify:

$$f_\theta(x) = \begin{cases} 0 & x < x^* \\ 1 & x \geq x^* \end{cases}$$

  – This is a somewhat reasonable rule to follow, since this allows us to potentially reduce the probability of classifying incorrectly at least from a probabilistic standpoint.

- In general, we want to pick a value of k (the classification) such that $p(k \mid x)$ is maximal. Equivalently, this means that we want to reduce the probability of error:

$$P(\text{error}) = \int p(\text{error} \mid x)p(x)\, dx$$

  In doing this, you are treating false positives and false negatives with eual weight. However, this is not always the case. For instance, if you're in the medical field, false negatives can be much more detrimental than false positives, often fatal, so in cases like that we should be more careful about how we classify things.

  What if you just classify the error as the probability of *misclassificaiton*? It seems that we're phrasing "error" in terms of testing positive, which isn't the best method of classification?

  It seems like they haven't covered exactly how to quantify error just yet.

6.2 Ways of Building Classifiers

- There are three main ways we can build classifiers:

  1. **Generative:** to model the conditional probabilities $p(x \mid k)$, then use Bayes' rule to get $p(k \mid x)$. This requires you to somehow model the prior $p(k)$.
  2. **Discriminative:** Model $p(k \mid x)$ directly.
  3. **Find Decision Boundaries:** Directly solve for the boundaries $f : \mathcal{X} \to \{0, \ldots, K-1\}$.

     This approach has the drawback that you lose information about how close your data point is to any of the boundaries, which means that you lose the ability to quantify your model's uncertainty in the system.

6.3 Logistic Regression

- Logistic regression is a classification method, where we model the posterior probability by a logistic function:

$$p(k = 0 \mid x) = \frac{1}{1 + e^{-(\theta_1 x + \theta_0)}}$$

  We can also extend this to K classes, but for now we will talk about two classes 0 and 1.

- If the class-conditional density $p(x \mid \{0, 1\})$ is Gaussian, then we claim that the posterior probability will be a logistic function in $z = \theta_1 x + \theta_0$.

  What is θ_1, θ_0? Are these fitted parameters, or what?

  No, θ_1 and θ_0 are parameters that you can mathematically solve for. You can solve for these variables by literally letting $P(x \mid k) \sim N(\mu_k, \sigma)$ (assume the variances are the same), and work out the math via Bayes' rule.

- In our case then, z is a linear function of x, plus a bias term θ_0. This is called an "affine" function.

- We can then generalize this to multivariate Gaussians, where we assume:

$$P(X \mid k) \sim N(\mu_k, \Sigma)$$

  here Σ is the covariance matrix.

  Review the proof in the slides, ask about what's happening.

- Intuitively, we want a sigmoid (logistic) function to model our probabilities because it's very nice – the probability is always between 0 and 1, so it's always well-behaved, and it's also smooth so there's nice properties there as well.

6.4 Modeling the Posterior Probability Distribution

- To give a preview, here's what we're going to do:

  We basically set up a label $Y \in \{0, 1\}$ to be a Bernoulli random variable (for simplicity I think), with its probability parameter so we have:

$$P(Y = 1 \mid x) = \frac{1}{1 + \exp(-\theta^\top x - \theta_0)} := \mu(x)$$

- In the binary case, we can take y to denote the values taken on by the random variables, so:

$$p(y \mid x) = \mu(x)^y (1 - \mu(x))^{1-y}$$

  (this is just law of total probability applied to Bernoulli RVs).

- In the next lecture, we will learn how to estimate θ_1, θ_0 using MLE methods.

7 Logistic Regression & Neural Networks

- Last time, we saw how the logistic equation came out of assuming Gaussian class-conditional variables. This lecture, we will continue that exploration, and see how we can use MLE to get the logistic function.
- In summary, we found that the posterior takes the form:

$$p(k \mid x) = \frac{1}{1+e^{-z}}$$

where z is a **linear** function $z = \theta^\top x + \theta_0$. Here, θ is a parameter that depends on $\{\mu_k\}_{k=1}^K$ and Σ.

This method of determining $p(k \mid x)$ is called the **generative** approach. Now, we will talk about the discriminative approach.

My understanding is that there is a generative and discriminative approach to classification problems and regression problems, is that true?

Probably? This approach probably works for anything where you have data and you want to find parameters that describe the data well.

7.1 Multi-Class Classification

- First, we will start by looking at how our two-class classification can be generalized to K classes. We will start with the generative approach, because that's what we did last time.
- We first start by writing out the class-conditional probability. This is assumed to be Gaussian (like before), though note that this isn't the only function we could pick.

$$p(x \mid k) \sim N(\mu_k, \Sigma)$$

Here, $\mu_k \in \mathbb{R}^d$ and $\Sigma \in \mathbb{R}^{d\times d}$. To find the posterior distribution, we will invoke Bayes' rule:

$$p(k \mid x) = \frac{p(x \mid k)p(x)}{p(k)} = \frac{p(x \mid k)p(k)}{\sum_{i=1}^k p(x \mid j)p(j)}$$

Here, we will now rewrite this in a "nicer" way, by letting $a_k = \log p(x \mid k)p(k)$, so we may write:

$$p(k \mid x) = \frac{e^{a_k}}{\sum_{j=1}^K e^{a_j}}$$

More explicitly, we have:

$$a_k = (\Sigma^{-1}\mu_k)^\top x - \frac{1}{2}\mu_k^\top \Sigma^{-1}\mu_k + \log p(k)$$

which you can get just by multiplying it out and collecting the terms appropriately. Here, we will denote $\theta_k = \Sigma^{-1}\mu_k$, and $\theta_{0,k}$ is just the remainder of the expression. This then allows us to write $a_k = \theta_k^\top x + \theta_{0,k}$, which is linear as desired.

As an aside, this mapping:

$$(a_1, \ldots, a_k) \mapsto \left(\frac{e^{a_1}}{\sum e^{a_j}}, \ldots, \frac{e^{a_k}}{\sum e^{a_j}}\right)$$

is called the *softmax function*. In particular, if we have $a_k \gg a_j$, then the softmax mapping will basically map this n-tuple into a basis vector.

7.2 Discriminative Learning

- We're given this hypothesis that $p_\theta(0 \mid x)$ has the functional form:

$$p_\theta(0 \mid x) = \frac{1}{1+e^{-(\theta^\top x+\theta_0)}}$$

As before, we will use the log-likelihood, and given that we assume i.i.d conditions we have a product which turns into a sum in the logarithm.

- Given this form, the likelihood is then given by:

$$\log\left(p_z^y(1-p_z)^{1-y}\right)$$

work out what z is in terms of θ, x and the other stuff to make it match the form above. Here, z depends on θ and p only depends on z. In order to find the optimal θ, then this is a quantity we'd like to maximize. At the same time, the loss is something we'd like to minimize, so we can define the loss to be the negative of this.

$$\mathcal{L}(\theta) = -y\log p_z - (1-y)\log\left(1-p_z\right)$$

- To find the maximum then, we can take $\nabla_\theta \mathcal{L}(\theta)$ and we get:

$$\nabla_\theta \mathcal{L}(\theta) = \left(-y\frac{p_z(1-p_z)}{p_z} + (1-y)\frac{p_z(1-p_z)}{1-p_z}\right)\nabla_\theta z = (p_z - y)\nabla_\theta z \tag{1}$$

To get this form, we note that since p_z is a sigmoid, then we have:

$$\partial_z p_z = \frac{e^{-z}}{(1+e^{-z})^2} = p_z(1-p_z)$$

The $\nabla_\theta z$ term is very deep – in linear this term is just equal to x, but in more complex systems (for instance, in deep learning), it need not be linear.

The rest of this lecture was a *very* light intro to neural nets by talking about neurons and human cells so I didn't feel like writing that stuff down. All you really need to know is that neural networks are just simplified models of biological neural nets.

8 Gradient Descent and Backpropagation

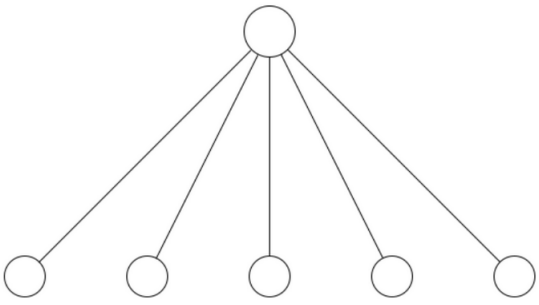
- Last lecture, we visited the logistic regression, talked about the softmax function, and also looked at the log-likelihood loss for gradient descent. We were also able to interpret the gradient $\nabla_\theta \mathcal{L}(\theta)$ geometrically in terms of how we update parameters.
- The gradient is written as:

$$\nabla_\theta \mathcal{L}(\theta) = (p_z - y)\nabla_\theta z$$

we write $p_z - y$ as the "error" term – so if we were to make an incorrect classification, then we would have a nonzero loss.

8.1 Single-Layer Neural Networks

- As a single layer, we can represent the neural network as follows:



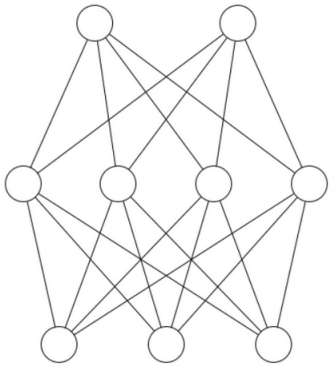
The bottom layer is the input $x^{(0)} \in \mathbb{R}^d$, and the top layer is the output $x^{(1)}$. Along each "edge" between $x_i^{(0)}$ and $x^{(1)}$, there are weights θ_{ji} which we multiply by their respective $x_i^{(0)}$ to get their contribution in $x^{(1)}$. So, to calcualte $x^{(1)}$, we have:

$$x^{(1)} = g\left(\sum_i \theta_{ji}^{(1)} x_i^{(0)}\right)$$

$g(z)$ is called the *activation function*. There are many different values of $g(z)$ that you can choose: you can use a sigmoid, a linear function, or like $\max(z,0)$. This last one is called ReLU. Another fun one is $g(z) = \frac{z}{1+e^{-z}}$, and is called SiLU.

8.2 Two-Layer Neural Networks

- As their name describes, this is the case where you have two layers:



with the following functions:

$$x_k^{(2)} = g\left(\sum_j \theta_{kj}^{(2)} x_j^{(1)}\right)$$
$$x_j^{(1)} = g\left(\sum \theta_{ji}^{(0)} x_i^{(0)}\right)$$

basically, we dot each $x_j^{(m)}$ with θ_{kj} to get $x_k^{(m+1)}$, then apply the activation function to get the result of the next layer.

- In general, each layer has an arbitrary number of layers, so the first layer has dimension d_0 and the second layer d_1, etc.. You also can change g at every layer, but because this is usually hard to track people just stick to a single value of g.
- We also add a unit node that is set to 1 on each layer, which is connected to every node in the next layer. This is called the *bias*.

  why do we introduce the bias?

  We introduce the bias for the same reason we do it in linear regression – we don't want to necessarily restrict the neural network at each layer to *have to* pass through $(0,0)$, so the bias corrects for that.
- We can then generalize this and make our neural net as complicated as we want, with the only restriction being that we don't allow any loops.
- As an aside, if you have a function $f : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$, this architecture that we've built is actually enough to approximate f to arbitrary accuracy.
- The last layer is also something that is given to us by the problem. If we're doing regression which is given by a real value, then the last layer is a single node. If we're doing K classification, then the last layer will have dimension K.

8.3 Gradient Descent

- Before we go any further, we will prove a theorem first:

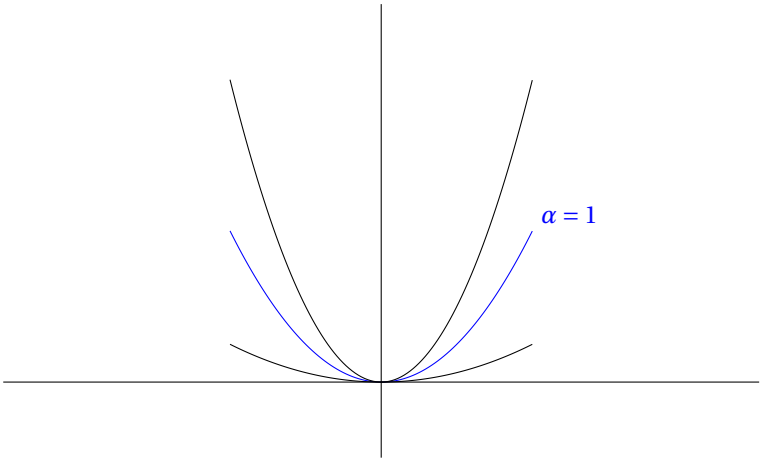
  Theorem 1. *The gradient $\nabla f(\theta)$ of a function $f : \Theta \to \mathbb{R}$ at any point θ is perpendicular to the level set of f at that point.*

Proof. We define a path t on the level set in question. Because it is a level set, then we know that the level set has the property that f is constant over that set. Thus, for any path $\theta(t)$ we define over the level set, we will have $\partial_t f(\theta(t)) = 0$. Expanding this:

$$0 = \partial_t f(\theta(t)) = \frac{\partial f}{\partial \theta_1}\frac{\partial \theta_1}{\partial t} + \ldots \frac{\partial f}{\partial \theta_m}\frac{\partial \theta_m}{\partial t} = (\nabla f) \cdot \vec{v}$$

so, we can only have that $\vec{v}$ (or the tangent to the path over the level set) is perpendicular to ∇f, as desired. □

- Now, consider $f(\theta) = \frac{\theta^2}{2\alpha}$. This is a parabola, where α controls the "flatness" of the parabola:



We ask the question, what does gradient descent look like here?[1] Based on the rule of gradient descent, we have:

$$\theta_{t+1} = \theta_t - \epsilon \nabla f(\theta) = \theta_t - \epsilon \frac{\theta_t}{\alpha} = \left(1 - \frac{\epsilon}{\alpha}\right)\theta_t$$

Now, suppose you started with some θ_1, and we ran this t times. The closed form of θ_t is:

$$\theta_t = \left(1 - \frac{\epsilon}{\alpha}\right)^t \theta_1$$

If $\epsilon < \alpha$, then θ_t decreases and approaches zero. If $\epsilon > \alpha$, then θ_t increases, and eventually diverges. If $\epsilon = \alpha$, then after a single step, we've already reached the minimum value so $\theta_2 = \theta_{min}$.

- So, hopefully this analysis gives some intuition on how the size of ϵ that we choose is dependent on α. If α is small, then we likewise need to choose a small ϵ in order to prevent our function from diverging.

  Now imagine this in higher dimensions, where the curvature of your level set varies depending on where you are on the level set. Then, the ϵ you choose is constrained by the smallest value of α that exists in the shape.

8.4 Stochastic Gradient Descent

- Now we will talk about stochastic gradient descent. Here, we will consider a loss function, which is expressed as the sum of individual losses:

$$\mathcal{L}(\theta) = \sum_{i=1}^{n} \mathcal{L}_i(\theta)$$

where $\mathcal{L}_i(\theta)$ is the loss for any particular data point:

$$\mathcal{L}_i(\theta) = \mathcal{L}(x_i, y_i, \theta)$$

The reason it takes this form is because of the i.i.d. assumption, so that when we compute the loss it just ends up being a sum over the loss of an individual point. Then, it follows that:

$$L_i(\theta) = -\log p_\theta(x_i, y_i)$$

[1] As a reminder, gradient descent is just a way to "walk along the curve".

- So far, we've seen two methods to solve this:
  - Linear regression:
    $$-\log p_\theta(x_i, y_i) = \frac{1}{2\sigma^2}(y_i - \theta^\top x_i^{(0)} + \text{const.}$$
    why do you have a $\frac{1}{2\sigma^2}$ term here?

    The exponent in the Gaussian has a $\frac{1}{2\sigma^2}$ term.
  - Logistic Regression:
    $$-\log p_\theta(x_i, y_i) = -y_i \log g(\theta^\top x_i^{(0)}) - (1 - y_i)\log\left(1 - g(\theta^\top x_i^{(0)})\right) + \text{const.}$$
- Stochastic gradient descent is a process in which you use one of these regression losses, and perform the following update scheme:
  $$\theta_{t+1} = \theta_t - \epsilon_t \nabla_\theta \mathcal{L}(x_i, y_i, \theta)$$
  Here, ϵ_t is chosen at random, typically without replacement. In essence, we are picking at random how large our ϵ is.
- This approach has worked in practice – we usually see that in high dimensions the loss is "dominated" by saddle points (so false minima), and the variance introduced by SGD seems to avoid these points.

  That said, it has also been shown that the noise hurts us when we get close to the optimal point, slowing the algorithm down. There is a compromise that you can make, which is to divide the dataset into b batches of size n/b, where for each batch you compute the loss and perform the corresponding update.

9 Backpropagation and Gradient Descent II

9.1 Summary of Previous Lecture

- Neural Networks. In particular, we said that an L-layer (feed forward) neural network is described as:
  $$a_j^{(\ell)} = \sum_{i \in [d_{l-1}]} \theta_{ji}^{(\ell)} x_i^{(\ell-1)}$$
  $$x_j^\ell = g_\ell(a_j^{(\ell)})$$
  $$g_\ell : \mathbb{R} \to \mathbb{R}$$
  a is usually called the "preactivation". g is a function that we have a lot of freedom over (linear, nonlinear, etc.). By convention however, we say that $g_L(z) = z$ is a linear function, and $g_\ell = g$ is typically held fixed.
- The loss $\mathcal{L}(\theta)$ is a function which we typically will formulate as the negative log likelihood.
  $$\mathcal{L} = -\log p_\theta$$
  Keep in mind that the loss doesn't have to be defined in terms of the log-likelihood, but we will stick to doing this for now.
- Gradient-based optimization: we minimize the loss $\mathcal{L}(\theta)$ using gradient descent:
  $$\theta_{t+1} = \theta_t - \epsilon_t \nabla_{\theta_t} \mathcal{L}(\theta_t)$$
  For neural networks, what is this loss? For 1-layer neural networks, this has the form:
  $$\partial_{\theta_{ki}^{(1)}} \mathcal{L} = \partial_{a_k^{(L)}} \mathcal{L} \partial_{\theta_{ki}^{(L)}} a_k^{(L)}$$
  When you talk about loss in this context, are you talking about the loss at a particular layer, or the loss of the network as a whole?

  The loss of the network as a whole is not well-defined, so this is probably the loss at layer.

9.2 Loss for K-classification

- If we have more than two classes, how do we compute the loss? We will show that

$$\mathcal{L} = -\sum_{k\in[K]} y_k \log \frac{\exp(a_k^{(L)})}{Z}$$

where Z is defined as the normalization:

$$Z = \sum_{k\in[K]} \exp(a_k^{(L)})$$

This loss $\mathcal{L}$ should be read as log over the product of p_k, and we "normalize" p_k using the softmax function. So in its unsimplified form, the following is what we're dealing with:

$$\mathcal{L} = -\log \prod_{k=1}^{K} p_k^{y_k}$$

- Combining this with what we have for linear regression, you can actually derive the answer. In the linear case, we have:

$$\partial a_1^{(L)} \mathcal{L} = a_1^{(L)} - y$$

so for k output neurons, we can just generalize this to k:

$$\partial_{a_k^{(L)}} \mathcal{L} = p_k - y_k$$

He keeps mentioning one-hot encoding, what does this mean again?

- To convince ourselves further let's actually work through the math for this problem. We are interested in $\Delta_k = \frac{\partial L}{\partial a_k^{(L)}}$. We will just write $a_k^{(L)}$ as a_k just for ease of notation. Writing out the partial derivative, we hvae:

$$\Delta_k = \frac{\partial \mathcal{L}}{\partial a_k} = -\frac{\partial}{\partial a_k}\left(\sum_j y_j \log \frac{e^{a_j}}{Z}\right)$$

Z is the normalization. For the denominator, because it's a sum over all e^{a_j}, then only the e^{a_k} term survives when we take $\frac{\partial Z}{\partial a_k}$. Now, let $p_j = \frac{e^{a_j}}{Z}$. Then:

$$\frac{\partial p_j}{\partial a_k} = \begin{cases} \frac{e^{a_j}}{Z} - \frac{e^{a_j}e^{a_j}}{Z^2} & j = k \\ -\frac{e^{a_j}e^{a_k}}{Z^2} & j \neq k \end{cases}$$

we can write this out in a more simplified form:

$$\frac{\partial p_j}{\partial a_k} = p_j(\delta_{jk} p_k)$$

where δ_{jk} is the kronecker delta. Thus, we have:

$$\begin{aligned} \Delta_k &= -\sum_j \frac{y_j}{p_j}\frac{\partial p_j}{\partial a_k} \\ &= -\sum_j y_j \frac{1}{p_j} p_j(\delta_{jk} - p_k) \\ &= -y_k + \left(\sum_j y_j\right) p_k \\ &= p_k - y_k \end{aligned}$$

The last step we use the fact that $\sum_j y_j = 1$, which is the case because it's zero for every class except the one we know it belongs to.

9.3 Generalized Loss

- We previously defined the error with respect to the last layer, but now let's try to generalize this to all layers. That is, we want to derive:

$$\partial_{\theta_{ji}^{(l)}} \mathcal{L}$$

for all the weights θ.

- There is an algorithm called *backpropagation* that gives us a way to compute the loss at the top layer $\ell = L$ and propagate it all the way down to $\ell = 1$. This algorithm is $O(m)$, which is better than the naive approach of computing the Taylor expansion which runs in $O(m^2)$.

9.4 Backpropagation

- So our objective is to compute the loss with respect to an arbitrary parameter, or $\frac{\partial \mathcal{L}}{\partial \theta_{ji}}$ from a neuron in layer ℓ to $\ell - 1$.
- Recall that the preactivation and the post-activation is written as:

$$a_j^{(\ell)} = \sum_i \theta_{ji}^{(\ell)} x_i^{(\ell-1)} \quad x_i^{(\ell-1)} = g(a_i^{(\ell-1)})$$

and the loss (depending on the type of model we use) is defined as:

$$\mathcal{L} = \begin{cases} \frac{1}{2}(y - a^{(L)})^2 & \text{Gaussian} \\ -y \log \sigma(a^{(L)}) - (1-y)\log\left(1 - a^{(L)}\right) & \text{logistic} \\ -\sum_k y_k \log \frac{a_k^{(L)}}{Z} & \text{multiclass} \end{cases}$$

And we also have the general formula from earlier:

$$\frac{\partial \mathcal{L}}{\partial \theta_{ji}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial a_j^{(\ell)}} \frac{\partial a_j^{(\ell)}}{\partial \theta_{ji}^{(\ell)}}$$

The second term is very simple because $a_j^{(\ell)}$ is linear in terms of $\theta_{ji}^{(\ell)}$, so the partial derivative just gives us $x_i^{(\ell-1)}$. For the first term, we can use our single-layer neural network to motivate the form. We know that the last layer is expressed in terms of the layers below, so to generalize this notion we introduce a chain rule because we have an activation function g:

$$\Delta_j^{(\ell)} := \frac{\partial \mathcal{L}}{\partial a_j^{(\ell)}} = \sum_k \underbrace{\frac{\partial \mathcal{L}}{\partial a_k^{(\ell+1)}}}_{\Delta_k^{(\ell+1)}} \frac{\partial a_k^{(\ell+1)}}{\partial a_j^{(\ell)}}$$

well we've just abstracted this out to the next layer up since the

first term in the sum is just the loss in the previous layer. For the second term in the chain rule, we know that

$$a_k^{(\ell+1)} = \sum_i \theta_{ik}^{(\ell+1)} g(a_j^{(\ell)})$$

so the derivative is:

$$\frac{\partial a_k^{(\ell+1)}}{\partial a_j^{(\ell)}} = \theta_{kj}^{(\ell+1)} g'(a_j^{(\ell)})$$

He mentioned something about a kronecker delta, where does it appear?

We are taking the derivative with respect to a particular $a_j^{(\ell)}$, so only one of these terms survive. This is why you have a kronecker delta and only one term in the result.

So finally, we can write:

$$\Delta_j^{(\ell)} = \left(\sum_{k \in [d_{l+1}]} \Delta_k^{(\ell+1)} \theta_{kj}^{(\ell+1)} \right) g'(a_j^{(\ell)})$$

- To interpret this, the error at any particular layer is determined by the accumulation of the errors at the layer above – this is why we call this "propagating" the errors back through the network. This is a recursive formula, which takes $O(m)$ time for m layers.
- In general, if you follow the structure of layer-by-layer computation, then this takes linear time, but if you have skip connections (connections between layer ℓ and $\ell-2$, for example), then this algorithm would take longer.

10 CNNs, ResNets, Batch Normalization

10.1 Summary of Last Lecture

- The structure of a neural network: we described the mathematical structure of a neural network and how each neuron behaves at every step:

$$a_j^{(\ell)} = \sum_{i \in [d_{\ell-1}]} \theta_{ji}^{(\ell)} x_i^{(\ell-1)}$$
$$x_j^{(\ell)} = g_\ell(a_j^{(\ell)})$$

- For maximum likelihood learning, we denote the total loss as the sum of the individual losses:

$$\mathcal{L}(\theta) = \sum_i \mathcal{L}_i(\theta) = \sum_i \mathcal{L}(f_\theta(x_i), y_i)$$

- Stochastic Gradient Descent (SGD): in its purest form, defined by the updates

$$\theta_{t+1} = \theta_t - \epsilon_t \nabla_{\theta_t} \mathcal{L}(\theta_t)$$

  where we select an ϵ_t at random at each iteration t.

- Backpropagation: we learned of an $O(m)$ algorithm to calculate the gradient of the loss throughout the neural net:

$$\Delta_j^{(\ell)} = \left(\sum_{k \in [d_{\ell+1}]} \Delta_k^{(\ell+1)} \theta_{kj}^{(\ell+1)} \right) g'(a_j^{(\ell)})$$

10.2 Convolutional Neural Networks (CNN)

- Firstly, fully connected neural networks are parameter hungry. Usually, too many parameters leads to overfitting of the data, which we know to be suboptimal.
- The motivation for CNNs comes from our desire to incorporate our "inductive bias" into the neural net. Inductive bias is our desire to force the neural network to learn certain aspects about the data.

  In our naive approach, if we wanted to forcefully change some parameters θ, then the network would respond in an unpredictable way because of its complexity. So, in order to control the neural networks in a way, we turn to convolutional neural networks.

10.2.1 Invariance and Equivariance

- They are both defined with respect to a particular transformation.
- As an example, consider a translation T:

$$\begin{array}{ccc} x & \longrightarrow & T(x) \\ \downarrow{\scriptstyle f_\theta} & & \downarrow{\scriptstyle f_\theta} \\ f_\theta(x) & \longrightarrow & f_\theta(T(x)) \end{array}$$

  the idea is that with our classification, we would like to have $f_\theta(x) = f_\theta(T(x))$. In this case, we call this **invariance.**

- On the other hand, if we have $f_\theta(T(x)) = T(f_\theta(x))$, then this operation is called **equivariance**.

  In general, because the spaces (dimensions) in which x and $T(x)$ live in may be different, what we really want in general is:

  $$f_\theta(T_x(x)) = T_y(f_\theta(x))$$

  In the case where $T_x = T_y$ we have the earlier form.

10.2.2 Structure of a CNN

- In a CNN, the operation from each layer to the next is a convolution, and no longer a dot product between the vector θ at layer ℓ and your weights. That is,

  $$a^{(1)}(i_1, i_2) = \sum_{j_1 \in [F]} \sum_{j_2 \in [F]} x^{(0)}(i_1 + j_1, i_2 + j_2)\theta(j_1, j_2)$$

  so now, θ is a 2-dimensional kernel we use to compute the convolution. This also means that each neuron is only connected to the neurons in the previous layer by the size of the convolution only.

  what does the bias actually mean in this case then? Is it some constant matrix θ_0?

  No, you do the convolution as normal, then add a bias b at the very end.

- Note that the filter at each layer doesn't have to be the same. If you were processing an image, one filter could be a horizontal gradient, and the other could be a vertical gradient. This is allowed.
- In the case of image processing, you may also need to take your convolutions channel-wise because your input is RGB.
- In general, visualizing the first layer of your network is relatively easy, because it's just a collection of the images you've learned.

  Is the first layer the output layer or the first layer through your convolution?

- Some technical terms related to neural nets:
  - **Padding:** When you do convolution on the edges, you may need to pad the edges with zeros (or some other value) so that the kernel completely overlaps with your matrix. In general, convolving a $(W \times W)$ matrix with an $(F \times F)$ filter, then you get a $(W - F + 1)$-dimensional matrix.
  - **Pooling:** You take the elements in a certain window of the matrix (say, the top left), and take the value according to some defined function. In the case of max-pooling, you take the max value, for example.
- When we finish training our model using SGD (or other methods), we can then look at the learned model by computing $\nabla_x a_i^{(\ell)}(x; \hat{\theta})$. Basically, we are looking for the behavior of the neural net to small changes in the data.
- One thing we find when training neural networks is that the depth of the neural net doesn't always lead to a better result, and is a result of the vanishing gradient.

  What is this vanishing gradient really mean here?

  Consider logistic regression, where your activation function is the sigmoid. You want to maximize the probability, which forces you to be in a regime where there are very small gradients. So, you hit a wall.

  Aside on vanishing gradients: Consider a function:

  $$f(x) = \left(\prod_{i=1}^{n} w_i \right) x$$

  and you have individual gradients:

  $$\frac{\partial \mathcal{L}}{\partial w^{(2)}} = \frac{\partial \mathcal{L}}{\partial x^{(3)}} \frac{\partial x^{(3)}}{\partial w^{(2)}}$$

  The reason vanishing gradients happen is because you multiply many of these terms together – and as a result, you're getting a *really* small term, and this leads to vanishing gradients.

10.3 ResNets

- One way to deal with the limitations of neural networks is to use "skip connections", or basically connect the previous layer directly to the next layer. That is, we have:

$$x^{(\ell+1)} = \mathcal{F}_{\ell+1}(x^{(\ell)}) + x^{(\ell)}$$

compared to what we had before (only the $\mathcal{F}_{\ell+1}(x^{(\ell)})$ the residual $x^{(\ell)}$ term (not sure)

so what does the residual term do that allows you to bypass the learning barrier?

It solves the issue of vanishing gradients, because the gradient of the last step directly influences the first step.

- As a result, deeper ResNets actually do perform progressively better.

10.4 Batch Normalization

- When you want to train a neural net, and you don't want inputs from different layers of the network to be in different ranges. Because of the way backpropagation works (each next layer is dependent on the previous one), then you'd like all the gradients and networks to live in similar ranges so that you don't get random spikes for no reason.
- To accomplish this, you normalize each hidden layer in the same way you normalize the input data! That is, in each batch, we can compute the statistics of the batch:

$$\mu_B = \frac{1}{n}\sum_{i\in[b]} x_i, \quad \sigma_B = \frac{1}{n}\sum_{i\in[b]} (x_i - \mu)^2$$

then normalize each layer based on the statistics of the batch. That is, for each node in the neural network, we then have:

$$x^{(\ell)} \leftarrow \frac{x^{(\ell)} - \mu_B}{\sigma_B + \epsilon}$$

this normalizes each $x^{(\ell)}$ in each batch so make sure that the ranges are consistent across each layer. Remember: this is done at *every layer*, so you compute μ_B and σ_B for each layer, and normalize it.

- You also want to augment $x^{(\ell)}$ by learned γ, β:

$$x^{(\ell)} \leftarrow \gamma x^{(\ell)} + \beta$$

and why do you do this?

Don't worry about it, it's not important.

11 Recurrent NNs, Attention & Transformers

11.1 Sequence-to-Sequence Models

- To motivate the later topics in this lecture, one question we should answer is how we deal with variable length inputs. In the case of protein folding, you have different length sequences for different proteins, so how do you get your neural net to predict parameters like the stability, or structure of the protein?

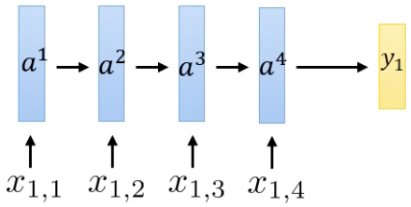
  In the most interesting case, your output is contingent on the length of your input. This is the case in language translation.

- In general, you need a way to handle variable length inputs, and models that do this are called **sequence-to-sequence models**.
- Let's first consider only variable length inputs, so we have something like:

$$x_1 = (x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4})$$
$$x_2 = (x_{2,1}, x_{2,2}, x_{2,3})$$
$$x_3 = (x_{3,1}, x_{3,2}, x_{3,3}, x_{3,4}, x_{3,5})$$

because you want to have variable length inputs, you can't just feed the neural network the entire x_1 because then you lose the ability to have variable length.

- One possible way you could do this is to feed each dimension of the input vector sequentially. That is, for x_1, we first feed it in $x_{1,1}$, then $x_{1,2}$, then $x_{1,3}$, etc. As a diagram:



so, the output a^3 hinges on the output a^2, and so on. For inputs with shorter length, we can just prepend a bunch of zeros, which does nothing to the data.

- In principle, this would work, but the major problem with this direct approach is that the number of layers in your network is equal to the length of the largest input. If you have very few long inputs, then the model won't be very good at predicting the long inputs because there isn't enough data out there.

  To fix this, we tie the layer parameters together, which is called a recurrent neural network. It's called recurrent because each a^ℓ depends recursively on $a^{\ell-1}$.

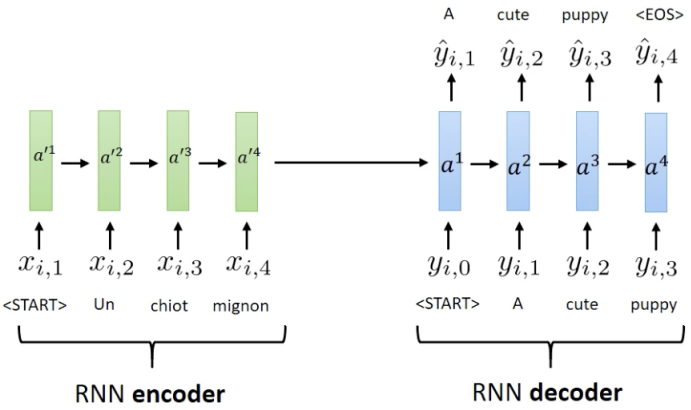
- Something else you could do is to have an output for each input, and "decode" the output from that specific layer. The computation is the same, the only thing we've changed is that we're decoding the information at every step.

- You could also use what's called an **autoregressive model**, which uses the output of the previous iteration to generate the next output. This is what ChatGPT does.

  How is this model different than the first one?

  You are getting the output at every step here, in the first type you're not. In other words, you're getting the output of each next word, which is generated based on the prevoius word.

- From this process, hopefully it's clear that there's this general setup going on: you have an encoder that "encodes" the input data, and a decoder that you run through your neural network to decode the output. That is, the general structure could look something like this:



Note that the RNN encoder could be replaced by a CNN encoder if we were dealing with images. Point is, you have a neural network that *encodes* the input data, and a decoder that *decodes* the data.

The encoder translates the input into a "content vector" that describes the content, then the RNN decoder does the actual translation.

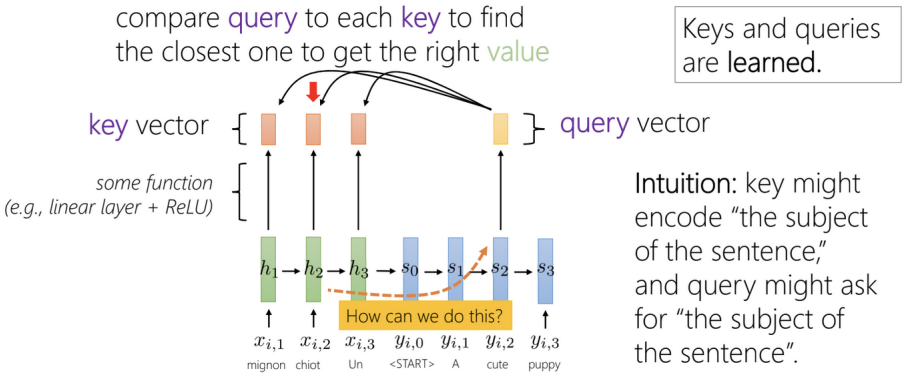
11.2 RNN Bottleneck, Attention

- One major problem with this encoder-decoder structure is the transition between the encoder and the decoder, called the bottleneck. In essence, if you have very long recurrences, then the earlier features that are passed in tend to get washed out the longer your recurrence goes.

  This is the motivation for solutions that involve attention and transformers.

- Suppose you don't want all the information to be bottlenecked by the last output of the encoder, is there a way we can just translate intermediate outputs? The answer is yes, via attention.

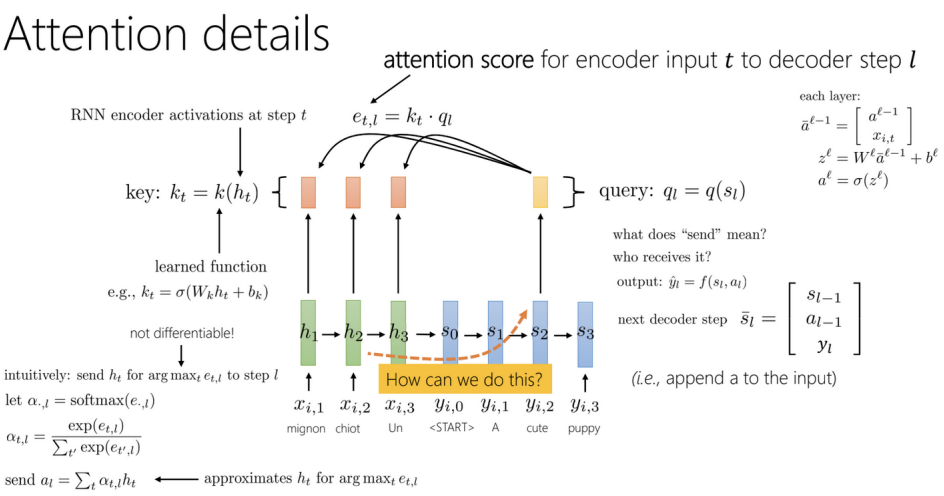
- Imagine the standard recurrent neural network (image below), and you're trying to decode s_2.



we can generate a query vector based on the hidden state at s_2, that allows us to query back in time and look for matches. This is a probabilistic lookup, so it picks the one that has the highest similarity, based on some heuristic (maybe a dot product or something).

The intuition to have is that if you're trying to decode the subject of the sentence, the query vector goes back in time and looks for the subject of the sentence from the encoder.

- Refer to the diagram below:



  - Suppose you're at s_2, and you're looking to query. To compute the keys, you take the hidden input h_t at every layer, and you put it through a function $k(h_t)$. k could be as simple as the identity, but it can also be a simple learned function as well (e.g. $k_t = \sigma(W_k h_t + b_k)$).

  - When you're query, you put it through a query function $q_l = q(s_l)$. q_l is a vector representing some concept, and we take the inner product of q_l with every input k_t. This quantity $e_{t,l}$ is called the **attention score**.

- Intuitively, we would like this to be a database lookup, which would basically be the same as looking for argmax $e_{t,l}$. But, because this is not differentiable, we normally use a softmax instead. That is:

$$\alpha_{t,l} = \frac{\exp(e_{t,l})}{\sum_{t'} \exp(e_{t',l})}$$

Now, the combination of all of these is a normalized probability distribution, then we take a linear combination of the attention score with the information in the input sequence back. That is, we send $a_l = \sum_t \alpha_{t,l} h_t$ back. Basically, this ensures that the things which are sent back with high probability are the ones that correspond highly with the data.

- After sending a_l back, we can then use this information combined with s_2 to finally get a prediction $\hat{y}_2$.
- The number of attention scores you have depends on the length of the input string. The computational complexity is then dependent on the length of your input.

- Some general examples of $k(t)$ and $q(t)$ are just the identity function:

$$k_t = h_t \quad q_l = s_l$$

so the decoder is just the inner product between the encoder and decoder. You can also use more complicated functions, such as:

$$k_t = W_k h_t \quad q_l = W_q s_l$$

where W is a parameter we augment by (is this learned?). On the decoder side, you then have:

$$e_{t,l} = h_t^\top W_k^\top W_q s_l = h_t^\top W_e s_l$$

- You can also do weird things with the returned a_l as well. Instead of just taking the dot product of the attention score with each layer, you can also use a learned function here, so you return

$$a_l = \sum_t \alpha_t v(h_t)$$

where $v(h_t)$ is some learned function.

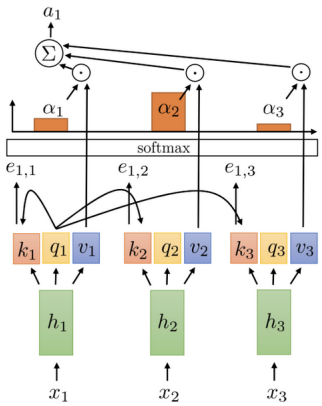
We keep saying this "learned function", how do you learn functions like this? Do you have another neural net that does this?

- Attention is a **very** powerful tool! There is no longer a bottleneck because each layer can just use attention. This is very similar to how resents work – we're shortening the gradient path, so this makes the neural net better.

11.3 Transformers

- Is attention all we need? Do we even need the recurrence or the autoregressive model? The answer is no, and what we can actually do is just rely only on attention. This is what transformers are.
- The only issue you have with the current model is that you can only look at the key vectors for the input pairs, but not the decoding layers. To solve this, we implement **self-attention**. So, we basically make a key vector for *every layer*, including the ones in the decoding network.
- Refer to the following diagram:

Self-Attention (one layer)



$a_l = \sum_t \alpha_{l,t} v_t$

$\alpha_{l,t} = \exp(e_{l,t}) / \sum_{t'} \exp(e_{l,t'})$

$e_{l,t} = q_l \cdot k_t$

$v_t = v(h_t)$ before just had $v(h_t) = h_t$, now e.g. $v(h_t) = W_v h_t$

$k_t = k(h_t)$ (just like before) e.g., $k_t = W_k h_t$

$q_t = q(h_t)$ e.g., $q_t = W_q h_t$

this is *not* a recurrent model!
but still weight sharing:

$h_t = \sigma(W x_t + b)$

shared weights at all time steps

(or any other nonlinear function)

  - The intermediate arrows are now gone, and instead it's replaced by these k_i, q_i, v_i values. We still have a shared weight $h_t = \sigma(W x_t + b)$, but they are no longer connected to each other.
  - Now, each layer has a value v_t, which is defined by $v(h_t)$, just as before. The key vector is also stored, in the same way as before $k_t = k(h_t)$. The query is also the same as before.
  - So basically, this is a very dense version of what we had before – instead of having the keys in the encoder and the queries in the decoder, they are now all in the same system (so to speak).
  - If you're querying for h_1, then you query through k_1, k_2, k_3, and compute the attention in the same way we did before. The only difference is that you're querying your own key k_1 as well.
  - a_l is defined the same way, through the softmax, etc.
- Now that we've removed the recurrence, this means that the network is now permutation equivariant: if you switch up the order of h_i the output doesn't change, which is bad.
- You can also stack attention layers on top of each other in the same way you stack a convolutional neural network.
- Some issues with our model so far:
  - Lack of sequence information: by removing the recurrence we've removed the ordering.

    To fix this, we just force-feed position information into $h(t) = f(x_t, t)$. If you just feed it in something simple (like 1, 2, 3, ...), it actually doesn't do very well. What people actually do is give it a positional encoding p_t which is a giant vector of sines and cosines.
  - Multi-headed attention: in the same way you can have multiple kernels for your neural network, you can also have an attention layer for each kernel, which is called multi-headed attention.

    You can do multi-headed attention in parallel, the mechanisms are exactly the same, except in a higher dimension because we have more queries, keys, and values.
  - Linearity: each successive layer is linear in the previous one

    Fixing this just requires you to add a some nonlinear function in between the attention layers.
  - Masked decoding: How do we prevent attention lookups into the future?

    This is mainly an issue caused by addressing the first issue – when we bake an ordering into the system, then we have to restrict what each layer can query, which we can do by modifying the dot product to be:

$$e_{l,t} = \begin{cases} q_l \cdot k_t & t \le l \\ -\infty & \text{otherwise} \end{cases}$$

    This solves the issue because now you'll get a $-\infty$ attention score for the things you're not supposed to look at.

Note that throwing this into the softmax will give you $e^{-\infty} = 0$, so it does work out as expected.

- And now we've essentially built a transformer. Nowadays, transformers usually just refer to this stacking of self attention layers.
- Some downsides is that this is pretty slow, as it runs in $O(n^2)$, compared to $O(n)$ (roughly) due to backpropagation. It's also slightly harder to implement.

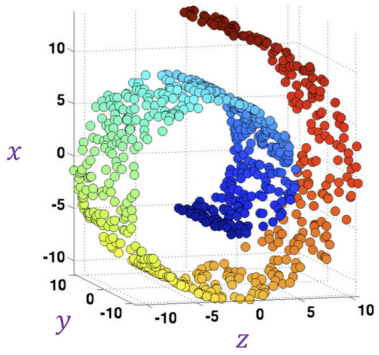
  That said, it also has many benefits, which outweigh the downsides: they have much better long-range connections, they're much easier to parallelise, and you can also make them much deeper than you can with RNNs.

12 Dimensionality Reduction, PCA

- Before we dive into this, we should talk a bit about unsupervised learning. So far, with supervised learning, we have data in the form of $\{(x_i, y_i)\}$ where you have an x and also an associated value y. In unsupervised learning, you only have $\{x_i\}$, with no associated outputs.

12.1 Dimensionality Reduction

- What does it even mean to reduce the dimension? Suppose you have 3-dimensional data that lives a spiral:



  Could you determine the dataset by using 2 coordinates? The answer looks like yes.

- In general, this is also the case. The manifold hypothesis says that high dimensional data tends to lie in the vicinity of a low dimensional manifold, basically motivating the fact that we can reduce dimensions without losing information.

  A manifold for us is basically just a space that locally feels like a euclidean space. For us, this manifold is going to be the lower dimensional part of the observational space in which our data will lie.

  The operation of dimensionality reduction is called an *embedding*.

12.2 Principal Component Analysis (PCA)

- The trick with dimensionality reduction is to pick a clever basis in which we can still faithfully represent all images, in a lower dimension.
- As an example, imagine first that your data lies in 2 dimensions. To reduce dimension, we basically want to look for the direction along which we retain the most information if you project the data onto it. To determine the direction, you find the direction which minimizes the *reconstruction loss*.

  It turns out, the direction is the one which maximizes the variance of the data. Intuitively, this is because projecting onto a high variance makes the region in which the data lies larger.
- The idea of PCA is to recursively apply this method – we pick the first direction, then we pick the second direction which maximizes variance (which is orthogonal to the first one), and so on. The maximum number of basis vectors is obviously the dimension of the original space, but obviously you don't want to do that because your whole goal is dimensionality reduction.

- As an aside, there is a probabilistic PCA that uses MLE which derives the exact same result.
- To do PCA, you take the spectral decomposition (aka eigendecomposition) of the covariance matrix $A = QDQ^\top$, to find the principal components.
- As an overview, this is what we do:
  - Start with a data matrix $X \in \mathbb{R}^{n \times d}$, center the data (zero mean) and then construct $X^\top X$, which is now symmetric.
  - Apply spectral theorem, so we get $X^\top X = QDQ^\top$ and take the top k eigenvectors to pick off the k directions you want.

    How are the eigenvalues related related to the variance?
  - Now, you approximate the new matrix with the best rank k matrix. You can do this by projecting all the original points X onto this new subspace, to make $\overline{X}_k = \overline{X}Q_k$. Here, Q_k is the matrix of the first k eigenvectors.
- To visually see what information you lost, you can reconstruct the data using $\overline{X}_{\text{reconstructed}} = \overline{X}_k Q_k^\top = \overline{X}Q_k Q_k^\top$. You can then compute the reconstruction loss, which is given by:

  $$\|\overline{X}_{\text{recon-k}} - \overline{X}\|_F$$

  It can also be proven via the Eckart-Young theorem that PCA gives a minimal reconstruction loss.
- So what do you do pick k? You can compute the cumulative variance you keep from picking the first k, and decide based on that.

12.3 Singular Value Decomposition (SVD)

- For any matrix M, we can decompose it into $M = U\Sigma V$, where U, V are square matrices, and Σ is a "diagonal" rectangular matrix. U, V have the property that they're both orthonormal, so $UU^\top = VV^\top = I$.
- If you have very high dimensional images, then spectral decomposition would take an extremely long time because it's cubic in that dimension. SVD on the other hand is linear in the higher dimension and quadratic in the smaller, so it's computationally cheaper.
- The column vectors in U are the eigenvectors of $MM^\top$, which is exactly what we wanted to find in PCA anyways! So, you get both spectral decompositions at once, for $M^\top M$ and also $MM^\top$.
- The eigenvalues are the same, and are given by $\lambda_i = \Sigma_{ii}^2$, where Σ_{ii} lies on the diagonal of the Σ matrix.

13 t-SNE

- So far in PCA, the assumption is that you have a dataset $\{x_i\}$ that you can find the principal components of. What if you instead are given $X^\top X$?
- As a practical example, what if you were just given the pairwise distances between cities M? The solution is that you can think of $M = XX^\top$ of some unobserved data X. This is slightly different from PCA, where we use $X^\top X$.

  Why do we assume $XX^\top$ instead of $X^\top X$?
- We can still use SVD, except we use the eigenvectors in U instead of V, which is what we used for PCA.
- The issue with PCA is that it is a linear mapping (using SVD), but in general this is not true with all the data we get. So, we need something more complex.

  In particular, we need an *embedding* of what we define to be neighbors in our dimensionality reduction. The idea is to look locally and make sure that things are embedded locally only, and not really care about what happens globally. In the case of PCA, it treats neighbors globally, which is why it fails to dimension-reduce some datasets.

13.1 Neighborhood Embedding (NE)

- The idea is basically to define what the neighbors of a point are. You can do this by iterating through each point, and build an "edge" between them, then compute the distances using this new function you've created.
- Then, once you get all the pairwise distances you can now create M, from which you can do PCA.
- One issue with this though is that the quality of your reduction depends heavily on the nearest neighbor graph you create. So, instead of *telling* you what the neighbors are, we can build a probability distribution instead.

13.2 Stochastic NE (SNE)

- Here, we make the event that two samples are neighbors be a random variable. More specifically, the probability that x_i "chooses" x_j is proportional to:

$$P_{j\leftarrow i} = \frac{\exp\left(-\|x_i - x_j\|^2/2\sigma_i^2\right)}{\sum_{k\neq i}\exp(-\|x_i - x_k\|^2/2\sigma_i^2)}$$

  and we also have $P_{j\leftarrow j} = 0$.

  Is this just softmax?

- We then symmetrize this and renormalize it, so we make $P = \frac{1}{2n}(P_{j\leftarrow i} + P_{i\leftarrow j})$.
- This value of σ tat you pick is that you choose it differently, so that the entropy of anything choosing x_i is constant. That is:

$$\sum_{j\neq i} P_{j\leftarrow i}\log P_{j\leftarrow i} = \text{const.}$$

- We're now going to posit that after the dimensionality reduction, we get a space Y, and the points live in Y somehow. We can also get the same neighborhoods in Y in the exact same way, and we can make these $Q = \{Q_{ij}\}$:

$$Q_{ij} = \frac{\exp\left(-\|y_i - y_j\|^2\right)}{\sum_{l\neq k}\exp\left(-\|y_l - y_k\|^2\right)}$$

  Here, we don't have to mess around with σ, because Y is in a space which we learn anyways. what does learn mean in this case?

- We now want to find a distribution of Y that best represents the neighborhoods in the original space X, so effectively we want $Q \approx P$. To do this, we want to minimize the KL-divergence between P and Q. The KL-divergence is defined as:

$$KL(P \parallel Q) = \sum_{i,j} P_{ij}\log\frac{P_{ij}}{Q_{ij}}$$

  This is not symmetric, so $KL(P \parallel Q) \neq KL(Q \parallel P)$.

- One of the problems with dimensionality reduction is that points tend to clump up, so intuitively in order to find better neighbors, the distribution on Y should be more sparse than the distribution on X. Based on this intuition, we arrive at t-SNE.
- In reality, t-SNE makes only one change to our approach, which is that instead of using a Gaussian we use a student-t distribution for $Q_{j\leftarrow i}$, so we have:

$$Q_{j\leftarrow i} = \frac{\left(1 + \|y_i - y_j\|^2\right)^{-1}}{\sum_{i\neq k}(1 + \|y_i - y_k\|^2)^{-1}}$$

  The student-t distribution has the benefit that along the edges you have more mass, so the probability of points further away in Y being neighbors is higher.

14 Clustering

- Recall before, we talked about unsupervised learning, and we talked about dimensionality reduction, and saw PCA (linear) and t-SNE (nonlinear) as ways to do that.
- Intuitively, the idea of clustering is to try to assign points that are obviously part of a cluster to a given label. These labels aren't given to us ahead of time, we're just given the data in high dimensions and are asked to classify them into clusters.
  - In terms of biology, this could be useful to identify various different organ cells in a digital scan.
  - You might also want to cluster subtypes of a disease, and cluster people based on their genetic data so you can figure out what particular gene causes a specific condition.
  - This is a very old technique that is still being used today.
- Clustering is also used for outlier detection, where points which do not fit into a particular cluster can be labeled as noise or outliers.

14.1 Approaches to Clustering

- Hierarchical clustering: we have to build a tree representing the distances between two points, and join them together to form clusters.
- **Model based approaches:** We maintain cluster "models" and infer membership from these models.

14.2 Aside: Distances, metrics

- Ultimately, the main property of clustering is that points within a given cluster are close to each other, but points in different clusters end up far away. This is the main feature point of clustering.
- The clustering also heavily depends on how we interpret the notion of distance! With different distance metrics, the result from clustering can be wildly different!
- The formal definition of a distance (metric) is:
  1. $j = k$ iff $d(j,k) = 0$.
  2. $j \neq k$ iff $d(j,k) > 0$
  3. symmetry: $d(j,k) = d(k,j)$
  4. triangle inequality: $d(i,j) + d(j,k) \geq d(i,k)$.
- There's also this other notion of dissimilarity, which does not satisfy metric properties. We sometimes use dissimilarity and similarity instead of distance, and there are reasons we do this.

14.3 k-means

- k-means is an example of centroid based clustering. Within each cluster, we basically define a "representative point" that somehow encapsulates all the features in the data, and acts as the "centroid" of the cluster. As it turns out, the parameters in k-means is the centroid of each cluster.
- The hyperparameter is k, and the model parameters are $\{c_k\}$. We want to choose c_k such that the distance to each point x_i to its assigned centroid is minimized.
- Formally, we have $X = \{x_i\}_{i=1}^n$, $x_i \in \mathbb{R}^d$, and the parameters are $\{c_k \in \mathbb{R}^d\}$. The optimization problem that defines k-means is:

$$\underset{S=\underbrace{C_1 \cup \cdots \cup C_K}_{\text{cluster partition}}, \underbrace{\{c_1, \ldots, c_K\}}_{\text{cluster centroids}}}{\operatorname{argmin}} \sum_k \sum_{x \in C_k} \|x - c_k\|^2$$

- To explain how we arrive at an answer, suppose we knew the partition $C_1 \cup C_2 \cup \dots C_K$. In other words, we knew which points were red, black, purple, etc. How could you find the clusters? Well, because you know the cluster labels, then for each cluster you'd have to minimize c_k, which comes down to

$$\hat{c}_k = \underset{c_k}{\operatorname{argmin}} \sum_{x \in C_k} \|x - c_k\|^2$$

  The answer to this is just to take the mean of the points: $\hat{c}_k = \frac{1}{N} \sum_{x \in C_k} x$.

- Now, suppose you only knew $\{c_k\}$, how would you find $C_1 \cup \dots \cup C_K$? Then, for each point you'd just assign each point to the cluster c_k that is closest to it. So, the label z_i for each point is just going to be $z_i = \operatorname{argmin}_k \|x_i - c_k\|^2$. Then, once you've done that for each point you get the resulting partition C_k.

- Now for the overall algorithm, called Lloyd's algorithm.

  - We start by initializing the cluster centers $\{c_k\}$. This initialization is random, so we just pick k random points from our data and set those as our starting c_k.
  - We repeat the following until convergence:
    i. Compute partition $C_1 \cup C_2 \cup \dots \cup C_k$ given the $\{c_k\}$.
    ii. Compute the centers $\{c_k\}$, given $C_1 \cup \dots \cup C_K$.

  The key point in this algorithm is that when we recompute the centers $\{c_k\}$, these centers will change until we've reached the optimal point. In other words, the objective function always decreases when we change the centers $\{c_k\}$ or compute $C_1 \cup \dots C_K$.

  You can also initialize the clusters $C_1 \cup \dots \cup C_K$ as well, there's no reason you need to initialize the centers $\{c_k\}$ first.

14.4 Gradient Descent

- We can also use gradient descent here! Note that once we found the clusters c_k, we can also compute the loss function over the c_k to find the partitions for every point:

$$L(\{c_k\}) = \sum_i \min_k \|x_i - c_k\|^2$$

  Now, all we'll do here is that we'll choose the c_k that is closest to x_i, and call this point z_i. Then, we can change our loss function slightly:

$$L(\{c_k\}) = \sum_i \sum_k \|x_i - c_k\|^2 [z_i = k]$$

  Then, to minimize this we can take the gradient of $L(\{c_k\})$ with respect to c_k:

$$\nabla_{c_k} L(\{c_k\}) = -2 \sum_i (x_i - c_k)[z_i = k]$$

  And now, with the gradient computed, we can do gradient descent! The formula becomes:

$$c_k' = c_k - \eta N_k c_k + \eta \sum_{x_i | z_i = k} x_i$$

  What is N_k? If we have a small enough learning rate, this algorithm is also guaranteed to converge. This algorithm converges to a local minimum however, so we'll have to run this a couple times to guarantee that the clustering is optimal.

  Does the Lloyd algorithm converge to a global minimum?

- The gradient descent is prone to being caught in local minima. In some cases, the k-means algorithm will split some clusters down the middle, and also sometimes waste a centroid to split two clusters that are actually supposed to be a single cluster.

14.5 Weaknesses of k-means

- This is not a weakness of k means, but in essence the symmetries that are preserved under some transformation (e.g. length under rotation), then the k-means algorithm will output clustering independent of rotation. Thus, you can imagine that sometimes desired symmetries we would like to have are not always respected.
- There are no likelihood functions, so it's hard to understand assumptions. We can't use MLE or the other cool things we determined in lecture.
- Also, the clusters are implicitly assumed to be of a spherical shape, because we pick a centroid which is effectively the "center" of the cluster when measured with Euclidean distance.
- There is also no uncertainty in the clustering, even though there really should be – sometimes it's hard to tell whether a point belongs in one cluster over the other, and this algorithm doesn't do that, it instead just throws the point into one of the two clusters.

14.6 Soft k-means

- This is a "softer" version of k-means, which allows for uncertainties in the clusters. Previously, we had:

$$z_i = \underset{k}{\operatorname{argmin}} \|x_i - c_k\|^2$$

We're going to convert this to a maximization problem:

$$z_i = \underset{k}{\operatorname{argmax}} \exp(-\|x_i - c_k\|^2) = \underset{k}{\operatorname{argmax}} \{v_{ik}\}$$

this doesn't change anything; the z_i that satisfies the earlier relation are the same z_i that satisfies this maximization. We now normalize v_{ik} such that $r_{ik} \equiv \frac{v_{ik}}{\sum_k v_{ik}}$.

- What we have done here is convert our clustering assignment from a hard "one hot" type of classification to a softer version, through the use of the r_{ik} normalization process.

  The v_{ik} contain information about the distances between x_i and cluster c_k, so the set $\{v_{ik}\}$ for every (x_i, c_k) pair gives you information about how close x_i is to each cluster. Taking the softmax is the same as taking the argmax in the sense that z_i doesn't change, but you do get more information this way.

  If this is the case, then what is the point of doing all this? Doesn't this method still just classify x_i into a designated cluster?

  I think it still does, but the key here is that we never use z_i anymore, and just use the vector r_{ik} instead to label. This gives you a probabilistic model.

- You can also add a hyperparameter β to the equation, and set:

$$r_{ik} = \frac{\beta v_{ik}}{\sum_j \beta v_{ij}}$$

You can do MLE on this β later. We don't have a likelihood function yet, but we will. The effect of β is that for small values, it has the effect of looking more uniform, and for large values it is basically a "one hot" function. This is what we call a "temperature" in machine learning, where we scale a distribution by a temperature T. Scaling with T usually increases the entropy of the distribution, or in other words it makes the distribution more uniform.

- Now, we can tweak the k-means algorithm as follows: instead of assigning z_i to a single x_i, we instead assign z_{ik} with:

$$z_{ik} = \text{softmax}(-\beta \|x_i - c_k\|^2)$$

this no longer returns a single label, but instead a vector of distances. Then, we replace $\hat{c}_k$ as:

$$\hat{c}_k = \underset{c_k}{\operatorname{argmin}} \sum_{i=1}^{N} z_{ik} \|x_i - c_k\|^2$$

The difference here is now that instead of taking the mean of every point assigned to the cluster C_k, we are now taking the mean over *all* data points, but weighting it with how much we assigned the point to C_k.

14.7 Mixture of Gaussians

- Even though we've "softened" the k-means algorithm, it still isn't good enough because it still assumes a spherical distribution as it treats each x_i with equal weight. So, this motivates the Mixture of Gaussians (MoG) model.
- This really is just a generalized version of a k-means algorithm – we can make it look like a k-means algorithm by constraining the covariance matrix of our Gaussians to be spherical. All we have to do is to write down the likelihood and crank through MLE like we've done before.
- To do this, we will make use of *latent variables*. Let $z_i = \{1, \dots, K\}$ be an unobserved assignment of x_i to a cluster. We define this because it will make our calculations easier, but because we don't know its value we will just sum it out first. Therefore, the likelihood is:

$$\mathcal{L}_i = p(x_i) = \sum_{k=1}^{K} p(x_i, z_i = k)$$

  So the likelihood function here is just a sum over the probability that x_i is assigned to cluster $z_i = k$, over all such z_i. Now, we can simplify this using Bayes' rule:

$$\begin{aligned}\mathcal{L}_i &= \sum_{k=1}^{K} p(x_i \mid z_i = k)\, p(z_i = k) \\ &= \sum_{k=1}^{K} N(x_i \mid \mu_k, \Sigma_k)\, p(z_i = k)\end{aligned}$$

  We know that $p(x_i \mid z_i = k) = N(x_i \mid \mu_k, \Sigma_k)$ by our assumption that we are using Gaussian distributions here. Then, we sum over all possible clusters k. We can then write $p(z_i = k) = \alpha_k$. Intuitively, α_k is basically a representation of the proportion of points that belong to that particular cluster.
- Ultimately, there are three things that we want to learn here: $\theta = \{\mu_k, \Sigma_k, \alpha_k\}$. We can then use MLE on this, by setting the log likelihood over all the points:

$$\mathcal{L} = \log \prod_i \mathcal{L}_i = \sum_i \log \mathcal{L}_i$$

  You can solve this problem in the same way as before, just cranking through MLE by computing gradients and such.

15 Nearest Neighbor and Metric Learning

15.1 Parametric vs. Non-Parametric Models

- So far, we've mostly focused on parametric models, where we aim to learn $p(y \mid x)$, through the use of a parameter θ:

$$p_\theta(y \mid x) \approx p(y \mid x)$$

- The parameters are determined through MLE (or some other method), and generally the data is then thrown away. Today, we will look at non-parametric models (i.e. models which don't consider a parameter θ). In this case, we will keep the training examples, and in effect the number of training parameters grows with $n = |\mathcal{D}|$. In some sense, the dataset itself is the parameters.
- As an aside, we should cover metric spaces first: a metric space is a set X together with a notion of distance d between its elements:
  1. $d(x, x) = 0$.
  2. $d(x_1, x_2) > 0$ if $x_1 \neq x_2$.
  3. $d(x_1, x_2) = d(x_2, x_1)$.
  4. Triangle inequality.

  One metric we commonly see in ML is called the *Mahalanobis distance*, written as:

$$d_M(x_1, x_2) = \sqrt{(x_1 - x_2)^\top M (x_1 - x_2)}$$

  this collapses to the Euclidean distance when $M = I_d$.

15.2 KNN Classifier

- Suppose you've classified some data into two classes, and now you want to classify a new point x. To do this, we first find the K closest (using a metric d) examples in the pre-existing data to inform our decision on what x should be classified as. We denote this set as $N_K(x, \mathcal{D})$.

  How is the pre-existing data generated?

- We then look at the labels y_i for the points $N_K(x, \mathcal{D})$, and use it to estimate what $p(y \mid x)$ should be (i.e. what label we assign it):

$$p(y = c \mid x, \mathcal{D}) = \frac{1}{K} \sum_{i \in N_k(x, \mathcal{D})} I[Y_i = c]$$

  note that this probability sums to 1 when we sum over all c, so this is actually a valid probability distribution. In words, this means that the probability a point y is assigned a label c is given by the proportion of the nearest K points that are assigned c.

- The $k = 1$ case is special - the partition that you get is called a *Voronoi tessellation*. In this case, if you scatter some points onto the plane, then this division will partition the space up strictly based on distance and all the boundaries are just straight edges between classes.

  In this special case, the classification of a new point is given by the nearest neighbor, so this is why you get such a special pattern.

  How do you tell the classifier how many classes you should make?

  This is probably something you decide ahead of time; there's probably no way for you to "train" a model to know how many you should make.

- Visually, you can also compare different K and K' nearest neighbor algorithms, and there is a distinction between them. For larger K, the general trend is that the boundary between classes gets finer and not as jagged, as is typically seen in a smaller K.

- There is also an optimal K for classification. If K is too large, then you risk comparing against points that are not near your target point y, and therefore you will get a bad classification from it.

  This is also a general trend we observe in model training: as the model complexity increases, the training error decreases monotonically, but the test error reaches an optimum somewhere in between. This is because there are two regimes on the extremes: extreme simplicity on one end, and overfitting on the other.

15.3 KNN Generative Classifier

- Now we move to the K-nearest neighbors except this time we have a generative classifier. In this case, we essentially define a "ball" around each point x until we encounter K points, and the classification is then given by the proportion of points in that ball $V_K(x)$ that are classified as class c.

- If you have a *lot* of data, it has been shown that the KNN test error can never be worse than twice the optimal Bayes classifier. This is interesting to note, because the Bayes classifier requires *much* more information about your data (the class conditionals, distribution, etc.), but KNN knows nearly nothing.

- This is a quick aside, but basically the idea is that the space you need to check increases exponentially with dimension. Because the volume grows exponentially large, it becomes increasingly more impractical to search such a space, and so this makes KNN much worse in higher dimensions.

  If you want to get ϵ close to a target, by increasing the dimension you can only get $O(n^{-1/d})$ closer to the target by increasing the dimension.

15.4 Pros and Cons of KNN

- Pros:

  - Fast, no training required.
  - Learns complex classification functions easily, because it requires no priors.
- Cons:
  - High storage cost
  - Slow at computing inference (actually performing the classification)
  - Very bad dimensionality scaling.

15.5 Manifold Hypothesis

- The manifold hypothesis basically states that "true" high dimensional data that we care about, like images, videos, etc. lie in a lower dimensional manifold which is embedded in a higher dimensional space.
- The idea then is that we essentially "embed" the data onto a lower dimensional space using a neural network, learn the space using a Euclidean metric, then you can classify using this Euclidean metric.

  Is this the approach that t-SNE uses?
- The challenge then becomes: how do you define an embedding that does exactly what you want? Ideally, you want an embedding that pulls similar objects close to each other and pushes dissimilar ones apart.
- One of the earliest approaches to doing this was based on *contrastive loss*. Basically, define a loss function over two points i, j:

  $$\mathcal{L}_{ij}(\theta; m) = \mathbb{I}(y_i = y_j) d(z_\theta(x_i), z_\theta(x_j))^2 + \mathbb{I}(y_i \neq y_j) \max(0, m - d(z_\theta(x_i), z_\theta(x_j))^2)$$

  (recall that $z(x)$ is your embedding function). Essentially, you minimize the loss between similar points, while also defining a "maximum acceptable distance" between two dissimilar points. You require this m because otherwise, your model will want to push dissimilar points increasingly farther away off to infinity.

  Tested this on the MNIST dataset and bringing this down to a 2-dimensions from 784 dimensions, and the results look very good.
- The problem with this approach is that the "pull" and "push" terms don't talk to each other, or in other words this is a very binary way to approach this embedding approach.
- This leads to the second approach: triplet loss. Here, the loss is given by:

  $$\mathcal{L}_i(\theta; m) = \max(d(z_\theta(x_i), z_\theta(x_i^+))^2 - d(z_\theta(x_i), z_\theta(x_i^-))^2 + m, 0)$$

  In essence this does the exact same thing: you want the first term to decrease while simultaneously wanting the second term to increase, but this is a more clever approach because the terms are combined together.

  Here, z^+ is a positive (similar) example, and z^- is a dissimilar (negative) example.

  what is the positive and negative example referring to?

  these are *anchors* in the data, and are given as true points in the dataset.
- One issue with the triplet loss is that because there are a lot of negative examples to compare to, you naturally need to perform that computation many many times to get the loss for a single point. This leads to the N-pair loss, which lets you take a batch of negaitve samples at a time:

  $$\mathcal{L}(\theta; x, x^+, \{x_k^-\}_{k=1}^N) = \log\left(1 + \sum_{k=1}^{N-1} \exp\left(\hat{z}_\theta(x)^\top \hat{z}_\theta(x_k^-) - \hat{z}_\theta(x)^\top \hat{z}_\theta(x^+)\right)\right)$$

  what this means is that you take a set of predefined dissimilar examples $\{x_k\}$ and use that as the "push" term.

  How does this algorithm work in practice? Would you have to define a set of similar and dissimilar classes for every class you're working with?

The reason you don't need a max function here is because $\hat{z}_\theta(x)$ is a projection that only projects into a volume of a unit sphere, so there is a predefined limit to how far apart dissimilar points can be apart. You can also convert this interpretation into an equivalent one using softmax.

- There's also the world of joint embeddings: where text and images are encoded together and embedded onto a space: OpenAI's CLIP model was trained on an N-pair loss in a jointly embedded space.

16 Multiway Classification, Decision Theory, Model Evaluation

- Recall cross-validation: the process of keeping a "test set" of data hidden to you that you will validate your data on. You can do this with classifiers as well (why wouldn't you be?)
- Then, in order to heuristically evaluate how "good" the classification is, this entire time we've been using the log-likelihood to do this.
- As it turns out, the log-likelihood is not the best possible heuristic to do this with, we will see different ways to evaluate a classifier here.

16.1 Choosing between classifiers

- Whenever we train a dataset and it sees a data point, the model is then able to classify, and also give a probability that the data point lies in the returned class. One assumption we've been working with is that the predicted probabilities returned by the model are very close to the "true" probability distribution.

  This is something that is made easier through the use of a prior, but even so there is the possibility that the model probability is drastically different than the true probability.
- One thing we could do is use a threshold of $p = 0.5$, and count the number of misclassifications the model makes. This would work if the model is *calibrated*, or in other words the distribution of classification reflects the true distribution. In non-calibrated models, you can see that this goes wrong, because the model probability does not reflect the true probability, so picking $p = 0.5$ doesn't make sense at all.
- If the model isn't calibrated, then there's basically another value of p that does this classification better. For other models like SVMs which don't return a probabilistic model, this approach just doesn't work because they don't work with probabilities at all.
- Miscalibrated models aren't totally useless either! Even though a model could be miscalibrated, you can treat the probabilities more as a score, so the higher the score the higher probability the object belongs to its class. So, there is still some threshold p that we can choose (that isnt 0.5) that minimizes error.

16.2 False Positives, False Negatives, etc.

- For now, we will only consider binary classifiers for simplicity.
- Now, some terms:
  - **True positives:** people who are classified "positive" and are actually "positive".
  - **False Positive:** people who are classified "positive" but are truly "negative".

  The notion of true/false negatives are the inverse of this, of course.
- The intersection between the positive and negative distributions represents the upper limit on the accuracy of the model – this makes sense, your model can only do as well as the true distribution.

  As an extension of this, it means that if two distributions have any intersection at all, then there is no model you can construct that will classify with 100% probability.
- Some more definitions:
  - TPR: the fraction of true positives/the number of actual positives.

  - TNR: the fraction of true negatives/number of negatives.
  - FNR: the fraction of false negatives/actual positives
  - FPR: the fraction of false positives/actual negatives .

  The rates are in general calculated as the number of instances you classify one way, divided by the total number of things you could have classified that way.

- It should be obvious, but changing the decision threshold will change the TP, TN, FP, FN rates.
- As we shift these around, we can plot a ROC curve, which plots one parameter in terms of the other. For instance, you can plot the true positive and false positive rate against each other.

  Remember, you want to maximize the true positive rate, so for a perfect classifier its ROC curve would just be a point in the top left corner. Then, how close we are to the perfect classifier gives us another way to evaluate how good a classifier is. There is a threshold to this however, since as mentioned before the perfect classifier may not be attainable.

- It's not true that classifiers always "do better" over the entire dataset. In the real world, two classifiers can have varying true positive vs. false positive curves, where one "beats" the other at different true/false positive values.

  So how do you determine which ROC curve is better? You look over the regime that you're interested in. If you're not willing to go above a false positive rate of above 20%, then you can restrict your analysis to false positive rates between 0-20%.

16.3 Making ROC Curves

- To make the ROC curve, we will leverage the fact that anything that is classified as positive using a given thershold will be classified as positive for lower thresholds.
- First, we sort the data based on the score our model gives. The scores then become the possible thresholds you can take for your data.
- Another way to compare two classifiers is to use the area under curve (AUC) method to rank classifiers. This gives the probability that the classifier will rank a positive higher than a negative. The bigger the AUC, the better the classifier is.

  Using this intuition it then makes sense that the random classifier has an AUC of 0.5, since it classifies a random data point correctly with 0.5 probability. So, it will classify a positive higher than a negative only half the time.

- You can either take the integral under the ROC curve, or you could calculate it directly using trapezoidal integration. You can also do the partial AUC, where you restrict your analysis to a limited set of false positive rates.
- ROC curves have the advantage of being independent of the number of actual positives and negatives in your dataset; this is because the rates are normalized to the total number of positives/negatives in your dataset.

  This invariance is actually not good, because it means that to find the misclassification rate you actually need information about the ratio of actual positives to actual negatives in the test set.

- An alternative is to use a precision recall curve, which does take into account the proportion of true positives and negatives. The precision is defined as:

$$\text{precision} = \frac{\text{true positive}}{\text{number of predicted positive}} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

  The recall is just the TPR. Then, you can plot one against the other to get the precision-recall curve. A perfect classifier in this case exists in the top right corner.

- ROC curves are not useful if you actually care about the true probability distribution, because the ROC curve is invariant with respect to the actual data.

  If you just care about rankings, then ROC is perfect, because you *only* care about the ranking, so the underlying distribution doesn't matter to you.

16.4 Models with AUC

- The question now becomes: why don't we just use the AUC as a loss function to train a model?
- The AUC and losses are not differentiable everywhere, so it's difficult to minimize this function. (Traditional methods like gradient descent and SGD are therefore not possible)
- You also can't use mini-batches anymore, since the AUC requires the entire data set.
- There are some very niche uses to make these metrics differentiable, but they are rarely used.

16.5 Training Classifiers

- Similar to how we changed the way we evaluate classifiers, there are also other ways we can train a model to learn from a dataset.
- Traditionally, we used MLE and minimized the loss function. In doing this, we become blind to the classes themselves, which is bad because classifications are not all held equal.
- In a more general sense, we should be using a *cost-sensitive* classifiers, where we penalize some errors much more severely than others. For instance, if a model predicts that you don't have cancer while you do, then this is an *extremely* expensive mistake, but the log-likelihood is blind to this fact.
- To fix this, you can use a weighted sum over the loss:

$$\underset{w,b}{\operatorname{argmin}} 1000 \sum_{y_i=1} L(y_i, f(x_i \mid w, b)) + \sum_{y_i=-1} L(y_i, f(x_i \mid w, b))$$

  you're basically forcing the model to care about some mistakes more than other mistakes, which gives you a different classifier. The exact ratios of the importance you should place to each class is something that we have to do on our own, and can't be done through machine learning.

17 Decision Trees, Ensemble Approaches

- Decision trees are very powerful – in instances where we can use decision trees to perform classification, they almost always outperform alternative methods.
- One nice thing about decision trees is that you don't really have to go through the trouble of hyperparameter tuning, and they are also highly interpretable, unlike neural networks.
- You can really think of these as a game of "20 questions", where you ask a bunch of yes or no questions and decide the classification based on the answers.
- The nonlinearity offered by recursive decision boundaries is very powerful, and is something linear regression models cannot do.
- Some cons: the decision boundary is not smooth, and they are relatively unstable and sensitive to changes in the data.

17.1 Structure

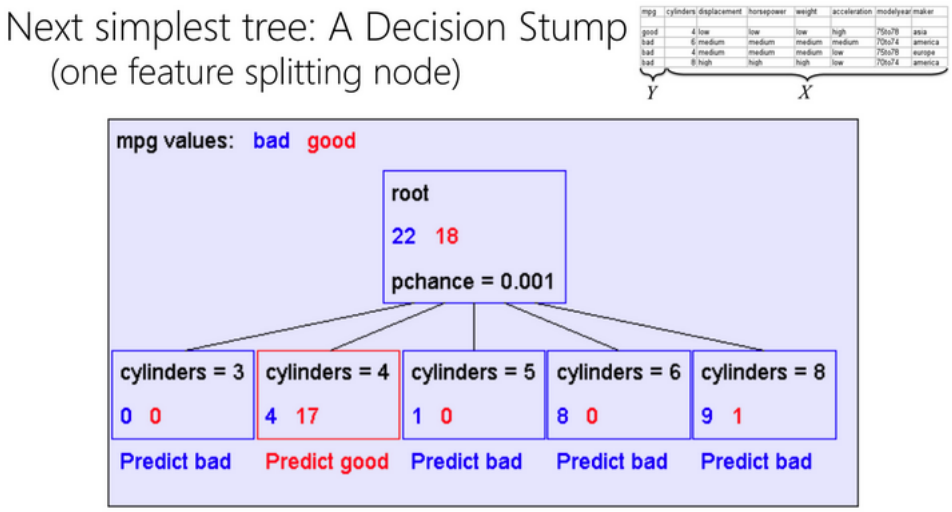
- The goal is that given a set of "features" X, if we can predict some other feature. For instance, given the horsepower, weight, maker of a car (X), we can be asked to predict the miles per gallon, Y.
- Mathematically, we are essentially trying to learn a class of functions $f : X \to Y$ where X is the features and Y are the possible outputs.
- Before we look at how the decision tree is built, we look at how we use one. Given some input data, the decision tree determines the order in which we query the parameters to determine a result. These queries may not be the same length, as sometimes a single query is enough to determine a result.

  Each node in the decision tree is essentially a query to an attribute x_i, and depending on its value we follow the corresponding edge. Leaves in this tree correspond to the classifications y.

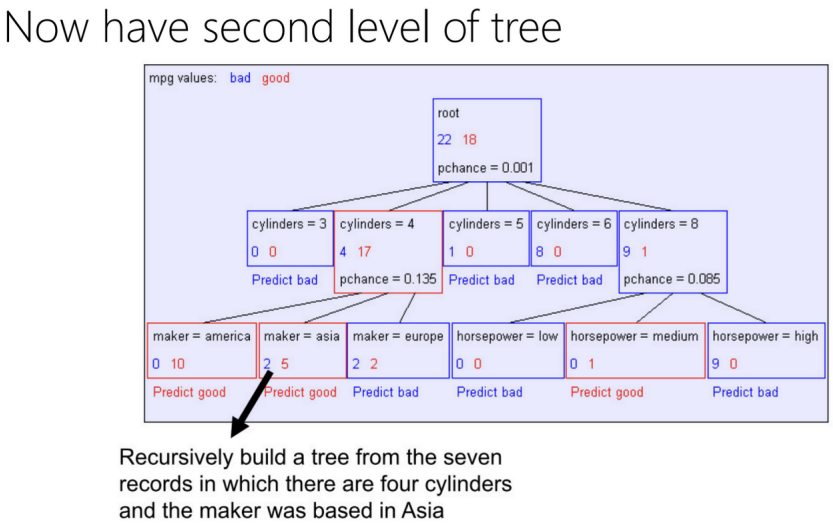
- Compared to linear models like KNNs, decision tree boundaries *must* be axis-aligned. This is an issue, because it means that the decision boundary is usually more complex than linear regression models, or it would take more information to replicate the same boundary.
- Another benefit of decision trees is that because they can grow arbitrarily large, its limit is basically a truth table, so this means that you can represent any function of the input attributes with a decision tree.

17.2 Building Decision Trees

- To start, the simplest decision tree we make is one which doesn't consider the data at all, and just classifies all data based on the y that is most common in the training data. Obviously this is not interesting, so we move on.
- Then, you look at a single feature, and create a node for every possible value that the feature could take on. Then for each of these values, find the y that is the most common, and that becomes your prediction.



So here we first query on the number of cylinders, then based on the majority mpg (either good or bad) in each cylinder category we can come up with a classification that makes sense. We then recursively do this, and find the next best query to look at. This will be different for different nodes:



- So how do we grow this tree? What feature do we split on? The goal is to make leaf nodes to be pure (consisting of only one class), so we want splits which increases the purity of the leaf nodes.

  One way to quantify this purity is to quantify how surprised we are at seeing a particular result, and then compute the

entropy, which is the expected surprise overall. The entropy is defined as:

$$H(Y) = -\sum_k P(Y = k) \log P(Y = k)$$

The negative is there to signify the fact that rare events are more entropic than common ones. Pure nodes which don't have misclassifications have zero surprise, so finding the best split is the same as finding the classification which minimizes entropy.

- Thinking of distributions more generally, high entropy tend to spread their mass over the entire space, whereas low entropy generally has its probability mass concentrated at a particular point.
- In the training data, we can't compute teh entroyp of Y, but we can control the entropy of $p(Y \mid X)$, since we control the feature X that we want to look at. So, our goal is to reduce the *conditional entropy* at each node so that we eventually reach a surprise of zero.

  The conditional entropy is just given by:

$$H(Y \mid X_{j,v}) = P(X_{j,v} = 1) H(Y \mid X_{j,v} = 1) + P(X_{j,v} = 0) H(Y \mid X_{j,v} = 0)$$

  By the rules of expectation, this conclusion should be relatively obvious. Here $X_{j,v}$ is a binary random variable, so we only have two terms, but you can imagine a case where we have more. The point here is that we split on the values that $X_{j,v}$ could equal, then compute the entropy for each subcategory and weight them accordingly.

  Some books use the base-2 logarithm here; it makes no difference versus base-10 log.

- Equivalently, we can phrase this as maximizing the *information gain*, which is defined as:

$$I(X_{j,v}; Y) := H(Y) - H(Y \mid X_{j,v})$$

  $H(Y)$ is the entropy you start with (constant), and compute the conditional entropy for the different $X_{j,v}$ labels and maximize this quantity.

  There are no parameters and "learning" here! This is really just chugging through to find best possible feature to split on. For real valued features, you pick a threshold to split on instead.

- When do you stop recursing on a node? There are two cases:

  1. If the category Y is pure after the classification, so all results after sorting are of a single class.
  2. If the remaining features to split on don't improve the data at all, then the node becomes unexpandable. Note, this is not the same as zero information gain. In cases like these, you just flip a coin to decide how to predict.

     In the slides, they write this as "all input values are the same", how is the lack of further classification (or lack of improvement) related to the fact that all input values are the same? Couldn't you just have a 1-1 tie with the input values not being the same?