CS 181 Spring 2023 Midterm I Review Guide:

Alyssa Huang, Dylan Hu

1 High Level Overview

For the full checklist of Midterm 1 content, please refer to this guide put together by Weiwei!

1.1 Midterm 1 Content (Supervised Learning)

In supervised learning, we are given both the input $\mathbf{x}$'s and labels y during training. There are two main types of supervised learning problems.

1. Regression - labels y are continuous real numbers (usually, e.g. predict future stock price in dollars).
2. Classification - labels y are discrete and categorical (e.g. is this a picture of a pig, panda, or parakeet?).

2 Broader Impact Analysis

- How does our choice of dataset affect our model's preditive abilities? Consider bias.
- How does our choice of evaluation metrics our model's preditive abilities? Consider the different loss functions we have.
- What should we optimize on top of just loss? Give examples related to real-world applications.
- What considerations must you make when selecting a model? Be able to give examples of domain-specific knowledge that may influence model selection.
- Why might we want more interpretable models?

3 Regression

3.1 Residuals and Loss Functions

How is a model (such as linear regression) related to a loss function (such as least squares)?

- The model (of the data) and the loss functions are both important pieces to the ML pipeline, but they are distinct. The model describes how you believe the data is related and/or generated. Very commonly, you will be optimizing over a family of models. The loss function measures how well a specific model (i.e. with specific parameters) fits the data, and it is used in the previously mentioned optimization.
- Least squares and linear regression are often used together, especially since there are theoretical justifications (i.e. MLE connection) to why least squares is a good loss function for linear

regression. However, you do **not** have to use them together. Another loss function that could be used with linear regression is an absolute difference (L1) loss.

The **least squares loss function** for a general model is given as follows:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2 .$$

where $\hat{y}_n$ is the predicted value for y_n. For linear regression, we can write

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \mathbf{w}^\top \mathbf{x}_n\right)^2 .$$

because our model assumption is linear.

If we minimize our least squares loss function with respect to the weights, we get the following solution (assuming $\mathbf{X}$ is full-rank to take the inverse):

$$\nabla\mathcal{L} = \mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y}) = 0 \rightarrow \mathbf{w}^* = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y} = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w})$$

However, we need not choose the least squares loss function as our loss function. For example, we could have mean squared error or mean absolute error:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2$$

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}|y_n - \hat{y}_n|$$

Low train loss but high test loss is indicative of **overfitting**. High train loss and high test loss is indicative of **underfitting**.

3.2 Probabilistic Regression

Suppose we have a dataset $\mathcal{D} = \{(x_1, y_1), \ldots, (x_N, y_N)\}$. where $x_n \in \mathbb{R}^D$. We will assume that the target variables y are **generated** from the data x as follows:

$$y = w^T x + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

where ϵ is the noise. **Note that the choice of distribution for ϵ can easily be changed to follow another distribution (i.e. Laplace or Student-t)**. We will treat σ as some known value that we are given. From the above, we have that each $y|x, w$ is a random variable with the following distribution:

$$y|x, w \sim \mathcal{N}(w^T x, \sigma^2)$$

For our model, we will select some value for w. We denote the **joint likelihood** of the model as:

$$p(y_1, \ldots, y_N | x_1, \ldots, x_N, w) = \prod_{i=1}^{N} p(y_i | x_i, w)$$

We have the above equality as we assume that the data is i.i.d, and each y_i only depends on the corresponding x_i.

Intuition: What is the likelihood? Looking at the equation, it tells us for some choice of w what the probability of the observed dataset being generated using that w is. From this, it would make sense that we want to find a w that **maximises the likelihood** of the data. We call this value the maximum likelihood estimate (MLE) for w, denoted by w^{MLE}. We can encode this objective as follows:

$$\begin{aligned} w^{MLE} &= \arg\max_w \; p(y_1, \ldots, y_N | x_1, \ldots, x_N, w) \\ &= \arg\max_w \prod_{i=1}^{N} p(y_i|x_i, w) \\ &= \arg\max_w \log \left(\prod_{i=1}^{N} p(y_i|x_i, w) \right) \\ &= \arg\max_w \sum_{i=1}^{N} \log(p(y_i|x_i, w)) \end{aligned}$$

This is known as the **log likelihood (denoted by ℓ)**. To find the maximising w we will take the derivative of the function we want to optimize, set the derivative to 0 and then solve for w. Taking the derivative of the likelihood is very difficult because it is a product of many terms, so we would have to use the product rule over and over again. Taking the derivative of the log likelihood is much easier however as it is a sum, so we can just differentiate each term separately.

Returning back to our example, we have the log likelihood $\ell(w)$ as:

$$\ell(w) = \log \left[\prod_{n=1}^{N} p(y_n|x_n, w) \right] = \sum_{n=1}^{N} \log p(y_n|x_n, w)$$

We now plug in the Gaussian PDF, remembering that we have $y_n|x_m, w \sim \mathcal{N}(w^T x_n, \sigma^2)$:

$$\begin{aligned} \ell(w) = \log \left[\prod_{n=1}^{N} p(y_n|x_n, w) \right] &= \sum_{n=1}^{N} \log p(y_n|x_n, w) \\ &= \sum_{n=1}^{N} \log \left[\left(\frac{1}{\sqrt{2\pi\sigma^2}} \right) \exp \left\{ -\frac{(y_n - w^T x_n)^2}{2\sigma^2} \right\} \right] \\ &= \sum_{n=1}^{N} \log \left[\frac{1}{\sqrt{2\pi\sigma^2}} \right] - \sum_{n=1}^{N} \frac{(y_n - w^T x_n)^2}{2\sigma^2}. \end{aligned}$$

If our noise follows another distribution, we would substitute in the PDF of that other distribution at this step.

To maximize the log-likelihood $\ell(w)$, we compute the gradient of $\ell(w)$ with respect to w, set it equal to 0 and solve for w. We see that the gradient of the log-likelihood is the same (within a multiplicative factor) as the gradient of the mean squared error loss function! **This means that maximizing the log-likelihood also minimizes the MSE**.

3.3 Non-probabilistic Regression

Non-parametric means that we do not make any assumptions about the parameters / characteristics of our data (in context, this means that we do not assume an underlying probability distribution). In this class, we have covered two non-probabilistic regression algorithms:

1. KNN
2. Kernelized Regression

3.3.1 K-Nearest Neighbors (KNN)

In KNN, we want to take an *average* of the k points in our training data that are closest to our new data point when outputting our prediction for this unknown point. Rundown of the KNN Algorithm:

1. Let $\mathbf{x}^*$ be the point that we would like to make a prediction about. Let's find the k nearest points $\{x_1, \dots x_k\}$ to $\mathbf{x}^*$, based on some predetermined distance function.
2. Denote the true y values of these k points as $\{y_1, \dots y_k\}$.
3. Output our prediction $\hat{y}^*$ for our point of interest $\mathbf{x}^*$:

   $$\hat{y}^* = \frac{\sum_{i=1}^{k} y_k}{k}$$

   Remark: the little "hat" in $\hat{y}^*$ is just notation telling us that this is our *prediction*, rather than the true y value.

3.3.2 Kernelized Regression

In Kernelized Regression, we want to take a *weighted average* of all the points in our training data when outputting our prediction for an unknown point. Intuitively, we want to weigh points that are "closer" to our unknown point of interest *more heavily* than points that are "farther" away. As such, our kernel function $k(\mathbf{x}^*, \mathbf{x_n})$ should be *larger* for a point $\mathbf{x_n}$ closer to our point of interest $\mathbf{x}^*$ than a point $\mathbf{x_n}$ farther away.

Rundown of the Kernelized Algorithm:

1. Let $\{x_1, \dots x_N\}$ (and their corresponding y values) be *all* of the N points comprising our training data set.

2. Let $\mathbf{x}^*$ be our point of interest that we want to make a prediction for. We make our prediction as follows:

$$\hat{y}^* = \frac{\sum_{i=1}^{N} k(\mathbf{x}^*, x_i) \cdot y_i}{\sum_{j=1}^{N} k(\mathbf{x}^*, x_j)}$$

   Remark: notice how we include *all* points of the training data in our calculation?

   Remark: we have to include the denominator term in order to normalize the sum of our weights to equal 1.

3.3.3 KNN vs Kernelized Regression

Intuitively, both KNN and Kernelized Regression rely on the assumption that similar datapoints will have similar predictions.

Kernelized Regression is considered to be a smoother, more general extension of KNN. Recall from the problem set graphs that KNN tends to form more discrete decision boundaries (step-like). We would also recommend reviewing from that problem set how choices of k and our kernel function (effectively τ in the problem set) affect our predictive abilities.

3.4 Inference and Optimization

In general, when we are optimizing a parameter (W) with respect to a loss function ($\mathcal{L}$), we want to do the following steps:

1. Take the derivative of the loss function with respect to the parameter ($\frac{d\mathcal{L}}{dW}$).
2. Set this derivative to 0 ($\frac{d\mathcal{L}}{dW} = 0$). Solve for the parameter (W).

3.5 Model Interpretation

A **validation set** contains data that are separate from our training set used to fit the regression. Here is a sample process:

1. Separate our full dataset into a training set and validation set (say in a 90/10 split).
2. Train your models with different parameters on the training set. Each time, check the performance on the validation set.
3. This gives you an optimal value for the parameter.

Cross validation is a more sophisticated technique for obtaining validation losses.

1. In k-fold cross validation, we split our data into k equal chunks.
2. For each chunk, we set it to be the validation set and use the rest of the $k-1$ chunks together as the training data to fit our model.
3. Then, we obtain a validation loss on our current chunk.

4. Averaging loss over the k chunks gives us a final validation loss.

Now, we have an improved way of computing validation losses by averaging. This reduces the variance in the resulting validation loss - since we've used every data point in doing so!

Ensemble methods take advantage of multiple models to obtain better predictive accuracy than with a single model alone. The two most common types are bagging and boosting.

1. Bootstrap Aggregating (Bagging)
   - To predict data in the test set, we either use an average of the predictions from the individual models (for regression) or take the majority vote (for classification).
   - This tends to lower variance without changing bias, since it's an average of models!
   - **Example:** Random forest, which is an average of predictions from decision trees!
2. Boosting
   - In boosting, we train the individual models sequentially. After training the i^{th} model on a sample of the training set, we train the $(i+1)^{th}$ model on a new sample based on the performance of the i^{th} model.
   - Thus, examples classified incorrectly in the previous step receive higher weights in the new sample, encouraging the new model to focus on those examples.
   - During testing, we take a weighted average or weighted majority vote of the models' predictions based on their respective training accuracies on their reweighted training data (i.e. higher models have larger weights).
   - **Example:** The Adaboost algorithm is a common example.

3.6 Bias Variance Tradeoff

We want to choose a model class that is expressive enough to learn the underlying function that generated the data (avoid underfitting), but not so expressive that it learns the training set perfectly and doesn't generalize (avoid overfitting). Note that once you have chosen a model class, f, the final learned model f_w is random, dependent on what our training set $\mathcal{D}$ is. We consider two different cased for our model class f:

1. f is too expressive. In this case, f_w will overfit the training data $\mathcal{D}$. This means that our model class f is thus **very sensitive** to the choice of training data. The model will have **high variance** and **low bias**.
2. f is not expressive enough. In this case, f_w will underfit the training data $\mathcal{D}$. So the model class f is **not sensitive** to the choice of training data. Thus f has **low variance** and **high bias**.

Ideally we want a model that is low variance and low bias. From the above there seems to be a tradeoff between the two however. As we make our models more expressive, the bias will go down but the variance will go up. One of the most important things you will learn in this class is how important choosing you model class is, and that is rooted in the **bias variance tradeoff** we have outlined here. The tradeoff is best captured in the following formula

$$\mathbb{E}_{(\mathbf{x},y),\mathcal{D}}\left[(y - f_{\mathbf{w}}(\mathbf{x}))^2\right] = \mathbb{E}_{\mathbf{x}}\left[\text{noise}(\mathbf{x}) + \text{bias}^2(f(\mathbf{x})) + \text{variance}\left(f_{\mathbf{w}}(\mathbf{x})\right)\right].$$

3.6.1 Regularization

Regularization refers to the general practice of modifying the model-fitting process to avoid over-fitting. Linear models are typically regularized by adding a *penalization term* to the loss function. The penalization term is simply any function R of the weights $\mathbf{w}$ scaled by a penalization factor λ. The loss then becomes:

$$\mathcal{L}_{reg}(D) = \sum_{i=1}^{N}(y_i - f(x_i;\mathbf{w}))^2 + \lambda R(\mathbf{w})$$

Here are two common choices for our function R:

1. LASSO Regression
   (a) One common choice for a penalization term is simply $R(\mathbf{w}) = ||\mathbf{w}||$. This is just the L_1-norm of the weights vector, which quite naively means that the penalization term here is just the sum of the magnitudes of all the weights for the model.
   (b) The full modified loss is then:
   $$\mathcal{L}_{LASSO}(D) = \sum_{i=1}^{N}(y_i - f(x_i;\mathbf{w}))^2 + \lambda||\mathbf{w}||$$
   (c) Does not have a closed-form solution, meaning that it cannot be analytically solved.
   (d) Leads to sparse solutions (shrink coefficients to zero).
2. Ridge Regression
   (a) Minimizes a modified least squares loss function:
   $$\mathcal{L}(D) = \sum_{i=1}^{N}(y_i - f(x_i;\mathbf{w}))^2 + \frac{\lambda}{2}||\mathbf{w}||^2$$
   (b) Has a closed form solution, which makes the solution much more computationally efficient. The analytical solution is:
   $$\mathbf{w}_{ridge} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$
   (c) Connected to a Normal prior.

4 Classification

Classification is a supervised and discrete model that predicts a category for a set of inputs, rather than a continuous value as in regression.

1. Goal: Given an input vector $\mathbf{x}$, assign it to one of K discrete classes C_k.
2. Strategy: Divide our input space into disjoint (i.e., no overlap) decision regions whose boundaries are called decision boundaries or decision surfaces; for K decision classes, we should have K decision regions.

4.1 Geometric Interpretation of Classification

One distinction in classification is between soft and hard classification. In hard classification, the boundaries between classes are well-defined. For example, if we are classifying pictures of animals, there are correct and unique classifications for each. In soft classification, these boundaries are not as well-defined; as such, we assign probabilities for each class for their likelihood of being the correct class. One example might be an spam email classifier, where whether or not the email is spam is not clear and could even depend on the receiver.

We can now focus on one fundamental model in classification: binary logistic regression. In binary logistic regression, we only have two classes, 0 and 1. For our model setup, say we have a probability distribution for the label of a certain data point y given its features $\mathbf{x}$ for some weights $\mathbf{w}$ with intercept implicitly rolled in:

$$p(y=1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x})$$
$$p(y=0|\mathbf{x}) = 1 - \sigma(\mathbf{w}^T\mathbf{x}).$$

Note that we can also use logistic regression on some feature mapping $\phi(\mathbf{x})$ in the same way (and likewise for all later expressions) such that

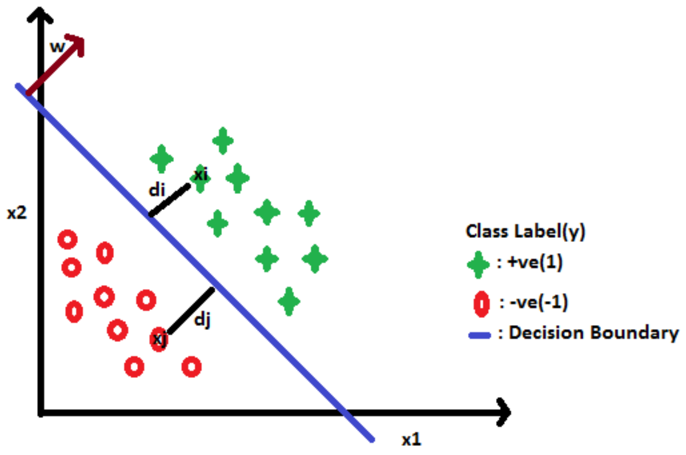
$$p(y=1|\mathbf{x}) = \sigma(\mathbf{w}^T\phi(\mathbf{x}))$$
$$p(y=0|\mathbf{x}) = 1 - \sigma(\mathbf{w}^T\phi(\mathbf{x})).$$

Here, the σ denotes the sigmoid function, which a non-linear function defined as

$$\sigma(z) = \frac{1}{1+\exp(-z)}$$

and used to translate any value on the real line to an output from (0, 1). The goal of the sigmoid function is to output some value interpretable as a probability of some label being in class 1. This nicely aligns with the goal of soft classification, lending well to a probabilistic interpretation. If we want a hard classification instead, we can discretely choose whichever class has a higher probability (or otherwise select a threshold to choose one class over the other) and assign our predicted label to that.

Visually, this binary logistic regression gives us a linear decision boundary for some set of weights $\mathbf{w}$, with green-colored data one class and red-colored data the other class.



We can see that some points are closer to the decision boundary and some are further away. We might naturally believe that points closer to the decision boundary are less certain to be the class they are assigned to, although in real-life, that might not strictly be the case.

4.2 Probabilistic Interpretation of Classification

In order to train our classifier, we can set up a loss function like we did for regression. For logistic regression, we will use the negative log-likelihood as the loss function, starting with the likelihood function:

$$\mathcal{L}(\mathbf{w}) = \prod_{i=1}^{n} p(y_i = 1 \mid \mathbf{x_i})^{y_i} p(y_i = 0 \mid \mathbf{x_i})^{1-y_i}$$

for $y_i \in \{0, 1\}$ based on the label's true classification. Note that the likelihood function is particularly convenient because our y_i values can be encoded as binary. Taking the log and the negation gives

$$L(\mathbf{w}) = -\ell(\mathbf{w}) = -\sum_{i=1}^{n} (y_i \ln p(y_i = 1|\mathbf{x}_i) + (1 - y_i) \ln p(y_i = 0|\mathbf{x}_i))$$

where we can recall that the probabilities depend on $\mathbf{w}$ from the $\sigma(\mathbf{w}^T x_i)$ term in each.

4.3 Inference and Optimization

To train this model, we want to minimize our negative log-likelihood. Note that because of the non-linearity introduced by the sigmoid function, we cannot analytically solve for the optimum like we did in least squares linear regression. Instead, we have to use (stochastic) gradient descent to iteratively approach the weights for a local minimum. Computing this gradient gives

$$\nabla L(\mathbf{w}) = \sum_{i=1}^{n} (\sigma(\mathbf{w}^T x_i) - y_i) x_i$$

which we can then use for gradient descent:

1. Randomly initialize some $\mathbf{w}^{(0)}$ weights.

2. Compute the gradient with the current set of weights.
3. Update the weights as
$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha \nabla L(\mathbf{w}^{(t)})$$
4. Iterate until convergence.

Some notes on gradient descent:

- Choosing the correct step size α is very important for an efficient descent without introducing numerical instability. If α is too small, convergence will be too slow. If α is too big, then the iterative descent can overshoot the minimum and therefore will not converge.
- Gradient descent can only compute local optima, as opposed to the global optima analytically solved for in least squares regression. As such, we can run trials gradient descent with different initializations of $\mathbf{w}^{(0)}$ to attempt to find the global minimum, but that is not guaranteed.

After the model is fit with optimized weights, we can then use some test data point $\mathbf{x}^*$ to predict y^* by computing
$$\hat{y}^* = \sigma(\mathbf{w}^T \mathbf{x}^*)$$
against some set threshold.

4.4 Model Evaluation

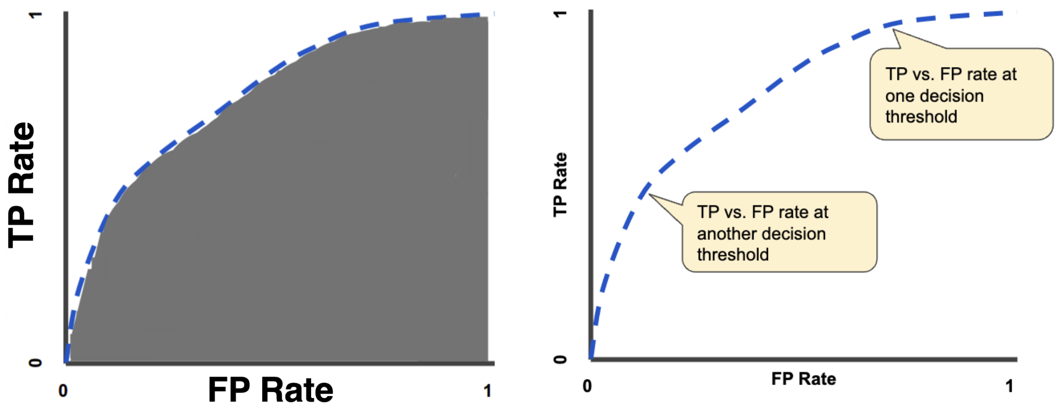
To evaluate how accurate a model is, a few statistics can help summarize model performance:

- True Positive/True Negative (TP/TN): Predicted values align with true values (a positive is actually a Positive, negative actually a negative).
- False Positive/False Negative (FP/FN): Falsely (incorrectly) predicted values for positive or negative labels (mislabeling a False as a True = False Positive).

To normalize these as percentages, we have the True Positive Rate (TPR), the ratio of true positive predictions to the total number of positive instances, and the False Positive Rate (FPR), the ratio of false positive predictions to the total number of negative instances, regardless of whether they were correctly or incorrectly classified. There are also symmetric negative rates.

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad \text{TNR} = \frac{\text{TN}}{\text{TN} + \text{FP}} = 1 - \text{FPR}$$
$$\text{FPR} = \frac{\text{FP}}{\text{TN} + \text{FP}}, \quad \text{FNR} = \frac{\text{FN}}{\text{TP} + \text{FN}} = 1 - \text{TPR}$$

However, these rates are not great evaluations. Consider a classifier that classifies everything as positive; this is not a good classifier, even though it would yield perfect TPR and FNR rates. We can then introduce the Receiver Operating Characteristics (ROC) curve, which is used to plot TPR and FPR at different classification thresholds. (Note that we only need to plot TPR and FPR because TNR and FNR are symmetric from them.)



Source: Google ML Crash Course

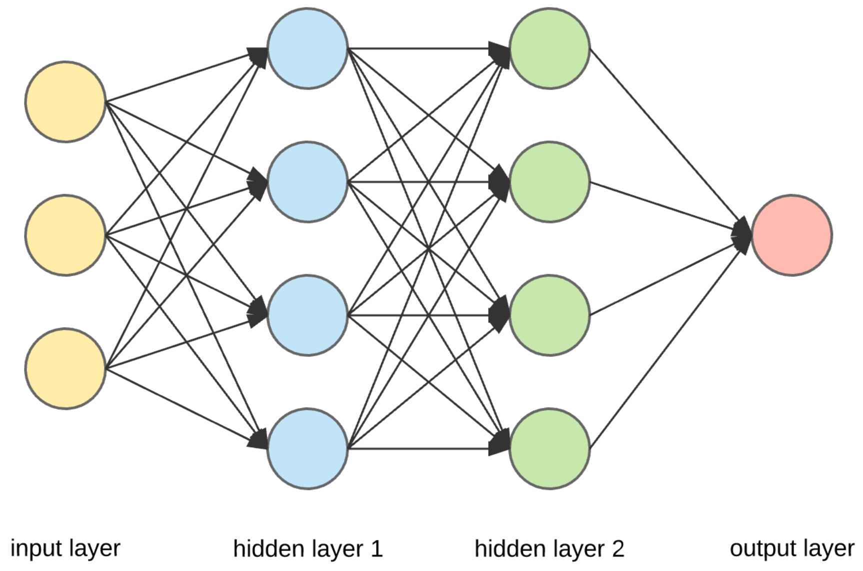
As our threshold for what is considered a positive case increases, we should expect that both the TPR and the FPR increase because some of those positive cases will be true and some will be false. We can intuitively think of moving right on the x-axis as increasing our threshold of what is determined to be a positive case. A completely random classifier is a straight line from (0, 0) to (1, 1) because it will be right and wrong with equal probability. A classifier that is better than random will be concave as shown above, where TPR increases faster than FPR; a perfect classifier would have a TPR of 1 for any threshold.

To measure this, we can use the Area Under the (ROC) Curve (AUC) metric, or the integral of the ROC curve from 0 to 1. For a perfect classifier, as discussed, the AUC would be 1, and for a random classifier, the AUC would be 0.5. This can help compare across different classifier models.

5 Neural Networks

5.1 Graphical Representation and Model Definition

One way to think about neural networks as a big picture is a logistic regression with parameterized, adaptive basis functions, where adaptive means we do not need to initially specify a basis function to our model. This is because neural networks can approximate very complex functions by composing a series of linear transformations with non-linear activation functions.



Looking at the image above, there are a few distinguishing features of the network architecture. First, a neural network is made up of layer of nodes, where the number of nodes in a layer is its width. Nodes are functions that take in a set of inputs from the arrows pointing into a node. Each node then applies a set of learned weights and an non-linear activation function to yield a set of outputs as the arrows pointing out to the next layer. Broadly, we would pass in data into the input layer, which, after a set of linear and non-linear transformations in the hidden layer, would give a resulting value in the output layer.

The flexibility of neural networks comes from the universal function approximation theorem, which states that for any continuous function f defined on a bounded domain I, we can find a neural network that approximates f arbitrarily well. In simpler terms, this means we can use neural networks can express virtually all functions within some bounded range for some level of model complexity. This model complexity comes from the network architecture. Increasing the depth of the network, or the number of layers, can help identify more abstract features with more complex relationships; increasing the width of the network, or the number of nodes in each layer, can help capture smaller, more fine-grained details in features.

Some high-level pros and cons for neural networks: neural networks are very flexible at modeling all kinds of problems and importantly do not require good priors on data or pre-specified feature mapping as would be necessary in basis regression. As a result, even though they tend to be more computationally expensive, neural networks are very widely-used as a machine learning model.

5.2 Inference and Optimization

For the model setup for a feed-forward neural network, denote $\mathbf{W}^{(l)}$ as the weight matrix for the weight from each node in layer l to each node in layer $l+1$, $\mathbf{A}^{(l)}$ the pre-activation values for layer l, and $\mathbf{Z}^{(l)}$ the post-activation values for layer l. That means when passing into layer $l+1$, we use the values $x_1, \dots, x_D$ from layer l; when computing node j in that layer, we use the weights $w_{j1}, \dots, w_{jD}$ that are specific to each node in layer l to node j in layer $l+1$.

Then, for layer $l+1$, at node j in that layer, compute

$$\mathbf{a}_j^{(l+1)} = \sum_{d=1}^{D} w_{jd}^{(l+1)} \mathbf{z}_d^{(l)} + w_{j0}^{(l+1)}$$

to get the pre-activation value at that node by taking the sum-product over all nodes in the previous layer and node weights. From here, transform the pre-activation value to the post-activation value with some activation function h:

$$\mathbf{z}_j^{(l+1)} = h(\mathbf{a}_j^{(l+1)})$$

where common activation functions include ReLU(z) = $\max(0, z)$ and tanh(z). Then, we repeat by passing these in as the x-values into layer $l+2$ until the output layer. Typically, we use an activation function like the sigmoid or softmax function for the last layer if we want to output a set of probabilities for classification.

How do we train these weights? We can use the backpropagation algorithm to compute the gradients for each weight before applying gradient descent. In backpropagation:

1. Randomly initialize weights $\mathbf{W}$
2. Perform forward pass where data is passed through the entire model and the entire network error L is calculated based on the model predictions.
3. Perform backward pass, propagating the error back through the layers to compute the gradients for total network loss on each weight $\frac{\partial L}{\partial w_{jm}^{(l)}}$. Note that because each layer depends on the adjacent layers, we need the chain rule to differentiate a neural network, which makes backpropagation an iterative algorithm.
4. Update weights via gradient descent.

A few notes on neural network optimization: training these networks can run into a number of problems. For one, neural networks are often non-convex, which means gradient descent will not easily find the global optimum. Gradient descent also might run into either divergence or slow convergence, which depend on the learning rate used to descend.

5.3 Model Interpretation

Neural networks are difficult to interpret because they are far more complex than other models, like linear regression. From outside the black box, it is often not clear which weights correspond to which observable features. However, that does not mean that neural networks are entirely uninterpretable; there is a ton of current research on interpretability.

6 Bayesian Regression

The Bayesian view of regression seeks to formulate regression with probability distributions rather than point estimates, treating all unknown quantities as random variables.

6.1 Model Definition

The Bayesian frame of reference helps us answer three types of questions regarding data X and labels Y by using Bayes' rule:

- The **posterior** over models:

  $$p(\theta|X, Y) \propto p(Y|X, \theta)P(\theta|X)$$

  This tells us how likely different values of the model θ are after updating the prior distribution with the observed data.

- The **posterior predictive** for new data:

  $$p(y^*|x^*, X, Y) = \int p(y^*|x^*, \theta)p(\theta|X, Y)d\theta$$

  This tells us how to predict the label of a new data point according to the posterior over models obtained by updating the prior with the observed data.

- The **marginal likelihood** of data:

  $$p(Y|X) = \int p(Y|X, \theta)p(\theta)d\theta$$

  This tells us how likely the data is, marginalizing over possible models θ. In contrast to the likelihoods we've seen before which are computed given a specific setting of weights θ, the marginal likelihood accounts for a distribution over weights θ. The marginal likelihood allows us to compare different priors or even different model classes in terms of how well they fit the data (this is model selection).

6.2 Example

Consider an example with a Bayesian model: Let $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$, $\mathbf{x}_i \in \mathbb{R}^m$, $y_i \in \mathbb{R}$.

$$y_i \sim \mathcal{N}(\mathbf{w}^\top \mathbf{x}_i, \beta^{-1}) \tag{1}$$

The likelihood of the data has the form:

$$p(\mathbf{y}|\mathbf{X}, \mathbf{w}) = \mathcal{N}(\mathbf{y}|\mathbf{X}\mathbf{w}, \beta^{-1}\mathbf{I}) \tag{2}$$

Put a conjugate prior on the weights (assume covariance $\mathbf{S}_0$ known):

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mu_0, \mathbf{S}_0) \tag{3}$$

We want a posterior distribution on $\mathbf{w}$. Using Bayes' Theorem:

$$p(\mathbf{w}|\mathcal{D}) \propto p(\mathcal{D}|\mathbf{w})p(\mathbf{w}) \tag{4}$$

It turns out that our posterior after N examples is also Gaussian:

$$p(\mathbf{w}|\mathcal{D}) = \mathcal{N}(\mathbf{w}|\mu_N, \mathbf{S}_N) \tag{5}$$

where

$$\mathbf{S}_N = \left(\mathbf{S}_0^{-1} + \beta \mathbf{X}^\top \mathbf{X}\right)^{-1} \tag{6}$$

$$\mu_N = \mathbf{S}_N(\mathbf{S}_0^{-1}\mu_0 + \beta \mathbf{X}^\top \mathbf{y}) \tag{7}$$

We have seen how to obtain a posterior distribution over $\mathbf{w}$. But, given this posterior and a new data point $\mathbf{x}^*$, how do we actually make a prediction y^*? How do we deal with *uncertainty* about $\mathbf{w}$? We can expand the predictive distribution over y^* given $\mathbf{x}^*$ using the Law of Total Probability:

$$p(y^*|\mathbf{x}^*, \mathcal{D}) = \int_{\mathbf{w}} p(y^*|\mathbf{x}^*, \mathbf{w}) p(\mathbf{w}|\mathcal{D}) d\mathbf{w} \tag{8}$$

$$= \int_{\mathbf{w}} \mathcal{N}(y^*|\mathbf{w}^\top \mathbf{x}^*, \beta^{-1}) \mathcal{N}(\mathbf{w}|\mu_N, \mathbf{S}_N) d\mathbf{w} \tag{9}$$

This is the posterior predictive distribution over y^*. This can be interpreted as a weighted average of many predictors, one for each choice of $\mathbf{w}$, weighted by how likely $\mathbf{w}$ is according to the posterior. Since each of the terms on the right hand side follows a normal distribution, we can use some math to find that:

$$p(y^*|\mathbf{x}^*, \mathcal{D}) = \mathcal{N}(y^*|\mu_N^\top \mathbf{x}^*, \mathbf{x}^{*\top} \mathbf{S}_N \mathbf{x}^* + \beta^{-1}) \tag{10}$$

6.3 Inference and Model Comparison

How does inference from a Bayesian model compare with a frequentist probabilistic model? While Bayesian models result in posterior distributions, we can also generate point estimates to more directly compare with non-Bayesian models. The two common point estimates are:

- Posterior mean: the "average" estimate of $\mathbf{w}$ under the posterior distribution

$$\mathbf{w}_{\text{post mean}} = E[\mathbf{w} \mid D] = \int_{\mathbf{w}} \mathbf{w} p(\mathbf{w} \mid D) d\mathbf{w}$$

- Maximum a posteriori (posterior mode): the most common estimate of $\mathbf{w}$ under the posterior distribution

$$\mathbf{w}_{\text{MAP}} = \text{argmax}_{\mathbf{w}} p(\mathbf{w} \mid D)$$

The major benefit of these point estimates is that they are more interpretable for making inferential decisions. If we want to estimate the expected value of a quantity, for example, it may be better to use the posterior mean as an estimate, rather than return the entire distribution. The Berstein-von Mises Theorem says that posterior point estimates approach the MLE for large samples sizes, and can often even be expressed as a form of regularized MLE. The downside of Bayesian modeling, however, is that it tends to be more computationally challenging for models that do not nicely follow conjugate pairs.