1678 Midterm

What are the different ways we have defined the goal of function approximation? What is the reason for each method?

- The function may be unknown to us
  - Approximation from a dataset is why DL is in style
- The function may be expensive or impossible to compute
- Compress a big model into a faster model
- Actual figures may be too big

Linear function approximation with a basis function:

- What functions can we represent exactly with linear function approximation? Be able to do this for different choices of f.
  - You can represent all functions with linearly separable features with straight-up linear function approximation. With the addition of a nonlinear basis (e.g. Fourier or polynomial), you can approximate any function (as linear approximators with a nonlinear basis function are dense in any function space).
- What are the different choices of basis functions?
  - Fourier or polynomial basis functions are two examples from class.
- How does adding more or less features change the approximation?
  - Adding more features allows you to approximate the underlying function more closely. If you're training from a finite dataset, though, it's likely that you're just overfitting.
- How do more or less features impact optimization?
  - As the size of the dataset gets arbitrarily bigger, you can approximate a function closer and closer. This takes longer. And you can be fitting to noise or imagining relationships between features that don't otherwise exist.
- Understand the basics of universal function approximation, e.g., let the features grow to infinity and we can get an arbitrarily good approximation.
  - For our purposes, the Universal Approximation Theorem says that all sufficiently-sized nonlinear functions are dense in some space– if we have a big-enough linear approximator with a Fourier or polynomial basis (or a MLP with an infinite-width hidden layer, or infinite layers), then we can approximate any function, in theory.

Objectives

- What are the objectives for regression and classification (mean squared error and negative log-likelihood)?
  - The objective for regression is to predict a numerical label. The objective for classification is to predict a categorical label.
  - Mean squared error is an objective/loss function for regression. It's the arithmetic mean of all squared differences between model label and actual label.
  - Negative log likelihood is an objective/loss function for classification. The likelihood function is the summation of the probabilities that the model outputs the correct label given the input features; the log likelihood is log of that. Thus the negative log likelihood is the negation of all that.
- What are the derivatives of these objectives with respect to the function outputs?
  - "Adjust your model weights so that loss values are smaller." This is what the derivative of both MSE and NLL say.
- Why do we use negative log-likelihood instead of just negative probability?
  - We take the log likelihood because it makes the gradient easier to compute; it transforms a product into a sum, which is easier to take partial derivatives of.

Optimization

- What is gradient descent?
  - Gradient descent is an iterative process to train some model. It's an iterative process wherein you start with some weights, find their loss, take the gradient of loss with respect to those weights according to your training data, and then adjust those weights in some quantity (step size) in the direction that the gradient says will make the new weights have a lower loss.

- What is stochastic gradient descent? Why would we want to use it?
  - SGD is gradient descent, but instead of taking the gradient of loss with respect to those weights according to all data, it's just one randomly-selected point from your data.
- What is the difference between using a batch of data and optimizing for the expectation?
  - This question is ambiguous, so I'm doing my best here. A batch of data is training the NN on the entire dataset. Minibatch is splitting the data up into some splits, then doing SGD on each of those splits.
  - Optimizing for expectation is trying to optimize for an infinite dataset's minimized loss. This is obviously impossible, so we just do the basic statistical principle of sampling subsets and just doing our best to make sure that our samples accurately capture information about our population. It's important to shuffle our dataset, because data collection often is ordered and we need samples to be independent and identically distributed.
- When should we use least squares versus a gradient descent method?
  - Least squares provides a closed-form solution to minimize loss. This can be computationally intractable. So we should use least squares when we have the computational resources to do so, and use gradient descent when we don't.
- What does the step size control in gradient descent?
  - The step size in gradient descent controls how much you adjust your weights by (according to that gradient) from iteration to iteration.
- How should the step size be set in linear function approximation?
  - The step size should be set large enough that the weights converge in a reasonable amount of time ("reasonable" is dictated by circumstances") but small enough that the weights don't step past the minimum of that loss function's topography that we're trying to find.
- How does a step size impact optimization with stochastic gradient descent?
  - Having too small of a step size can result in you still taking forever to converge. Having too large of a step size can result in you weighting a misleading point's gradient too much and stepping further away from the minimum.
- What is maximization bias in relationship gradient descent on a dataset?
  - Bias is the difference in the EV of an estimator's output and the EV of what we're trying to estimate. Maximization bias is when we're trying to maximize some objective function (minimize a loss function. Primal/dual. Don't worry about it) and that introduces bias, for some reason or another.
  - In the context of gradient descent, it happens when you minimize the loss function on the training set to the point where it generalizes poorly (to the test set), making it a form of overfitting. It's mitigated by stopping the learning process when that happens.
- How does overfitting occur in producing a model?
  - Overfitting occurs when you train a model to fit to relationships between features too closely– the model is finding relationships that don't actually exist.
- How does overfitting occur as people try to create new and better models on existing datasets?
  - Overfitting occurs as we keep optimizing on existing models on the same dataset to the degree that we can tune hyperparameters better and better on the same dataset (CV aside) that it generalizes poorly to new points in that space.
- What is cross validation?
  - Cross-validation is splitting data into different folds, so that you evaluate hyperparameters independently on different parts of the dataset, taking the average of these values, such that it's not the case that hyperparameters are good on the training set but poor on the test set.
- How does cross validation reduce overfitting?
  - It prevents you from tuning hyperparameters so that they're good on the training set but perform poorly on the test set (and generalize poorly).
- What is early stopping and why does it help with gradient descent?
  - Early stopping is stopping the learning process when you fit to the training set better but fit to the validation set worse. We use that to simulate overfitting. We don't want to overfit.
- What happens as we increase or decrease the train and testing datasets sizes?
  - A reasonable proportion is to have train:test as 2:1. If you make training bigger, you increase the risk of your model being shitty because it overfit, and if you make training smaller, you increase the risk of your model being shitty because it couldn't learn enough.

Neural Networks

- Why do we want neural networks if we can just use more basis functions?
  - Neural networks are computationally efficient. The universal approximation theorem basically says that any nonlinear approximator can approximate any function. A NN with a nonlinear activation is a nonlinear approximator. Especially with ReLU, you can train the approximator sooooo cheaply that you can blow its size the fuck up and do crazy shit like train fucking ChatGPT. You couldn't do that with a linear model with a polynomial basis. Jeff Bezos would personally break your kneecaps.
- Neural networks and the universal function approximation (network with infinite width and single hidden layer can approximate any function). Can we find this function even if we can represent it?
  - Not necessarily. The UAT says that nonlinear functions (neural networks, in this case) are dense in an output space. It doesn't say that any training method (including backpropagation) will definitely get you arbitrarily close to what you're trying to approximate.
- How does increasing width or depth change what the neural network can represent? What benefits does the compositional structure of neural networks provide?
  - Increasing width or depth allows the NN to learn more relationships between features. It's the case that a MLP with an infinite-width hidden layer is a universal approximator. The compositional structure of NNs allows more effective training with backpropagation.
- What are the different parts of a neural network (MLP)?
  - The mighty multi-layer perceptron has an input vector, one or more hidden layers, and a last layer of one or more output units. Each unit (including the input vector's entries) has connections to each unit in the following layer. These are weights (and sometimes biases). We can model each layer as a vector and each set of connections from layer to layer as a matrix (because a matrix is a transformation on a vector). Each unit is the sum of the `wx+b` of all units feeding into it, fed into some activation function that introduces nonlinearity.
- What are the different choices of activation functions and why would we want to choose one over another?
  - Some choices of activation functions are sigmoid, hyperbolic tangent, and rectified linear unit. Sigmoid and hyperbolic tangent both cause vanishing gradients, and they're harder to compute derivatives of. ReLU doesn't cause vanishing gradients, and it's really easy to compute derivatives of. And I felt like a pro neural network guy when I started using it. That's best.
- How do we initialize neural networks and why?
  - Xavier initialization is a good choice for NNs. We don't want vanishing or exploding gradients. Xavier initialization mitigates those. It's a uniform random distribution according to the number of neurons in the previous and next layers.
- What are exploding and vanishing gradients? What causes them?
  - Recall that backpropagation is just taking the derivative of a neural network backwards through each layer. This is a big chain rule. (See the next answer for more details.) Anyway, this product can quickly approach 0, in which case it's called a vanishing gradient, or get huge, in which case it's called an exploding gradient. Both are problematic for training a neural network.
  - Improper initializations, too-small or too-large, can cause vanishing or exploding gradients, respectively. It's important to use proper initialization schemes (Xavier). Tanh and sigmoid as activation functions can also cause vanishing gradients. ReLU does not cause vanishing gradients but can result in dead neurons (as its gradient is just 0 for $x < 0$).
- How does backprop compute the derivatives of the network?
  - Backprop is just the chain rule applied backwards through the neural network. The derivative of loss with respect to all weights is the derivative of loss with respect to the output layer's weights, times the derivative of the output layer's weights with respect to the ultimate hidden layer's weights, times the derivative of the ultimate hidden layer's weights with respect to the penultimate hidden layer's weights... This all repeats until we've found the derivative of the first hidden layer's weights. These are cached in memory, making this an instance of memoization and thus a form of dynamic programming.
- How does the choice of output activation and loss function matter in training the neural network?
  - This is clearly important: in regression, a sigmoid or tan activation function would be disastrous, as it would distort (squash and transform) results for no reason, and in binary classification, a linear activation could make a result a value close to 0.5, which, with the goal of ~0 or ~1, is not interpretable.
  - The same line of reasoning holds for loss function. Using MSE for binary classification would make no sense– it would say severe errors (0 for 1, or vice versa) aren't severe. This could lead to poor training times, and model performance.

  - (This all holds for multiple classification, using softmax as a generalization of logistic/sigmoid activation functions, and categorical cross-entropy as a generalization of NLL).
- What are the derivatives of different activation functions, and do they impact the gradients of the whole network, training?
  - I'll touch on three activation functions: ReLU, tanh, and logistic function– henceforth expit.
    - The derivative of ReLU is 1 if the function is nonnegative, and 0 otherwise.
    - The derivative of tanh(x) is 1 - (tanh(x))ˆ2.
    - The derivative of expit is [expit(x) * (1 - expit(x))].
  - The "squashing" that tanh and expit do can lead to vanishing gradients– the gradients get smaller as backpropagation propagates backwards (lol) through the network, until they say just about nothing meaningful.
  - This obviously makes training take damn forever, waiting for the change in loss from iteration to iteration to converge. If it ever does.
- Why do we want adaptive step sizes for neural networks?
  - The basic idea behind why we want adaptive step sizes for neural networks is that if we're far off, we want to take big steps, so we don't take forever to train, and if we're close to our minimum, we want to take small steps, so we don't step past it.
  - These also scale– the gradient for a deep layer might need a smaller update than a shallower one, because some activation functions "squash" gradients (I talked about expit and tanh doing that earlier).
- What problem do adaptive step sizes like Adam and RMSprop fix? Know the mathematical form of the update rules for exam.
  - I mentioned why we might want adaptive step sizes above.
  - RMSprop divides the step size by the moving average of squared gradients– if gradients are large, the step size reduces, and if gradients are small, it increases.
    - gradient_moving_average = decay_rate * old_gradient_moving_average + (1 - decay_rate) gradientˆ2
    - new_weights = old_weights - [eta * gradient] / [sqrt(gradient_moving_average) + epsilon]
      - epsilon is added to prevent division by zero
  - Adam also keeps a moving average of past gradients' means, but they're exponentially decaying, such that old gradients aren't anywhere near as important as more recent ones, as well as a moving average of past gradients' variance.
    - I won't put the math equation here but basically eta is multiplied propto momentum (mean) and divided propto the variance.
- What do the derivatives for the weights and inputs for a hidden layer look like?
  - ...with respect to what?
  - If the question means *what do the derivatives of loss/output with respect to the weights and inputs for a hidden layer look like*, then the question makes more sense.
    - The derivative of loss with respect to some hidden layer's weights is "how does loss change as hidden layer H_i's weights do?" The answer is the derivative of the activation function, multiplied by the activation of H_{i-1}.
    - The derivative of loss with respect to some hidden layer's inputs is "how does loss change as hidden layer H_i's weights do? The answer is the derivative of the activation function, multiplied by the weights of the connections between H_i and H_{i-1}.