CS 182/282A: Designing/Visualizing and Understanding Deep Neural Networks

Lecture 2: Machine Learning Review

23 January 2023

Lecturer: Professor Anant Sahai *Scribe: Arm Wonghirundacha*

We will cover the standard optimization-based paradigm for supervised learning.

1 The ingredients

The primary ingredients in a standard optimization-based supervised learning problem are:

- Data: (x_i, y_i) pairs where x_i is the input/covariates and y_i is label/output and the index $i = 1, \ldots, n$ where n is the size of the training data.
- Model: $f_\theta(\cdot)$ for parameters θ
- Loss Function: $\ell(y_i, f_\theta(x_i))$ which returns a real number
- Optimization Algorithm

We will now expand on these ingredients and highlight complications which arise along with the standard solutions to address them.

2 The model

Training the model means choosing the parameters θ. We do so by using empirical risk minimization. One example of this is to choose θ which minimizes average loss:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell_{\text{train}}(y_i, f_\theta(x_i)) \tag{1}$$

For our setup, model performance is based on the average loss evaluated on our data, however this is not reflective of our true goal, which is to ensure the model performs well when deployed into the real world since real world data could look different to our training data. Because of this, we must have proxies to help us get a better understanding of the true performance of our model.

A mathematical proxy is given when we make an assumption about the probability distribution $P(X, Y)$ of the underlying data, that is, we assume what the data in the real world would look like. With this distribution assumption, we can evaluate the expected loss: $E_{X,Y}\left[\ell(Y, f_\theta(X)\right]$ which we want to be as low as possible. Making such an assumption about

the underlying distribution causes a few complications.

Complication 1: We have no access to $P(X, Y)$, any probability distribution we would use comes with many strong assumptions about the real world.

Solution: A standard way to tackle such a complication is to partition the initial data we collected into a train and test set which we define as $(x_{\text{test},i}, y_{\text{test},i})_{i=1}^{n_{\text{test}}}$ pairs. We withhold the test data during the model training and then use the test data to observe the test error, defined as:

$$\frac{1}{n_{\text{test}}} \sum_{i=1}^{n_{\text{test}}} \ell_{\text{true}} \left(y_{\text{test},i}, f_{\hat{\theta}}(x_{\text{test},i}) \right) \tag{2}$$

Test error should be a faithful representation of real world, so we should refrain from using test data until our model training is complete, otherwise we would get caught in a feedback loop and have to deal with additional complications of model overfitting.

3 Loss Functions

A loss function maps a set of values to a real number which in theory should reflect some sort of cost associated with event we are trying to model. There are many different loss functions which we may have seen in previous coursework including: for binary classifiers, hinge loss or logistic loss and for multi-class classifiers cross-entropy loss. With many different loss functions in mind, we must decide on which loss function is the best for our use case. This loss function decision comes with some complications.

Complication 2: Our loss function $\ell_{\text{true}}(\cdot, \cdot)$ that we actually care about is incompatible with our optimizer. e.g. Our loss function is non differentiable but the optimizer requires its derivatives.

Solution: Use a surrogate loss, $\ell_{\text{train}}(\cdot, \cdot)$, that satisfies the conditions of the optimizer to train our model, but still evaluate the performance of our model, that is, calculate test error, with $\ell_{\text{true}}(\cdot, \cdot)$ since we no longer have to deal with the constraint of the optimizer.
e.g. $y \in \{\text{cat, dog}\}$ where ℓ_{true} : Hamming Loss, but we can't take derivatives of cat and dog, so we use a surrogate loss: $y \to \mathbb{R}$ where the training data becomes cat $\to -1$ and dog $\to +1$, which is now differentiable when we choose ℓ_{train} : squared error loss.

A side note: $\frac{1}{n} \sum_{i=1}^{n} \ell_{\text{train}} \left(y_i, f_{\hat{\theta}}(x_i) \right)$ and $\frac{1}{n} \sum_{i=1}^{n} \ell_{\text{true}} \left(y_i, f_{\hat{\theta}}(x_i) \right)$ are two different quantities since we are finding the error using different loss functions. You might ask, why do we need to find the training error using ℓ_{true}, when we can just evaluate the model using test data instead? We use this as a debugging method since evaluating the training error using the true loss function helps us understand if the surrogate loss is doing an adequate job in training our model.

4 Overfitting and Hyperparameters

4.1 Overfitting

Complication 3: We get 'crazy' values for $\hat{\theta}$ and/or we get really bad test performance. One reason this could happen is due to model overfitting.

Solution: Add an explicit regularizer during training, that is,

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell_{\text{train}}(y_i, f_\theta(x_i)) + R(\theta) \tag{3}$$

where $R(\theta) = \lambda \|\theta\|^2$ is an example of a regularizer we could choose and is known as ridge regularization.

Side note: An alternative solution is to simplify your model or reduce the model order (e.g. the depth of the model). The suggestion to simplify models is often suggested in statistics, however it is uncommon in deep learning to simplify models when crazy values of $\hat{\theta}$ arise from model training.

4.2 Hyperparameter tuning

By adding a regularization term, we notice the addition of a parameter λ, which raises the question, how do we choose λ?

The Naive approach: $\hat{\theta} = \underset{\theta,\lambda}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f_\theta(x_i)) + \lambda \|\theta\|^2$ but this doesn't work well because we can optimise this equation by assigning an absurd value to λ (e.g. 0 or -inf) This is where we let λ be a hyperparameter, where a hyperparameter can be described as a parameter 'that if you let the optimizer deal with, it will go crazy' and let θ be treated as a normal parameter.

Solution: Separate the parameters from the hyperparameters. Then withhold some data from our training set to create a validation set, which we can use specifically to optimise hyperparameters. An example of how to partition the data is given in Figure 1. We can optimize hyperparameters using methods such as gradient descent or by performing a brute-force grid search.

Further Complication: The optimizer could also contain hyperparameters such as the learning rate or step size η in gradient descent.

5 Optimization Algorithm

There are many different optimization algorithms when it comes to minimizing a loss function. A commonly used optimization algorithm is **Gradient Descent**. Gradient descent is an iterative optimization algorithm which changes the parameter of interest θ a little bit at a time by looking at the local neighborhood of loss around θ_t. We look at this neighborhood by taking the first order Taylor expansion: $L_{\text{train}}(\theta_t + \Delta\theta) \approx L_{\text{train}}(\theta_t) + \frac{\partial}{\partial\theta} L_{\text{train}}|_{\theta_t} \Delta\theta$. We

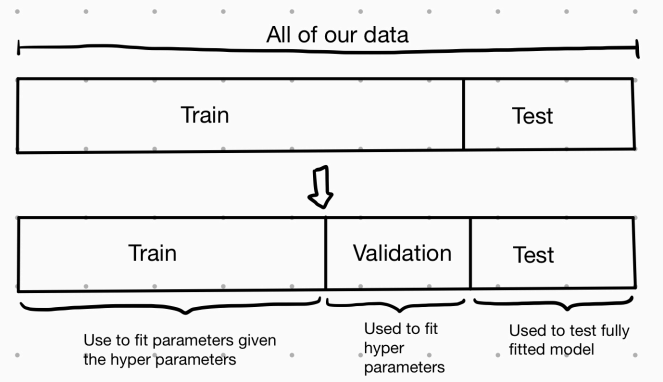

Figure 1: Data partitions for fitting parameters and hyperparameters

want to move in such a way that maximizes the change of the loss ($L_{\text{train}}(\theta_t + \Delta\theta)$) so we must match it. That is, move in the negative direction of the gradient. So gradient descent updates θ by the following:

$$\theta_{t+1} = \theta_t + \eta\left(-\nabla_\theta L_{\text{train},\theta}\right) \tag{4}$$

where η is the learning rate, and an example of $L_{\text{train},\theta} = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f_\theta(x_i)) + R(\theta)$. So we see that for the update, we are taking small steps in the opposite direction of the gradient to reach the minima.

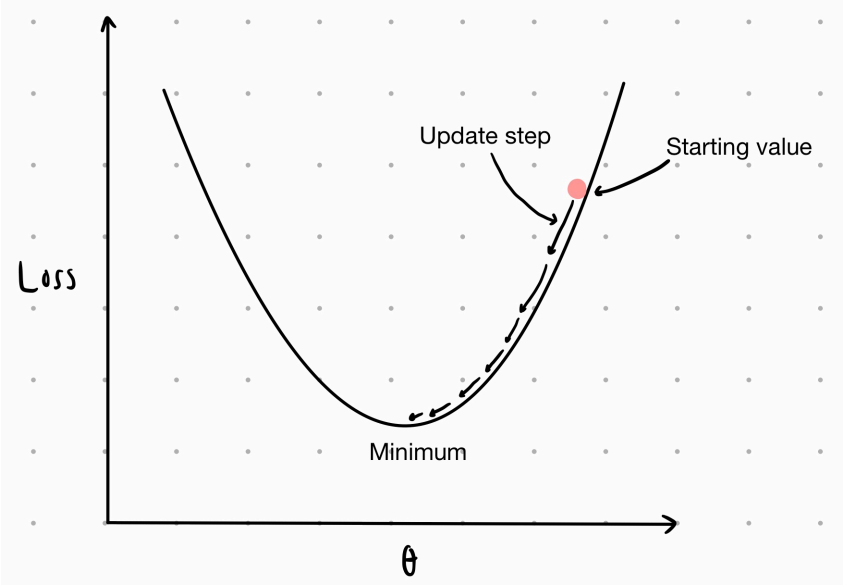

Figure 2: Example of the iterative gradient descent update

We can interpret Eq. 4 as a dynamical system in discrete time, which means we can ask

questions about the stability of the system. In our case, η controls the stability of our system, where if η is too large, the dynamics become unstable (we could possibly diverge), but if η is too small, practically speaking, it could take too long to reach the minima.

Time Consideration: There are different interpretations for time when training models. On one hand we could look at how many training iterations have passed, or how much data we have ingested (e.g. Epochs). Another important time consideration is Wall Clock time, which is simply the amount of real world time being used in training. When training our model, both interpretations of time must be used since we want to train our model on enough iterations, while not using too much wall clock time (depending on our budget).