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

Fall 2022

Lecture 2: August 30 (Tuesday)

Lecturer: Prof. Anant Sahai

Scribes: Connie Huang, Jaewoong Lee

Today

1. Recap. Basic Standard ML Doctrine
2. Empirical Risk Minimization (ERM)-Optimization Perspective (e.g. Generalization)
3. Hyperparameters vs. Parameters
4. Gradient Descent & SGD
5. Intro. to Neural Nets via ReLU Nets

1 Recap. Basic Standard ML Doctrine

1.1 Typical Supervised ML Setup

- Training Data: x_i, y_i, where x_i is input (or covariants), y_i is label, and i = 1, 2, ..., n
- Model: $f_\theta(-)$
- Loss Function: $l(y, \hat{y})$
- Optimizer

Our goal of a supervised ML setup is to make an inference $\hat{y}$ on new data X as follows: $\hat{y} = f_{\hat{\theta}}(X)$, where $\hat{\theta}$ are the learned parameters.

2 Empirical Risk Minimization (ERM)-Optimization Perspective

How do we learn parameters θ? The basic approach is to find the optimal θ for our optimization problem,

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} l(y_i, f_\theta(x_i)) \tag{2.1}$$

We can then extend this to the probabilistic interpretation, maximum likelihood (ML) estimation, where our loss function $l(y, \hat{y})$ is interpreted as the negative log-likelihood function.

The big picture goal for supervised machine learning is to achieve good performance in the real world when the model is deployed. In practice, this is difficult to achieve because in the real world, there are unexpected circumstances that we do not have data for and therefore cannot actually represent in our model. As a result, we must use a mathematical proxy so that our model has a

low generalization error. We can model the real world using a probability distribution $P(X, y)$ and aim to minimize the expectation of our loss function with respect to this probability function:

$$E_{X,y}[l(y, f_{\hat{\theta}}(X))]$$

However, our mathematical proxy introduces a few complications.

Complication 1: We do not have access to $P(X, y)$.

Solution: collect a test set $(x_{i,test}, y_{i,test})_{i=1}^{n_{test}}$ to be used once to evaluate our learned model by getting test error.

$$\frac{1}{n_{test}} \sum_{i=1}^{n_{test}} l(y_{i,test}, f_{\hat{\theta}}(x_{i,test}))$$

The model is desired to be tested once because it is not only hard to collect test data but also there is a risk of data incest of test data while designing the model. Test data are not supposed to affect the model.

Complication 2: The loss we care about may be incompatible with our optimizer. For example, our optimizer will use derivatives to find optimal parameters, but our loss function may not have nice derivatives everywhere.

Solution: Use a surrogate loss function $l_{train}(.,.)$ that does have nice derivatives, computes fast, and works with the optimizer. We use this surrogate loss function to *guide* learning of the model. The real loss function is used to evaluate the model. Some standard loss functions include squared error (regression); logistic, hinge, and exponential loss (binary classification); and cross-entropy loss (multiclass classification). You may want to choose a loss function based on the application settings of the problem and model.

This surrogate loss function is different from the evaluation loss function from *complication 1.* The surrogate loss function is used for training the model, and the evaluation loss function is to see how well your model works with new test data. A few things to remember for choosing an appropriate surrogate loss function are *it should be compatible with the optimizer, guide the model to the correct solutions, and run fast enough (e.g. easy to take derivatives).* The squared loss ($l_{train} = (y_i - \hat{y}_i)^2$ or in the vector form, $l_{train} = ||y - \hat{y}||^2$) is a good example of running fast enough.

3 Hyper-parameters & Parameters

Complication 3: We might get *crazy* values for $\hat{\theta}$ (e.g. *over-fitting*). How do we solve this problem?

Solution A: Add a *regularizer* during training.

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \left[\frac{1}{n} \sum_{i=1}^{n} l_{train}(y_i, f_\theta(x_i)) + R(\theta) \right] \tag{3.1}$$

In the above equation, $R(\theta)$ is the regularizer that can be chosen based on what *loss function* is used. For example, if squared loss is used as the loss function, then the *ridge regularization* ($R(\theta) = \lambda||\theta||^2$) might be used as the corresponding *regularizer*. The *ridge regularization* prevents the $\hat{\theta}$ values from becoming too big. The probability interpretation of *regularization* is **Maximum**

A Posteriori (MAP) estimation where $R(\theta)$ corresponds to a prior, which means we want to achieve optimal thetas, given $R(\theta)$, that find a good balance between the unpenalized loss function and $R(\theta)$. Now, realize that we added a new parameter λ to the regularizer. How do we handle λ?

Solution B: Split parameters into two groups: The normal parameters (θ) and hyperparameters (θ_H).

A hyperparameter is a parameter that cannot be trained or the optimizer cannot deal with. For example, if λ was considered as a normal parameter in the above *ridge regularization* example, then λ would end up with an absurd number being assigned (e.g. *0* and *-inf*). Another example of a hyperparameter is the model order (degree) of a polynomial function $f_\theta(x_i)$.

How does the optimizer work with hyperparameters?

Figure 3.1 shows the classical division of data into three categories for hyperparameter fitting. The process of parameters and hyperparameters fitting can be represented as a nested optimization problem with the equations below. Notice that equation (3.3) is equal to equation (3.1) except $R_{\theta_H}(\theta)$.

$$\hat{\theta}_H = \underset{\theta_H}{\operatorname{argmin}} \frac{1}{n_{val}} \sum_{i=1}^{n_{val}} l_{val}(y_{i,val}, f_{\tilde{\theta},\theta_H}(x_{i,val})) \tag{3.2}$$

$$\tilde{\theta} = \underset{\theta}{\operatorname{argmin}} \left[\frac{1}{n} \sum_{i=1}^{n} l_{train}(y_i, f_\theta(x_i)) + R_{\theta_H}(\theta) \right] \tag{3.3}$$

Process

1. Initiate the values of hyperparameters (θ_H)
2. Based on the values of hyperparameters (θ_H), compute the regularized loss with $R_{\theta_H}(\theta)$ on the training data set to get $\tilde{\theta}$ in equation (3.3)
3. Based on the values of normal parameters ($\tilde{\theta}$) and hyperparameters (θ_H), find the best $\hat{\theta}_H$ on the validation data set using equation (3.2)

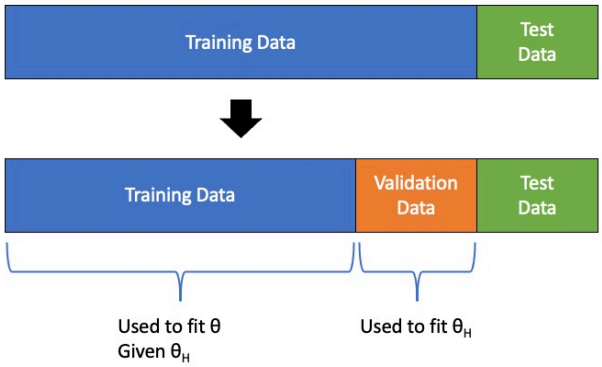

Figure 3.1: Partitioning data for hyperparameter Tuning

You may split the original training data set into the new training and validation data set. However, be careful about data contamination. (e.g. Duplicated data points in each data set. The training and validation data set should be distinct)

Complication 4: The optimizer might have "knobs" (other parameters) associated with it. This might include, for example, the learning rate (or step size) η in gradient descent.

Solution: Include these in θ_H or ignore this problem (i.e. pick a value that has worked in the past. This is a reasonable approach in the light of the limit of the experimentation budget).

4 Gradient Descent and SGD

Gradient Descent (GD) is an iterative approach to optimization (with the spirit of Newton's method) that seeks the local optima taking repeated steps in the opposite direction of the gradient around the current point. Also, Gradient Descent operates under the assumption that it's a linear system. What does the linear assumption mean? The basic idea is to look at the loss function $L_\theta = \frac{1}{n}\sum_{i=1}^{n} l_{train}(y_i, f_\theta(X_i)) + R(\theta, \theta_H)$ in the neighborhood of θ_0 in any place using Taylor Expansion.

$$L_\theta(\theta_0 + \Delta\theta) \approx L_\theta(\theta_0) + (\boldsymbol{\nabla}_\theta L_\theta(\theta_0))^T \Delta\theta$$

Here, θ_0, $\Delta\theta$, and $(\boldsymbol{\nabla}_\theta L_\theta(\theta_0))$ are vectors, and $L_\theta(\theta_0)$ is a scalar. From the equation above, $(\boldsymbol{\nabla}_\theta L_\theta(\theta_0))$ is the gradient around θ_0.

Using this approximation, we can iterate to find our optimal θ.

$$\hat{\theta}_{t+1} = \theta_t + \eta(-\boldsymbol{\nabla}_\theta L_\theta(\theta_t))$$

Notice that the gradient $((\boldsymbol{\nabla}_\theta L_\theta(\theta_t))$, multiplied by a scalar factor (η), at the current time step t is subtracted (taking a negative step) from θ_t. η is the learning rate, which we set to be small enough so that the system converges and big enough so that optimization is not too slow. One problem we introduce with this method is that computing gradients for extremely large datasets can be very computationally intensive. As a result, we introduce **Stochastic Gradient Descent (SGD)**, where instead of using the entire dataset of size n, we randomly choose a representative subset of size n_{batch} from n every iteration to reduce the computation of gradients to only this batch. Because we randomly choose a subset of size n_{batch} every iteration, the overall result over time is a good estimate and trustful.

5 Intro. to Neural Nets via ReLU (Rectified Linear Unit) Nets

5.1 What is a Neural Net (Differentiable Programming)?

A neural net is an object that is easy to take derivatives (e.g. Analog circuits realized as computation graphs with (mostly) differentiable operations compatible with nice vectorization). Moreover, differentiable operations allow nonlinearities.

5.2 Two goals of the analog circuits

1. **Expressivity**: Use the circuit to express the patterns that we want to learn. In other words, the circuit is realization of the function $f_\theta(-)$, and the θs are tunable resistors in the circuit.

2. **Reliably Learnable**: Think of your machine learning system as a microscope where you look at data and the right patterns come into focus.

5.3 Example of Neural Networks

Figure 5.1 shows a 1-D nonlinear (Blue) function, piecewise linear (Red) functions, and data points (Black). As shown in the figure, the piecewise functions describe the nonlinear function pretty well. Our goal is to find a set of piecewise linear functions (Red) that best match the nonlinear function (Blue) based on the data points using Neural Nets. Then, how do we create the piecewise linear functions? A linear combination of elbows (Rectified Linear Units) in **Figure 5.2**! The rectifier circuit in **Figure 5.2** is composed of a diode and a resistor. The diode prevents the current from flowing in the opposite (or negative) direction. Setting the positive direction of the current to be from left to right, it means that the current can never flow in the negative direction (from right to left). All negative currents will be set to zero resulting in V_{out} readings being zero. On the other hand, positive currents (from left to right) will go through the diode resulting in V_{out} readings on the other side of the diode. The standard ReLU function is shown below.

$$f(x) = max(0, x) = \begin{cases} 0 & \text{if x} \leq 0 \\ x & \text{if x} > 0 \end{cases}$$

In this example, V_{in} is x and V_{out} is $f(x)$. Also, the standard ReLU function can be modified if needed. For example, x can be replaced with $wx + b$, so that the modified ReLU function becomes $f(x) = max(0, wx + b)$. Here, w and b are the parameters(θ) we want to minimize using a loss function as mentioned in the previous sections. More details and visualization will be covered in the discussion session and next lecture.

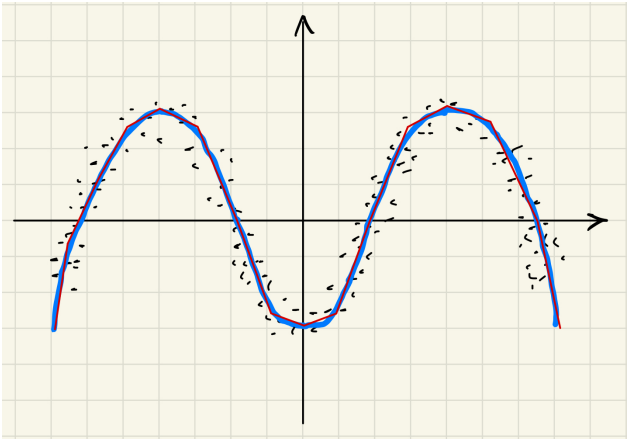

Figure 5.1: 1-D nonlinear and Piecewise linear functions

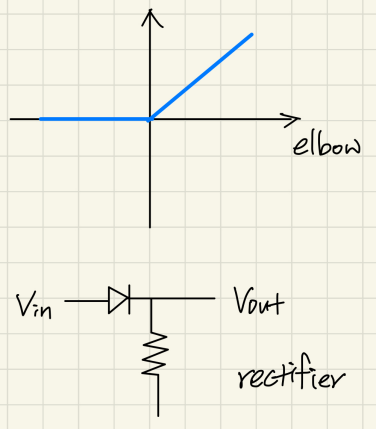

Figure 5.2: Elbow and Rectifier

6 What we wish this lecture also had to make things clearer?

- It would be helpful if more Empirical Risk Minimization (ERM) is covered in this lecture more directly and potentially with diagrams.