**Deep Neural Networks** **(UC Berkeley CS182/282A, Spring 2023)**

# Lecture 3: Optimizers and Neural Networks

27 January 2023

*Lecturer: Anant Sahai* *Scribe: Mathias Weiden*

## 1 Optimizers

A model's parameters are solved for using an optimizer, which itself usually has its own hyperparameters (learning rate, choice of optimizer, etc.). The most basic optimizer is gradient descent, an iterative local optimizer. The idea is to change parameters by a little bit each time step, and see how that changes the model's performance. If we have some parameter vector $\theta \in \mathbb{R}^n$, we update it according to:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L_{train,\theta}$$

where $\theta_t$ denotes the parameter settings at time step $t$, $\eta$ is the learning rate hyperparameter, and $L_{train,\theta}$ is the training loss function. At each iteration, we take a small step in the direction opposite the steepest ascent of the training loss function. If we look at a first order Taylor series approximation of the training loss function,

$$L_{train}(\theta_t + \Delta\theta) \approx L_{train}(\theta_t) + \frac{\partial}{\partial\theta} L_{train}(\theta)\Big|_{\theta_t} \Delta\theta$$

it is clear that we are looking at the training loss function locally, and asking the question, "where will we be if we take a small step in this direction?" This lets us gain an understanding of how the loss landscape depends on our parameters.

The updated parameters $\theta_{t+1}$ are a discrete time dynamic system, and the learning rate $\eta$ controls the system stability. If $\eta$ is too large, we have instability and our model's loss will diverge. If $\eta$ is too small, we make (almost) no progress towards getting a working model. The badness of choosing a large learning rate $\eta$ is very clear, if our loss function diverges, our model doesn't learn anything. Although choosing a small learning rate does not cause divergence, it is an equally bad scenario as it corresponds to more real world wall clock time, energy usage, and cost.

If we look at our training loss function more closely,

$$L_{train,\theta} = \frac{1}{n} \sum_{i=1}^{n} l_{train}(y_i, f_\theta(x_i)) + R(\cdot)$$

we notice that it includes a huge summation over the entire dataset. Remember here that $l_{train}(y_i, f_\theta(x_i))$ is a surrogate loss function that is compatible with our choice of optimizer, and $R(\cdot)$ is a regularization term that ensures our parameters $\theta$ satisfy some predetermined

constraints (like sparsity). The size of the dataset $n$ is typically huge, so evaluating $L_{train,\theta}$ can be extremely wall clock time consuming. To address this, stochastic gradient descent (SGD) is typically used.

In SGD, instead of all $n$ training points, $n_{batch} << n$ random samples are used. Our loss function now becomes

$$L_{train,\theta} = \frac{1}{n_{batch}} \sum_{i=1}^{n_{batch}} l_{train}(y_i, f_\theta(x_i)) + R(\cdot)$$

and $\{(x_i, y_i)\}_{i=1}^{n_{batch}}$ are randomly drawn pairs of training samples. The quantity we want is an average. If we take a random draw of a subset of points, we get an estimate of the average. We can assume that this gives us an unbiased estimate of the actual loss function across the entire dataset, albeit one with some amount of variability (because $n_{batch} << n$). In a deep learning context, the batch size $n_{batch}$ is the largest that your system can sustain, and ensure adequate wall clock time.

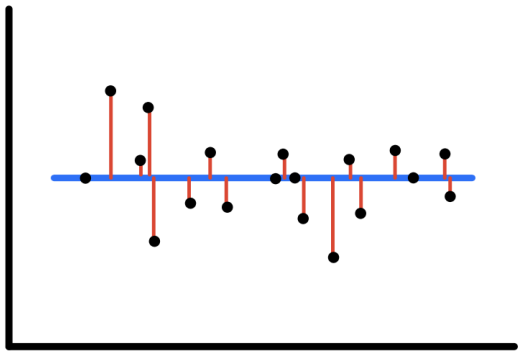

Figure 1: Contributions from each data point in the dataset balance, so the gradient is zero.

Why do we use SGD? The standard *Deep Learning* answer is that we can't afford to look at all the data points, but we can afford to look at a few of them. Beyond that, we can expect that the loss surface has different regions of varying "quality" (flat/non-flat, near global optima, etc.). We have no expectation that we'll begin training in a "good" region, so going in the right general (but not exact) direction might help in this scenario. It's also very probable that our estimate based off a few points very closely aligns with the direction of the gradient across all data points. We're only taking a small $\eta$-sized step in the direction opposite the gradient, so we only need a level of precision that ensures we take our step appropriately.

Changing the step size $\eta$ might help us (or not) depending on the loss surface. We can use a learning rate schedule to enforce some step-size-changing policy. Doing this is especially useful in some non-vanilla niches of machine learning, such as online learning (where data is only made available to you in a sequential order).

A step size selection that causes convergence for regular Gradient Descent might not guarantee convergence for SGD. If we pick $\eta$ so that the gradient doesn't diverge when taken across *all* points, we can generally expect that it won't cause divergence when evaluated across *some* points. What actually needs to happen, is that the model needs to converge to some setting that works well. Convergence happens when the parameters stop changing in our gradient descent iterations. Once we settle so that $\theta_{t+1} \approx \theta_t$, we converge unless the gradient term $-\eta \nabla_\theta L_{train,\theta}$ suddenly spikes. At optima, derivatives/gradients should all be zero, meaning we converge. The derivative is zero because the sum of contributions to the gradient at all the different points cancels out to zero (see Figure 1). In SGD, we're not using all the data points, so we might be at some optimum, but our small-batch estimate of the gradient will likely be unbalanced (see Figure 2). So we might step away from an optimum and "only sort of converge." This wiggling behavior is illustrated in Figure 3.

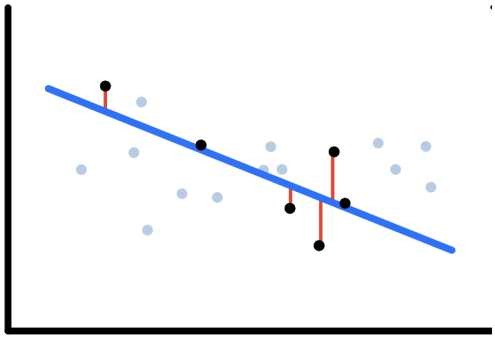

Figure 2: Contributions from this subset of data points are unbalanced, so the gradient found by SGD is nonzero even though it should balance.

So to actually converge, we may need to make sure that $\eta$ is decaying.

Figure 3: Without a decaying learning rate, we may just dance around the optimum.

There are other explanations for why learning rate ought to decrease over time. Picking a large step size at the beginning may help explore the space in a "path less traveled" sort of way, while a small step size at the end helps refine our evaluation.

It's also worth noting that there's technically a difference between oscillatory and random walk type behavior. A non-decreasing step size in gradient descent might cause oscillatory behavior, but the stochasticity in SGD will cause random walk behavior.

Nonetheless, in deep learning, even a constant step size may lead to convergence.

The time it takes to go through an entire shuffled instance of all the training data set is called an epoch. Shuffling (and reshuffling) data between epochs can help avoid us learning subtle patterns that arise because of the ordering of our dataset. Sometimes, this sort of structured approach and commitment to see all data points is not desired. Because datasets can be so large, if we just randomly sample from the training dataset we are likely to get similar results. In these cases, sampling with and without replacement are equally performant.

Generally, getting an optimizer to work well involves lots of consideration to be taken as to the application and the underlying hardware that the model runs on.

# 2 Neural Networks

A neural net is a computation graph/analog circuit that plays nice with automatic differentiation or "back propagation." Neural nets are prevalent today because they're able to exploit gradient descent/SGD to do learning/computation.

What do we want from these circuits? They should exhibit:

1. Expressivity: the ability to represent a wide variety of functions.
2. Learnability: the ability to learn functions without too much trouble.

We want expressivity in our circuits because we need to be able to express patterns we're interested in. Usually we don't know the patterns we're interested in; they need to be learned. We don't have any guarantee that some model is expressive enough to capture these patterns. Traditionally, the solution to this is to be expressive enough to capture any pattern. Surely then, if we can capture any pattern, we can capture the interesting ones. If a model can learn any function, to within some amount of precision, we call it a universal function approximator.

There's a lot of ways we can go about approximating a function. One way, that happens to be easy to draw, is by using the sum of piece-wise linear functions. Given enough pieces, we can approximate any function, arbitrarily well. We could also choose to use piece-wise constant functions, but linear functions tend to play nicer with gradient based methods.

We can think about piece-wise linear functions as being constructed from "elbow" functions. The function that looks like this is called the Rectified Linear Unit, or ReLU.

We can think of the circuit that combines some number of elbows using a one hidden layer network. There's a lot of other non-linearities we could use. Sigmoids and hyperbolic tangents are also used, but ReLUs are more intuitive to visualize as universal approximators. It's also very easy to calculate partial derivatives when using ReLUs.

$$ReLU_{w,b}(x) = \max(0, wx + b) = \begin{cases} wx + b & \text{if } wx \geq -b \\ 0 & \text{else} \end{cases}$$

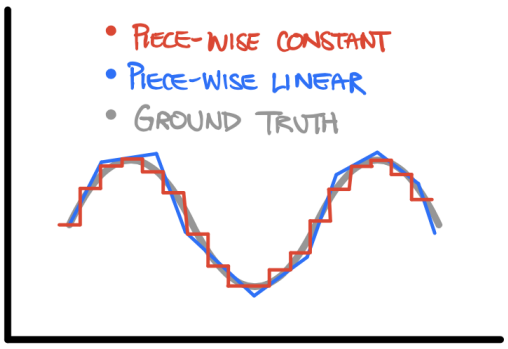


Figure 4: We can approximate functions using piece-wise functions. In red is shown an approximation using piece-wise constant functions, blue shows an approximation using piece-wise linear functions.

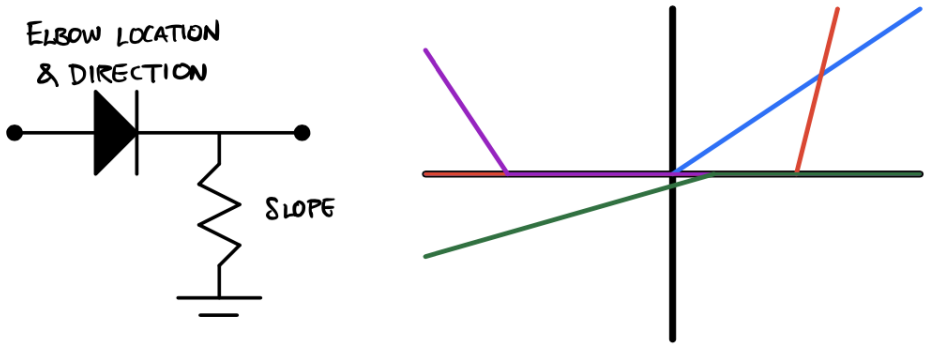


Figure 5: An analog circuit that creates elbow waveforms, and four example (scaled) ReLU functions.

If $w > 0$,

$$= \begin{cases} wx + b & \text{if } x \geq -\frac{b}{w} \\ 0 & \text{else} \end{cases}$$

if $w < 0$,

$$= \begin{cases} wx + b & \text{if } x \leq -\frac{b}{w} \\ 0 & \text{else} \end{cases}$$

To obtain values less than zero, we need to modify our circuit a bit more. The output of the ReLU function will always be non-negative. If we want to represent negative numbers (like the green elbow in Figure 5) we need to multiply the ReLU output by a negative number. We can do this by inserting an additional weight multiplication element on the output of each elbow unit before summing each output together (see Figure 7).

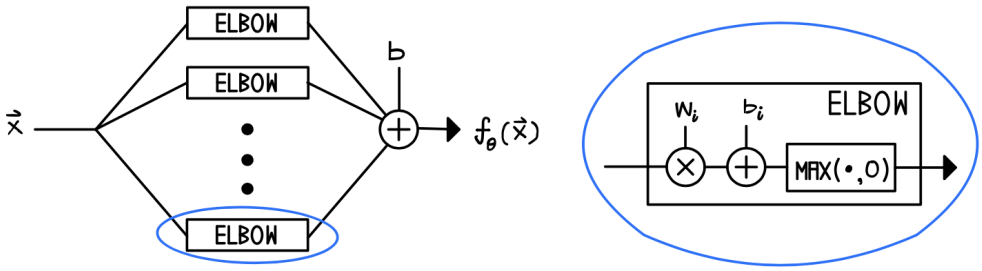


Figure 6: A circuit that combines $d$ ReLU elbows.

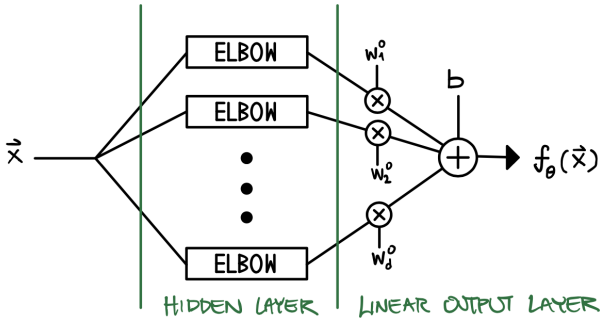


Figure 7: Scaling our elbow outputs lets us form linear combinations of non-linear functions. This change allows us to approximate any function, and reveals the basic structure of deep neural nets.

With one layer and an arbitrarily large layer width (number of ReLUs), we can approximate any function. Historically, people have just stuck with using single layer universal approximators. So why move into the domain of deep networks?

We use deep neural networks because they tend to exhibit *learnability*, meaning they can actually be trained to learn patterns of interest.

When we make this shift to deep networks, some of our parameters become redundant, so the number of parameters is not directly (though highly correlated with) the degrees of freedom our network has.

A final note on ReLUs: they work well with gradient based methods because they are non-saturating (unlike sigmoid, for example). For ReLUs, the gradient doesn't depend on the value of $x$ other than its value relative to the threshold. For a saturating non-linearity, the magnitude of the gradient approaches zero as the magnitude of $x$ itself increases to infinity. Using a non-saturating non-linearity ensures that our gradients don't diminish in deep networks with long partial derivative chains.

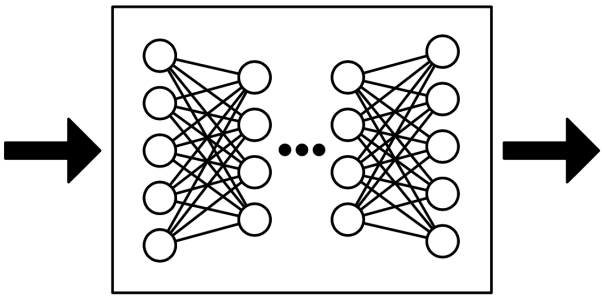

Figure 8: Deep neural networks repeat the basic structure shown in Figure 7. Even though a single hidden layer is suitable for universal function approximation (very expressive), in practice deep neural nets are easier to train (very learnable).

# 3 What we wish this lecture included

When some function is not successfully learned, Hornik's theorem proves that this is due to either "inadequate learning, insufficient numbers of hidden units or the lack of a deterministic relationship between input and target" [1]. This theorem captures what can go wrong in a typical machine learning flow, and implicitly describes learnability and expressivity. It also points out the importance of ensuring that there are patterns of interest to learn in the dataset.

Although the notion has proven somewhat outdated, discussing ways that expressivity can be measured (namely, VC dimension) may prove helpful.

# References

[1] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, jul 1989.