EECS 182 Deep Neural Networks
Spring 2023 Anant Sahai
Note 2

## 1 Gradient Descent: SGD and Learning Rate

Optimizers are used to solve for models' parameters. The optimizers themselves usually have their own hyperparameters (learning rate, choice of the optimizer, etc.). The most basic optimizer is gradient descent, an iterative local optimizer that we have touched on briefly in Note 1. Just to recap a bit, the idea is to change parameters a little bit each time 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 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.

### 1.1 SGD

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.

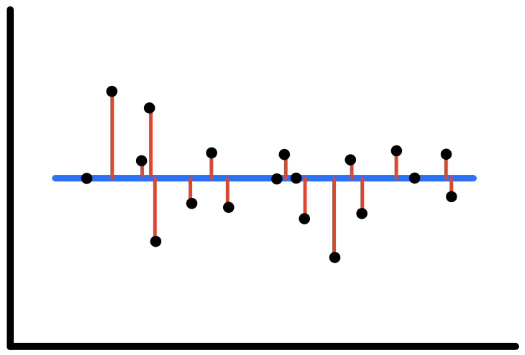


**Figure 1:** Contributions from each data point in the dataset balance, so the gradient is zero.

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.

## 1.2 Learning Rate

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.

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.

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

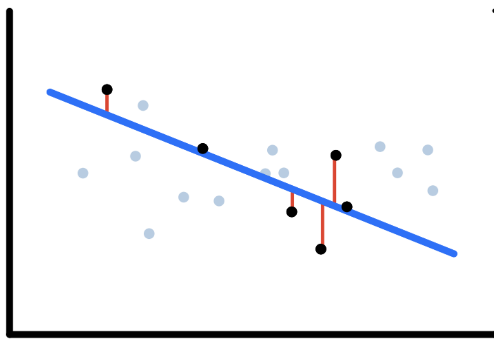


**Figure 2:** Contributions from this subset of data points are unbalanced, so the gradient found by SGD is nonzero even though it should balance.

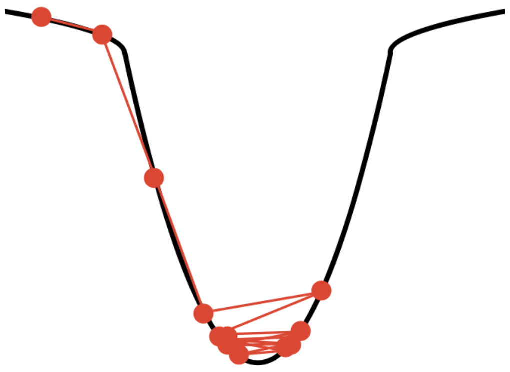


**Figure 3:** Without a decaying learning rate, we may just dance around the optimum.

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

### 2.1 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). Neural nets are prevalent today because they're able to exploit gradient descent/SGD to do learning/computation.

### 2.2 Two goals of the analog circuits

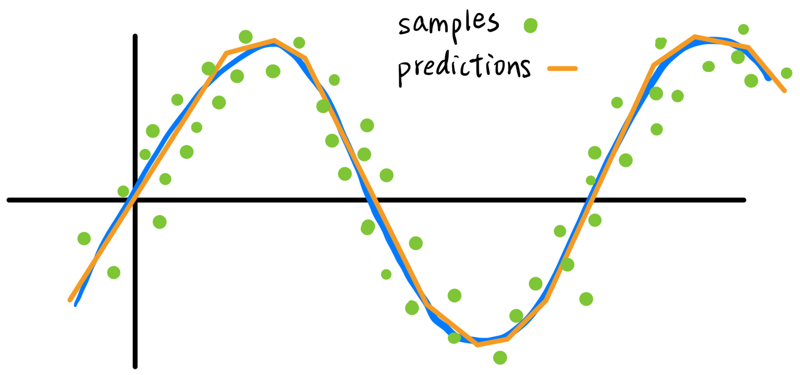
(a) **Expressivity**: the ability to represent a wide variety of functions.

(b) **Learnability**: the ability to learn functions without too much trouble.

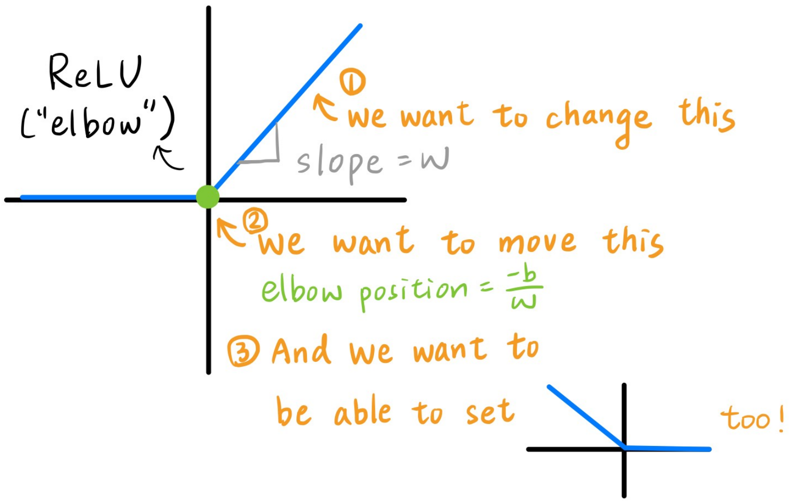
Expressivity is important because we need to be able to express patterns we're interested in, which we don't know and need to be learned. We can't guarantee that some model is expressive enough to capture these patterns. Traditionally, the solution to this is to be expressive enough to capture any pattern. If a model can learn any function, to within some amount of precision, we call it a universal function approximator.

### 2.3 Example of Neural Networks

The figure below shows a 1-D nonlinear (Blue) function, piecewise linear (Orange) functions, and data points (Green). 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 (Orange) that best match the nonlinear function (Blue) based on the data points using Neural Nets.



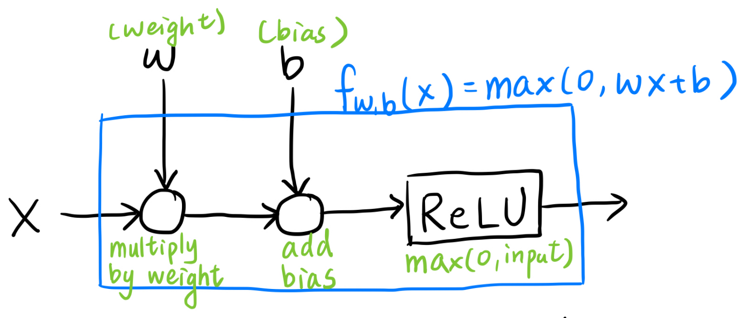
Then, how do we create the piecewise linear functions? We do it by forming a linear combination of elbows (Rectified Linear Units) that looks like:

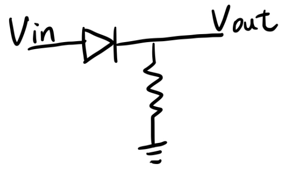


The standard ReLU function is shown below.

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

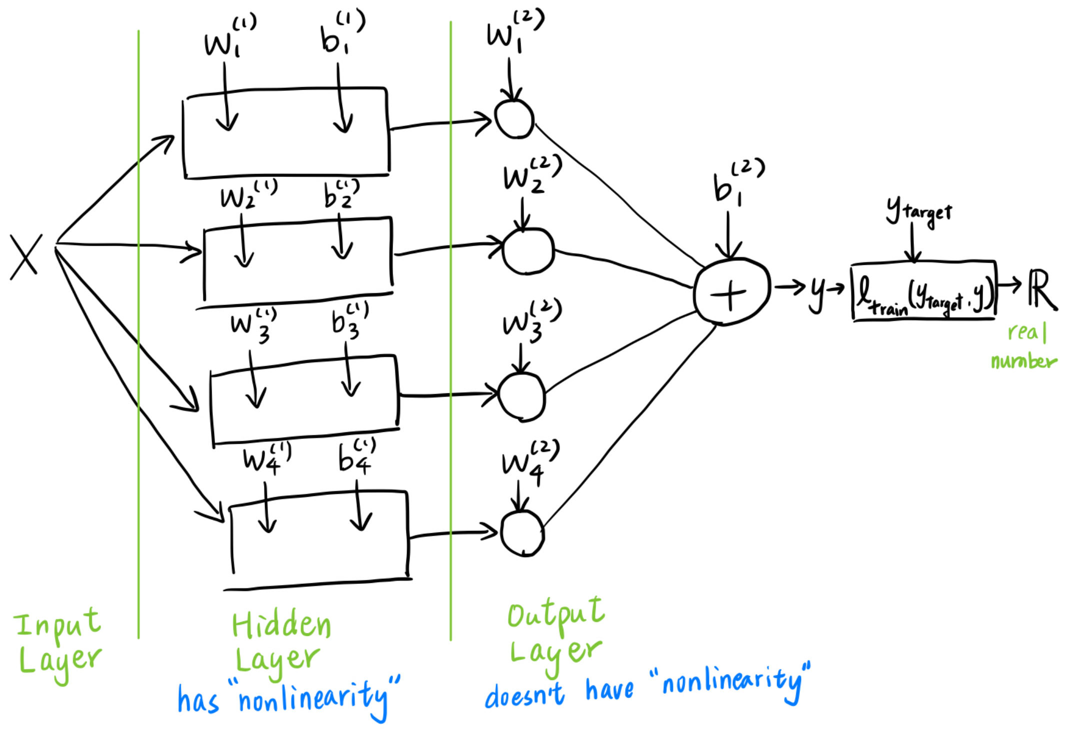
Also, the standard ReLU function can be modified if needed. For example, $x$ can be replaced with $wx + b$, so that the modified ReLU function becomes $f(x) = max(0, wx + b)$, which can be represented by a computational graph like this:





We can manipulate an elbow to fit our line by changing $w$ (weight) and $b$ (bias). More specifically, Elbow is located at $-\frac{b}{w}$ and slope = $w$. Here, w and b are the parameters($\theta$) we want to minimize using a loss function.

To provide the circuit interpretation of ReLU, we are going to look at an example of a rectifier circuit, which is shown in the figure below the computational graph. It 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. In this example, $V_{in}$ is $x$ and $V_{out}$ is $f(x)$.



The above image illustrates the big picture of what our simple ReLU Neural Net looks like. We tune the elbows to fit our training points, the optimizer is trying to make the final real number (loss) as small as possible across all the training points. To push the loss down, how much do I have to move $y$, $b$'s, and $w$'s? This process goes backward and the elbow changes.

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.

## 2.4 Asides for ReLUs

- Non-saturating

  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.

- Dead ReLUs

  If both weights and biases are distributed with normal distributions, the ratio $\frac{b}{w}$ will be a Cauchy distribution. This causes some ReLUs to be located far apart from the others. These ReLUs are considered "dead" since they output 0, and changing the weights or biases slightly doesn't affect their output.