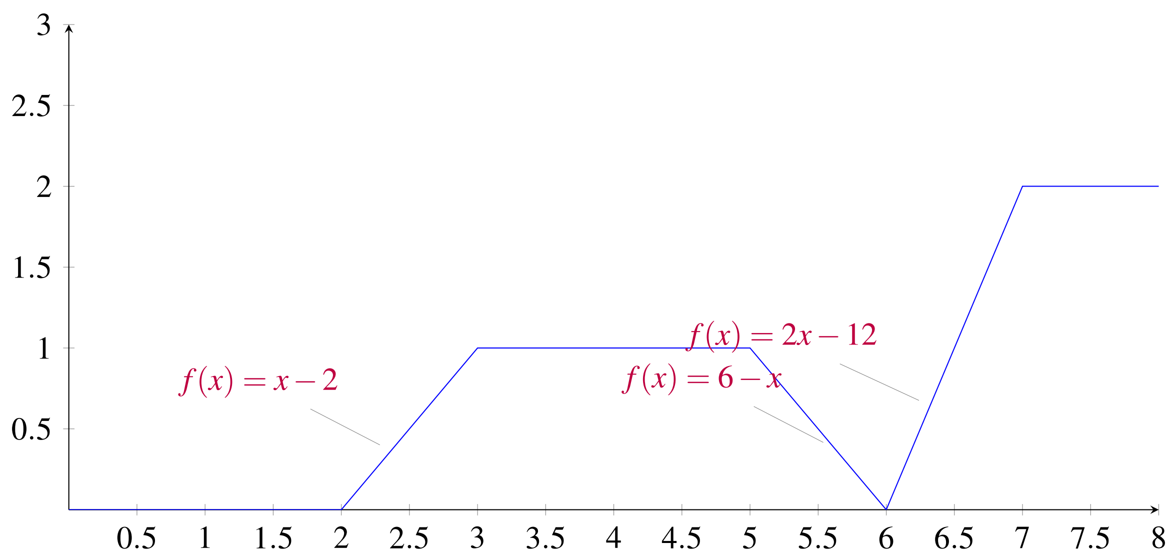
1 ReLUs as Universal Function Approximators

Last time we saw that the second-most simple universal function approximator was the **piecewise linear function**. Specifically, we talked about a specific component of piecewise linear functions called the **ReLU**, which is defined as $f(x) = \max(0,x)$.

In our discussion of neural nets, we saw that we would have the ReLUs act on linear combinations of neural net units to introduce nonlinearity to the hypothesis function encoded by the neural net. For example, when acting on one input (and a bias term) our ReLUs will take in arguments of the form $a+bx$. Let's see an example of how expressive they can be. Suppose we wanted to build this function from ReLUs:



All we would need to do is center a ReLU at each hinge of the function and give it the appropriate parameters. For example, to match f from 0 to 3, we would only need the ReLU defined by $\max(0,x-2)$. The full function can be matched with this linear combination of ReLUs (and a constant bias term):

$$f(x) = -1 + \max(0,x-2) - \max(0,x-2) + \max(0,6-x) - \max(0,5-x) + \max(0,2x-12)$$

Here's the plot on Wolfram Alpha.

In higher dimensions, i.e. when a ReLU takes in an arbitrarily long dot-product as input: $f(x) = \max(0,w^\top x)$, the unit can be viewed as representative of a "ramp" in the higher-dimensional space. Here's a plot of a 3-D ramp:

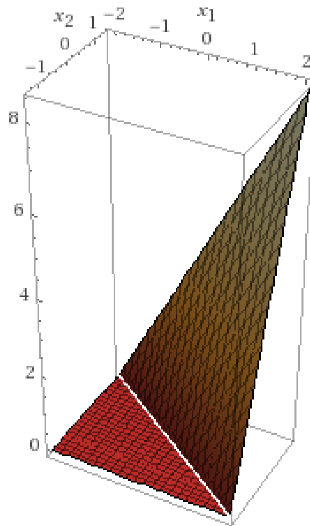
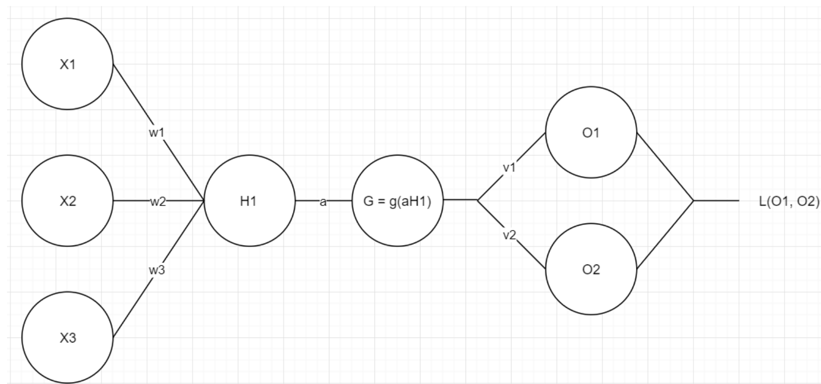


Figure 1: $f(x_1, x_2) = \max(0, 2x_1 + 3x_2)$

2 Derivatives in Neural Nets

We have a very powerful tool at our disposal in neural nets, but it does us no good if we can't train them. Let's talk about how this happens. The output units of a neural net can be thought of as akin to a **regression** or **hypothesis function** determined by the parameters of some model. For example, in ordinary least-squares regression, we learned a hypothesis function $f(x) = w^\top x$ determined by the parameter w. It is just the same with neural nets, except that our hypothesis function can be arbitrarily complex. Consider the following neural net:



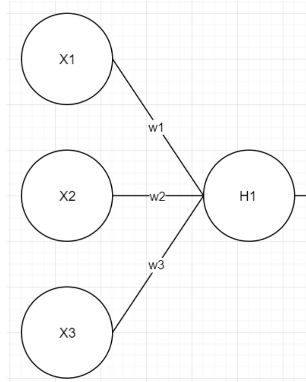
The hypothesis function that this neural net encodes is represented by the two outputs, $O = [O_1, O_2]$.

Since neural net outputs are not linear functions of their inputs, there is no closed-form solution for the minimum to any reasonable loss function defined on them. Thus, we resort to using gradient descent to train them. To run gradient descent, we must calculate the derivative of the loss function with respect to the parameters of the neural net. In the network depicted above, the parameters are $W = [w_1, w_2, w_3]$, a, and $V = [v_1, v_2]$. These are the values of the model which we are allowed to tweak. **Backpropagation** is an efficient manner of computing these gradients that exploits the

nested structure of neural nets.

2.1 The Chain Rule

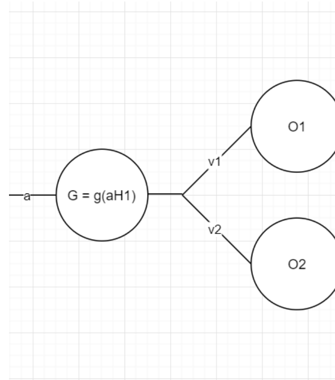
This section is an aside meant to recall your knowledge about the chain rule in multivariable calculus. Let's take a look at two slices of our neural net from above.



If you want to compute the derivatives of upstream stuff with respect to the weights on these connections, you need only consider the input of a single connection at a time. That is to say:

$$\frac{\partial L}{\partial w_1} = \text{upstream_terms} \cdot \frac{\partial H_1}{\partial w_1}$$

completely independent of x_2 and x_3.



If you want to compute the derivatives of upstream stuff with respect to the weights *downstream of these connections*, you'll need to sum over the contributions of the inputs to these connections. That is to say:

$$\frac{\partial L}{\partial a} = \sum_i \text{upstream_terms}_i \cdot \frac{\partial O_i}{\partial a}$$

2.2 Backpropagation

A naive way of calculating the gradients might be to differentiate the loss with respect to each parameter we're interested in updating, one at a time. However, because the later layers of a neural

net are just functions of the earlier layers, doing this would be wasteful. We can see this by taking a look at the derivatives of L with respect to v_1, a, and w_1 in our example:

$$\begin{aligned}
\frac{\partial L}{\partial v_1} &= \frac{\partial L}{\partial O_1}\frac{\partial O_1}{\partial v_1} \\
\frac{\partial L}{\partial a} &= \frac{\partial L}{\partial O_1}\frac{\partial O_1}{\partial a} + \frac{\partial L}{\partial O_2}\frac{\partial O_2}{\partial a} \\
\frac{\partial L}{\partial w_1} &= \frac{\partial L}{\partial O_1}\frac{\partial O_1}{\partial w_1} + \frac{\partial L}{\partial O_2}\frac{\partial O_2}{\partial w_1}
\end{aligned}$$

You should notice that by invoking the chain rule, we see that the term $\frac{\partial L}{\partial O_1}$ is common to all three derivatives, and the term $\frac{\partial L}{\partial O_2}$ is common to the second two. This suggests that we should consider caching the results of the derivatives of weights in the later layers and reuse them in our computation of the derivatives of the weights of earlier layers: a dynamic programming approach.

The following is a general outline of how backpropagation might be implemented. Keep in mind that the specifics, especially those pertaining to the structure of each successive layer, will depend heavily on the architecture of the neural net in question.

1. The **forward pass**: populate each unit of the neural net with the value it's supposed to have (i.e., invoke all your dot-products and ReLUs).
2. Start at the upstream end, i.e. the outputs. Compute the gradient of the loss function with respect to the outputs (these are just numbers), and memoize/cache them.
3. Go back one layer. Now, treat your outputs O_i as endpoints of your neural net, and compute the gradients w.r.t. the previous layer, caching these as well. The contributions to the final loss should be summed appropriately over the paths through which they influence the loss.
4. Repeat until you hit the last layer (the inputs). You should now have all the necessary components to compute the derivative of the loss function with respect to *any* parameter of the neural net.

3 Speeding up Gradient Descent

Backpropagation efficiently computes gradients, so we can run gradient descent to optimize neural nets. However, computing the full gradient may be expensive, particularly if we have a lot of data. Consider that the loss function in machine learning typically is an average (or sum, but this is the same up to a constant factor) of errors over the training points:

$$L(w) = \frac{1}{n}\sum_{i=1}^{n} \ell(h(x_i, w), y_i)$$

By the linearity of differentiation, we easily see that

$$\nabla L(w) = \frac{1}{n}\sum_{i=1}^{n} \nabla_w \ell(h(x_i, w), y_i)$$

So computing the gradient takes time in linear in n. In the "big data" regime, n is very large, so this cost may be unacceptable.

For this reason, we have reason to try to approximate the gradient of the loss function of a neural net by *taking a representative random sample* of the inputs to calculate the gradient over. Since gradient descent is an iterative process, we might consider forfeiting the exactness of the update of each individual iteration in exchange for better speed of computation. If we take the gradient over a random sample of these training samples at each step, we will get a noisy but unbiased estimate of the true gradient. The noise in each step is often an acceptable tradeoff in exchange for the opportunity to take many more steps.

In the regular gradient descent update, we have the policy:

$$w^{(k+1)} \leftarrow w^{(k)} - \alpha_k \nabla_w L(w^{(k)})$$

In **stochastic gradient descent**, we have the update rule:

$$w^{(k+1)} \leftarrow w^{(k)} - \alpha_k G_k$$

where G_k is a random variable which satisfies $\mathbb{E}[G_k] = \nabla L(w^{(k)})$. Typically G_k is constructed by sampling $m < n$ training points uniformly at random (say the index is $I_k \subset \{1, \ldots, n\}$) and computing the gradient on these points only:

$$G_k = \frac{1}{m} \sum_{i \in I_k} \nabla_w \ell(h(x_i), y_i)$$

We denote the true optimum by w^*. Our goal is to show the following:

$$\lim_{k \to \infty} \mathbb{E}[\|w^{(k)} - w^*\|^2] = 0$$

We make the following assumptions in the analysis below:

1. The loss function f is ℓ-strongly convex; that is, there exists a constant $\ell > 0$ such that $(\nabla f(x) - \nabla f(y))^\top (x - y) \geq \ell \|x - y\|^2$ for all x, y.
2. The expected squared norm of the stochastic gradient is bounded as $\mathbb{E}[\|G_k\|^2] \leq M^2 < \infty$.

We begin by expanding the desired term:

$$\begin{aligned}
&\mathbb{E}[\|w^{(k+1)} - w^*\|^2] \\
&= \mathbb{E}[(w^{(k+1)} - w^*)^\top (w^{(k+1)} - w^*)] \\
&= \mathbb{E}[(w^{(k)} - w^* - \alpha_k G_k)^\top (w^{(k)} - w^* - \alpha_k G_k)] \\
&= \mathbb{E}[\|w^{(k)} - w^*\|^2] - 2\alpha_k \mathbb{E}[(w^{(k)} - w^*)^\top G_k] + \alpha_k^2 \mathbb{E}[\|G_k\|^2]
\end{aligned}$$

For brevity, define

$$d_k = \mathbb{E}[\|w^{(k)} - w^*\|^2]$$

Our assumption on the expected squared norm of G_k implies

$$d_{k+1} \le d_k - 2\alpha_k \mathbb{E}[(w^{(k)} - w^*)^\top G_k] + \alpha_k^2 M^2$$

To evaluate the expectation, we condition on the past (i.e. all random decisions that contributed to $w^{(k)}$, including past choices of points to evaluate the gradient at). By the law of iterated expectation, we have

$$\mathbb{E}[(w^{(k)} - w^*)^\top G_k] = \mathbb{E}_{\text{past}}\left[\mathbb{E}[(w^{(k)} - w^*)^\top G_k \mid \text{past}]\right]$$

Here the inner expectation is taken over the choice of training points used to compute the stochastic gradient. But given the past (which includes $w^{(k)}$), we already know $w^{(k)}$, so

$$\mathbb{E}[(w^{(k)} - w^*)^\top G_k \mid \text{past}] = (w^{(k)} - w^*)^\top \mathbb{E}[G_k \mid \text{past}]$$

The current gradient G_k depends on our new choice of evaluation point, which is presumed independent of the past and thus $\mathbb{E}[G_k \mid \text{past}] = \mathbb{E}[G_k] = \nabla f(w^{(k)})$, where the second equality holds because G_k is an unbiased estimator for the true gradient at $w^{(k)}$. Putting all this together, we have

$$\mathbb{E}[(w^{(k)} - w^*)^\top G_k] = \mathbb{E}_{\text{past}}[(w^{(k)} - w^*)^\top \nabla f(w^{(k)})]$$

First order necessary conditions for optimality imply that $\nabla f(w^*) = 0$. Then by the assumption of ℓ-strong convexity, we have

$$(w^{(k)} - w^*)^\top \nabla f(w^{(k)}) = (w^{(k)} - w^*)^\top (\nabla f(w^{(k)}) - \nabla f(w^*)) \ge \ell \|w^{(k)} - w^*\|_2^2$$

Taking expectations yields

$$\mathbb{E}[(w^{(k)} - w^*)^\top \nabla f(w^{(k)})] \ge \ell \mathbb{E}[\|w^{(k)} - w^*\|_2^2] = \ell d_k$$

Putting this back into our inequality for d_{k+1}, we get

$$d_{k+1} \le d_k - 2\alpha_k \ell d_k + \alpha_k^2 M^2 = (1 - 2\alpha_k \ell) d_k + \alpha_k^2 M^2$$

The $\alpha_k^2 M^2$ term was incurred by the randomness in our updates. We can try to send this term to 0 by diminishing the step size α_k over time – but decreasing α_k will also decrease the effect of the $(1 - 2\alpha_k \ell) d_k$ term, so we need to choose our step size carefully.

Setting $\alpha_k = \frac{1}{2\ell k}$ gives

$$d_{k+1} \le \left(1 - \frac{1}{k}\right) d_k + \frac{1}{(2\ell k)^2} M^2$$

Let $S = \frac{M^2}{(2\ell)^2}$ so that this inequality becomes

$$d_{k+1} \le \left(1 - \frac{1}{k}\right) d_k + \frac{1}{k^2} S$$

To analyze this recurrence, we expand the first few terms:

$$\begin{aligned}
d_2 &\leq S \\
d_3 &\leq \left(1 - \frac{1}{2}\right) d_2 + \frac{1}{2^2} S = \frac{1}{1 \cdot 2} S + \frac{1}{2^2} S \\
d_4 &\leq \left(1 - \frac{1}{3}\right) d_3 + \frac{1}{3^2} S = \frac{1}{1 \cdot 3} S + \frac{1}{2 \cdot 3} S + \frac{1}{3^2} S \\
d_5 &\leq \left(1 - \frac{1}{4}\right) d_4 + \frac{1}{4^2} S = \frac{1}{1 \cdot 4} S + \frac{1}{2 \cdot 4} S + \frac{1}{3 \cdot 4} S + \frac{1}{4^2} S
\end{aligned}$$

Inductively, we find that

$$d_n \leq \frac{S}{n-1} \sum_{j=1}^{n-1} \frac{1}{j}$$

Because the harmonic sum $\sum_{j=1}^{n-1} \frac{1}{j}$ behaves as $\ln(n)$, we see that d_n is upper bounded in the limit by $S \frac{\ln(n)}{n} \to 0$. Hence $\lim_{n \to \infty} d_n = 0$ as desired.