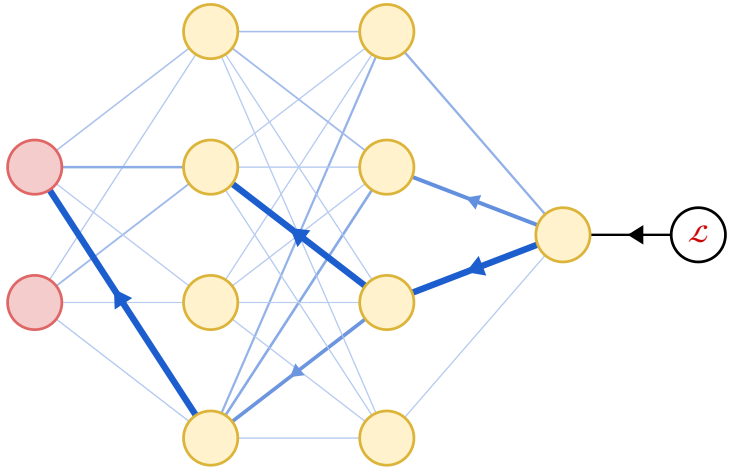


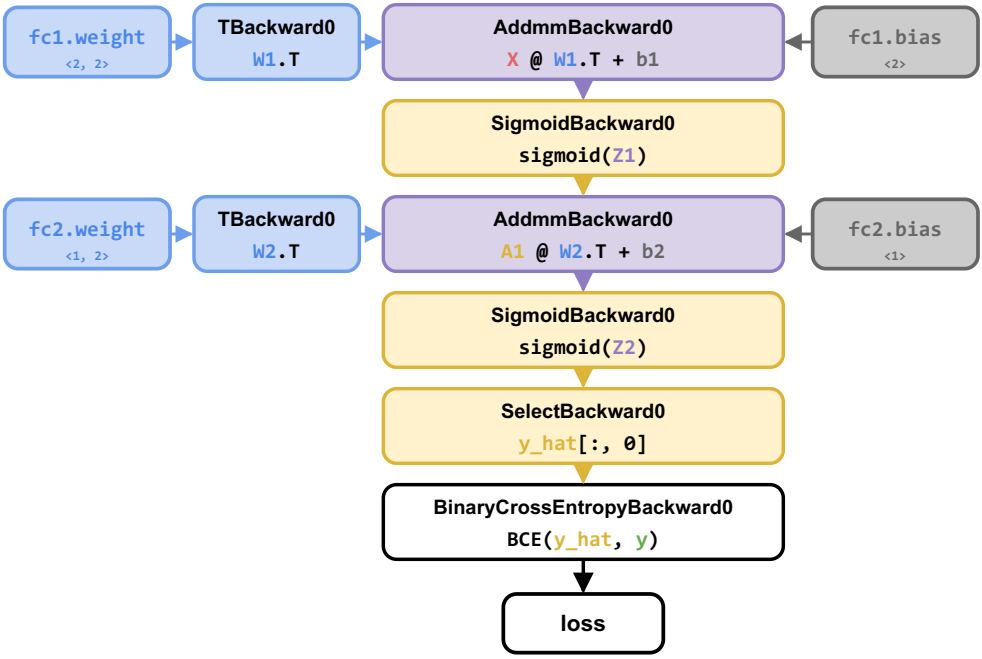
UNIT 03 · CHAPTER 1

Backpropagation

CSE 30124 · Introduction to Artificial Intelligence
William Theisen · University of Notre Dame

Learning a Network's Weights

Feed-Forward Networks ended on the graph PyTorch records as our close-game network runs, every operation from the games to the loss:



That network gets every game right only because we picked its weights by hand, which worked because we already knew the answer (the margin). For a bigger network, or a question we can't answer ourselves, the weights have to be learned. Gradient descent itself is nothing new, it's how our logistic regression found its weights: work out how the loss changes with every weight, step each weight a little the other way, and repeat.

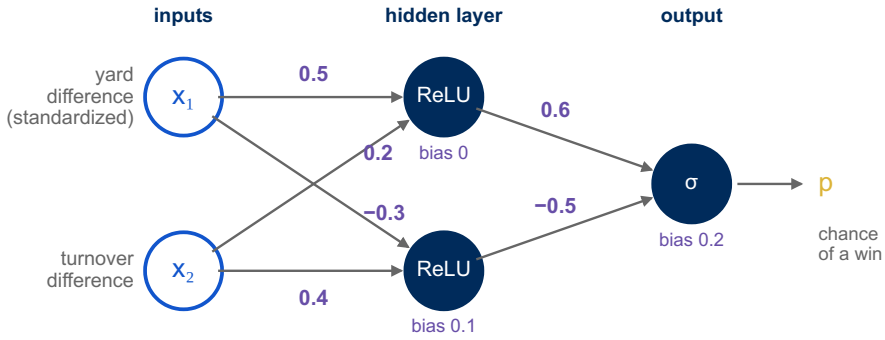
Weights Without Labels

For logistic regression the chain rule gave us that gradient through three links, and the answer came out as a tidy $(p - y)x_j$. A network also has weights in its hidden layer, and as we saw with the multilayer perceptron, **nobody can tell us what a hidden neuron should have said**. There's no label for it to be wrong about.

Notice that gradient descent never needed a label for each weight though. All it asks is how much the loss would change if we nudged a weight, and that question makes just as much sense for a weight in the hidden layer as for one in the output. So given a network, our goal is to **find the derivative of the loss with respect to every weight**, cheaply enough to do it thousands of times. That is what the graph is for.

A Small Network

To keep the arithmetic short we'll keep the close-game network's 2-2-1 shape, but go back to "Did we win?" (the score already says whether a game was close). Ours takes the **standardized** yard difference and turnover difference (Notre Dame's turnovers minus the opponent's), passes them to two ReLU neurons, and ends in one sigmoid neuron giving the chance of a win:



That's nine weights, counting the biases, picked by hand to keep the numbers readable (in practice they start as small random numbers). Written out, it's the same weighted sum as always, done twice with a ReLU in between (a row of $\mathbf{w}_1$ per hidden neuron, as in Feed-Forward Networks), and the loss is Logistic Regression's cross-entropy:

the hidden layer

$$\mathbf{z}_1 = \mathbf{x}\,\mathbf{w}_1^{\mathsf{T}} + \mathbf{b}_1 \qquad \mathbf{a}_1 = \mathrm{ReLU}(\mathbf{z}_1) = \max(0,\ \mathbf{z}_1)$$

the output and the loss

$$\mathbf{z}_2 = \mathbf{a}_1\mathbf{w}_2^{\mathsf{T}} + \mathbf{b}_2 \qquad \mathbf{p} = \sigma(\mathbf{z}_2) \qquad \mathcal{L} = \mathrm{BCE}(\mathbf{p},\ \mathbf{y})$$

The Forward Pass

Our one game is Clemson in 2015, the 24 to 22 loss in the rain that kept turning up in Logistic Regression. Notre Dame outgained Clemson by 136 yards but turned the ball over 4 times to Clemson's 1. Standardizing as in High Dimensional Data, our games average a yard difference of 76 (standard deviation 135) and a turnover difference of −0.22 (1.79), so $\mathbf{x} = [0.44, 1.80]$, and $y = 0$. Running it through the network one layer at a time:

hidden layer, before the ReLU

$$z_1 = [\,0.5(0.44) + 0.2(1.80) + 0, \quad -0.3(0.44) + 0.4(1.80) + 0.1\,] = [\,0.58, \;\; 0.69\,]$$

both positive, so the ReLU keeps them

$$a_1 = [\,0.58, \;\; 0.69\,]$$

output, before the sigmoid

$$z_2 = 0.6(0.58) - 0.5(0.69) + 0.2 = 0.20$$

chance of a win

$$p = \sigma(0.20) = 0.55$$

the game was a loss

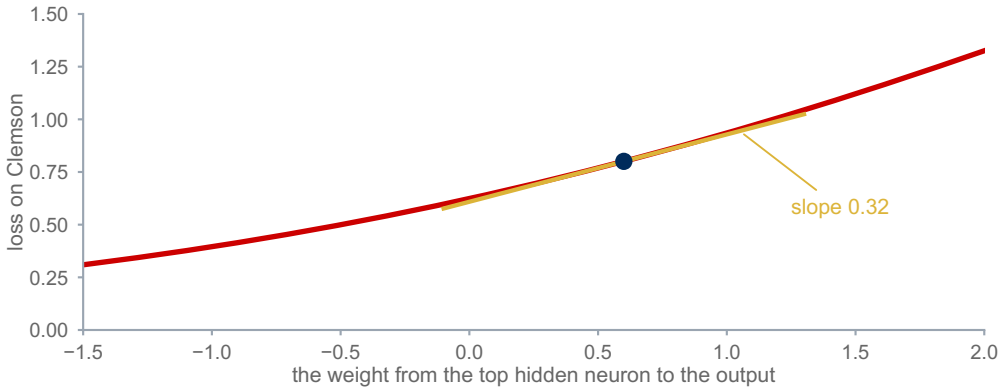
$$L = -\log(1 - 0.55) = 0.80$$

That's the **forward pass** from Feed-Forward Networks, one layer after another from the inputs to the loss. With its starting weights our network gives Clemson a 55% chance of a win that never came, and the game costs it 0.80.

Those starting weights are no good, and we'd like to know which way to move each of them. Unfortunately the forward pass only tells us how wrong we are, not whose fault it is.

Numerical Gradients

The most direct way to find out how the loss depends on a weight is to nudge the weight and watch. Raise the top hidden neuron's weight into the output from `0.6` to `0.601`, run the forward pass again, and the loss on Clemson goes from `0.80053` to `0.80085`. A change of `0.00032` for a nudge of `0.001` is a slope of about `0.32`:



The slope is positive, so to lower the loss we should lower this weight, which makes sense (that neuron pushes toward a win, and this was a loss). Estimating a derivative by nudging is called a **numerical gradient**, and it works just as well for a hidden weight as for any other.

Unfortunately it costs a forward pass for every weight (two if we nudge both ways, which is more accurate). Our nine weights need `18` per step, and a network with a million weights would need **two million forward passes per step**.

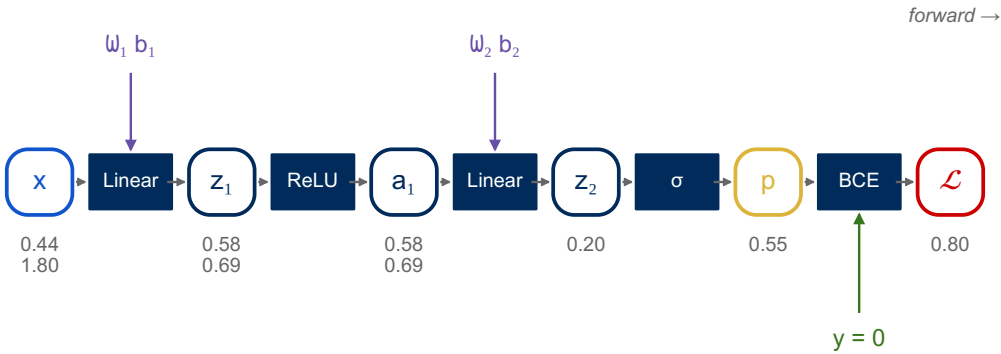
Computational Graphs

Written as a single expression, our network's loss is a function of a function of a function:

the whole network, nested

$$\mathcal{L} = \mathrm{BCE}\Big(\sigma\big(\mathrm{ReLU}(x\,W_1{}^{\mathsf{T}} + b_1)\,W_2{}^{\mathsf{T}} + b_2\big),\quad y\Big)$$

As in Feed-Forward Networks, we can draw it as a **computational graph**, each operation a box and each value a node. Here's ours, run forward on Clemson, with every value written under its node (Linear is $x\,W^{\mathsf{T}} + b$, as in PyTorch):



Notice that every weight enters the graph in exactly one place, and from there it can only affect the loss through the nodes to its right. W_1 only matters through z_1, which only matters through a_1, and so on down the line to $\mathcal{L}$. So **how much the loss depends on a weight is built out of how much each node depends on the one before it**.

The Chain Rule

That's the chain rule, which we used in Logistic Regression. The derivative of nested functions is the product of the derivatives of each layer:

[1.1] the chain rule

$$\frac{d}{dx}\left[f(g(x))\right] = f'(g(x))\,g'(x)$$

On a graph we can apply it one box at a time. Each box only needs its own **local derivative**, how much its output moves when its input does, and the derivative of the loss with respect to any node is the product of the local derivatives on the way back from the loss. For the output weights we nudged:

the loss on a loss ($y = 0$)

$$\frac{\partial \mathcal{L}}{\partial p} = \frac{1}{1-p} = 2.23$$

times the sigmoid's $p(1-p)$

$$\frac{\partial \mathcal{L}}{\partial z_2} = 2.23 \times 0.55(1-0.55) = 0.55 = p - y$$

times a_1, since $z_2 = a_1 W_2^T + b_2$

$$\frac{\partial \mathcal{L}}{\partial W_2} = 0.55 \times [\,0.58,\ \ 0.69\,] = [\,0.32,\ \ 0.38\,] \qquad \frac{\partial \mathcal{L}}{\partial b_2} = 0.55$$

The first of those is the `0.32` we found by nudging, from three multiplications and no extra forward passes. The sigmoid's $p(1-p)$ cancels out against the loss exactly as it did in Logistic Regression, leaving $p - y$. The output layer really is just a logistic regression whose features happen to be a_1.

Gradients of Hidden Weights

The hidden weights are further from the loss, but the chain only gets longer. We already know $\partial \mathcal{L} / \partial z_2 = 0.55$, so there's no need to start from $\mathcal{L}$ again, we carry on from there:

times w_2

$$\frac{\partial \mathcal{L}}{\partial a_1} = 0.55 \times [\,0.6,\ -0.5\,] = [\,0.33,\ -0.28\,]$$

times the ReLU's 1 or 0

$$\frac{\partial \mathcal{L}}{\partial z_1} = [\,0.33,\ -0.28\,] \odot [\,1,\ 1\,] = [\,0.33,\ -0.28\,]$$

times x, for each neuron and input

$$\frac{\partial \mathcal{L}}{\partial w_1} = \begin{bmatrix} 0.33(0.44) & 0.33(1.80) \\ -0.28(0.44) & -0.28(1.80) \end{bmatrix} = \begin{bmatrix} 0.15 & 0.59 \\ -0.12 & -0.50 \end{bmatrix}$$

times 1

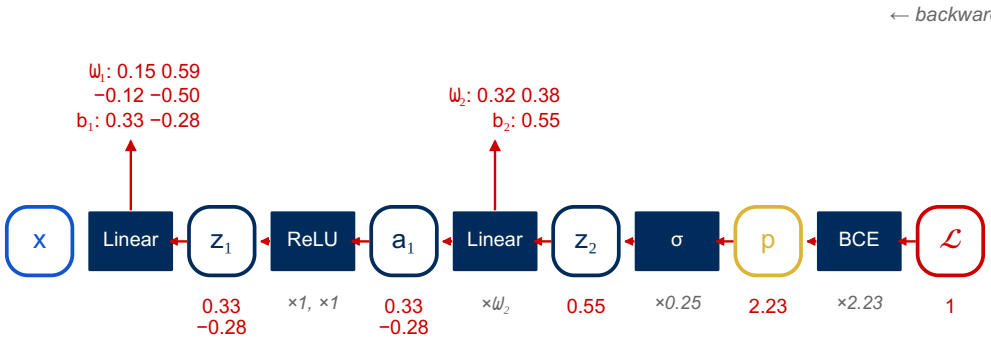
$$\frac{\partial \mathcal{L}}{\partial b_1} = [\,0.33,\ -0.28\,]$$

The ReLU's local derivative is `1` where its input was positive and `0` where it wasn't (the ⊙ multiplies entry by entry), so it either passes the derivative straight through or blocks it. Both of Clemson's hidden values were positive, so nothing was blocked.

The top neuron's weight on turnovers gets `0.59`, the biggest of the nine. Nudging it by `0.001` moves the loss by `0.00059`, so the chain rule and the nudge **agree here too**. A positive number means we should lower that weight, and lowering a weight on turnovers is what you'd want after a game lost with `4` of them.

Backpropagation

Pictured below is everything we just did, drawn on the graph. Starting from `1` at the loss, each box multiplies the derivative coming in from its right by its local derivative and passes the result left, and the weights' derivatives drop out along the way.



Computing the gradient this way, a forward pass that keeps every value, then a backward pass that applies the chain rule one box at a time from the loss back toward the inputs, is called **backpropagation**. The backward pass costs about as much as the forward pass, however many weights there are.

Backpropagation isn't a new way of learning. It's **an algorithm for getting the gradient**, and gradient descent does the rest. It was worked out more than once (Seppo Linnainmaa published the general method in `1970`), but a `1986` paper in *Nature* by David Rumelhart, Geoffrey Hinton and Ronald Williams made it the standard way to train hidden layers, the piece missing in Minsky and Papert's day.

Learning in the Brain

Since our networks are loosely modeled on brains, it's natural to ask whether the brain learns this way. Probably not, at least not exactly. In 1949 Donald Hebb proposed that a connection between two neurons gets stronger when both are active together (later summed up as "cells that fire together, wire together"). That rule is **local**, a synapse only needs to know about the two neurons it connects.

Backpropagation isn't local. To update one weight we need the derivatives from every layer after it, sent backwards through **the very same weights** the forward pass used. Neurons only send signals one way, so the brain would need a second set of connections running backwards with exactly matching weights, and nobody has found one (this is called the **weight transport problem**).

It may not need to be exact. Lillicrap et al. (2016) sent the errors backwards through **fixed random weights** instead, and the networks still learned. How the brain really assigns credit to its synapses is still an open question.

Gradient Checking

Backpropagation is easy to get slightly wrong, and a slightly wrong gradient still trains, just badly. So it's common to check it against numerical gradients on a small example. Here's the backward pass for Clemson, checked on W_1 (the last line printed is the biggest disagreement):

```python
import torch
f64 = torch.float64  # numerical gradients need precision
x = torch.tensor([0.4407, 1.7996], dtype=f64)  # Clemson
W1 = torch.tensor([[0.5, 0.2], [-0.3, 0.4]], dtype=f64)
b1 = torch.tensor([0.0, 0.1], dtype=f64)
W2 = torch.tensor([0.6, -0.5], dtype=f64)
b2 = 0.2
def forward(W1):
    z1 = x @ W1.T + b1
    a1 = torch.relu(z1)
    p = torch.sigmoid(a1 @ W2 + b2)
    return z1, p, -torch.log(1 - p)        # y = 0
z1, p, L = forward(W1)
dz1 = (p - 0) * W2 * (z1 > 0)     # backward pass
dW1 = torch.outer(dz1, x)
eps, num = 1e-5, torch.zeros(2, 2, dtype=f64)
for i in range(2):                # nudge each weight
    for j in range(2):
        E = torch.zeros(2, 2, dtype=f64)
        E[i, j] = eps
        num[i, j] = (forward(W1 + E)[2]
                     - forward(W1 - E)[2]) / (2 * eps)
print(dW1.float().round(decimals=4))
print((dW1 - num).abs().max().item())
```

OUTPUT

```
tensor([[ 0.1457,  0.5949],
        [-0.1214, -0.4957]])
8.088751890511503e-12
```

Automatic Differentiation

What PyTorch has that numpy doesn't is gradients. Gradient descent needs the gradient of the loss for every weight, which in Logistic Regression we derived by hand. Create a tensor with `requires_grad=True` and PyTorch records everything done with it, then `.backward()` on the loss works the gradients out:

```
import torch
A1 = torch.tensor([[0., 0.], [1., 0.], [0., 1.]])
y = torch.tensor([1., 0., 0.])  # only UNC was close
W2 = torch.zeros(1, 2, requires_grad=True)
b2 = torch.zeros(1, requires_grad=True)
yhat = torch.sigmoid(A1 @ W2.T + b2)[:, 0]
loss = torch.nn.functional.binary_cross_entropy(yhat, y)
loss.backward()  # fills in .grad for W2 and b2
print(round(loss.item(), 4))
print(W2.grad, b2.grad)
```

OUTPUT

```
0.6931
tensor([[0.1667, 0.1667]]) tensor([0.1667])
```

With its weights at zero the output neuron says `0.5` for every game, so the loss is $\ln 2 =$ `0.6931`. Both weight gradients are positive (the $(\hat{y} - y)\,x$ from Logistic Regression, averaged), so gradient descent pushes both weights down, toward the `-4`s we picked by hand for the close-game network in Feed-Forward Networks.

This is called **automatic differentiation** (autograd in PyTorch), and it's **why nobody differentiates a network with millions of weights by hand**.

Gradients on Every Game

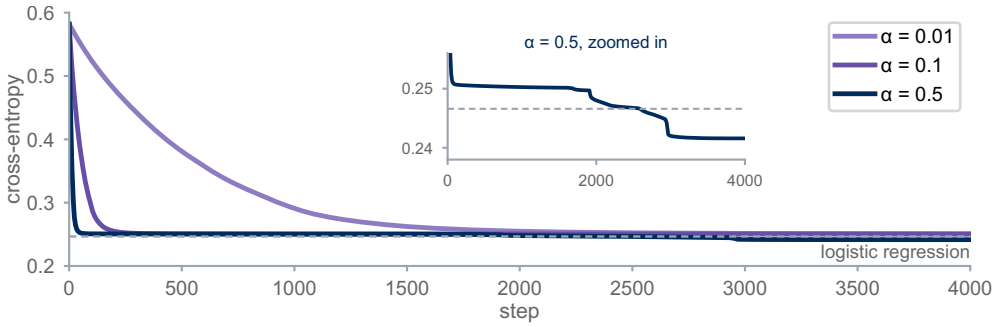
One step of gradient descent with $\alpha = 0.5$ moves all nine weights against their derivatives, and the network's chance of beating Clemson drops from **55%** to **27%**. Unfortunately a network that's only good at Clemson isn't much use, so as in Logistic Regression we average the loss over all **141** games. With the games as rows of X, every line of the backward pass works on all of them at once:

every game's p − y, averaged in

$$dZ_2 = (p - y)/N \qquad \frac{\partial \mathcal{L}}{\partial W_2} = dZ_2^{\top} A_1$$

back through W_2 and the ReLU

$$dZ_1 = dZ_2 W_2 \odot (Z_1 > 0) \qquad \frac{\partial \mathcal{L}}{\partial W_1} = dZ_1^{\top} X$$



As usual, **picking the learning rate is most of the work**. With $\alpha = 0.5$ the average loss falls from **0.58** to **0.25** in about **30** steps, while $\alpha = 0.01$ is still at **0.29** after **1,000**.

Training the Network

Training is a loop of forward pass, backward pass and step:

```
import pandas as pd
import torch
games = pd.read_csv("nd_football.csv")
cols = ["yard_diff", "turnover_diff"]
X = torch.tensor(games[cols].values, dtype=torch.float32)
X = (X - X.mean(dim=0)) / X.std(dim=0)  # standardize
y = torch.tensor(games["win"].values, dtype=torch.float32)
W1 = torch.tensor([[0.5, 0.2], [-0.3, 0.4]])
b1 = torch.tensor([0.0, 0.1])
W2, b2 = torch.tensor([0.6, -0.5]), torch.tensor(0.2)
alpha = 0.5
for step in range(4000):
    Z1 = X @ W1.T + b1                    # forward
    A1 = torch.relu(Z1)
    p = torch.sigmoid(A1 @ W2 + b2)
    dZ2 = (p - y) / len(y)                # backward
    dZ1 = torch.outer(dZ2, W2) * (Z1 > 0)
    W2 = W2 - alpha * A1.T @ dZ2          # one step
    W1 = W1 - alpha * dZ1.T @ X
    b2 = b2 - alpha * dZ2.sum()
    b1 = b1 - alpha * dZ1.sum(dim=0)
L = -(y * p.log() + (1 - y) * (1 - p).log()).mean()
acc = ((p >= 0.5) == (y == 1)).float().mean()
print(round(L.item(), 4), round(acc.item(), 3))
```

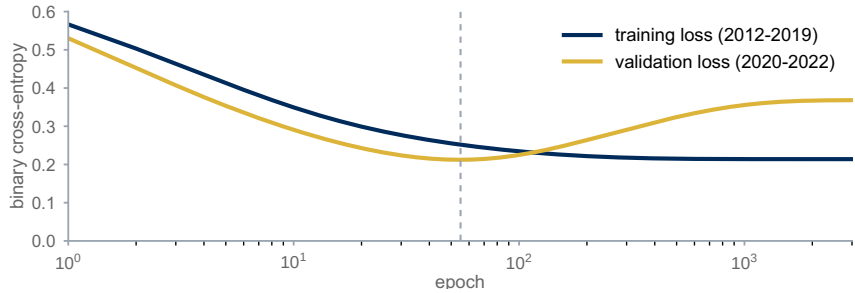
OUTPUT

```
0.2416 0.901
```

After `4,000` steps the loss is `0.242`, under a logistic regression's `0.247` on the same features, but it gets `90.1%` of the games right to `90.8%` (`88.6%` to `90.0%` on unseen games). Two features don't leave a hidden layer much to find.

Learning Curves Over Epochs

Anything fit by gradient descent can be watched while it learns. Here the logistic regression on the `15` box-score stats from Ensemble Methods is fit to the `2012` to `2019` seasons, one step per pass through the training games (a pass is called an **epoch**), and after each epoch we measure the cross-entropy on those seasons and on `2020` to `2022`, which it never trains on:



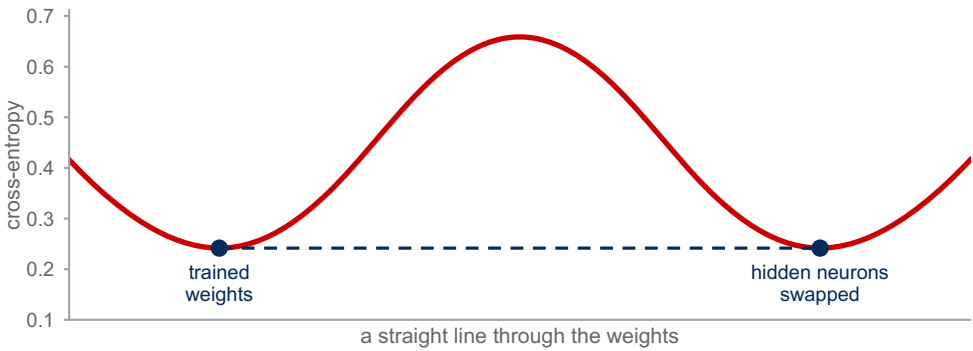
The training loss falls all the way, from `0.57` to `0.21`. The loss on the held-out seasons, called the validation loss, falls with it until epoch `55` (`0.21`), then climbs to `0.37` as the weights keep growing to fit the training games ever more confidently. Past epoch `55`, **every extra epoch makes the model worse on games it hasn't seen**.

The obvious fix is to stop at the bottom of the validation curve, called **early stopping**. For logistic regression it's a curiosity (we could just regularize it). For models with millions of weights, trained epoch after epoch for hours, a plot like this is often the main way to tell if training is going well.

Non-Convex Losses

The α = 0.5 curve on Gradients on Every Game sat on a ledge near 0.25 for over 2,500 steps before dropping, which logistic regression's loss never did. Its loss is **convex**: the straight line between any two points on it lies on or above the curve, which is why every path downhill ended at the same bottom.

Our network's loss isn't. Swapping its two hidden neurons (their rows of W_1 and b_1 and their weights in W_2) gives different weights with exactly the same predictions, and the same loss of 0.2416. Halfway between the two, both hidden neurons become the average of the originals and the loss is 0.66, worse than where we started:



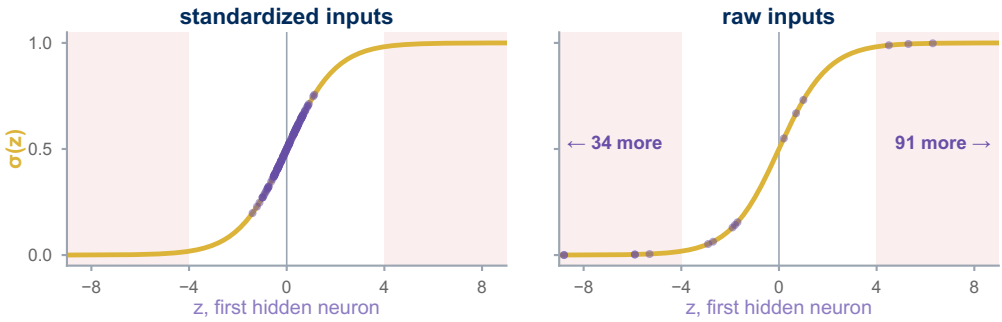
So the loss has more than one bottom, and in general they aren't all equally good. Training from 20 random starting weights ends at losses anywhere from 0.238 to 0.249. **Gradient descent only promises a local minimum**, a point lower than everything right around it.

Saturated Neurons

Getting stuck in a local minimum isn't the only way training can stall. Back on A Small Network we standardized both inputs, so Clemson's 136 yards became 0.44, which looked like housekeeping. Suppose our hidden layer used sigmoids instead of ReLUs and we fed it the raw numbers. With our starting weights Clemson's first hidden neuron gets $z = 0.5(136) + 0.2(3) = 68.6$, and $\sigma(68.6)$ is 1 to as many digits as anyone would write down.

The sigmoid's local derivative $\sigma(z)(1 - \sigma(z))$ is 0.25 at $z = 0$ and falls off quickly on both sides. At 68.6 it's about 1.6×10^{-30}, so whatever gradient reaches this neuron gets multiplied by essentially nothing on its way back to w_1. A neuron stuck out on one of the flat ends like this is **saturated**, and **a saturated neuron barely learns**.

Standardizing the Inputs

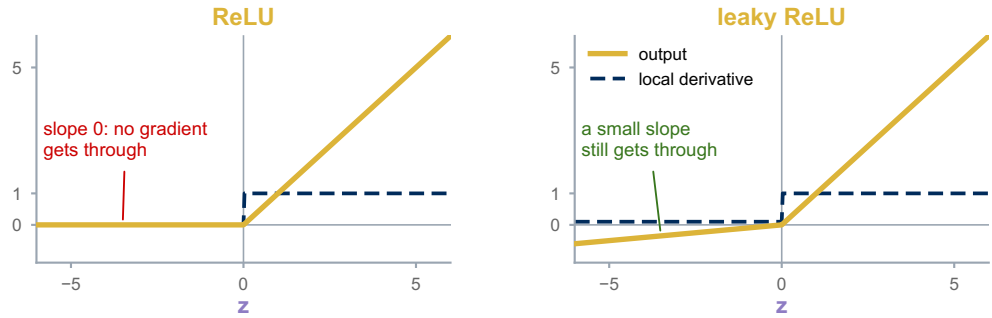


Pictured above is every game's starting `z` for that neuron. Standardized, all `141` land on the steep part of the curve, between `-1.40` and `1.13`. Raw, `133` of them land past `±4`, where the local derivative is under `0.018`.

Trained the same way as before (`4,000` steps with α = `0.5`), the sigmoid network on standardized inputs gets to a loss of `0.244` and `91.5%` of the games right. On raw inputs it stops at `0.441` and `79.4%`, hardly better than always guessing "win" (`75.2%`). So **standardize the inputs**, which puts them around `0` with a spread of `1`, where the pre-activations start on the steep part of the curve. It's also one reason hidden layers use ReLU, whose slope for positive `z` is always `1`.

Dead Neurons

ReLU never saturates on the positive side, but its local derivative is exactly `0` for negative inputs, which is worse. A ReLU neuron whose pre-activation is negative on every game outputs `0` for all of them and passes no gradient back, so gradient descent never changes its weights again. That's a **dead neuron**. A big step can kill one, by pushing its bias so far negative that no game switches it on. Standardizing helps here too: with inputs averaging `0`, a neuron's `z` averages its bias over the games, so a neuron that starts with a bias of `0` can't start out negative on all of them.



The usual fix is the **leaky ReLU**, `max(0.01z, z)`, which gives the negative side a small slope (drawn bigger above, so it shows). A neuron that's negative on every game still gets a little gradient and can come back. In practice plain ReLU works well enough that it's still the most common choice, and a few dead neurons in a wide layer don't matter much.

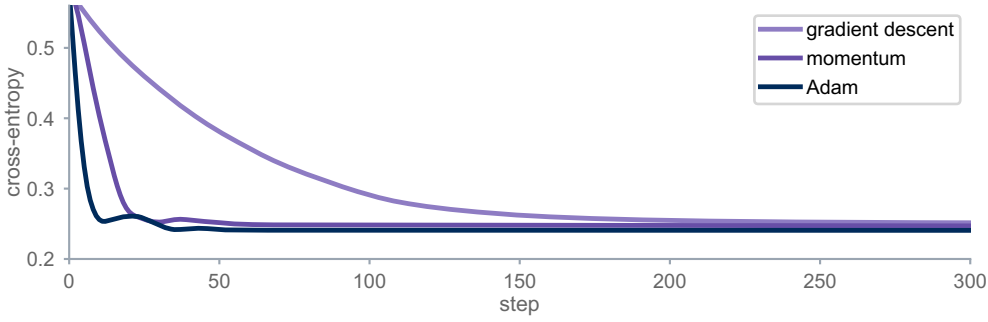
Momentum and Adam

The other problem is getting stuck on ledges and in shallow dips. A common fix is **momentum**, stepping along a running total of recent gradients instead of just this one, so the weights keep rolling the way they've been going:

β = 0.9 lets roughly the last ten gradients count

$$v \leftarrow \beta\, v + \frac{\partial \mathcal{L}}{\partial w} \qquad w \leftarrow w - \alpha\, v$$

The most common choice today is **Adam** (Adaptive Moment Estimation, from Diederik Kingma and Jimmy Ba in `2014`). It keeps a running average of the gradient like momentum, plus a running average of its square, and divides one by the square root of the other, so every weight gets its own step size.



On our network, all three with α = `0.1`, plain gradient descent is still at `0.251` after `300` steps and momentum at `0.247`, while Adam gets below `0.245` in `32` steps and ends at `0.241`, slightly below where plain gradient descent got in `4,000` steps at α = `0.5`.