Author: Benjamin Smidt
Created: September 11, 2022
Last Updated: September 13, 2022

Assignment 2: Two Layer Neural Net

Note to reader.

This is my work for assignment two of Michigan's course EECS 498: Deep Learning for Computer Vision. This document is meant to be used as a reference, explanation, and resource for the assignment, not necessarily a comprehensive overview of Neural Networks. Furthermore, this document is well-informed and thoroughly researched but may not be perfect. If there's a typo or a correction needs to be made, feel free to email me at benjamin.smidt@utexas.edu so I can fix it. Thank you! I hope you find this document helpful.

1 Functions

1.1 NN Forward Pass

For this function we simply want to compute the forward pass through our two layer, fully connected network. Our input X has shape NxD, our first weight matrix W_1 shape DxH, first bias vector b_1 length H, second weight matrix W_2 shape HxC, and second bias vector b_2 length C. N is the number of examples, D the dimension of each example, H the number of layers in our hidden layer, and C the number of classes to classify our examples into.

Moving forward through the network is pretty simply. We begin by matrix multiplying XW_1 yielding the matrix H (for hidden) of shape NxH. Each column represents the computation that a given hidden layer does for *every single example.* Said differently, each row indicates the computations for *all the hidden layers* for a given example in N. Because of this we add our bias vector b_1 to every single row in our H matrix (using broadcasting).

Next, we compute our ReLU function using a mask and pass H off to W_2 to produce our final S matrix (for scores). Similar to our hidden layer computation, we matrix multiply HW_2 and add b_2 using broadcasting. This final S matrix has shape NxC as you would expect. Each example has a score for each class.

1.2 NN Forward Backward

Loss

This function has two parts. First we compute the loss from our forward pass (previous function). Then we find the gradient using backpropagation. The loss is fairly simply so the first part will be brief. For this neural network we're using softmax as our final classifier and adding L2 regularization for every weight matrix (just two, W_1 and W_2). As such, our loss function is as follows.

$$L = 1/N \sum_{i=1}^{N} -log(\frac{e^{f_{y_i}}}{\sum_{j=1}^{C} e^{f_j}}) + \lambda \sum_{k}^{D} \sum_{l}^{C} (w_1)_{l,k}^2 + \lambda \sum_{k}^{D} \sum_{l}^{C} (w_2)_{l,k}^2 \qquad (1)$$

For more information about Softmax and regularization see A2-Softmax, the second part of the "Linear Classifiers" assignment in A2. The *important detail* here is that I did normalize the output scores that we performed the data loss on (using softmax) to prevent numeric instability. For more info on numeric instability check out CS 231N: Linear Classifiers

Gradient

Now time for the fun part, backpropagating the gradient through our network! This section may be quite long and tedious because I want to work through the computation from beginning to end. Before we begin, remember back propagation takes advantage of the chain rule. Recall that a neural network is really a chain of functions, each one using the output of the previous function as the input of its own until we reach the end of our network and make a prediction.

What we're actually doing in our backpropagation algorithm is finding the gradient or jacobian with respect to a variable (often a weight matrix W) for a modular portion of the network, a single function. Since the chain rule simply multiplies the derivatives (or gradients or jacobians depending on your circumstance) of each function in the chain, we're left with a nice, modular method of computing the gradient from beginning to end by just multiplying gradients.

That was wordy so let's clear things up and first walk through the forward pass of computing the loss function of the two layer neural network in A2 using a single example x_i. You may wonder why I'm using a lot of row vectors. This is because when we generalize this process to N examples, each row will be an example. So for now many vectors are 1xK (K just represents some dimension, not a specific dimension used in the neural network) but in practice they will be NxK.

x_i: 1xD, W_1: DxH, b_1: 1xH

$$H_0 = x_i W_1 + b_1 \tag{2}$$

H: 1xH

$$H = max(0, H_0) = max(0, h_{0_i}) \; \forall \, h_{0_i} \in H_0 \tag{3}$$

W_2: HxC, b_2: 1xC

$$S = HW_2 + b_2 \tag{4}$$

S: 1xC

$$L_i = -log(\frac{e^{S_y}}{\sum_{j=1}^{C} e^{S_j}}) + \lambda \sum_{k}^{D} \sum_{l}^{C} (w_1)_{l,k}^2 + \lambda \sum_{k}^{D} \sum_{l}^{C} (w_2)_{l,k}^2 \tag{5}$$

This is our forward propagation through the network for our loss function. If we wanted to find the prediction instead of the loss, we'd replace our last equation $L_i = ...$ with

$$x_i Prediction = max(\frac{e^{S_j}}{\sum_{j=1}^{C} e^{S_j}} \; \forall \, j \in C) \tag{6}$$

However, we're trying to compute the gradient of the loss function L so we need to focus on the output of L_i. Before moving on, I do quickly want to point out that the graphical representation of this network looks very different (at least to me) from the algebraic plug and chug we see here. They are the same though. The number of nodes in our first layer, our hidden layer, is defined by the number of columns H in W_1. Similarly, the number of nodes in our second layer, which passes scores off to our softmax function, is defined by the number of columns in W_2. As you would expect, the number of nodes is C since we're trying to classify our data into 1 of C categories.

Alright, let's get backpropagate through this small network. Remember, because of the chain rule we're just multiply gradients together. For instance, if we want to find $\frac{\partial L_i}{\partial W_2}$ we can just compute $\frac{\partial L_i}{\partial S}\frac{\partial S}{\partial W_2}$. This is much easier to do rather than substituting S for $HW_2 + b_2$ in the equation for L_i and then finding $\frac{\partial L_i}{\partial W_2}$ directly. Let's begin with $\frac{\partial L_i}{\partial S_y}$ and $\frac{\partial L_i}{\partial S_j}$

$$\frac{\partial L_i}{\partial S_y} = \frac{\partial}{\partial S_y} - log(\frac{e^{S_y}}{\sum_{j=1}^{C} e^{S_j}}) + \frac{\partial}{\partial S_y}\lambda \sum_{k}^{D}\sum_{l}^{C}(w_1)_{l,k}^2 + \frac{\partial}{\partial S_y}\lambda \sum_{k}^{D}\sum_{l}^{C}(w_2)_{l,k}^2 \tag{7}$$

$$\frac{\partial L_i}{\partial S_y} = \frac{\partial}{\partial S_y} - log(\frac{e^{S_y}}{\sum_{j=1}^{C} e^{S_j}}) \tag{8}$$

$$\frac{\partial L_i}{\partial S_y} = \frac{\partial}{\partial S_y} log(\sum_{j=1}^{C} e^{S_j}) - \frac{\partial}{\partial S_y} S_y \tag{9}$$

$$\frac{\partial L_i}{\partial S_y} = (\frac{1}{\sum_{j=1}^{C} e^{S_j}})\frac{\partial}{\partial S_y}(e^{S_1} + ... + e^{S_y} + ... + e^{S_C}) - 1 \tag{10}$$

$$\frac{\partial L_i}{\partial S_y} = (\frac{e^{S_y}}{\sum_{j=1}^{C} e^{S_j}}) - 1 \tag{11}$$

$$\frac{\partial L_i}{\partial S_j} = \frac{\partial}{\partial S_j} - log(\frac{e^{S_y}}{\sum_{j=1}^{C} e^{S_j}}) + \frac{\partial}{\partial S_j}\lambda \sum_{k}^{D}\sum_{l}^{C}(w_1)_{l,k}^2 + \frac{\partial}{\partial S_j}\lambda \sum_{k}^{D}\sum_{l}^{C}(w_2)_{l,k}^2 \tag{12}$$

$$\frac{\partial L_i}{\partial S_j} = \frac{\partial}{\partial S_j} - log(\frac{e^{S_y}}{\sum_{j=1}^{C} e^{S_j}}) \tag{13}$$

$$\frac{\partial L_i}{\partial S_j} = \frac{\partial}{\partial S_j} log(\sum_{j=1}^{C} e^{S_j}) - \frac{\partial}{\partial S_j} S_y \tag{14}$$

$$\frac{\partial L_i}{\partial S_j} = (\frac{1}{\sum_{j=1}^{C} e^{S_j}})\frac{\partial}{\partial S_j}(e^{S_1} + ... + e^{S_j} + ... + e^{S_C}) \tag{15}$$

$$\frac{\partial L_i}{\partial S_j} = (\frac{e^{S_j}}{\sum_{j=1}^{C} e^{S_j}}) \tag{16}$$

This is the gradient of the softmax function. For more information on it, check out Assignment 2: Softmax where I explain it more in depth. Notice that we have different derivatives depending on if the score is the correct score or not. We will have to program this but for now, let's store both $\frac{\partial L_i}{\partial S_y}$ and $\frac{\partial L_i}{\partial S_y}$ in a single variable $\frac{\partial L_i}{\partial S}$ and continue backward through the network (and no I haven't forgotten the regularization term, I want to add it later).

$$\frac{\partial S}{\partial W_2} = \frac{\partial}{\partial W_2} HW_2 + b_2 \tag{17}$$

$$\frac{\partial S}{\partial W_2} = H \tag{18}$$

$$\frac{\partial L_i}{\partial W_2} = \frac{\partial L_i}{\partial S}\frac{\partial S}{\partial W_2} \tag{19}$$

$$\frac{\partial L_i}{\partial W_2} = \frac{\partial L_i}{\partial S} H \tag{20}$$

If you're attentive you may be looking at equation 20 with some skepticism. And you would certainly be correct in doing so. $\frac{\partial L_i}{\partial S}$ has shape 1xC since S has shape 1xC (actually it's technically just a column vector of length C but thinking about it as a row vector will generalize to more examples much more intuitively). H has shape 1xH. Clearly this matrix multiplication won't work. So what gives?

This will be a recurring theme in backpropagation of needing to analyze shapes. Remember that $\frac{\partial y}{\partial Z}$, where y is a scalar and Z is just some variable (scalar, vector, matrix, tensor, etc.), will *always* have the same shape as Z. So $\frac{\partial L_i}{\partial W_2}$ has the same shape as W_2 since L_i is (always) a scalar. Since W_2 has shape HxC, we actually need to take the outer product between $\frac{\partial L_i}{\partial S}$, shape 1xC, and H, shape 1xH. The following results (note that $H.T$ is the transpose of H).

$$\frac{\partial L_i}{\partial W_2} = H.T \frac{\partial L_i}{\partial S} \tag{21}$$

Although it maybe unintuitive at first, thinking about what these gradients actually mean makes everything fall into place (at least for me). Each scalar in the vector $\frac{\partial L_i}{\partial S}$ indicates the the proportional change in L_i given a small change in S. Similarly, each scalar in the the vector $\frac{\partial S}{\partial W_2}$ indicates the proportional change in S given a small change W_2.

Recall that we found S using the function $S = HW_2 + b_2$. You can see from this operation that every single value in the first column of W_2 scales the first scalar in S by the value it multiplies by in the matrix multiply. For instance, $w_{2,1}$ (first row, second column) scales the s_1 by h_2; $w_{4,1}$ scales s_1 by h_4; $w_{3,2}$ scales s_2 by h_3. Thus, it makes sense when computing the gradient to take the outer product between H and S since each combination of values in H and S can be traced back to a different scalar in W_2 when were simply computing S.

Whew! That was a mouthful. If you don't understand, I'd draw out the matrices and trace through them to see what I'm talking about. If that doesn't work, checkout this 10 minute segment of Justin Johnson explaining backpropagation of matrix multiplication in his lecture. Anyhow, let's continue.

$$\frac{\partial L_i}{\partial H} = \frac{\partial L_i}{\partial S}\frac{\partial S}{\partial H} \tag{22}$$

$$\frac{\partial S}{\partial H} = \frac{\partial}{\partial H} HW_2 + b_2 \tag{23}$$

$$\frac{\partial S}{\partial H} = W_2 \tag{24}$$

$$\frac{\partial L_i}{\partial H} = \frac{\partial L_i}{\partial S} W_2.T \tag{25}$$

Again, looking that the shapes, you should be able to verify equation 25. $\frac{\partial L_i}{\partial H}$ has shape 1xH, $\frac{\partial L_i}{\partial S}$ 1xC, and $W_2.T$ shape CxH. Okay, onto the ReLU function.

$$\frac{\partial L_i}{\partial H_0} = \frac{\partial L_i}{\partial S}\frac{\partial S}{\partial H}\frac{\partial H}{\partial H_0} \tag{26}$$

$$\frac{\partial H}{\partial H_0} = \mathbb{1}(H_0 > 0) = \mathbb{1}(h_{0_i} > 0)\ \forall\, h_{0_i} \in H_0 \tag{27}$$

$$\frac{\partial L_i}{\partial H_0} = (\frac{\partial L_i}{\partial S}\, W_2.T) * \mathbb{1}(H_0 > 0) \tag{28}$$

Okay if you're not confused here I'd be slightly surprised. First thing, in our forward pass we computed $H = max(0, H_0)$, which just means for every element in the matrix, change it to 0 if it's less than 0 and leave it alone otherwise. This is also known as the ReLU function. It's not difficult to show that the derivative for a given value is 1 if that value is > 0 and 0 if that value is 0. Let $f = max(0, x)$ be the ReLU function.

$$f = \begin{cases} 0 \text{ if } x \leq 0 \\ x \text{ if } x > 0 \end{cases} \tag{29}$$

$$\frac{df}{dx} = \begin{cases} 0 \text{ if } x \leq 0 \\ 1 \text{ if } x > 0 \end{cases} \tag{30}$$

Thus, we can form a matrix that is of equivalent size to the matrix passed through our ReLU function. By filling it with 0's where the original input is ≤ 0 and 1's where the original input is > 0, we can perform a simple element wise multiplication between this matrix (which is a representation of $(\frac{\partial L_i}{\partial H})$ and $\frac{\partial L_i}{\partial H}$.

It is important to note that this is *not* the explicit jacobian but an implicit representation of the jacobian. Said differently, this is not the mathematically rigorous way to write the gradient but it is an equivalent and simpler way that proves computationallly more efficient. I've actually been using an implicit representation the whole way through but I didn't want to muddle the explanation and make it more confusing than it already is. For more information on the gradient/jacobian of the ReLU function, see this 6 segment of Justin Johnson explaining it in his lecture.

Okay, almost finished I promise.

$$\frac{\partial L_i}{\partial x_i} = \frac{\partial L_i}{\partial S}\frac{\partial S}{\partial H}\frac{\partial H}{\partial H_0}\frac{\partial H_0}{\partial x_i} \tag{31}$$

$$\frac{\partial H_0}{\partial x_i} = \frac{\partial}{\partial x_i} x_i W_1 + b_1 \tag{32}$$

$$\frac{\partial H_0}{\partial x_i} = W_1 \tag{33}$$

$$\frac{\partial L_i}{\partial x_i} = [(\frac{\partial L_i}{\partial S} W_2.T) * \mathbb{1}(H_0 > 0)]W_1.T \tag{34}$$

$$\frac{\partial L_i}{\partial W_1} = \frac{\partial L_i}{\partial S}\frac{\partial S}{\partial H}\frac{\partial H}{\partial H_0}\frac{\partial H_0}{\partial W_1} \tag{35}$$

$$\frac{\partial H_0}{\partial W_1} = \frac{\partial}{\partial W_1} x_i W_1 + b_1 \tag{36}$$

$$\frac{\partial H_0}{\partial W_1} = x_i \tag{37}$$

$$\frac{\partial L_i}{\partial W_1} = x_i.T[(\frac{\partial L_i}{\partial S} W_2.T) * \mathbb{1}(H_0 > 0)] \tag{38}$$

And we're ... almost done. You may have noticed we never added the regulatization term to our losses! Let's to do that quickly. I'll write our original loss function for reference and then add the loss for the regularization terms (Note that "x += 1" means "$x = x + 1$")

$$L = 1/N \sum_{i=1}^{N} -log(\frac{e^{f_{y_i}}}{\sum_{j=1}^{C} e^{f_j}}) + \lambda \sum_{k}^{D} \sum_{l}^{C} (w_1)_{l,k}^2 + \lambda \sum_{k}^{D} \sum_{l}^{C} (w_2)_{l,k}^2 \tag{39}$$

$$\frac{\partial L_i}{\partial W_1} += \lambda * 2 * W_1 \tag{40}$$

$$\frac{\partial L_i}{\partial W_2} += \lambda * 2 * W_2 \tag{41}$$

All I did here was *add* the gradient with respect to L_i to the gradient we originally calculated.

We're done! Hopefully this helped you understand how backpropagation works, how it's computed, etc. While we walked through it using only one example x_i, it should be easy for you to generalize to N examples, especially considering I worked through it with all those row vectors (1xK)

1.3 NN Train

In this function we're just implementing all our hardwork. We use stochastic gradient descent (SGD) (as you pretty much always should with deep learning) to update *each* of our alterable parameters. For this network that would be W_1, b_1, W_2, and b_2.

A few things are important to point out here. The first is *num-iters*, an input variable that tells our function how many total steps to take. You can see the default is 100. At the beginning of the function we also compute *iterations-per-epoch*, a variable that tells how many iterations it takes us to reach a single epoch (provided you know *batch-size*). What are epochs? It's the number of passes through the entire training dataset that your algorithm has completed. Since, we're using SGD, one pass through the entire training dataset would be *batch-size* * *iterations-per-epoch*.

Lastly, you may also notice we're storing and returning important data about the training process. In particular, the loss history, training accuracy history, and validation accuracy history provide us important information about training our neural network. We benchmark these at each epoch, a natural cycle to use since after each epoch that network has seen the entire training dataset again.

1.4 NN Predict

This one is pretty easy. You simply compute your forward pass until you get your final output from softmax. Then, for each example, just return the (column) index of the maximum value. Don't forget to alter your scores for numerical stability!. I know this is pretty vague but you should be able to figure it out pretty quickly if you've gotten this far.

1.5 NN Get Search Params and Find Best Net

Instead of going over the actual function, I'll instead explain the larger themes at work here regarding why we need to tune parameters, how they affect the network, and trouble shooting. You can see with the default parameters our network performs pretty bad: a measly 9.77% validation accuracy. Let's see what parameters may need to be tuned.

We begin by using some basic debugging strategies. The most obvious is plotting the loss history and classification accuracy. As you can see, they're all over the place. The loss just spreads out but doesn't seem to be getting any better and the classification accuracy is oscillating like crazy! We expect the accuracy to improve tremendously at first before flattening out asymptotically toward at some accuracy. But this isn't what happens at all.

We also expect there to be at least some gap between the training and validation accuracy. This is because we want to find where the parameters that *slightly* overfit the data to make sure we juice the dataset for all the information it can give us. If the training and validation accuracy are the same, it suggests we might be underfitting our data leaving some important information on the table. Of course, we want to find the goldilocks area between underfitting and overfitting but that first requires us to overfit and then scale backward to find that goldilocks place.

Furthermore, when we visualize the weights learned in the first layer, which we saw in our SVM and Softmax assignments can be interpreted as some kind of template, all we get is noise. This might suggest that our learning rate wasn't large enough and thus our gradient descent didn't move our parameters to any meaningful place.

Our first changeable parameter is the capacity, which is the size of our hidden layer. To be clear, we aren't changing the number of layers, we're changing the number of nodes in our hidden layer. For instance, if a single layer has eight nodes, adding eight more nodes would double the capacity of that layer. More nodes allow the network to learn more complex patterns. Too many nodes and the network will use that extra capacity to overfit the training data. Not enough nodes and the network won't have enough capacity to learn all the generalizable information that the training data has to offer.

Our next parameter is regularization, something we've run into before. Regularization, as we saw particularly with SVM's, is important. However, if the regularization coefficient (λ) is too large, the model won't have enough flexibility to learn what the dataset has to offer. It will focus too much on minimizing the regularization loss and not enough on minimizing the data loss.

Our last parameter is the learning rate. You should be familiar with this but the essential idea is that our gradient descent needs to take meaningful steps. Too small of a step size (learning rate) and we'll learn nothing. Too large of a step size and we'll either run right past our minimum (never to find it again) or waste a bunch of computation oscillating around the minimum. With too large of a step size, we don't have enough fine motor control to step close to or onto the minimum and instead continually walk over it since our step size it too large.

2 References

1. CS 231N: Neural Networks