Ryan George
Deep Learning hw3
April 1, 2018

Question 1

1. In tflecture1 2.py, make sure you understand what tf.Variable, tf.Session() do. Explain what init = tf.global variables initializer() and init.run() do, and how this code differs from tflecture1 1.py.

> The command `tf.global_variables_initializer` creates the placeholder variables (X, Y) that are contained in the graph. The command `init.run` actually runs the instructions held in that graph. In `tflecture1_1.py`, by contrast, the variables are initiated with separate runs of `_.initializer.run()`.

2. Compare tflecture1 3.py with tflecture1 4.py. What do line 32, line 78-87, line 94-102 in tflecture1 3.py do? What does line 57 in tflecture1 4.py do? Briefly Explain. Make sure you have installed sklearn before testing with the code.

> Line 32 in `tflecture1_3.py` creates a point in the TF graph that solves the normal equations. This effectively performs linear regression.
>
> Lines 78-87 attempt to solve the same problem via gradient descent. We start with a randomly initiated θ variable, compute the mean standard error, line of gradient descent, and an updated estimate of θ after gradient descent. No code is run in these lines; they're just setting up the TF tree to run later
>
> Line 94-102 actually runs the code written in previous lines in Tensorflow (C++ implementation)
>
> Line 57 in `tflecture1_4.py` programmatically calculates the gradient on the mean squared error at point *theta*.

3. Based on what tflecture1 4.py does, implement the prob1.py TO DO part using the TensorFlow optimizer. Is the output different from tflecture1 4.py? Save your code as prob1.py. Hint: You might only need two lines to set up the optimizer and use the optimizer to minimize the MSE.

We do get the same answer:

$ python3 tflecture1_4.py	**$ python3 prob1.py**
Epoch 0 MSE = 9.161543	Epoch 0 MSE = 9.161543
Epoch 100 MSE = 0.7145006	Epoch 100 MSE = 0.7145006
Epoch 200 MSE = 0.56670463	Epoch 200 MSE = 0.56670463
Epoch 300 MSE = 0.5555716	Epoch 300 MSE = 0.5555716
Epoch 400 MSE = 0.5488117	Epoch 400 MSE = 0.5488117
Epoch 500 MSE = 0.5436362	Epoch 500 MSE = 0.5436362
Epoch 600 MSE = 0.53962916	Epoch 600 MSE = 0.53962916
Epoch 700 MSE = 0.53650916	Epoch 700 MSE = 0.53650916
Epoch 800 MSE = 0.5340678	Epoch 800 MSE = 0.5340678
Epoch 900 MSE = 0.53214717	Epoch 900 MSE = 0.53214717

Question 2

1. It's tempting to use gradient descent to try to learn good values for hyper-parameters such as λ and η. Can you think of an obstacle to using gradient descent to determine λ? Can you think of an obstacle to using gradient descent to determine η?

The difficulty with using gradient descent in order to find hyper-parameters is that more hyper-hyper-parameters would need to be set in order to find in order to find those variables.

One obstacle to computing regularization parameter λ specifically is that the network would need to run for many epochs before we knew whether the parameter led to optimal performance.

One obstacle to computing learning rate η specifically is that some other learning rate would need to be chosen in order to initialize the search process.

2. L2 regularization sometimes automatically gives us something similar to the new approach to weight initialization (i.e., we initialize the weights as Gaussian random variables with mean 0 and standard deviation 1/√nin where nin is the number inputs to a neuron). Suppose we are using the old approach to weight initialization (i.e., we initialize the weights as Gaussian random variables with mean 0 and standard deviation 1). Sketch a heuristic argument that:

(a) So long as λ is not too small, the penalty in the first few epochs of training will have cost:

$$C_0 = C + \frac{\lambda}{2n} \sum_w w^2$$

In this case, C (if we are using cross-entropy) is a sum of variables less than one. Supposing we drew weights from $N(0, 1)$, many of these weights' squares will approach or exceed 1. In addition, there will be far more weights than examples, so the regularization term will dominate. It will mainly have the effect of reducing the absolute weights.

(b) Provided ηλ ≪ n, the backpropogation equations for updates is:

weights will decay by a factor of exp(−ηλ/m) per epoch.

(c) Supposing λ is not too large, the weight decay will tail off when the weights are down to a size
around 1/ n, where n is the total number of weights in the network.

3. Modify your code for problem 1.3 to use an adaptive learning rate using AdagradOptimizer in Tensor- Flow. Record your output and save your code as prob2.py. Comment on how you chose any of the parameters.

I chose the `initial_accumulator_value` as a value below 1 but not as small as the learning rate.

Question 3

1. When discussing the vanishing gradient problem, we made use of the fact that |σ′(z)| < 1/4. Suppose we use a different activation function, one whose derivative could be much larger. Would that help us avoid the unstable gradient problem? (Nielsen book, chapter 5)

> This would not help, because the activation function would still need to have end behavior such that the activation function looks flat-ish towards the end. Hyperbolic behavior, which would be needed to avoid the vanishing gradient problem, would lead to huge unstability

2. Consider the product |wσ′(wa + b)| where σ is the sigmoid function. Suppose |wσ′(wa + b)| ≥ 1.
(a) Argue that this can only ever occur if |w| ≥ 4.

> We know that $|\sigma'(\cdot)| < 1/4$.
> Thus $|w\sigma'(wa+b)| \leq |w||\sigma'(wa+b)| \leq |w|/4 \leq 1$, which implies that $|w| > 1$ ..

(b) Supposing that |w| ≥ 4, consider the set of input activations a for which |wσ′(wa + b)| ≥ 1. Show that the set of a satisfying that constraint can range over an interval no greater in width than 2 |w|(1 + 1 − 4/|w|) |w|ln 2 −1 .

> We know that $\sigma'(wx+b) = \sigma(wx+b) - \sigma(wx+b)^2$. Thus:
> $$\begin{aligned} |w\sigma'(wx+b)| &= |w(\sigma(wx+b) - \sigma(wx+b)^2)| \\ &= |w||(\sigma(wx+b) - \sigma(wx+b)^2)| \\ &= |w||\frac{1}{1+e^{-wx-b}} - \frac{1}{(1+e^{-wx-b})^2}| \\ &= |w||\frac{e^{-wx-b}}{(1+e^{-wx-b})^2}| \\ &= |w||\frac{e^{-wx-b}}{1+2e^{-wx-b}+e^{-2wx-2b}}| \\ &\geq 1 \end{aligned}$$
> This implies:
> $$\begin{aligned} |w| \geq |\frac{1+2e^{-wx-b}+e^{-2wx-2b}}{e^{-wx-b}}| &= |\frac{1}{e^{-wx-b}} + \frac{2e^{-wx-b}}{e^{-wx-b}} + \frac{e^{-2wx-2b}}{e^{-wx-b}}| \\ &= |e^{wx+b} + 2 + e^{-wx+b}| \\ &= e^{wx+b} + 2 + e^{-wx+b} \text{ (since all values are positive)} \end{aligned}$$
> So then:
> $$0 \geq e^{wx+b} + e^{-wx-b} + 2 - |w| = (e^{wx+b})^2 + 1 + (2-|w|)e^{wx+b}$$
> If we treat this is a quadratic equation in e^{wx+b} , we get bounds/solutions:
> $$\begin{aligned} e^{wx+b} &\leq \frac{-(2-|w|) \pm \sqrt{(2-|w|)^2 - 4(1)(1)}}{2} \\ &= \frac{-2+|w| \pm \sqrt{4-4|w|+|w|^2-4}}{2} \end{aligned}$$

$$= \frac{-2+|w|\pm|w|\sqrt{-4/|w|+1}}{2}$$
$$= \frac{|w|(1\pm\sqrt{1-4/|w|})}{2} - 1$$

Since we're concerned with the maximum range, we'll consider only the "+" scenario:

$$e^{wx+b} \leq \frac{|w|(1+\sqrt{1-4/|w|})}{2} - 1$$
$$wx + b \leq ln\left(\frac{|w|(1+\sqrt{1-4/|w|})}{2} - 1\right)$$

Since we're looking for a range of x, we can drop b:

$$wx \leq ln\left(\frac{|w|(1+\sqrt{1-4/|w|})}{2} - 1\right)$$

So we conclude

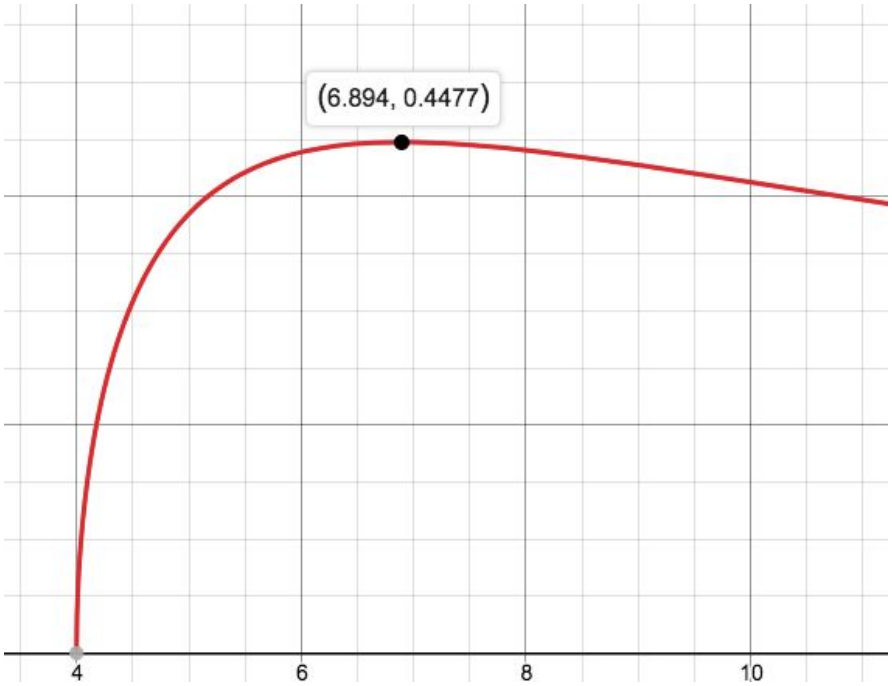
$$x \leq \frac{1}{|w|} ln\left(\frac{|w|(1+\sqrt{1-4/|w|})}{2} - 1\right)$$

But we have to consider the positive and negative scenarios, so the range will be twice that:

$$range(x) = \frac{2}{|w|} ln\left(\frac{|w|(1+\sqrt{1-4/|w|})}{2} - 1\right)$$

(c) Show numerically that the above expression bounding the width of the range is greatest at |w| ≈ 6.9, where it takes a value ≈ 0.45. This demonstrates that even if everything lines up just perfectly, we still have a fairly narrow range of input activations which can avoid the vanishing gradient problem.

Graphing in desmos, we find that the maximum of the function is a=.4477, at w=6.894



3. Identity neuron: Consider a neuron with a single input, x, a corresponding weight w1, a bias b, and a weight w2 on the output. Show that by choosing the weights and bias appropriately, we can ensure w2σ(w1x + b) ≈ x for x ∈ [0, 1], where σ is the sigmoid function. Such a neuron can thus be used as a kind of identity neuron, that is, a neuron whose output is the same (up to rescaling by a weight factor) as its input.

√
2

Hint: It helps to rewrite x = 1/2 + ∆, to assume w1 is small, and to use a Taylor series expansion in w1 ∆.

The Taylor series of a function f(x) at a is the power series

f(a)+ f′(a)(x−a)+ f″(a)(x−a)2 + f‴(a)(x−a)3 +⋯ . 1! 2! 3!

We begin with:

$$f(x) = w_2\sigma(w_1 x + b) = w_2\sigma(w_1(1/2+\Delta) + b)$$ (substituting $1/2+\Delta$ for x)

Then $f'(\Delta) = w_2\sigma'(w_1(1/2+\Delta) + b)w_1\Delta$

And $f''(\Delta) = w_2\sigma''(w_1(1/2+\Delta) + b)w_1{}^2\Delta^2 + w_1 w_2\sigma'(w_1(1/2+\Delta) + b)$

A Taylor series expansion of $f(x)$ at $1/2+\Delta$ yields:

$$f(x) = f(1/2+\Delta) + \frac{f'(1/2+\Delta)}{1}(x-1/2-\Delta) + \frac{f''(1/2+\Delta)}{2}(x-1/2-\Delta)^2 + \ldots$$

$$= w_2\sigma(w_1(1/2+\Delta) + b) + w_2\sigma'(w_1(1/2+\Delta) + b)w_1\Delta\cdot(x-1/2-\Delta) + w_2\sigma''(w_1(1/2+\Delta) + b)w_1{}^2\Delta^2\Delta) + w_1 w_2\sigma'(w_1(1/2 ++ b) + \ldots$$

(Uncle)

4. Recall the momentum-based gradient descent method we discussed in class where the parameter μ controls the amount of friction in the system. What would go wrong if we used μ > 1? What would go wrong if we used μ < 0?

If we used $\mu > 1$ for momentum, then earlier gradients would become *more influential* in subsequent update steps. In the n^{th} iteration, the initial gradient θ' would have approximate weight $\theta'(\mu^n)$, which will be large when n is large. If $\mu < 0$, then we would go in the *opposite direction* as previous gradients. This would actually make the "wandering" effect worse -- and that's exactly what we're trying to avoid.

5. Modify your code for problem 2.3 to implement momentum using MomentumOptimizer and save your code as prob3 a.py. Then modify the code to implement momentum with an adaptive learning rate by using the AdamOptimizer, and save your code as prob3 b.py. Record your outputs for both cases, and compare to your results from problems 1.3 and 2.3. Comment on how you chose any of the tuning parameters.

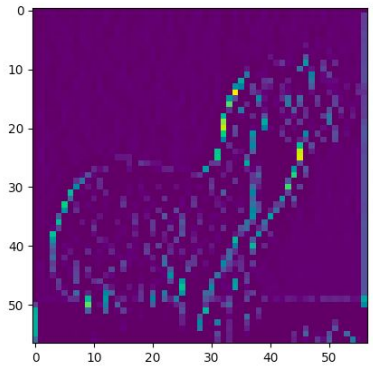
For MomentumOptimizer, I chose a small momentum parameter μ to keep the avoid a large departure from our previous work.

For AdamOptimizer, I chose β_1 and β_2 such that $(1-\beta_2) < (1-\beta_1)$ to make sure that the first. I also kept $(1-\beta_2)$ small. ε is chose to be very small but still numerically representable.

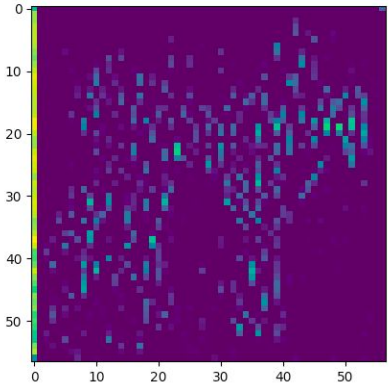
Question 4

2. Reading out from a layer Write code that runs a Tensorflow session on one of the sample images to extract the output of the first convolutional layer. What is its output shape? In this layer, readout the output of one of the 96 57×57 arrays. Plot this using matplotlib's imshow or a similar matrix-to-image function for a couple of sample images and include them in your writeup.

The first convolutional layer shows some of the outlines of the laska:



The shape of this output is a 57x57 array showing the loose outline of the laska.
For the poodle, the output is a looser outline with some edge effects on the left side:

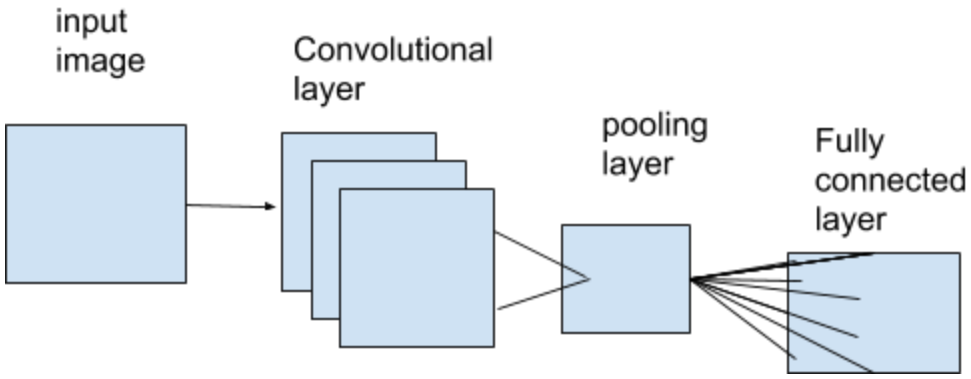


3. Write code that extracts the output of the final layer? Write down the dimensions.

The final layer is a 1x1000 vector.

4. Backpropagation in a convolutional network: In class, we've discussed the four fundamental equations of backpropagation in a network with fully-connected layers. Suppose we have a network containing a convolutional layer, a max-pooling layer, and a fully-connected output layer. How are the equations of backpropagation modified?

We have the following network architecture:



For the final connected layer, the backpropogation equations are not modified.
For the max pooling layer, the gradient of the weights with respect to the cost function will be zero for all of the neurons that weren't the max, and 1 for those the were:

$$\delta_j^L = \frac{\delta C}{\delta z_j^L} = \sum_k \frac{\delta C}{\delta a_k^L}\frac{\delta a_k^L}{\delta z_j^L} = \frac{\delta C}{\delta a_{max}^L}\frac{\delta a_{max}^L}{\delta z_j^L} = \frac{\delta C}{\delta a_{max}^L} \cdot 1 = \frac{\delta C}{\delta a_{max}^L}$$

Uncle.

5. Design and train a CNN for MNIST. Save your code as prob4.py. Record your result and give a description on how you design the network and briefly on how you make those choices. (e.g., numbers of layers, initialization strategies, parameter tuning, adaptive learning rate or not, momentum or not, etc.)

My CNN has the following architecture distinct from the:

- Batch size of 50 to be closer to on-line learning. This may make weights update more quickly.
- In order to adjust for this, I slowed down the learning rate.
- The architecture of the network remained the same as the default; I couldn't get it to work when changing kernel or feature size.

This network achieved 94.1% accuracy, as compared to the default performance of 91%.