Stat 6910, Section 002
Deep Learning: Theory and Applications
Spring 2019

Homework 5
Jared Hansen

Due: 8:00 AM, Monday 04/01/2019

A-number: `A01439768`

e-mail: `jrdhansen@gmail.com`

Homework V

STAT 6910/7810-002 - Spring semester 2019

Due: Friday, March 29, 2019 - 5:00 PM

Please put all relevant files & solutions into a single folder titled `<lastname and initials>_assignment5` and then zip that folder into a single zip file titled `<lastname and initials>_assignment4.zip`, e.g. for a student named Tom Marvolo Riddle, `riddletm_assignment5.zip`. Include a single PDF titled `<lastname and initals>_assignment4.pdf` and any Python scripts specified. Any requested plots should be sufficiently labeled for full points.

Unless otherwise stated, programming assignments should use built-in functions in Python, Tensorflow, and PyTorch. In general, you may use the scipy stack; however, exercises are designed to emphasize the nuances of machine learning and deep learning algorithms - if a function exists that trivially solves an entire problem, please consult with the TA before using it.

Problem 1

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?

- **<u>For λ</u>:** the idea behind using a parameter λ for regularization is to prevent overfitting to (training) data, penalizing model complexity (large weights) if it doesn't give a large enough reduction in the cost function. However, if we use gradient descent (GD) to try and learn λ we can be almost certain that λ will be set to 0. Why is this? The goal in training our model is to minimize whatever our chosen cost function is, relative to training data. When we add a regularization term that penalizes large weights (typically scaled by λ) to the original cost, this term alone increases the value of the cost function. So, the easiest way to minimize the newly augmented cost function ($C_{new} = C_O + C_{regrztn}$) is to "learn", via GD, that $\lambda = 0$ gives the smallest value of the cost. This happens since we're training on reduction of the cost function, C_{new}, on the training data. When using GD to learn the values of weights that give the smallest cost function value, it's easy to see that $C_{regrztn} = 0$ via $\lambda = 0$ will make C_{new} as small as possible, since (1.) it is the minimum value of $C_{regrztn}$ and (2.) it will allow the network to over-fit on the training data (in the form of minimizing the cost function).
  This entirely defeats the purpose of regularzation in the first place. As such, GD should not be used to tune λ.
- **<u>For η</u>:** any time GD is used to learn we must select a learning rate η. Let's say that the learning rate of our original network is η_0, and we want to tune this parameter via GD. Well, in order to learn the optimal η_0^* we'll need yet another η, call it η_1, to learn optimal η_0^* in our new GD. And now that we have η_1 to tune, we could do a couple of things: (1.) do some kind of grid search as we typically do for tuning learning rate, or (2.) tack on yet another layer of GD to learn the ideal η_1^*, meaning we'll have to tune some η_2.
  It comes down to this: at some point we'll have to use something other than GD in order to tune one of the η_i since GD necessarily involves tuning the learning rate. Because of an "Inception" type problem (but instead of dreams within dreams we have η's within η's) we will keep making this problem more and more complex. We should just use something like a grid search at the very first η to minimize training computation and structural clarity for the network.

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 $1/\sqrt{n_{in}}$ where n_{in} 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) Supposing λ is not too small, the first epochs of training will be dominated almost entirely by weight decay.

   - From the Lecture 11 notes on slide 22 (group exercise) we have the weight update rule(s):
     - <u>Without regularization:</u> $\left[w_{new} = \left(w_{old}\right) - \left(\eta \frac{\partial C_0}{\partial w_{old}}\right)\right]$
     - <u>With regularization:</u> $\left[w_{new} = \left(1 - \frac{\eta\lambda}{n}\right)\left(w_{old}\right) - \left(\eta \frac{\partial C_0}{\partial w_{old}}\right)\right]$

     Here, we are using the weight update rule <u>with regularization</u>.
   - Let's make another assumption to make our mathematical thinking easier. Let's assume that the second term in the weight update rule, $\left[\left(\eta \frac{\partial C_0}{\partial w_{old}}\right) \approx 0\right]$ relative to the magnitude of the first term; in other words, this second term doesn't play a significant role in this "weight decay during the first epochs of training" when compared to the second term. The loose thinking behind why this is permissible is that the learning rate η is often a very, very small number, likely making the second term very small. On the other hand, the first term is the old weight scaled by $\left(1 - \frac{\eta\lambda}{n}\right)$. This scaling term is going to be fairly close to 1, because the numerator $\left(\eta\lambda\right)$ will be very small (η and λ are typically very small numbers), and the number of training examples n is likely to be quite large. Thus $\frac{\eta\lambda}{n} \approx \frac{\text{very small}}{\text{fairly large}} = \text{very small} \approx 0 \implies$ $\left(1 - \frac{\eta\lambda}{n}\right) \approx \left(1 - 0\right) \approx 1$. However, it isn't entirely true to say that this scaling term for w_{old} is ≈ 1 otherwise the weights wouldn't be decaying.
   - Instead, let's say that $\left[\text{scaling term} < 1\right] \implies \left[\text{weight decay}\right]$ because multiplying the old weight by some value that's slightly less than 1 will make the new weight slightly less than the old weight (we're also assuming that the second term is negligible in effect, and close to 0).
   - For ease of math, let's assume a learning rate of $\eta = 0.07$ for illustrative purposes. Therefore, the first term of the regularized weight update rule will look like:
     $\left[1 - \frac{(0.07)(\lambda)}{n} < 1\right] \rightarrow \left[-\frac{(0.07)(\lambda)}{n} < 0\right] \rightarrow \left[-(0.07)(\lambda) < 0\right] \rightarrow \left[(0.07)(\lambda) > 0\right]$
     Here we can see the "...supposing λ is not too small..." condition given in the prompt illustrated mathematically. We need the numerator of the fraction in the scaling term to be big enough that the scaling term doesn't go all the way to 1, otherwise we wouldn't have decay.
   - So given these arguments, we can conclude that the $\left[\text{scaling term} < 1\text{, but close to }1\right]$, and a second term that is negligible will give us weight decay during the first epochs of training.

(b) Provided $\eta\lambda \ll n$ the weights will decay by a factor of $\exp(-\eta\lambda/m)$ per epoch.

- Let's examine the Taylor Series expansion of:
  $exp\left(-\frac{\eta\lambda}{n}\right) = 1 - \frac{\eta\lambda}{n} + \frac{\left(\frac{\eta\lambda}{n}\right)^2}{2!} - \frac{\left(\frac{\eta\lambda}{n}\right)^3}{3!} + \cdots$
- Now, let's make some assumptions so that the math works out as it should for this problem:
  - first, let's use the given fact that $\left[\eta\lambda \ll n\right] \implies \left[\frac{\eta\lambda}{n} \ll 1\right] \implies \left[\left(\frac{\eta\lambda}{n}\right)^{j\in\{2,3,\dots\}} \lll 1\right]$
  $\implies \left[\frac{\left(\frac{\eta\lambda}{n}\right)^{j\in\{2,3,\dots\}}}{j!} \approx 0\right]$ In other words, we've given a rough mathematical argument that the 3^{rd}, 4^{th}, ... and so on terms in the Taylor Series expansion given above are very close to 0, and can be ignored.
  Therefore, our Taylor Series approximation becomes: $\left(1 - \frac{\eta\lambda}{n}\right) \approx exp\left(-\frac{\eta\lambda}{n}\right)$.
- Now we can use this new Taylor Series approximation to change the weight update rule:
  $\left[w_{new} = \left(1 - \frac{\eta\lambda}{n}\right)\left(w_{old}\right) - \left(\eta\frac{\partial C_0}{\partial w_{old}}\right)\right]$ (assuming the 2^{nd} term is negligible) becomes
  $\left[w_{new} \approx exp\left(-\frac{\eta\lambda}{n}\right)\left(w_{old}\right)\right]$
- We also need to consider that each weight will get updated $\frac{n}{m} = b$ times during one training epoch (n is the total number of training examples, m is the number of examples per mini-batch, and b is the number of mini-batches).Therefore weight updates over a training epoch will look like: $w_{new} \approx \left[exp\left(-\frac{\eta\lambda}{n}\right)\cdots exp\left(-\frac{\eta\lambda}{n}\right)\left(w_{old}\right)\right]$ where there are $b = \frac{n}{m}$ of the $exp\left(-\frac{\eta\lambda}{n}\right)$ terms being multiplied together.
- Simplifying this we'll have:
  $w_{new} \approx \left[exp\left(-\frac{\eta\lambda}{n}\right)\cdots exp\left(-\frac{\eta\lambda}{n}\right)\left(w_{old}\right)\right] = \left[exp\left(-\left(\frac{\eta\lambda}{n}\right)\left(b\right)\right)\left(w_{old}\right)\right]$
  $= \left[exp\left(-\left(\frac{\eta\lambda}{n}\right)\left(\frac{n}{m}\right)\right)\left(w_{old}\right)\right]$
  $= \boxed{exp\left(-\left(\frac{\eta\lambda}{m}\right)\right)\left(w_{old}\right) \text{ exhibiting that the weights will decay by a factor of } exp(-\eta\lambda/m) \text{ per epoch.}}$

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

- Let's recall a general definition of the cost function when employing regularization (here L2): $C = C_0 + \frac{\lambda}{2m} \sum_{i=1}^{n} w^2$
- Also, some assumptions for our argument: assume $C_0 = 0$ since this is the ideal value of the cost function and it just leaves the regularization term (which is what we're trying to analyze).
  Also, assume the weights w are equal in order to allow for simpler, clearer math.
- Now, simplifying $C = C_0 + \frac{\lambda}{2m} \sum_{i=1}^{n} w^2$ we'll have: $C = 0 + \frac{(\lambda)(n)(w^2)}{2m}$ and solving for w gives:
  $\left[w = \sqrt{\frac{2mC}{(\lambda)(n)}} = \sqrt{\frac{2mC}{\lambda}}\left(\frac{1}{\sqrt{n}}\right)\right] \Longrightarrow \left[w \ \alpha\left(\frac{1}{\sqrt{n}}\right)\right]$
- Once we've reached a point where $\left[w \ \alpha\left(\frac{1}{\sqrt{n}}\right)\right]$ regularization won't have much/any additional effect on the cost function.
  In effect, we (don't need)/(don't use) regularization anymore and weight updates become:
  $\left[w_{new} = w_{old} - \eta\left(\frac{\partial C_0}{\partial w}\right)\right]$ instead of $\left[w_{new} = \left(1 - \frac{\eta\lambda}{n}\right)(w_{old}) - \eta\left(\frac{\partial C_0}{\partial w}\right)\right]$
  As we know from part (a.), the weights decay quickly thanks to the scaling factor for w_{old} when using regularization. Therefore, once this factor "disappears" since regularization has little/no effect on weights $\alpha\frac{1}{\sqrt{n}}$, the weight decay tails off since the only thing causing them to decay now is the $\eta\left(\frac{\partial C_0}{\partial w}\right)$ term, which yields much smaller changes in magnitude than did the scaling factor.

Problem 2

1. When discussing the vanishing gradient problem, we made use of the fact that $|\sigma'(z)| < 1/4$ for the sigmoid activation function. 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)

   - No! If the question were phrased "...would that help us avoid the **vanishing** gradient problem?" the answer would be yes. BUT, what happens is that we may see the **exploding** gradient problem come into play. So although we may not have to worry about the vanishing gradient now, we may still have an unstable gradient in the form of an exploding gradient.
   - Thinking of backpropogation in the most simple case (a network that is basically just a straight line with a few neurons along it, as in the notes), if the activations can be large (i.e. much larger than 1) instead of having $\left[\nabla = \left[<\frac{1}{4}\right]\cdots\left[<\frac{1}{4}\right] = \text{really small gradient}\right]$ as with the sigmoid activation function we might instead have $\left[\nabla = \left[>1\right]\cdots\left[>1\right] = \text{really large gradient}\right]$

2. Consider the product $|w\sigma'(wa+b)|$ where σ is the sigmoid function. Suppose $|w\sigma'(wa+b)| \geq 1$.

   (a) Argue that this can only ever occur if $|w| \geq 4$.

   - As we know from the prompt, for the sigmoid activation function, $\left[|\sigma'(z)| < \frac{1}{4}\right]$
   - In the context of this problem, we'll define z such that $z := wa + b$ (similar to how we've defined z using subscripts and such in earlier notes. Here it's a bit more informal in order to match with what the prompt uses.)
   - Using this new definition for z gives $\left[0 < |\sigma'(wa+b)| < \frac{1}{4}\right]$. For notational ease, let $sigPrime = |\sigma'(wa+b)| \implies \left[0 < sigPrime < \frac{1}{4}\right]$
   - Now, using properties of absolute value we can manipulate the last thing given in the prompt: $\left[\left(|w\sigma'(wa+b)|\right) \geq 1\right] \longrightarrow \left[\left(|w||sigPrime|\right) \geq 1\right] \longrightarrow \left[\left(|w|\left(0 < sigPrime < \frac{1}{4}\right)\right) \geq 1\right]$
     $\boxed{\text{Examining the last expression, the inequality is only satisfied for } |w| \geq 4 \text{, proving the desired result.}}$

(b) Supposing that $|w| \geq 4$, consider the set of input activations a for which $|w\sigma'(wa+b)| \geq 1$. Show that the set of a satisfying that constraint can range over an interval no greater in width than

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

- First, let's define something from the prompt for notational ease. Let $\left[x := |w| \geq 4\right]$.

  Also, from part (a) we will again use $\left[z = wa + b\right]$ and should also recall the fact that $\sigma'(z) = \dfrac{e^{-z}}{(1+e^{-z})^2}$
- Now let's manipulate the expression $|w\sigma'(wa+b)|$ using properties of absolute value, and then substituting in things we've defined just above:
  $\left[|w\sigma'(wa+b)|\right] = \left[|w||\sigma'(wa+b)|\right] = \big(x\big)\big(\sigma'(z)\big)$ since we have defined x such that $\left[x > 0\right] \implies \left[|x| = x\right]$, and we also know that derivatives of the sigmoid function are > 0 for all inputs $z \implies \left[|\sigma'(z)| = \sigma'(z)\right]$. So now we have $\left[\big(x\big)\sigma'(z)\right]$
- Now instead of $\left[|w\sigma'(wa+b)| \geq 1\right]$ we have $\left[\big(x\big)\sigma'(z) \geq 1\right]$. Substituting in the definition of $\sigma'(z)$ we have:
  $\left[\big(x\big)\left(\dfrac{e^{-z}}{(1+e^{-z})^2}\right) \geq 1\right] = \left[\dfrac{xe^{-z}}{1+2e^{-z}+e^{-2z}} \geq 1\right] = \left[1 + 2e^{-z} + e^{-2z} \leq xe^{-z}\right]$
  $= \left[1 + 2e^{-z} - xe^{-z} + e^{-2z} \leq 0\right] = \left[1 + (2-x)e^{-z} + e^{-2z} \leq 0\right]$
- Now let $y := e^{-z}$ giving $\left[1 + (2-x)y + y^2 \leq 0\right]$
- Using the quadratic formula we can solve for y:
  $y = \dfrac{-(2-x) \pm \sqrt{(2-x)^2 - 4(1)(1)}}{2}$,
  examining $\left[(2-x)^2 - 4\right] = \left[4 - 4x + x^2 - 4\right] = \left[x^2 - 4x\right] = \left[x(x-4)\right]$ giving
  $\left[y = \dfrac{x \pm \sqrt{x(x-4)}}{2} - \dfrac{2}{2}\right]$
- Now in the radical manipulate $\left[x(x-4)\right] = \left[\left(x(x-4)\right)\left(\dfrac{x}{x}\right)\right] = \left[x^2\left(1 - \dfrac{4}{x}\right)\right]$ giving
  $$\left[y = \frac{x \pm x\sqrt{1-\frac{4}{x}}}{2} - 1\right] \longrightarrow \left[y = \frac{x\left(1 \pm \sqrt{1-\frac{4}{x}}\right)}{2} - 1\right]$$
- The expression we have is close to what we're looking for in the prompt. We need to make just a couple of adjustments and we're there.
  First, let's replace y with $\left[y := e^{-z} = e^{-(wa+b)}\right]$ and replace x with $\left[x := |w|\right]$ giving:
  $$\left[e^{-(wa+b)} = \frac{|w|\left(1 \pm \sqrt{1-\frac{4}{|w|}}\right)}{2} - 1\right] \longrightarrow \left[ln\big(e^{-(wa+b)}\big) = ln\left(\frac{|w|\left(1 \pm \sqrt{1-\frac{4}{|w|}}\right)}{2} - 1\right)\right] \rightarrow$$

$\left[-wa - b = ln\left(\frac{|w|\left(1 \pm \sqrt{1 - \frac{4}{|w|}}\right)}{2} - 1\right)\right]$ We can just remove the $(-b)$ term since all it does is shift the interval without affecting the width of the interval (the width of the interval is what we care about). Also, we'll replace $(-w)$ with $|w|$ since that's what we're looking at in this problem, giving:

$\left[a = \frac{1}{|w|}ln\left(\frac{|w|\left(1 \pm \sqrt{1 - \frac{4}{|w|}}\right)}{2} - 1\right)\right]$ This looks very close to what the prompt is looking for.

We must realize that this is only half of the width of the interval due to the ± in the expression. Therefore, we put in a + instead of the ± and double the quantity to get the maximial width of the set of activations that satisfies the given constraints is $\boxed{\frac{2}{|w|}ln\left(\frac{|w|\left(1 + \sqrt{1 - \frac{4}{|w|}}\right)}{2} - 1\right)}$

(c) Show numerically (e.g., you could make a plot) that the above expression bounding the width of the range is greatest at $|w| \approx 6.9$, 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.

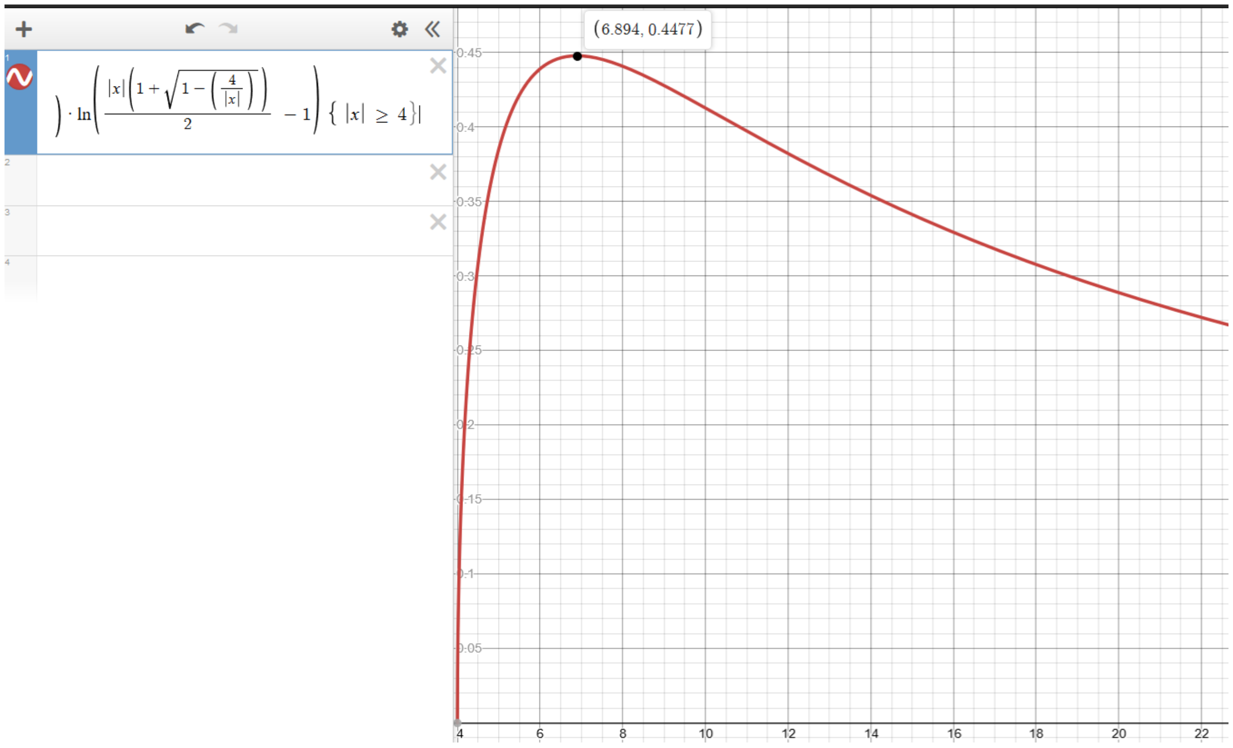

$) \cdot \ln\left(\frac{|x|\left(1+\sqrt{1-\left(\frac{4}{|x|}\right)}\right)}{2}-1\right)\{|x| \geq 4\}|$
(6.894, 0.4477)
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
4
6
8
10
12
14
16
18
20
22

Figure 1: This plots the function given above in part (b) (the whole function representation isn't quite visible on the LHS of the screen, but it's the same function). We can see the labeled maximal point of this function with an input of $|w| \approx 6.9$ yielding an output of interval width ≈ 0.45 (zoom into the PDF if blurry). As the prompt mentions, this is illustrating that a narrow range of weights yields a narrow range of activation values that yield a stable gradient. Outside of this, we'd get a vanishing gradient for the sigmoid activation function.

3. 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 $\mu > 1$? What would go wrong if we used $\mu < 0$?

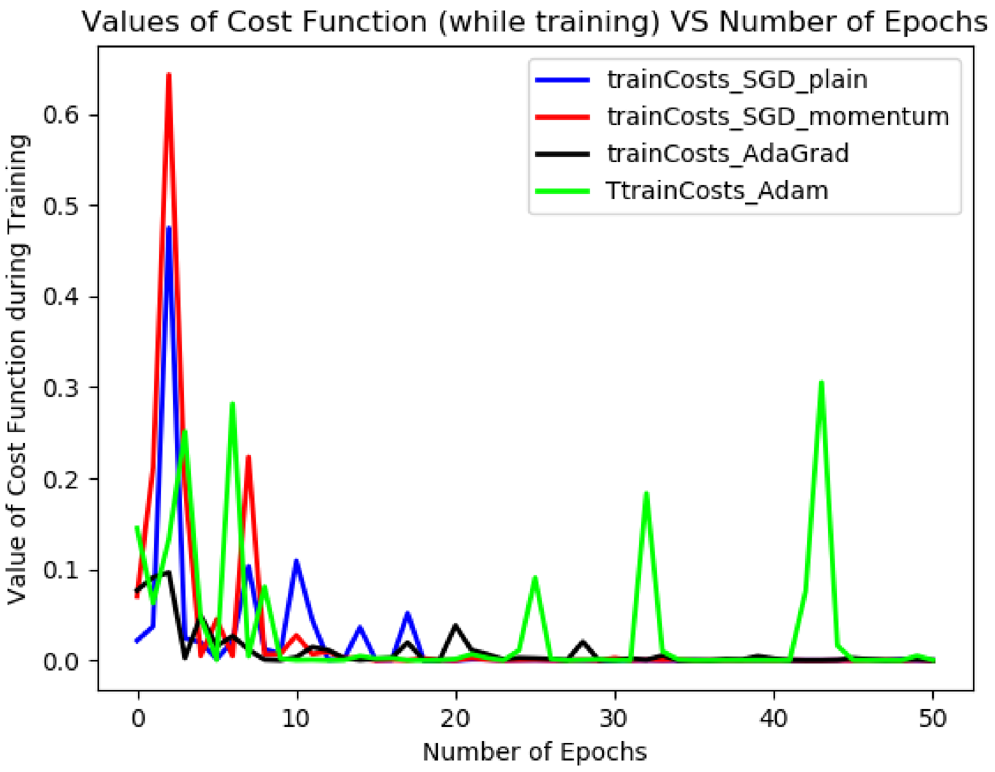
- For reference, from Lecture 13 "Variations on SGD", approx slide 30:
  - Introduce velocity variables $\boldsymbol{v} = v_1, v_2, ...$ for each corresponding w_j variable
  - New update rule:
    - $\boldsymbol{v} \rightarrow \boldsymbol{v'} = \mu \boldsymbol{v} - \eta \nabla C$
    - $\boldsymbol{w} \rightarrow \boldsymbol{w'} = \boldsymbol{w} + \boldsymbol{v'}$
    - where μ is a hyper-parameter that controls the damping/friction of the system
- **What would go wrong if we used $\mu > 1$?**
  This parameter value is essentially saying that the velocity is always increasing. This would mean we could easily go past minima (either "shooting out" of them or just entirely skipping over them in the surface of the cost function). Each successive step we take in the direction of the negative gradient becomes larger, meaning that we lose stability and will very likely not have convergence to a local minima of the cost function.
- **What would go wrong if we used $\mu < 0$?**
  This parameter value results in the opposite of what we want. Ideally, we want to take bigger steps down the cost surface in the direction of the negative gradient when the gradient is large (i.e. we're far from a minimum), and take smaller steps in the direction of the negative gradient when the gradient is small (i.e. we're close to a minimum).
  Using a value of $\mu < 0$ would do the opposite of this, causing us to speed up (take larger steps) when the gradient is small and slowing down (taking smaller steps) when the gradient is large. Similarly to using a value of $\mu > 1$, we'd have a hard time converging to a minimum of the cost function since we're either stepping too far when we're close to it, or taking increasingly smaller steps when we're far from it.

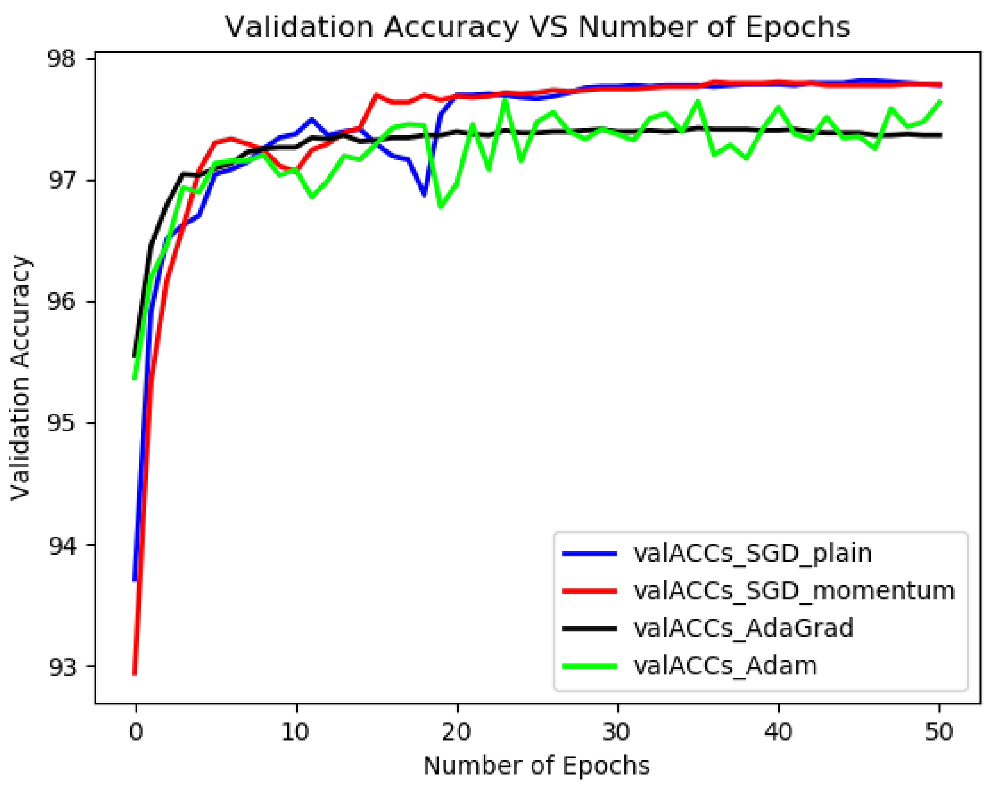
Problem 3

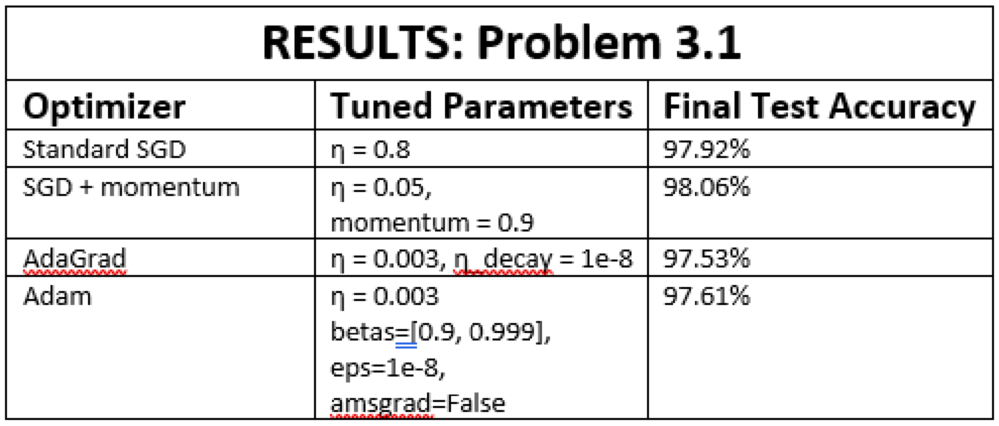
For this problem, you will reuse the code you wrote for Homework 4 that trained a fully connected neural network with 2 hidden layers on the MNIST dataset. You should retain the same number of nodes in each layer as in your final design. Keep the same values for the tuning parameters unless requested otherwise below.

1. Use the following approaches to optimization: 1) standard SGD; 2) SGD with momentum; 3) AdaGrad; 4) Adam. Tune any of the associated parameters including the global learning rate using the validation accuracy. Report the final parameters selected in each case, and the final test accuracy in each case. Provide two plots with the results from all four approaches: 1) the training cost vs the number of epochs and 2) the validation accuracy vs the number of epochs. Which optimization approach seems to be working the best and why?

My answers to the questions can be found in the caption for the last plot (table of results).







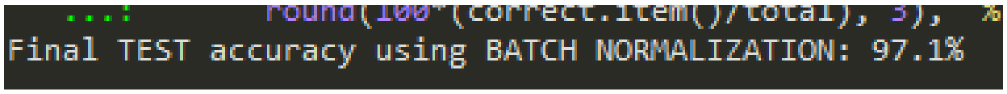
RESULTS: Problem 3.1

Optimizer	Tuned Parameters	Final Test Accuracy
Standard SGD	η = 0.8	97.92%
SGD + momentum	η = 0.05, momentum = 0.9	98.06%
AdaGrad	η = 0.003, η_decay = 1e-8	97.53%
Adam	η = 0.003 betas=[0.9, 0.999], eps=1e-8, amsgrad=False	97.61%

Figure 2: As we can see from this table, SGD (stochastic gradient descent) with momentum yields the best test accuracy, followed closely by standard SGD. AdaGrad and Adam also did quite well. However, I'd go with SGD+momentum as "...the optimization approach (which) seems to be working the best..." Here is my "why": (1) the value of the cost function stabilized more quickly than for any of the other methods (see the red line in the first plot), (2) the validation accuracy stabilized either more quickly or just as quickly as for the other optimizers (see the red line in the second plot), and (3) it gives the overall best test accuracy. These results may just be due to the way I tuned parameters, and the other optimizers might give better results, but it looks like SGD+momentum is the way to go.

2. Pick one of the optimization approaches above. Using the same network, apply batch normalization to each of the hidden layers and retune the (global) learning rate using the validation accuracy. Report the new learning rate and the final test accuracy. Does batch normalization seem to help in this case?

   - I am picking standard SGD since the only parameter I need to tune is learning rate (also, SGD was the second-best performer in part 1 of the problem).
   - **Newly-tuned gloabl learning rate:** 2.0
   - **Final test accuracy using batch normalization:** 97.1%



```
   ...:        round(100*(correct.item()/total), 3),  %
Final TEST accuracy using BATCH NORMALIZATION: 97.1%
```

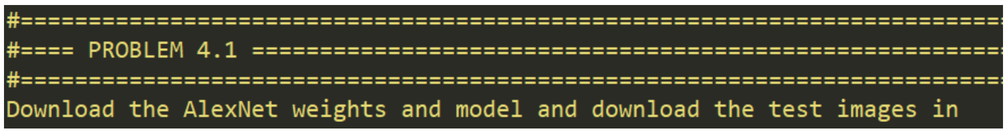
   - As we can see from comparing this test accuracy to the one above, batch normalization doesn't help in this case, hurting our test accuracy just slightly (going down from 97.92% without normalization to 97.1% with normalization).

Problem 4

Note: we are in the process of verifying that the PyTorch code for AlexNet works. If you are using PyTorch, you may want to wait a little bit on this problem.

1. **AlexNet** Download the AlexNet weights and model and download the test images in the AlexNet folder on Canvas. In what follows, attach your code for parts 2 and 3.

   See code for how I did this, located under this heading in the prob4 script:



```
#==============================================================================
#==== PROBLEM 4.1 =============================================================
#==============================================================================
Download the AlexNet weights and model and download the test images in
```

2. **Reading out from a layer:** Write code that extracts the output of the first convolutional layer for each of the test images. What output shapes do you get? In this layer, readout the output of one of the 96 57×57 arrays. Plot this for each test image using matplotlib's imshow or a similar matrix-to-image function for a couple of sample images and include them in your writeup.

   - I got that the output of the first convolutional layer is a tensor having 64 channels (channel = matrix/array) here, with each of these channels being 57×57. I don't know where the 96 number from the prompt is coming from, especially when referring to the AlexNet source code on GitHub: `https://github.com/pytorch/vision/blob/master/torchvision/models/alexnet.py`
   - The prompt is unclear here, because it says "...for each test image..." and later in the same sentence says "...for a couple of sample images..." Being an efficiency-minded student, I went with the latter interpretation, and have included a couple of the channels for a couple of the images (the original images are included for reference).

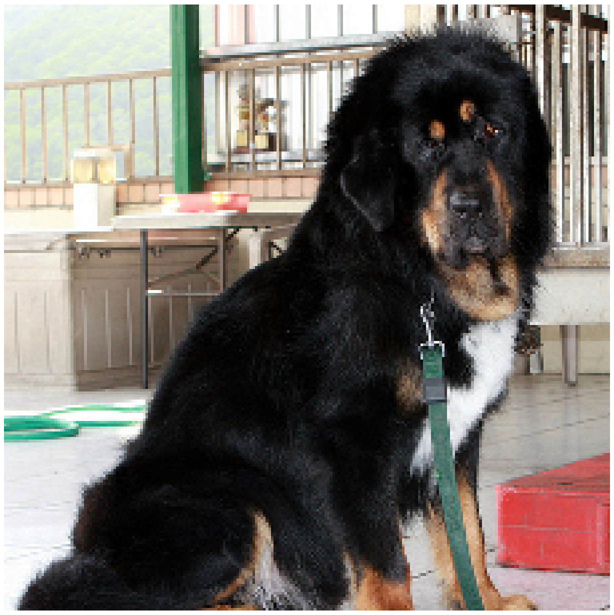


Figure 3: This is the original 'dog' image from Canvas (a Tibetan mastiff).

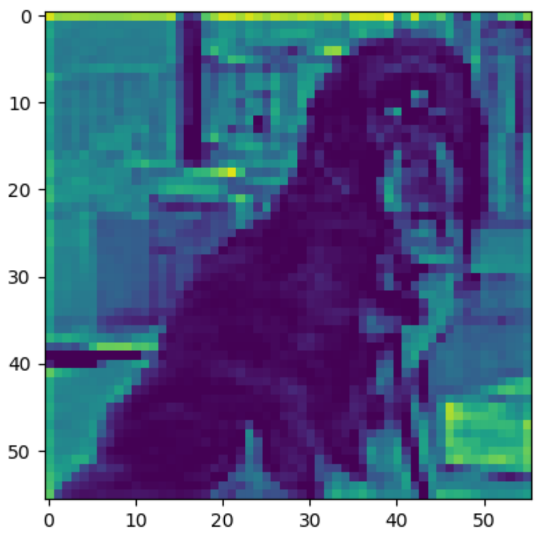


Figure 4: This is the 58th of 64 channels for the 'dog' image. It looks to be the image, but with different contrast.

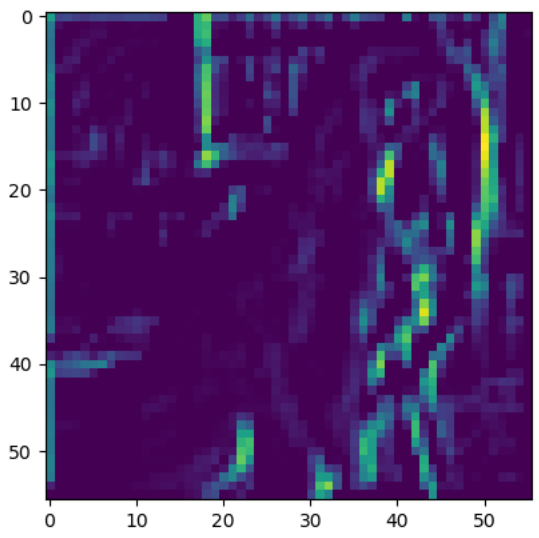


Figure 5: This is the 56th of 64 channels for the 'dog' image. It looks to be some kind of edge detection.

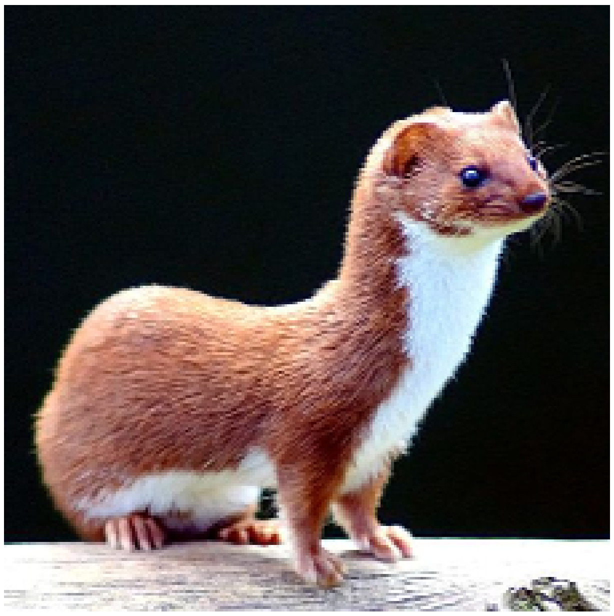


Figure 6: This is the original 'laska' image from Canvas (a weasel).

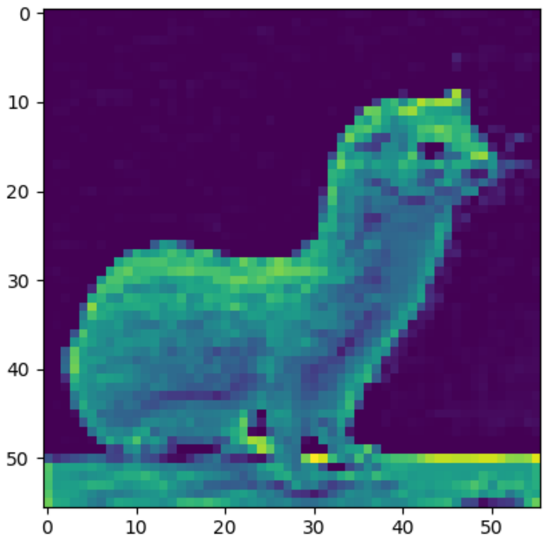


Figure 7: This is the 58th of 64 channels for the 'laska' image. It looks to be the image, but with different contrast.

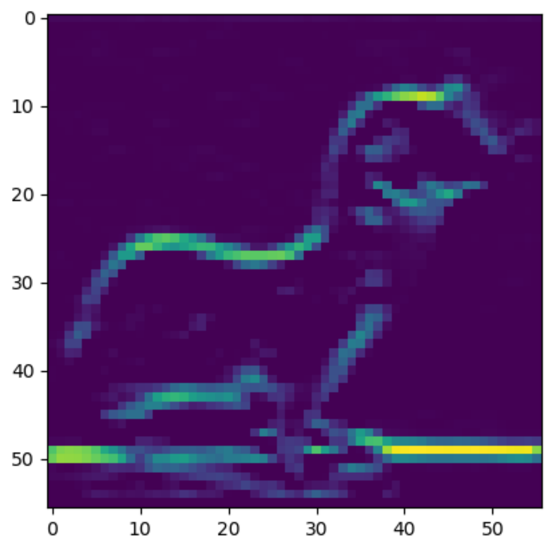


Figure 8: This is the 56th of 64 channels for the 'laska' image. It looks to be some kind of edge detection.

3. Write code that extracts the output of the final layer. What is the dimension of this layer?

   The code that I wrote shows it's a 1-dim vector of length 1000, which we would know simply by thinking about the architecture: we have 1000 classes of image from which the network can "choose", therefore the output layer should just have 1000 nodes, which it does ☺.

4. Try feeding in another image (not one of the test images) and reading out the top 5 probabilities of classification. Does the classification seem correct? Image inputs should be of the form 227×227 pixels and 3 channels (use .png to be safe). You may need to crop or zero pad your image.

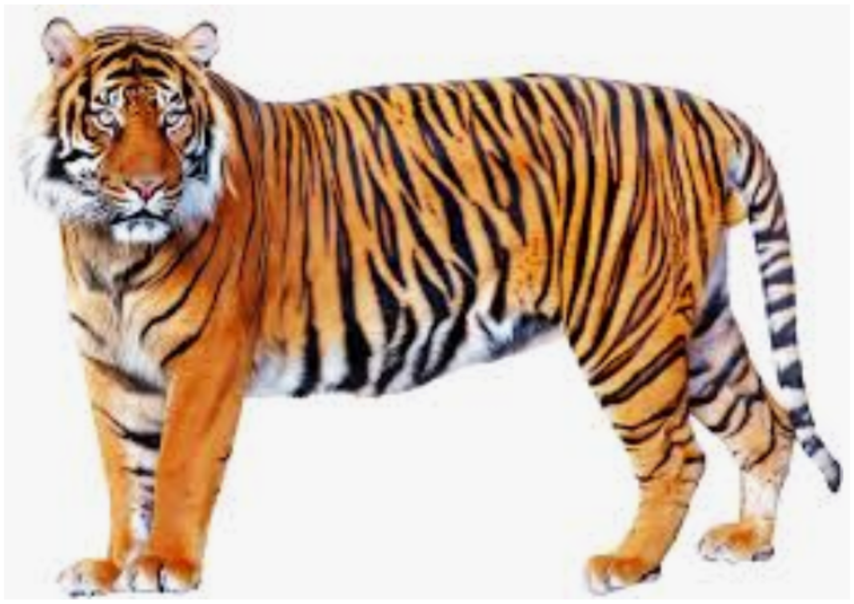
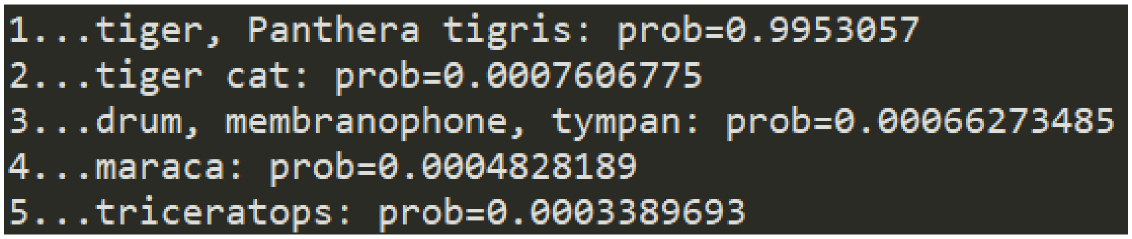


Figure 9: This is the tiger image on which I tested the network.



```
1...tiger, Panthera tigris: prob=0.9953057
2...tiger cat: prob=0.0007606775
3...drum, membranophone, tympan: prob=0.00066273485
4...maraca: prob=0.0004828189
5...triceratops: prob=0.0003389693
```

Figure 10: These are the predicted probabilities of the network for the given image: **crushed it**!! We can obviously see that the classification seems correct: the network is giving a probability of over 99% that it's a 'tiger, Panthera tigris', and even the second option is still a 'tiger cat' (however that might be different from a 'tiger, Panthera tigris'). AlexNet did a great job of classifying this image. The 3^{rd}, 4^{th}, and 5^{th} most likely options are funny to see: a drum, a maraca, and a triceratops haha.

Problem 5

1. **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?

   - Refer to the diagram on the page below (zoom in to read! Image quality is good ☺). I'll be giving my answers in the context of the network diagrammed below. When talking with Dr. Moon in office hours he suggested this would be sufficient (in lieu of specifying the exact mathematical changes I could give them in the context of this particular network).
   - Another thing to note: this diagram is only showing the network for 1 channel. In an actual CNN we would have much more than 1 channel for our [conv. + max-pooling] layer. But it is really only feasible to draw one channel. As such, the math given for this single channel would just be generalized to each channel in a CNN with many channels.
   - **<u>The Backpropogation Algorithm for SPECIFIED CNN</u>**
     1. **Input $\boldsymbol{x}$:** set the activation $\boldsymbol{a}^1$ for the input layer
     2. **Feedforward:** for each $l = 2, 3, \cdots, L$, compute $\boldsymbol{z^l} = \boldsymbol{W^l a^{l-1}} + \boldsymbol{b^l}$ and $\boldsymbol{a^l} = \sigma(\boldsymbol{z^l})$
        THIS IS DIFFERENT FOR A CNN: see the blue writing at the top of the diagram for how to perform the feed forward part for a CNN of the desired architecture.
     3. **Output error $\boldsymbol{\delta^L}$:** compute $\boldsymbol{\delta^L} = \left[\nabla_{\boldsymbol{a}^L} C\right] \odot \left[\sigma'(\boldsymbol{z^L})\right]$ (stays the same for the CNN)
     4. **Backpropogate the error:** this is very different for a CNN. Based on the diagram I've created below, here are the backpropogated errors for each layer (**<u>these same formulae are written on the diagram, with specifics of what each component consists of</u>**):
        - $\boldsymbol{\delta^{pool}} = \left[\left(\boldsymbol{w^L}\right)^T \left(\boldsymbol{\delta^L}\right)\right] \odot \left[\sigma'\left(\boldsymbol{z^{pool}}\right)\right]$
        - $\boldsymbol{\delta^{conv}} = \left[\left(\boldsymbol{W^{pool}}\right)^T \left(\boldsymbol{\delta^{pool}}\right)\right] \odot \left[\sigma'\left(\boldsymbol{z^{conv}}\right)\right]$
     5. **Output:** the cost function gradient is (recall $*$ symbolizes convolution):
        - For the weights in the filter (here, a 4×1 vector):

          $$\left[\nabla_{\boldsymbol{W}} C = \boldsymbol{X} * \boldsymbol{\delta^{conv}}\right] \longrightarrow \text{e.g. } \frac{\partial C}{\partial w_{11}} = \sum_{i=1}^{4} \left(\begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{41} \end{bmatrix} \odot \begin{bmatrix} \delta_1^{conv} \\ \delta_2^{conv} \\ \delta_3^{conv} \\ \delta_4^{conv} \end{bmatrix} \right)$$

          $$= \left[\frac{\partial C}{\partial w_{11}} = \left(x_{11}\right)\left(\delta_1^{conv}\right) + \left(x_{21}\right)\left(\delta_2^{conv}\right) + \left(x_{31}\right)\left(\delta_3^{conv}\right) + \left(x_{41}\right)\left(\delta_4^{conv}\right) \right]$$
        - For the values v_{ij} in the convolutional layer let's illustrate with an example.
          But first, let's remember that $\frac{\partial C}{\partial v_{ij}} = 0$ except when v_{ij} is max(window of interest in conv. layer), meaning only the max values for a window will affect the cost
          Also, we must note that $\frac{\partial m_{rs}}{\partial v_{ij}} = 0$ except for $(i = r), (j = s)$ (at least for this example), for example $\frac{\partial m_{12}}{\partial v_{11}} = 0$ but $\frac{\partial m_{11}}{\partial v_{11}} = \delta_1^{conv}$

          NOW: $\left[\frac{\partial C}{\partial v_{11}} = \frac{\partial C}{\partial m_{11}} \frac{\partial m_{11}}{\partial v_{11}} + \frac{\partial C}{\partial m_{12}} \frac{\partial m_{12}}{\partial v_{11}} + \frac{\partial C}{\partial m_{21}} \frac{\partial m_{21}}{\partial v_{11}} + \frac{\partial C}{\partial m_{22}} \frac{\partial m_{22}}{\partial v_{11}} \right]$

          $\longrightarrow \left[\frac{\partial C}{\partial v_{11}} = \frac{\partial C}{\partial m_{11}} \frac{\partial m_{11}}{\partial v_{11}} + 0 + 0 + 0 \right]$ where $\left(\frac{\partial C}{\partial m_{11}} = \delta_1^{pool} \right)$ and $\left(\frac{\partial m_{11}}{\partial v_{11}} = \delta_1^{conv} \right)$

          $\longrightarrow \boxed{\frac{\partial C}{\partial v_{11}} = \left(\delta_1^{pool}\right)\left(\delta_1^{conv}\right)}$ (this will only hold true for the v_{ij} that are the max of a respective window, otherwise $\frac{\partial C}{\partial v_{ij}} = 0$)

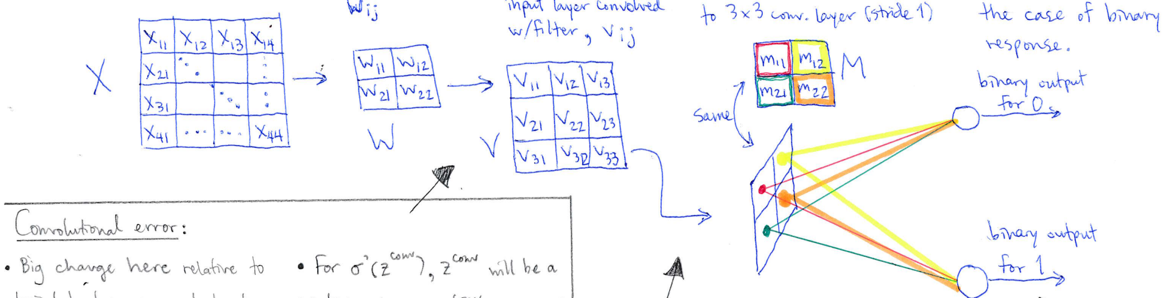
Input layer: here a 4x4 matrix of pixel values, X_{ij}

Apply filters: 2x2, stride of 1, W_{ij}

Conv. layer: a 3x3 matrix of input layer convolved w/filter, V_{ij}

Max-Pooled layer: applied 2x2 max-pooling to 3x3 conv. layer (stride 1)

Output layer: a 2x1 vector in the case of binary response.



Convolutional error:

- Big change here relative to typical backprop: we look at each value in the original conv layer and use an indicator fctn to identify if that max value is present in that "window", moving L-to-R across and shifting down to start of next row. This gives a square matrix, call it (w^{pool}), of dim. cardinality(max-pool matrix).

Ex: Say orig conv. matrix V is

$$\begin{bmatrix} 3 & 2 & 5 \\ 4 & 1 & 7 \\ 3 & 5 & 0 \end{bmatrix}$$

$$\downarrow$$

max-pooled matrix M

$$\begin{bmatrix} 4 & 7 \\ 5 & 7 \end{bmatrix}$$

$$\downarrow$$

reverse max-pool for each submatrix ("window")

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

- For $\sigma'(z^{conv})$, z^{conv} will be a vector, where z_i^{conv} is one of the max-pooled values in the max-pool matrix M. In other words, turn M into a column vector and apply σ' to each element of z^{conv} to get $\sigma'(z^{conv})$.

- FINALLY:

$$\underbrace{\delta^{conv}}_{4\times1} = \left[\underbrace{(w^{pool})^T}_{4\times4}\underbrace{(\delta^{pool})}_{4\times1}\right] \odot \left[\underbrace{\sigma'(z^{conv})}_{4\times1}\right]$$

Max-Pool Error:

- Same as for a FF-NNet, but we vectorize the weights connecting each m_{ij} and the output nodes. Thus:
- # rows in W^L = 4 = # elements in M.
- # cols in W^L = 2 = # nodes in output layer.
- FINALLY:

$$\underbrace{\delta^{pool}}_{4\times1} = \left[\underbrace{(W^L)^T}_{4\times2}\underbrace{(\delta^L)}_{2\times1}\right] \odot \left[\underbrace{\sigma'(z^{pool})}_{4\times1}\right]$$

Output error:

- Same as for a FF-NNet

$$\underbrace{\delta^L}_{2\times1} = \left[\underbrace{\nabla_{a^L} C}_{2\times1}\right] \odot \left[\underbrace{\sigma'(z^L)}_{2\times1}\right]$$

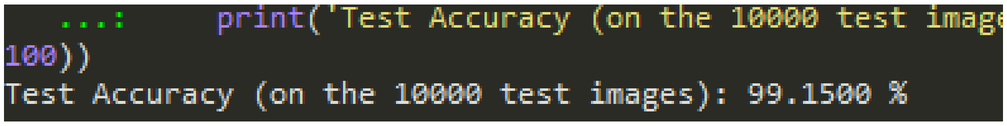
- NOTE: if we turn the 2x2 filter into a 4x1 vector we see that [dim(filter) = 4x1] == [dim(δ^{conv}) = 4x1]. Dimensions align!

2. Design and train a CNN with at least two convolutional layers, each followed by a max-pooling layer, for the MNIST dataset. Save your code as `prob5.py`. Record your final test 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.). Based on the results you obtain, does the CNN seem to do better or worse than other models you've trained?

   If you're using Tensorflow, you may want to modify the network based on the following link, if you don't know where to start with. If you're using PyTorch, the following link may still be helpful in the design process. https://www.tensorflow.org/tutorials/layers

Going in order of the prompt, I'll report the following things:

- **Final test result:**



```
   ...:     print('Test Accuracy (on the 10000 test image
100))
Test Accuracy (on the 10000 test images): 99.1500 %
```

Figure 11: As we can see from this screenshot, my final test accuracy was 99.15%. In retrospect, it was unnecessary to include more than 2 decimal places of accuracy since we're only testing on 10,000 images (i.e. $\frac{9,915 \text{ correct}}{10,000 \text{ total}} = (.9915)(100) = 99.15\%$, never getting more than 2 decimal places).

- **Design description/parameter tuning:**
  - My thought process for this problem was to go for parsimony unless my validation accuracy dictated otherwise. In other words, a minimal amount of network architectural complexity would make my life easier on several fronts: shorter training time, less coding, and fewer places to "get lost" in the design. Also, I'd be remiss if I didn't mention that I referenced this tutorial https://adventuresinmachinelearning.com/convolutional-neural-networks-tutorial-in-pytorch/ for initial design help, as well as the one Dr. Moon gives in the prompt.
  - Since we had developed a 2-hidden-layer network for MNIST in homework 4 and we're required to have 2 [conv+MP] layers by the prompt, I decided to combine these two things to give the following architecture:
    $\boxed{\text{input layer}} \longrightarrow \boxed{\text{conv + MP layer 1}} \longrightarrow \boxed{\text{conv + MP layer 2}} \longrightarrow \boxed{\text{fully-connected hidden layer 1}}$
    $\longrightarrow \boxed{\text{fully-connected hidden layer 2}} \longrightarrow \boxed{\text{10-node output layer}}$
  - Additionally, I borrowed some design choices from the tutorial I reference above. Specifically, deciding how many channels to use (I used 32 for the first conv layer and 64 for the second), as well as the size of filters and strides to use. My rationale was this: trying to decide these values adds an additional 6 parameters to tune, and I have no experience building a CNN. If they have found that these values are good for image classification, then my time is best spent tuning other things and using someone else's time investment to my advantage.
  - As for other parameters (learning rate, stopping rule (number of epochs), batch size, and number of neurons in hidden layers), I used the validation set to tune these. First, I started with number of neurons in the hidden layers. I set them both to 100 since that's what we used in hw4, and it seemed to give good results here as it did there. Along with this, I was playing around with the learning rate. I started with a larger value (somewhere around 5.0) and it was taking forever for each epoch step to compute. So I terminated the process and started

adjusting it downward until settling around 0.001. Next, I monkeyed around with batch size, recalling that a batch size of $2^n, n \in \{2, 3, 4, 5, ...\}$ was recommended during class to minimize GPU compute time. I ended up going with a batch size of $2^7 = 128$ as this seemed to minimize compute time and give good validation accuracy. As for the stopping rule, I just stuck with specifying a fixed number of epochs to run after having tuned these other parameters. The compute time for this CNN was much, much greater than for the feedforward MNIST network that we built, so I cut the number of epochs down to only 4. (This still gave good validation and test accuracy though.)
As for weight initialization, adaptive learning rate, and momentum, I didn't incorporate these parameters into my process. For the momentum parameter it was a simple matter of the fact that I used the Adam optimizer; since I wasn't using SGD there wasn't an option to use momentum. As for initialization and the learning rate, I figured it was easiest to just use the defaults (set learning rate and default weight strategy). As I mentioned before, the list of parameters to tune and architecture to develop was already considerable, so I just decided to cut these things out of the process to save on time and have fewer things to tune.

– **Does the CNN do better/worse than other models?**
Based on the test accuracy, the CNN does quite a bit better than the feedforward network I built for hw4. That network just eked over 98% accuracy (something like 98.13%), so we see a relative increase in accuracy of just over 1%. And obviously this CNN blows the eariler MNIST networks we built out of the water.
BUT, my bet would be that this CNN would get even higher accuracy if we turned it loose for a larger number of epochs. I trained the feedforward net from hw4 (98.13% test acc.) for 50 epochs, and only trained this CNN for 4 epochs (due to compute time). My guess is that we could get something more like 99.5% if we just trained the CNN for more epochs.