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?

Ans. Cost function is not the function of eta (η) so you can’t use gradient descent to tune eta. However, you can still use gradient descent to tune lamda λ but when doing so λ will keep decreasing to make the reduce the overall cost. λ will decrease until negative infinity. Thus it is not feasible to use gradient descent to learn the values of eta and lamda.

Problem 2

1. When discussing the vanishing gradient problem, we made use of the fact that |σ 0 (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)

   Ans. Surely it helps us to avoid the vanishing gradient problem but it does not help us avoid the exploding gradient problem. Thus it is unlikely it will solve the unstable gradient problem. There is really narrow range of **a (activation_func(z))** that won’t result in unstable gradients.

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 μ > 1? What would go wrong if we used μ < 0?

   Ans.

   $$\boldsymbol{v} \to \boldsymbol{v}' = \mu\boldsymbol{v} - \eta\nabla C$$
   $$\boldsymbol{w} \to \boldsymbol{w}' = \boldsymbol{w} + \boldsymbol{v}'$$

   μ > 1 means the weights would keep increasing unless the gradient and learning rate is high enough to compensate the high μ. It means the gradient descent would move backwards. I.e the gradient descent is moving up the hill instead of down the hill.

   If μ < 0 weights would move is such a way increasing the cost function, that is they would be moving far away from the minima.

   In both cases

- The current weights have large inertia why try to refuse change.
- We're not updating the weight by small amount that we intend to.

Problem 3

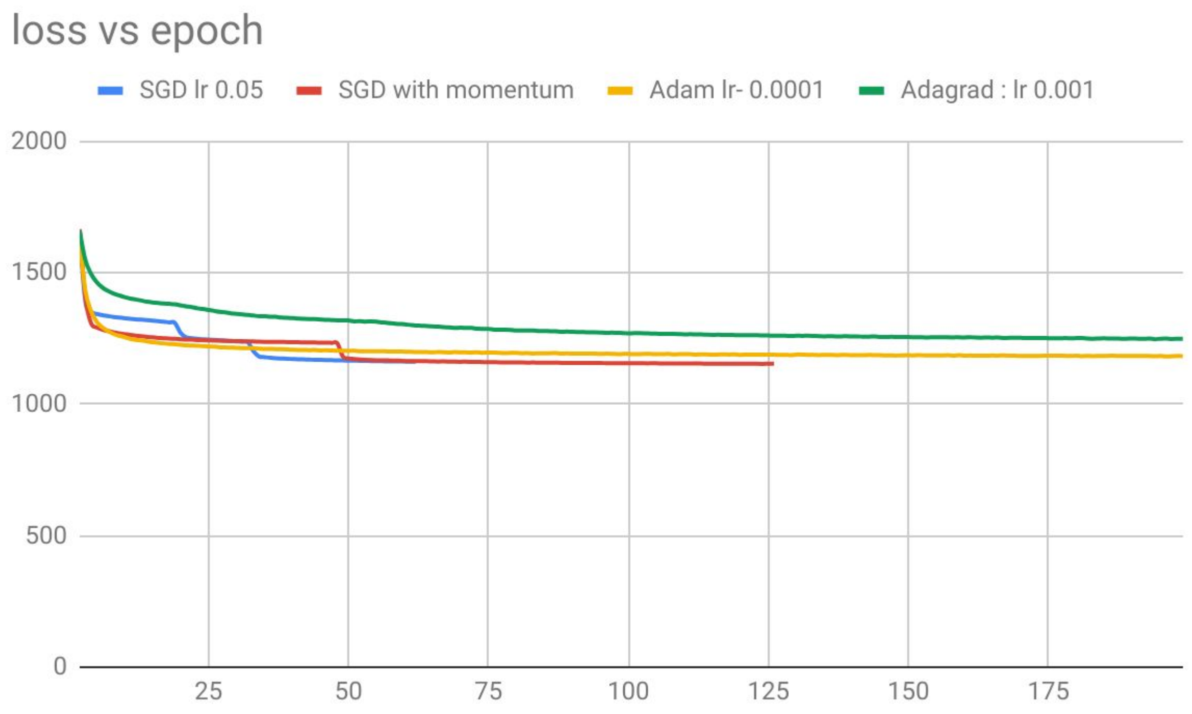
For this problem, you will reuse the code you wrote for Homework 4 that trained a fully connected neural network with 2 hidden layers on the MNIST dataset. You should retain the same number of nodes in each layer as in your final design. Keep the same values for the tuning parameters unless requested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

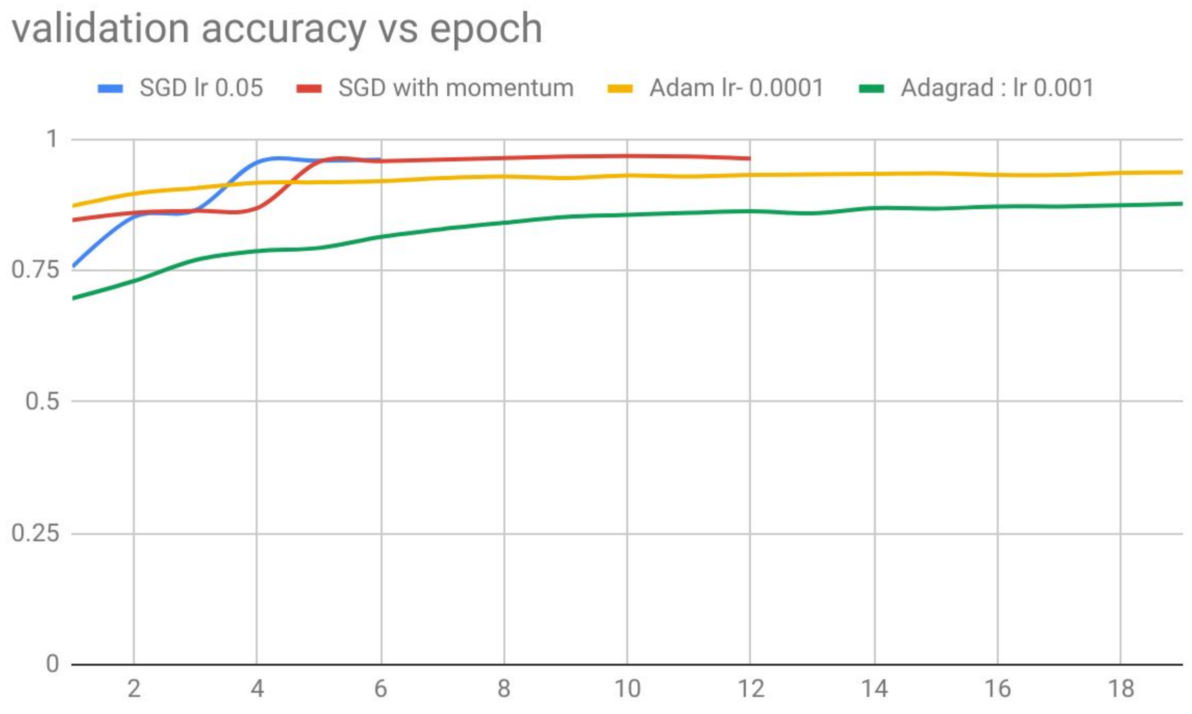
1. Use the following approaches to optimization: 1) standard SGD; 2) SGD with momentum; 3) AdaGrad; 4) Adam. Tune any of the associated parameters including the global learning rate using the validation accuracy. Report the final parameters selected in each case, and the final test accuracy in each case.

- Stopping criteria : when losses between consecutive epochs is less than 0.001
- Learning rates were tuned to avoid oscillations.

Optimizer	SGD	SGD with momentum 0.1	Adam	AdaGrad
Learning rate	0.05	0.05	0.0001	0.001
Dropout	0.1	0.5	0.5	0.5
Test Accuracy	96.3	96.67	93.89	87.65
~~Dropout with even bad result~~	~~0.1~~	~~0.1~~	~~0.1~~	~~0.1~~

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?





Legend : One box represents 10 epoch

For my case, SGD with momentum works best as it starts with initially high scores and converges quickly to really high validation accuracy and low loss as well.

I believe Adam and AdaGrad would work too but they do not work in my case because of my lack of experience of tuning them.

I had to try different values for learning rate for Adam and Adagrad until the losses and the validation accuracy did not oscillate.

From observation, it seems like AdaGrad and Adam perform better with dropout.

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?

Learning rate

- 0.1 for first 30 epochs reached 96.8% validation accuracy
- 0.05 for another 30 epochs reached 97.3% validation accuracy
- 0.01 for another 10 epochs reached 97.4% validation accuracy

Final test accuracy : 97.3%

I used dropout in some of the models/ architectures. They performed worse. I suggest not using dropout together with batch normalization. A paper suggests using Maxout architectures with dropout for optimal performance.

Problem 4

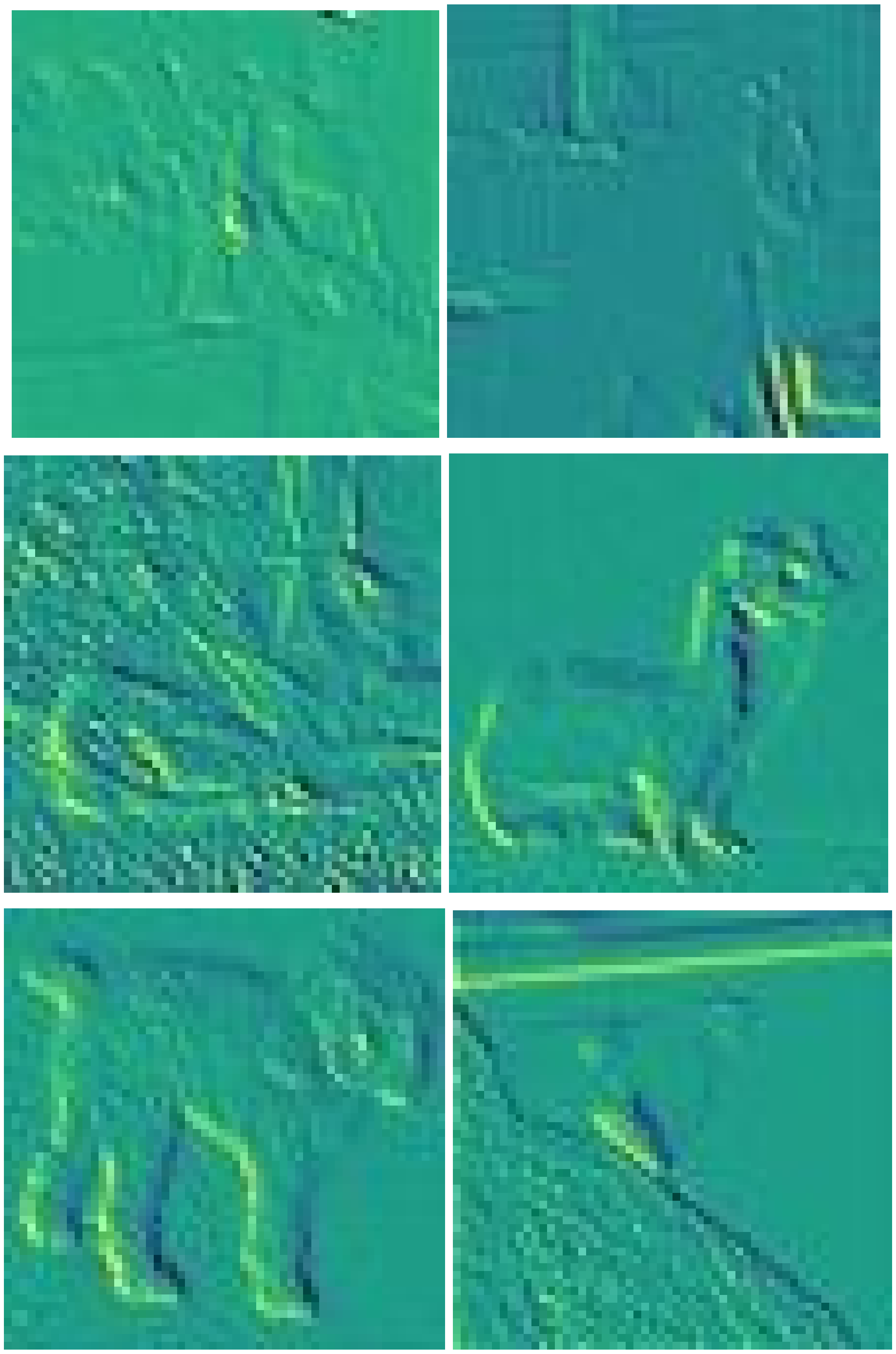
My random image : robot.png

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.

Ans. Downloaded the weights and ensured that it works correctly.

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.

Ans.
Without padding, the first convolution layer produces 64 56*56 outputs.
The output of third channel has been displayed / plotted for all images including one random image.
The first one is random.png's output.



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

Ans. DImension of the final layer is 1*1000 meaning that there are thousand classes.

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.

Top five classes for robot.png : [586 603 971 883 178] corresponding to
[Half track, horse cart, bubble, vase, weimaraner]

No the classification does not seem correct.

Problem 5.2

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?

Ans.
I began the network small.

Architecture

- With 11 filters in first layer and 20 layers in second layer. Saturates quickly at 88%
- With 64 filters in first layer and 20 layers in second layer. Quickly at 98% in 40 epochs and saturates
- With 20 filters in first layer and 64 layers in second layer. Quickly at 97.8% in 10 epochs and saturates. I thought this would not happen as large number of filters are important at the early stage rather than later stage. This was quite counter intuitive. Saturates at 98.2% in 30 epochs. Trained for a total of 80 epochs.

Parameters

- Initialization strategy : (PyTorch's default initialization strategy) Initialize weights as Gaussian random variables with mean 0 and standard deviation $1/\sqrt{n_{in}}$ where n_{in} in is the number inputs to a neuron
- Momentum : 0
- Dropout : 0
- Batch size : 64 to make optimum tradeoff between optimal use of my GPU and faster convergence.
- learning rate : 0.05 (fixed)

THe CNN network tends to do better because of the following observations :

- Faster convergence owing to the shared weights and exploitation of the fact that numbers that are input to the network together form an image

- Better accuracy