Due Date: February 12th (11pm), 2019
Instructions

- *For all questions, show your work!*
- *Submit your code ("solution.py" only) and your report (pdf) separately and electronically via the course Gradescope page.*
- *Work off the template and DO NOT modify the name of the file "solution.py" since the code will be automatically graded.*
- ***TAs for this assignment are Jie Fu, Sai Rajeswar, and Akilesh B***

Problem 1

The first part of this assignment is the coding part. We provide the template in the course GitHub repository [1]. You need to fill up the blanks in the 'solution.py' script, WITHOUT modifying the template. In this problem, we will build a Multilayer Perceptron (MLP) and train it on the MNIST handwritten digit dataset.

Building the Model (code) [35] Consider an MLP with two hidden layers with h^1 and h^2 hidden units. For the MNIST dataset, the number of features of the input data h^0 is 784. The output of the neural network is parameterized by a softmax of $h^3 = 10$ classes. Build an MLP and choose the values of h^1 and h^2 such that the total number of parameters (including biases) falls within the range of [0.5M, 1.0M]. Implement the forward and backward propagation of the MLP in numpy without using any of the deep learning frameworks that provides automatic differentiation. Train the MLP using the probability loss (*cross entropy*) as the optimization criterion. We minimize this criterion to optimize the model parameters using *stochastic gradient descent*.

The following parts will be automatically graded:

1. Write the function `initialize_weights`.
2. Write all of the given activvation functions, i.e. `relu`, `tanh`, `sigmoid` and `softmax`, and the selector function `activation` that reads the string "activation_str" and select the corresponding activation function.
3. Write the `forward` pass function.
4. Write the `backward` pass function.
5. Write the `loss` functions (cross entropy for multi-class classification).

[1] `https://github.com/CW-Huang/IFT6135H20_assignment`

6. Write the `update` function that takes in the gradient to update the parameters according to the stochastic gradient descent update rule.

7. Finish the `train_loop` by performing the `forward` pass and the `backward` pass and updating the parameters (using the `update` function).

Problem 2

The second part of this assignment will be graded based on your report (including comments and figures). You may create additional scripts aside from the template that we provide and/or reuse the template WITHOUT modifying or renaming it! One way to do it is to create another class object that inherits from the `NN` template.

In the following sub-questions, please specify the *model architecture* (number of hidden units per layer, and the total number of parameters), the *nonlinearity* chosen as neuron activation, *learning rate*, *mini-batch size*. The following tasks should be written as a report and submitted via Gradescope (separately from the code).

Initialization (report) [15] In this sub-question, we consider different initial values for the weight parameters. Set the biases to be zeros, and consider the following settings for the weight parameters:

- **Zero**: all weight parameters are initialized to be zeros (like biases).
- **Normal**: sample the initial weight values from a standard Normal distribution; $w_{i,j} \sim \mathcal{N}(w_{i,j}; 0, 1)$.
- **Glorot**: sample the initial weight values from a uniform distribution; $w^l_{i,j} \sim \mathcal{U}(w^l_{i,j}; -d^l, d^l)$ where $d^l = \sqrt{\frac{6}{h^{l-1}+h^l}}$.

1. Train the model for 10 epochs [2] using the initialization methods above and record the average loss measured on the training data at the end of each epoch (10 values for each setup).

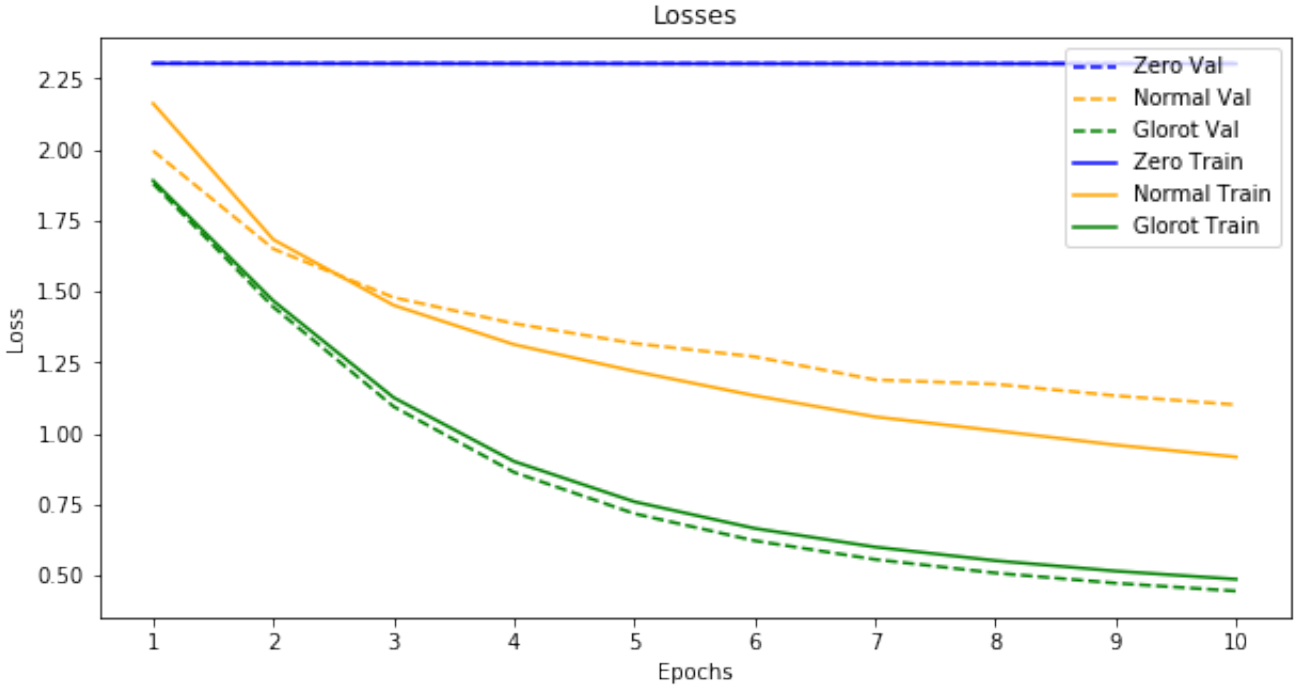
2. Compare the three setups by plotting the losses against the training time (epoch) and comment on the result.

[2] One epoch is one pass through the whole training set.

Trained the model using three different initialization methods i.e. normal, zeros, and glorot. All three models with their validation and training losses are plotted against each epoch.

MODEL DETAILS

Neuron Activation: **Relu**
Learning Rate: **7e-4**
Batch Size: **64**
Input Dimensions: **784** $\times 1$
Output Dimensions: **10** $\times 1$
Hidden Layers: **2**
Hidden layer dimension: **512, 256**
Weights: $784 \times 512 + 515 \times 256 + 256 \times 10 =$ **535050**
Biases: $512 + 256 =$ **768**
Total No. of parameters: **535818**



Zero Initialization: When the weights are initialized with zero, the derivative with respect to the loss will be same for every weight, and thus all the weights will have same value in all iterations thus producing a constant loss as shown in the graph above. Although, the biases can be initialized as zero.

Normal vs Glorot Initialization: Random normal initialization is far better than initializing zero but the deep neural networks can experience problems such as vanishing/exploding gradients if the weights are too close to 0/1. As evident in the graph, the glorot initialization performs much better than the normal random initialization. This method is a good starting point for the weights and mitigate the problems of vanishing/exploding gradients. Moreover, Glorot and Bengio in their paper demonstrated that networks initialized with Glorot achieved substantially quicker convergence and higher accuracy and it is evident in our results too.

Hyperparameter Search (report) [10] From now on, use the Glorot initialization method.

1. Find out a combination of hyper-parameters (model architecture, learning rate, nonlinearity, etc.) such that the average accuracy rate on the validation set ($r^{(valid)}$) is at least 97%.
2. Report the hyper-parameters you tried and the corresponding $r^{(valid)}$.

Tried multiple combinations of hyper parameters and found many combinations that achieves the accuracy of 97% on validation set.

HYPER-PARAMETERS
Model Architecture: (512,256), (512, 256, 128)
Learning Rate: 0.1, 0.01, 0.001, 0.0001
Non linearity: Relu, Sigmoid, Tanh

Hyper-Parameters				
Hidden Layers	Learning Rate	Non Linearity	Batch Size	Validation Accuracy
(512, 256, 128)	0.01	Sigmoid	32	0.8123
(512, 256, 128)	0.01	Relu	32	0.9716
(512, 256)	0.01	Tanh	64	0.9394
(512, 256)	0.001	Sigmoid	64	0.5368
(512, 256)	0.0001	Relu	64	0.7477
(512, 256)	0.01	Relu	32	0.9767
(512, 256)	0.1	Relu	64	0.9801

MODEL WITH 97% VALIDATION ACCURACY
Neuron Activation: **Relu**
Learning Rate: **0.01**
Batch Size: **32**
Input Dimensions: $\mathbf{784} \times 1$
Output Dimensions: $\mathbf{10} \times 1$
Hidden Layers: **2**
Hidden layer dimension: **512, 256**
Weights: $784 \times 512 + 515 \times 256 + 256 \times 10 = \mathbf{535050}$
Biases: $512 + 256 = \mathbf{768}$
Total No. of parameters: **535818**

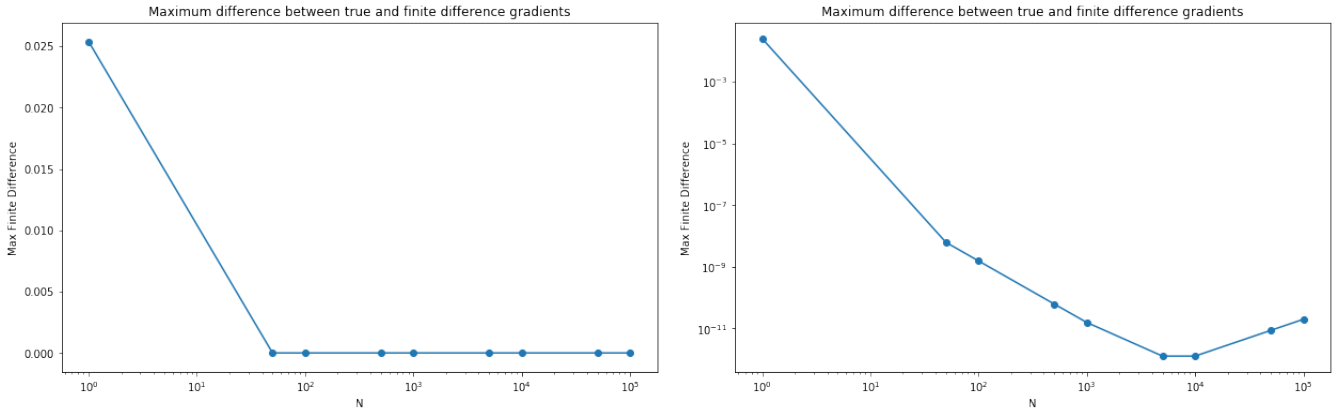
Validate Gradients using Finite Difference (report) [15] The finite difference gradient approximation of a scalar function $x \in \mathbb{R} \mapsto f(x) \in \mathbb{R}$, of precision ϵ, is defined as $\frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$. Consider the second layer weights of the MLP you built in the previous section, as a vector $\theta = (\theta_1, \ldots, \theta_m)$. We are interested in approximating the gradient of the loss function L, evaluated using **one** training sample, at the end of training, with respect to $\theta_{1:p}$, the first $p = \min(10, m)$ elements of θ, using finite differences.

1. Evaluate the finite difference gradients $\nabla^N \in \mathbb{R}^p$ using $\epsilon = \frac{1}{N}$ for different values of N

$$\nabla_i^N = \frac{L(\theta_1, \ldots, \theta_{i-1}, \theta_i + \epsilon, \theta_{i+1}, \ldots, \theta_p) - L(\theta_1, \ldots, \theta_{i-1}, \theta_i - \epsilon, \theta_{i+1}, \ldots, \theta_p)}{2\epsilon}$$

   Use at least 5 values of N from the set $\{k10^i : \ i \in \{0, \ldots, 5\}, \ k \in \{1, 5\}\}$.

2. Plot the maximum difference between the true gradient and the finite difference gradient ($\max_{1 \le i \le p} |\nabla_i^N - \frac{\partial L}{\partial \theta_i}|$) as a function of N. Comment on the result.



Both the graphs are same. The graph on the right is the log scaled version of y-axis to show the precision of the errors and to point out that for large N values, the error can increase because of rounding error due to finite precision.

N = [1, 50, 100, 500, 1000, 5000, 10000, 50000, 100000]
where (i,k) = [(0,1), (1,5), (2,1), (2,5), (3,1), (3,5), (4,1), (4,5), (5,1)]

As N increases, the ϵ decreases and the difference between true gradient and finite gradient also decreases. It is clear that with almost zero error, we can evaluate the gradient of the weights accurately with this method with a very small value of ϵ. This is also a nice way to test our gradient computation by comparing it against a finite-difference. Moreover, for low dimension functions, it is better to compute gradient using the finite-difference approximation instead of rolling the code to compute gradients.

Convolutional Neural Networks (report) [25] Many techniques correspond to incorporating certain prior knowledge of the structure of the data into the parameterization of the model. Convolution operation, for example, was originally designed for visual imagery. For this part of the assignment we will train a convolutional network on MNIST for 10 epochs using PyTorch. Plot the train and valid errors at the end of each epoch for the model.

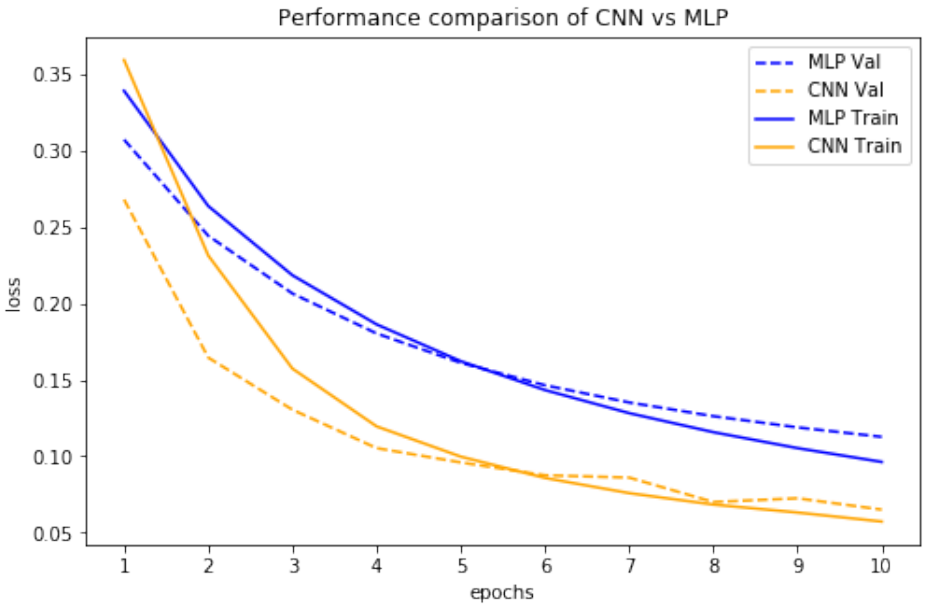
1. Come up with a CNN architecture with more or less similar number of parameters as MLP trained in Problem 1 and describe it.
2. Compare the performances of CNN vs MLP. Plot the training loss and validation loss curve.
3. Explore *one* regularization technique you've seen in class (e.g. early stopping, weight decay, dropout, noise injection, etc.), and compare the performance with a vanila CNN model without regularization.

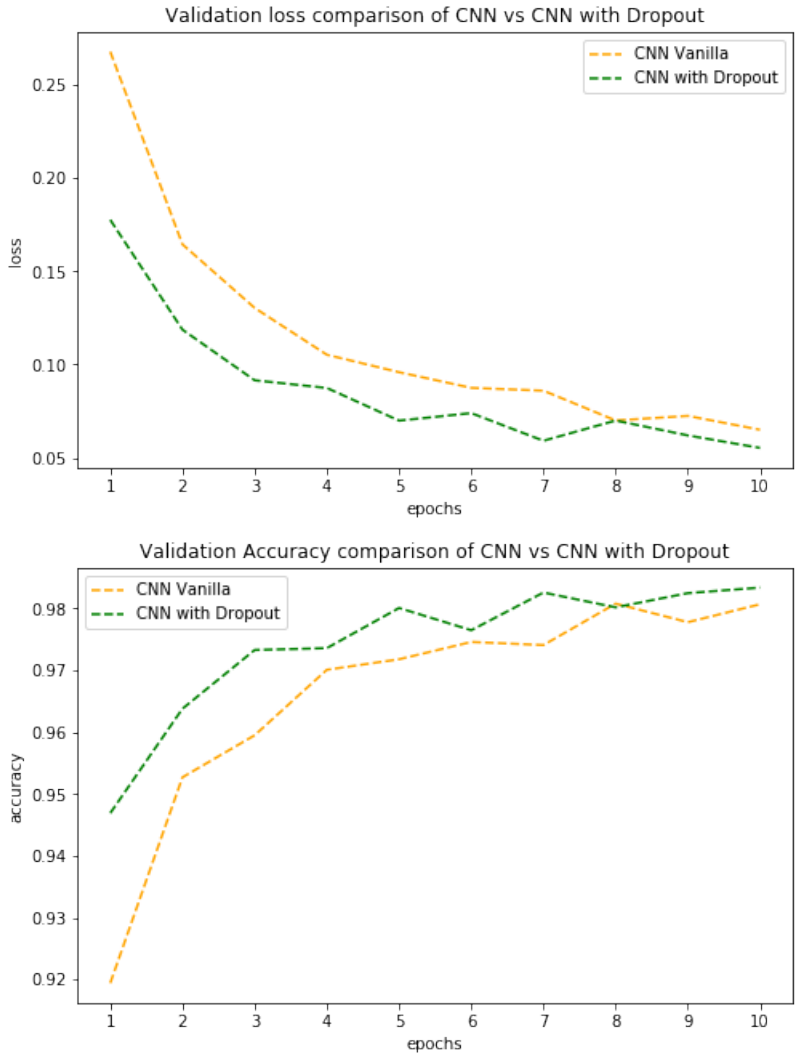
You could take inspiration from the architecture mentioned here (hyperlink).

CNN ARCHITECTURE with similar number of parameters

Conv Layer: $k = 64$, kernel size: (3,3) with Relu Activation
Pooling Layer: (2,2)
Conv Layer: $k = 32$, kernel size: (3,3) with Relu Activation
Pooling Layer: (2,2)
Fully Connected Layer: 256 with Relu Activation
Fully Connected Layer: 128 with Relu Activation
Fully Connected Layer: 10 with SoftMax Activation
Total Number of parameters: 462,858

CNN performed much better than the MLP on the MNIST dataset and a comparison is plotted in the graph below.





Although the model seems to be already performing good due to a simpler MNIST dataset,, but in general we can improve the model's performance by any regularization techniques. Here, I use **dropout** to increase the performance. It can be seen that the loss is decreased quickly in less number of epochs when compared to the vanilla CNN without regularizer. By the last epoch, the model with dropout outperforms the one without any regularization technique by some margin.

MLP vs CNN:
MLP is inefficient because it disregards the spatial information in the image and the input is converted to a 1D vector and then every neuron is connected to each input node thus increasing the number of parameters and making it difficult to learn and converge easily. The main difference is that the convolutional neural network (CNN) has layers of convolution and pooling and these layers take advantage of the local spatial coherence of the input. Each filter is panned around the entire image according to certain size and stride, and it allows the filter to find and match patterns in the image. The weights are shared among each other and thus is much easier and faster to train a CNN than a MLP. Every node does not connect to each node unlike in MLP where it takes a large number of parameters to do that.