CS69000-DPL Spring 2020 Homework — Homework 2— Last update: February 22, 2020

https://www.cs.purdue.edu/homes/ribeirob/courses/Spring2020 Due: **6:00am**, Sunday, February 9th

Instructions and Policy: Each student should write up their own solutions independently, no copying of any form is allowed. You MUST to indicate the names of the people you discussed a problem with; ideally you should discuss with no more than two other people.
YOU MUST INCLUDE YOUR NAME IN THE HOMEWORK
You need to submit your answer in PDF. LaTeX is typesetting is encouraged but not required. Please write clearly and concisely - clarity and brevity will be rewarded. Refer to known facts as necessary.

Q0 (0pts correct answer, -1,000pts incorrect answer: (0,-1,000) pts): A correct answer to the following questions is worth 0pts. An incorrect answer is worth -1,000pts, which carries over to other homeworks and exams, and can result in an F grade in the course.

(1) Student interaction with other students / individuals:

(a) I have copied part of my homework from another student or another person (plagiarism).

(b) Yes, I discussed the homework with another person but came up with my own answers. Their name(s) is (are) ____

(c) No, I did not discuss the homework with anyone

(2) On using online resources:

(a) I have copied one of my answers directly from a website (plagiarism).

(b) I have used online resources to help me answer this question, but I came up with my own answers (you are allowed to use online resources as long as the answer is your own). Here is a list of the websites I have used in this homework:
____

(c) I have not used any online resources except the ones provided in the course website.

Learning Objectives: Let students understand basic feed-forward neural networks and Backpropagation algorithm.

Learning Outcomes: After you finish this homework, you should be capable of explaining and implementing feed-forward neural networks with arbitrary architectures or components from scratch.

Concepts

Q1 (2.5 pts): Please answer the following questions **concisely**. All the answers, along with your name and email, should be clearly typed in some editing software, such as Latex or MS Word.

1. (0.5) Most practitioners will not use linear activations in deep multilayer perceptron. Prove that such deep neural networks would only be able to model linear functions (no matter how many layers or how many hidden neurons).

   If we only use linear activations in a neural network, the output entire network becomes a single matrix multiplication $W' = W_K \cdots W_2 W_1 \mathbf{x}$, where W_i are the weights of layer $i = 1, \ldots, K$. The same is true for the bias.

2. (0.5) Following the previous question, what if we place all the activations with Rectified Linear Unit (ReLU)? Would it solve the problem you mentioned in the previous question? Why or why not?

   ReLU is a non-linear activation, so that the output is not simply a linear combination operation over the input.

3. (0.5) Learning with ReLUs. Could a ReLU activation cause problems when learning a model with gradient descent? Could some layers of neural network to stop learning from data? Under which conditions?

   When a neuron with a ReLU activation receives negative inputs for all examples in the training data, the neuron is dead as the backpropagation will never update the weights of this neuron.

4. (0.5) Prove that unsupervised methods can be used to learn supervised tasks.

   Suppose a supervised task:

   (a) Given $\{(x_i^{\text{tr}}, y_i^{\text{tr}})\}_{i=1}^{n^{\text{tr}}}$, $(x_i^{\text{tr}}, y_i^{\text{tr}}) \sim \mathrm{P}(x, y)$

   (b) Given $\{(x_i^{\text{te}}, y_i^{\text{te}})\}_{i=1}^{n^{\text{te}}}$, $(x_i^{\text{te}}, y_i^{\text{te}}) \sim \mathrm{P}(x, y)$

   (c) - Model $\mathrm{P}(y|x)$

   It can be converted into two unsupervised tasks.

   (a) $\mathrm{P}(x, y)$

      i. Given $\{(x_i^{\text{tr}}, y_i^{\text{tr}})\}_{i=1}^{n^{\text{tr}}}$, $(x_i^{\text{tr}}, y_i^{\text{tr}}) \sim \mathrm{P}(x, y)$

      ii. Given $\{(x_i^{\text{te}}, y_i^{\text{te}})\}_{i=1}^{n^{\text{te}}}$, $(x_i^{\text{te}}, y_i^{\text{te}}) \sim \mathrm{P}(x, y)$

      iii. Model $\mathrm{P}(x, y)$

(b) $\mathrm{P}(x)$

i. Given $\{x_i^{\text{tr}}\}_{i=1}^{n^{\text{tr}}}$, $x_i^{\text{tr}} \sim \mathrm{P}(x) = \int_y \mathrm{P}(x,y)dy$

ii. Given $\{x_i^{\text{te}}\}_{i=1}^{n^{\text{te}}}$, $x_i^{\text{te}} \sim \mathrm{P}(x) = \int_y \mathrm{P}(x,y)dy$

iii. Model $\mathrm{P}(x)$

Then $\mathrm{P}(y|x) = \frac{\mathrm{P}(x,y)}{\mathrm{P}(x)}$.

5. (0.5) In multitask learning we are given a dataset $\mathcal{D} = \{(x_i^{(\text{tr})}, y_{i,1}^{(\text{tr})}, \dots, y_{i,T}^{(\text{tr})})\}_{i=1}^{n^{(\text{tr})}}$, where the ys are labels of T distinct tasks. Our goal is to learn $p(y_1, \dots, y_T|x)$ from $\mathcal{D}$. Describe the difference between multitask learning and transfer learning. Consider the task of facial recognition to distinguish males from females. To train this model, suppose we have a small dataset with images of humans but a very large dataset with pictures of dogs and cats. Should we use multitask learning or transfer learning. And how?

In multitask learning, we are learning all labels jointly

$$\mathrm{P}(y_1, y_2, \cdots, y_T|x),$$

but in transfer learning, we are using other labels help to learn a label, e.g.

$$\mathrm{P}(y_1|x, y_2, y_3, \cdots, y_T).$$

where T is the number of different tasks.

We should use transfer learning to train. First, we train an image feature extractor and a classifier used to distinguish dogs and cats over extracted features jointly on the large dataset. Then, we copy the model and parameters of image feature extractor, use it to extract features of facial images, and train another classifier used to distinguish male and female over extracted features separately.

Programming

Throughout this semester you will be using Python and PyTorch as the main tool to complete your homework, which means that getting familiar with them is required. PyTorch (`http://pytorch.org/tutorials/index.html`) is a fast-growing Deep Learning toolbox that allows you to create deep learning projects on different levels of abstractions, from pure tensor operations to neural network blackboxes. The official tutorial and their github repository are your best references. Please install the latest stable version on your machine (version 0.4.1 with cuda 8.0 was used while I was creating this homework). Linux machines with GPU installed are suggested. Moreover, following PEP8 coding style is recommended.

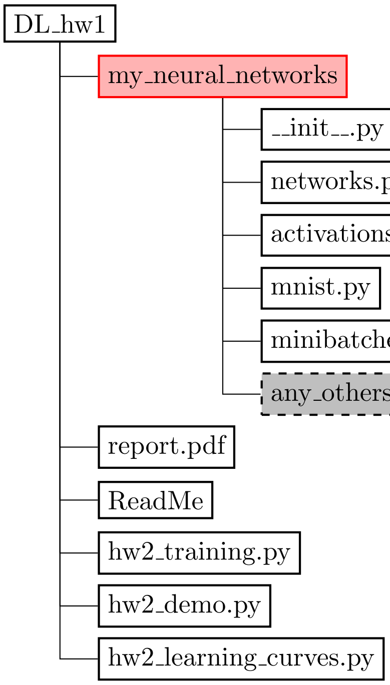
Skeleton Package: A skeleton package is available on the github website "`https://github.com/gao462/Fall2018CS690DL/tree/master/HW1`". You should download it and use the folder structure provided. In some homework, skeleton code might be provided. If so, you should based on the prototype to write your implementations.

Introduciton to PyTorch

PyTorch, in general, provides three modules, from high-level to low-level abstractions, to build up neural networks. We are going to study 3 specific modules in this homework. First, the module that provides the highest abstraction is called **torch.nn**. It offeres layer-wise abstraction so that you can define a neural layer through a function call. For example, **torch.nn.Linear(.)** creates a fully connected layer. Coupling with contains like **Sequential(.)**, you can connect the network layer-by-layer and thus easily define your own networks. The second module is called **torch.AutoGrad**. It allows you to compute gradients with respect to all the network parameters, given the feed-forward function definition (the objective function). It means that you don't need to analytically compute the gradients, but only need to define the objective function while coding your networks. The last module we are going to use is **torch.tensor** which provides effecient ways of conducting tensor operations or computations so that you can customize your network in the low-level. The official PyTorch has a thorough tutorial to this (`http://pytorch.org/tutorials/beginner/pytorch_with_examples.html#`). You are required to go through it and understand all three modules well before you move on.

HW Overview

In this homework, you are going to implement vanilla feed-forward neural networks om a couple of different ways. The overall submission should be structured as below:



- **DL_hw2**: the top-level folder that contains all the files required in this homework. You should replace the file name with your name and follow the naming convention mentioned above.
- **report.pdf**: Your written solutions to all the homework questions, including theoretical and programming parts. Should be submitted in pdf format.

- **ReadMe**: Your ReadMe should begin with a couple of **example commands**, e.g., ”python hw2.py data”, used to generate the outputs you report. TA would replicate your results with the commands provided here. More detailed options, usages and designs of your program can be followed. You can also list any concerns that you think TA should know while running your program. Note that put the information that you think it's more important at the top. Moreover, the file should be written in pure text format that can be displayed with Linux ”less” command.
- **hw2_training.py**: One executable we prepared for you to run training with your networks.
- **hw2_learning_curves.py**: One executable for training models and plotting learning curves.
- **hw2_learning_demo.py**: Demonstrate some basic Python packages. Just FYI.
- **my_neural_networks**: Your Python neural network package. The package name in this homework is **my_neural_networks**, which should NOT be changed while submitting it. Two modules should be at least included:
  - **networks.py**
  - **activations.py**

  Except these two modules, a package constructor **__init__.py** is also required for importing your modules. You are welcome to architect the package in your own favorite. For instance, adding another module, called utils.py, to facilitate your implementation.

  Two additional modules, **mnist.py** and **minibatcher.py**, are also attached, and are used in the main executable to load the dataset and create minibatches (which is not needed in this homework.). You don't need to do anything with them.

Data: MNIST

You are going to conduct a simple classification task, called MNIST (`http://yann.lecun.com/exdb/mnist/`). It classifies images of hand-written digits (0-9). Each example thus is a 28 × 28 image.

- The full dataset contains 60k training examples and 10k testing examples.
- We provide a data loader (read_images(.) and read_labels(.) in **my_neural_networks/mnist.py**) that will automatically download the data.

Warm-up: Implement Activations

Open the file **my_neural_networks/activations.py**. As a warm up activity, you are going to implement the **activations** module, which should realize activation functions and objective functions that will be used in your neural networks. Note that whenever you see ”raise NotImplementedError”, you should implement it.

Since these functions are mathematical equations, the code should be pretty short and simple. The main intuition of this section is to help you get familiar with basic Python programming, package structures, and test cases. As an example, a Sigmoid function is already implemented in the module. Here are the functions that you should complete:

- **relu**: Rectified Linear Unit (ReLU), which is defined as

$$a_k^l = relu(z_k^l) = \begin{cases} 0 & \text{if } z_k^l < 0 \\ z_k^l & \text{otherwise .} \end{cases}$$

- **softmax**: the basic softmax

$$a_k^L = softmax(z_k^L) = \frac{e^{z_k^L}}{\sum_c e^{z_c^L}}, \tag{1}$$

- **stable_softmax**: the **numerically stable softmax**. You should test if this outputs the same result as the basic softmax.

$$\begin{aligned} softmax(x_i) &= \frac{e^{x_i}}{\sum_j e^{x_j}} && (2) \\ &= \frac{Ce^{x_i}}{C\sum_j e^{x_j}} && (3) \\ &= \frac{e^{x_i + \log C}}{\sum_j e^{x_j + logC}} && (4) \end{aligned}$$

  A common choice for the constant is $logC = -\max_j x_j$.

- **cross_entropy**:

$$E = -\sum_d t_d \log a_k^L = -\sum_d t_d (z_d^L - \log \sum_c e^{z_c^L}). \tag{5}$$

  where d is a data point; t_d is its true label; a_k^L is the propability predicted by the network.

Hints: make sure you tested your implementation with corner cases before you move on. Otherwise, it would be hard to debug.

Warm-up: Understand Example Network

Open the files **hw2_training.py** and **my_neural_networks/example_networks.py**.

hw2_training.py is the main executable (trainer). It controls in a high-level view. The task is called MNIST, which classifies images of hand-written digits. The executable uses a class called **TorchNeuralNetwork** fully implemented in **my_neural_networks/example_networks.py**.

In this task, you don't need to write any codes, but only need to play with the modules/executables provided in the skeleton and answer qeustions. A class called **TorchNeuralNetwork** is fully implemented in **my_neural_networks/example_networks.py**. You can run the trainer with it by feeding correct arguments into **hw2_training.py**. Read through all the related code and write down what is the correct command (”python hw2_training.py” with arguments) to train such example networks in the report.

Here is a general summary about each method in the **TorchNeuralNetwork**.

- **__init__(self, shape, gpu_id=-1)**: the constructor that takes network shape as parameters. The network weights are declared as matrices in this method. You should not make any changes to them, but need to think about how to use them to do vectorized implementations.
  - Your implementation should support arbitrary network shape, rather than a fixed one. The shape is in specified in tuples. For exapmles, "shape=(784, 100, 50, 10)" means that the numbers of neurons in the input layer, first hidden layer, second hidden layer, and output layer are 784, 100, 50, and 10 respectively.
  - All the hidden layers use **ReLU** activations.
  - The output layer uses **Softmax** activations.
  - **Cross-Entropy** loss should be used as the objective.
- **train_one_epoch(self, X, y, y_1hot, learning_rate)**: conduct network training for one epoch over the given data **X**. It also returns the loss for the epoch.
  - this method consists of three important components: feed-forward, backpropagation, and weight updates.
  - (Non-stochastic) **Gradient descent** is used. The gradient calculatation should base on all the input data. However, this part is given.
- **predict(self, X)**: predicts labels for **X**.

You need to understand the entire skeleton well at this point. **TorchNeuralNetwork** should give you a good starting point to understand all the method semantics, and the **hw2_training.py** should demonstrate the training process we want. In the next task, you are going to implement another two classes supporting the same set of methods. The inputs and outputs for the methods are the same, while the internal implementations have different constrains. Therefore, make sure you understand all the method semantics and inputs/outputs before you move on.

Q2 (2 pts): Implement Feedforward Neural Network with Autograd

Open the file **my_neural_networks/networks.py**.

The task here is to complete the class **AutogradNeuralNetwork**. In your implementation, several constrains are enforced:

- You are NOT allowed to use any high-level neural network modules, such as torch.nn, unless it is specified. No credits will be given if similar packages or modules are used.
- You need to follow the methods prototypes given in the skeleton. This contrain might be removed in the future. However, as the first homework, we want you to know what do we expect you to complete in a PyTorch project.
- You should left at least the **hw2_training.py** untouched in the final submission. During grading, we will replace whatever you have with the original **hw2_training.py**.

For **AutogradNeuralNetwork**, you only need to complete the **feed-forward part**. Other parts should already be given in the skeleton. You should be able to run the **hw2_training.py** in a way similar to what you discovered in the last task. Specifically, what you need to is as follows:

- Understand semantics of all the class members (variables), especially the few defined in the constructor.
- Identify the codes related three main components for training: feed-forward, backpropagation, and weight updates.
- The second and third components are given. Only the **feed-forward** is left for you, so go ahead and complete the **_feed_forward()** method.

Things to be included in the report: **Random seed of PyTorch is fixed by 29 for all the following codes**.

1. command line arguments for running this experiment with **hw2_training.py**.

   I run command

   ```
   python hw\homeworknumber_training.py -e 100 -l 0.001 -i torch.autograd -g 0 -v mnist/gz
   ```

   to train samples from data folder ‘mnist/gz‘ by **AutogradNeuralNetwork** with learning rate 0.001 for 100 epochs on GPU-0.

2. Specify network shape as (784, 300, 100, 10). Collect results for **100 epochs**. Make two plots: ”Loss vs. Epochs” and ”Accuracy vs. Epochs”. The accuracy one should include results for both training and testing data. Analyze and compare each plot generated in the last step. Write down your observations.

   There is a small gap between performance of training and testing data.

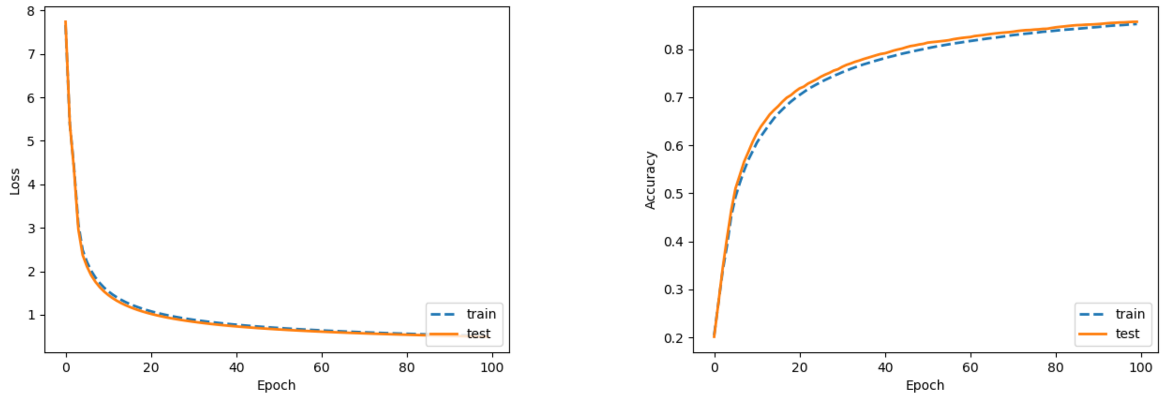


Figure 1: *
Loss VS. Epochs

Figure 2: *
Accuracy VS. Epochs

Figure 3: Q2.2

Hints:

- The given skeleton has all the input/output definitions. Please read through it, and if you found any typos or unclear parts, feel free to ask.
- In general, you don't need to change any codes given in the skeleton, unless it is for debugging.
- Feel free to define any helper functions/modules you need.
- You might need to figure out how to conduct vectorized implementations so that the pre-defined members can be utilized in a succinct and efficient way.
- You are welcome to use GPUs to accelerate your program
- For debugging, you might want to load less amount of training data to save time. This can done easily by make slight changes to **hw2_training.py**.
- For debugging, you might want to explore some features in a Python package called **pdb**.

Q3 (1 pts): Learning Curves: Deep vs Shallow

Create a trainer file called **hw2_learning_curves.py**

This executable has very similar structure to the **hw2_training.py**, but you are going to vary training data size to plot learning curves introduced in the lecture. Specifically, you need to do the followings:

1. Load MNIST data: `http://yann.lecun.com/exdb/mnist/` into torch tensors
2. Use **AutogradNeuralNetwork**.
3. Vary training data size ranged from 250 to 10000. You can decide a proper step.
4. Train and select a model for each data size. You need to design an **early stop** strategy to select the model so that the learning curves will be correct.
5. Plot learning curves for training and testing sets with
   (a) a network shape (784, 10)
   (b) a network shape (784, 300, 100, 10)

Things that should be included in the report:

- command line arguments for running this experiment with **hw2_learning_curves.py**.
- The early stop strategy you used in selecting models.
- The 2 learning curve plots for the 2 network shapes.
- Analyze and compare each plot generated in the last step. Write down your observations.

I run command

```
python hw\homeworknumber_learning_curves.py -e 100 -l 0.001 -i torch.autograd -g 0 -a 2 -v mnist/gz
python hw\homeworknumber_learning_curves.py -e 100 -l 0.001 -i torch.autograd -g 0 -a 4 -v mnist/gz
```

to train samples from data folder ‘mnist/gz‘ by **AutogradNeuralNetwork** with learning rate 0.001 for 100 epochs on GPU-0 for shape (784, 10) and (784, 300, 100, 10).

For early stopping, I will only keep the model with best test accuracy in memory. If test accuracy become worse, it will load the best model from disk, otherwise, it will save to disk.

Here are learning curves

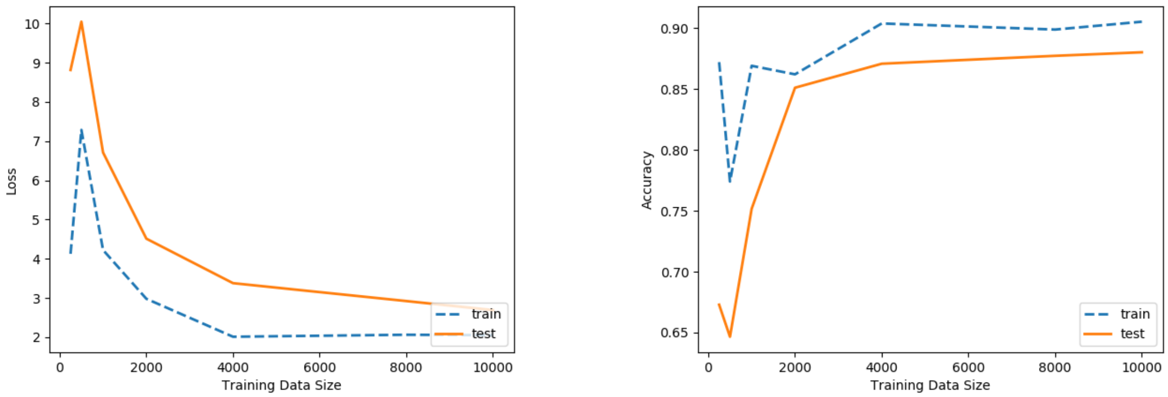


Figure 4: *
Loss VS. Training Data Size

Figure 5: *
Accuracy VS. Training Data Size

Figure 6: Q3 (784, 10)

With number of training samples increasing, the gap between performance of training and testing data are becoming smaller and smaller. Training performance has a tendency to decrease, while test performance have a tendency to increase. Deep network is more stable with this tendency than shallow network.

Hints: You should understand the information embedded in the learning curves and what it should look like. If your implementation is correct, you should be able to see meaningful differences.

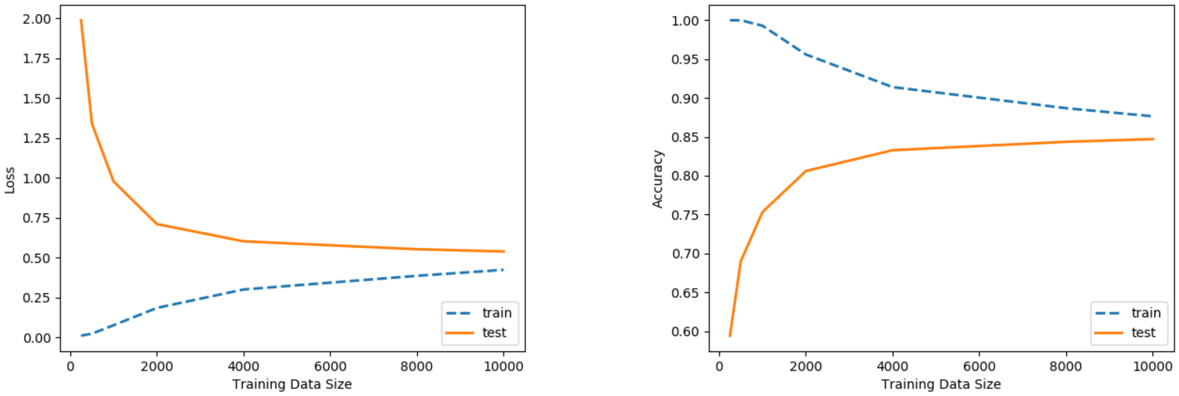


Figure 7: *
Loss VS. Training Data Size

Figure 8: *
Accuracy VS. Training Data Size

Figure 9: Q3 (784, 300, 100, 10)

Q4 (4.5 pts): Implement Backpropagation from Scratch

Open the file **my_neural_networks/networks.py**.

Implement **BasicNeuralNetwork**, but you can NOT use **torch.Autograd**. All the other instructions are similar to what is in Q2. That is, you need to implement the entire ”train_one_epoch” method, including backpropagation, feed forward, and weight updates. For the backpropagation, you need to analytically compute the gradients.

Here, we will use pytorch the same way that we have used numpy in the lecture notes. You will need to write your own backpropagation function from scratch, following what would be the correct gradients of the already-implemented forward pass.

Things to be included in the report:

1. (2.0) Implement the above in the file provided (**networks.py**). Make sure your code runs with the command line arguments for running **hw2_training.py**. Points will only be awarded if the code runs with the original command line.

   See uploaded files.

2. (1.0) Write down all the mathematical formulas used in your backpropagation implementation.

   - Suppose input x of a layer is a 2D matrix of #Features$_1$ × #Samples
   - Suppose output y of a layer is a 2D matrix of #Features$_2$ × #Samples
   - Suppose possible one-hot target $y^{\{0,1\}}$ is a 2D matrix of #Classes × #Samples

Cross Entropy Loss and Softmax Activation

For feed forward process

$$L = \text{cross\_entropy}(\text{softmax}(x), y^{\{0,1\}})$$

The back propagate process will be

$$\frac{\partial L}{\partial x} = \frac{1}{\#\text{Samples}}(\text{softmax}(x) - y^{\{0,1\}}) \tag{6}$$

Linear

For feed forward process

$$y = Wx + b$$

The back propagate process with gradient $\frac{\partial L}{\partial y}$ of y will be

$$\begin{aligned} \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial y} \\ \frac{\partial L}{\partial W} &= \frac{\partial L}{\partial y} x^{\mathrm{T}} \\ \frac{\partial L}{\partial x} &= W^{\mathrm{T}} \frac{\partial L}{\partial y} \end{aligned} \tag{7}$$

ReLU Activation

For feed forward process

$$y = \text{relu}(x)$$

The back propagate process with gradient $\frac{\partial L}{\partial y}$ of y will be

$$\frac{\partial L}{\partial x_{ij}} = \begin{cases} 0 & x_{ij} < 0 \\ \frac{\partial L}{\partial y_{ij}} & \text{o.w.} \end{cases} \tag{8}$$

3. (0.5) Use **BasicNeuralNetwork**. Specify network shape as (784, 300, 100, 10). Collect results for **100 epochs**. Write in your report PDF two plots: ”Loss vs. Epochs” and ”Accuracy vs. Epochs”. The accuracy one should include results for both training and testing data. Analyze and compare each plot generated in the last step. Write down your observations.

   I run command

   ```
   python hw\homeworknumber_training.py -e 100 -l 0.001 -i my -g 0 -v mnist/gz
   ```

   to train samples from data folder ‘mnist/gz‘ by **BasicNeuralNetwork** with learning rate 0.001 for 100 epochs on GPU-0.

4. (1.0) Modify **hw2_learning_curves.py** to support creating learning curves of **BasicNeuralNetwork**. ll the other instructions are similar to what is in Q3. **Things should be included in the report**:

   - Implement the above in the file provided (**hw2_learning_curves.py**). Command line arguments for running this experiment with **hw2_learning_curves.py**.
   - The early stop strategy you used in selecting models.
   - The 2 learning curve plots for the 2 network shapes.

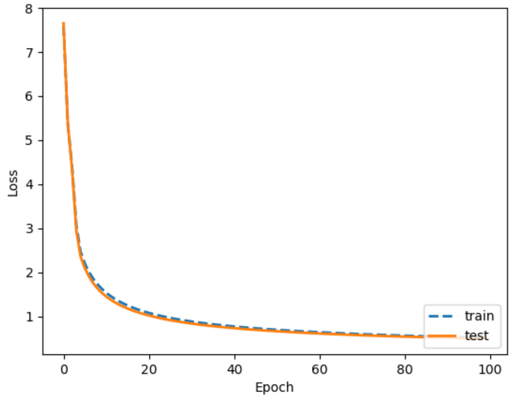


Figure 10: *
Loss VS. Epochs

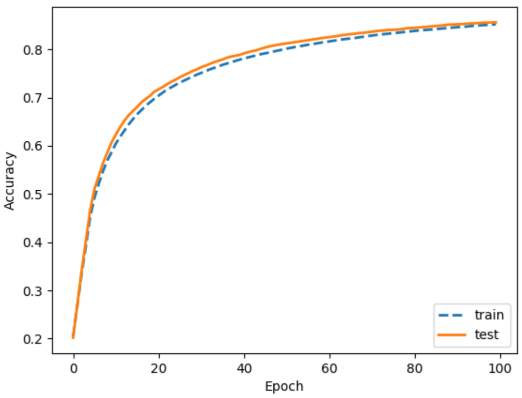


Figure 11: *
Accuracy VS. Epochs

Figure 12: Q4.3

- Analyze and compare each plot generated in the last step. Write down your observations in the report PDF.

I run command

```
python hw\homeworknumber_learning_curves.py -e 100 -l 0.001 -i my -g 0 -a 2 -v mnist/gz
python hw\homeworknumber_learning_curves.py -e 100 -l 0.001 -i my -g 0 -a 4 -v mnist/gz
```

to train samples from data folder ‘mnist/gz‘ by **BasicNeuralNetwork** with learning rate 0.001 for 100 epochs on GPU-0 for shape (784, 10) and (784, 300, 100, 10).

For early stopping, I will only keep the model with best test accuracy in memory. If test accuracy become worse, it will load the best model from disk, otherwise, it will save to disk.

Here are learning curves

The observations are same as Q3.

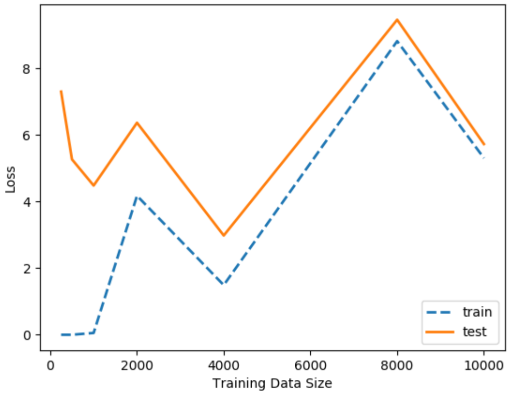


Figure 13: *
Loss VS. Training Data Size

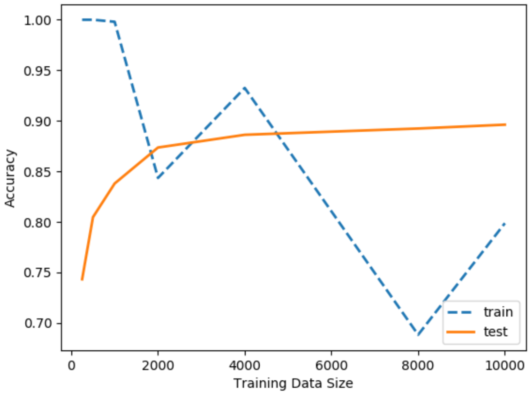


Figure 14: *
Accuracy VS. Training Data Size

Figure 15: Q4.4 (784, 10)

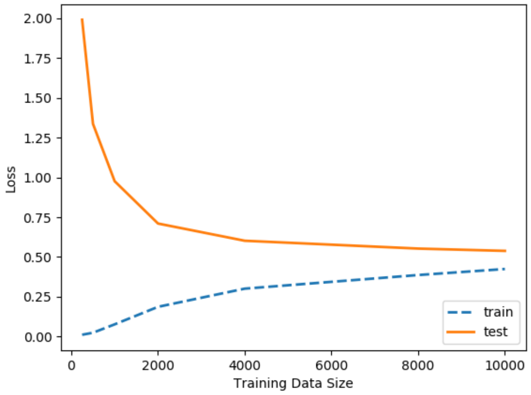


Figure 16: *
Loss VS. Training Data Size

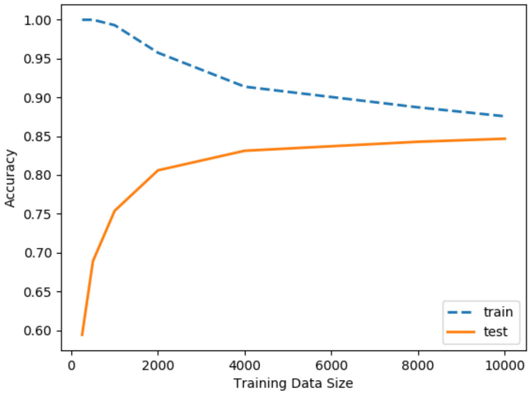


Figure 17: *
Accuracy VS. Training Data Size

Figure 18: Q4.4 (784, 300, 100, 10)