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.

Name: Amit Roy
PUID: 0034778209

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) Yucheng Zhang, Taisuke Mori.**
 (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: stackoverflow.com, pytorch and numpy documentation, stackexchange.com, medium.com and and discussion forums.**
 (c) I have not used any online resources except the ones provided in the course website.

Learning Objectives: Let students understand deep learning tasks, basic feedforward neural network properties, backpropagation, and invariant representations.

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

Concepts

Q1 (2.0 pts): In what follows we give a paper and ask you to classify the paper into tasks.

Mark ALL that apply and EXPLAIN YOUR ANSWERS. Answers without explanations will get deducted -0.05 points.

Note 1: This includes marking both a specific answer and its more general counterpart. E.g., Covariate shift adaptation is also a type of Domain adaptation. Your answer explanation can help assign partial credits.

Note 2: *Papers may describe multiple tasks. Please make sure you describe which task you focused on in "Explain Your Answers"..*

Point distribution:

- Each question starts with 0.5 points.
- Each missing task counts as -0.1 (should be marked but was not).
- Each extra task counts as -0.1 (was marked but should not).
- Each MARKED answer not explained in "Explain Your Answers" gets deducted -0.05. Items left unmarked need not be explained.
  - Example of an explanation: The image task in the paper is a supervised learning task: the training data is $\{(x_i, y_i)\}_i$, where x_i is an image and y_i is the image's label. The data $\{(x_i, y_i)\}_i$ is assumed independent, where the train and test distributions are assumed to be the same. The learning is transductive since the test data is provided during training in the form of an extra dataset where...
- The minimum score is zero.

1. (0.5) Kipf, Thomas N., and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks." In International Conference on Learning Representations, 2017.

   (a) Independent observations
   (b) **Dependent observations**
   (c) Supervised learning
   (d) Unsupervised learning
   (e) Self-supervised learning
   (f) **Semi-supervised learning**
   (g) **In-distribution test data**
   (h) Out-of-distribution test data
   (i) Transfer learning
   (j) **Transductive learning**
   (k) Inductive learning
   (l) Covariate shift adaptation
   (m) Target shift adaptation
   (n) Domain adaptation
   (o) **Associational task (i.e., not causal)**
   (p) Causal task

   Explain your answers

   - The node classification task in GCN paper is an example of **dependent observation** as it deals with graphs where each node u depends on its neighborhood (each row of the adjacency matrix A_u) for learning the node representation.
   - The node classification task is also a **semi-supervised learning** task where a small number of training nodes contain labels as visible in Table 1 in the paper.
   - The node classification task is an example of **in-distribution test data** as it is a transductive task hence both the training and test data is from the same distribution.
   - The node classification task of GCN is also a **transductive learning** task because GCN is trained on a fixed size graph A which is required to compute $H^{l+1} = \sigma(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^lW^l)$ and can not classify unseen nodes at the test time.
   - The node classification task of GCN is not a causal task. We classify the nodes based on their representation obtained from the GCN layer hence it is an **associational task** with no causality involved here.

2. (0.5) Ho, Jonathan, Ajay Jain, and Pieter Abbeel. "Denoising diffusion probabilistic models." Advances in Neural Information Processing Systems 33 (2020): 6840-6851.

   (a) **Independent observations**
   (b) Dependent observations
   (c) Supervised learning
   (d) **Unsupervised learning**
   (e) Self-supervised learning
   (f) Semi-supervised learning
   (g) **In-distribution test data**
   (h) Out-of-distribution test data
   (i) Transfer learning
   (j) Transductive learning
   (k) Inductive learning
   (l) Covariate shift adaptation
   (m) Target shift adaptation
   (n) Domain adaptation
   (o) **Associational task (i.e., not causal)**
   (p) Causal task

   Explain your answers

   - The image generation task from noise via the denoising diffusion probabilistic models is an example of **independent observation** as the generation of images doesn't depend on each other.
   - The image generation task from noise via the denoising diffusion probabilistic models is an **unsupervised learning** task as there is no ground truth label available.
   - The image generation task from noise via the denoising diffusion probabilistic models is an example of **in-distribution test data** as the diffusion model can generate examples from the trained distribution only.
   - The image generation task from noise via the denoising diffusion probabilistic models is an example of the **associational task** as there is no causality involved here

3. (0.5) Kenton, Jacob Devlin Ming-Wei Chang, and Lee Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In Proceedings of NAACL-HLT, pp. 4171-4186. 2019.

   (a) **Independent observations**
   (b) Dependent observations
   (c) **Supervised learning**
   (d) Unsupervised learning
   (e) Self-supervised learning
   (f) Semi-supervised learning
   (g) In-distribution test data
   (h) Out-of-distribution test data
   (i) **Transfer learning**
   (j) Transductive learning
   (k) **Inductive learning**
   (l) Covariate shift adaptation
   (m) Target shift adaptation
   (n) Domain adaptation
   (o) **Associational task (i.e., not causal)**
   (p) Causal task

   Explain your answers

   - The BERT model takes sentences as input where the representation of each word in a sentence depends on the words present in its context. So, it can be thought of as an example of dependent observations within a single sentence. But the sentences are independent hence it is **independent observations**.
   - The BERT model is pre-trained with unlabeled data and it uses its own masked input as labels while pre-training so it can be thought of as an example of self-supervised learning. However, the BERT model is finetuned using labeled data for the downstream tasks hence it is an example of **supervised learning**.
   - The BERT model is first pre-trained and then finetuned on supervised tasks. So, it is an example of **transfer learning**.
   - The BERT model is an example of **inductive learning** as it can work well on unseen data not availble in the training time.

4. (0.5) S Chandra Mouli, Bruno Ribeiro, “Asymmetry Learning for Counterfactually-invariant Classification in OOD Tasks”, International Conference on Learning Representations, 2022.

   (a) Independent observations
   (b) Dependent observations
   (c) **Supervised learning**
   (d) Unsupervised learning
   (e) Self-supervised learning
   (f) Semi-supervised learning
   (g) In-distribution test data
   (h) **Out-of-distribution test data**
   (i) Transfer learning
   (j) Transductive learning
   (k) **Inductive learning**
   (l) **Covariate shift adaptation**
   (m) Target shift adaptation
   (n) **Domain adaptation**
   (o) Associational task (i.e., not causal)
   (p) **Causal task**

   Explain your answers

   - The task define in this paper uses labeled data (X^{tr}, Y^{tr}) hence it is an **supervised learning** task.
   - The task used in this paper can perform well on shifted images data which is an example of **out-of-distribution** test data.
   - The task used in this paper can perform well on shifted data and can be considered an **inductive learning** task.
   - The task in this paper classifies shifted images which is a shift of the input feature therefore it's an example of **Covariate shift adaptation** and **Domain adaptation**.
   - This paper described the task of finding symmetries that affect the label as a causal structure discovery task and show that, under certain conditions, we can use the predictive performance of invariant representations on the observational data to predict whether an edge exists in the causal DAG. Hence it is an **causal task**.

Q2 (3.0 pts): Please answer the following questions **concisely** but with enough details to get partial credit if the answer is incorrect.

1. (0.5) For neural network models, regularization terms are usually applied on weight parameters. Describe why we do not regularize the bias term in the model?
   Hint: Biases are activation thresholds. Assume a very large regularization on the bias. Now assume an input of all zeros.

   A neuron can be represented by the following equation: $y = \sigma(W^\top X + b)$

   where W and b are the weight matrix and bias vector. If the input x is all zeros and a very high regularization is applied on the bias b then the regularization on bias will force the bias b to be close to zero. Hence, the neuron will always output zero for the training example. Also, the bias term b is the activation threshold that is used to shift the output of the model which is different from the purpose of model parameters w which defines the model complexity. If the bias terms are regularized then they must be regularized in a different manner and not like the weight parameters otherwise they will the model will suffer from overfitting.

2. (0.5) Learning feedforward networks with ReLUs: Could a ReLU activation cause problems when learning a model with gradient descent? Let $\{X_i\}_{i=1}^N, X_i \in \mathbb{R}^d$ be the training data. Let $W_j^{(1)} \in \mathbb{R}^d$ be the j-th neuron weight in the first layer. Give a non-trivial subset $\mathcal{W}$ where $\forall W_j^{(1)} \in \mathcal{W}$ the gradient $\frac{\partial L(\{X_i\}_{i=1}^N)}{\partial W_j^{(1)}}$ is zero.
   Hint: We say a neuron is *dead* when its output is constant for all training examples.

   ReLU activation can cause problems when learning a model with gradient descent. If the learning rate is too high or with a negative bias, the ReLU function will permanently return output zero due to negative inputs of the ReLU function. This phenomenon is called "dying relu". Also, a high learning rate can cause exploding gradient with ReLU activation.

   If $W_j^{(1)} = \mathbf{0} and b = \mathbf{0}$ for all the j-th neuron weight in the first layer then the output of all the neurons of the first layer will be constant (zero vector). Hence, the gradient $\frac{\partial(L\{X_i\}_{i=1}^N)}{\partial W_j^{(1)}=0}$

3. (0.5) Suppose the symmetrization (Reynolds operator) $\bar{T}$ of a finite linear transformation group G has rank 0. Prove that any G-invariant neuron has just the bias term.
Hint: For example, a zero matrix has rank 0.

A neuron is G-invariant if $\sigma(W^\top x' + b) = \sigma(w^\top X + b)$ where $x' = \bar{T}x, \bar{T} \in G$ and $w^\top = \sum_{i=1}^k \alpha_i v_i$

Here, α_i are the coefficients and v_i are the eigenvectors that span the right eigenspace corresponding to the eigenvalue 1 of the Reynolds operator $\bar{T}$ for the group G. If the $\bar{G}$ has rank 0 then $k = 0$ and $w^\top = 0$ and the G-invariant neuron will be $\sigma(b)$, hence only the bias term.

4. (0.5) Consider a supervised learning task, where the training data is $(Y, X) \sim P^{\text{tr}}(Y, X)$ and $A \sim b$ means random variable A is sampled from distribution b. Consider two finite linear transformation groups G_1 and G_2. Let $f_i : \mathbb{R}^d \to [0, 1]$ be single neuron that is G_i-invariant, for $i = 1, 2$. Describe how we could test (and be sure) that f_1 is also invariant to G_2 or f_2 is also invariant to G_1. Assume we have access to the neuron weights $\mathbf{w_i}$ of f_i, $i = 1, 2$.
Hint 1: Just testing the f's with some transformed inputs will not guarantee they are invariant to all inputs and all transformations.
Hint 2: Pay attention to the invariant subspace that defines the parameters of f_1 and f_2, which forces the neurons to be G_1- and G_2-invariant, respectively.

Suppose, the neuron f_1 and f_2 have weights $w_1 = \sum_{i=1}^{k_1} \alpha_i v_{i,1}$ and $w_2 = \sum_{i=1}^{k_2} \alpha_i v_{i,2}$ respectively. We can check whether each of the k_1 eigenvector $v_{i,1}$ of the first neuron f_1 can be represented as a linear combination of the set of eigenvectors $v_{i,2}, i = 1, 2, ..., k_2$ to check whether f_1 is G_2 invariant. To check whether f_2 is G_1-invariant we need to check whether each of the k_2 eigenvector $v_{i,2}$ of the second neuron f_2, can be represented as a linear combination of the eigenvectors $v_{i,1}, i = 1, 2, ..., k_1$ of the first neuron f_1.

5. (1.0) Consider the neurons f_1 and f_2 of the previous question. For each of the groups $i = 1, 2$, let $\bar{T}_i$ be the symmetrization (Reynolds) operator of group G_i. We also have a set of left eigenvectors vectors $\mathbf{v}_{i,k}^T \bar{T}_i = \mathbf{v}_{i,k}^T$, $k = 1, \ldots, K$, for each of the groups. Explain how to create a neuron that is both G_1- and G_2-invariant. Prove that this neuron is invariant to any composition of transformations from both G_1 and G_2, such as $T_1 T_2 T_1'$, $T_1, T_1' \in G_1$, $T_2 \in G_2$.

   To create a neuron f_3 that is both G_1- and G_2-invariant we can create a third group G_3 that will satisfy $v_{i,k}^\top \bar{T}_i = v_{i,k}^\top, k = 1,, K$. The new neuron will span the intersection of the eigenspaces of f_1 and f_2. Suppose the eigenspace of the neuron f_1 and f_2 is represented by U and V.

   Now, Let z be a vector that lies in the intersection of U and V. So, z can be written as a linear combination of the vectors of U and V. Therefore,

$$\begin{aligned}
z &= Ux = Vy \\
Ux &= Vy \\
U^\top Ux &= U^\top Vy \\
x &= (U^\top U)^{-1} U^\top Vy \\
\text{Similarly, } y &= (V^\top V)^{-1} (V^\top U) x \\
\text{Therefore, } x &= (U^\top U)^{-1} U^\top V (V^\top V)^{-1} (V^\top U) x \\
\text{And } x &= \hat{M} x \\
\text{where } \hat{M} &= (U^\top U)^{-1} U^\top V (V^\top V)^{-1} (V^\top U)
\end{aligned}$$

   So, x is a eigenvector of $\hat{M}$ with an eigenvalue of 1. The required basis that is the intersection of eigenspaces of two neurons is the linearly independent eigenvectors such that

$$Ux : \hat{M}x = x \tag{1}$$

   In another way, we can define $\hat{M}$ as the product of two projection matrices of eigenspace U and V. We can write them as $P_u = U(U^\top U)^{-1} U^\top$ and $P_v = V(V^\top V)^{-1} V^\top$

   Now if a vector s lies in the subspace of U and V, it will remain unchanged.

   Therefore, $s = P_u s$ and $s = P_v s$.

   We can write $s = P_u P_v s$ and $s(P_u P_v - I) = 0$.

   So, we can perform eigendecomposition of the matrix $P_u P_V - I$ and the eigenvectors that correspond to zero or nearly zero eigenvectors will be in the intersection of the eigenspace of U and V. We can use them to define the neuron f_3 that will be G-invariant to both f_1 and f_2.

Programming (5.0 pts)

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 make sure you have the latest stable version on the machine. Linux machines with GPU installed are suggested. Moreover, following PEP8 coding style is recommended.

Skeleton Package: A skeleton package is available at
`https://www.dropbox.com/s/fsfxp1g2kgmw0kv/hw1_skeleton.zip?dl=0`. 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.

Introduction 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 feedforwardfunction 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 feedforwardneural networks on a couple of different ways. The overall submission should be structured as below:

```
bruno_ribeiro_hw1
├── my_neural_networks
│   ├── __init__.py
│   ├── networks.py
│   ├── activations.py
│   ├── mnist.py
│   ├── minibatcher.py
│   └── any_others.py
├── ReadMe
├── hw1_training.py
├── hw1_demo.py
├── hw1_learning_curves.py
└── G-equivariant_Auto-Encoder
    ├── G-equivariant-AE-skeleton.ipynb
    ├── example-G-equi-ae-output.png
    ├── example-ae-output.png
    └── example_image.png
```

- **bruno_ribeiro_hw1**: 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.
- **ReadMe**: Your ReadMe should begin with a couple of **example commands**, e.g., "python hw1.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.
- **hw1_training.py**: One executable we prepared for you to run training with your networks.
- **hw1_learning_curves.py**: One executable for training models and plotting learning curves.
- **hw1_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.

- **G-equivariant_Auto-Encoder**: Follow the python notebook inside for constructing a G-equivariant Auto-Encoder step-by-step.

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

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

- **softmax**: the basic softmax

$$a_k^L = \text{softmax}(z_k^L) = \frac{e^{z_k^L}}{\sum_c e^{z_c^L}}, \tag{2}$$

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

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{3}$$
$$= \frac{Ce^{x_i}}{C\sum_j e^{x_j}} \tag{4}$$
$$= \frac{e^{x_i + \log C}}{\sum_j e^{x_j + logC}} \tag{5}$$

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

- **average cross_entropy**:

$$E = -\frac{1}{\text{sizeof(mini-batch)}} \sum_{d \in \text{mini-batch}} t_d \log a_k^L = -\frac{1}{\text{sizeof(mini-batch)}} \sum_{d \in \text{mini-batch}} t_d (z_d^L - \log \sum_c e^{z_c^L}). \tag{6}$$

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 **hw1_training.py** and **my_neural_networks/example_networks.py**.

hw1_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 **hw1_training.py**. Read through all the related code and write down what is the correct command ("python hw1_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: feedforward, 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 **hw1_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.

Q3 (1 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 **hw1_training.py** untouched in the final submission. During grading, we will replace whatever you have with the original **hw1_training.py**.

For **AutogradNeuralNetwork**, you only need to complete the **feedforward part**. Other parts should already be given in the skeleton. You should be able to run the **hw1_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.

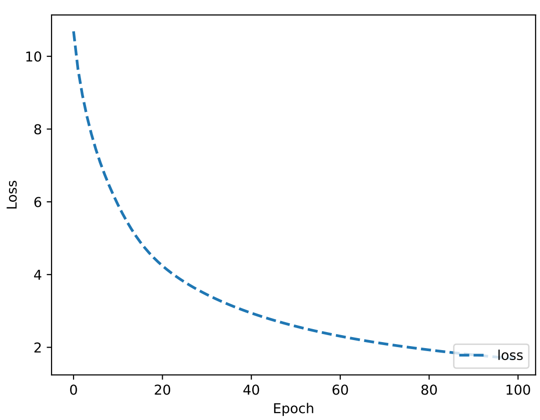
- Identify the codes related three main components for training: feedforward, backpropagation, and weight updates.
- The second and third components are given. Only the **feedforward** is left for you, so go ahead and complete the **_feed_forward()** method.

Things to be included in the report:

1. command line arguments for running this experiment with **hw1_training.py**.

```
python hw1_training.py ./data -e 100 -l 1e-4 -i torch.autograd -g 0 -m -1 -n -1
```

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.



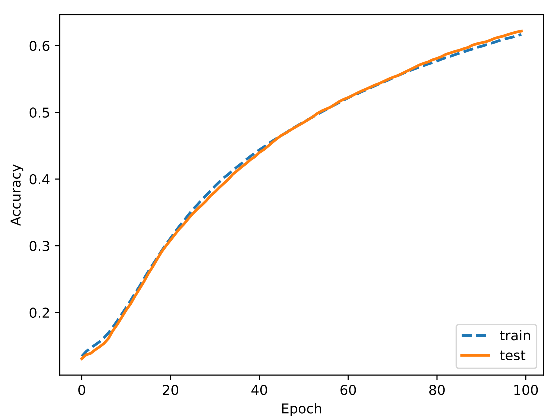


Figure 1: Loss vs Epochs and Accuracy vs. Epochs for network shape (784,300,100,10)

We can observe from Figure 2 that the training loss decreases which indicates that the model is learning. Also for the accuracy plot, the accuracy is increasing over the epochs and test accuracy is **62.18%** after 100 epochs and also the accuracy **82.56%** if we train the model for 500 epochs.

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 **hw1_training.py**.
- For debugging, you might want to explore some features in a Python package called **pdb**.

Q4 (1 pts): Learning Curves: Deep vs Shallow

Create a trainer file called **hw1_learning_curves.py**

This executable has very similar structure to the **hw1_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 in the range between 250 to 10000 in increments of 250.
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 **hw1_learning_curves.py**.

```
python hw1_learning_curves.py ./data -e 500 -n 1000 -l 1e-6 -i torch.autograd -g 0
```

- The early stop strategy you used in selecting models.

  If the absolute loss difference between two consecutive epoch is less then $1e-3$ then we stop training the model.
- The 2 learning curve plots for the 2 network shapes.
- Analyze and compare each plot generated in the last step. Write down your observations.

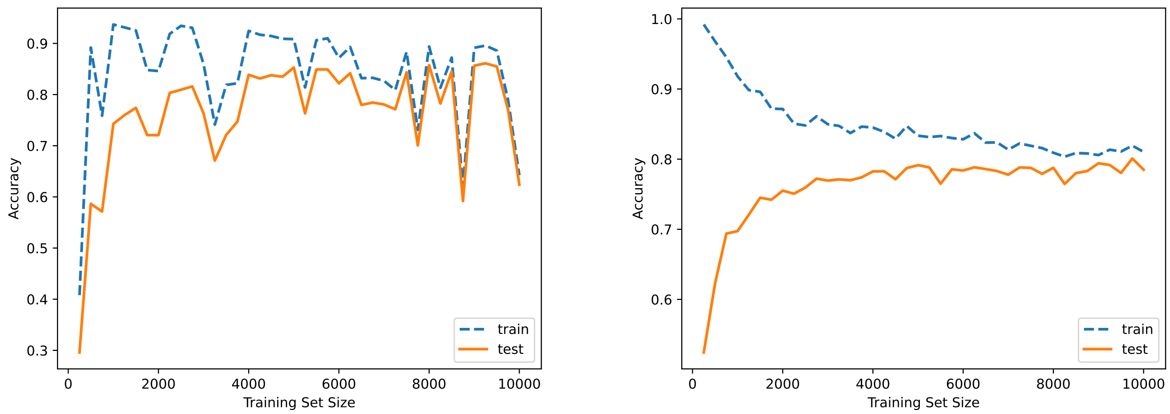


Figure 2: Learning curve for network shape (784,10) and (784,300,100,10)

As the size of the dataset increases the test accuracy increases and training accuracy decreases. For large models with shapes (784, 300, 100, 10), this behavior is more visible as visible in the right plot whereas for small models with shape (784, 10) this behavior is not that clear.

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.

Q5 (2.0 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, 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. (0.5) Implement the above in the file provided (**networks.py**). Make sure your code runs with the command line arguments for running **hw1_training.py**. Points will only be awarded if the code runs with the original command line.

   Reduced the learning rate for avoiding the exploding gradient issue.

   ```
   python hw1_training.py ./data -e 100 -l 1e-4 -i my -g 0 -m -1 -n -1
   ```

2. (0.5) Write down all the mathematical equations used in your backpropagation implementation.

   Forward Propagation:

$$
\begin{aligned}
z^{(1)} &= x^\top W^{(1)} + b^{(1)} \\
h^{(1)} &= ReLU(z^{(1)}) \\
z^{(2)} &= h^{(1)\top} W^{(2)} + b^{(2)} \\
h^{(2)} &= ReLU(z^{(2)}) \\
z^{(3)} &= h^{(2)\top} W^{(3)} + b^{(3)} \\
\hat{y} &= softmax(z^{(3)}) \\
L &= ylog(\hat{y})
\end{aligned}
$$

   Back-Propagation:

   For the output layer:

$$
\begin{aligned}
\frac{\partial L}{\partial z^{(3)}} &= (\hat{y} - y) \\
\frac{\partial L}{\partial W^{(3)}} &= \frac{\partial z^{(3)}}{\partial W^{(3)}} \frac{\partial L}{\partial z^{(3)}} = h^{(2)\top}(\hat{y} - y) \\
\frac{\partial L}{\partial b^{(3)}} &= \frac{\partial z^{(3)}}{\partial b^{(3)}} \frac{\partial L}{\partial z^{(3)}} = (\hat{y} - y)
\end{aligned}
$$

For 2-nd layer:

$$
\begin{aligned}
\frac{\partial L}{\partial h^{(2)}} &= \frac{\partial z^{(3)}}{\partial h^{(2)}} \frac{\partial L}{\partial z^{(3)}} = W^{(3)}(\hat{y} - y) \\
\frac{\partial L}{\partial W^{(2)}} &= \frac{\partial z^{(2)}}{\partial W^{(2)}} \frac{\partial h^{(2)}}{\partial z^{(2)}} \frac{\partial L}{\partial h^{(2)}} \\
\frac{\partial L}{\partial b^{(2)}} &= \frac{\partial z^{(2)}}{\partial b^{(2)}} \frac{\partial h^{(2)}}{\partial z^{(2)}} \frac{\partial L}{\partial h^{(2)}} \\
\frac{\partial z^{(2)}}{\partial W^{(2)}} &= h^{(1)\top} \\
\frac{\partial z^{(2)}}{\partial b^{(2)}} &= 1 \\
\frac{\partial h^{(2)}}{\partial z^{(2)}} &= \begin{cases} 1 & \text{if } z^{(2)} > 0 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

For input layer:

$$
\begin{aligned}
\frac{\partial L}{\partial h^{(1)}} &= \frac{\partial z^{(2)}}{\partial h^{(1)}} \frac{\partial L}{\partial z^{(2)}} = W^{(2)} \frac{\partial h^{(2)}}{\partial z^{(2)}} \frac{\partial L}{\partial h^{(2)}} \\
\frac{\partial L}{\partial W^{(1)}} &= \frac{\partial z^{(1)}}{\partial W^{(1)}} \frac{\partial h^{(1)}}{\partial z^{(1)}} \frac{\partial L}{\partial h^{(1)}} \\
\frac{\partial L}{\partial b^{(1)}} &= \frac{\partial z^{(1)}}{\partial b^{(1)}} \frac{\partial h^{(1)}}{\partial z^{(1)}} \frac{\partial L}{\partial h^{(1)}} \\
\frac{\partial z^{(1)}}{\partial W^{(1)}} &= X^\top \\
\frac{\partial z^{(1)}}{\partial b^{(1)}} &= 1 \\
\frac{\partial h^{(1)}}{\partial z^{(1)}} &= \begin{cases} 1 & \text{if } z^{(1)} > 0 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

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.

   We can observe from Figure 3 that the training loss decreases which indicates that the model is learning. Also for the accuracy plot, the accuracy is increasing over the epochs and test accuracy is **70.52**% after 100 epochs.

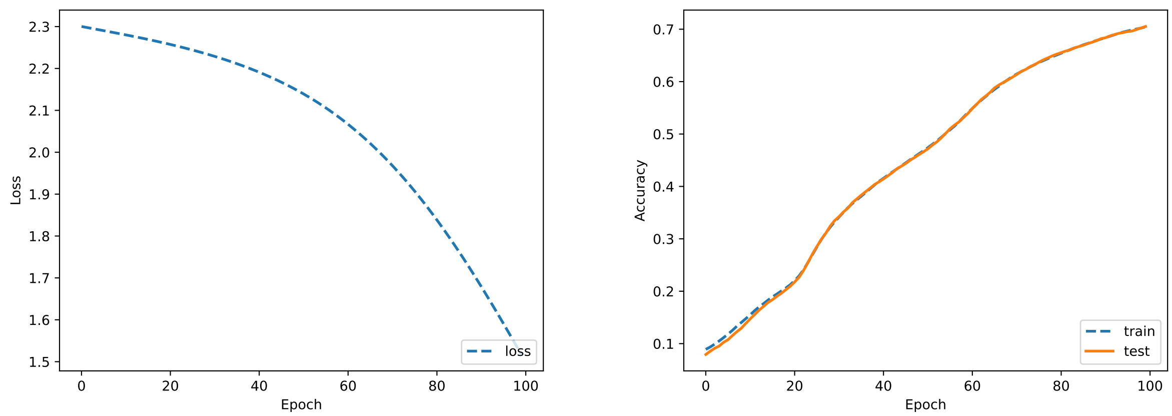


Figure 3: Loss vs Epochs and Accuracy vs. Epochs for network shape (784,300,100,10)

4. (0.5) Modify **hw1_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 (**hw1_learning_curves.py**). Command line arguments for running this experiment with **hw1_learning_curves.py**.

   ```
   python hw1_learning_curves.py ./data -e 500 -n 1000 -l 1e-6 -i my -g 0
   ```

   - The early stop strategy you used in selecting models.
     If the absolute loss difference between two consecutive epoch is less then $1e-3$ then we stop training the model.
   - The 2 learning curve plots for the 2 network shapes.
   - Analyze and compare each plot generated in the last step. Write down your observations in the report PDF.

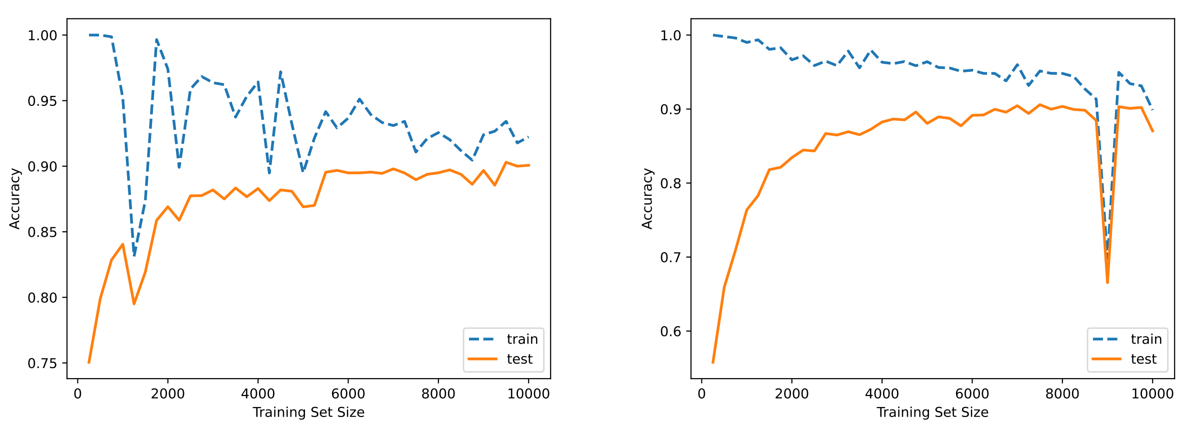


Figure 4: Learning curve for network shape (784,10) and (784,300,100,10)

We can observe from Figure 4 that as the size of the dataset increases the test accuracy increases and training accuracy decreases. For large models with shapes (784, 300, 100, 10), we can see a performance drop but for small models (784, 10) the accuracy change follows a zig-zag pattern.

Q6 (2.0 pts): Implement an G-Equivariant Autoencoder

Open the python notebook file **G-equivariant_Auto-Encoder/G-equivariant-AE-skeleton.ipynb**. Follow the instruction step-by-step to implement an G-equivariant Auto-Encoder that is equivariant to transformations in the group formed by 90^o rotations (of bounded squared images).

Consider a bounded square image input defined as a column vector in $\mathbb{R}^m$. Consider a single layer whose hidden neuron output is in $\mathbb{R}^k$, such that unvectorizing it also gives a square. Let $\mathbf{W} \in \mathbb{R}^{k\times m}$ be the neuron parameters. G-equivariance requires that transforming the input is the same as transforming the output: $\mathbf{x} \in \mathbb{R}^m, \forall g \in G, \quad \rho_2(g)\mathbf{W}\mathbf{x} = \mathbf{W}\rho_1(g)\mathbf{x}$, where $\rho_1 : G \to \mathbb{R}^{\sqrt{m}\times\sqrt{m}}$ and $\rho_2 : G \to \mathbb{R}^{\sqrt{k}\times\sqrt{k}}$. Since the above is true for all $\mathbf{x}$, then $\rho_2(g)\mathbf{W}\rho_1(g)^{-1} = \mathbf{W}$, or equivalently $\forall g \in G,\ \rho_2(g) \otimes \rho_1(g^{-1})^T \text{vec}(\mathbf{W}) = \text{vec}(\mathbf{W})$, where vec() flattens the matrix into a vector. Please complete the missing parts of the skeleton python notebook to implement a G-equivariant Auto-Encoder. Save the .ipynb notebook as part of your tar.gz file for your code upload.

1. (0.25) Describe all the transformations in your transformation group **(in the PDF)**. For instance, T^{90} the transformation that rotates the image counter-clockwise by 90 degrees must of course be present. $T^{90}, T^{180}, T^{270}, T^{360}$

2. (1.25) This part grades your .ipynb notebook code. Your MLP is trained with upright digits. Show in the .ipynb notebook that the accuracy of your outputs in both in-distribution (upright digits) and out-of-distribution (rotated digits) is approximately the same. Compare one predicted output of your G-equivariant MLP against the output of the traditional MLP, showing how they differ.

3. (0.5) Can you explain your choice of the dimension of the bias parameter in the G-equivariant layers. Why making a different choice could break the equivariance property?

   The choice of the dimension of the bias parameter should be 1 in the G-equivariant layers. The purpose of the bias term in a neuron is to shift the output of the weight parameter W multiplied by the input X or previous layer $h^{(l-1)}$. For the G-equivariant layers, the bias should be shared by all neurons of a particular layer. Hence, the bias dimension should be 1. If a different bias is chosen for the different neurons of a particular layer then it will shift the output of the neurons in a different way and not preserve the equivariance property.

Submission Instructions

Please read the instructions carefully. Failed to follow any part might incur some score deductions.

PDF upload

The report PDF must be uploaded on Gradescope (see link on Brightspace)

Code upload

Naming convention: [firstname]_[lastname]_hw1

All your submitting code files, a ReadMe, should be included in one folder. The folder should be named with the above naming convention. For example, if my first name is "Bruno" and my last name is "Ribeiro", then for Homework 1, my file name should be "bruno_ribeiro_hw1".

Tar your folder: [firstname]_[lastname]_hw1.tar.gz

Remove any unnecessary files in your folder, such as training datasets. Make sure your folder structured as the tree shown in Overview section. Compress your folder with the the command: **tar czvf bruno_ribeiro_hw1.tar.gz czvf bruno_ribeiro_hw1** .

Submit: **TURNIN INSTRUCTIONS**

Please submit your compressed file on **data.cs.purdue.edu** by turnin command line, e.g. **"turnin -c cs690dpl -p hw1 bruno_ribeiro_hw1.tar.gz"**. Please make sure you didn't use any library/source explicitly forbidden to use. If such library/source code is used, you will get 0 pt for the coding part of the assignment. If your code doesn't run on scholar.rcac.purdue.edu, then even if it compiles in another computer, your code will still be considered not-running and the respective part of the assignment will receive 0 pt.