CS58700 Spring 2026 Homework — Homework 1— Last update: February 3, 2026
https://www.cs.purdue.edu/homes/ribeirob/courses/Spring2026 Due: **11:59pm**, Wednesday, Feb. 18th (open until 4:00am next day)

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 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 (2.4 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"..*

Note 3: *Do not take the author's words for granted!* Many papers claim they are doing one task (e.g. OOD generalization), but in reality, their methods are really doing another task (e.g. domain adaptation or few-shot learning). Use your own judgment and make sure that the task can be formally cast into one of the tasks discussed in the lecture.

Point distribution:

- Each question starts with 0.4 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. (0.4) Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, Ilya Sutskever. "Learning Transferable Visual Models From Natural Language Supervision." International Conference on Machine Learning (ICML), 2021.

   (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

2. (0.4) Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, Guillaume Lample "LLaMA: Open and Efficient Foundation Language Models." arXiv preprint arXiv:2302.13971, 2023.

   (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

3. (0.4) Jianan Zhao, Zhaocheng Zhu, Mikhail Galkin, Hesham Mostafa, Michael Bronstein, Jian Tang, Fully-inductive Node Classification on Arbitrary Graphs, ICLR 2025.

   (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

4. (0.4) Yangyi Shen, Jincheng Zhou, Beatrice Bevilacqua, Joshua Robinson, Charilaos Kanatsoulis, Jure Leskovec, Bruno Ribeiro, “Zero-Shot Generalization of GNNs over Distinct Attribute Domains”, International Conference on Machine Learning (ICML), 2025.

   (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

5. (0.4) Mahmoud Assran, Quentin Duval, Ishan Misra, Piotr Bojanowski, Pascal Vincent, Michael Rabbat, Yann LeCun, Nicolas Ballas, “Self-Supervised Learning from Images with a Joint-Embedding Predictive Architecture”, IEEE/CVF International Conference on Computer Vision, 2023.

   (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

6. (0.4) Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, Ryan Lowe, "Training language models to follow instructions with human feedback", Advances in neural information processing systems, 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

Q2 (2.6 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.

   The reason we do not regularize the bias, is because the role of the bias to prevent the network from calculating the gradient of a value of zero. If we regularize the bias, with a large value, then we will diminish its value and make it zero, causing the entire loss function 0 and when we backpropagate, we will run into an error

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.

   Yes a ReLU could cause problems learning a model with gradient descent if a bias does not exist.

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.

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 only have access to the neuron weights* $\mathbf{w_i}$ *of* f_i*,* $i = 1, 2$ *and the Reynolds operator* $\overline{T}_1$ *and* $\overline{T}_2$ *for the groups* G_1 *and* G_2*, respectively.*
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.

5. (0.6) 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 $T_1 T_2 T_1'$, $T_1, T_1' \in G_1$, $T_2 \in G_2$.

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.

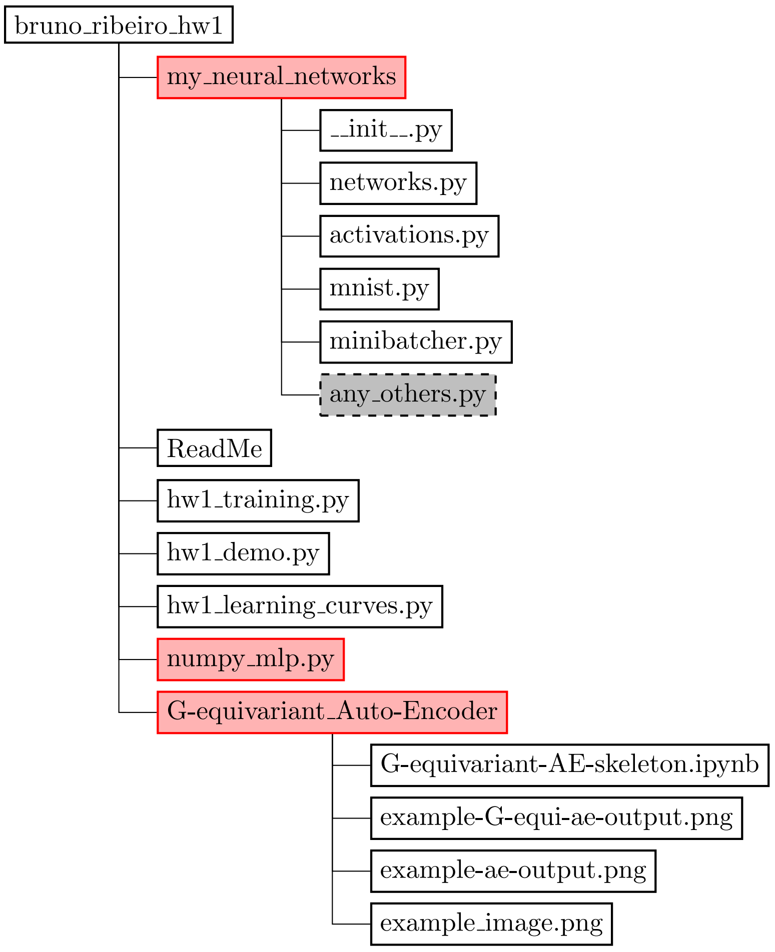
Skeleton Package: A skeleton package is available at **Brightspace**. You should download it and use the folder structure provided. In some homework, skeleton code might be provided. If so, you should based on the prototype to write your implementations.

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 feedforward 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. Note that in most cases, we don't need to directly import methods from torch.AutoGrad. Most of the time, it is sufficient to call **loss.backward()**, which calls torch.AutoGrad functionalities under the hood and computes the backpropagation gradients for us. 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**: 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.

- **numpy_mlp.py**: Your numpy-only three-layer DenseNet-style MLP with Layer Normalization and dense connections. The training skeleton code is provided, but you will need to implement the forward, backward (including LayerNorm gradients), and SGD update all in numpy.
- **G-equivariant_Auto-Encoder**: Follow the python notebook inside for constructing a G-equivariant Auto-Encoder with G-invariant attention pooling.

Data: MNIST

You are going to conduct a simple classification task, called 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.

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

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

- **average cross_entropy**:

$$E = -\frac{1}{\text{sizeof(mini-batch)}} \sum_{d \in \text{mini-batch}} t_d \log a_k^L = -\frac{1}{\text{sizeof(mini-batch)}} \sum_{d \in \text{mini-batch}} t_d (z_d^L - \log \sum_c e^{z_c^L}). \tag{5}$$

  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 **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 (e.g., torch.nn.Linear, torch.nn.Sequential), only torch.Tensor, torch.nn.Parameter, and basic autograd functionalities are permitted. No credits will be given if high-level NN modules are used.
- You must follow the method prototypes given in the skeleton (do not modify input/output signatures).
- Leave **hw1_training.py** untouched in your final submission (we will use the original version for grading).

For **AutogradNeuralNetwork**, you need to complete three key components:

- **Weight/Bias Initialization**: Implement the **init_weights()** method to initialize weights and biases with a uniform distribution scaled by $1/\sqrt{n_{in}}$ (where n_{in} is the number of input neurons to the layer). This ensures stable initialization and avoids vanishing/exploding gradients.

- **Feedforward Pass**: Complete the **_feed_forward()** method to compute the weighted sums ($Wx+b$) and activated outputs (ReLU for hidden layers, stable softmax for output layer) for all layers. Use the pre-defined `self.weights` and `self.biases` (torch.nn.Parameter objects) for vectorized computations.
- **Parameter Update**: In **train_one_epoch()**, implement the gradient descent update step (including gradient zeroing for parameters) after backpropagation. Ensure gradients are reset to zero before the next epoch to avoid gradient accumulation, and update weights/biases using the computed gradients and learning rate.
- **Prediction Logic**: In **predict()**, implement the logic to return class predictions (argmax over the output layer activations) for input data.

Things to be included in the report:

1. Command line arguments for running this experiment with **hw1_training.py**.
2. Use network shape (784, 300, 100, 10) and train for 100 epochs. Plot: (1) "Loss vs. Epochs" (training loss) (2) "Accuracy vs. Epochs" (training + testing accuracy) Analyze and compare each plot generated in the last step. Write down your observations.
3. Discuss why gradient zeroing is critical for correct training, and what happens if this step is omitted.

Hints:

- Use vectorized operations (avoid for-loops over individual examples/neurons) for efficiency, i.e., to leverage PyTorch's tensor broadcasting.
- For stable softmax in the output layer, use your implemented `stable_softmax` from **activations.py**.
- Gradients for `torch.nn.Parameter` objects are stored in `.grad.data`, ensure you do not overwrite the `.grad` attribute directly (modify `.grad.data` instead).
- Test your weight initialization by printing the mean/std of weights for the first layer.

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

Take a look at the 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: `https://huggingface.co/datasets/ylecun/mnist` into torch tensors
2. Use **AutogradNeuralNetwork**.
3. Vary training data size in the range between 250 to 10000 in increments of 250.
4. For each training data size:
   (a) Instantiate the neural network.
   (b) Train the model with an **early stop strategy**.
   (c) Select the final model based on your early stop criteria to ensure valid learning curves.
5. Complete the `save_learning_curve` function to plot training and testing accuracy curves for:
   (a) A shallow network shape: $(784, 10)$ (input dimension: 784, output dimension: 10)
   (b) A deep network shape: $(784, 300, 100, 10)$
6. Ensure full functionality of the `one_hot` encoding function (verify/complete its implementation to correctly encode labels for training).

Things that should be included in the report:

- Full command-line arguments used to run the experiment with **hw1_learning_curves.py**.
- A description of your early stop strategy and rationale for design choices.
- Two learning curve plots (one for each network shape), with clear labels for axes, legends (training/test accuracy), and network shape annotation.
- Analysis and comparison of the two learning curves generated in the last step. Write down your observations in detail.

Hints:

- Verify the `one_hot` function correctly encodes labels (critical for training with cross-entropy loss).
- The early stop strategy must balance stopping too early (unconverged model) and too late (overfitting).
- 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): Backpropagation with Dense Residual MLP and Layer Normalization

Open the file **numpy_mlp.py**.

Implement **NumpyDenseResMLP** as a three-layer MLP with dense (DenseNet-style) connections and Layer Normalization, using only **numpy**. You need to implement forward propagation, backpropagation, and weight updates entirely in numpy, analytically deriving all gradients.

Architecture Definition

The DenseNet-style MLP with Layer Normalization is defined as follows. Given a mini-batch input $\mathbf{X} \in \mathbb{R}^{B\times d}$ with batch size B, dimension d, and ground-truth labels $\mathbf{y} \in \{1, \ldots, C\}^B$. We denote AppendOnes$(\cdot)$ as the operation of appending a column of all-ones to the right side of a matrix (to incorporate bias into weight matrices).

Layer Normalization is defined as:

$$\mathrm{LN}(\mathbf{z}; \boldsymbol{\gamma}, \boldsymbol{\beta}) = \boldsymbol{\gamma} \odot \hat{\mathbf{z}} + \boldsymbol{\beta}, \quad \text{where}$$
$$\hat{\mathbf{z}}_j = \frac{\mathbf{z}_j - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad \mu = \frac{1}{h}\sum_{j=1}^{h} \mathbf{z}_j, \quad \sigma^2 = \frac{1}{h}\sum_{j=1}^{h} (\mathbf{z}_j - \mu)^2$$

where $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^h$ are learnable scale and shift parameters, $\odot$ is element-wise multiplication, and $\epsilon = 10^{-5}$ for numerical stability. Note that μ and σ^2 are computed **per sample** across the hidden dimension.

Forward Pass:

1. **Layer 1** with $\mathbf{W}^{(1)} \in \mathbb{R}^{(d+1)\times h}$:

$$\begin{aligned} \overline{\mathbf{X}} &= \mathrm{AppendOnes}(\mathbf{X}), \\ \mathbf{Z}^{(1)} &= \overline{\mathbf{X}}\mathbf{W}^{(1)}, \\ \mathbf{N}^{(1)} &= \mathrm{LN}(\mathbf{Z}^{(1)}; \boldsymbol{\gamma}^{(1)}, \boldsymbol{\beta}^{(1)}), \\ \mathbf{H}^{(1)} &= \max(0, \mathbf{N}^{(1)}) && \text{ReLU activation} \end{aligned}$$

2. **Layer 2 (Dense Connection)** with $\mathbf{W}^{(2)} \in \mathbb{R}^{(d+h+1)\times h}$:

$$\begin{aligned} \mathbf{C}^{(1)} &= [\mathbf{X} \| \mathbf{H}^{(1)}] \in \mathbb{R}^{B\times(d+h)} && \text{Concatenate input and Layer 1 output} \\ \overline{\mathbf{C}}^{(1)} &= \mathrm{AppendOnes}(\mathbf{C}^{(1)}), \\ \mathbf{Z}^{(2)} &= \overline{\mathbf{C}}^{(1)}\mathbf{W}^{(2)}, \\ \mathbf{N}^{(2)} &= \mathrm{LN}(\mathbf{Z}^{(2)} + \mathbf{H}^{(1)}; \boldsymbol{\gamma}^{(2)}, \boldsymbol{\beta}^{(2)}) && \text{Residual before LayerNorm} \\ \mathbf{H}^{(2)} &= \max(0, \mathbf{N}^{(2)}) \end{aligned}$$

3. **Layer 3 (Dense Connection)** with $\mathbf{W}^{(3)} \in \mathbb{R}^{(d+2h+1)\times h}$:

$$\begin{aligned}
\mathbf{C}^{(2)} &= [\mathbf{X} \| \mathbf{H}^{(1)} \| \mathbf{H}^{(2)}] \in \mathbb{R}^{B\times(d+2h)} && \text{Concatenate all previous outputs} \\
\overline{\mathbf{C}}^{(2)} &= \text{AppendOnes}(\mathbf{C}^{(2)}), \\
\mathbf{Z}^{(3)} &= \overline{\mathbf{C}}^{(2)} \mathbf{W}^{(3)}, \\
\mathbf{N}^{(3)} &= \text{LN}(\mathbf{Z}^{(3)} + \mathbf{H}^{(2)}; \boldsymbol{\gamma}^{(3)}, \boldsymbol{\beta}^{(3)}) && \text{Residual from } \mathbf{H}^{(2)} \\
\mathbf{H}^{(3)} &= \max(0, \mathbf{N}^{(3)})
\end{aligned}$$

4. **Output Layer** with $\mathbf{W}^{(4)} \in \mathbb{R}^{(h+1)\times C}$:

$$\begin{aligned}
\overline{\mathbf{H}}^{(3)} &= \text{AppendOnes}(\mathbf{H}^{(3)}), \\
\mathbf{Z}^{(4)} &= \overline{\mathbf{H}}^{(3)} \mathbf{W}^{(4)}, \\
\hat{\mathbf{Y}} &= \mathbf{Z}^{(4)} - \log\left(\sum_{j=1}^{C} \exp(\mathbf{Z}^{(4)}_{:,j})\right) && \text{Log-Softmax}
\end{aligned}$$

5. **Loss**: $\mathcal{L}(\mathbf{X}, \mathbf{y}) = \frac{1}{B}\sum_{n=1}^{B} -\hat{\mathbf{Y}}_{n,\mathbf{y}_n}$ (Negative Log-Likelihood)

Questions:

1. (0.75) Derive the complete backpropagation equations. Your derivation **must include**:
   (a) The gradient through Layer Normalization. Given upstream gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{o}}$ where $\mathbf{o} = \text{LN}(\mathbf{z})$, derive $\frac{\partial \mathcal{L}}{\partial \mathbf{z}}$, $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\gamma}}$, and $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\beta}}$.
   Hint: For a single sample with hidden dimension h, the gradient through LayerNorm is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \frac{\gamma}{\sigma}\left(\frac{\partial \mathcal{L}}{\partial \mathbf{o}} - \frac{1}{h}\sum_j \frac{\partial \mathcal{L}}{\partial \mathbf{o}_j} - \frac{\hat{\mathbf{z}}}{h}\sum_j \frac{\partial \mathcal{L}}{\partial \mathbf{o}_j}\hat{\mathbf{z}}_j\right)$$

   where $\hat{\mathbf{z}}$ is the normalized input. Derive this formula and extend it to batch computation.
   (b) The gradient flow through dense connections. Show how gradients from Layer 3 propagate back through the concatenation to **both** $\mathbf{H}^{(1)}$ and $\mathbf{H}^{(2)}$ simultaneously. Draw a diagram showing all gradient paths.
   (c) Gradients for all weight matrices $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{W}^{(3)}, \mathbf{W}^{(4)}$ and all LayerNorm parameters.

2. (0.75) Implement **NumpyDenseResMLP** in **numpy_mlp.py**. Your implementation must include:
   - `__init__()`: Initialize all weight matrices and LayerNorm parameters ($\boldsymbol{\gamma}$, $\boldsymbol{\beta}$). Use He initialization for weights and initialize $\boldsymbol{\gamma} = \mathbf{1}$, $\boldsymbol{\beta} = \mathbf{0}$.
   - `forward()`: Implement the forward pass, caching all intermediate values needed for backpropagation (including μ, σ, $\hat{\mathbf{z}}$ for each LayerNorm).
   - `backward()`: Implement backpropagation through all layers including LayerNorm, and update all parameters using SGD.

3. (0.25) Train with hidden size 256, batch size 128, learning rate 0.01, for 30 epochs. The command is:

```
python numpy_mlp.py --hidden-size 256 --batch-size 128 --epochs 30 --lr 0.01
```

Provide “Loss vs Epochs” and “Accuracy vs Epochs” plots (include both train and test accuracy).

Additionally, train a version of your model **without Layer Normalization** (you may need to reduce the learning rate for stability, e.g., 0.001). Compare the two models: Which converges faster? Which achieves better final accuracy? Explain your observations.

4. (0.25) Implement `main_learning_curve()` to plot learning curves. Vary training data from 5500 to 56000 in increments of 5500, i.e., `range(5500, 56001, 5500)`. Run the following three configurations:

   - Hidden size 128 with Layer Normalization (lr=0.01)
   - Hidden size 256 with Layer Normalization (lr=0.01)
   - Hidden size 256 **without** Layer Normalization (lr=0.001)

   Report requirements:

   - Your early stopping strategy
   - Learning curve plots (training set size vs. accuracy) for all three configurations
   - Analysis: How does hidden size affect the learning curves? How does Layer Normalization affect convergence and final accuracy?

Q6 (1.0 pts): G-Equivariant Autoencoder with Invariant Attention Pooling

Open the notebook **G-equivariant_Auto-Encoder/G-equivariant-AE-skeleton.ipynb**.

In this question, you will extend the G-equivariant autoencoder by incorporating **G-invariant attention pooling** to create a global embedding. This architecture combines:

- **Local G-equivariant features**: Features that transform predictably under group actions
- **Global G-invariant embedding**: A pooled representation that remains unchanged under group actions

The key insight is that **attention mechanisms can be made G-invariant** by averaging attention computations over group orbits—this is analogous to applying the Reynolds operator to the attention weights.

Background: G-Equivariant Layers (Review)

Consider the rotation group $G = \{T^{(0^\circ)}, T^{(90^\circ)}, T^{(180^\circ)}, T^{(270^\circ)}\}$ acting on vectorized images. For input $\mathbf{x} \in \mathbb{R}^m$ and output $\mathbf{y} \in \mathbb{R}^k$, G-equivariance requires:

$$\forall g \in G, \quad \rho_2(g)\mathbf{W}\mathbf{x} = \mathbf{W}\rho_1(g)\mathbf{x}$$

where $\rho_1 : G \to \mathbb{R}^{m\times m}$ and $\rho_2 : G \to \mathbb{R}^{k\times k}$ are group representations. The weight matrix $\mathbf{W}$ must satisfy:

$$\forall g \in G, \quad \rho_2(g) \otimes \rho_1(g^{-1})^T \text{vec}(\mathbf{W}) = \text{vec}(\mathbf{W})$$

New Component: G-Invariant Attention Pooling

Given G-equivariant features $\mathbf{h} \in \mathbb{R}^k$ from the encoder, we compute a **G-invariant global embedding** $\mathbf{g} \in \mathbb{R}^k$ using attention over group orbits:

$$\alpha_i = \frac{\exp(\mathbf{q}^T \rho(g_i)\mathbf{h})}{\sum_{j=1}^{|G|} \exp(\mathbf{q}^T \rho(g_j)\mathbf{h})} \quad \text{for each } g_i \in G$$

$$\mathbf{g} = \sum_{i=1}^{|G|} \alpha_i \cdot \rho(g_i)\mathbf{h}$$

where $\mathbf{q} \in \mathbb{R}^k$ is a **learnable query vector** and $\rho : G \to \mathbb{R}^{k\times k}$ is the group representation on the feature space.

The final encoder output combines local and global information: $\mathbf{z} = \text{MLP}([\mathbf{h} \| \mathbf{g}])$

Questions

1. (0.15) **Understanding the Rotation Group**

   Describe all transformations in the rotation group $G = \{T^{(0^\circ)}, T^{(90^\circ)}, T^{(180^\circ)}, T^{(270^\circ)}\}$. What is the group operation? Verify that this set forms a group (identity, closure, inverse, associativity).

2. (0.20) **Proving Invariance of Attention Pooling**

   Prove that the global embedding $\mathbf{g}$ computed via attention pooling is G-invariant. That is, show that for any $g' \in G$:

   $$\mathbf{g}(\rho(g')\mathbf{h}) = \mathbf{g}(\mathbf{h})$$

   Hint: Use the rearrangement property of groups: for any fixed $g' \in G$, the set $\{g' \circ g_i : g_i \in G\} = G$. How does this relate to the Reynolds operator from lecture?

3. (0.15) **Combined Representation Analysis**

   The encoder outputs a concatenation $[\mathbf{h}\|\mathbf{g}]$ where $\mathbf{h}$ is G-equivariant and $\mathbf{g}$ is G-invariant.

   (a) What is the transformation behavior of $[\mathbf{h}\|\mathbf{g}]$ under group action $g \in G$? Express it as a block diagonal matrix.

   (b) For the decoder to produce G-equivariant outputs end-to-end, what constraint must the first decoder layer's weight matrix satisfy when taking $[\mathbf{h}\|\mathbf{g}]$ as input?

4. (0.50) **Implementation**

   Complete the notebook implementation:

   (a) Implement `GInvariantAttentionPool`: A module that computes G-invariant global embeddings via attention over group orbits.

```
class GInvariantAttentionPool(nn.Module):
    def __init__(self, feature_dim, group_reps):
        # feature_dim: dimension k (must be perfect square)
        # group_reps: list of rho(g) matrices for g in G
        # TODO: Initialize learnable query vector self.query
        # TODO: Store group representations

    def forward(self, h):
        # h: (batch, feature_dim) - G-equivariant features
        # Returns: (batch, feature_dim) - G-invariant embedding
        # TODO: Compute rho(g_i) @ h for each g_i in G
        # TODO: Compute attention weights using self.query
        # TODO: Return weighted sum
```

   (b) Integrate `GInvariantAttentionPool` into the G-equivariant autoencoder. The encoder should output both local features **h** and global embedding **g**.

   (c) Train both architectures (standard G-equiv AE vs. attention-augmented) and compare:

   - In-distribution test loss
   - Out-of-distribution (rotated) test loss
   - Provide reconstruction visualizations for both

   (d) Visualize the learned attention weights α_i for several test images. Are the weights uniform across group elements, or does the model learn to favor certain transformations? What does this tell you about what the attention mechanism has learned?

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: Please submit through Brightspace. There is an option to upload the code of hw1.