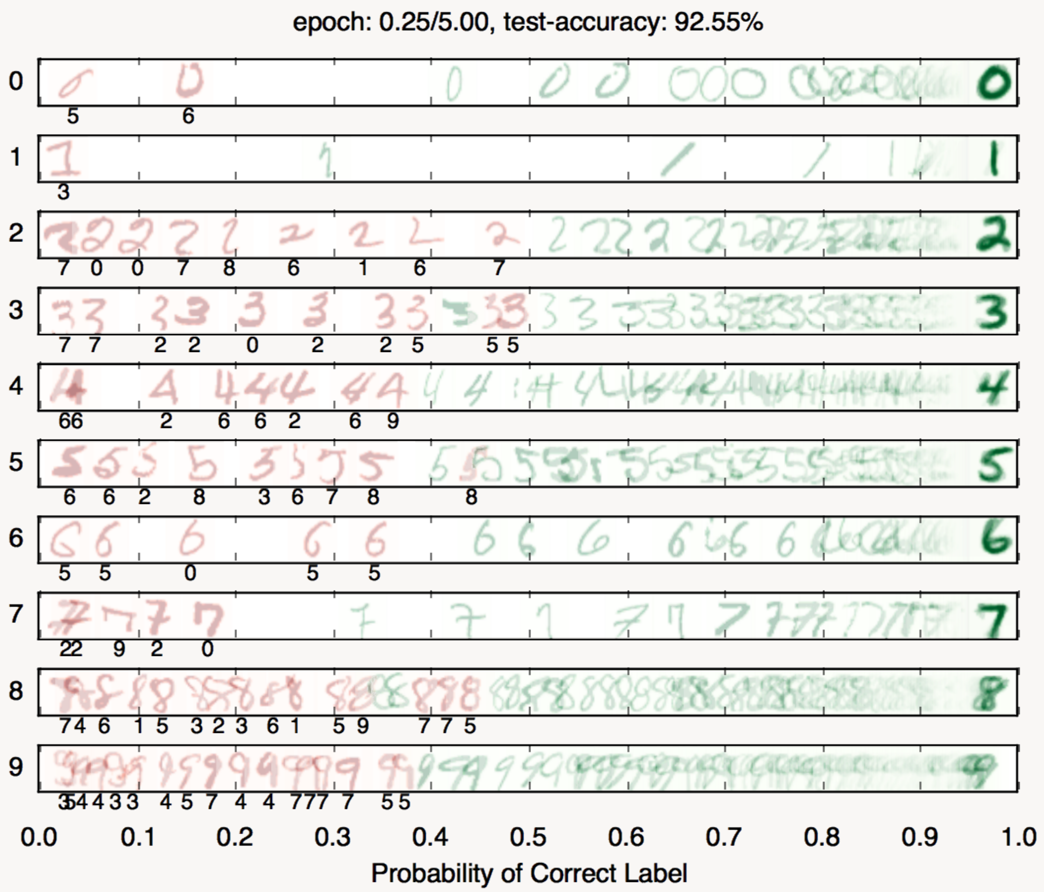
Project 4: Machine Learning

Version 1.1. Last Updated: 12/13/2022.
Due: **See Canvas**



In this project you will build a neural network to classify digits, and more!

Introduction

This project will be an introduction to machine learning.

The code for this project contains the following files, available as a zip archive.

Files you'll edit:	
`models.py`	Perceptron and neural network models for a variety of applications
Files you should read but NOT edit:	
`nn.py`	Neural network library
Files you will not edit:	
`autograder.py`	Project autograder
`backend.py`	Backend code for various machine learning tasks
`data`	Datasets for digit classification and language identification
`submission_autograder.py`	Submission autograder (generates tokens for submission)

Files to Edit and Submit: You will fill in portions of `models.py` during the assignment. Please *do not* change the other files in this distribution. You do not need to submit any other files.

Evaluation: Your code will be autograded for technical correctness. Please *do not* change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's judgements – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

Academic Dishonesty: Copying someone else's code and submitting it as your own is asking for a grade you did not earn and claiming mastery of skills of which you have not demonstrated mastery. We may or may not use a plagiarism tool on your code in this class.

Proper Dataset Use: Part of your score for this project will depend on how well the models you train perform on the test set included with the autograder. We do not provide any APIs for you to access the test set directly. Any attempts to bypass this separation or to use the testing data during training will be considered cheating.

Getting Help: You are not alone, though we do expect you to know and practice basic problem-solving skills. If you find yourself stuck on something, contact the instructor or a classmate for help. Class time, Office Hours, Discord and Teams are there for your support; please use them. If you need to, set up an appointment for help. These projects should be rewarding and instructional, not frustrating and demoralizing—but I don't know when or how to help unless you ask.

Discord and Teams: Please be careful not to post spoilers nor executable code.

Installation

For this project, you will need to install the following two libraries:

- numpy, which provides support for large multi-dimensional arrays - installation instructions
- matplotlib, a 2D plotting library - installation instructions

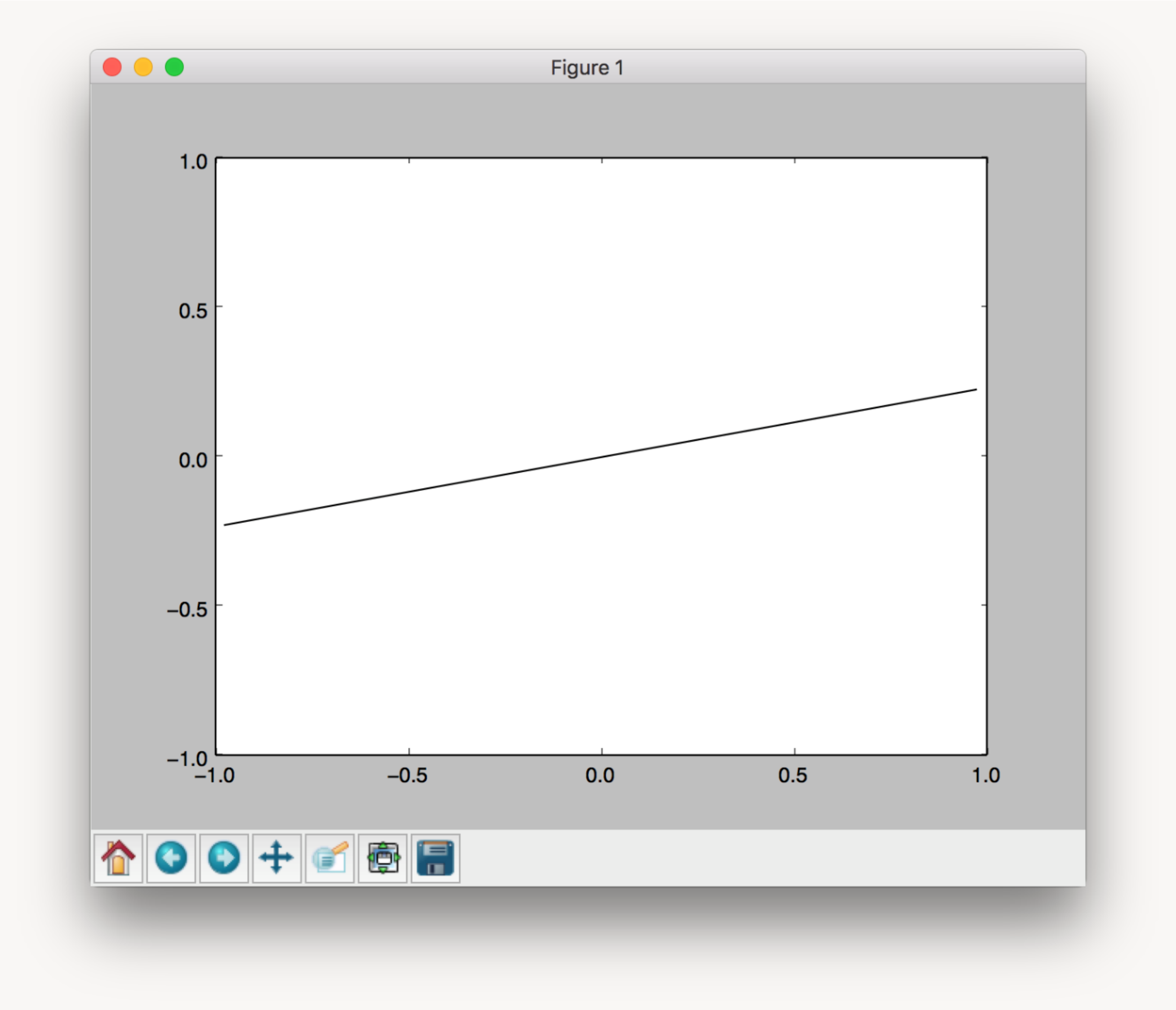
If you have a conda environment, you can install both packages on the command line by running:

```
conda activate [your environment name]

pip install numpy

pip install matplotlib
```

If you have Python installed on your computer but you are not using Anaconda, then use pip without the conda command. You will not be using these libraries directly, but they are required by the provided code and autograder. To test that everything has been installed, run:

```
python autograder.py --check-dependencies
```

If `numpy` and `matplotlib` are installed correctly, you should see a window pop up where a line segment spins in a circle:



Provided Code (Part I)

For this project, you have been provided with a neural network mini-library (`nn.py`) and a collection of datasets (`backend.py`).

The library in `nn.py` defines a collection of node objects. Each node represents a real number or a matrix of real numbers. Operations on node objects are optimized to work faster than using Python's built-in types (such as lists).

Here are a few of the provided node types:

- `nn.Constant` represents a matrix (2D array) of floating point numbers. It is typically used to represent input features or target outputs/labels. Instances of this type will be provided to you by other functions in the API; you will not need to construct them directly.
- `nn.Parameter` represents a trainable parameter of a perceptron or neural network.
- `nn.DotProduct` computes a dot product between its inputs.

Additional provided functions:

- `nn.as_scalar` can extract a Python floating-point number from a node.

When training a perceptron or neural network, you will be passed a `dataset` object. You can retrieve batches of training examples by calling `dataset.iterate_once(batch_size)`:

```
for x, y in dataset.iterate_once(batch_size):

...
```

Let's try to extract a batch of size 1 (a single training example) from the perceptron training data:

```
>>> batch_size = 1

>>> for x, y in dataset.iterate_once(batch_size):

...     print(x)

...     print(y)

...     break

...

<Constant shape=1x3 at 0x11a8856a0>

<Constant shape=1x1 at 0x11a89efd0>
```

The input features `x` and the correct label `y` are provided in the form of `nn.Constant` nodes. The shape of `x` will be `batch_size x num_features`, and the shape of `y` is `batch_size x num_outputs`. Here is an example of computing a dot product of `x` with itself, first as a node and then as a Python number.

```
>>> nn.DotProduct(x, x)

<DotProduct shape=1x1 at 0x11a89edd8>

>>> nn.as_scalar(nn.DotProduct(x, x))

1.9756581717465536
```

Question 1 (8 points): Perceptron

Before starting this part, be sure you have `numpy` and `matplotlib` installed!

In this part, you will implement a binary perceptron. Your task will be to complete the implementation of the `PerceptronModel` class in `models.py`.

For the perceptron, the output labels will be either 1 or −1, meaning that data points `(x, y)` from the dataset will have `y` be a `nn.Constant` node that contains either 1 or −1 as its entries.

We have already initialized the perceptron weights `self.w` to be a $1\times\mathrm{dimensions}$1×dimensions parameter node. The provided code will include a bias feature inside `x` when needed, so you will not need a separate parameter for the bias.

Your tasks are to:

- Implement the `run(self, x)` method. This should compute the dot product of the stored weight vector and the given input, returning an `nn.DotProduct` object.
- Implement `get_prediction(self, x)`, which should return 1 if the dot product is non-negative or −1 otherwise. You should use `nn.as_scalar` to convert a scalar `Node` into a Python floating-point number.

- Write the train(self) method. This should repeatedly loop over the data set and make updates on examples that are misclassified. Use the update method of the nn.Parameter class to update the weights. When an entire pass over the data set is completed without making any mistakes, 100% training accuracy has been achieved, and training can terminate.

In this project, the only way to change the value of a parameter is by calling parameter.update(direction, multiplier), which will perform the update to the weights $weights \leftarrow weights + direction \cdot multiplier$. The direction argument is a Node with the same shape as the parameter, and the multiplier argument is a Python scalar.

To test your implementation, run the autograder:

```
python autograder.py -q q1
```

Big Note: the autograder should take at most 20 seconds or so to run for a correct implementation. If the autograder is taking forever to run, your code probably has a bug.

Neural Network Tips

In the remaining parts of the project, you will implement the following models:

- Q2: Regression
- Q3: Handwritten Digit Classification

Building Neural Nets

Throughout the applications portion of the project, you'll use the framework provided in nn.py to create neural networks to solve a variety of machine learning problems. A simple neural network has layers, where each layer performs a linear operation just like a perceptron. Layers are *separated by a non-linearity*, which allows the network to approximate general functions. We'll use rectified linear units (ReLU) for our non-linearity, defined as $relu(x) = max(x, 0)$. For example, a simple two-layer neural network for mapping an input row vector x to an output vector $f(x)$ would be given by the function: $f(x) = relu(x \cdot W_1 + b_1) \cdot W_2 + b_2$ where we have weight matrices W_1 and W_2 and bias weight vectors b_1 and b_2 to learn during gradient descent. W_1 will be an $i \times hi \times h$ matrix, where i is the dimension of our input vectors x, and h is the *hidden layer size*. b_1 will be a size h vector. We are free to choose any value we want for the hidden layer size. We will just need to make sure the dimensions of all matrices and vectors agree so that operations are well-defined and executable. More nodes in the hidden layer can make the network more powerful, but can make the network harder to train and can lead to overfitting on the training data.

We can also create deeper networks by adding more hidden layers. An example three-layer net computes this function: $f(x) = relu(relu(x \cdot W_1 + b_1) \cdot W_2 + b_2) \cdot W_3 + b_3$

On Batching

For efficiency, you will be required to process minibatches of data at once rather than a single example at a time. This means that instead of a single input row vector x with size i, you will be presented with a batch of b inputs represented as a $b \times ib \times i$ matrix X. We provide an example for linear regression to demonstrate how a linear layer can be implemented in the batched setting.

On Randomness

The parameters of your neural network will be randomly initialized, and data in some tasks will be presented in shuffled order. Your network may fail to learn some tasks, but this should be rare if you have a good architecture. If your code fails the autograder twice in a row for a question, you should modify your architecture.

Practical tips

Designing neural nets can take some trial and error. Here are some tips to help:

- Be systematic.
  Keep a log of every architecture you’ve tried, what the hyperparameters, such as layer sizes, learning rate, were, and what the resulting performance was. As you try more things, you can start seeing patterns about which parameters matter. If you find a bug in your code, be sure to cross out past results that are invalid due to the bug.

- Start with a shallow—single hidden layer--network.
  Deeper networks combinatorically more weights and can be more difficult to debug. Use the small network to find a good learning rate and layer size, then consider adding more layers of similar size.
- If your learning rate is wrong, none of your other hyperparameter choices matter.
  You can take a state-of-the-art model from a research paper and change the learning rate such that it performs no better than random. A learning rate too low will result in the model learning too slowly, and a learning rate too high may cause loss to diverge to infinity. Begin by trying different learning rates while looking at how the loss decreases over time.
- Smaller batches require lower learning rates.
  When experimenting with different batch sizes, be aware that the best learning rate may be different depending on the batch size.
- Refrain from making the hidden layer too wide.
  If you keep adding more nodes to the hidden layer, accuracy will eventually decline, and computation time will increase quadratically in the layer size. The full autograder for all parts of the project takes 2-12 minutes to run with staff solutions; if your code is taking much longer you should check it for efficiency.
- If your model is returning Infinity or NaN, your learning rate is probably too high for your current architecture.
- Recommended values for your hyperparameters:
  - Hidden layer sizes: between 10 and 400.
  - Batch size: between 1 and the size of the dataset. For Q2 and Q3, we require that total size of the dataset be evenly divisible by the batch size.
  - Learning rate: between 0.001 and 1.0.
  - Number of hidden layers: between 1 and 3.

Provided Code (Part II)

Here is a full list of nodes available in `nn.py`. You will make use of these in the remaining parts of the assignment:

- `nn.Constant` represents a matrix (2D array) of floating point numbers. It is typically used to represent input features or target outputs/labels. Instances of this type will be provided to you by other functions in the API; you will not need to construct them directly.

- `nn.Parameter` represents a trainable parameter of a perceptron or neural network. All parameters must be 2-dimensional.
  - Usage: `nn.Parameter(n, m)` constructs a parameter with shape n×mn×m
- `nn.Add` adds matrices element-wise.
  - Usage: `nn.Add(x, y)` accepts two nodes of shape $batch\_size \times num\_features$ and constructs a node that also has shape $batch\_size \times num\_features$.
- `nn.AddBias` adds a bias vector to each feature vector.
  - Usage: `nn.AddBias(features, bias)` accepts `features` of shape $batch\_size \times num\_features$ and `bias` of shape $1 \times num\_features$, and constructs a node that has shape $batch\_size \times num\_features$.
- `nn.Linear` applies a linear transformation (matrix multiplication) to the input.
  - Usage: `nn.Linear(features, weights)` accepts `features` of shape $batch\_size \times num\_input\_features$ and `weights` of shape $num\_input\_features \times num\_output\_features$, and constructs a node that has shape $batch\_size \times num\_output\_features$.
- `nn.ReLU` applies the element-wise Rectified Linear Unit nonlinearity $relu(x) = max(x, 0)$. This nonlinearity clips all negative values to zero.
  - Usage: `nn.ReLU(features)`, which returns a node with the same shape as the `input`.
- `nn.SquareLoss` computes a batched square loss, used for regression problems
  - Usage: `nn.SquareLoss(a, b)`, where `a` and `b` both have shape $batch\_size \times num\_outputs$.
- `nn.SoftmaxLoss` computes a batched softmax loss, used for classification problems.
  - Usage: `nn.SoftmaxLoss(logits, labels)`, where `logits` and `labels` both have shape $batch\_size \times num\_classes$. The term “logits” refers to scores produced by a model, where each entry can be an arbitrary real number. It is short for logistic function. The labels, however, must be non-negative and have each row sum to 1. Be sure not to swap the order of the arguments!
- *Do not use `nn.DotProduct` for any model other than the perceptron.*

The following methods are available in `nn.py`:

- `nn.gradients` computes gradients of a loss with respect to provided parameters.
    - Usage: `nn.gradients(loss, [parameter_1, parameter_2, ..., parameter_n])` will return a list `[gradient_1, gradient_2, ..., gradient_n]`, where each element is an `nn.Constant` containing the gradient of the loss with respect to a parameter.
- `nn.as_scalar` can extract a Python floating-point number from a loss node. This can be useful to determine when to stop training.
    - Usage: `nn.as_scalar(node)`, where `node` is either a loss node or has shape `(1,1)`.

The datasets provided also have two additional methods:

- `dataset.iterate_forever(batch_size)` yields an infinite sequences of batches of examples. They are not guaranteed to be unique.
- `dataset.get_validation_accuracy()` returns the accuracy of your model on the validation set. This can be useful to determine when to stop training.

Example: Linear Regression

As an example of how the neural network framework works, let's fit a line to a set of data points. We'll start four points of training data constructed using the function $y = 7x_0 + 8x_1 + 3$. In batched form, our data is:

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad Y = \begin{bmatrix} 3 \\ 11 \\ 10 \\ 18 \end{bmatrix}$$

Suppose the data is provided to us in the form of `nn.Constant` nodes:

```
>>> x

<Constant shape=4x2 at 0x10a30fe80>

>>> y
```

```
<Constant shape=4x1 at 0x10a30fef0>
```

Let’s construct and train a model of the form $f(x) = x_0 \cdot m_0 + x_1 \cdot m_1 + b$. If done correctly, we should be able to learn than $m_0 = 7,\ m_1 = 8, and\ b = 3$.

First, we create our trainable parameters. In matrix form, these are:

$$M = \begin{bmatrix} m_0 \\ m_1 \end{bmatrix} \qquad B = [b]$$

Which corresponds to the following code:

```
m = nn.Parameter(2, 1)

b = nn.Parameter(1, 1)
```

Printing them gives:

```
>>> m

<Parameter shape=2x1 at 0x112b8b208>

>>> b

<Parameter shape=1x1 at 0x112b8beb8>
```

Next, we compute our model’s predictions for y:

```
xm = nn.Linear(x, m)

predicted_y = nn.AddBias(xm, b)
```

Our goal is to have the predicted y-values match the provided data. In linear regression we do this by minimizing the square loss: $\mathcal{L} = \frac{1}{2N}\sum_{(x,y)}(y - f(x))^2$

We construct a loss node:

```
loss = nn.SquareLoss(predicted_y, y)
```

In our framework, we provide a method that will return the gradients of the loss with respect to the parameters:

```
grad_wrt_m, grad_wrt_b = nn.gradients(loss, [m, b])
```

Printing the nodes used gives:

```
>>> xm

<Linear shape=4x1 at 0x11a869588>

>>> predicted_y

<AddBias shape=4x1 at 0x11c23aa90>

>>> loss

<SquareLoss shape=() at 0x11c23a240>

>>> grad_wrt_m

<Constant shape=2x1 at 0x11a8cb160>

>>> grad_wrt_b

<Constant shape=1x1 at 0x11a8cb588>
```

Copy

We can then use the `update` method to update our parameters. Here is an update for `m`, assuming we have already initialized a `multiplier` variable based on a suitable learning rate of our choosing:

```
m.update(grad_wrt_m, multiplier)
```

If we also include an update for `b` and add a loop to repeatedly perform gradient updates, we will have the full training procedure for linear regression.

Question 2 (8 points): Non-linear Regression

For this question, you will train a neural network to approximate $sin(x)$ over $[-2\pi, 2\pi]$.

You will need to complete the implementation of the `RegressionModel` class in `models.py`. For this problem, a relatively simple architecture should suffice (see Neural Network Tips for architecture tips.) Use `nn.SquareLoss` as your loss.

Your tasks are to:

- Implement `RegressionModel.__init__` with any needed initialization
- Implement `RegressionModel.run` to return a $batch\_size \times 1$ node that represents your model's prediction.
- Implement `RegressionModel.get_loss` to return a loss for given inputs and target outputs.
- Implement `RegressionModel.train`, which should train your model using gradient-based updates.

There is only a single dataset split for this task--there is only training data and no validation data or test set. Your implementation will receive full points if it gets a loss of 0.02 or better, averaged across all examples in the dataset. You may use the training loss to determine when to stop training: use `nn.as_scalar` to convert a loss node to a Python number. It should take the model a few minutes to train, so be patient.

To test your implementation:

```
python autograder.py -q q2
```

Question 3 (9 points): Digit Classification

For this question, you will train a network to classify handwritten digits from the MNIST dataset.

Each digit is of size 28×28 pixels, the values of which are stored in a 784-dimensional vector of floating-point numbers. Each output we provide is a 10-dimensional vector which has zeros in all positions, except for a one in the position corresponding to the correct class of the digit.

Complete the implementation of the `DigitClassificationModel` class in `models.py`. The return value from `DigitClassificationModel.run()` should be a $batch\_size \times 10$ node containing scores, where higher scores indicate a higher probability of a digit belonging to a particular class (0-9). You should use `nn.SoftmaxLoss` as your loss. Do not put a ReLU activation after the last layer of the network.

This question and Q4 use training data, a validation set and a test set. You can use `dataset.get_validation_accuracy()` to compute validation accuracy for your model, which can be useful when deciding whether to stop training. The test set will be used by the autograder.

To receive points for this question, your model should achieve an accuracy of at least 97% on the test set. For reference, our staff implementation consistently achieves an accuracy of 98% on the validation data after training for about 5 epochs. The test grades you on **test accuracy**, while you only have access to **validation accuracy** - so if your validation accuracy meets the 97% threshold, you may still fail the test if your test accuracy does not meet the threshold. It may help to set a slightly higher stopping threshold on validation accuracy, such as 97.5% or 98%. If validation accuracy is high but test accuracy is low, it means your model is overfitting the training data.

To test your implementation, run the autograder:

```
python autograder.py -q q3
```

Submission

To submit your solution, run `autograder.py` on your code, then zip the files indicated above as p5.zip. Submit the zip file under **Project 4** on Canvas.

The full project autograder takes 2-12 minutes to run for the staff reference solutions to the project. If your code takes significantly longer, consider checking your implementations for efficiency. Also

Please specify any partner you may have worked with and verify that both you and your partner are associated with the submission after submitting. As with other projects, if you work with a partner, you must be able to orally explain every line of code you submit, whether your partner wrote it, or you did. ***Each of you must make a separate submission in Gradescope.***

Congratulations! This was the last project for CS 4470!