Homework #3

ECE 461/661: Introduction to Machine Learning for Engineers
Prof. Carlee Joe-Wong and Prof. Gauri Joshi

Due: Friday, March 28, 2025
8:59 pm PT/11:59 pm ET/Saturday, March 29, 2025 at 6:59 am CAT

Please remember to show your work for all problems and to write down the names of any students that you collaborate with. Although you are encouraged to work together, **you must write your own solutions, which should reflect your understanding of the problem.** When answering coding questions, you may use packages such as `numpy` for incidental operations like matrix multiplication. However, **you may not use built-in packages that directly implement the specified functions for you**, unless explicitly specified in the question. The full collaboration and grading policies are available on the course website: https://www.andrew.cmu.edu/course/18-661/. You are strongly encouraged (but not required) to use Latex to typeset your solutions.

Your solutions should be uploaded to Gradescope (https://www.gradescope.com/) in PDF format by the deadline. We will not accept hardcopies. **If you choose to hand-write your solutions, please make sure the uploaded copies are legible.** Gradescope will ask you to identify which page(s) contain your solutions to which problems, so make sure you leave enough time to finish this before the deadline. We will give you a 30-minute grace period to upload your solutions in case of technical problems.

1 Dimensionality of k-Nearest Neighbors *[10 points]*

When the number of features d is large, the performance of k-nearest neighbors, which makes predictions using only observations that are near the test observation, tends to degrade. This phenomenon is known as the *curse of dimensionality*, and it ties into the fact that non-parametric approaches often perform poorly when d is large.

a. *[2 points]* Suppose that we have a set of training observations, each corresponding to a one-dimensional ($d = 1$) feature, X. We assume that X is uniformly (evenly) distributed on $[0, 1]$. Associated with each training observation is a response value.

   Suppose that we wish to predict a test observation x's response using only training observations that are within 10% of the range of x closest to that test observation. In other words, if $x \in [0.05, 0.95]$ then we will use training observations in the range $[x - 0.05, x + 0.05]$, as shown in Figure 1 when $x = 0.6$. When $x \in [0, 0.05)$ we use the range $[0, 0.1]$, and when $x \in (0.95, 1]$ we use training observations in the range $[0.9, 1]$. Figure 1 shows this range for $x = 0.02$.

   On average (assuming x is uniformly distributed on $[0, 1]$), what fraction of the available observations will we use to make the prediction?

b. *[2 points]* Now suppose that we have a set of observations, each corresponding to two features, X_1 and X_2 (i.e., $d = 2$). We assume that (X_1, X_2) are uniformly distributed on $[0, 1] \times [0, 1]$. We wish to predict the response of a test observation (x_1, x_2) using only training observations that are within 10% of the range of x_1 and within 10% of the range of x_2 closest to that test observation. For instance, in order to predict the response for a test observation with $x_1 = 0.6$ and $x_2 = 0.04$, we will use training observations (X_1, X_2) such that $X_1 \in [0.55, 0.65]$ *and* $X_2 \in [0, 0.1]$.

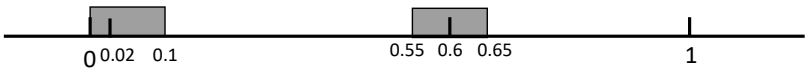


Figure 1: Range of observation, $d = 1$

On average, assuming x_1 and x_2 are each uniformly distributed on $[0, 1]$, what fraction of the available observations will we use to make the prediction?

c. *[2 points]* Now suppose that we have a set of training observations on $d = 100$ features. Again the observations are uniformly distributed on each feature, and again each feature ranges in value from 0 to 1. We wish to predict a test observation's response using observations within the 10% of each feature's range that is closest to that test observation. What fraction of the available observations will we use to make the prediction?

d. *[2 points]* Using your answers to parts a–c, argue that a drawback of k-nearest neighbors when d is large is that there are very few training observations "near" any given test observation.

e. *[2 points]* Now suppose that we wish to make a prediction for a test observation by creating a d-dimensional hypercube centered around the test observation that contains, on average, 10% of the training observations. For $d = 1, 2$, and 100, what is the length of each side of the hypercube? How does your answer change as d increases, and what does this imply for the accuracy of k-nearest neighbors when d is large?

Note: A hypercube is a generalization of a cube to an arbitrary number of dimensions. When $d = 1$, a hypercube is simply a line segment, when $d = 2$ it is a square, and when $d = 100$ it is a 100-dimensional cube.

a. Since X is uniformly distributed on $[0, 1]$,the probability that a given observation will be used to make the prediction is 10%. Thus, when d = 1, on average 10% of the available observations will be used to make the prediction.

b. When $d = 2$, the probability that a given observation will be used to make the prediction is $0.1 \times 0.1 = 0.01$. Thus, on average, 1% of the available observations will be used to make the prediction.

c. For the same reason, When d = 100, on average, a fraction of 0.1^d of the available observations will be used to make the prediction.

d. When d is large, 0.1^d will be very small. So there are very few training observations near any given test observation.

e. For d = 1, the length of each side of the hypercube is 0.1^1. For d = 2, the length of each side of the hypercube is $0.1^{\frac{1}{2}}$. For d = 100, the length of each side of the hypercube is $0.1^{\frac{1}{100}}$. Generally, for d dimensional cases, the length of each side of the hypercube is $0.1^{\frac{1}{d}}$. When d is large, the length of each side will be near to 1. This means that it is very rare that a random sample is close in every dimension to a test point, in consequence, a non-parametric approach like K-nearest neighbors often performs poorly when d is large.

2 Decision Trees *[10 points]*

You obtained the following data from interviewing 15 people on the street. Based on a person's relationship status, age, education level and income, you can now build a decision tree to predict a person's phone usage.

Relationship Status	Age	Education	Income	Phone Usage
Single	>25	University	≤50K	Low
In a relationship	≤25	College	≤50K	Medium
Single	>25	University	>50K	Low
Married	≤25	University	≤50K	High
Single	>25	University	>50K	Low
Married	>25	College	≤50K	Medium
In a relationship	≤25	College	>50K	Medium
In a relationship	>25	High School	≤50K	Low
Married	>25	University	≤50K	High
Single	>25	High School	>50K	Low
In a relationship	≤25	College	>50K	Medium
In a relationship	>25	High School	≤50K	Low
Married	>25	University	≤50K	High
Single	≤25	High School	>50K	Low
In a relationship	≤25	College	>50K	Medium

a. *[2 points]* What is the entropy of Phone Usage? (Calculate entropy in $\log_2$ base and round to 4 decimal places)

$P(\text{low}) = \frac{7}{15}, P(\text{medium}) = \frac{1}{3}, P(\text{high}) = \frac{1}{5},$

The initial entropy of usage (H(Usage)) is $-\frac{7}{15}\log_2\frac{7}{15} - \frac{1}{3}\log_2\frac{1}{3} - \frac{1}{5}\log_2\frac{1}{5} \approx 1.5058$

b. *[4 points]* Find the information gain (IG) from each feature (relationship status, age, education levarianceel and income). Which feature should be chosen at the root of the tree? Show your calculations for the information gain (IG) and explain your choice in a sentence. (Calculate entropy in $\log_2$ base and round to 4 decimal places)

$\text{H(Usage|Relationship Status)} = -\frac{2}{5}\left(\frac{1}{3}\log\frac{1}{3} + \frac{2}{3}\log\frac{2}{3}\right) - \frac{4}{15}\left(\frac{1}{4}\log\frac{1}{4} + \frac{3}{4}\log\frac{3}{4}\right) \approx 0.5837$
$\therefore \text{I(Usage, RelationshipStatus)} \approx 0.9221$

$\text{H(Usage|Age)} = -\frac{9}{15}\left(\frac{6}{9}\log\frac{6}{9} + \frac{2}{9}\log\frac{2}{9} + \frac{1}{9}\log\frac{1}{9}\right) - \frac{6}{15}\left(\frac{1}{6}\log\frac{1}{6} + \frac{4}{6}\log\frac{4}{6} + \frac{1}{6}\log\frac{1}{6}\right) \approx 1.2352$
$\therefore \text{I(Usage, Age)} = 0.2705$

$\text{H(Usage|Education)} = -\frac{2}{5}\left(\frac{1}{2}\log\frac{1}{2} + \frac{1}{2}\log\frac{1}{2}\right) \approx 0.4$
$\therefore \text{I(Usage, Education)} \approx 1.1058$

$\text{H(Usage|Income)} = -\frac{7}{15}\left(\frac{4}{7}\log\frac{4}{7} + \frac{3}{7}\log\frac{3}{7}\right) - \frac{8}{15}\left(\frac{2}{8}\log\frac{2}{8} + \frac{3}{8}\log\frac{3}{8} + \frac{3}{8}\log\frac{3}{8}\right) \approx 1.2924$
$\therefore \text{I(Usage, Income)} = 0.2133$

We pick attribute Education at root as it has the highest information gain.

c. *[4 points]* Use the root you found in part (b) to determine the rest of the nodes in the decision tree for the abovariancee data. Draw the full decision tree, i.e., keep splitting the nodes until further splits do not lead to any information gain (you may not need all the features for this). For each split, show your working on why you chose this feature based on information gain.

1) Split root with Education. College branch ends up with leaf labeled Medium Usage and High School branch ends up with leaf labeled Low Usage.

2) At Univarianceersity branch, split on Relationship Status. Single branch ends up with leaf labeled Low Usage and Married branch ends up with leaf labeled High Usage.

3 Random Forest *[10 points]*

Consider the data samples in Table 1, which record whether a person gets sick, together with features about their age (A), social distancing (S), and hand-washing frequency (H). Our goal is to learn a random forest to predict whether the person is prone to getting sick.

Sample	Age (A)	Social distancing (S)	Hand washing (H)	Getting Sick
1	Old	Yes	Yes	No
2	Old	Yes	No	No
3	Old	Yes	Yes	No
4	Young	No	No	Yes
5	Young	Yes	Yes	No
6	Old	No	No	Yes
7	Old	No	Yes	Yes
8	Old	No	No	Yes

Table 1

Instead of using all features to build a single decision tree, we decide to use the random forest algorithm. Specifically, we will build three trees, each using only two features, and then combine their outcomes for the final prediction.

a. *[6 points]* Build 3 decision trees using two features out of the three for each tree. Use information gain to decide the feature to split on. Use majority voting if all the samples in a leaf node do not have the same label.

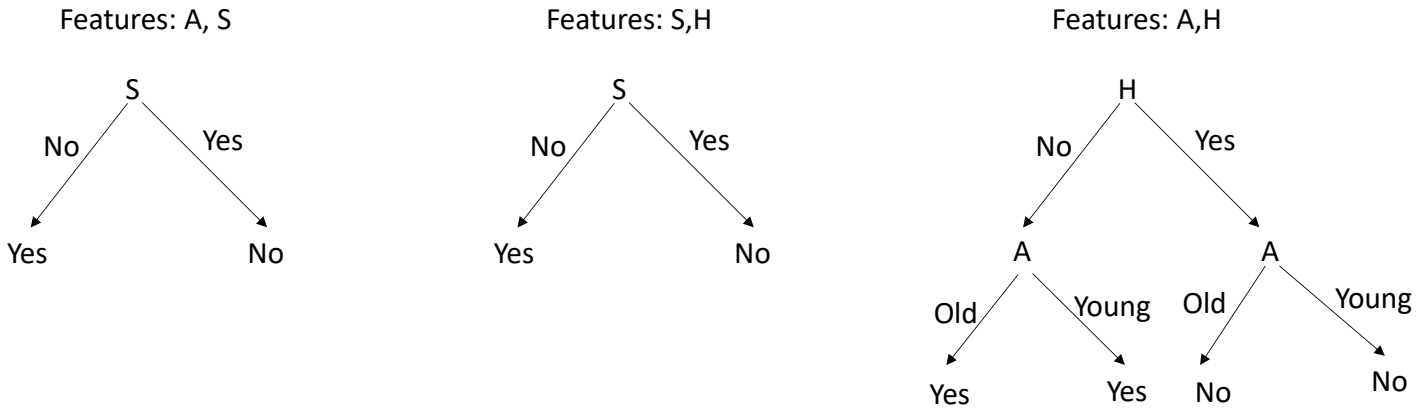


Figure 2: Ans for Q 2a.

b. *[2 points]* Given a new data point with the features (A = old, S = yes, H = no), use the trees you learned from part (a) to predict whether this person will get sick.

c. *[2 points]* Briefly (in 1-2 sentences) comment on the advantage of random forests over decision trees from the perspective of bias-variance trade-off.

First 2 trees predict No, third tree predicts Yes, majority vote is No.

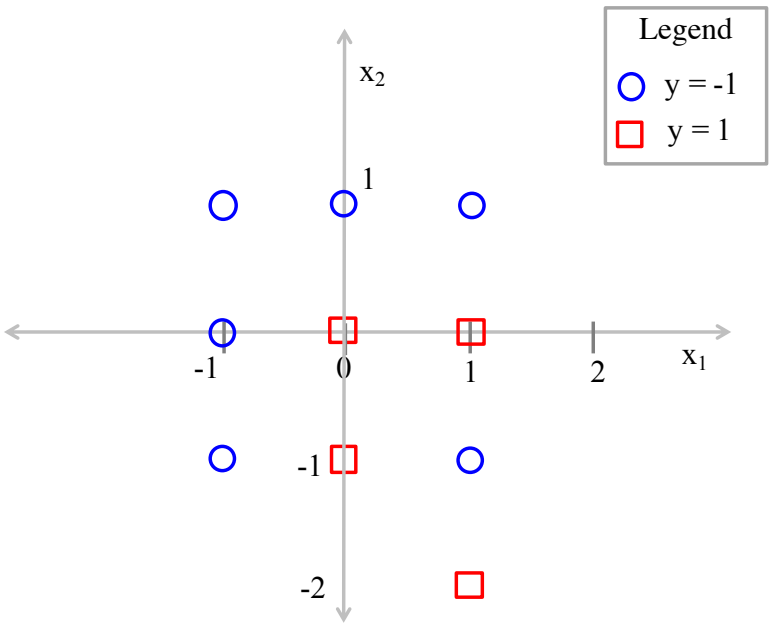
With a small choice of m, random forests reduce variance. The bias can increase; however this not always true.

4 Adaboost *[15 points]*

Recall the AdaBoost algorithm for classification of a training dataset $(\mathbf{x}_n, y_n)$ for $n = 1, \ldots N$ and $y \in \{-1, 1\}$. AdaBoost sequentially combines T weak classifiers $h_1(\mathbf{x})$, $h_2(\mathbf{x})$, $\ldots h_T(\mathbf{x})$ to build one strong classifier $\text{sign}(f_T(\mathbf{x}))$. The steps of the algorithm are as follows:

- Initialize the weights $w_1(n) = 1/N$, for all points $n = 1, 2, \ldots N$
- For $t = 1, \ldots T$:
  a. Learn classifier $h_t(\mathbf{x})$ that minimizes the error $\epsilon_t = \sum_{n=1}^{N} w_t(n) \mathbb{1}(y_n \neq h_t(\mathbf{x}_n))$
  b. Update the contribution $\beta_t = \frac{1}{2} \log_2 \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$.
  c. Update weights of training points: $w_{t+1}(n) \propto w_t(n) 2^{-\beta_t y_n h_t(\mathbf{x}_n)}$, where the weights are normalized to ensure that $\sum_{n=1}^{N} w_{t+1}(n)$ is equal to 1.
- Return the final classifier $\text{sign}(f_T(\mathbf{x}))$, where $f_T(\mathbf{x}) = \sum_{t=1}^{T} \beta_t h_t(\mathbf{x})$ for a given $\mathbf{x}$.

Note the difference from the algorithm described during the lecture: we have replaced $\log_e$ by $\log_2$ and $e^{-\beta_t}$ by $2^{-\beta_t}$ everywhere.



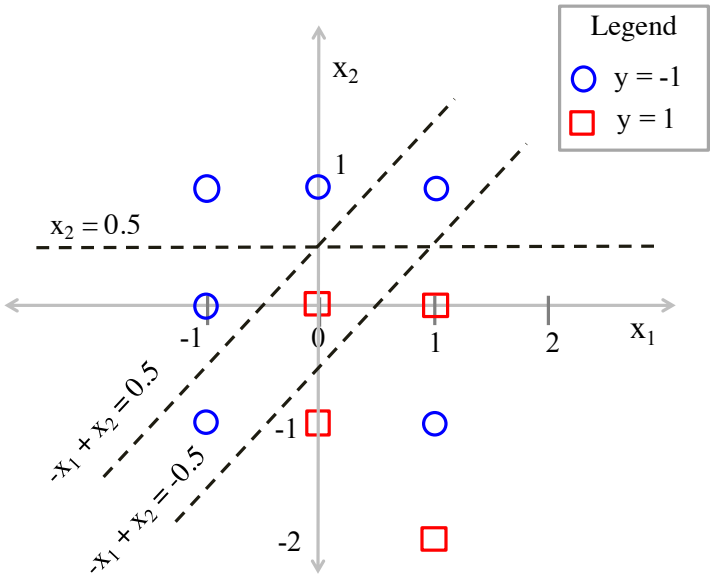
Consider the training dataset with $N = 10$ points and two dimensional features $\mathbf{x} = (x_1, x_2)$ as shown in the figure above. In this problem we will use a binary linear decision boundary as the base classifier and perform two iterations of AdaBoost.

a. *[1 points]* Starting with equal weights $w_1(n) = 1/10$ for all $N = 10$ points, which of the decision boundaries shown below gives lowest error ϵ_1? Select one of the following answers:

(a) Predict $y = 1$ if $x_2 \leq 0.5$

(b) Predict $y = 1$ if $-x_1 + x_2 \leq -0.5$

(c) Predict $y = 1$ if $-x_1 + x_2 \leq 0.5$



$-x_1 + x_2 = -0.5$ since it misclassifies 2 points while the others misclassify 3 points

b. *[3 points]* Compute the error ϵ_1 and the contribution β_1 of the decision boundary that you chose in part (a) above.

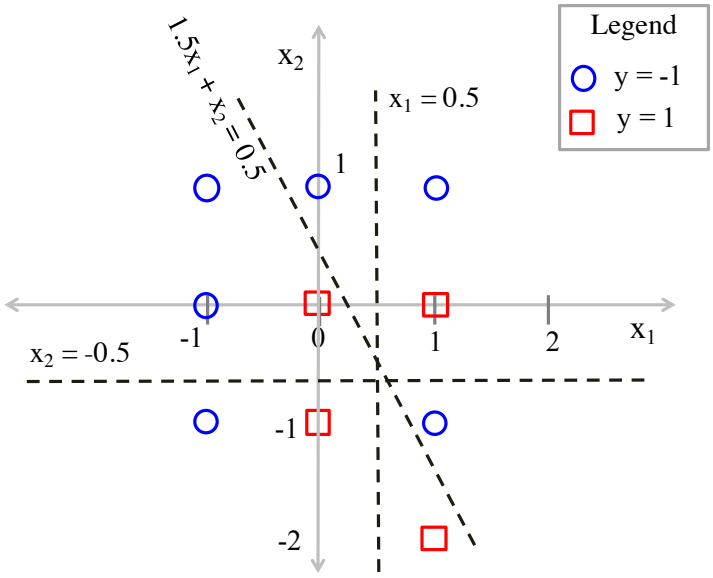
The line $-x_1 + x_2 = -0.5$ gives an error of $\epsilon_1 = 2/10$. The contribution $\beta_1 = \frac{1}{2}\log_2(0.8/0.2) = 1$.

c. *[3 points]* Compute the updated and normalized weights $w_2(n)$ of each of the data points as follows.

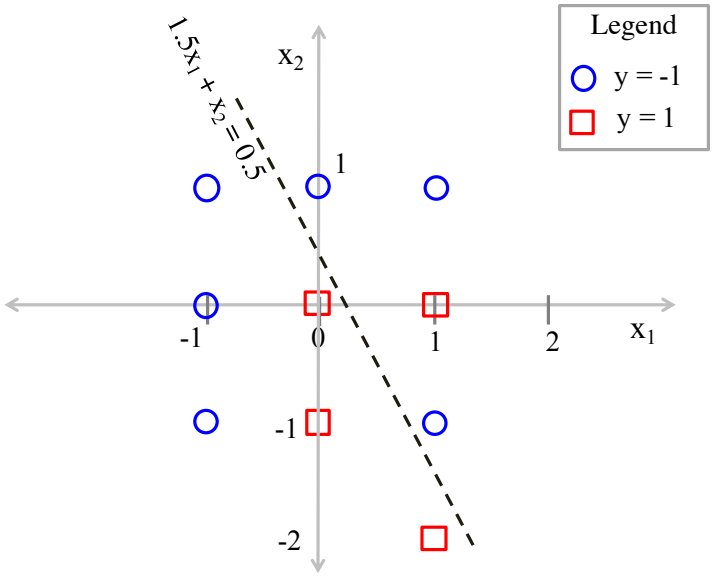
n	Coordinates	$w_2(n)$
1.	$(-1, 1)$	
2.	$(0, 1)$	
3.	$(1, 1)$	
4.	$(-1, 0)$	
5.	$(0, 0)$	
6.	$(1, 0)$	
7.	$(-1, -1)$	
8.	$(0, -1)$	
9.	$(1, -1)$	
10.	$(1, -2)$	

Table 2

The two mis-classified points ($n = 5, 9$) have weights 1/4 each and the remaining 8 points have weight 1/16 each.



d. *[3 points]* Suppose you are given one more weak classifier that predicts $y = 1$ if $1.5x_1 + x_2 \leq 0.5$, as shown in the figure below. Compute the contribution β_2 of this classifier for the updated set of weights $w_2(n)$. Use the approximation $\log_2 3 \approx 1.6$.



The error is $\epsilon_2 = 4/16 = 0.25$ and $\beta_2 = \frac{1}{2}\log_2 0.75/0.25 = \frac{1}{2}\log_2 3 = 0.8$.

Note: After the HW was released, we realized there was a typo; the classifier should be $1.5x_1 + x_2 < 0.5$ for $(1, -1)$ to be correctly classified. We have also given points if you worked out an answer with $(1, -1)$ incorrectly classified. In this you would have $\epsilon_2 = 1/2$ and $\beta_2 = 0$. The answer to parts (e) and (f) will remain unchanged.

e. *[3 points]* Combine the new classifier with the classifier that you obtained in part (a). Specify the

label $y \in \{-1, 1\}$ predicted by this combined classifier for each data point. Do you observe any change in the prediction, compared to the prediction from only using part 9a)'s classifier?

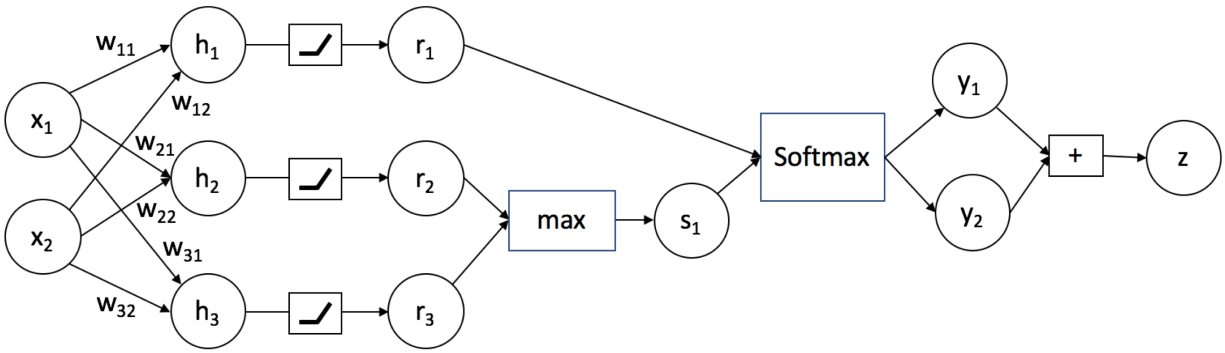
We get the same decision boundary/predictions as in part (a).

f. *[2 points]* If we continue adding more classifiers, each with error less than 0.5, how does the training error of the combined classifier $f_T(\mathbf{x})$ change? Briefly explain your answer in 1-2 sentences.

Non-increasing with T.

5 Neural Networks *[20 points]*

Below is a deep network with inputs x_1, x_2. The internal nodes and activation functions are as shown in the figure and equations below. All variables are scalar values, and $\exp(x)$ refers to the function e^x The activation functions of the nodes h_1, h_2, h_3 are ReLU(i.e. $r_1 = \max(h_1, 0)$ etc.), for node s_1 we have $s_1 = \max(r_2, r_3)$ and for the other nodes: $y_1 = \frac{\exp(r_1)}{\exp(r_1)+\exp(s_1)}$, $y_2 = \frac{\exp(s_1)}{\exp(r_1)+\exp(s_1)}$, $z = y_1 + y_2$.



a. *[6 points]* **Forward propagation.** Now, given $x_1 = 1$, $x_2 = -2$, $w_{11} = 6$, $w_{12} = 2, w_{21} = 4, w_{22} = 7, w_{31} = 5, w_{32} = 1$, compute the values of the internal nodes (shown in the table below). You may leave e in your answer.

h_1	h_2	h_3	r_1	r_2	r_3	s_1	y_1	y_2	z

h_1	h_2	h_3	r_1	r_2
2	-10	3	2	0

r_3	s	y_1	y_2	z
3	3	$\frac{1}{1+e}$	$\frac{e}{1+e}$	1

b. *[4 points]* **Bounds on variables.**

- Find the range of feasible values for y_1.
  $y_1 \in (0, 1)$. The output of a softmax is a probability distribution. Each element of the output is between 0 and 1.
- Find the range of feasible values for z.
  $z = 1$. The sum of the probability distribution is 1.

c. *[10 points]* **Backpropagation.** Compute the gradient expressions shown in the table below analytically. The answer should be an expression that may include any of the nodes in the network $(x_1, x_2, h_1, h_2, h_3, r_1, r_2, r_3, s_1, y_1, y_2, z)$ or weights $w_{11}, w_{12}, w_{21}, w_{22}, w_{31}, w_{32}$.

$\frac{\partial h_1}{\partial w_{12}}$	$\frac{\partial h_1}{\partial x_1}$	$\frac{\partial r_1}{\partial h_1}$	$\frac{\partial y_1}{\partial r_1}$	$\frac{\partial y_1}{\partial s_1}$	$\frac{\partial z}{\partial y_1}$	$\frac{\partial z}{\partial x_1}$	$\frac{\partial s_1}{\partial r_2}$

$\frac{\partial h_1}{\partial w_{12}}$	$\frac{\partial h_1}{\partial x_1}$	$\frac{\partial r_1}{\partial h_1}$	$\frac{\partial y_1}{\partial r_1}$
x_2	w_{11}	$1[h_1 > 0]$	$y_1(1 - y_1)$

$\frac{\partial y_1}{\partial s_1}$	$\frac{\partial z}{\partial y1}$	$\frac{\partial z}{\partial x_1}$	$\frac{\partial s_1}{\partial r_2}$
$-y_1 y_2$	1	0	$1[r_2 > r_3]$

Expanded solutions for selected examples below:

$r_1 = \max(h_1, 0)$. This is known as a ReLU (rectified linear unit) function. When h_1 is positive, $r_1 = h_1$, so the derivative is 1. When h_1 is negative, r_1 is flat, so the derivative is 0.

$$y_1 = \frac{\exp(r_1)}{\exp(r_1) + \exp(s_1)} = \frac{1}{1 + \exp(s_1 - r_1)}$$

$$\begin{aligned}
\frac{dy_1}{dr_1} &= \frac{-1}{(1 + \exp(r_2 - r_1))^2} \times (-\exp(r_2 - r_1)), \text{ by chain rule} \\
&= \frac{1}{1 + \exp(r_2 - r_1)} \times \frac{\exp(r_2 - r_1)}{1 + \exp(r_2 - r_1)} \\
&= y_1(1 - y_1) \\
&= y_1 y_2
\end{aligned}$$

$\frac{dy_1}{ds_1} = \frac{-1}{(1 + \exp(s_1 - r_1))^2} \times (\exp(s_1 - r_1))$, by chain rule. Notice that this is identical to the case above, but with a negative sign missing on the 2nd term.

$$\begin{aligned}
&= \frac{-1}{1 + \exp(s_1 - r_1)} \times \frac{\exp(s_1 - r_1)}{1 + \exp(s_1 - r_1)} \\
&= -y_1(1 - y_1) \\
&= -y_1 y_2
\end{aligned}$$

No matter how x_1, x_2 change, z is always 1, so the gradient with respect to x_1 is 0. When $r_2 > r_3, s_1 = r_2$, so $\frac{\partial s_1}{\partial r_2} = 1$. When $r_2 < r_3, s_1 = r_3$, so $\frac{\partial s_1}{\partial r_2} = 0$

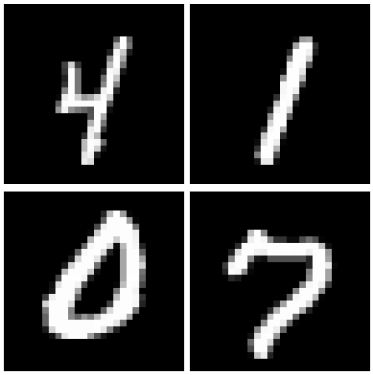


Figure 3: Sample images from the MNIST dataset.

6 Neural Networks for MNIST Digit Recognition *[35+5 points]*

The classic MNIST dataset (Figure 3) consists of 28x28 pixel (784 features) grayscale images of handwritten digits 0-9 (10 classes).

In this problem, you will implement a neural network to classify the MNIST dataset using raw pixels as features. Specifically, you will implement a type of architecture called a Multi-Layered Perceptron (MLP), which consists of several Dense layers (i.e. "Perceptrons") connected sequentially, with nonlinear activation functions after each layer. You will then train this network using the Categorical Cross-Entropy loss function.

Getting Started Download the `hw3q6.zip` file, which contains starter code for this question, as well as the MNIST train split (`mnist.npz`) and the MNIST test split with its labels removed (`mnist_test.npz`). For this lab, you will need to install Python and Numpy. While our autograder will use Python 3.6.9, you should be able to use any recent version of Python 3 and Numpy, though you should avoid the newest Python features such as dataclasses and the walrus operator. We also recommend you install tqdm (progress bar) and Pandas (tabular data), which will be available on the autograder as well.

6.1 Gradient Derivation *[8 points]*

We will begin by deriving the gradients we will need to implement backpropagation. You will derive backward passes for two types of layers: the Exponential Linear Unit (ELU) activation, and a Dense (Fully Connected) layer, then use these derivations to implement the backpropagation algorithm for your neural network.

Backpropagation Let our feed-forward (also referred to as "sequential") neural network be described by a series of functions $f_1, \dots f_n$ with input $\mathbf{x}_0$, output $\mathbf{x}_n$, loss $L = \mathcal{L}(\mathbf{x}_n, \mathbf{y}^*)$ where $\mathbf{y}^*$ is the true label, and the feed-forward relationship

$$\mathbf{x}_k = f_k(\mathbf{w}_k, \mathbf{x}_{k-1}).$$

In our model, these functions $f_1, f_2, \dots f_n$ will represent our Dense layers and activation functions, and are referred to as `modules` in our code.

In order to perform gradient descent, we need to compute the gradient for each parameter $\frac{\partial L}{\partial \mathbf{w}_k}$, which we can express recursively using the chain rule as

$$\frac{\partial L}{\partial \mathbf{w}_k} = \frac{\partial L}{\partial \mathbf{x}_k}\frac{\partial \mathbf{x}_k}{\partial \mathbf{w}_k}$$
$$\frac{\partial L}{\partial \mathbf{x}_{k-1}} = \frac{\partial L}{\partial \mathbf{x}_k}\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_{k-1}}.$$

where $\frac{\partial L}{\partial \mathbf{x}_k}$ are the gradients flowing to the previous layer, and $\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_{k-1}}$, $\frac{\partial \mathbf{x}_k}{\partial \mathbf{w}_k}$ are the Jacobians of layer f_k with respect to $\mathbf{x}_{k-1}$ and $\mathbf{w}_k$, respectively. Note that since $\frac{\partial L}{\partial \mathbf{x}_k}$ is the gradient of a scalar L with respect to a column vector $\mathbf{x}_k$, this gradient is a row vector. And $\frac{\partial \mathbf{x}_k}{\partial \mathbf{w}_k}$ is the gradient of a vector with respect to a vector, which is a matrix.

Softmax Cross Entropy This layer combines the Softmax function together with the Cross Entropy Function. The softmax function for the output layer $\mathbf{x}_n = \begin{bmatrix} x_n^{(1)} & \dots & x_n^{(D)} \end{bmatrix}$ is described by

$$x_n^{(i)} = \frac{\exp(x_{n-1}^{(i)})}{\sum_{j=1}^{D} \exp(x_{n-1}^{(j)})}$$

for input $\mathbf{x}_{n-1}$, and the Cross Entropy loss function is given by

$$\mathcal{L}(\mathbf{x_n}, \mathbf{y}^*) = -\sum_{i=1}^{D} y^{*(i)} \log \mathbf{x}_n^{(i)},$$

where $\mathbf{x}_n$ are the output weights and $\mathbf{y}^*$ are the true labels. Backward propagation through the Softmax Cross Entropy layer can be found by computing the first-order derivatives:

$$\begin{aligned}
\frac{\partial L}{\partial x_{n-1}^{(i)}} &= \frac{\partial}{\partial x_{n-1}^{(i)}} \left(-\sum_{i=1}^{D} y^{*(i)} \log \frac{\exp(x_{n-1}^{(i)})}{\sum_{j=1}^{D} \exp(x_{n-1}^{(j)})} \right) = \frac{\partial}{\partial x_{n-1}^{(i)}} \left(-\sum_{i=1}^{D} y^{*(i)} x_i + \sum_{i=1}^{D} y^{*(i)} \log \sum_{j=1}^{D} \exp(x_{n-1}^{(j)}) \right) \\
&= \frac{\partial}{\partial x_{n-1}^{(i)}} \left(-\sum_{i=1}^{D} y^{*(i)} x_{n-1}^{(i)} + \log \sum_{j=1}^{D} \exp(x_{n-1}^{(j)}) \right) \quad \text{(only one } y^{*(i)} \text{ is 1)} \\
&= -y^{*(i)} + \frac{\exp(x_{n-1}^{(i)})}{\sum_{j}^{D} \exp(x_{n-1}^{(j)})} = -y^{*(i)} + y_i.
\end{aligned}$$

The implementation of this layer is provided to you in `npnn/layers.py`.

Exponential Linear Unit The Exponential Linear Unit is a variant of the Rectified Linear Unit (ReLU) activation function that eliminates the zero gradient problem when the input is less than 0. Using ELU as an activation function tends to produce faster convergence with more accurate results than using ReLU. Unlike other activation functions, ELU has an extra α constant, which should be a positive number (a typical choice of α is 1 or 0.9).

The expression of an ELU unit is as follows:

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$$

Dense Layer A Dense layer is fully connected with the input features. For a weight matrix $\mathbf{W}$ and bias $\mathbf{b}$, the output is

$$f(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}.$$

a. *[2 points]* Find the Jacobian $\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_{k-1}}$ for the exponential linear unit, $\mathbf{x}_k = \text{ELU}(\mathbf{x}_{k-1})$.

b. *[2 points]* Find the Jacobian $\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_{k-1}}$ for the Dense layer, $\mathbf{x}_k = \mathbf{W}\mathbf{x}_{k-1} + \mathbf{b}$.

c. *[2 points]* Find the Jacobian $\frac{\partial \mathbf{x}_k}{\partial \mathbf{b}}$ for the Dense layer, $\mathbf{x}_k = \mathbf{W}\mathbf{x}_{k-1} + \mathbf{b}$.

d. *[2 points]* Since $\mathbf{W}$ is two-dimensional, the Jacobian $\frac{\partial \mathbf{x}_k}{\partial \mathbf{W}}$ can only be expressed by matrix-vector multiplications if we flatten it to a vector. Instead, to preserve its dimension, find the gradient $\frac{\partial \mathbf{x}_k[a]}{\partial \mathbf{W}[b,c]}$ for indices a, b, c (so that we can write $\frac{\partial \mathbf{x}_k}{\partial \mathbf{W}}$ as a 3-dimensional tensor). Here, a is an index of $\mathbf{x}_k$, b indexes a row of $\mathbf{W}$, and c indexes a column of $\mathbf{W}$.

a. Note that each $\mathbf{x}_k^{(i)}$ only depends on the corresponding $\mathbf{x}_{k-1}^{(i)}$ since the ELU activation is applied element-wise to each feature. Therefore,

$$\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_{k-1}} = \operatorname{diag}\left(\mathbf{1}_{\mathbf{x}_{k-1}>0} + \alpha e^{\mathbf{x}} \mathbf{1}_{\mathbf{x}\leq 0}\right).$$

b. Since the Dense layer is a linear transformation of $\mathbf{x}$, the gradient is straight-forward:

$$\frac{\partial \mathbf{W}\mathbf{x} + \mathbf{b}}{\partial \mathbf{x}_{k-1}} = \frac{\partial \mathbf{W}\mathbf{x}}{\partial \mathbf{x}_{k-1}} = \mathbf{W}.$$

c. The bias is similarly straight-forward:

$$\frac{\partial \mathbf{W}\mathbf{x} + \mathbf{b}}{\partial \mathbf{b}} = \frac{\partial \mathbf{b}}{\partial \mathbf{b}} = \mathbf{I}$$

d. Examine the indices a, b, c where a is an index of $\mathbf{x}_k$, b is a row of $\mathbf{W}$, and c is a column of $\mathbf{W}$. Then,

$$\begin{aligned}
\mathbf{x}_k[a] &= \sum_i \mathbf{W}[a,i]\mathbf{x}_{k-1}[i] \\
\frac{\partial \mathbf{x}_k[a]}{\partial \mathbf{W}[b,c]} &= \frac{\partial \sum_i \mathbf{W}[a,i]\mathbf{x}_{k-1}[i]}{\partial \mathbf{W}[b,c]} = \sum_i \frac{\partial \mathbf{W}[a,i]\mathbf{x}_{k-1}[i]}{\partial \mathbf{W}[b,c]} \\
&= \sum_i \mathbf{1}_{a=b}\mathbf{1}_{i=c}\mathbf{x}_{k-1}[i] = \mathbf{1}_{a=b}\mathbf{x}_{k-1}[c]
\end{aligned}$$

6.2 Module Implementation *[11 points]*

Implement the `ELU` and `Dense` modules based on your answers in the previous part. The correctness of your implementation for parts 6.2 and 6.3 along with your prediction accuracy for part 6.4a will be checked in unit tests, where we will compare your implementation against our implementation.

Note: Do not use any python libraries other than the ones provided in `requirements.txt`; otherwise, the autograder will not be able to run your code. In particular, you may not utilize any libraries performing automatic differentiation such as PyTorch, Tensorflow, and JAX in your submitted code, though you may use these libraries to test your implementation if you wish.

a. *[2 points]* Complete the initialization `__init__` of the dense layer in `npnn/modules.py`. You should initialize the bias b to all zeros; W should be initialized using the Glorot Uniform initialization.

   In the Glorot Uniform initialization, each element is drawn from a uniform distribution $\text{Unif}(-u, u)$, where $u = \sqrt{6/(\texttt{dim\_in} + \texttt{dim\_out})}$.

b. *[2 points]* In `npnn/modules.py`, implement `forward` passes for the `ELU` and `Dense` modules.

c. *[7 points]* In `npnn/modules.py`, implement `backward` passes for the `ELU` and `Dense` modules based on your answer in Problem 6.1.

See the released solution files.

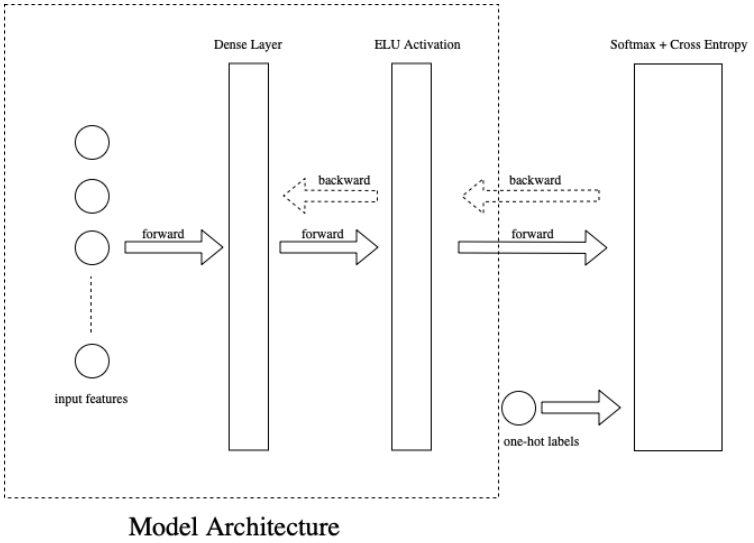


Figure 4: Neural Network Training Architecture.

6.3 Model Training Loop *[8 points]*

a. *[2 points]* In `npnn/optimizer.py`, implement the `SGD.apply_gradients` method.

b. *[2 points]* In `npnn/model.py`, implement forward and backward passes for the model in the `Sequential.forward`, `Sequential.backward` methods.

c. *[4 points]* In `npnn/model.py`, implement the training loop in `Sequential.train`, and the testing method in `Sequential.test`. To help you organize your implementation, we have provided the signature of `categorical_cross_entropy` and `categorical_accuracy` functions for you to use.

See the released solution files.

6.4 Training and Evaluation *[8 points]*

In order to train your neural network, we have loaded the MNIST dataset included with the starter code (`mnist.npz`). We have randomly split the dataset into a training dataset with 50,000 elements and a validation dataset with 10,000 elements. Note, we are splitting the MNIST dataset into train and validation and leaving the test set explicitly for testing the models predictions after completing training. Next, create a neural network with 3 Dense layers with 256, 64, and 10 units, respectively. The first two Dense layers should be followed by an ELU activation with $\alpha = 0.9$, and the last Dense layer should be followed by the SoftmaxCrossEntropy module. See figure 4 for an illustration of the neural network structure.

When correctly vectorized, your implementation should take fewer than 10 seconds to perform a single epoch during training; see the module docstring for more expected runtime details. While you will not be graded on runtime, excessive runtime may indicate the presence of errors in your code, and it may impact your ability to debug your code and complete experiments in a timely manner.

Note: at this point, you should have implemented all functions or methods annotated with `# TODO` or `raise NotImplementedError()` except for the extra credit implementation of the Adam optimizer.

a. *[2 points]* Train this network for 20 epochs with a learning rate of 0.1 and a batch size of 32, evaluating the network on your validation split after each epoch. Then, plot the train and validation accuracy after each epoch during training.

You should submit two plots along with your answer: one showing a training loss curve and a validation loss curve with respect to the training epochs, and one showing a training accuracy curve and a validation accuracy curve with respect to the training epochs. Make sure to label both curves!

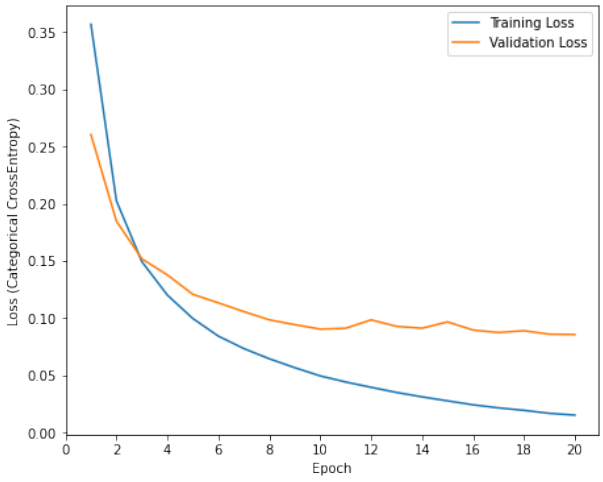
b. *[3 points]* Test your trained network on the test dataset (`mnist_test.npz`), and submit your predictions to gradescope. You will receive credit if your test accuracy is greater than 97%. **Note**: the `mnist_test.npz` dataset does not include the true labels. You can use your validation accuracy to estimate your test accuracy in advance (as long as you perform your train-val split in an unbiased manner).

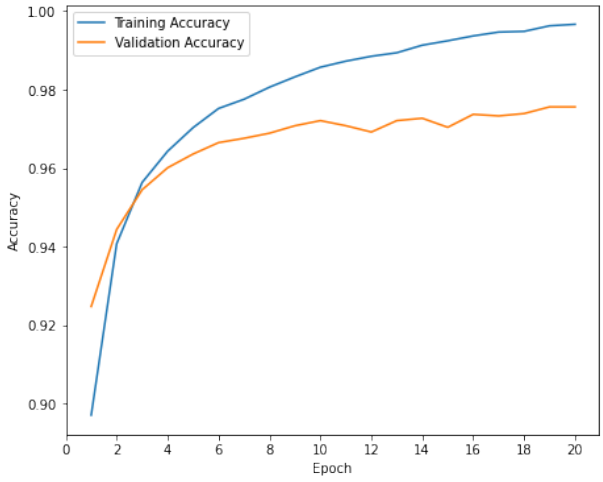
c. *[3 points]* Repeat the training procedure using learning rates of [0.05, 0.1, 0.2, 0.5, 1.0], and plot the lowest validation loss obtained for each learning rate. What is the effect of changing the learning rate?

Your plot should be a line graph with the learning rate on the x-axis and the best validation loss on the y-axis. Since there is a large difference in magnitude for the learning rate, you should use the log-learning rate in your graph, and set the labels manually:

```
fig, ax = plt.subplots()
# ... draw plot here ...
ax.set_xticks([0.05, 0.1, 0.2, 0.5, 1.0])
ax.set_xticklabels([0.05, 0.1, 0.2, 0.5, 1.0])
```

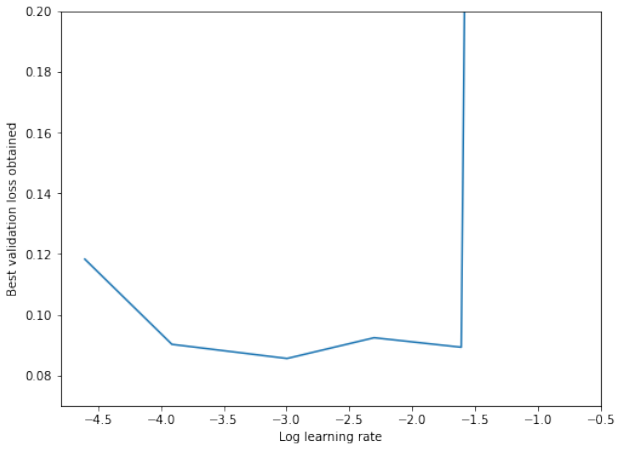
a. As we train our model to completion (loss reaches near zero), our validation loss eventually plateaus. It is worth noting that many neural networks do not tend to "overfit" harmfully, in the sense that the validation loss does not degrade as we train to completion.





b. If you have split your data into train and validation sets correctly, your validation accuracy should closely match your testing accuracy.

c. The graph below shows the best validation loss obtained against the log learning rate; in addition to the learning rates specified, we have included 0.01 and 0.02 to better show the trend. The y-axis is also cropped for clarity. The most important takeaway here is that if the SGD learning rate becomes too large, the training will diverge since each SGD step will overshoot its target.



6.5 Bonus: Adam *[+5 points]*

The Adam (Adaptive Moment) optimizer is a popular SGD variant (third most popular after SGD and Momentum by citations, according to this paper) that adds "momentum" to the gradients, and tries to normalize the gradients by their current magnitude. Adam defines two values for each parameter, m and v, which are initialized by zero, and updated by the relationship

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1^t) g_t$$
$$v_{t+1} = \beta_2 v_t + (1 - \beta_2^t) g_t^2$$

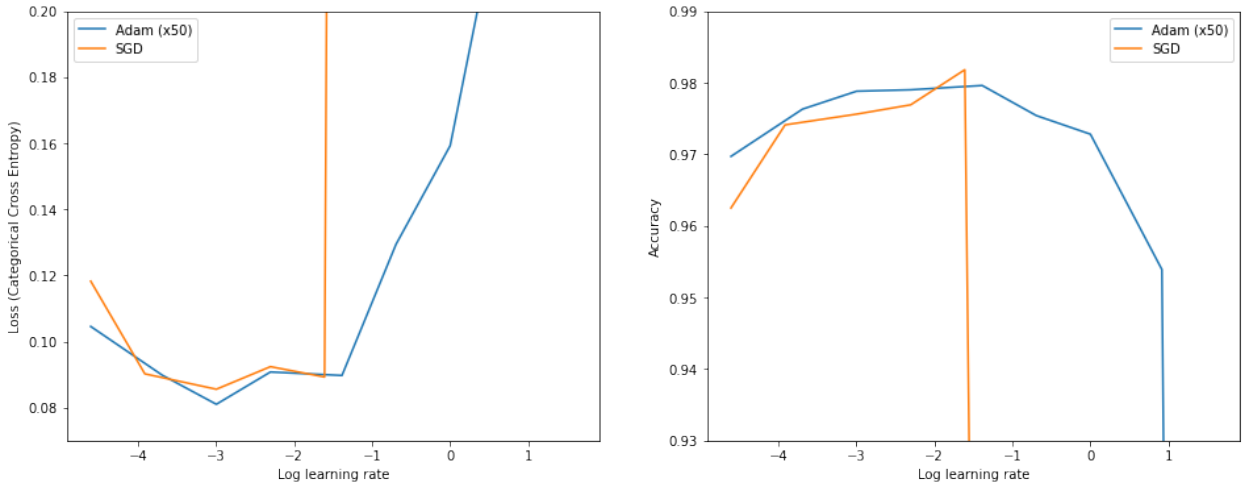
where g_t is the gradient at time step t. Then, the final parameter update is defined by

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}} + \varepsilon} \hat{m}$$

where $\hat{m} = \frac{m_t}{1-\beta_1}$ and $\hat{v} = \frac{v_t}{1-\beta_2}$. These updates are controlled by four hyperparameters: β_1, the momentum decay constant; β_2, the second moment decay constant; η, the learning rate; and ε, a small constant added to the denominator

Implement the `Adam` optimizer in `npnn/optimizer.py`, and tune the learning rate, setting the other hyperparameters to $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-7}$. How does a properly-tuned Adam optimizer compare to SGD? How does the sensitivity of Adam to its learning rate compare to SGD?

The following figure shows a comparison of Adam against SGD. Like in the previous part, we plot the best validation loss (and now accuracy) obtained against the log learning rate and crop the figure vertically. Crucially, since Adam divides by the second moment, it requires much lower learning rates, so we shift the Adam learning rate by log(50) in order to use the same x-axis as SGD.



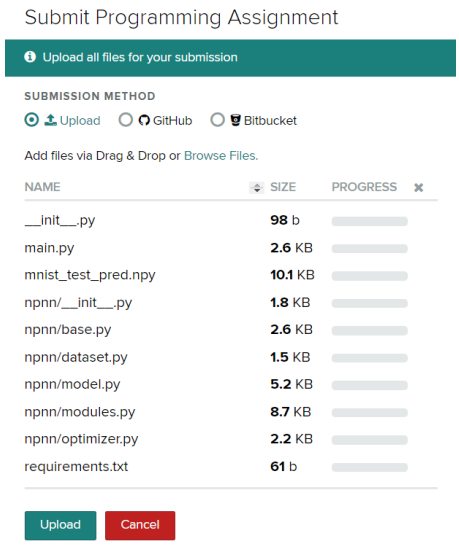
From these figures, we can see that Adam performs similarly to well-tuned SGD, but is much less sensitive to learning rate than SGD. In particular, if the learning rate is too high, SGD will diverge; however, if Adam's learning rate is too high, the erratic behavior caused by the high learning rate also causes the second moment $\hat{v}$ to increase, in turn decreasing the effective learning rate. While this interaction does harm performance (as evidenced by decreasing best accuracy/loss), it is enough to prevent complete divergence as in SGD.

Grading Notes Since this question is for bonus points, the instructions are purposefully vague, and our grading criteria more strict. Solutions receiving full credit should do the following:

- Correct implementation of Adam
- Learning rate experiments on Adam; the range should be selected so that the full U-shape (slow convergence on the left and divergence on the right) is visible.
- Conclude that Adam performs similarly to SGD and is less sensitive to learning rate.

6.6 Submission Instructions

In addition to your answers to part 6.1 and your plots for part 6.4, you should submit your code and predictions for the test set to Gradescope using the file structure shown in Figure 5. You may also submit additional helper files if you would like. You should not submit any `.npz` files or `.npy` files other than your predictions (`mnist_test_pred.npy`); you should also not submit any `__pycache__` files, `.pyc` files, or any artifacts of your text editor.




```
submission.zip
    __init__.py
    main.py
    mnist_test_pred.npy
    npnn/
        __init__.py
        base.py
        dataset.py
        model.py
        modules.py
        optimizer.py
    requirements.txt
```

Figure 5: Gradescope Submission. Make sure to include these files in this exact directory structure!