Spring 2025 CS 4641/7641 A: Machine Learning Homework 4

Instructor: Dr. Mahdi Roozbahani

Deadline: Friday, April 18, 2025 11:59 pm EST

For Homework 4, the April 18th deadline is a hard and strict deadline. This means that this deadline cannot be even extended for students with GT-approved accomodations.

- Discussion is encouraged on Ed as part of the Q/A. However, all assignments should be done individually.
- Plagiarism is a **serious offense**. You are responsible for completing your own work. You are not allowed to copy and paste, or paraphrase, or submit materials created or published by others, as if you created the materials. All materials submitted must be your own.
- All incidents of suspected dishonesty, plagiarism, or violations of the Georgia Tech Honor Code will be subject to the institute's Academic Integrity procedures. If we observe any (even small) similarities/plagiarisms detected by Gradescope or our TAs, **WE WILL DIRECTLY REPORT ALL CASES TO OSI**, which may, unfortunately, lead to a very harsh outcome. **Consequences can be severe, e.g., academic probation or dismissal, grade penalties, a 0 grade for assignments concerned, and prohibition from withdrawing from the class.**

Instructions for the assignment

- This assignment consists of both programming and theory questions.
- Unless a theory question explicitly states that no work is required to be shown, you must provide an explanation, justification, or calculation for your answer.
- To switch between cell for code and for markdown, see the menu -> Cell -> Cell Type
- You can directly type Latex equations into markdown cells.
- If a question requires a picture, you could use this syntax `<img src="" style="width: 300px;"/>` to include them

within your ipython notebook.

- Your write up must be submitted in PDF format. Please ensure all questions are answered within the Jupyter Notebook using either Markdown or LaTeX. We will **NOT** accept handwritten work. Make sure that your work is formatted correctly, for example submit $\sum_{i=0} x_i$ instead of \text{sum_{i=0} x_i}
- When submitting the non-programming part of your assignment, you must correctly map pages of your PDF to each question/subquestion to reflect where they appear. **Improperly mapped questions may not be graded correctly and/or will result in point deductions for the error.**
- All assignments should be done individually, and each student must write up and submit their own answers.
- **Graduate Students**: You are required to complete any sections marked as Bonus for Undergrads

Using the autograder

- You will find three assignments (for grads) on Gradescope that correspond to HW4: "Assignment 4 Programming", "Assignment 4 - Non-programming" and "Assignment 4 Programming - Bonus for all". Undergrads will have an additional assignment called "Assignment 4 Programming - Bonus for Undergrads".
- You will submit your code for the autograder in the Assignment 4 Programming sections. Please refer to the Deliverables and Point Distribution section for what parts are considered required, bonus for undergrads, and bonus for all".
- We provided you different .py files and we added libraries in those files please DO NOT remove those lines and add your code after those lines. Note that these are the only allowed libraries that you can use for the homework
- You are allowed to make as many submissions until the deadline as you like. Additionally, note that the autograder tests each function separately, therefore it can serve as a useful tool to help you debug your code if you are not sure of what part of your implementation might have an issue
- **For the "Assignment 4 - Non-programming" part, you will need to submit to Gradescope a PDF copy of your Jupyter Notebook with the cells ran.** Please refer to the **Deliverables and Point Distribution** section for an outline of the non-programming questions.
- **When submitting to Gradescope, please make sure to mark the page(s) corresponding to each problem/sub-problem. The pages in the PDF should be of size 8.5" x 11", otherwise there may be a deduction in points for extra long sheets.**

- You **MUST** pass the Autograder Test to gain points for the programming section. There will not be any partial credit or manual grading for this part.

Using the local tests

- For some of the programming questions we have included a local test using a small toy dataset to aid in debugging. The local test sample data and outputs are stored in localtests.py
- There are no points associated with passing or failing the local tests, you must still pass the autograder to get points.
- **It is possible to fail the local test and pass the autograder** since the autograder has a certain allowed error tolerance while the local test allowed error may be smaller. Likewise, passing the local tests does not guarantee passing the autograder.
- **You do not need to pass both local and autograder tests to get points, passing the Gradescope autograder is sufficient for credit.**
- It might be helpful to comment out the tests for functions that have not been completed yet.
- It is recommended to test the functions as it gets completed instead of completing the whole class and then testing. This may help in isolating errors. Do not solely rely on the local tests, continue to test on the autograder regularly as well.

Deliverables and Points Distribution

Q1: Classification with Two Layer NN [85pts: 60pts + 25pts Grad / 3.6% Undergrad Bonus]

Deliverables: NN.py

- **1.1 NN Implementation** [55pts: 45pts + 10pts Grad / 1.6% **Bonus for Undergrad**] - *programming*
  - 1.1.1 Softmax [2.5pts]
  - 1.1.2 SiLU [2.5pts]
  - 1.1.3 Dropout [5pts]
  - 1.1.4 Cross Entropy loss [2.5pts]

    - 1.1.5 Forward propagation with and without dropout [5pts + 5pts]
    - 1.1.6 Compute gradients [7.5pts]
    - 1.1.7 Update gradients without Adam [5pts]
    - 1.1.8 Update gradients with Adam [5pts / 0.8% **Bonus for Undergrad**]
    - 1.1.9 Backward [5pt]
    - 1.1.10 Gradient Descent [2.5pt]
    - 1.1.11 Mini-batch Gradient Descent [2.5pts]
    - 1.1.12 Gradient Descent with Adam [2.5pts / 0.4% **Bonus for Undergrad**]
    - 1.1.13 Mini-batch Gradient Descent with Adam [2.5pts / 0.4% **Bonus for Undergrad**]
- **1.2 Training with Gradient Descent** [7.5pts] - *programming*
- **1.3 Training with Mini-batch Gradient Descent** [7.5pts] - *programming*
- **1.4 Training with Gradient Descent and Adam** [7.5pts Grad / 1% **Bonus for Undergrad**] - *programming*
- **1.5 Training with Mini-batch Gradient Descent and Adam** [7.5pts Grad / 1% **Bonus for Undergrad**] - *programming*

Q2: Image Classification based on CNNs [26pts: 10pts + 16pts Grad / 2.4% Bonus for Undergrad + 1.5% Bonus for All]

Deliverables: cnn.py, cnn_trainer.py, cnn_image_transformations.py and Written Report

- **2.1 Image Classification using Pytorch CNN** [26pts: 10pts + 16pts Grad / 2.4% **Bonus for Undergrad**] - *programming*
    - 2.1.1 Data Augmentation [5pts]
    - 2.1.2 Building the Model [5pts]
    - 2.1.3 Training and Tuning the Model [12pts Grad / 1.8% **Bonus for Undergrad**]

    - 2.1.4 Examining Loss Plots [2pts Grad / 0.3% **Bonus for Undergrad**]
    - 2.1.5 Evaluating Confusion Matrix [2pts Grad / 0.3% **Bonus for Undergrad**]
- **2.2 Exploring Deep CNN Architectures** [1.5% **Bonus for All**] - *non-programming*

Q3: Random Forest [30pts + 3% Bonus for All]

Deliverables: random_forest.py and Written Report

- **3.1 Random Forest Implementation** [15pts] - *programming*
    - 3.1.1 Bootstrapping [5pts] - *programming*
    - 3.1.2 Fitting random forest [10pts] - *programming*
- **3.2 Hyperparameter Tuning with a Random Forest** [15pts] - *programming*
    - 3.2.1 Implementing Grid Search [5pts] - *programming*
    - 3.2.2 Tuning random forest [5pts] - *programming*
    - 3.2.3 Examining confusion matrix [5pts] - *programming*
- **3.3 Plotting Feature Importance** [1% **Bonus for All**] - *non-programming*
- **3.4 ADABoost [2% **Bonus for All**]** - *programming*

Q4: SVM [15 pts]

Deliverables: Written Report

- **4.1 Fitting an SVM Classifier** [10 pts]
    - 4.1.1 Fit the SVM Classifier [7 pts] - *non programming*
    - 4.1.2 Plot the SVM Classifier [3 pts] - *non programming*
- **4.2 Using Kernels** [5 pts] - *non programming*

Q5: Next Character Prediction using Recurrent Neural Networks (RNNs) [5.5% Bonus for

all]

Deliverables: rnn.py, lstm.py and Written Report

- **5.1: Model Architecture** [3% **Bonus for All**] - *programming*
  - 5.1.1: Defining the Simple RNN model [1.5% **Bonus for All**]
  - 5.1.2: Defining the LSTM model [1.5% **Bonus for All**]
- **5.2: Simple RNN vs LSTM Model Text Generation Training Comparison Analysis** [2.5% **Bonus for All**] - *non programming*

Points Totals:

- Undergrad base: 115 pts
- Grad base: 156 pts
- Bonus for All: 10%
- Bonus for Undergrad: 6%

Submission Instructions

Run `python submission.py` from the root of the repository. This will generate two files in the repository:

- HW4_programming.zip - submit to the programming assignments on gradescope
- HW4_non_programming.pdf - submit to the non-programming assignments on gradescope

The autograder will automatically detect which files to test.

Alternatively, for the programming portions, you can manually submit the following files to their respective assignments on Gradescope:

- **Assignment 4 Programming**
  - NN.py
  - random_forest.py
  - cnn_image_transformations.py (for grads only)

    - cnn.py (for grads only)
    - cnn_trainer.py (for grads only)
- **Assignment 4 Programming - Bonus for All**
    - random_forest.py
    - rnn.py
    - lstm.py
    - base_sequential_model.py (you don't need to modify this file but need to submit it)
- **Assignment 4 Programming - Bonus for Undergrad**
    - NN.py
    - cnn_image_transformations.py
    - cnn.py
    - cnn_trainer.py

Environment Setup

In [1]:
```
import os
import random
import sys

import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import requests
from sklearn.model_selection import train_test_split
from utilities.utils import get_housing_dataset

print("Version information")

print("python: {}".format(sys.version))
print("matplotlib: {}".format(matplotlib.__version__))
print("numpy: {}".format(np.__version__))

%load_ext autoreload
%autoreload 2
%reload_ext autoreload
```

```
Version information
python: 3.11.11 (main, Dec 11 2024, 10:25:04) [Clang 14.0.6 ]
matplotlib: 3.10.0
numpy: 1.26.4
```

Coding and Emissions

Coding and computational research contribute to greenhouse gas emissions. The main source of these emissions is the power draw of computers during compute- and data-intensive computational analyses. In 2020, the sector of information and communication technologies was responsible for between 1.8% and 2.8% of GHG emissions, surprisingly more than the sector of aviation [1]. Machine learning models, especially large ones, can consume significant amounts of energy during training and inference, which contributes to greenhouse gas emissions. Artificial intelligence, including large language models, is also a significant emitter of carbon [2].

Carbon footprint of coding impacts several Sustainable Development Goals (SDGs), particularly SDG 13 (Climate Action) and SDG 12 (Responsible Consumption and Production).[3] This means writing clean and efficient code transcends functionality—it's an environmental imperative. As coders, we can play a role in mitigating this impact.

Measuring Our Impact:

CodeCarbon estimates the amount of CO2 produced by the cloud or personal computing resources used to execute the code[4] .

Using CodeCarbon in your upcoming assignment will help you understand the environmental impact of your code and explore ways to reduce it.

The code below will start tracking your carbon consumption and will print out total consumption at the end of the notebook.

In [2]:
```
from codecarbon import EmissionsTracker

emissions_dir = "./emissions"
if not os.path.exists(emissions_dir):
    os.makedirs(emissions_dir)
tracker = EmissionsTracker(
    log_level="error",
    save_to_file=True,
    output_dir=emissions_dir,
    allow_multiple_runs=True,
)
tracker.start()
```

```
[codecarbon WARNING @ 16:07:12] Multiple instances of codecarbon are allowed to run at the same time.
```

1: Two Layer Neural Network [85 pts: 60pts + 25pts Grad / 3.6% Undergrad Bonus] [P]

1.1 NN Implementation [55pts: 45pts + 10pts Grad / 1.6% Bonus for Undergrad] [P]

In this section, you will implement a two layer fully connected neural network to perform a Classification Task. You will also experiment with different activation functions and optimization techniques. We provide two activation functions here - SiLU and Softmax. You will implement a neural network where the first hidden layer uses a SiLU activation and the output layer

uses Softmax.

You'll also implement Gradient Descent (GD) and Mini-batch Gradient Descent (MBGD) algorithms for training these neural nets. In the **NN.py** file, complete the following functions:

- **softmax**
- **silu**
- **derivative_silu**
- **_dropout**
- **cross_entropy_loss**
- **forward**
- **compute_gradients**
- **update_weights**
- **backward**
- **gradient_descent**
- **minibatch_gradient_descent**

We'll train this neural network on sklearn's California Housing dataset.

Perceptron

A single layer perceptron can be thought of as a linear hyperplane as in logistic regression followed by a non-linear activation function.

$$u_i = \sum_{j=1}^{d} \theta_{ij} x_j + b_i$$

$$o_i = \phi\left(\sum_{j=1}^{d} \theta_{ij} x_j + b_i\right) = \phi(\theta_i^T x + b_i)$$

where x is a d-dimensional vector i.e. $x \in R^d$. It is one datapoint with d features. $\theta_i \in R^d$ is the weight vector for the i^{th} hidden unit, $b_i \in R$ is the bias element for the i^{th} hidden unit and $\phi(.)$ is a non-linear activation function that has been described below. u_i is a linear combination of the features in x_j weighted by θ_i whereas o_i is the i^{th} output unit from the

activation layer.

Fully connected Layer

Typically, a modern neural network contains millions of perceptrons as the one shown in the previous image. Perceptrons interact in different configurations such as cascaded or parallel. In this part, we describe a fully connected layer configuration in a neural network which comprises multiple parallel perceptrons forming one layer.

We extend the previous notation to describe a fully connected layer. Each layer in a fully connected network has a number of input/hidden/output units cascaded in parallel. Let us a define a single layer of the neural net as follows:
m denotes the number of hidden units in a single layer l whereas n denotes the number of units in the previous layer $l-1$.

$$u^{[l]} = \theta^{[l]} o^{[l-1]} + b^{[l]}$$

where $u^{[l]} \in R^m$ is a m-dimensional vector pertaining to the hidden units of the l^{th} layer of the neural network after applying linear operations. Similarly, $o^{[l-1]} \in R^n$ is the n-dimensional output vector corresponding to the hidden units of the $(l-1)^{th}$ activation layer. $\theta^{[l]} \in R^{m \times n}$ is the weight matrix of the l^{th} layer where each row of $\theta^{[l]}$ is analogous to θ_i described in the previous section i.e. each row corresponds to one hidden unit of the l^{th} layer. $b^{[l]} \in R^m$ is the bias vector of the layer where each element of b pertains to one hidden unit of the l^{th} layer. This is followed by element wise non-linear activation function $o^{[l]} = \phi(u^{[l]})$. The whole operation can be summarized as,

$$o^{[l]} = \phi(\theta^{[l]} o^{[l-1]} + b^{[l]})$$

where $o^{[l-1]}$ is the output of the previous layer.

Activation Function

There are many activation functions that are used for various purposes. For this question, we use SiLU and the softmax activation functions. We encourage you to explore the plethora of options, many of which are listed on Wikipedia.

Sigmoid

The sigmoid function is a non-linear function with an S-shaped curve and is regarded as a foundational activation function. Its

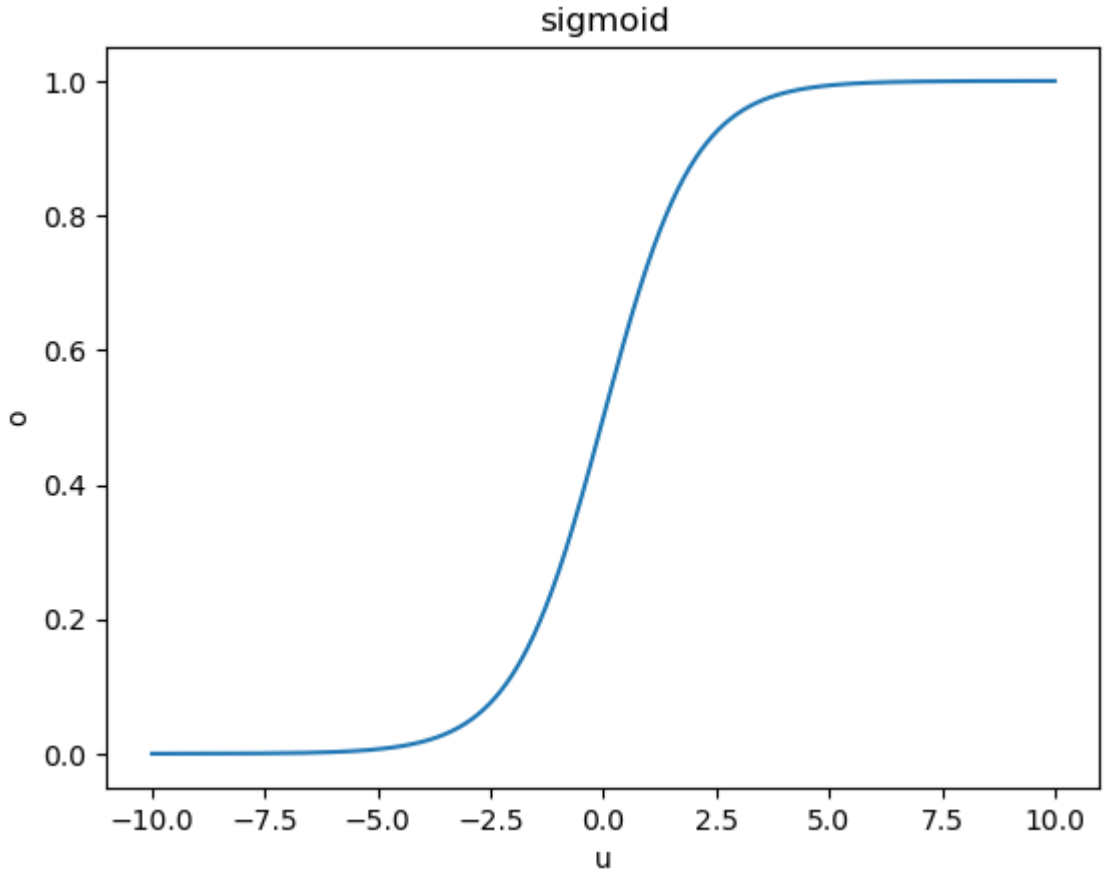
output is in the range $(0, 1)$, making it the function to use for binary classification output. The function is expressed as

$$o = \phi(u) = \frac{1}{1 + e^{-u}}$$

The derivation of the sigmoid function is given by

$$o' = \phi'(u) = \frac{1}{1 + e^{-u}}\left(1 - \frac{1}{1 + e^{-u}}\right) = o(1 - o)$$

Note: We do not use sigmoid in this homework; it is only included for the sake of completeness.



1.1.1 Softmax

Softmax is a common activation function used in neural networks, especially for multiclass classification problems like the one we are tackling. It is used to convert a vector of raw outputs from the last layer of the Neural Network into a probability distribution over multiple classes. The softmax function takes as input a vector of real numbers and transforms them into a probability distribution, ensuring that the probabilities sum to 1.

Mathematically, given an input vector of [x1, x2, ..., xn], the softmax function calculates the probability p(y=i) for each class i as follows:

p(y=i) = $e^{xi}/(e^{x1} + e^{x2} + \ldots + e^{xn})$

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

As discussed in class, the equation that we will use in this Neural network accounts for both the x values and the weights:

$$softmax(x\theta) = \frac{\exp(x\theta)_m}{\sum_{j=0}^{k} \exp(x\theta)_j}$$

TODO: Implement the function **softmax** in **NN.py**.

In [3]:
```
from utilities.localtests import TestNN

TestNN("test_softmax").test_softmax()
```

```
test_softmax passed!
```

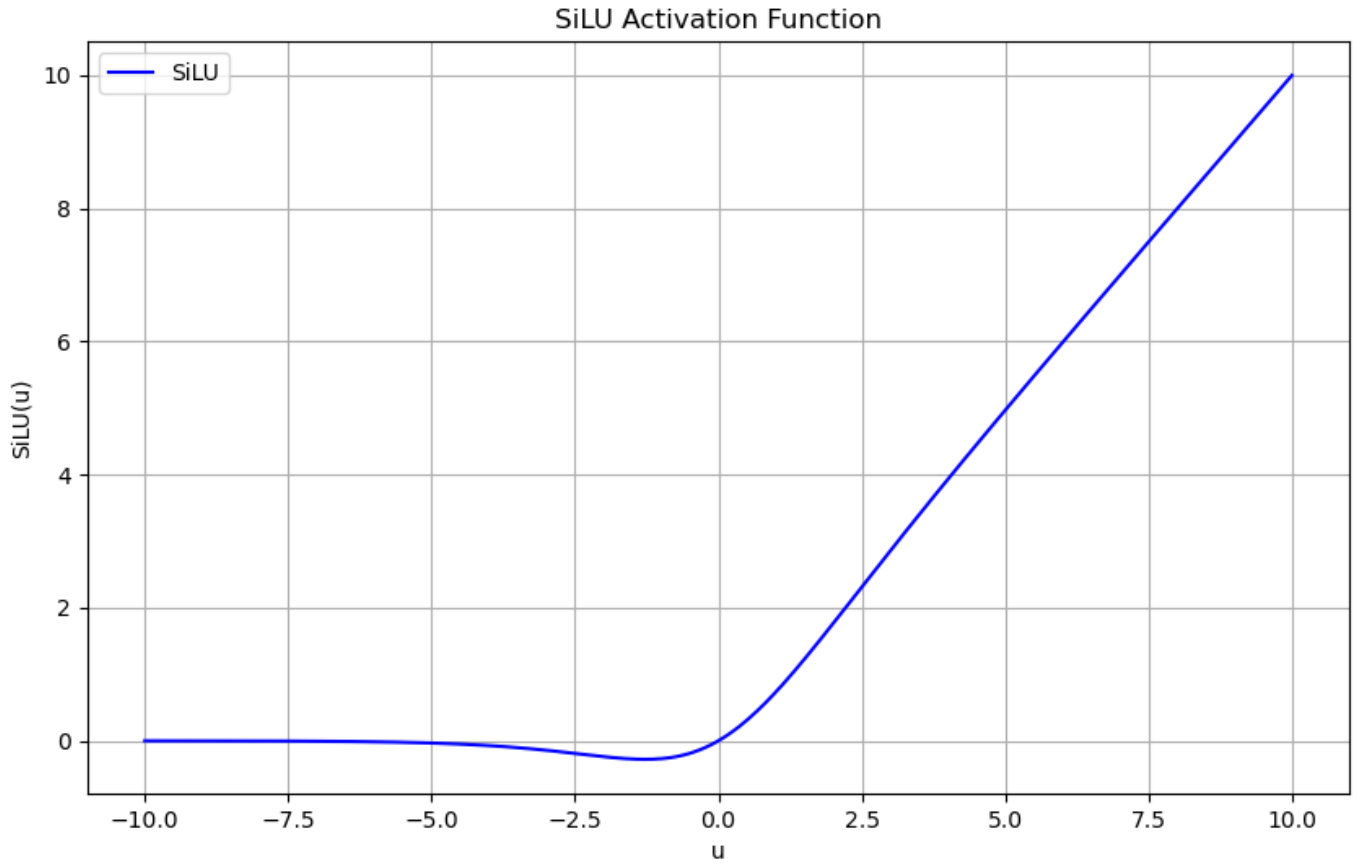
1.1.2 SiLU (Sigmoid Linear Unit)

The Sigmoid Linear Unit (SiLU), also known as the Swish activation function, is defined as:

$$o = \phi(u) = u \cdot \sigma(u)$$

where $\sigma(u)$ is the sigmoid function:

$$\sigma(u) = \frac{1}{1 + e^{-u}}$$

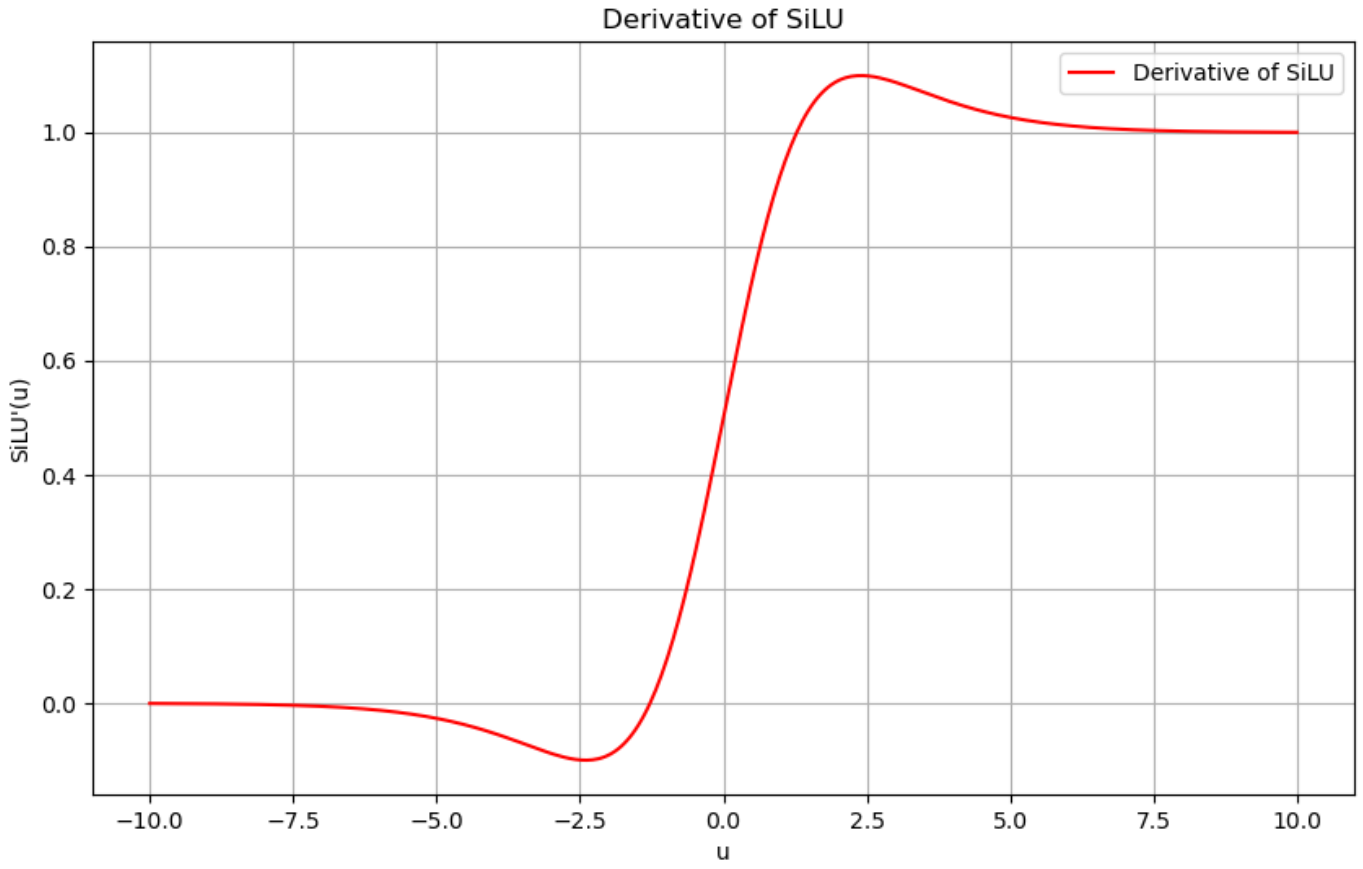


The derivative of SiLU, $\phi'(u)$, is given by:

$$\phi'(u) = \sigma(u) \cdot (1 + u \cdot (1 - \sigma(u)))$$

Unlike ReLU, SiLU is a smooth and non-linear activation function that retains gradients for negative inputs, which helps during training by improving gradient flow and enabling better convergence.

In this homework, we implement SiLU.



TODO: Implement the function **silu** and **derivative_silu** in **NN.py**.

In [4]:
```
from utilities.localtests import TestNN

TestNN("test_silu").test_silu()
TestNN("test_d_silu").test_d_silu()
```

```
test_silu passed!
test_d_silu passed!
```

1.1.3 Dropout

A dropout layer is a regularization technique used in neural networks to reduce overfitting. During training, a dropout layer looks at each input unit and randomly decide if it will be dropped (set to zero) with some given probability p. The decision for each unit is made independently. Formally, given an input of shape $N \times K$ (where N is the number of data points and K is the number of features), it samples from $\text{Bernoulli}(p)$ for each unit, resulting in an output where approximately pNK of the units are zero (in expectation). This forces the network to learn more robust and generalizable features, since it cannot rely too much on any particular input. During inference, the dropout layer is turned off, and the full network is used to make predictions.

The dropout probability p is a hyperparameter than can be tuned to adjust the strength of regularization. Setting $p = 0$ is equivalent to no dropout.

Note that the derivative of $\text{dropout}(u)$ with respect to u has the same shape as u. The values of the derivative depend on the random mask.

Use this as a reference for your implementation.

Note that after applying the mask, we must scale the result by a factor of $1/(1-p)$. Why is this necessary?

TODO: Implement the **_dropout** function in **NN.py**.

In [5]:
```
from utilities.localtests import TestNN

TestNN("test_dropout").test_dropout()
```

```
test_dropout passed!
```

1.1.4 Cross Entropy Loss

Cross-Entropy Loss is a widely used loss function in machine learning and deep learning, especially for classification tasks. It measures the dissimilarity between the predicted probability distribution and the true probability distribution of a classification problem. If it is closer to zero, the better the learnt function is.

Implementation details

For classification problems as in this exercise, we compute the loss as follows:

$$CE = -\frac{1}{N}\sum_{i=1}^{N}(y_i \cdot log(\hat{y}_i))$$

where y_i is the true label and $\hat{y}_i$ is the estimated label. Here, y_i/$\hat{y}_i$ are (1 x D) vectors and y/$\hat{y}$ are (N x D) vectors.

TODO: Implement the **cross_entropy_loss** function in **NN.py**.

In [6]:
```
from utilities.localtests import TestNN

TestNN("test_loss").test_loss()
```

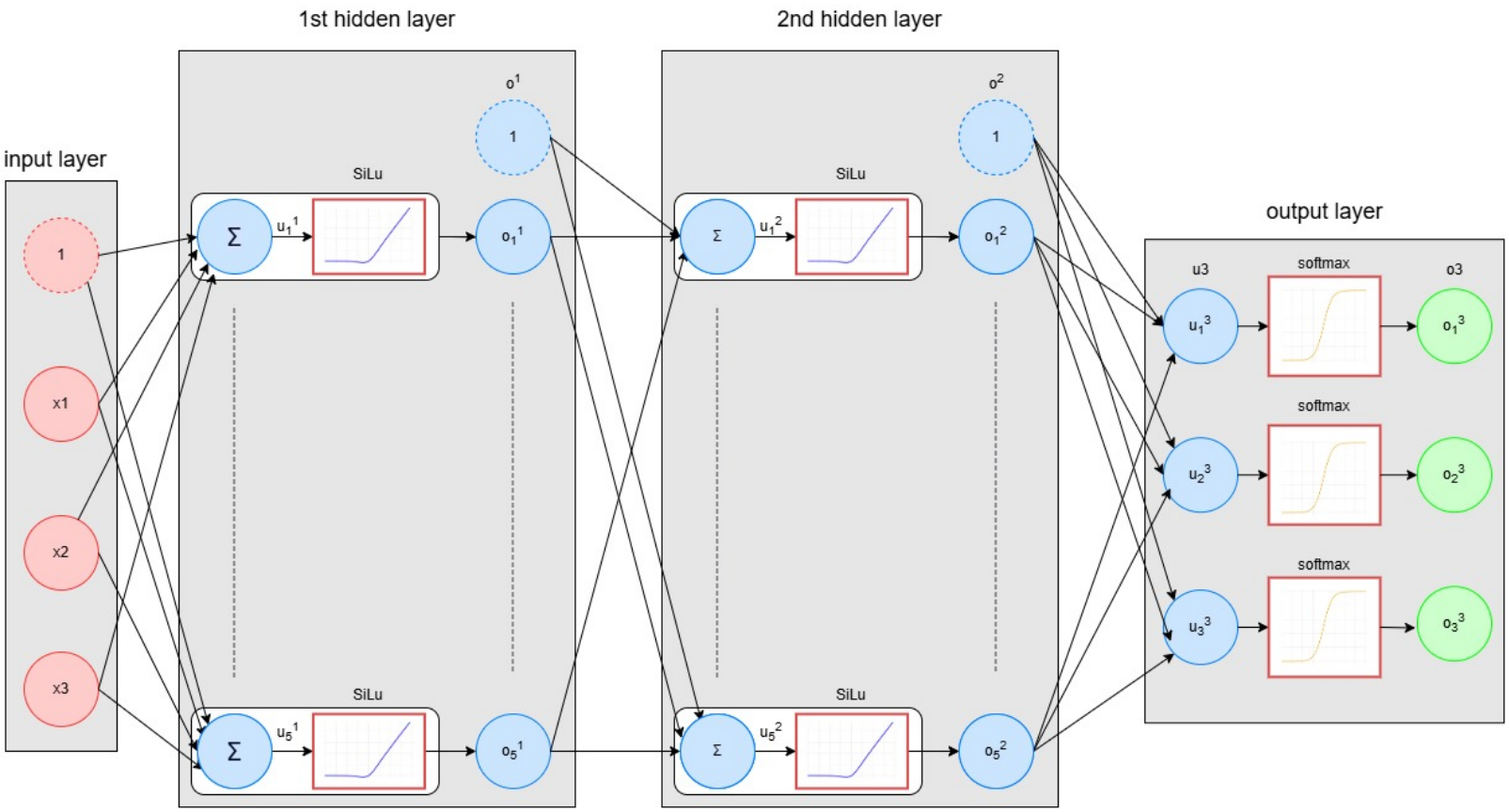
```
test_loss passed!
```

Neural Network Architecture

Now we will build a two layer neural network consisting of two hidden layers, each followed by a SiLU activation function. The logits outputted by the second hidden linear layer are then passed through the softmax function, which turns them into probability distributions over the 3 classes. Mathematically,

$$\begin{aligned}
u_1 &= \theta_1 X + b_1 \\
o_1 &= \text{silu}(u_1) \\
o_1 &= dropout(o1) \text{ if applicable} \\
u_2 &= \theta_2 o_1 + b_2 \\
o_2 &= \text{silu}(u_2) \\
u_3 &= \theta_3 o_2 + b_3 \\
o_3 &= \text{softmax}(u_3) \\
l &= \text{cross_entropy}(o_3)
\end{aligned}$$

Here is a diagram of the same architecture:



1.1.5 Forward Pass

TODO: Implement the **forward** function in **NN.py**.

Follow the equations given above to implement a full forward pass through the network. More details in the function description.

Here is a helpful guide that walks through the matrix multiplication operations and shapes involved in a forward and backward pass.

In [7]:
```
from utilities.localtests import TestNN

TestNN("test_forward_without_dropout").test_forward_without_dropout()
TestNN("test_forward_with_dropout").test_forward_with_dropout()
```

```
test_forward_without_dropout passed!
test_forward passed!
```

Backward Propagation: Compute Gradients and Update Weights

After the forward pass, we do back propagation to update the weights and biases in the direction of the negative gradient of the loss function.

1.1.6 Compute Gradients

In order to compute the gradients of the loss with respect to each parameter, we use the equations that make up the forward pass:

$$
\begin{aligned}
u_1 &= \theta_1 X + b_1 \\
o_1 &= \text{silu}(u_1) \\
u_2 &= \theta_2 o_1 + b_2 \\
o_2 &= \text{silu}(u_2) \\
u_3 &= \theta_3 o_2 + b_3 \\
o_3 &= \text{softmax}(u_3) \\
l &= \text{cross}\_\text{entropy}(o_3)
\end{aligned}
$$

When computing gradients, we travel backwards from the loss all the way back to the input. We first seek to obtain the derivative of the loss l with respect to the logits u_3. Note that they have the relation

$$
l = \text{cross}\_\text{entropy}(\text{softmax}(u_3))
$$

Computing the derivative of this seems very involved, but it actually has a very elegant result:

$$
\frac{\partial l}{\partial u_3} = \text{softmax}(u_3) - y = o_3 - y = \hat{y} - y.
$$

where $\hat{y}$ is predicted y or o_3.

While this is given to you, we encourage you to derive it for yourself! You can find a great explanation of the derivation in this article.

Now that we have $\frac{\partial l}{\partial u_3}$, we seek to move further back and compute $\frac{\partial l}{\partial \theta_3}$ and $\frac{\partial l}{\partial b_3}$. This is done using the chain rule:

$$\frac{\partial l}{\partial \theta_3} = \frac{\partial l}{\partial u_3} \cdot \frac{\partial u_3}{\partial \theta_3}$$
$$\frac{\partial l}{\partial b_3} = \frac{\partial l}{\partial u_3} \cdot \frac{\partial u_3}{\partial b_3}.$$

The quantities $\frac{\partial u_3}{\partial \theta_3}$ and $\frac{\partial u_3}{\partial b_3}$ are easy to derive from the relation $u_3 = \theta_3 o_2 + b_3$. We see that

$$\frac{\partial l}{\partial \theta_3} = \frac{\partial l}{\partial u_3} \cdot o_2$$
$$\frac{\partial l}{\partial b_3} = \frac{\partial l}{\partial u_3} \cdot 1.$$

Note that the derivative involves o_2, which we computed during the forward pass. Fortunately, we saved that value in `self.cache`, so we don't need to compute it again!

The same procedure is repeated to obtain the gradients for the upstream parameters θ_2 and b_2. We must first perform the intermediate steps of computing the derivative of the loss with respect to o_2 and then u_2. These are given by

$$\frac{\partial l}{\partial o_2} = \frac{\partial l}{\partial u_3} \cdot \theta_3$$
$$\frac{\partial l}{\partial u_2} = \frac{\partial l}{\partial o_2} \cdot \frac{\partial \, \mathrm{SiLu}}{\partial u_2}.$$

The same procedure is repeated to obtain the gradients for the upstream parameters θ_1 and b_1. We must first perform the intermediate steps of computing the derivative of the loss with respect to o_1 and then u_1. These are given by

$$\frac{\partial l}{\partial o_1} = \frac{\partial l}{\partial u_2} \cdot \theta_2$$
$$\frac{\partial l}{\partial u_1} = \frac{\partial l}{\partial o_1} \cdot \frac{\partial \, \text{SiLu}}{\partial u_1}.$$

In the second relation, we must consider our use of dropout! If we applied dropout on a particular neuron, it should not be adjusted. To account for this, in the case of `use_dropout=True`, we must instead use

$$\frac{\partial l}{\partial u_1} = \frac{\partial l}{\partial o_1} \cdot \frac{\partial \, \text{SiLu}}{\partial u_1} \cdot \text{dropout\_mask} \cdot \frac{1}{1-p},$$

where $1/(1-p)$ is the scaling factor and dropout_mask is stored in `self.cache`.

The final step! We can use these values to compute the gradients for θ_1 and b_1, using the relation $u_1 = \theta_1 X + b_1$, which are given by

$$\frac{\partial l}{\partial \theta_1} = \frac{\partial l}{\partial u_1} \cdot X$$
$$\frac{\partial l}{\partial b_1} = \frac{\partial l}{\partial u_1} \cdot 1.$$

Implementation Tips

The above equations are given in matrix notation. When implementing these computations in code, the easiest way to make sure you are calculating the values correctly and in the right order is to check shapes. Any time you are doing a matrix/vector operation in NumPy, **check the shapes**.

Since we are computing these gradients over N data points, we must divide the gradients by N to take the *average* gradient. Make sure you are dividing by N exactly once, no more and no less!

TODO: Implement the **compute_gradients** function in **NN.py**.

Note: Implement drop out function only on the first hidden layer!

Hint: Refer to this guide for more detail on computing gradients.

In [8]:
```
from utilities.localtests import TestNN

TestNN(
    "test_compute_gradients_without_dropout"
).test_compute_gradients_without_dropout()
TestNN("test_compute_gradients_with_dropout").test_compute_gradients_with_dropout()
```

```
test_compute_gradients_without_dropout passed!
test_compute_gradients_with_dropout passed!
```

1.1.7 Update Weights

So, we update the weights and biases using the following formulas

$$\theta^{[3]} := \theta^{[3]} - lr \times \frac{\partial l}{\partial \theta^{[3]}}$$
$$b^{[3]} := b^{[3]} - lr \times \frac{\partial l}{\partial b^{[3]}}$$
$$\theta^{[2]} := \theta^{[2]} - lr \times \frac{\partial l}{\partial \theta^{[2]}}$$
$$b^{[2]} := b^{[2]} - lr \times \frac{\partial l}{\partial b^{[2]}}$$
$$\theta^{[1]} := \theta^{[1]} - lr \times \frac{\partial l}{\partial \theta^{[1]}}$$
$$b^{[1]} := b^{[1]} - lr \times \frac{\partial l}{\partial b^{[1]}}$$

where lr is the learning rate. It decides the step size we want to take in the direction of the negative gradient.

TODO: Implement the **update_weights** function in **NN.py** with use_adam=False.

Hint: Refer to this guide for more detail on the backward pass.

In [9]:
```
from utilities.localtests import TestNN

TestNN("test_update_weights").test_update_weights()
```

```
test_update_weights passed!
```

1.1.8 Update Weights with Adam [Bonus for Undergrad]

Gradient descent does a generally good job of facilitating the convergence of the model's parameters to minimize the loss function. However, the process of doing so can be slow and/or noisy.

Adam (Adaptive Moment Estimation) is an advanced optimization algorithm that combines the benefits of two other popular optimization techniques: RMSprop and momentum. Introduced by Kingma and Ba in 2014, Adam has become one of the most widely used optimization algorithms in deep learning due to its efficiency and effectiveness across a wide range of problems. It adapts the learning update for each parameter individually, making it particularly well-suited for problems with sparse gradients or noisy data.

As a reminder, vanilla gradient descent applies the following update function to the parameters:

$$\theta_{t+1} = \theta_t - \alpha \nabla f(\theta_t) \tag{1}$$

where θ_t represents the parameters at time t, α represents the learning rate, and f is the loss function.

Adam proposes the following tweak to our parameter update function:

Firstly, it defines two variables M_t and V_t, every variable in the layer would need these two to update in for each iteration.

$$M_t = \beta_1 M_{t-1} + (1 - \beta_1) \nabla f(\theta_t)$$
$$V_t = \beta_2 V_{t-1} + (1 - \beta_2) \nabla f(\theta_t)^2$$

Here:

- t is a single parameter in our network (e.g. theta3 or bias1)
- M_t **(First Moment)**
  - This tracks the exponentially weighted moving average of the gradients of parameter t. It's similar to momentum and helps the optimization continue moving in consistent directions. Essentially, it's an estimate of the mean of the gradients.

    - Intuition: Think of it as a ball rolling down a hill - it builds up momentum in promising directions and helps overcome small obstacles (local minima) along the way. It remembers where we've been going and helps us stay on course.
- V_t **(Second Moment)**
    - This tracks the exponentially weighted moving average of the squared gradients of parameter t. It captures the variance of gradients, which helps adapt the learning update for each parameter.
    - Intuition: Imagine driving through varied terrain - you'd slow down on rough roads (high gradient variance) and speed up on smooth highways (low gradient variance). V_t provides this terrain-specific speed control for each parameter.
- β_1 **and** β_2: These hyperparameters control how much the algorithm relies on information from new gradient vs. previously calculated M_t or V_t for the parameter t:
    - β_1 (typically 0.9) controls the decay rate of the moving average of the gradient
    - β_2 (typically 0.999) controls the decay rate of the moving average of the squared gradient
    - The values show we trust our history (90% for direction, 99.9% for variance) more than any single new piece of information. This makes Adam robust to noisy gradients in a single batch.

Before we use these moment estimates to update our variables, they need to be standardized to correct for initialization bias:

$$\hat{M}_t = \frac{M_t}{1 - \beta_1^t}$$
$$\hat{V}_t = \frac{V_t}{1 - \beta_2^t}$$

This is also called bias correction. It's useful because the initial estimates are usually not reliable. These correction terms prevent the early updates from being too small by compensating for the fact that M_0 and V_0 start at zero.

Now we can have our updating formula for Adam:

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{M}_t}{\sqrt{\hat{V}_t} + \epsilon}$$

ϵ is a constant equal to 10^{-8} to avoid numerical error in the denominator.

Overall, Adam works really well because we maintain M_t and V_t for every parameter.

- Parameters with clear, consistent gradients get larger updates (high M_t, low V_t)

- Parameters with noisy or sparse gradients get smaller, more careful updates (high V_t, low M_t)
- The overall direction benefits from momentum, smoothing the optimization path
- Each parameter gets its own custom learning rate, automatically tuned based on its gradient history

Here are some resources for learning more about Adam optimization:

- Who's Adam and What's He Optimizing? | Deep Dive into Optimizers for Machine Learning!
- Complete Guide to the Adam Optimization Algorithm

TODO: Implement the **update_weights** function in **NN.py** with use_adam=True.

In [8]:
```
from utilities.localtests import TestNN

TestNN("test_update_weights_with_adam").test_update_weights_with_adam()
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
Cell In[8], line 3
      1 from utilities.localtests import TestNN
----> 3 TestNN("test_update_weights_with_adam").test_update_weights_with_adam()

File ~/Downloads/GT/CS4641/HW4/utilities/localtests.py:976, in TestNN.test_update_weights_with_adam(self)
    973 dLoss = pickle.load(dLoss_file)
    974 dLoss_file.close()
--> 976 nn.update_weights(dLoss)
    977 student = nn.parameters
    978 # print(student)

File ~/Downloads/GT/CS4641/HW4/NN.py:374, in NeuralNet.update_weights(self, dLoss)
    350 def update_weights(self, dLoss):
    351     """
    352     Update the weights in the network using the gradients of the parameters (dLoss). You should upd
ate self.parameters.
    353     For every parameter:
   (...)
    372     Otherwise, you only need to update self.parameters.
    373     """
--> 374     raise NotImplementedError()

NotImplementedError:
```

1.1.9 Backward Pass

Now, we can combine the two functions `compute gradients` and `update_weights` to perform a complete backward pass through the network.

TODO: Implement the **backward** function in **NN.py**.

In [10]:
```
from utilities.localtests import TestNN

TestNN("test_backward").test_backward()
```

```
test_backward passed!
```

1.1.10 Gradient Descent [2.5pts]

TODO: Implement the **gradient_descent** function in **NN.py**.

This method is also commonly known as batch gradient descent. Look at the function documentation in gradient_descent for guidance. You may test your implementation of the gradient descent function contained in **NN.py** in the cell below. See Using the Local Tests for more details.

Below, we test the first three iterations as a sanity check. For the final Gradescope evaluation, your implementation will be trained for a larger number of iterations to fully optimize the network. To achieve full credit, your network must attain a final cross entropy loss of 1 or lower.

In [11]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
from utilities.localtests import TestNN

TestNN("test_gradient_descent").test_gradient_descent()
```

```
Loss after iteration 0: 1.086276
Loss after iteration 1: 1.086233
Loss after iteration 2: 1.086189

Your GD losses works within the expected range: True
```

1.1.11 Mini-batch Gradient Descent [2.5 pts]

TODO: Implement the **minibatch_gradient_descent** function in **NN.py**.

Look at the function documentation in gradient_descent for guidance. You may test your implementation of the mini-batch gradient descent function contained in **NN.py** in the cell below. See Using the Local Tests for more details.

Below, we test the first three iterations as a sanity check. For the final Gradescope evaluation, your implementation will be trained for a larger number of iterations to fully optimize the network. To achieve full credit, your network must attain a final cross entropy loss of 1 or lower.

In [12]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

from utilities.localtests import TestNN

TestNN("test_minibatch_gradient_descent").test_minibatch_gradient_descent()
```

```
Loss after iteration 0: 1.085044
Loss after iteration 1: 1.110789
Loss after iteration 2: 1.109005
Your batch_y works within the expected range: True

Your mini-batch GD losses works within the expected range: True
```

1.1.12 Local Test: Gradient Descent with Adam [2.5 pts / 0.35% Bonus for Undergrad]

You may test your implementation of the GD function with adam contained in **NN.py** in the cell below. See Using the Local Tests for more details. Revisit your implementation for update_weights.

Below, we test the first three iterations as a sanity check. For the final Gradescope evaluation, your implementation will be trained for a larger number of iterations to fully optimize the network. To achieve full credit, your network must attain a final cross entropy loss of 1 or lower.

In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
```

```
##############################

from utilities.localtests import TestNN

TestNN("test_gradient_descent_with_adam").test_gradient_descent_with_adam()
```

1.1.13 Mini-batch Gradient Descent with Adam [2.5 pts / 0.35% Bonus for Undergrad]

You may test your implementation of the mini-batch gradient descent function contained in **NN.py** when use_adam=True in the cell below. See Using the Local Tests for more details.

Below, we test the first three iterations as a sanity check. For the final Gradescope evaluation, your implementation will be trained for a larger number of iterations to fully optimize the network. To achieve full credit, your network must attain a final cross entropy loss of 1 or lower.

In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

from utilities.localtests import TestNN

TestNN(
    "test_minibatch_gradient_descent_with_adam"
).test_minibatch_gradient_descent_with_adam()
```

1.2 Loss plot and cross-entropy(CE) value with Gradient Descent [7.5pts] [P]

Now, you can fully train your neural network implementation with gradient descent. The following cells will plot the loss vs epoch graph and calculate the final test cross-entropy(CE).

You can test and debug your network here, but your implementation will be tested on gradescope so there is no partial credit for notebook output.

To achieve full credit on gradescope:

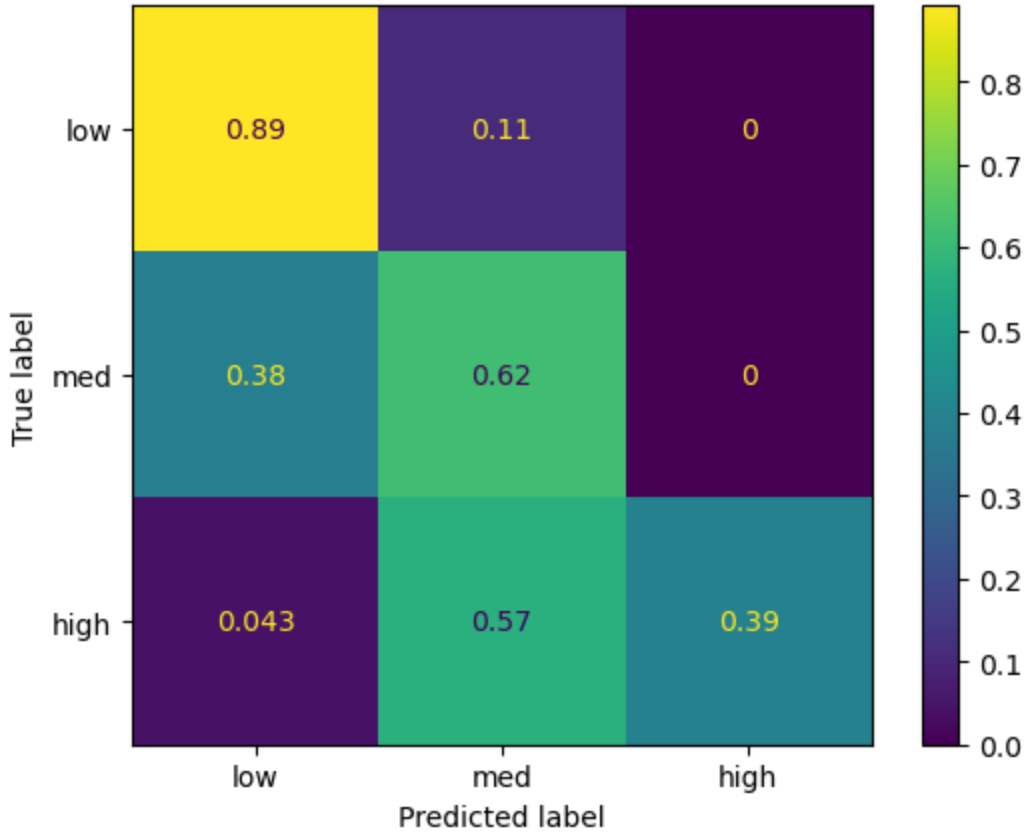
1. Your loss trajectory should be smooth, decreasing and similar to expected trajectory (check expected output PDF) (2.5 pts)
2. Your final cross entropy loss must be 0.8 or lower (2.5 pts)
3. The diagonal entries in your confusion matrix must be within 0.15 of the expected values (check expected output PDF) (2.5 pts)

In [75]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from NN import NeuralNet
from sklearn.metrics import ConfusionMatrixDisplay

x_train, y_train, x_test, y_test = get_housing_dataset()

nn = NeuralNet(
    y_train, lr=0.01, use_dropout=False, use_adam=False
)  # initalize neural net class
nn.gradient_descent(x_train, y_train, iter=60000)  # train
```

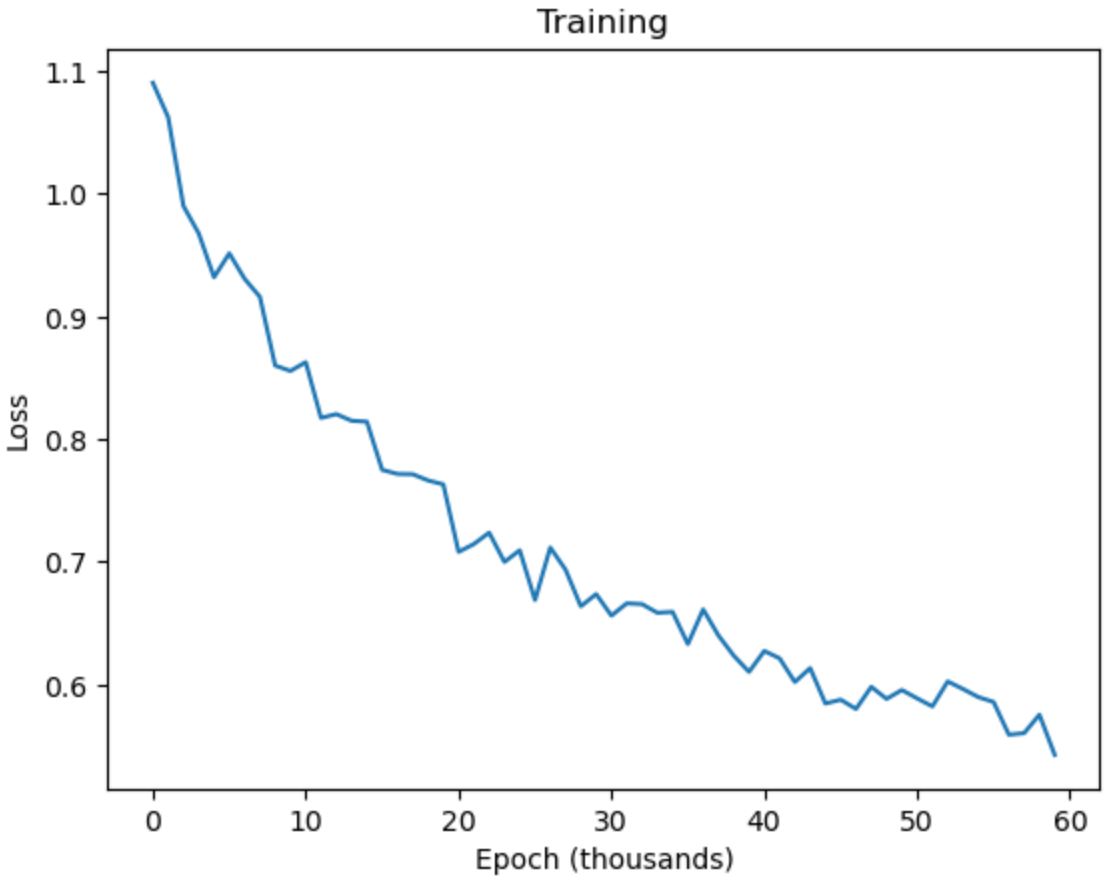
```
Loss after iteration 0: 1.090494
Loss after iteration 1000: 1.062430
Loss after iteration 2000: 0.990095
Loss after iteration 3000: 0.967734
Loss after iteration 4000: 0.932072
Loss after iteration 5000: 0.951524
Loss after iteration 6000: 0.930790
Loss after iteration 7000: 0.916113
Loss after iteration 8000: 0.860116
Loss after iteration 9000: 0.855675
Loss after iteration 10000: 0.862852
Loss after iteration 11000: 0.817382
Loss after iteration 12000: 0.820510
Loss after iteration 13000: 0.815087
Loss after iteration 14000: 0.814353
Loss after iteration 15000: 0.775090
Loss after iteration 16000: 0.771753
Loss after iteration 17000: 0.771533
Loss after iteration 18000: 0.766447
Loss after iteration 19000: 0.763241
Loss after iteration 20000: 0.708224
Loss after iteration 21000: 0.714631
Loss after iteration 22000: 0.723890
Loss after iteration 23000: 0.700031
Loss after iteration 24000: 0.709524
Loss after iteration 25000: 0.668985
Loss after iteration 26000: 0.711763
Loss after iteration 27000: 0.693683
Loss after iteration 28000: 0.663898
Loss after iteration 29000: 0.673841
Loss after iteration 30000: 0.656272
Loss after iteration 31000: 0.666280
Loss after iteration 32000: 0.665686
Loss after iteration 33000: 0.658613
Loss after iteration 34000: 0.659308
Loss after iteration 35000: 0.633127
Loss after iteration 36000: 0.661329
Loss after iteration 37000: 0.640109
Loss after iteration 38000: 0.623684
Loss after iteration 39000: 0.610423
Loss after iteration 40000: 0.627478
Loss after iteration 41000: 0.621514
```

```
Loss after iteration 42000: 0.602149
Loss after iteration 43000: 0.613525
Loss after iteration 44000: 0.584646
Loss after iteration 45000: 0.587781
Loss after iteration 46000: 0.580227
Loss after iteration 47000: 0.598253
Loss after iteration 48000: 0.588436
Loss after iteration 49000: 0.595655
Loss after iteration 50000: 0.588847
Loss after iteration 51000: 0.582291
Loss after iteration 52000: 0.602659
Loss after iteration 53000: 0.596477
Loss after iteration 54000: 0.589918
Loss after iteration 55000: 0.585913
Loss after iteration 56000: 0.559253
Loss after iteration 57000: 0.560517
Loss after iteration 58000: 0.575534
Loss after iteration 59000: 0.542832
```

In [76]:
```
# Plot confusion matrix
y_true = np.argmax(y_test, axis=1)
y_pred = nn.predict(x_test)
display_labels = ["low", "med", "high"]
ConfusionMatrixDisplay.from_predictions(
    y_true, y_pred, normalize="true", display_labels=display_labels
)
plt.show()
```



In [77]:
```
# Plot training loss
fig = plt.plot(np.array(nn.loss).squeeze())
plt.title(f"Training")
plt.xlabel("Epoch (thousands)")
plt.ylabel("Loss")
plt.show()
```



In [78]:
```
# Total loss
y_hat = nn.forward(x_test, use_dropout=False)
print("Cross entropy loss:", round(nn.cross_entropy_loss(y_test, y_hat), 3))
```

```
Cross entropy loss: 0.828
```

1.3 Loss plot and CE value for Mini-batch GD [7.5pts] [P]

Train your neural network implementation with mini-batch gradient descent and print out the loss at every 1000th iteration (starting at iteration 0). The following cells will plot the loss vs epoch graph and calculate the final test CE.

You can test and debug your network here, but your implementation will be tested on gradescope so there is no partial credit

for notebook output.

To achieve full credit on gradescope:

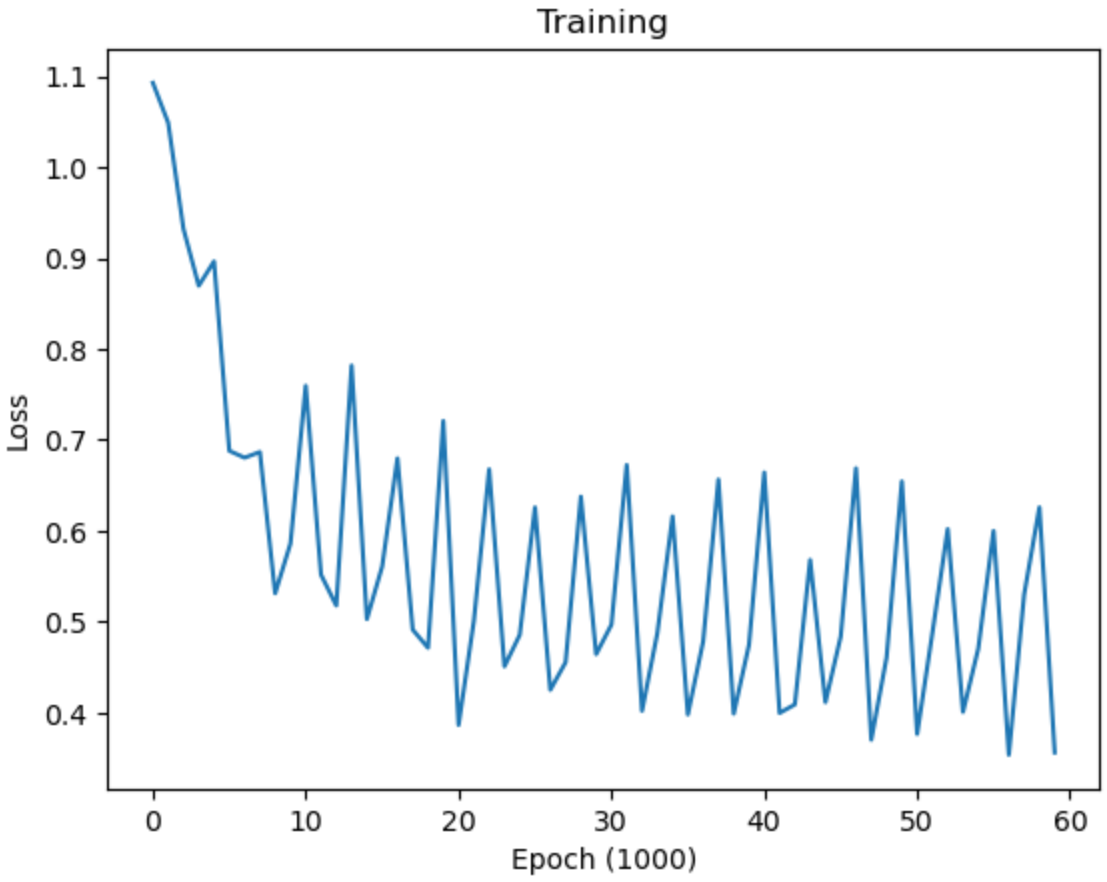
1. Your loss trajectory should be decreasing and similar to expected trajectory (check expected output PDF) (2.5 pts)
2. Your final cross entropy loss must be 0.8 or lower (2.5 pts)
3. The diagonal entries in your confusion matrix must be within 0.15 of the expected values (check expected output PDF) (2.5 pts)

In [17]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from NN import NeuralNet
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

x_train, y_train, x_test, y_test = get_housing_dataset()

nn = NeuralNet(
    y_train, lr=0.01, use_dropout=True, batch_size=64, use_adam=False
)  # initalize neural net class
nn.minibatch_gradient_descent(x_train, y_train, iter=60000)
```

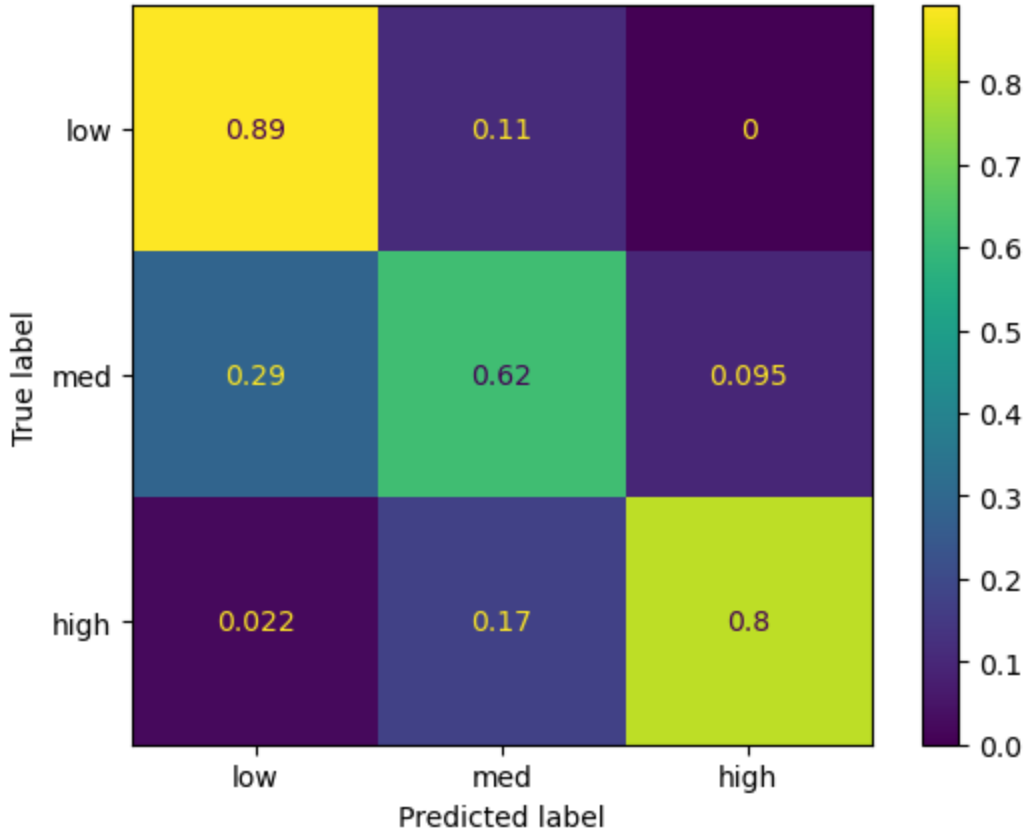
```
Loss after iteration 0: 1.092869
Loss after iteration 1000: 1.048985
Loss after iteration 2000: 0.932282
Loss after iteration 3000: 0.869664
Loss after iteration 4000: 0.896471
Loss after iteration 5000: 0.688057
Loss after iteration 6000: 0.680472
Loss after iteration 7000: 0.686783
Loss after iteration 8000: 0.531214
Loss after iteration 9000: 0.586094
Loss after iteration 10000: 0.759610
Loss after iteration 11000: 0.551759
Loss after iteration 12000: 0.517801
Loss after iteration 13000: 0.781921
Loss after iteration 14000: 0.502840
Loss after iteration 15000: 0.560831
Loss after iteration 16000: 0.679815
Loss after iteration 17000: 0.491290
Loss after iteration 18000: 0.471381
Loss after iteration 19000: 0.720984
Loss after iteration 20000: 0.386434
Loss after iteration 21000: 0.500964
Loss after iteration 22000: 0.667739
Loss after iteration 23000: 0.450866
Loss after iteration 24000: 0.485587
Loss after iteration 25000: 0.625858
Loss after iteration 26000: 0.425065
Loss after iteration 27000: 0.455403
Loss after iteration 28000: 0.637729
Loss after iteration 29000: 0.464308
Loss after iteration 30000: 0.496887
Loss after iteration 31000: 0.672519
Loss after iteration 32000: 0.402011
Loss after iteration 33000: 0.487205
Loss after iteration 34000: 0.615984
Loss after iteration 35000: 0.397972
Loss after iteration 36000: 0.478280
Loss after iteration 37000: 0.656518
Loss after iteration 38000: 0.398895
Loss after iteration 39000: 0.474652
Loss after iteration 40000: 0.664267
Loss after iteration 41000: 0.399627
```

```
Loss after iteration 42000: 0.409015
Loss after iteration 43000: 0.568077
Loss after iteration 44000: 0.411560
Loss after iteration 45000: 0.483707
Loss after iteration 46000: 0.668879
Loss after iteration 47000: 0.370174
Loss after iteration 48000: 0.460681
Loss after iteration 49000: 0.654509
Loss after iteration 50000: 0.376518
Loss after iteration 51000: 0.487919
Loss after iteration 52000: 0.602000
Loss after iteration 53000: 0.400822
Loss after iteration 54000: 0.470511
Loss after iteration 55000: 0.599949
Loss after iteration 56000: 0.353616
Loss after iteration 57000: 0.529899
Loss after iteration 58000: 0.625933
Loss after iteration 59000: 0.356204
```

In [18]:
```
# Plot training loss
fig = plt.plot(np.array(nn.loss).squeeze())
plt.title(f"Training")
plt.xlabel("Epoch (1000)")
plt.ylabel("Loss")
plt.show()
```



In [19]:
```
# Plot confusion matrix
y_true = np.argmax(y_test, axis=1)
y_pred = nn.predict(x_test)
display_labels = ["low", "med", "high"]
ConfusionMatrixDisplay.from_predictions(
    y_true, y_pred, normalize="true", display_labels=display_labels
)
plt.show()
```



In [20]:
```
# Total loss
y_hat = nn.forward(x_test, use_dropout=False)
print("Cross entropy loss:", round(nn.cross_entropy_loss(y_test, y_hat), 3))
```

```
Cross entropy loss: 0.692
```

1.4 Loss plot and CE value for Gradient Descent with Adam [7.5pts Grad / 1% Bonus for Undergrad] [P]

Train your neural net implementation using gradient descent with Adam and print out the loss at every 1000th iteration (starting at iteration 0). The following cells will plot the loss vs epoch graph and calculate the final test CE.

To achieve full credit on gradescope:

1. Your loss trajectory should be decreasing and similar to expected trajectory (check expected output PDF) (2.5 pts)
2. Your final cross entropy loss must be 0.8 or lower (2.5 pts)
3. The diagonal entries in your confusion matrix must be within 0.15 of the expected values (check expected output PDF) (2.5 pts)

In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

from NN import NeuralNet
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

x_train, y_train, x_test, y_test = get_housing_dataset()

nn = NeuralNet(
    y_train, lr=0.0001, use_dropout=True, use_adam=True
)  # initalize neural net class
nn.gradient_descent(x_train, y_train, iter=60000)  # train
```

In [ ]:
```
# Plot training loss
fig = plt.plot(np.array(nn.loss).squeeze())
plt.title(f"Training")
plt.xlabel("Epoch (1000)")
plt.ylabel("Loss")
plt.show()
```

In [ ]:
```
# Plot confusion matrix
y_true = np.argmax(y_test, axis=1)
y_pred = nn.predict(x_test)
display_labels = ["low", "med", "high"]
ConfusionMatrixDisplay.from_predictions(
    y_true, y_pred, normalize="true", display_labels=display_labels
)
plt.show()
```

In [ ]:
```
# Total loss
y_hat = nn.forward(x_test, use_dropout=False)
print("Cross entropy loss:", round(nn.cross_entropy_loss(y_test, y_hat), 3))
```

1.5 Loss plot and CE value for Mini-batch GD with Adam [7.5pts Grad / 1% Bonus for Undergrad] [P]

Now, you can train your neural network implementation with mini-batch gradient descent with Adam and print out the loss at every 1000th iteration (starting at iteration 0). The following cells will plot the loss vs epoch graph and calculate the final test CE.

To achieve full credit on gradescope:

1. Your loss trajectory should be decreasing and similar to expected trajectory (check expected output PDF) (2.5 pts)
2. Your final cross entropy loss must be 0.8 or lower (2.5 pts)
3. The diagonal entries in your confusion matrix must be within 0.15 of the expected values (check expected output PDF) (2.5 pts)

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from NN import NeuralNet
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

x_train, y_train, x_test, y_test = get_housing_dataset()

nn = NeuralNet(
    y_train, lr=0.0001, batch_size=64, use_dropout=True, use_adam=True
)  # initalize neural net class
nn.minibatch_gradient_descent(x_train, y_train, iter=60000)
```

In [ ]:
```
# Plot training loss
fig = plt.plot(np.array(nn.loss).squeeze())
plt.title(f"Training")
plt.xlabel("Epoch (1000)")
plt.ylabel("Loss")
plt.show()
```

In [ ]:
```
# Plot confusion matrix
y_true = np.argmax(y_test, axis=1)
```

```
y_pred = nn.predict(x_test)
display_labels = ["low", "med", "high"]
ConfusionMatrixDisplay.from_predictions(
    y_true, y_pred, normalize="true", display_labels=display_labels
)
plt.show()
```

In [ ]:
```
# Total loss
y_hat = nn.forward(x_test, use_dropout=False)
print("Cross entropy loss:", round(nn.cross_entropy_loss(y_test, y_hat), 3))
```

2: Image Classification based on CNNs [26pts: 10pts + 16pts Grad / 2.4% Bonus for Undergrad + 1.5% Bonus for all] [P] [W]

2.1 Image Classification using Pytorch and CNN [26pts: 10pts + 16pts Grad / 2.4% Bonus for Undergrad] [P] [W]

Pytorch Description

Pytorch is a Machine Learning/Deep Learning tensor library based on Python and Torch that uses dynamic computation graphs. Pytorch is used for applications using GPUs and CPUs.

Helpful Links

- Install Pytorch
- Pytorch Quickstart Tutorial

Setup Pytorch

Make sure you installed pytorch and torchvision (directions here).

Please also see Pytorch Quickstart Tutorial to see how to load a data set, build a training loop, and test the model. Another good resource for building CNNs using Pytorch is here.

Environment Setup

In [21]:
```
import numpy as np
import torch

%load_ext autoreload
%autoreload 2
%reload_ext autoreload
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

2.1.1 Load Brain Tumor MRI Dataset and Data Augmentation

We use Brain Tumor MRI Dataset to train our model. This dataset contains 7023 images of human brain MRI images which are classified into 4 classes: glioma, meningioma, no tumor, and pituitary. There are 5712 training images and 1311 test images. We adapt the code from Brain-Tumor-MRI-Classification to preprocess the downloaded Brain Tumor MRI Dataset.

Data Augmentation [5pts]

Data augmentation is a technique to increase the diversity of your training set by applying random (but realistic) transformations such as image rotation and flipping the image around an axis. If the dataset in a machine learning model is rich and sufficient, the model performs better and more accurately. We will preprocess the training and testing set, but only the training set will undergo augmentation.

Go through the Pytorch torchvision.transforms.v2 documentation to see how to apply multiple transformations at once.

In the **cnn_image_transformations.py** file, complete the following functions to understand the common practices used for preprocessing and augmenting the image data:

- **create_training_transformations**
  - In this function, you are going to preprocess and augment training data.
    - PREPROCESS: Convert the given PIL Images to Tensors
    - AUGMENTATION: Apply Random Horizontal Flip and Random Rotation

- **create_testing_transformations**
  - In this function, you are going to only preprocess testing data.
    - PREPROCESS: Convert the given PIL Images to Tensors

Please note that the Gradescope only checks if expected preprocessing layers are existent.

References

v2.Compose()

v2.ToTensor() (Hint: Look at the warning)

v2.RandomHorizontalFlip()

v2.RandomApply()

v2.RandomRotation()

Article about performance regarding transformations

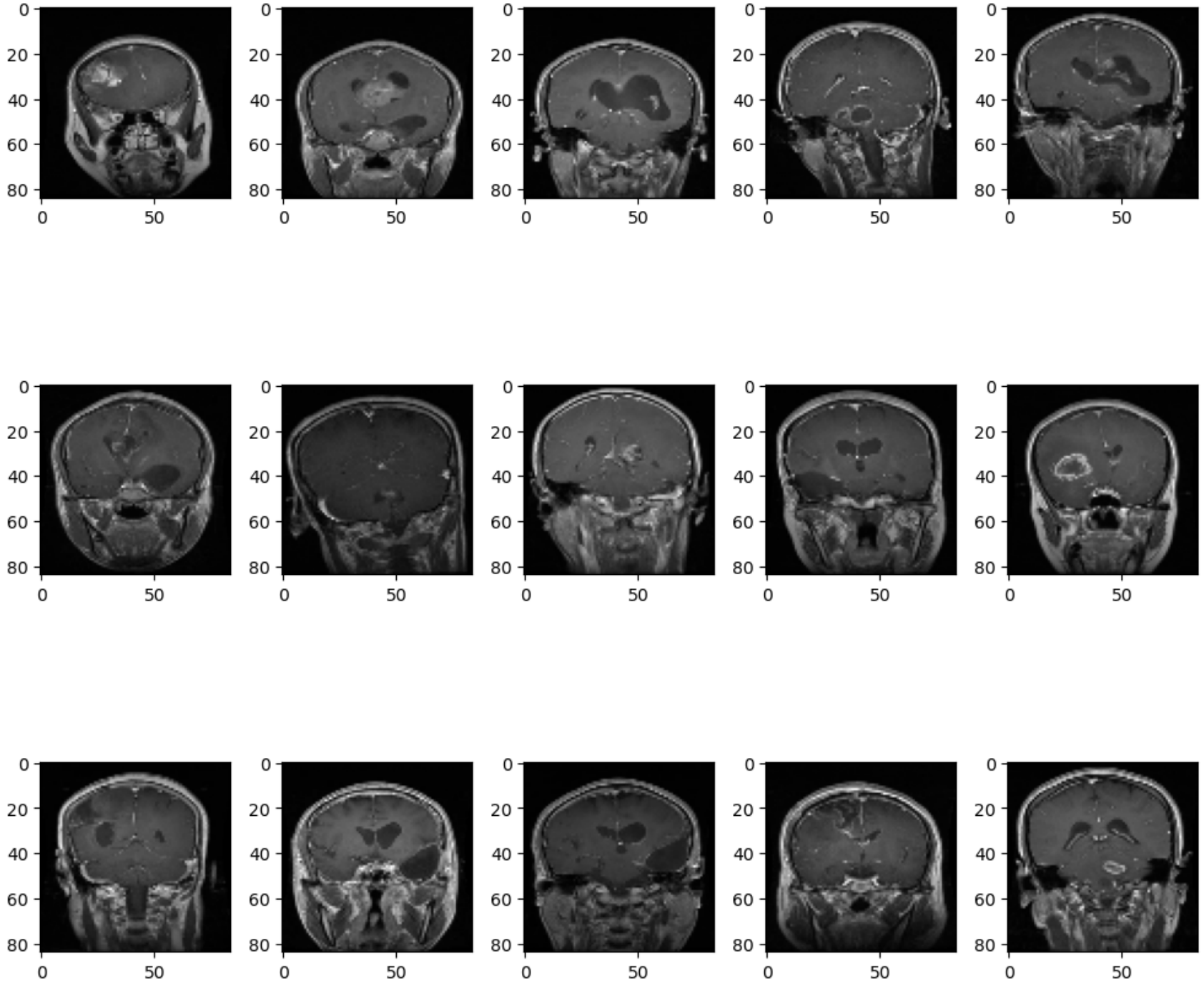
In [22]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from cnn_image_transformations import (
    TransformedDataset,
    create_testing_transformations,
    create_training_transformations,
)
from utilities.utils import get_mri_dataset

# Load data
classes = ["glioma", "meningioma", "notumor", "pituitary"]
x_train, y_train, x_test, y_test = get_mri_dataset(classes)

# Create Transformations
train_transform = create_training_transformations()
test_transform = create_testing_transformations()
```

```
# Transform data
trainset = TransformedDataset(x_train, y_train, transform=train_transform)
testset = TransformedDataset(x_test, y_test, transform=test_transform)

print(trainset.data.shape)
print(testset.data.shape)
```

```
Loading preprocessed data from disk...
torch.Size([5712, 1, 84, 84])
torch.Size([1311, 1, 84, 84])
```

Load some sample images from Brain Tumor MRI Dataset

In [23]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

import matplotlib.pyplot as plt

trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=32, shuffle=True, num_workers=2
)
testloader = torch.utils.data.DataLoader(
    testset, batch_size=32, shuffle=False, num_workers=2
)

# show sample images

images = [x_train[i] for i in range(15)]
fig, axes = plt.subplots(3, 5, figsize=(10, 10))
axes = axes.flatten()
for img, ax in zip(images, axes):
    ax.imshow(img, cmap="gray")
plt.tight_layout()
plt.show()
```


0
20
40
60
80
0
50

As you can see from above, the Brain Tumor MRI Dataset contains different types of brain MRI images. The images have been size-normalized and objects remain centered in fixed-size images.

2.1.2 Build convolutional neural network model

In this part, you need to build a convolutional neural network as described below. The architecture of the model is outlined.

In the **cnn.py** file, complete the following functions:

- **__init__**: See Defining Variables section
- **forward**: See Defining Model section

[INPUT - CONV - CONV - MAXPOOL - DROPOUT - CONV - CONV - MAXPOOL - DROPOUT - AVERAGEPOOL - FC1 - DROPOUT - FC2 - DROPOUT - FC3]

> INPUT: $[1 \times 84 \times 84]$ will hold the raw pixel values of the image, in this case, an image of width 84, height 84, and 1 RGB channel (grayscale). This layer should give 8 filters and have appropriate padding to maintain shape.

> CONV: Conv. layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to the input volume. In our example architecture, we decide to set the kernel_size to be 3×3. For example, the output of the Conv. layer may look like $[8 \times 84 \times 84]$ if we set out_channels to be 8 and use appropriate paddings to maintain shape.

> CONV: Additional Conv. layer take outputs from above layers and applies more filters. We set the kernel_size to be 3×3 and out_channels to be 32.

> MAXPOOL: MAXPOOL layer will perform a downsampling operation along the spatial dimensions (width, height). With pool size of 2×2, resulting shape takes form 16×16.

> DROPOUT: DROPOUT layer with the dropout rate of 0.2 to prevent overfitting.

> CONV: Additonal Conv. layer takes outputs from above layers and applies more filters. We set the kernel_size to be 3×3 and out_channels to be 32. Appropriate paddings are used to maintain shape.

> CONV: Additonal Conv. layer takes outputs from above layers and applies more filters. We set the kernel_size to be 3×3 and out_channels to be 64. Appropriate paddings are used to maintain shape.

> MAXPOOL: MAXPOOL layer will perform a downsampling operation along the spatial dimensions (width, height).

> DROPOUT: Dropout layer with the dropout rate of 0.2 to prevent overfitting.

> AVERAGEPOOL: AVERAGEPOOL layer will perform a downsampling operation along the spatial dimension (width, height). Checkout AdaptiveAvgPool2d below.

> FC1: Dense layer which takes output from above layers, and has 256 neurons. Flatten() operations may be useful.

> DROPOUT: Dropout layer with the dropout rate of 0.2 to prevent overfitting.

> FC2: Dense layer which takes output from above layers, and has 128 neurons.

> DROPOUT: Dropout layer with the dropout rate of 0.2 to prevent overfitting.

> FC3: Dense layer with 4 neurons, and Softmax activation, is the final layer. The dimension of the output space is the number of classes.

Activation function: Use LeakyReLU with negative_slope 0.01 as the activation function for Conv. layers and Dense layers unless otherwise indicated to build your model architecture

Note that while this is a suggested model design, you may use other architectures and experiment with different layers for better results.

The following links are Pytorch documentation for the layers you are going to use to build the CNN.

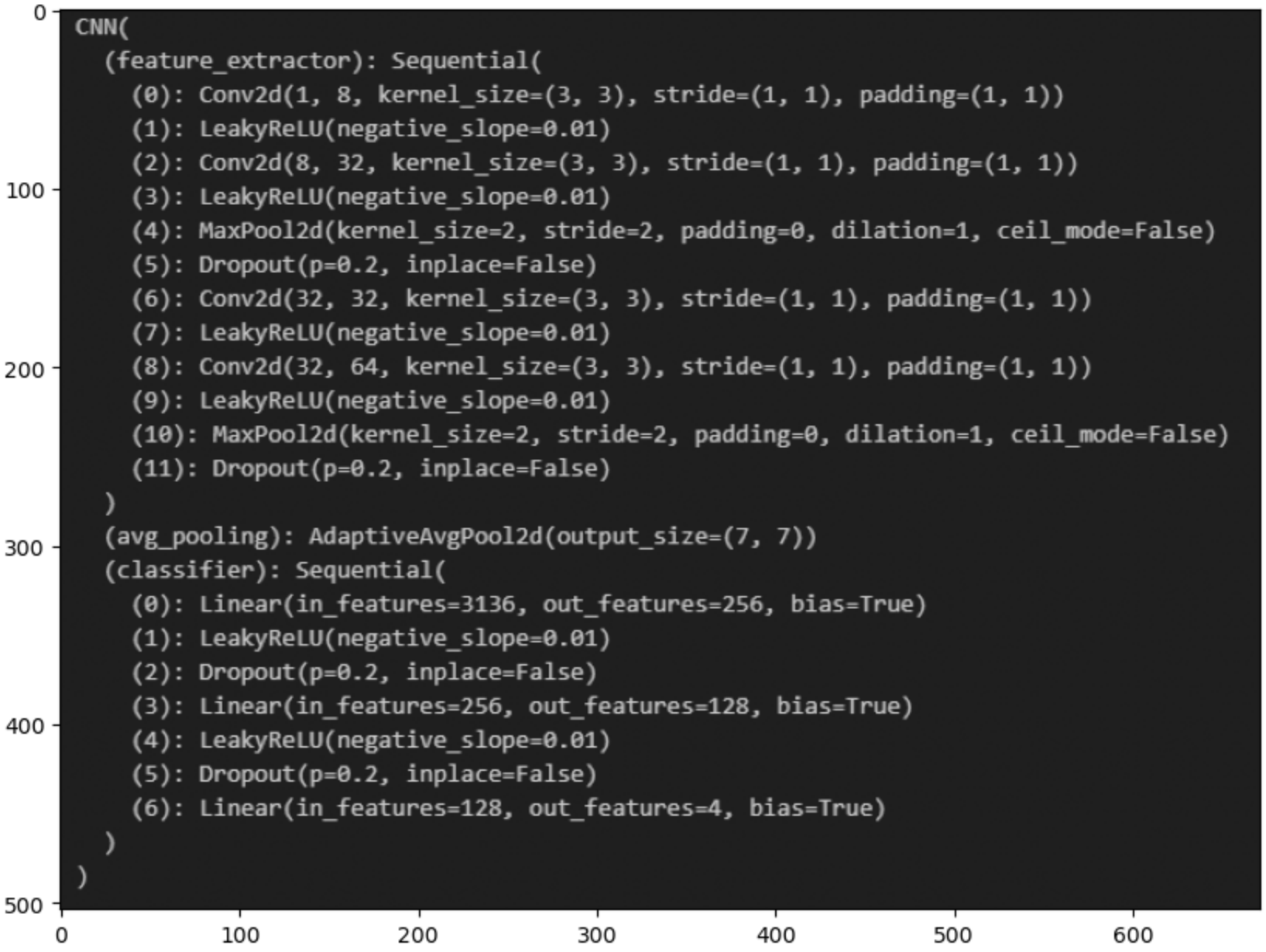
- Conv2d
- Dense
- MaxPool
- AdaptiveAvgPool2d
- Dropout
- LeakyReLU

- Flatten

Lastly, if you would like to experiment with additional layers, explore the torch.nn api.

In [24]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

# Show the architecture of the model
achi = plt.imread("./data/images/Architecture.png")
fig = plt.figure(figsize=(10, 10))
plt.imshow(achi)
```

Out[24]:
```
<matplotlib.image.AxesImage at 0x36c1f96d0>
```



Defining model

You now need to complete the `__init__()` function and the `forward()` function in **cnn.py** to define your model

structure.

Your model is required to have at least 2 convolutional layers and at least 2 dense layers. Ensuring that these requirements are met will earn you 5pts.

Once you have defined a model structure you may use the cell below to visually examine your architecture. However, there's no points for the notebook cell output because the architecture is tested on Gradescope.

In [25]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

# You can compare your architecture with the 'Architecture.png'

from cnn import CNN

net = CNN()
print(net)
```

```
CNN(
  (feature_extractor): Sequential(
    (0): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.01)
    (2): Conv2d(8, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): LeakyReLU(negative_slope=0.01)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Dropout(p=0.2, inplace=False)
    (6): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): LeakyReLU(negative_slope=0.01)
    (8): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): LeakyReLU(negative_slope=0.01)
    (10): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (11): Dropout(p=0.2, inplace=False)
  )
  (avg_pooling): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=3136, out_features=256, bias=True)
    (2): LeakyReLU(negative_slope=0.01)
    (3): Dropout(p=0.2, inplace=False)
    (4): Linear(in_features=256, out_features=128, bias=True)
    (5): LeakyReLU(negative_slope=0.01)
    (6): Dropout(p=0.2, inplace=False)
    (7): Linear(in_features=128, out_features=4, bias=True)
    (8): Softmax(dim=1)
  )
)
```

Local Test - Model Architecture

The test below tests if your model architecture is compatible with the input.

In [26]:
```
from utilities.localtests import TestCNN

TestCNN("test_model_architecture").test_model_architecture()
```

```
test_model_architecture passed!
```

2.1.3 Training and Tuning the Model [Bonus for Undergrad]

Training: You have to implement the function to be used in both the train step and val step of each epoch. Implement `run_epoch()` in **cnn_trainer.py**.

Hint 1: If you see any mismatch errors for torch tensors, your tensors are on different devices. Use `.to(self.device)` on each tensor to move the tensors to the same device.

Hint 2: The model, criterion/loss function, optimizer, and scheduler are all set in `train()`, so you don't need to make new ones yourself.

Hint 3: Think about what is different between the training loop and the evaluation loop. What needs to be done during training that is not done in evaluation?

The following links are to Pytorch documentation you may find helpful.

- optim
- Adamax
- CrossEntropyLoss
- ExponentialLR

Tuning: Next, we train and optimize the network. You can set the hyperparameters in `tune()` in **cnn_trainer.py**. Start with default values to verify your training loop works correctly, then experiment with different settings to improve accuracy.

If your hyperparameters are set properly, you should see the loss of the validation set decreased and the value of accuracy increased.

- Recommended Batch Sizes fall in the range 32-512 (use powers of 2)
- Recommended Epoch Counts fall in the range 5-10
- Recommended Learning Rates fall in the range .001-.01

Expected Result:

Your model's performance will be evaluated based on its highest validation accuracy across all epochs. The point distribution is as follows:

- Below 70% earns 2 pts (0.3% Bonus for Undergrad)
- 70% to 74.9% earns 2pts (4pts total, 0.6% Bonus for Undergrad)
- 75% to 79.9% earns 2pts more (6pts total, 0.9% Bonus for Undergrad)
- 80%+ earns 2pts more (8pts total, 1.2% Bonus for Undergrad)

Your training must take around 5 - 10 minutes, otherwise it will timeout on gradescope. For this reason, we highly recommend keeping number of epochs below 10 and optimizing other hyperparameters if you're unable to achieve optimal accuracy with 10 epochs. As a reference, it took us 5 minutes to achieve 80%+ accuracy on CPU.

Note: If you would like to automate the tuning process, you can use a nested for loop to search for the hyperparameter that achieves the accuracy. You could also look into grid search for hyperparameter optimization.

Train your own CNN model

In [ ]:
```
from cnn import CNN
from cnn_trainer import Trainer

net = CNN()

# Choose best device to speed up training
if torch.cuda.is_available():
    device = "cuda"
else:
    device = "cpu"
print(f"Using {device} device")

trainer = Trainer(
    net,
    trainset,
    testset,
    device=device,
)
trainer.tune()
```

2.1.4 Examine loss plots [Bonus for Undergrad]

To achieve full credit, your loss must be decreasing and accuracy must be increasing gradually.

In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

# list all data in history
train_loss, train_accuracy, val_loss, val_accuracy = trainer.get_training_history()

# summarize history for accuracy and loss
plt.plot(train_accuracy)
plt.plot(val_accuracy)
plt.title("model accuracy")
plt.ylabel("accuracy")
plt.xlabel("epoch")
plt.legend(["train", "val"], loc="upper left")
plt.show()

plt.plot(train_loss)
plt.plot(val_loss)
plt.title("model loss")
plt.ylabel("loss")
plt.xlabel("epoch")
plt.legend(["train", "val"], loc="upper left")
plt.show()
```

In [ ]:
```
from utilities.localtests import TestCNN

TestCNN("test_cnn_train_loss_plot").test_cnn_train_loss_plot(trainer)
TestCNN("test_cnn_test_loss_plot").test_cnn_test_loss_plot(trainer)
```

2.1.5 Examine Confusion Matrix [Bonus for Undergrad]

To get full credit, all the diagonal entries in your confusion matrix should be above 0.5. This ensures our model has reasonable accuracy across all classes. You can use the test below to verify the same.

In [ ]:
```
TestCNN("test_cnn_confusion_matrix").test_cnn_confusion_matrix(trainer, testloader)
```

Test accuracy is not tested on this homework but we include it here for completeness.

In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

y_pred, y_pred_classes, y_gt_classes = trainer.predict(testloader)
y_pred_prob = torch.max(y_pred, dim=1).values

from sklearn.metrics import ConfusionMatrixDisplay, accuracy_score

print(f"Test accuracy: {accuracy_score(y_gt_classes, y_pred_classes)}")
```

2.2 Exploring Deep CNN Architectures [1.5% Bonus for All] [W]

The network you have produced is rather simple relative to many of those used in industry and research. Researchers have worked to make CNN models deeper and deeper over the past years in an effort to gain higher accuracy in predictions. While your model is only a handful of layers deep, some state of the art deep architectures may include up to 150 layers. However, this process has not been without challenges.

One such problem is the problem of the exploding gradient. The initial weights assigned to the neural nets creating large losses. Big gradient values can accumulate to the point where large parameter updates are observed, causing gradient descents to oscillate without coming to global minima. What's even worse is that these parameters can be so large that they overflow and return NaN values that cannot be updated anymore.

Many tactics have been used in an effort to solve this problem. One architecture, named Gradient Clipping, solves the vanishing gradient problem in a unique way. Researchers from Massachusetts Institute of Technology dicussed about the mechanism and a theoretical explanation for the effectiveness of gradient clipping in training deep neural networks. Take a moment to explore how Gradient Clipping tackles the vanishing gradient problem by reading the original research paper here: https://arxiv.org/pdf/1905.11881 (also included as PDF in papers directory).

Question: In your own words, explain how Gradient Clipping addresses the exploding gradient problem in 1-2 sentences below: (Please type answers directly in the cell below.)

Answer: Gradient clipping caps the gradients during the backpropagation step, effectively preventing exploding gradient

problem. With gradient clipping, we can ensure that the parameters stay stable.

3: Random Forests [30pts + 3% Bonus for All] [P] [W]

NOTE: Please use sklearn's ExtraTreeClassifier in your Random Forest implementation. You can find more details about this classifier here.

For context, the general difference between an extra tree and decision tree classifier is that the decision tree optimizes which feature to reduce entropy on and at what value to split, while an extra tree randomly splits on the features given.

3.1 Random Forest Implementation [15pts] [P]

The decision boundaries drawn by decision or extra trees are very sharp, and fitting a tree of unbounded depth to a list of examples almost inevitably leads to **overfitting**. In an attempt to decrease the variance of an extra tree, we're going to use a technique called 'Bootstrap Aggregating' (often abbreviated 'bagging'). This stems from the idea that a collection of weak learners can learn decision boundaries as well as a strong learner. This is commonly called a Random Forest.

We can build a Random Forest as a collection of extra trees, as follows:

1. For every tree in the random forest, we're going to

   a) Subsample the examples with replacement. Note that in this question, the size of the subsample data is equal to the original dataset.

   b) From the subsamples in part a, choose attributes at random without replacement to learn on in accordance with a provided attribute subsampling rate. Based on what it was mentioned in the class, we randomly pick features in each split. We use a more general approach here to make the programming part easier. Let's randomly pick some features (65% percent of features) and grow the tree based on the pre-determined randomly selected features. Therefore, there is no need to find random features in each split.

   c) Fit an extra tree to the subsample of data we've chosen to a certain depth.

You can refresh your understanding with the lecture notes on random forests.

Classification for a random forest is then done by taking a majority vote of the classifications yielded by each tree in the forest after it classifies an example.

In the **random_forest.py** file, complete the following functions:

- **_bootstrapping**: this function will be used in `bootstrapping()`
- **fit**: Fit the extra trees initialized in `__init__` with the datasets created in `bootstrapping()`. You will need to call `bootstrapping()`. **NOTES:**

1. In the Random Forest Class, X is assumed to be a matrix with num_training rows and num_features columns where num_training is the number of total records and num_features is the number of features of each record. y is assumed to be a vector of labels of length num_training.
2. Look out for TODO's for the parts that need to be implemented
3. If you receive any `SettingWithCopyWarning` warnings from the Pandas library, you can safely ignore them.
4. Hint: when bootstrapping, set replace = False while creating col_idx

In [27]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
from utilities.localtests import TestRandomForest

"""
Once you have implemented Random forest, you can run this cell. If you implemented _bootStrapping correctly
then this cell should execute without any errors.
"""
TestRandomForest("test_bootstrapping").test_bootstrapping()
```

```
test_bootstrapping passed!
```

The `fit` method doesn't have any local tests available. When your code is submitted to Gradescope, we'll evaluate your `fit` method using optimal hyperparameters. To receive full credit on this assignment, your random forest implementation must achieve at least 85% accuracy on our test dataset. Please submit your code on gradescope to verify it is correct.

3.2 Hyperparameter Tuning with a Random Forest [15pts] [P]

In machine learning, hyperparameters are parameters that are set before the learning process begins. The max_depth,

num_estimators, or max_features variables from 3.1 are examples of different hyperparameters for a random forest model. Let's first review the dataset in a bit more detail.

Dataset Objective

Imagine that we are a team of researchers working to track and document various information related to dry beans for a machine learning model that predicts what type of bean is represented. We know that there are multiple things to keep track of, such as the shapes and sizes that differentiate different types of beans. We will use the information we track and document in order to publish it for the general public.

After much reflection within the research team, we come to the conclusion that we can use past observations on bean images to create a model.

We will use our random forest algorithm from Q3.1 to predict the bean type.

You can find more information on the dataset here.

The barbunya bean, also known as the cranberry bean, was first bred in Colombia.

Loading the dataset

The dataset that the company has collected has the following features:

There were 16 features used in this dataset.

Inputs:

1. Area: The area of a bean zone and the number of pixels within its boundaries
2. Perimeter: Bean circumference is defined as the length of its border
3. MajorAxisLength: The distance between the ends of the longest line that can be drawn from a bean
4. MinorAxisLength: The longest line that can be drawn from the bean while standing perpendicular to the main axis
5. AspectRatio: Defines the relationship between MajorAxisLength and MinorAxisLength
6. Eccentricity: Eccentricity of the ellipse having the same moments as the region
7. ConvexArea: Number of pixels in the smallest convex polygon that can contain the area of a bean seed
8. EquivDiameter Equivalent diameter, the diameter of a circle having the same area as a bean seed area
9. Extent Feature: The ratio of the pixels in the bounding box to the bean area
10. Solidity: Also known as convexity. The ratio of the pixels in the convex shell to those found in beans.
11. Roundness: Calculated with the following formula: (4piA)/(P^2)
12. Compactness: Measures the roundness of an object
13. ShapeFactor1
14. ShapeFactor2
15. ShapeFactor3
16. ShapeFactor4

Output:

17. Target value:
    - Seker
    - Barbunya
    - Bombay
    - Cali
    - Dermosan
    - Horoz
    - Sira

Your random forest model will try to predict this variable.

In [28]:
```
import numpy as np
import pandas as pd
```

```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
from sklearn import preprocessing

dry_bean_dataset = "./data/Dry_Bean_Dataset.csv"
df = pd.read_csv(dry_bean_dataset)

label_encoder = preprocessing.LabelEncoder()

X = df.drop(["Class"], axis=1)
y = label_encoder.fit_transform(df["Class"])

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=42
)
X_test = np.array(X_test)
X_train, y_train, X_test, y_test = (
    np.array(X_train),
    np.array(y_train),
    np.array(X_test),
    np.array(y_test),
)
```

In [29]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
assert X_train.shape == (9119, 16)
assert y_train.shape == (9119,)
assert X_test.shape == (4492, 16)
assert y_test.shape == (4492,)
```

```
(9119, 16) (9119,) (4492, 16) (4492,)
```

3.2.1 Grid Search [5pts]

When defining our random forest models, there are three main hyperparameters that we can adjust:

- **n_estimators**: Defines the number of trees in the random forest.
- **max_depth**: Sets a limit on how deep each tree can grow.

- **max_features**: Determines the fraction of features used to train each tree.

Tuning hyperparameters manually can be time-consuming, so we use Grid Search to automate the process. Grid Search exhaustively tests different combinations of hyperparameter values to find the best one based on a chosen metric (e.g., accuracy, F1 score, etc.). This is generally very computationally expensive. For instance, if there are `n` hyperparameters, and `m` possibilities for each one, then the train loop must be executed `m^n` times to exhaustively test each combination. For large models, this is prohibitively expensive. For smaller models, however, grid search can deliver optimal results without requiring manual hyperparameter tuning in a reasonable amount of time.

In the **random_forest.py** file, complete the following function:

- **hyperparameter_grid_search**: This function generates all possible combinations of the given hyperparameter values based on the ranges defined in **random_forest.py**. It does not perform the actual grid search but provides the set of hyperparameter permutations that will be tested in the next code block. The resulting array of hyperparameter combinations will be used to train the random forest model. If implemented correctly, this cell should run without errors, and the trained model with our selected hyperparameter ranges should achieve >85% accuracy.

In [33]:
```
import sklearn.ensemble
from random_forest import RandomForest
from sklearn import preprocessing

################ DO NOT CHANGE THIS RANDOM SEED ###################
student_random_seed = 4641 + 7641
###################################################################

TestRandomForest("test_hyperparameter_grid_search").test_hyperparameter_grid_search()
```

```
test_hyperparameter_search passed!
```

3.2.2 Finding optimal random forest [5pts]

You will now find the `best_random_forest` using our grid search method. To get full credit on gradescope, your best random forest must attain an accuracy of 85% or higher.

In [34]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
```

```
random_forest = RandomForest(0, 0, None)

n_estimators_range = (8, 10, 1)
max_depth_range = (8, 10, 1)
max_features_range = (0.7, 1.0, 0.1)

permutations = random_forest.hyperparameter_grid_search(
    n_estimators_range, max_depth_range, max_features_range
)

best_random_forest = None
best_accuracy = 0
count = 0
for n, (n_estimators, max_depth, max_features) in enumerate(permutations):
    random_forest = RandomForest(
        n_estimators, max_depth, max_features, random_seed=student_random_seed
    )
    random_forest.fit(X_train, y_train)
    accuracy = random_forest.OOB_score(X_test, y_test)
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_random_forest = random_forest
    if n % 10 == 0:
        print("Checked {}/{} combinations".format(n, len(permutations)))

print("accuracy: %.4f" % best_accuracy)
```

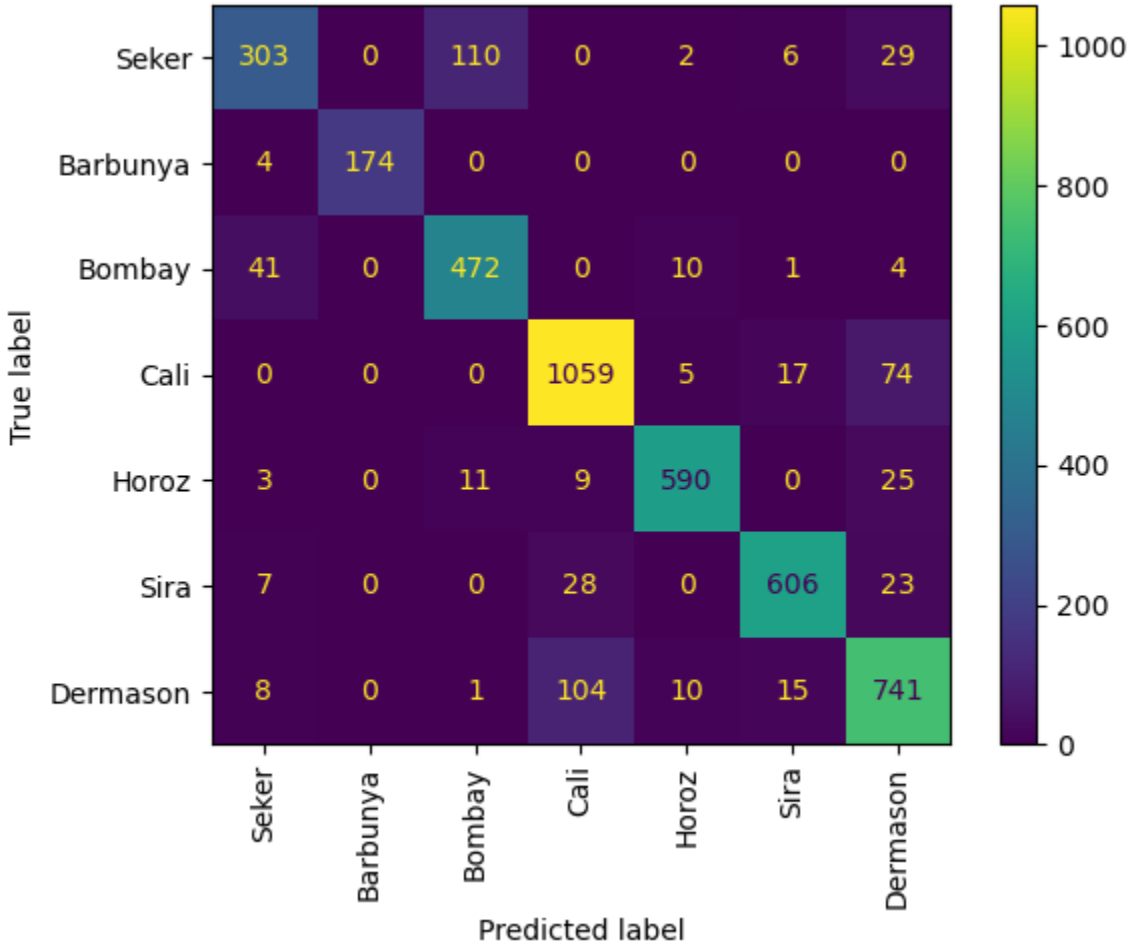
```
Checked 0/36 combinations
Checked 10/36 combinations
Checked 20/36 combinations
Checked 30/36 combinations
accuracy: 0.8705
```

3.2.3 Examining Confusion Matrix [5pts]

To ensure our model is not biased towards one class, it is helpful to look at the performance per class. To acheive full credit, all the diagonal entries in your confusion matrix must be >= 150

In [35]:
```
# Results on Test Set
from sklearn.metrics import ConfusionMatrixDisplay
```

```
pred = best_random_forest.predict(X_test)
labels = ["Seker", "Barbunya", "Bombay", "Cali", "Horoz", "Sira", "Dermason"]
ConfusionMatrixDisplay.from_predictions(
    y_test, pred, display_labels=labels, xticks_rotation="vertical"
)
plt.show()
```



3.3 Plotting Feature Importance [1% Bonus for All] [W]

While building tree-based models, it's common to quantify how well splitting on a particular feature in an extra tree helps with predicting the target label in a dataset. Machine learning practitioners typically use "Gini importance", or the (normalized)

total reduction in entropy brought by that feature to evaluate how important that feature is for predicting the target variable.

Gini importance is typically calculated as the reduction in entropy from reaching a split in an extra tree weighted by the probability of reaching that split in the extra tree. Sklearn internally computes the probability for reaching a split by finding the total number of samples that reaches it during the training phase divided by the total number of samples in the dataset. This weighted value is our feature importance.

Let's think about what this metric means with an example. A high probability of reaching a split on feature A in an extra tree trained on a dataset (many samples will reach this split for a decision) and a large reduction in entropy from splitting on feature A will result in a high feature importance value for feature A. This could mean feature A is a very important feature for predicting the probability of the target label. On the other hand, a low probability of reaching a split on feature B in an extra tree and a low reduction in entropy from splitting on feature B will result in a low feature importance value. This could mean feature B is not a very informative feature for predicting the target label. **Thus, the higher the feature importance value, the more important the feature is to predicting the target label.**

Fortunately for us, fitting a sklearn.ExtraTreeClassifier to a dataset automatically computes the Gini importance for every feature in the extra tree and stores these values in a **feature_importances_** variable. Review the docs for more details on how to access this variable

In the **random_forest.py** file, complete the following function:

- **plot_feature_importance**: Make sure to sort the bars in descending order and remove any features with feature importance of 0

In the cell below, call your implementation of `plot_feature_importance()` and display a bar plot that shows the feature importance values for at least one extra tree in your tuned random forest from Q3.2.

In [ ]:
```
# TODO: Complete plot_feature_importance() in random_forest.py

random_forest.plot_feature_importance(X)
```

Note that there isn't one "correct" answer here. We simply want you to investigate how different features in your random forest contribute to predicting the target variable.

Also note that: the number of features can be different if you change max_features value since it ends up changing the

number of features considered in bootstrapped datasets.

3.4 ADABoost [2% Bonus for All] [P]

In lecture we learn how to implement bootstrapping, but there is another common method used to prevent overfitting which also incorporates multiple decision trees: boosting. For our implementation, boosting is where you assign importances to multiple models (weak learners) and return the result of their averaged output (a single strong learner). This is done sequentially, with importances being updated after the training of each new tree rather than in parallel like in bootstrapping

Specifically, you will be implementing adaptive boosting (ADABoost) which reassigns importances to the randomized decision trees depending on their error weight.

Additional Resource: For a detailed walkthrough of ADABoost implementation, see Implementing the AdaBoost Algorithm from Scratch

Using their error weights you will recalculate the importances AKA `alpha` values. You can use this formula to do it per model:

$$\alpha = \frac{1}{2}\log\left(\frac{1-\text{error}}{\text{error}+10^{-10}}\right)$$

Then you must update the error weights for all models for the next importance calculation by multiplying all weights using this formula:

$$\text{Updated Weights (Not Normalized)} = \text{Weights} \times e^{\alpha \times (\text{Indicator of Incorrect Classifications})}$$

Here, the Indicator of Incorrect Classifications is a binary value (0 or 1) that indicates whether a specific instance was classified incorrectly by the current model. This ensures that only the weights of incorrectly classified instances are updated.

We give you predict_adaboost which returns the prediction from the result of all the trees. You must implement `adaboost()` which iterates over the number of estimators, training a new tree and then updating all trees' weights depending on its error rate.

In [ ]:
```
# TODO: Complete adaboost() in random_forest.py
```

```
TestRandomForest("test_adaboost").test_adaboost()
```

Now we can test the accuracy of our model using the helper function `predict_adaboost()` ! To pass Gradescope your accuracy should be above 85%!

In [ ]:
```
adaboost_model = RandomForest(
    n_estimators, max_depth, max_features, random_seed=student_random_seed
)

# Train the AdaBoost ensemble using adaboost method
adaboost_model.adaboost(X_train, y_train)

# Evaluate the accuracy on the test set
y_pred = adaboost_model.predict_adaboost(X_test)
accuracy = np.mean(y_pred == y_test)
print("AdaBoost accuracy: %.4f" % accuracy)
```

4: SVM [15 pts] [W]

Consider a dataset with the following points in two-dimensional space:

x_1	x_2	y
-1.5	0.1	1
-2.3	-1.5	1
1.9	0.75	1
0.5	2.25	1
-2.1	5.75	-1
-1.0	6.0	-1
1.4	6.5	-1
0.1	6.0	-1

Here, x_1 and x_2 are features and y is the label.

Support Vector Machines (SVMs) aim to find a hyperplane that separates data points of different classes with the maximum margin. The larger the margin, the better the model can generalize to unseen data. Fortunately, scikit-learn's SVC class handles this computation for us programmatically.

4.1 Fitting an SVM classifier [10 pts] [W]

4.1.1 Fit the SVM Classifier [7 pts]

Since the points are already linearly separable, determine the separating hyperplane using a linear SVM programatically. Record the weights (theta) and bias (intercept) terms for this separating hyperplane, rounded to three decimal places.

Hint: To do this, you'll need to import the `SVC` class from the `sklearn.svm` module and initialize the SVC with a linear kernel. Then you can fit the data and find the separating hyperplane.

In [ ]:
```
from sklearn.svm import SVC

x_1 = np.array([-1.5, -2.3, 1.9, 0.5, -2.1, -1.0, 1.4, 0.1])
x_2 = np.array([0.1, -1.5, 0.75, 2.25, 5.75, 6.0, 6.5, 6.0])
y = np.array([1, 1, 1, 1, -1, -1, -1, -1])

################################################################################
# TODO: Create and fit an instance of the SVC class.                           #
################################################################################
# https://numpy.org/doc/stable/reference/generated/numpy.column_stack.html
x = np.column_stack((x_1, x_2))
clf = SVC(kernel='linear')
clf.fit(x, y)

w = clf.coef_[0]
b = clf.intercept_[0]
slope = -w[0] / w[1]
intercept = -b / w[1]

print(round(slope, 3))
print(round(intercept, 3))
```

```
################################################################################
#                                END OF YOUR CODE                              #
################################################################################
```

```
0.114
4.091
```

Record the slope and intercept of the separating hyperplane obtained by fitting an instance of SVC on the given data

Therefore, slope = 0.114 and intercept = 4.091

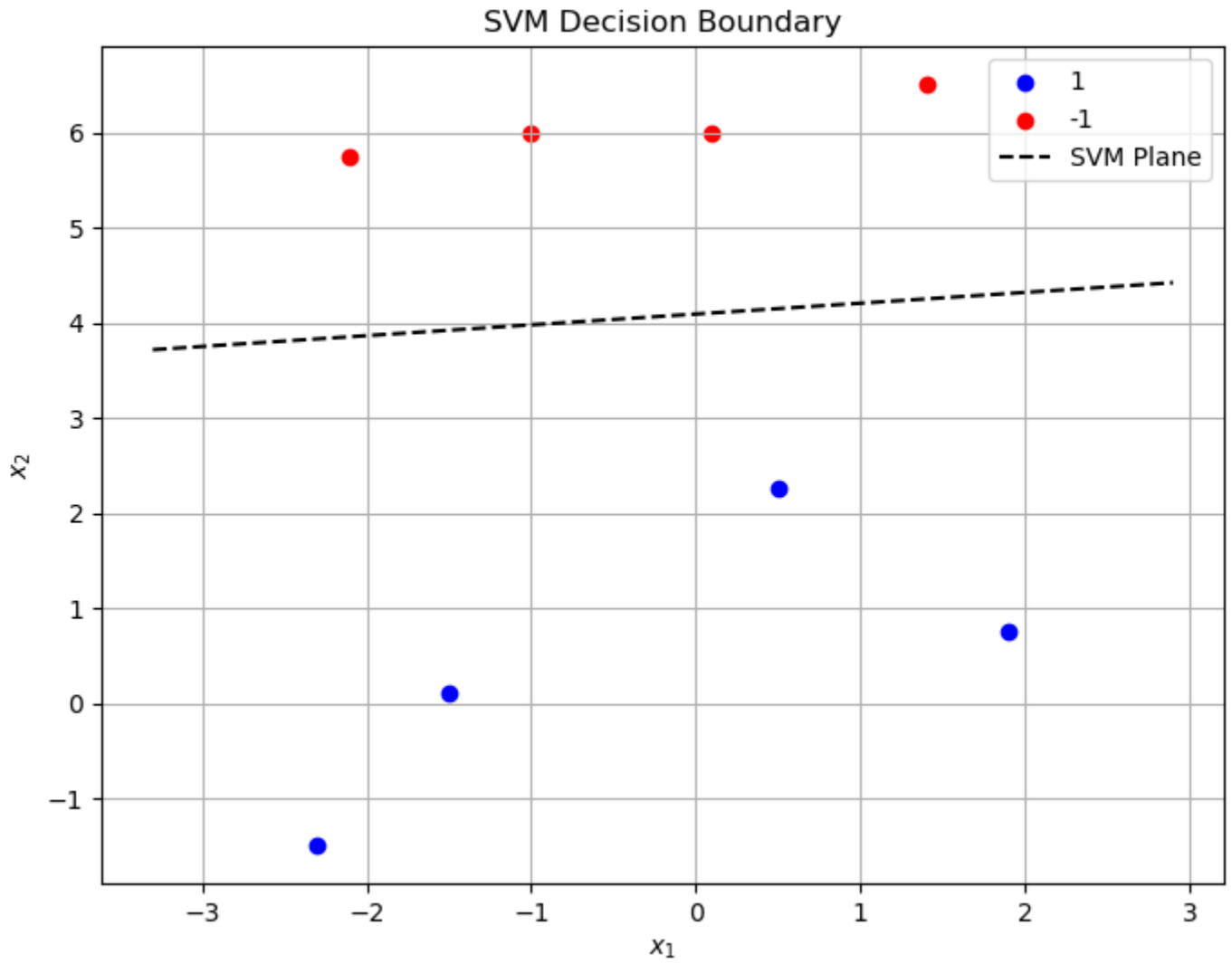
4.1.2 Plot the SVM Classifier [3 pts]

Plot the features x_1 and x_2 using different colors to represent the labels y (e.g., red for -1, blue for 1). Include the separating hyperplane on the plot.

Make sure your plot clearly distinguishes the two classes and visually demonstrates the hyperplane. Make sure to include the resulting plot output in your gradescope submission; the plot will be **handgraded**.

In [ ]:
```
import matplotlib.pyplot as plt

################################################################################
# TODO: Use matplotlib to plot data points and separating hyperplane           #
################################################################################
plt.figure(figsize=(8, 6))
plt.scatter(x[y == 1][:, 0], x[y == 1][:, 1], color='blue', label='1')
plt.scatter(x[y == -1][:, 0], x[y == -1][:, 1], color='red', label='-1')
# https://numpy.org/doc/stable/reference/generated/numpy.linspace.html
x_vals = np.linspace(min(x_1) - 1, max(x_1) + 1, 100)
y_vals = slope * x_vals + intercept

plt.plot(x_vals, y_vals, 'k--', label='SVM Plane')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('SVM Decision Boundary')
plt.legend()
plt.grid(True)
plt.show()
```

```
##############################################################################
#                             END OF YOUR CODE                               #
##############################################################################
```



4.2 Using Kernels [5 pts] [W]

Now, suppose we add another point, (0, -10) with label -1. This makes the dataset non-linearly separable, meaning no

straight-line hyperplane can perfectly classify all points.

To address this, you will **manually define a transformation function** that maps the original features to a new space where a linear SVM can successfully classify all points.

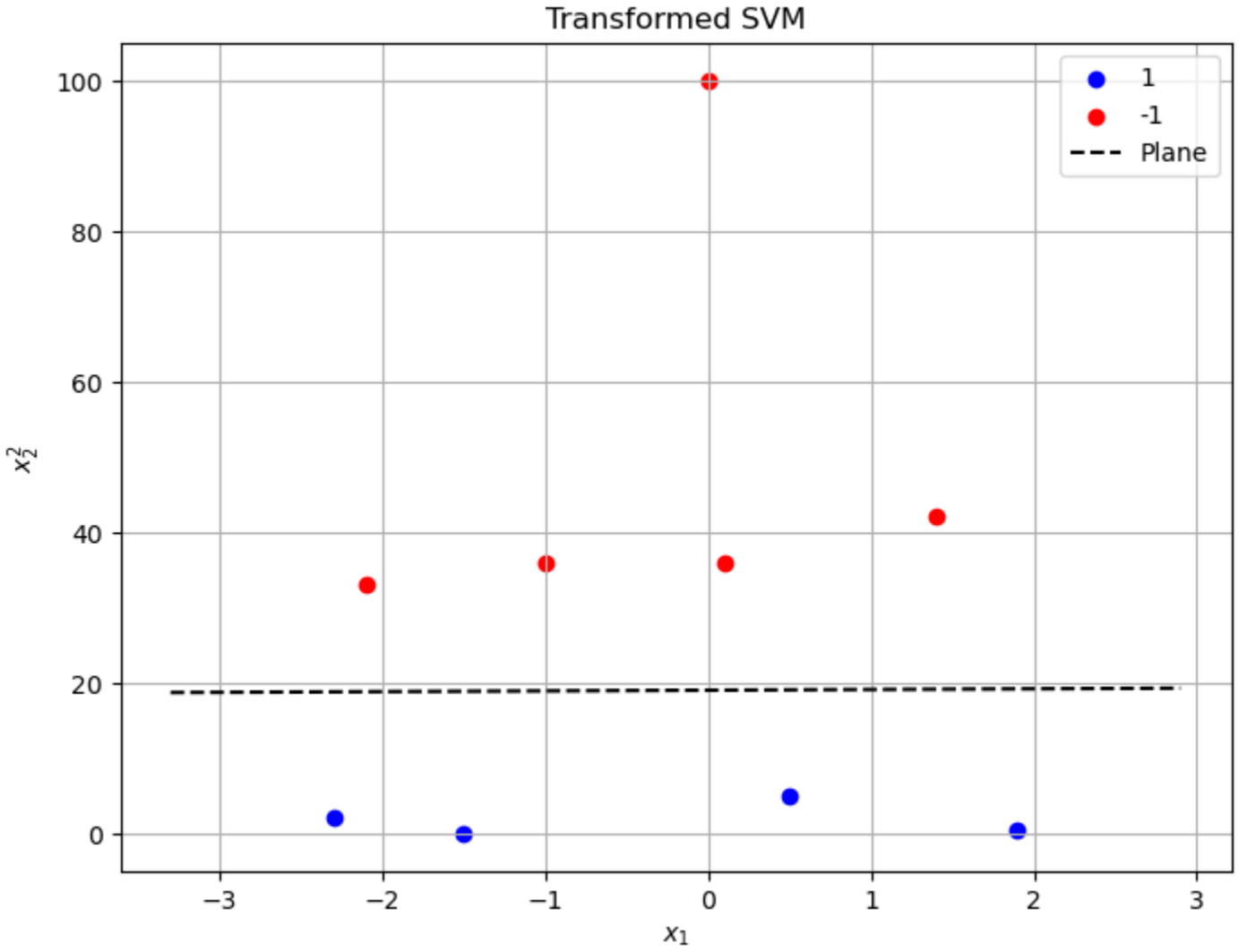
Important: You should still use `SVC` with a linear kernel —`the transformation should be applied **before** passing the data to SVC`. Your task is to write a function that transforms the dataset in a way that makes it linearly separable.

After applying your transformation, train an SVM with a **linear kernel** on the transformed data and determine the new separating hyperplane.

Make sure your plot clearly distinguishes the two classes and visually demonstrates the hyperplane. Make sure to include the resulting plot output in your gradescope submission; the plot will be **handgraded**.

In [62]:
```
from sklearn.svm import SVC

x_1 = np.array([-1.5, -2.3, 1.9, 0.5, -2.1, -1.0, 1.4, 0.1, 0])
x_2 = np.array([0.1, -1.5, 0.75, 2.25, 5.75, 6.0, 6.5, 6.0, -10])
y = np.array([1, 1, 1, 1, -1, -1, -1, -1, -1])

################################################################################
# TODO: Transform the data so it's linearly separable, then plot the new hyperplane   #
################################################################################
z = x_2 ** 2
x_transformed = np.column_stack((x_1, z))
clf = SVC(kernel='linear')
clf.fit(x_transformed, y)
w = clf.coef_[0]
b = clf.intercept_[0]

plt.figure(figsize=(8, 6))
plt.scatter(x_1[y == 1], z[y == 1], color='blue', label='1')
plt.scatter(x_1[y == -1], z[y == -1], color='red', label='-1')

x1_vals = np.linspace(min(x_1) - 1, max(x_1) + 1, 100)
z_vals = -(w[0] * x1_vals + b) / w[1]
plt.plot(x1_vals, z_vals, 'k--', label='Plane')
```

```
plt.xlabel('$x_1$')
plt.ylabel('$x_2^2$')
plt.title('Transformed SVM')
plt.legend()
plt.grid(True)
plt.show()
##########################################################################################
#                                    END OF YOUR CODE                                    #
##########################################################################################
```



Describe your chosen transformation function below and why it works (one brief sentence is fine)

Answer: I applied transformation z=x_2^2 to map data to higher dimension in order to make it linearly separable.

5: Next Character Prediction using Recurrent Neural Networks (RNNs) [5.5% Bonus for All] [W] [P]

Recurrent Neural Networks are a class of neural networks designed to handle sequential or time-series data, where the order of inputs matters. Unlike feedforward neural networks that treat each input independently, sequential networks maintain memory of previous inputs, making them ideal for tasks involving ordered data like text, time series, or video frames. These networks allow previous outputs to be used as inputs while having hidden states.

Common applications include:

- Text processing (language modeling, translation)
- Machine translation (translating from one language to the other)
- Time series prediction (stock prices, weather forecasting)

In this section, we'll compare two foundational types of recurrent neural network architectures: Simple Recurrent Neural Networks (Simple RNNs) and Long Short-Term Memory networks (LSTMs). The goal is to train these models to generate text in the style of Macbeth by predicting the next character in a given sequence. This exercise will highlight how each architecture manages sequential dependencies in text generation.

Check out the guide under `utilities/q5_guide` for more details on RNNs.

Data Preparation

- We'll use Shakespeare's Macbeth from Project Gutenberg
- We vectorize the text by treating every character in our text as an individual unit (e.g., 'macbeth' -> ['m', 'a', 'c'...])
- We use a dictionary to store this mapping: {'a':1, 'b':2, 'c':3, ...}
- This mapping enables bidirectional conversion between characters and integers for model input and output interpretation
- We assign each each character a learnable embedding vector

- Create fixed-sized batches of characters using sliding window approach
- For example, with text "macbeth" (context window=4):
  `Window 1: "macb" → predict "e"`
  `Window 2: "acbe " → predict "t"`
  `Window 3: "cbet" → predict "h"`

Our final preprocessed data contains:

- **X**: Input sequences
  - (shape: [`NUM_SEQUENCES` , `SEQUENCE_LEN`])
  - Contains all character sequences of length SEQUENCE_LEN
- **Y**: Target characters
  - (shape: [`NUM_SEQUENCES` , `1`])
  - Contains the next character that follows each sequence in X
- **VOCABULARY MAP**: The mapping from all unique characters in the text and their numerical representations
- **VOCAB_SIZE**: Total number of unique characters
- **SEQUENCE_LEN**: Length of input sequences

You can also refer to **preprocess_text_data** located in utilities>utils.py for more details.

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from utilities.utils import preprocess_text_data

# load and preprocess text
text = requests.get("https://www.gutenberg.org/files/1533/1533-0.txt").text
DATA = preprocess_text_data(text)

# unpack processed data components
X, Y, TEXT, CHAR_INDICES, INDICES_CHAR, VOCAB, VOCAB_SIZE, SEQUENCE_LEN = (
    DATA["x"],
    DATA["y"],
    DATA["text"],
    DATA["char_indices"],
    DATA["indices_char"],
    DATA["vocab"],
```

```
    DATA["vocab_size"],
    DATA["sequence_len"],
)

print("Length of Corpus: ", len(TEXT))
print("Vocabulary Map: ", CHAR_INDICES)
print(f"Vocabulary Size: ", VOCAB_SIZE)
print(f"X shape: {X.shape}")
print(f"Y shape: {Y.shape}")
```

5.1 Model Architecture [3% Bonus for All] [P]

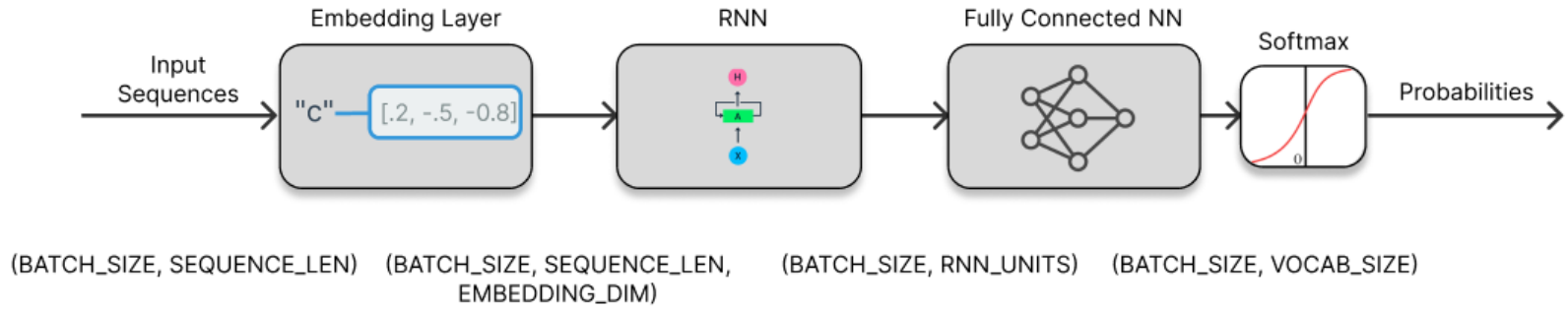
Before diving into the specific architectures, let's understand how data shapes transform through the embedding layer.

Input Sequence Shape Flow:

1. Initial input: (BATCH_SIZE, SEQUENCE_LEN)
    - BATCH_SIZE sequences containing SEQUENCE_LEN integers, where each integer represents a character from our vocabulary
    - Example: If BATCH_SIZE=32 and SEQUENCE_LEN=15, shape is (32, 15)
2. Embedding Layer: (BATCH_SIZE, SEQUENCE_LEN, EMBEDDING_DIM)
    - Transforms each integer into a vector of size EMBEDDING_DIM
    - Example: If EMBEDDING_DIM=64:
        - Each character index becomes a vector of 64 numbers
        - Shape expands from (32, 15) to (32, 15, 64)
        - This means: 32 sequences, each 15 characters long, each character now represented by 64 numbers

5.1.1 Defining the Simple RNN Model [1.5% Bonus for All]

In this part, you need to build a simple recurrent neural network using tensorflow. The architecture of the model is outlined below:



[EMBEDDING - Simple RNN - FC - ACTIVATION]

> **EMBEDDING**: The Embedding layer maps each integer (representing a character) in the input sequence to a dense vector representation. Each character index becomes a vector of **embedding dimension**. It has an input dimension of **vocab_size** (the total number of unique characters or tokens) and an output dimension defined by **embedding_dim**. This transformation allows the model to capture semantic relationships in the data.
>
> - Input shape: (batch_size, sequence_length) - A sequence of character indices
> - Output shape: (batch_size, sequence_length, embedding_dim) - Each character transformed into an embedding vector

> **Simple RNN**: This layer processes the sequence data, passing information through time steps to learn temporal patterns. It has **rnn_units** neurons, determining the model's ability to capture dependencies in the sequential data.
>
> - Input shape: (batch_size, sequence_length, embedding_dim) - Sequence of embedding vectors
> - Output shape: (batch_size, rnn_units) - Final state output

> **FC (Dense Layer)**: A fully connected layer that transforms the RNN output to match the number of classes or possible output tokens. It has **vocab_size** neurons, ensuring that each output corresponds to a unique token or class.
>
> - Input shape: (batch_size, rnn_units) - RNN final state
> - Output shape: (batch_size, vocab_size) - Raw scores for each possible character

> **ACTIVATION (Softmax)**: The softmax activation function is applied to the output layer, producing a probability distribution over the vocabulary, allowing the model to interpret each output as a confidence level for each class.
>
> - Input shape: (batch_size, vocab_size) - Raw scores
> - Output shape: (batch_size, vocab_size) - Probability distribution over characters

You can refer to the following documentation on tensorflow layers for more details:

- Embedding
- Simple RNN
- Dense
- Activation

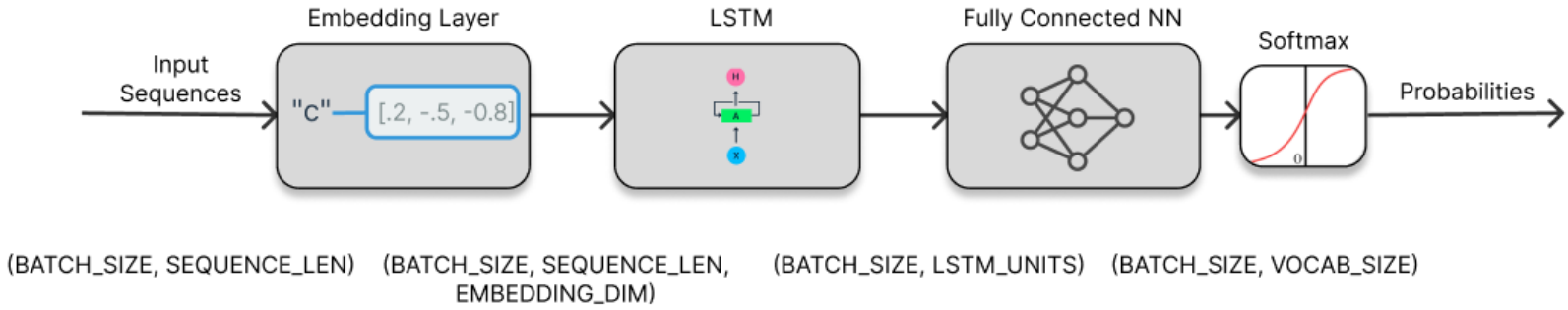
TODO: Implement the **define_model** function in **rnn.py**.

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from rnn import RNN

rnn_model = RNN(VOCAB_SIZE, SEQUENCE_LEN)
rnn_model.set_hyperparameters()
rnn_model.define_model()
rnn_model.build_model()
rnn_model.model.summary()
```

In [ ]:
```
from utilities.localtests import TestRNN

tester = TestRNN("test_rnn_architecture").test_rnn_architecture()
```

5.1.2 Defining the LSTM Model [1.5% Bonus for All]

In this part, you need to build a long short-term memory (LSTM) network as described below. The architecture of the model is outlined below:



[EMBEDDING - LSTM - FC - ACTIVATION]

> **EMBEDDING**: The Embedding layer maps each integer in the input sequence to a dense vector representation. It has an input dimension of **vocab_size** (the total number of unique characters or tokens) and an output dimension defined by **embedding_dim**. This transformation allows the model to capture semantic relationships in the data.
>
> - Input shape: (batch_size, sequence_length) - A sequence of character indices
> - Output shape: (batch_size, sequence_length, embedding_dim) - Each character transformed into an embedding vector

> **LSTM**: This layer processes the sequence data, passing information through time steps to learn temporal patterns. It has **lstm_units** neurons, determining the LSTM ability to capture dependencies in the sequential data.
>
> - Input shape: (batch_size, sequence_length, embedding_dim) - Sequence of embedding vectors
> - Output shape: (batch_size, lstm_units) - Final state output

> **FC (Dense Layer)**: A fully connected layer that transforms the LSTM output to match the number of classes or possible output tokens. It has **vocab_size** neurons, ensuring that each output corresponds to a unique token or class.
>
> - Input shape: (batch_size, lstm_units) - LSTM final state
> - Output shape: (batch_size, vocab_size) - Raw scores for each possible character

> **ACTIVATION (Softmax)**: The softmax activation function is applied to the output layer, producing a probability distribution over the vocabulary, allowing the model to interpret each output as a confidence level for each class.
>
> - Input shape: (batch_size, vocab_size) - Raw scores
> - Output shape: (batch_size, vocab_size) - Probability distribution over characters

You can refer to the following documentation on tensorflow layers for more details:

- Embedding
- LSTM
- Dense
- Activation

TODO: Implement the **define_model** function in **lstm.py**.

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from lstm import LSTM

lstm_model = LSTM(VOCAB_SIZE, SEQUENCE_LEN)
lstm_model.set_hyperparameters()
lstm_model.define_model()
lstm_model.build_model()
lstm_model.model.summary()
```

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from utilities.localtests import TestLSTM

tester = TestLSTM("test_lstm_architecture").test_lstm_architecture()
```

5.2 RNN vs LSTM Model Text Generation Training Comparison Analysis [2.5% Bonus for All] [W]

Training Configuration

- **Optimizer**: RMSprop (Root Mean Square Propagation) optimizer for efficient training of recurrent networks. Read more about it here.
- **Loss Function**: Categorical crossentropy to measure accuracy of predicted probability distribution. Read more about it here.
- **Training Parameters**: Number of epochs and batch size are defined in the hyperparameters section of `rnn.py` and `lstm.py` respectively

The core training loop implementation can be found in `base_sequential_model.py` . A plot showing loss metrics across epochs will be generated after training completes.

NOTE: The initial model training may take 10-15 minutes per model. After that, the function will automatically load the saved weights stored in the `rnn_model_weights` directory, making subsequent runs much faster. If you want to retrain from scratch instead of using saved weights, you can either:

- Set `train_from_scratch=True` in the parameters
- Delete the existing weights from the `rnn_model_weights` directory

In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

rnn_model.train(x=X, y=Y, train_from_scratch=False)
rnn_model.plot_loss()
```

In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

lstm_model.train(x=X, y=Y, train_from_scratch=False)
lstm_model.plot_loss()
```

We can now generate text in the style of Macbeth by recursively:

- Sampling next character using model's predicted probabilities
- Including the generated character in current window, and generate the next character

You can refer to text_generator.py for more details.

In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

from text_generator import TextGenerator

generator = TextGenerator(CHAR_INDICES, INDICES_CHAR, SEQUENCE_LEN)
```

In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

start_index = random.randint(0, len(TEXT) - SEQUENCE_LEN - 1)
seed_text = TEXT[start_index : start_index + SEQUENCE_LEN]

generator.generate(model=rnn_model, seed_text=seed_text, length=150, temperature=0.75)
generator.generate(model=lstm_model, seed_text=seed_text, length=150, temperature=0.75)
```

Written Question

Analyze your experience training Simple RNN and LSTM models for Macbeth character prediction and answer the following:

1. Identify and explain a real-world application where Simple RNN would be more suitable than LSTM
2. Identify and explain a real-world application where LSTM would be more suitable than Simple RNN

For each case, support your answer using evidence from atleast 1 metric observed during training and 1 other metric. For the observed metric, please include specific evidence from the training in the notebook.

Some ideas for metrics you can consider:

- Inference time / Training time

- Final Loss Achieved
- Generated Text Quality
- Memory requirements
- Loss convergence
- Simplicity of architecture

YOUR ANSWER HERE

Carbon Impact

Running this notebook generates carbon emissions that we can measure. For context, a typical passenger vehicle emits approximately 200 grams of CO_2 for each kilometer driven. Below, we compare our computational carbon cost to this everyday activity.

In [ ]:
```
emissions = tracker.stop()

car_emissions_per_km = 0.2
students = 700
equivalent_distance_per_student_m = emissions / car_emissions_per_km * 1000
equivalent_distance_all = emissions * students / car_emissions_per_km

print(f"Total Emissions for one run: {emissions} kg CO2eq")
print(
    f"\nDriving Equivalent per student: {equivalent_distance_per_student_m:.2f} meters driven in an average
)
print(
    f"\nDriving Equivalent for all {students} students: {equivalent_distance_all:.4f} km driven in an avera
)
```