Fall 2024 CS 4641/7641 A: Machine Learning Homework 4

Instructor: Dr. Mahdi Roozbahani

Deadline: Monday, Dec 2nd, 2024 11:59 pm EST

For Homework 4, the December 2nd 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. See this EdStem Post for multiple ways on to convert your .ipynb into a .pdf file.** 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 [80 pts: 55pts + 25pts Grad / 3.3% Undergrad Bonus]

Deliverables: NN.py

- **1.1 NN Implementation** [65pts: 50pts + 15pts Grad / 2% **Bonus for Undergrad**] - *programming*
  - SiLU [5pts]
  - Softmax [5pts]
  - Cross Entropy loss [5pts]
  - dropout [5pts]
  - forward propagation and with and without dropout [5pts + 5pts]
  - compute gradients and update weights [2.5pts + 2.5pts]

  - backward without momentum [5pt]
  - Gradient Descent [5pts]
  - Batch Gradient Descent [10pts Grad / 1.3% **Bonus for Undergrad**]
  - Momentum [5pts Grad / 0.7% **Bonus for Undergrad**]
- **1.2 Loss plot and CE for Gradient Descent** [5pts] - *programming*
- **1.3 Loss plot and CE for Batch Gradient Descent** [5pts Grad / 0.7% **Bonus for Undergrad**] - *programming*
- **1.4 Loss plot and CE value for NN with Gradient Descent with Momentum** [5pts Grad / 0.6% **Bonus for Undergrad**] - *programming*

Q2: CNN [20pts Grad / 2.7% Bonus for Undergrad + 1.1% Bonus for All]

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

- **2.1 Image Classification using Pytorch CNN** [20pts Grad / 2.7% **Bonus for Undergrad**]
  - 2.1.1 Loading the Model [5pts Grad / 0.7% **Bonus for Undergrad**] - *programming*
  - 2.1.2 Building the Model [5pts Grad / 0.7% **Bonus for Undergrad**] - *non-programming*
  - 2.1.3 Training the Model [8pts Grad / 1% **Bonus for Undergrad**] - *non-programming*
  - 2.1.4 Examining Accuracy and Loss [2pts Grad / 0.3% **Bonus for Undergrad**] - *non-programming*
- **2.2 Exploring Deep CNN Architectures** [1.1% **Bonus for All**] - *non-programming*

Q3: Random Forest [40pts + 2.1% Bonus for All]

Deliverables: random_forest.py and Written Report

- **3.1 Random Forest Implementation** [35pts] - *programming*
- **3.2 Hyperparameter Tuning with a Random Forest** [5pts] - *programming*

- **3.3 Plotting Feature Importance** [1.1% **Bonus for All**] - *non-programming*
- **3.4 ADABoost [1%** 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) [6.8% Bonus for all]

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

- **5.1: Model Architecture** [4.4% **Bonus for All**] - *programming*
  - 5.1.1: Defining the Simple RNN model [2.2% **Bonus for All**]
  - 5.1.2: Defining the LSTM model [2.2% **Bonus for All**]
- **5.2: Simple RNN vs LSTM Model Text Generation Training Comparison Analysis** [2.4% **Bonus for All**] - *non programming*

Points Totals:

- Total Base: 150 pts for grads / 105 pts for undergrads
- Total Undergrad Bonus: 6%
- Total Bonus for All: 10%

Submission Instructions

Submit the following files to their respective assignments on Gradescope for the programming portions:\n",

- **Assignment 4 Programming**
  - nn.py
  - random_forest.py
  - cnn_image_transformations.py (for grads only)
- **Assignment 4 Programming - Bonus for All**
  - random_forest.py
  - lstm.py
  - rnn.py
  - base_sequential_model.py
- **Assignment 4 Programming - Bonus for Undergrad**
  - NN.py
  - cnn_image_transformations.py

Environment Setup

In [29]:
```
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: 3.11.10 | packaged by Anaconda, Inc. | (main, Oct  3 2024, 07:22:26) [MSC v.1929 64 bit (AMD64)]
matplotlib: 3.9.2
numpy: 1.26.4
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

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.

```
In [30]: from codecarbon import EmissionsTracker

         tracker = EmissionsTracker()
         tracker.start()
```

```
[codecarbon ERROR @ 16:27:00] Error: Another instance of codecarbon is probably running as we find `C:\Users\ceeja\Ap
pData\Local\Temp\.codecarbon.lock`. Turn off the other instance to be able to run this one or use `allow_multiple_run
s` or delete the file. Exiting.
[codecarbon WARNING @ 16:27:00] Another instance of codecarbon is already running. Exiting.
[codecarbon WARNING @ 16:27:00] Another instance of codecarbon is already running. Exiting.
```

1: Two Layer Neural Network [80 pts; 55pts + 25pts Grad / 3.3% Undergrad Bonus] **[P]**

1.1 NN Implementation [65pts; 50pts + 15pts Grad / 2% 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 Batch Gradient Descent (BGD) algorithms for training these neural nets. **GD is mandatory for all. BGD is bonus for undergraduate students but mandatory for graduate students.**

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

- **silu**
- **derivative_silu**
- **softmax**
- **cross_entropy_loss**
- **_dropout**
- **forward**
- **compute_gradients**
- **update_weights**
- **backward**
- **gradient_descent**
- **batch_gradient_descent**: **Mandatory for graduate students, bonus for undergraduate students.** Please batch your data in

a wraparound manner. For example, given a dataset of 9 numbers, [1, 2, 3, 4, 5, 6, 7, 8, 9], and a batch size of 6, the first iteration batch will be [1, 2, 3, 4, 5, 6], the second iteration batch will be [7, 8, 9, 1, 2, 3], the third iteration batch will be [4, 5, 6, 7, 8, 9], etc...

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

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

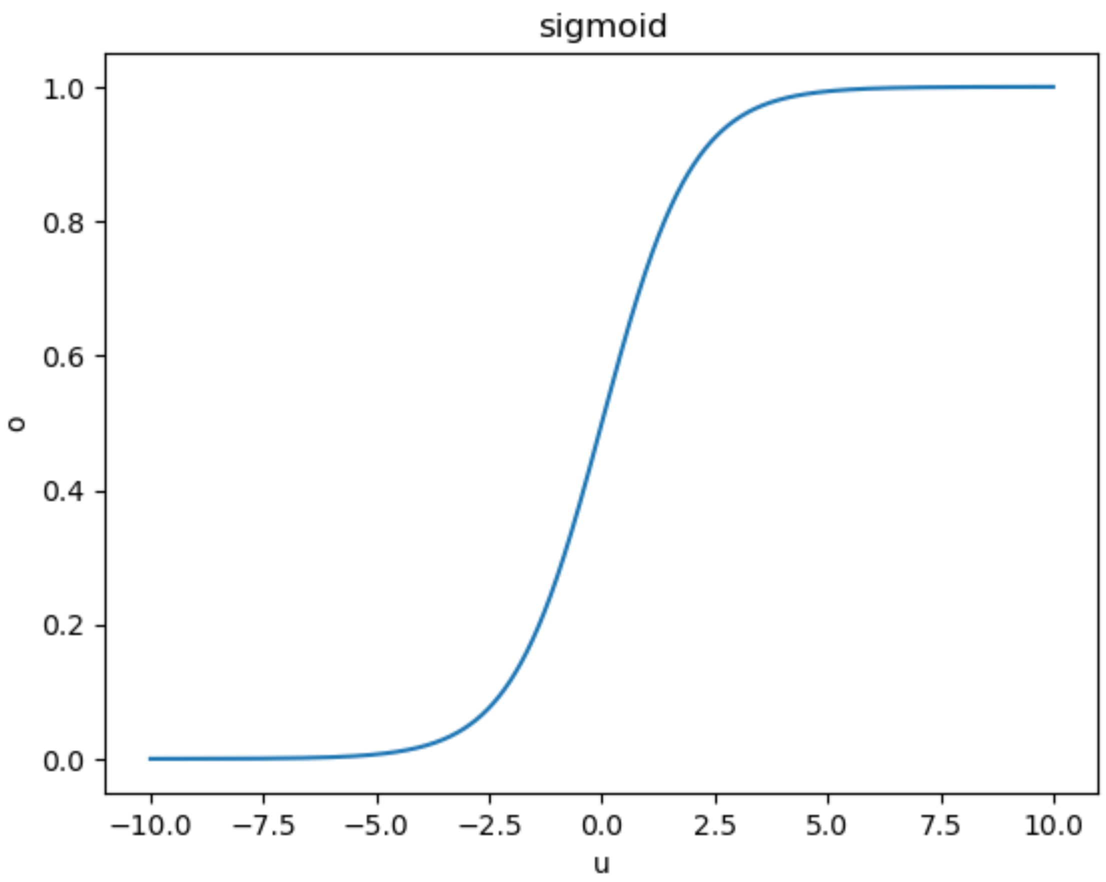
The sigmoid function is a non-linear function with an S-shaped curve and is regarded as a foundational activation function. Its output is in the range $(0, 1)$, making it the function to use for binary classification output. The function is expressed as

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

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.



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) = $e^{xi}/(e^{x1} + e^{x2} + ... + e^{xn})$

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

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 [31]:
```
from utilities.localtests import TestNN

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

```
test_softmax passed!
```

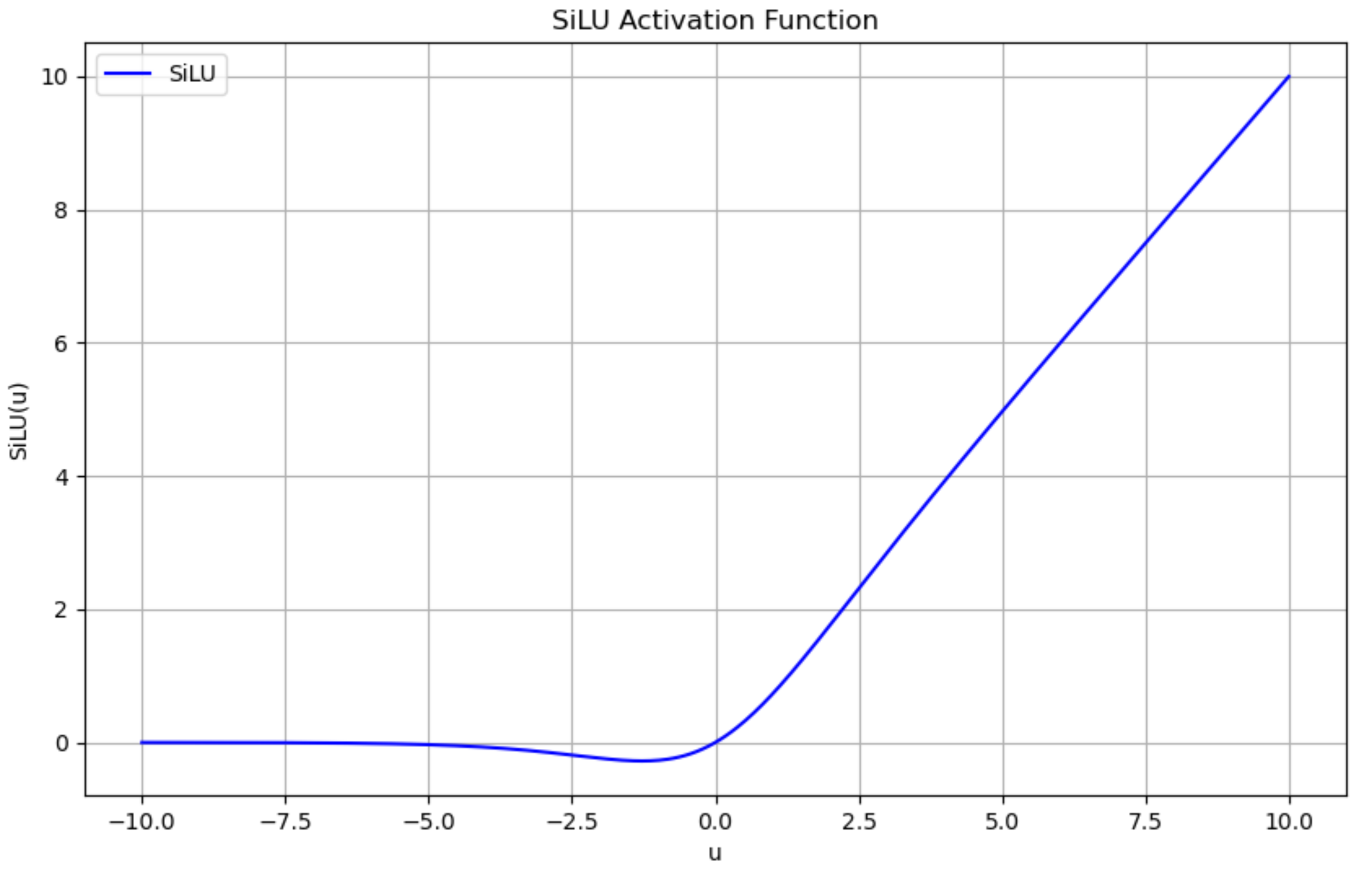
SiLU (Sigmoid Linear Unit)

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:

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

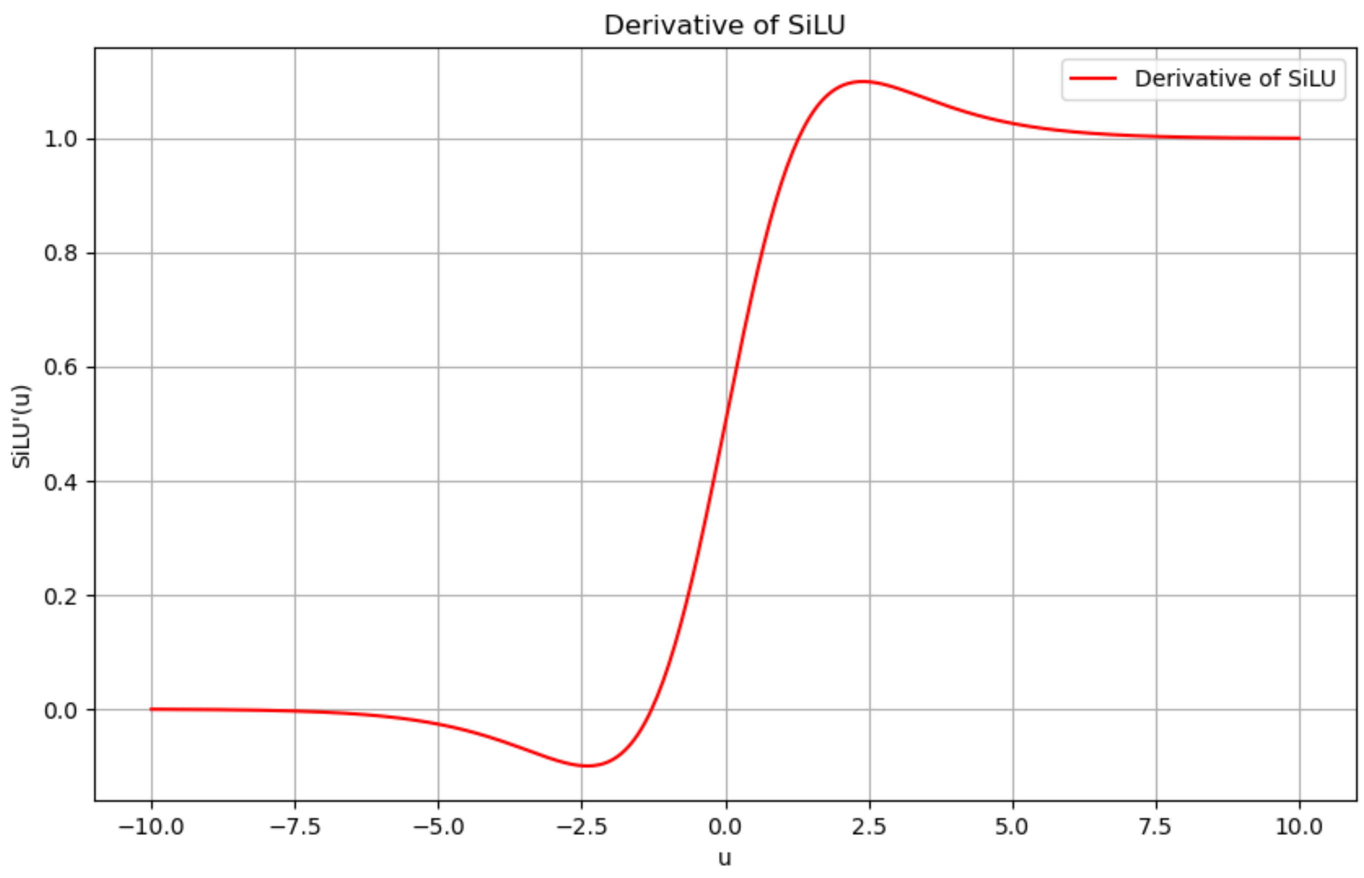


The derivative of SiLU, $\phi'(u)$, is given by:

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

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 [32]:
```
from utilities.localtests import TestNN

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

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

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.

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 $\mathrm{Bernoulli}(p)$ 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 $\mathrm{dropout}(u)$ 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 [33]:
```
from utilities.localtests import TestNN

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

```
test_dropout passed!
```

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.

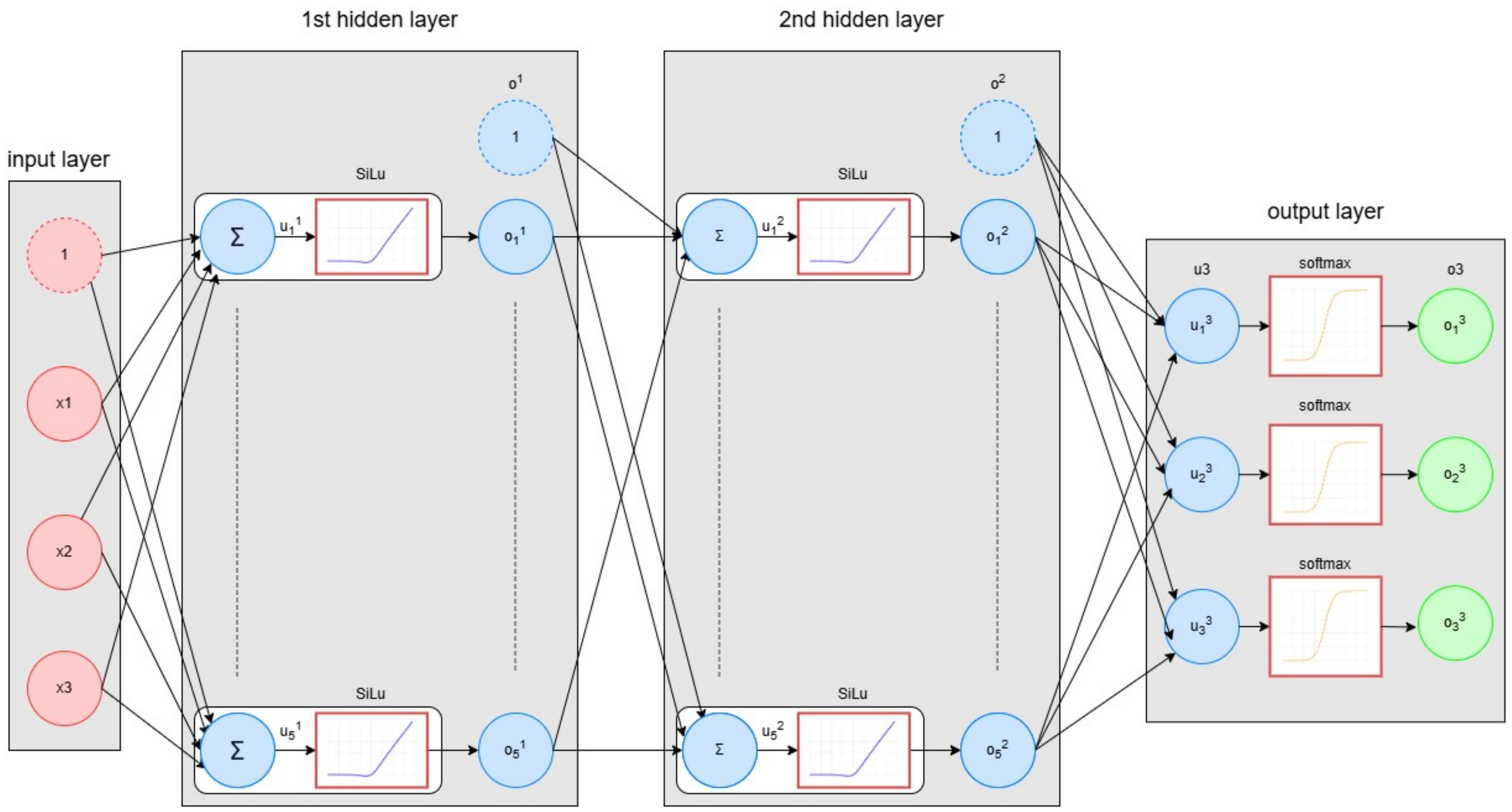
TODO: Implement the **cross_entropy_loss** function in **NN.py**.

```
In [34]: from utilities.localtests import TestNN

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

```
test_loss passed!
```

Neural Network Architecture

The architecture of our neural network.



The above diagram shows the dimensions of the neural network you will implement, along with the relationships between the quantities. Note that the neural network consists of two hidden linear 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.

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

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

In [35]:
```
from utilities.localtests import TestNN

TestNN("test_forward_without_dropout").test_forward_without_dropout()
TestNN("test_forward").test_forward()
```

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

Backward Propagation: Update Weights and Compute Gradients

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.

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_momentum=False.

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

In [36]:
```
from utilities.localtests import TestNN

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

```
test_update_weights passed!
```

Update Weights with Momentum [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. **Momentum** is a technique used to stabilize this convergence.

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.

Momentum proposes the following tweak to our parameter update function:

$$\begin{aligned} z_{t+1} &= \beta z_t + \nabla f(\theta_t) \\ \theta_{t+1} &= \theta_t - \alpha z_{t+1} \end{aligned}$$

where $\beta \in [0, 1]$ is the momentum constant and z_t represents the momentum records at time t.

You can think of momentum as taking our previous changes into consideration. If we've been moving in a certain direction recently, it's likely we should keep moving in that direction. The recurrence relation given shows that we use an exponentially-weighted average of the previous updates for our current update.

A useful analogy about momentum from this great article on Distill:

> Here’s a popular story about momentum: gradient descent is a man walking down a hill. He follows the steepest path downwards; his progress is slow, but steady. Momentum is a heavy ball rolling down the same hill. The added inertia acts both as a smoother and an accelerator, dampening oscillations and causing us to barrel through narrow valleys, small humps and local minima.

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

HINT: z is stored in `self.change`

In [37]:
```python
from utilities.localtests import TestNN

TestNN("test_update_weights_with_momentum").test_update_weights_with_momentum()
```

```
test_update_weights_with_momentum passed!
```

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:

$$\begin{aligned} u_1 &= \theta_1 X + b_1 \\ o_1 &= \text{silu}(u_1) \\ u_2 &= \theta_2 o_1 + b_2 \\ o_2 &= \text{silu}(u_2) \\ u_3 &= \theta_3 o2 + b_3 \\ o_3 &= \text{softmax}(u_3) \\ l &= \text{cross\\_entropy}(o_3) \end{aligned}$$

When computing gradients, we travel backwards from the loss all the way back ot 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

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

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:

$$\begin{aligned} \frac{\partial l}{\partial \theta_3} &= \frac{\partial l}{\partial u_3} \cdot \frac{\partial u_3}{\partial \theta_3} \\ \frac{\partial l}{\partial b_3} &= \frac{\partial l}{\partial u_3} \cdot \frac{\partial u_3}{\partial b_3}. \end{aligned}$$

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

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

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

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

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 [38]:
```
from utilities.localtests import TestNN

TestNN(
    "test_compute_gradients_without_dropout"
).test_compute_gradients_without_dropout()
TestNN("test_compute_gradients").test_compute_gradients()
```

```
test_compute_gradients_without_dropout passed!
test_compute_gradients passed!
```

1.1.1 Local Test: Gradient Descent

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

In [39]:
```
################################
### 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
```

1.1.2 Local Test: Batch Gradient Descent [No Points]

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

In [40]:
```
###############################
### DO NOT CHANGE THIS CELL ###
###############################

from utilities.localtests import TestNN

TestNN("test_batch_gradient_descent").test_batch_gradient_descent()
```

```
Loss after iteration 0: 1.085044
Loss after iteration 1: 1.110789
Loss after iteration 2: 1.109005

Your BGD losses works within the expected range: True
Your batch_y works within the expected range: True
```

1.1.3 Local Test: Gradient Descent with Momentum

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

In [41]:
```
###############################
### DO NOT CHANGE THIS CELL ###
###############################
```

```python
from utilities.localtests import TestNN

TestNN("test_gradient_descent_with_momentum").test_gradient_descent_with_momentum()
```

```
Loss after iteration 0: 1.086276
Loss after iteration 1: 1.086233
Loss after iteration 2: 1.086168

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

1.2 Loss plot and cross-entropy(CE) value for NN with Gradient Descent [5pts] **[P]**

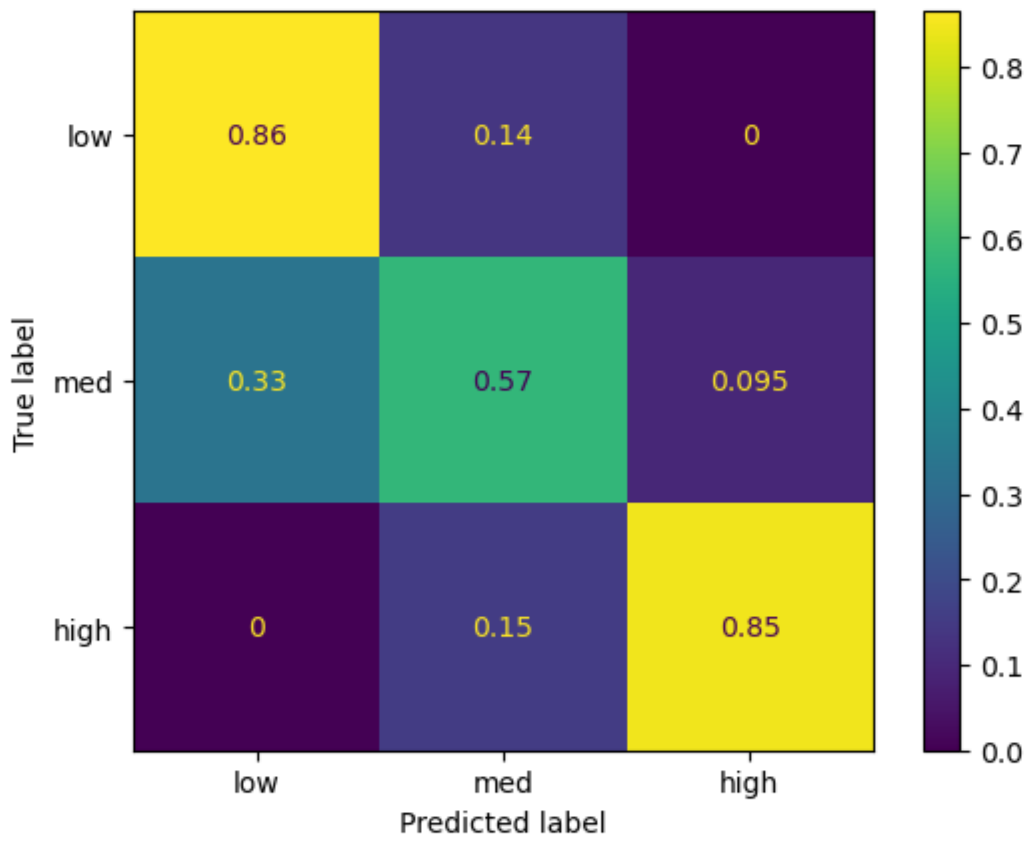
Train your neural network implementation with gradient descent and print out the loss at every 1000th iteration (starting at iteration 0). The following cells will plot the loss vs epoch graph and calculate the final test cross-entropy(CE).

In [42]:
```python
###############################
### 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=False, use_momentum=False
)  # initalize neural net class
nn.gradient_descent(x_train, y_train, iter=60000)  # train
```

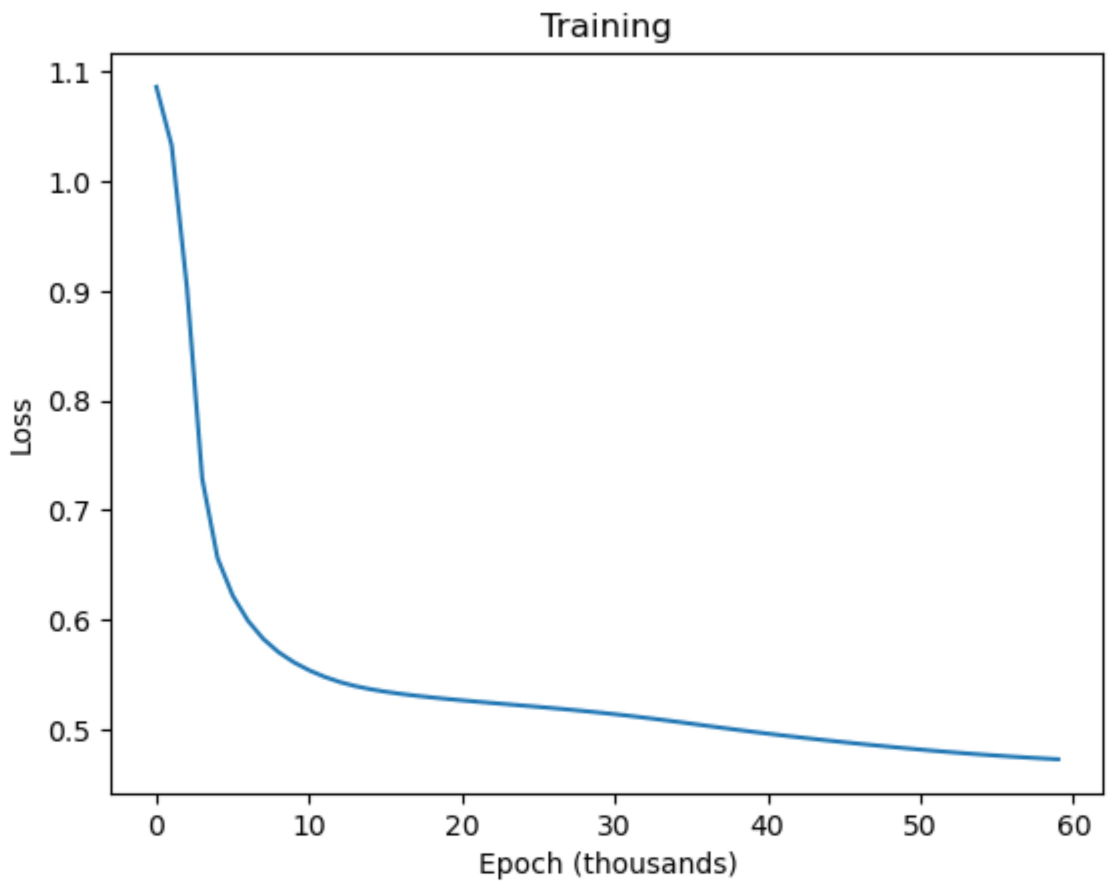
```
Loss after iteration 0: 1.086276
Loss after iteration 1000: 1.032790
Loss after iteration 2000: 0.903132
Loss after iteration 3000: 0.728430
Loss after iteration 4000: 0.656171
Loss after iteration 5000: 0.621733
Loss after iteration 6000: 0.598812
Loss after iteration 7000: 0.582288
Loss after iteration 8000: 0.570177
Loss after iteration 9000: 0.561003
Loss after iteration 10000: 0.553710
Loss after iteration 11000: 0.547769
Loss after iteration 12000: 0.543023
Loss after iteration 13000: 0.539342
Loss after iteration 14000: 0.536473
Loss after iteration 15000: 0.534156
Loss after iteration 16000: 0.532210
Loss after iteration 17000: 0.530515
Loss after iteration 18000: 0.528996
Loss after iteration 19000: 0.527603
Loss after iteration 20000: 0.526301
Loss after iteration 21000: 0.525063
Loss after iteration 22000: 0.523866
Loss after iteration 23000: 0.522691
Loss after iteration 24000: 0.521519
Loss after iteration 25000: 0.520332
Loss after iteration 26000: 0.519114
Loss after iteration 27000: 0.517849
Loss after iteration 28000: 0.516524
Loss after iteration 29000: 0.515125
Loss after iteration 30000: 0.513643
Loss after iteration 31000: 0.512071
Loss after iteration 32000: 0.510410
Loss after iteration 33000: 0.508666
Loss after iteration 34000: 0.506852
Loss after iteration 35000: 0.504993
Loss after iteration 36000: 0.503116
Loss after iteration 37000: 0.501250
Loss after iteration 38000: 0.499419
Loss after iteration 39000: 0.497640
Loss after iteration 40000: 0.495918
Loss after iteration 41000: 0.494254
Loss after iteration 42000: 0.492645
Loss after iteration 43000: 0.491086
Loss after iteration 44000: 0.489571
Loss after iteration 45000: 0.488099
Loss after iteration 46000: 0.486665
Loss after iteration 47000: 0.485271
Loss after iteration 48000: 0.483918
Loss after iteration 49000: 0.482609
Loss after iteration 50000: 0.481346
Loss after iteration 51000: 0.480134
Loss after iteration 52000: 0.478976
Loss after iteration 53000: 0.477877
Loss after iteration 54000: 0.476838
Loss after iteration 55000: 0.475859
Loss after iteration 56000: 0.474941
Loss after iteration 57000: 0.474082
Loss after iteration 58000: 0.473277
Loss after iteration 59000: 0.472523
```

In [43]:

```
# 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 [44]:
```
# 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 [45]:
```
# 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: 0.729
```

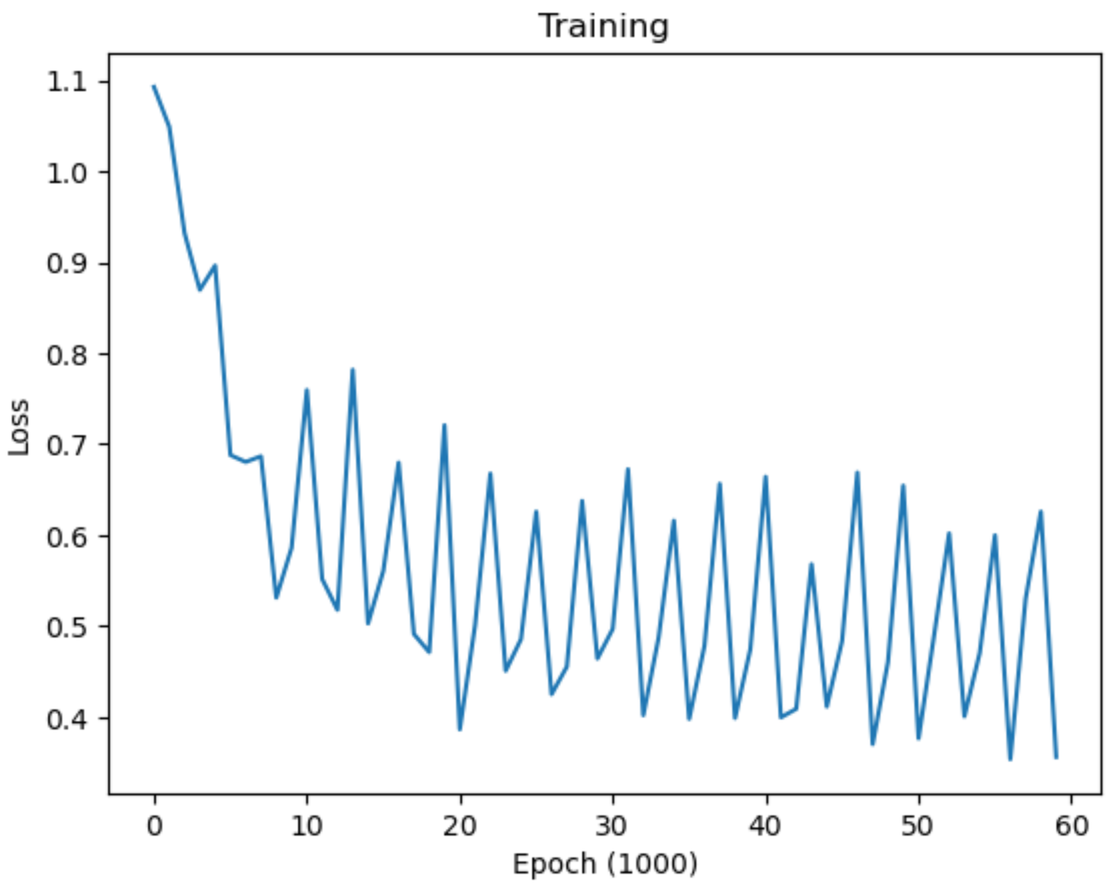
1.3 Loss plot and CE value for NN with BGD [5pts Grad / 0.7% Bonus for Undergrad] **[P]**

Train your neural network implementation with 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.

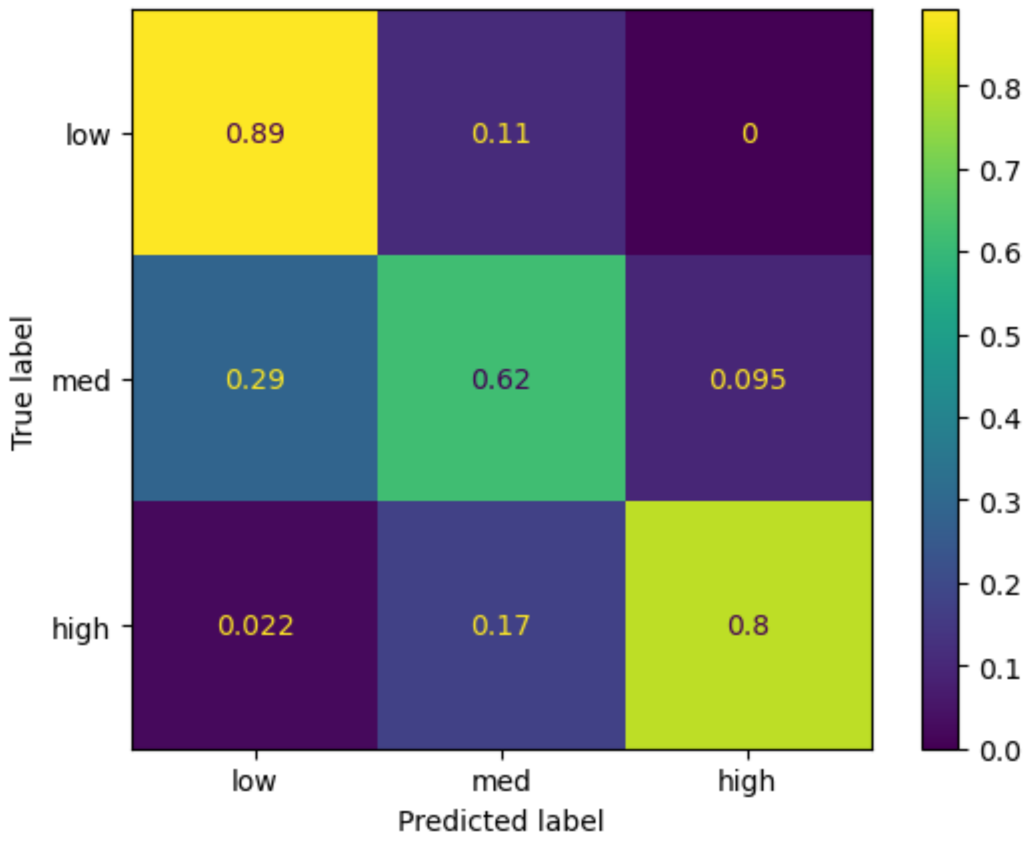
In [46]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
from NN import NeuralNet
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

x_train, y_train, x_test, y_test = get_housing_dataset()

nn = NeuralNet(
    y_train, lr=0.01, use_dropout=True, use_momentum=False
)  # initalize neural net class
nn.batch_gradient_descent(x_train, y_train, iter=60000, use_momentum=False)
```

```
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 [47]:
```
# 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 [48]:
```
# 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 [49]:
```
# 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 NN with Gradient Descent with Momentum [5pts Grad / 0.6% Bonus for Undergrad] **[P]**

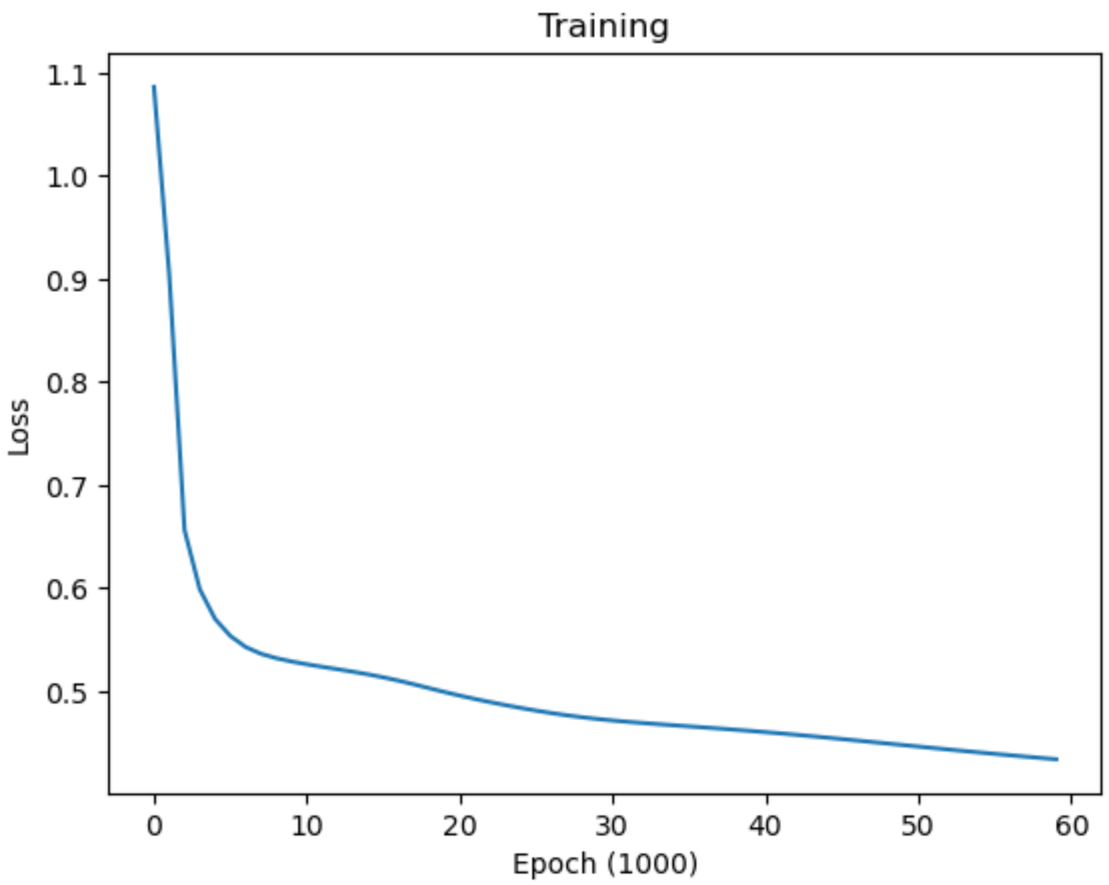
Train your neural net implementation using gradient descent with momentum and print out the loss at every 1000th iteration (starting at iteration 0). The following cells will plot the loss vs epoch graph and calculate the final test CE.

In [50]:
```
################################
```

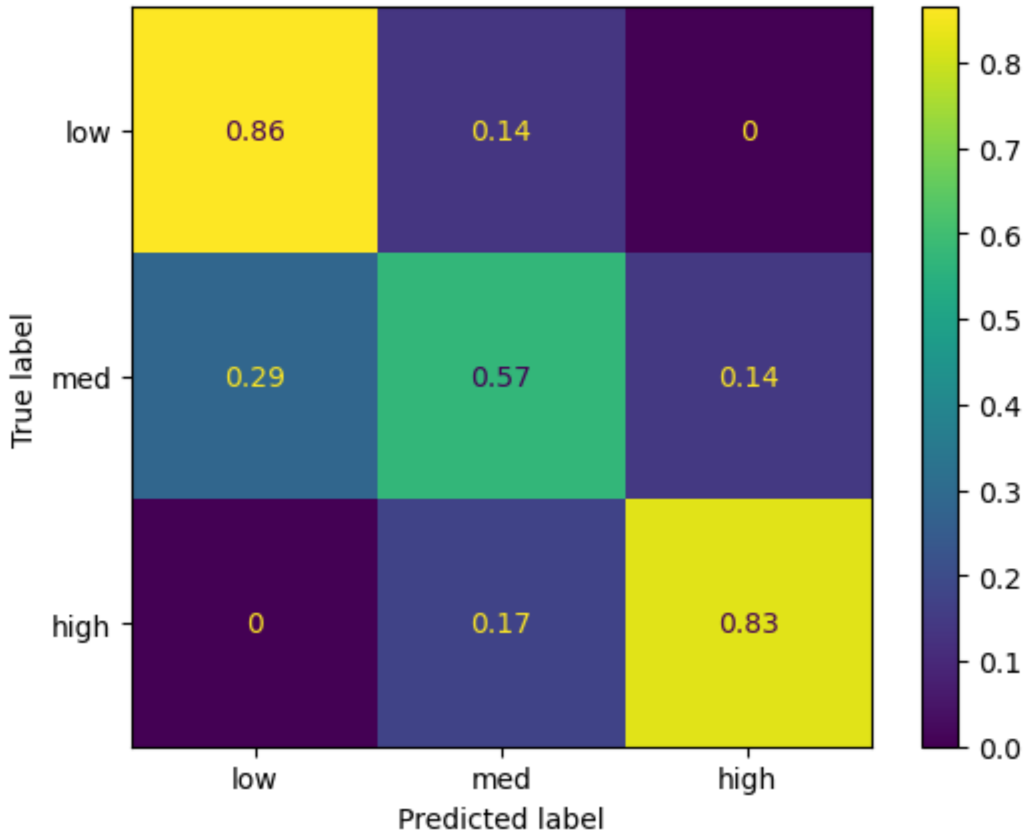
```
### DO NOT CHANGE THIS CELL ###
###############################
from NN import NeuralNet
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

x_train, y_train, x_test, y_test = get_housing_dataset()

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

```
Loss after iteration 0: 1.086276
Loss after iteration 1000: 0.904120
Loss after iteration 2000: 0.656440
Loss after iteration 3000: 0.598945
Loss after iteration 4000: 0.570251
Loss after iteration 5000: 0.553760
Loss after iteration 6000: 0.543053
Loss after iteration 7000: 0.536490
Loss after iteration 8000: 0.532221
Loss after iteration 9000: 0.529004
Loss after iteration 10000: 0.526308
Loss after iteration 11000: 0.523874
Loss after iteration 12000: 0.521527
Loss after iteration 13000: 0.519123
Loss after iteration 14000: 0.516534
Loss after iteration 15000: 0.513655
Loss after iteration 16000: 0.510424
Loss after iteration 17000: 0.506868
Loss after iteration 18000: 0.503132
Loss after iteration 19000: 0.499435
Loss after iteration 20000: 0.495933
Loss after iteration 21000: 0.492660
Loss after iteration 22000: 0.489586
Loss after iteration 23000: 0.486679
Loss after iteration 24000: 0.483932
Loss after iteration 25000: 0.481358
Loss after iteration 26000: 0.478988
Loss after iteration 27000: 0.476848
Loss after iteration 28000: 0.474951
Loss after iteration 29000: 0.473285
Loss after iteration 30000: 0.471821
Loss after iteration 31000: 0.470516
Loss after iteration 32000: 0.469328
Loss after iteration 33000: 0.468217
Loss after iteration 34000: 0.467149
Loss after iteration 35000: 0.466099
Loss after iteration 36000: 0.465045
Loss after iteration 37000: 0.463971
Loss after iteration 38000: 0.462864
Loss after iteration 39000: 0.461714
Loss after iteration 40000: 0.460514
Loss after iteration 41000: 0.459260
Loss after iteration 42000: 0.457953
Loss after iteration 43000: 0.456598
Loss after iteration 44000: 0.455204
Loss after iteration 45000: 0.453780
Loss after iteration 46000: 0.452338
Loss after iteration 47000: 0.450887
Loss after iteration 48000: 0.449431
Loss after iteration 49000: 0.447975
Loss after iteration 50000: 0.446521
Loss after iteration 51000: 0.445070
Loss after iteration 52000: 0.443628
Loss after iteration 53000: 0.442202
Loss after iteration 54000: 0.440797
Loss after iteration 55000: 0.439420
Loss after iteration 56000: 0.438073
Loss after iteration 57000: 0.436757
Loss after iteration 58000: 0.435469
Loss after iteration 59000: 0.434208
```

In [51]:
```
# 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 [52]: # 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 [53]: # 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: 0.696
```

2: Image Classification based on Convolutional Neural Networks [20pts: 20pts Grad / 2.7% Bonus for Undergrad + 1.1% Bonus for all] **[P][W]**

2.1 Image Classification using Pytorch and CNN [20pts Grad / 2.7% 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 [54]:
```
import torch
import torchvision
from torch.utils.data import non_deterministic
from torchvision.transforms import v2

%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 FashionMNIST Dataset and Data Augmentation [5pts - Bonus for Undergrad]

We use Fashion-MNIST dataset to train our model. This is a dataset of 70,000 28x28 grayscale images in 10 classes. There are 60,000 training images and 10,000 test images. We provide code for you to download Fashion-MNIST dataset below.

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

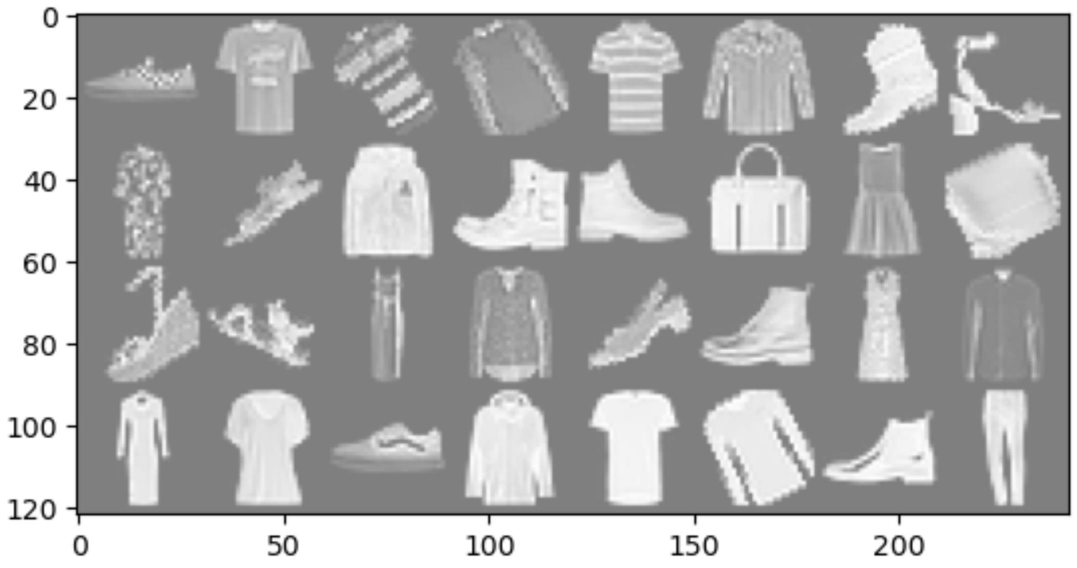
In [55]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from cnn_image_transformations import (
    create_testing_transformations,
    create_training_transformations,
)

# Create Transformations
training_transformations = create_training_transformations()
testing_transformation = create_testing_transformations()

# Load data
trainset = torchvision.datasets.FashionMNIST(
    root="./data", train=True, download=True, transform=training_transformations
)
testset = torchvision.datasets.FashionMNIST(
    root="./data", train=False, download=True, transform=testing_transformation
)

classes = (
    "Top",
    "Trouser",
    "Pullover",
    "Dress",
    "Coat",
    "Sandal",
    "Shirt",
    "Sneaker",
    "Bag",
    "Ankle boot",
)

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

```
torch.Size([60000, 28, 28])
torch.Size([10000, 28, 28])
```

Load some sample images from Fashion-MNIST [Setup - No points]

In [56]:
```
###############################
### DO NOT CHANGE THIS CELL ###
###############################

import matplotlib.pyplot as plt
import numpy as np

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
)

# functions to show an image


def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = next(dataiter)

print("Image size")
print(v2.functional.get_size(images[0]))

# show images
imshow(torchvision.utils.make_grid(images))
```

```
Image size
[28, 28]
```



As you can see from above, the FashionMNIST dataset contains different types of objects. The images have been size-normalized and objects remain centered in fixed-size images.

2.1.2 Build convolutional neural network model [5pts Grad / 0.7% Bonus for Undergrad] [W]

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: $[28 \times 28 \times 1]$ will hold the raw pixel values of the image, in this case, an image of width 28, height 28. 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 $[28 \times 28 \times 8]$ 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 10 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 you model architecture

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

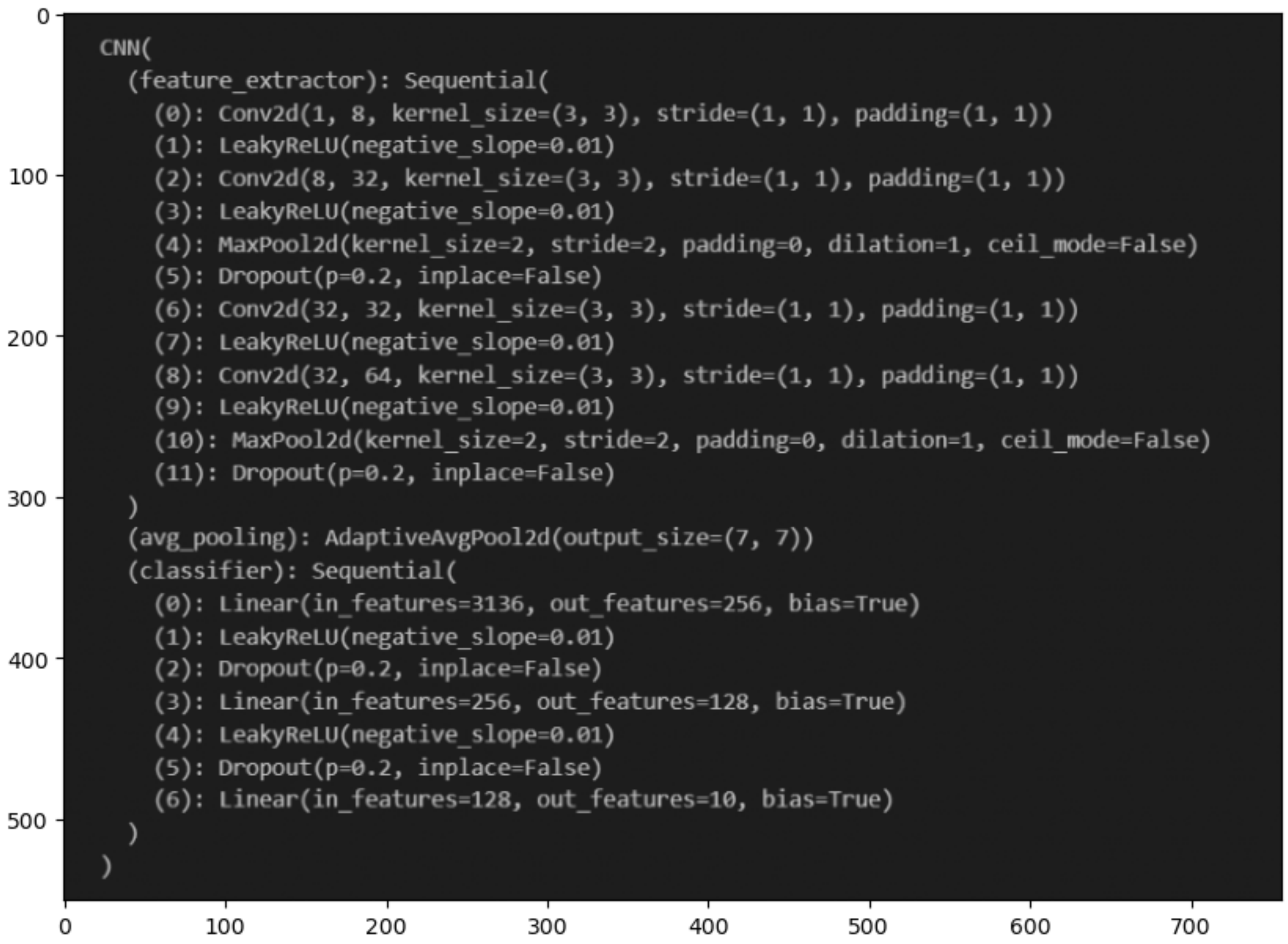
The following links are Pytorch documentation for the layers you are going to use to build the CNN.

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

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

In [57]:
```
################################
### 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[57]:
```
<matplotlib.image.AxesImage at 0x1bc067b7410>
```



```
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): Linear(in_features=3136, out_features=256, bias=True)
    (1): LeakyReLU(negative_slope=0.01)
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=256, out_features=128, bias=True)
    (4): LeakyReLU(negative_slope=0.01)
    (5): Dropout(p=0.2, inplace=False)
    (6): Linear(in_features=128, out_features=10, bias=True)
  )
)
```

Defining model [5pts Grad / 0.7% Bonus for Undergrad]**[W]**

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 examine your architecture.

In [58]:
```
###############################
### 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): Linear(in_features=3136, out_features=256, bias=True)
    (1): LeakyReLU(negative_slope=0.01)
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=256, out_features=128, bias=True)
    (4): LeakyReLU(negative_slope=0.01)
    (5): Dropout(p=0.2, inplace=False)
    (6): Linear(in_features=128, out_features=10, bias=True)
    (7): Softmax(dim=1)
  )
)
```

2.1.3 Train the network [8pts Grad / 1% Bonus for Undergrad] **[W]**

Tuning: Training the network is the next thing to try. You can set the hyperparameters in the cell below. If your hyperparameters are set properly, you should see the loss of the validation set decreased and the value of accuracy increased. **It may take more than 15 minutes to train your model.**

- Recommended Batch Sizes fall in the range 32-512 (use powers of 2)
- Recommended Epoch Counts fall in the range 5-20
- Recommended Learning Rates fall in the range .0001-.01

Expected Result: You should be able to achieve more than 90% accuracy on the test set to get full points. If you achieve accuracy between 75% to 84%, you will only get 3 points. An accuracy between 84% to 90% will earn an additional 3pts.

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.

- 75% to 84% earns 3pts
- 84% to 90% earns 3pts more (6pts total)
- $90\%+$ earns 2pts more (8pts total)

Train your own CNN model

In [59]:
```
from cnn import CNN
from cnn_trainer import Trainer
import itertools


# # TODO: Change hyperparameters here
# batch_sizes = [32, 64, 128]
# epochs_list = [10, 15, 20]
# lrs = [1e-3, 5e-3, 1e-2]

# for batch_size, epoch, init_lr in itertools.product(batch_sizes, epochs_list, lrs):
#     print(f"Training with batch_size={batch_size}, num_epochs={epoch}, init_lr={init_lr}")

#     net = CNN()

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

#     trainer = Trainer(
#         net,
#         trainset,
#         testset,
#         num_epochs=epoch,
#         batch_size=batch_size,
#         init_lr=init_lr,
#         device=device,
#     )
#     trainer.train()

batch_size = 32
epoch = 20
lr = 5e-3

print(f"Training with batch_size={batch_size}, num_epochs={epoch}, init_lr={lr}")

net = CNN()

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

trainer = Trainer(
    net,
    trainset,
    testset,
    num_epochs=epoch,
    batch_size=batch_size,
    init_lr=lr,
    device=device,
)
trainer.train()
```

```
Training with batch_size=32, num_epochs=20, init_lr=0.005
Using cpu device
Epoch 1/20: 100%|██████████| 1875/1875 [00:52<00:00, 35.61batch/s, accuracy=0.683, loss=1.78]
Epoch 1: Validation Loss: 1.65, Validation Accuracy: 0.811
Epoch 2/20: 100%|██████████| 1875/1875 [00:53<00:00, 35.19batch/s, accuracy=0.797, loss=1.66]
Epoch 2: Validation Loss: 1.63, Validation Accuracy: 0.830
Epoch 3/20: 100%|██████████| 1875/1875 [00:52<00:00, 35.79batch/s, accuracy=0.819, loss=1.64]
Epoch 3: Validation Loss: 1.61, Validation Accuracy: 0.847
Epoch 4/20: 100%|██████████| 1875/1875 [00:57<00:00, 32.81batch/s, accuracy=0.835, loss=1.63]
Epoch 4: Validation Loss: 1.62, Validation Accuracy: 0.844
Epoch 5/20: 100%|██████████| 1875/1875 [00:53<00:00, 35.25batch/s, accuracy=0.843, loss=1.62]
Epoch 5: Validation Loss: 1.61, Validation Accuracy: 0.853
Epoch 6/20: 100%|██████████| 1875/1875 [00:54<00:00, 34.44batch/s, accuracy=0.851, loss=1.61]
Epoch 6: Validation Loss: 1.59, Validation Accuracy: 0.868
Epoch 7/20: 100%|██████████| 1875/1875 [00:56<00:00, 33.30batch/s, accuracy=0.856, loss=1.6]
Epoch 7: Validation Loss: 1.59, Validation Accuracy: 0.872
Epoch 8/20: 100%|██████████| 1875/1875 [00:54<00:00, 34.28batch/s, accuracy=0.862, loss=1.6]
Epoch 8: Validation Loss: 1.58, Validation Accuracy: 0.877
Epoch 9/20: 100%|██████████| 1875/1875 [00:54<00:00, 34.47batch/s, accuracy=0.866, loss=1.59]
Epoch 9: Validation Loss: 1.58, Validation Accuracy: 0.877
Epoch 10/20: 100%|██████████| 1875/1875 [00:53<00:00, 34.93batch/s, accuracy=0.874, loss=1.59]
Epoch 10: Validation Loss: 1.58, Validation Accuracy: 0.885
Epoch 11/20: 100%|██████████| 1875/1875 [00:53<00:00, 35.17batch/s, accuracy=0.876, loss=1.58]
Epoch 11: Validation Loss: 1.57, Validation Accuracy: 0.892
Epoch 12/20: 100%|██████████| 1875/1875 [01:02<00:00, 29.99batch/s, accuracy=0.882, loss=1.58]
Epoch 12: Validation Loss: 1.58, Validation Accuracy: 0.884
Epoch 13/20: 100%|██████████| 1875/1875 [00:59<00:00, 31.43batch/s, accuracy=0.884, loss=1.58]
Epoch 13: Validation Loss: 1.56, Validation Accuracy: 0.896
Epoch 14/20: 100%|██████████| 1875/1875 [00:53<00:00, 34.87batch/s, accuracy=0.886, loss=1.57]
Epoch 14: Validation Loss: 1.56, Validation Accuracy: 0.898
Epoch 15/20: 100%|██████████| 1875/1875 [00:56<00:00, 32.98batch/s, accuracy=0.89, loss=1.57]
Epoch 15: Validation Loss: 1.56, Validation Accuracy: 0.902
Epoch 16/20: 100%|██████████| 1875/1875 [00:58<00:00, 32.26batch/s, accuracy=0.892, loss=1.57]
Epoch 16: Validation Loss: 1.56, Validation Accuracy: 0.901
Epoch 17/20: 100%|██████████| 1875/1875 [00:55<00:00, 33.96batch/s, accuracy=0.894, loss=1.57]
Epoch 17: Validation Loss: 1.56, Validation Accuracy: 0.905
Epoch 18/20: 100%|██████████| 1875/1875 [00:52<00:00, 35.71batch/s, accuracy=0.896, loss=1.57]
Epoch 18: Validation Loss: 1.56, Validation Accuracy: 0.904
Epoch 19/20: 100%|██████████| 1875/1875 [00:52<00:00, 35.78batch/s, accuracy=0.896, loss=1.56]
Epoch 19: Validation Loss: 1.56, Validation Accuracy: 0.904
Epoch 20/20: 100%|██████████| 1875/1875 [00:52<00:00, 36.05batch/s, accuracy=0.899, loss=1.56]
Epoch 20: Validation Loss: 1.56, Validation Accuracy: 0.904
```

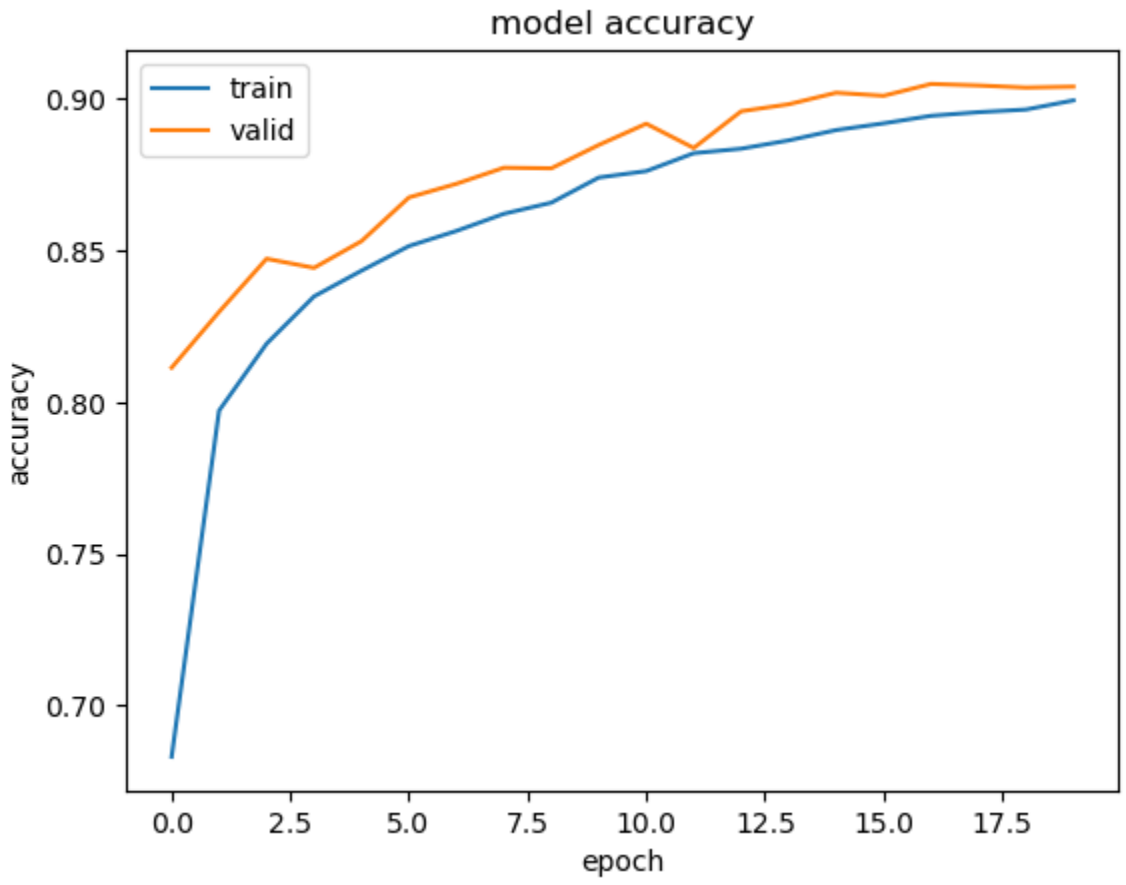
```
Output 1: Training with batch_size=32, num_epochs=10, init_lr=0.001 Using cpu device Epoch 1/10: 100%|██████████|
1875/1875 [00:54<00:00, 34.58batch/s, accuracy=0.66, loss=1.8]
Epoch 1: Validation Loss: 1.72, Validation Accuracy: 0.740 Epoch 2/10: 100%|██████████| 1875/1875 [00:53<00:00, 35.23batch/s,
accuracy=0.747, loss=1.72] Epoch 2: Validation Loss: 1.67, Validation Accuracy: 0.789 Epoch 3/10: 100%|██████████| 1875/1875
[00:53<00:00, 35.33batch/s, accuracy=0.766, loss=1.69] Epoch 3: Validation Loss: 1.66, Validation Accuracy: 0.805 Epoch 4/10: 100%|
██████████| 1875/1875 [00:56<00:00, 33.29batch/s, accuracy=0.776, loss=1.68] Epoch 4: Validation Loss: 1.64, Validation
Accuracy: 0.816 Epoch 5/10: 100%|██████████| 1875/1875 [00:52<00:00, 35.98batch/s, accuracy=0.784, loss=1.68] Epoch 5:
Validation Loss: 1.64, Validation Accuracy: 0.815 Epoch 6/10: 100%|██████████| 1875/1875 [00:51<00:00, 36.27batch/s,
accuracy=0.79, loss=1.67] Epoch 6: Validation Loss: 1.64, Validation Accuracy: 0.821 Epoch 7/10: 100%|██████████| 1875/1875
[00:51<00:00, 36.18batch/s, accuracy=0.794, loss=1.67] Epoch 7: Validation Loss: 1.64, Validation Accuracy: 0.826 Epoch 8/10: 100%|
██████████| 1875/1875 [00:53<00:00, 35.18batch/s, accuracy=0.797, loss=1.66] Epoch 8: Validation Loss: 1.63, Validation
Accuracy: 0.830 Epoch 9/10: 100%|██████████| 1875/1875 [00:51<00:00, 36.09batch/s, accuracy=0.8, loss=1.66]
Epoch 9: Validation Loss: 1.63, Validation Accuracy: 0.831 Epoch 10/10: 100%|██████████| 1875/1875 [00:54<00:00, 34.39batch/
s, accuracy=0.802, loss=1.66] Epoch 10: Validation Loss: 1.63, Validation Accuracy: 0.828 Training with batch_size=32,
num_epochs=10, init_lr=0.005 Using cpu device Epoch 1/10: 100%|██████████| 1875/1875 [00:53<00:00, 35.07batch/s,
accuracy=0.675, loss=1.79] Epoch 1: Validation Loss: 1.65, Validation Accuracy: 0.807 Epoch 2/10: 100%|██████████| 1875/1875
[00:52<00:00, 35.43batch/s, accuracy=0.799, loss=1.66] Epoch 2: Validation Loss: 1.63, Validation Accuracy: 0.834 Epoch 3/10: 100%|
██████████| 1875/1875 [00:57<00:00, 32.76batch/s, accuracy=0.825, loss=1.64] Epoch 3: Validation Loss: 1.63, Validation
Accuracy: 0.829 Epoch 4/10: 100%|██████████| 1875/1875 [00:53<00:00, 34.88batch/s, accuracy=0.834, loss=1.63] Epoch 4:
Validation Loss: 1.61, Validation Accuracy: 0.854 Epoch 5/10: 100%|██████████| 1875/1875 [00:53<00:00, 35.00batch/s,
accuracy=0.844, loss=1.62] Epoch 5: Validation Loss: 1.60, Validation Accuracy: 0.858 Epoch 6/10: 100%|██████████| 1875/1875
[00:54<00:00, 34.67batch/s, accuracy=0.848, loss=1.61] Epoch 6: Validation Loss: 1.59, Validation Accuracy: 0.867 Epoch 7/10: 100%|
██████████| 1875/1875 [00:54<00:00, 34.55batch/s, accuracy=0.856, loss=1.6] Epoch 7: Validation Loss: 1.59, Validation
Accuracy: 0.874 Epoch 8/10: 100%|██████████| 1875/1875 [00:56<00:00, 33.37batch/s, accuracy=0.862, loss=1.6] Epoch 8:
Validation Loss: 1.59, Validation Accuracy: 0.876 Epoch 9/10: 100%|██████████| 1875/1875 [00:54<00:00, 34.68batch/s,
accuracy=0.867, loss=1.59] Epoch 9: Validation Loss: 1.58, Validation Accuracy: 0.882 Epoch 10/10: 100%|██████████|
1875/1875 [00:53<00:00, 34.76batch/s, accuracy=0.871, loss=1.59] Epoch 10: Validation Loss: 1.58, Validation Accuracy: 0.883
Training with batch_size=32, num_epochs=10, init_lr=0.01 Using cpu device Epoch 1/10: 100%|██████████| 1875/1875
[00:52<00:00, 35.54batch/s, accuracy=0.427, loss=2.03] Epoch 1: Validation Loss: 2.01, Validation Accuracy: 0.454 Epoch 2/10: 100%|
██████████| 1875/1875 [00:52<00:00, 35.84batch/s, accuracy=0.453, loss=2.01] Epoch 2: Validation Loss: 1.94, Validation
Accuracy: 0.524 Epoch 3/10: 100%|██████████| 1875/1875 [00:52<00:00, 35.48batch/s, accuracy=0.466, loss=2]
Epoch 3: Validation Loss: 1.94, Validation Accuracy: 0.519 Epoch 4/10: 100%|██████████| 1875/1875 [00:53<00:00, 35.03batch/s,
accuracy=0.481, loss=1.98] Epoch 4: Validation Loss: 1.94, Validation Accuracy: 0.518 Epoch 5/10: 100%|██████████| 1875/1875
[00:54<00:00, 34.67batch/s, accuracy=0.491, loss=1.97] Epoch 5: Validation Loss: 1.95, Validation Accuracy: 0.515 Epoch 6/10: 100%|
██████████| 1875/1875 [00:52<00:00, 35.46batch/s, accuracy=0.492, loss=1.97] Epoch 6: Validation Loss: 1.92, Validation
Accuracy: 0.542 Epoch 7/10: 100%|██████████| 1875/1875 [00:52<00:00, 35.77batch/s, accuracy=0.503, loss=1.96] Epoch 7:
Validation Loss: 1.92, Validation Accuracy: 0.537 Epoch 8/10: 100%|██████████| 1875/1875 [00:53<00:00, 35.34batch/s,
accuracy=0.507, loss=1.95] Epoch 8: Validation Loss: 1.95, Validation Accuracy: 0.511 Epoch 9/10: 100%|██████████| 1875/1875
[00:51<00:00, 36.06batch/s, accuracy=0.506, loss=1.95] Epoch 9: Validation Loss: 1.92, Validation Accuracy: 0.542 Epoch 10/10:
100%|██████████| 1875/1875 [00:51<00:00, 36.12batch/s, accuracy=0.509, loss=1.95] Epoch 10: Validation Loss: 1.93, Validation
Accuracy: 0.528 Training with batch_size=32, num_epochs=15, init_lr=0.001 Using cpu device Epoch 1/15: 100%|██████████|
1875/1875 [00:52<00:00, 35.95batch/s, accuracy=0.625, loss=1.84] Epoch 1: Validation Loss: 1.74, Validation Accuracy: 0.717 Epoch
2/15: 100%|██████████| 1875/1875 [00:53<00:00, 35.14batch/s, accuracy=0.715, loss=1.75] Epoch 2: Validation Loss: 1.70,
Validation Accuracy: 0.760 Epoch 3/15: 100%|██████████| 1875/1875 [00:52<00:00, 35.87batch/s, accuracy=0.746, loss=1.72]
Epoch 3: Validation Loss: 1.68, Validation Accuracy: 0.780 Epoch 4/15: 100%|██████████| 1875/1875 [00:52<00:00, 36.04batch/s,
accuracy=0.756, loss=1.7] Epoch 4: Validation Loss: 1.67, Validation Accuracy: 0.790 Epoch 5/15: 100%|██████████| 1875/1875
[00:53<00:00, 35.00batch/s, accuracy=0.764, loss=1.7] Epoch 5: Validation Loss: 1.67, Validation Accuracy: 0.795 Epoch 6/15: 100%|
██████████| 1875/1875 [00:52<00:00, 36.01batch/s, accuracy=0.768, loss=1.69] Epoch 6: Validation Loss: 1.66, Validation
Accuracy: 0.797 Epoch 7/15: 100%|██████████| 1875/1875 [00:51<00:00, 36.14batch/s, accuracy=0.776, loss=1.68] Epoch 7:
Validation Loss: 1.66, Validation Accuracy: 0.804 Epoch 8/15: 100%|██████████| 1875/1875 [00:51<00:00, 36.15batch/s,
accuracy=0.793, loss=1.67] Epoch 8: Validation Loss: 1.65, Validation Accuracy: 0.809 Epoch 9/15: 100%|██████████| 1875/1875
[00:52<00:00, 35.97batch/s, accuracy=0.797, loss=1.66] Epoch 9: Validation Loss: 1.65, Validation Accuracy: 0.807 Epoch 10/15:
100%|██████████| 1875/1875 [00:52<00:00, 35.95batch/s, accuracy=0.801, loss=1.66] Epoch 10: Validation Loss: 1.65, Validation
Accuracy: 0.808 Epoch 11/15: 100%|██████████| 1875/1875 [00:52<00:00, 36.01batch/s, accuracy=0.803, loss=1.66] Epoch 11:
Validation Loss: 1.65, Validation Accuracy: 0.813 Epoch 12/15: 100%|██████████| 1875/1875 [00:53<00:00, 35.36batch/s,
accuracy=0.806, loss=1.66] Epoch 12: Validation Loss: 1.65, Validation Accuracy: 0.813 Epoch 13/15: 100%|██████████|
1875/1875 [00:52<00:00, 35.92batch/s, accuracy=0.807, loss=1.65] Epoch 13: Validation Loss: 1.64, Validation Accuracy: 0.816 Epoch
14/15: 100%|██████████| 1875/1875 [00:52<00:00, 35.89batch/s, accuracy=0.809, loss=1.65] Epoch 14: Validation Loss: 1.64,
Validation Accuracy: 0.817 Epoch 15/15: 100%|██████████| 1875/1875 [00:55<00:00, 33.86batch/s, accuracy=0.811, loss=1.65]
Epoch 15: Validation Loss: 1.64, Validation Accuracy: 0.818 Training with batch_size=32, num_epochs=15, init_lr=0.005 Using cpu
device Epoch 1/15: 100%|██████████| 1875/1875 [00:52<00:00, 35.85batch/s, accuracy=0.592, loss=1.87] Epoch 1: Validation
Loss: 1.69, Validation Accuracy: 0.770 Epoch 2/15: 100%|██████████| 1875/1875 [00:53<00:00, 35.04batch/s, accuracy=0.779,
loss=1.68] Epoch 2: Validation Loss: 1.66, Validation Accuracy: 0.799 Epoch 3/15: 100%|██████████| 1875/1875 [00:53<00:00,
35.16batch/s, accuracy=0.826, loss=1.63] Epoch 3: Validation Loss: 1.62, Validation Accuracy: 0.840 Epoch 4/15: 100%|
██████████| 1875/1875 [00:54<00:00, 34.25batch/s, accuracy=0.838, loss=1.62] Epoch 4: Validation Loss: 1.60, Validation
Accuracy: 0.860 Epoch 5/15: 100%|██████████| 1875/1875 [00:56<00:00, 32.90batch/s, accuracy=0.85, loss=1.61] Epoch 5:
Validation Loss: 1.59, Validation Accuracy: 0.873 Epoch 6/15: 100%|██████████| 1875/1875 [00:53<00:00, 34.76batch/s,
accuracy=0.856, loss=1.6] Epoch 6: Validation Loss: 1.59, Validation Accuracy: 0.874 Epoch 7/15: 100%|██████████| 1875/1875
[00:56<00:00, 32.99batch/s, accuracy=0.863, loss=1.6] Epoch 7: Validation Loss: 1.58, Validation Accuracy: 0.878 Epoch 8/15: 100%|
██████████| 1875/1875 [00:53<00:00, 34.95batch/s, accuracy=0.867, loss=1.59] Epoch 8: Validation Loss: 1.58, Validation
Accuracy: 0.879 Epoch 9/15: 100%|██████████| 1875/1875 [00:53<00:00, 34.94batch/s, accuracy=0.873, loss=1.59] Epoch 9:
Validation Loss: 1.58, Validation Accuracy: 0.882 Epoch 10/15: 100%|██████████| 1875/1875 [00:53<00:00, 35.00batch/s,
accuracy=0.874, loss=1.59] Epoch 10: Validation Loss: 1.58, Validation Accuracy: 0.884 Epoch 11/15: 100%|██████████|
1875/1875 [00:53<00:00, 34.75batch/s, accuracy=0.878, loss=1.58] Epoch 11: Validation Loss: 1.57, Validation Accuracy: 0.893 Epoch
12/15: 100%|██████████| 1875/1875 [00:53<00:00, 34.95batch/s, accuracy=0.883, loss=1.58] Epoch 12: Validation Loss: 1.56,
Validation Accuracy: 0.896 Epoch 13/15: 100%|██████████| 1875/1875 [00:53<00:00, 34.92batch/s, accuracy=0.885, loss=1.58]
Epoch 13: Validation Loss: 1.56, Validation Accuracy: 0.898 Epoch 14/15: 100%|██████████| 1875/1875 [00:54<00:00,
34.70batch/s, accuracy=0.889, loss=1.57] Epoch 14: Validation Loss: 1.56, Validation Accuracy: 0.899 Epoch 15/15: 100%|
██████████| 1875/1875 [00:53<00:00, 34.76batch/s, accuracy=0.891, loss=1.57] Epoch 15: Validation Loss: 1.56, Validation
Accuracy: 0.902 Training with batch_size=32, num_epochs=15, init_lr=0.01 Using cpu device Epoch 1/15: 100%|██████████|
1875/1875 [00:52<00:00, 36.02batch/s, accuracy=0.225, loss=2.24] Epoch 1: Validation Loss: 1.96, Validation Accuracy: 0.501 Epoch
2/15: 100%|██████████| 1875/1875 [00:52<00:00, 35.57batch/s, accuracy=0.485, loss=1.98] Epoch 2: Validation Loss: 1.96,
Validation Accuracy: 0.505 Epoch 3/15: 100%|██████████| 1875/1875 [00:53<00:00, 35.00batch/s, accuracy=0.501, loss=1.96]
Epoch 3: Validation Loss: 1.96, Validation Accuracy: 0.496 Epoch 4/15: 100%|██████████| 1875/1875 [00:53<00:00, 35.36batch/s,
accuracy=0.521, loss=1.94] Epoch 4: Validation Loss: 1.89, Validation Accuracy: 0.571 Epoch 5/15: 100%|██████████| 1875/1875
[00:55<00:00, 33.58batch/s, accuracy=0.569, loss=1.89] Epoch 5: Validation Loss: 1.89, Validation Accuracy: 0.570 Epoch 6/15: 100%|
██████████| 1875/1875 [00:54<00:00, 34.34batch/s, accuracy=0.579, loss=1.88] Epoch 6: Validation Loss: 1.87, Validation
Accuracy: 0.587 Epoch 7/15: 100%|██████████| 1875/1875 [00:55<00:00, 33.98batch/s, accuracy=0.59, loss=1.87] Epoch 7:
Validation Loss: 1.86, Validation Accuracy: 0.602 Epoch 8/15: 100%|██████████| 1875/1875 [00:54<00:00, 34.13batch/s,
accuracy=0.597, loss=1.86] Epoch 8: Validation Loss: 1.85, Validation Accuracy: 0.608 Epoch 9/15: 100%|██████████| 1875/1875
[00:54<00:00, 34.68batch/s, accuracy=0.6, loss=1.86]
Epoch 9: Validation Loss: 1.85, Validation Accuracy: 0.612 Epoch 10/15: 100%|██████████| 1875/1875 [00:54<00:00, 34.11batch/
s, accuracy=0.604, loss=1.86] Epoch 10: Validation Loss: 1.85, Validation Accuracy: 0.614 Epoch 11/15: 100%|██████████|
1875/1875 [00:54<00:00, 34.68batch/s, accuracy=0.609, loss=1.85] Epoch 11: Validation Loss: 1.84, Validation Accuracy: 0.620 Epoch
12/15: 100%|██████████| 1875/1875 [00:55<00:00, 33.55batch/s, accuracy=0.612, loss=1.85] Epoch 12: Validation Loss: 1.84,
Validation Accuracy: 0.619 Epoch 13/15: 100%|██████████| 1875/1875 [00:55<00:00, 34.00batch/s, accuracy=0.608, loss=1.85]
```

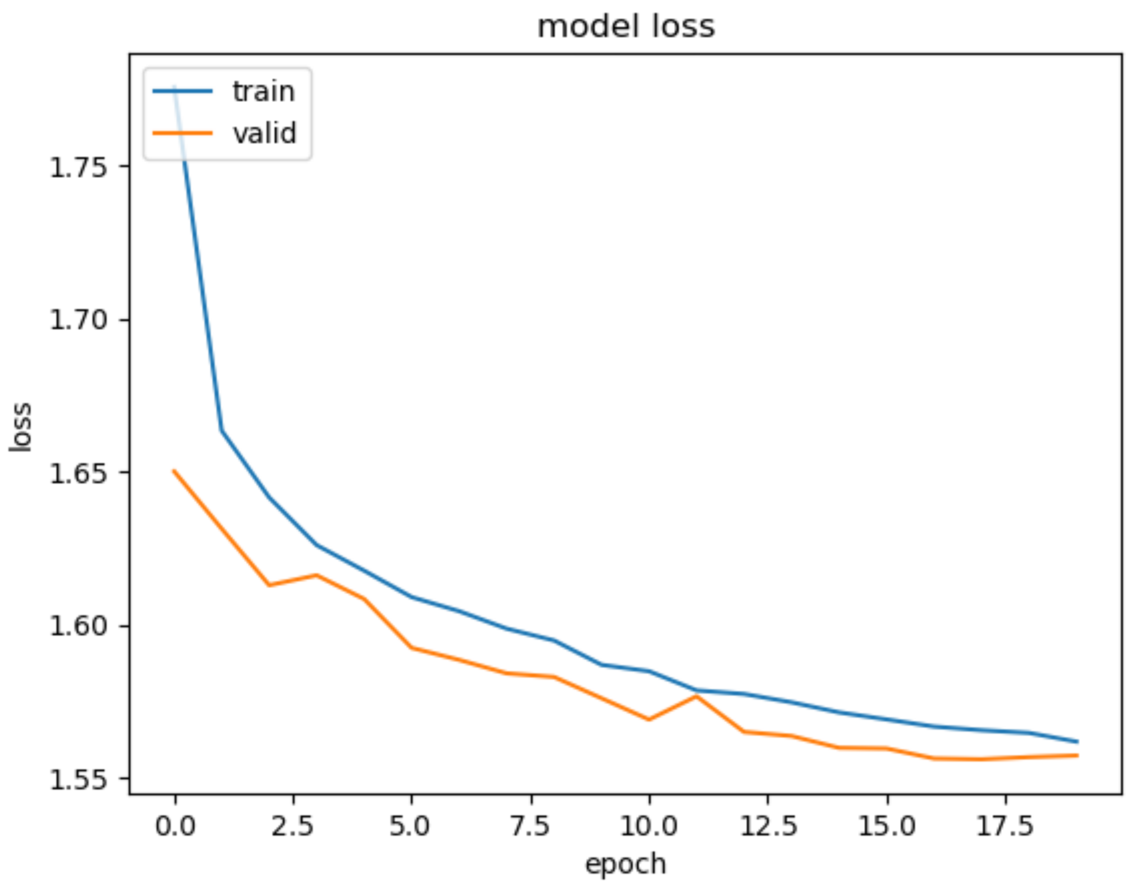
```
Epoch 13: Validation Loss: 1.85, Validation Accuracy: 0.614 Epoch 14/15: 100%|██████████| 1875/1875 [00:56<00:00,
32.94batch/s, accuracy=0.613, loss=1.85] Epoch 14: Validation Loss: 1.84, Validation Accuracy: 0.621 Epoch 15/15: 100%|
██████████| 1875/1875 [00:58<00:00, 31.87batch/s, accuracy=0.618, loss=1.84] Epoch 15: Validation Loss: 1.83, Validation
Accuracy: 0.631 Training with batch_size=32, num_epochs=20, init_lr=0.001 Using cpu device Epoch 1/20: 100%|██████████|
1875/1875 [00:53<00:00, 35.26batch/s, accuracy=0.622, loss=1.84] Epoch 1: Validation Loss: 1.74, Validation Accuracy: 0.718 Epoch
2/20: 100%|██████████| 1875/1875 [00:52<00:00, 35.56batch/s, accuracy=0.715, loss=1.75] Epoch 2: Validation Loss: 1.70,
Validation Accuracy: 0.763 Epoch 3/20: 100%|██████████| 1875/1875 [00:52<00:00, 35.76batch/s, accuracy=0.751, loss=1.71]
Epoch 3: Validation Loss: 1.68, Validation Accuracy: 0.784 Epoch 4/20: 100%|██████████| 1875/1875 [00:52<00:00, 35.61batch/s,
accuracy=0.771, loss=1.69] Epoch 4: Validation Loss: 1.67, Validation Accuracy: 0.790 Epoch 5/20: 100%|██████████| 1875/1875
[00:53<00:00, 35.28batch/s, accuracy=0.78, loss=1.68] Epoch 5: Validation Loss: 1.67, Validation Accuracy: 0.791 Epoch 6/20: 100%|
██████████| 1875/1875 [00:53<00:00, 35.34batch/s, accuracy=0.786, loss=1.68] Epoch 6: Validation Loss: 1.66, Validation
Accuracy: 0.796 Epoch 7/20: 100%|██████████| 1875/1875 [00:53<00:00, 34.78batch/s, accuracy=0.81, loss=1.65] Epoch 7:
Validation Loss: 1.61, Validation Accuracy: 0.850 Epoch 8/20: 100%|██████████| 1875/1875 [00:55<00:00, 34.05batch/s,
accuracy=0.842, loss=1.62] Epoch 8: Validation Loss: 1.60, Validation Accuracy: 0.865 Epoch 9/20: 100%|██████████| 1875/1875
[00:54<00:00, 34.58batch/s, accuracy=0.849, loss=1.61] Epoch 9: Validation Loss: 1.60, Validation Accuracy: 0.866 Epoch 10/20: 11%|
█ | 208/1875 [00:09<01:13, 22.75batch/s, accuracy=0.852, loss=1.61]
```

2.1.4 Examine accuracy and loss [2pts Grad / 0.3% Bonus for Undergrad] **[W]**

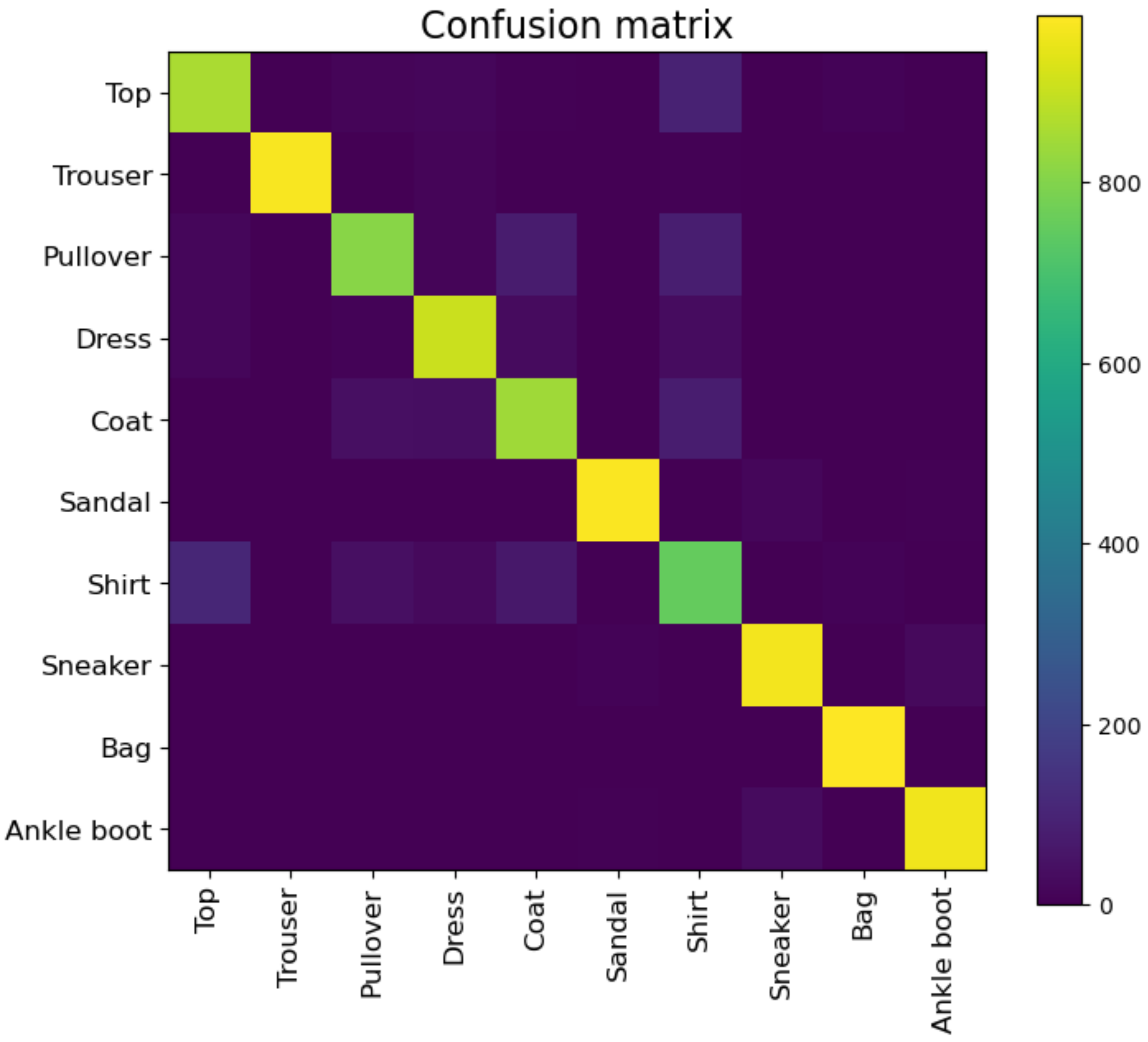
You should expect to see gradually decreasing loss and gradually increasing accuracy. Examine loss and accuracy by running the cell below, no editing is necessary. Having appropriate looking loss and accuracy plots will earn you the last 2pts for your convolutional neural net.

In [60]:
```python
################################
### DO NOT CHANGE THIS CELL ###
################################

# list all data in history
train_loss, train_accuracy, valid_loss, valid_accuracy = trainer.get_training_history()

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

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





In [61]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

# make predictions
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 accuracy_score, confusion_matrix

plt.figure(figsize=(8, 7))
plt.imshow(confusion_matrix(y_gt_classes, y_pred_classes))
plt.title("Confusion matrix", fontsize=16)
plt.xticks(np.arange(10), classes, rotation=90, fontsize=12)
plt.yticks(np.arange(10), classes, fontsize=12)
plt.colorbar()
plt.show()
```



2.2 Exploring Deep CNN Architectures [1.1% 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.)

All that's done is simply placing a threshold on the gradient's magnitude during backpropagation to prevent exceedingly large weight updates, stabilizing training.

3: Random Forests [40pts + 2.1% 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 [35pts] **[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

3.2 Hyperparameter Tuning with a Random Forest [5pts] **[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 [62]:
```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
```

```
################################
### 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 [63]:
```
################################
### 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,)
```

In the following codeblock, train your random forest model with different values for max_depth, n_estimators, or max_features and evaluate each model on the held-out test set. Try to choose a combination of hyperparameters that maximizes your prediction accuracy on the test set (aim for 85%+).

In **random_forest.py**, once you are satisfied with your chosen parameters, update the following function:

- **select_hyperparameters**: change the values for `max_depth` , `n_estimators` , and `max_features` to your chosen values

Submit this file to Gradescope. You must achieve at least a **85% accuracy** against the test set in Gradescope to receive full credit for this section.

In [64]:
```
################################
### 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!
```

In [65]:
```
"""
TODO:
n_estimators defines how many Extra trees are fitted for the random forest.
max_depth defines a stop condition when the tree reaches to a certain depth.
max_features controls the percentage of features that are used to fit each extra tree.

Tune these three parameters to achieve a better accuracy. n_estimators and max_depth must both
be at least 3 in value for moderately reliable answers. While you can use the provided test set
to evaluate your implementation, you will need to obtain 85% on the test set to receive full
credit for this section.
"""

import sklearn.ensemble
from random_forest import RandomForest
from sklearn import preprocessing

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

################# CHANGE THESE VALUES ##############################
```

```
n_estimators = 15 # Hint: Consider values between 3-15.
max_depth = 15  # Hint: Consider values betweeen 3-15.
max_features = 1.0  # Hint: Consider values betweeen 0.3-1.0.
```

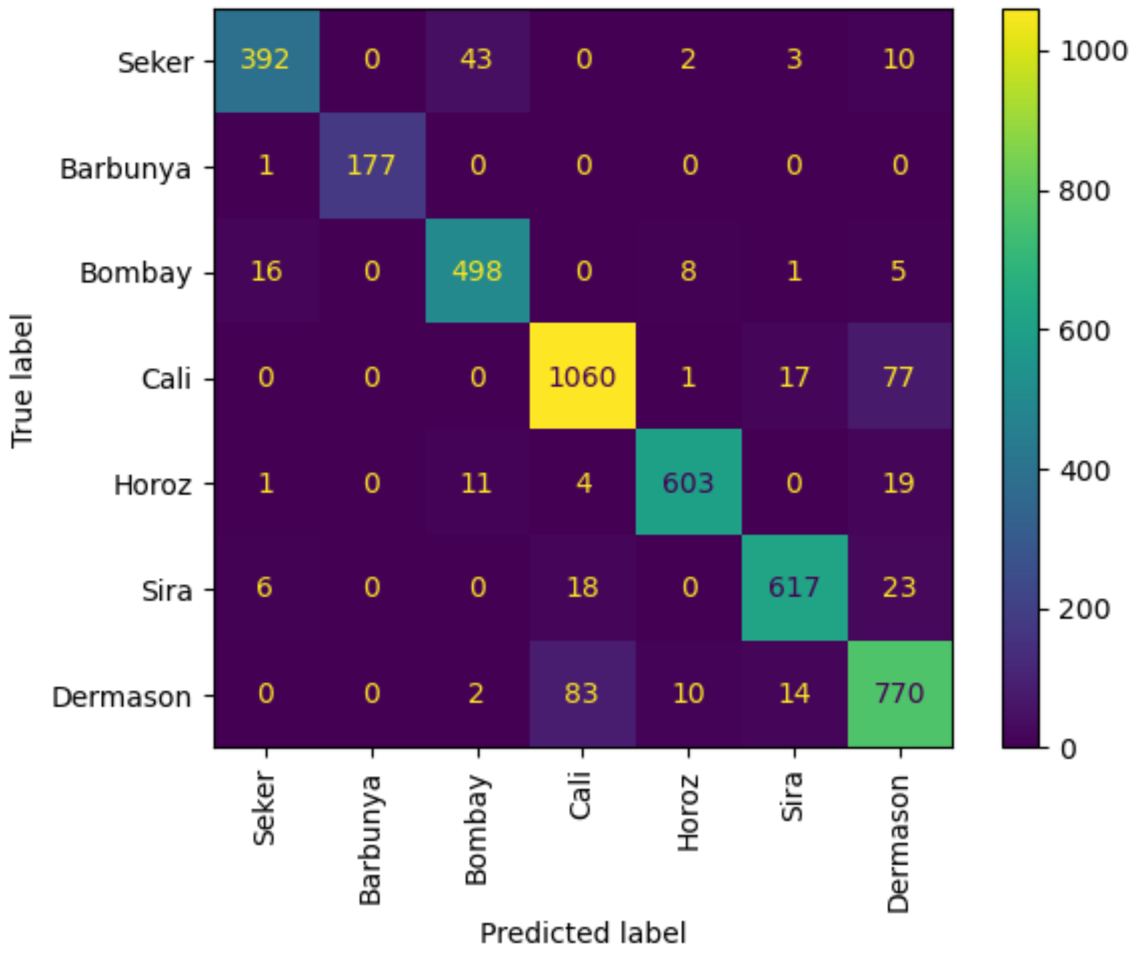
In [66]:
```
###############################
### DO NOT CHANGE THIS CELL ###
###############################
random_forest = RandomForest(
    n_estimators, max_depth, max_features, random_seed=student_random_seed
)
random_forest.fit(X_train, y_train)
accuracy = random_forest.OOB_score(X_test, y_test)
print("accuracy: %.4f" % accuracy)
```

```
accuracy: 0.8763
```

DON'T FORGET: Once you are satisfied with your chosen parameters, change the values for `max_depth` , `n_estimators` , and `max_features` in the `select_hyperparameters()` function of your RandomForest class in `random_forest.py` to your chosen values, and then submit this file to Gradescope. You must achieve at least a **85% accuracy** against the test set in Gradescope to receive full credit for this section.

Below is a code block that plots a confusion matrix for the classifier's predictions on the test set. A few things to think about: What are some trends seen in the matrix? Why do they happen?

In [67]:
```
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt

pred = 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 [1.1% 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.**

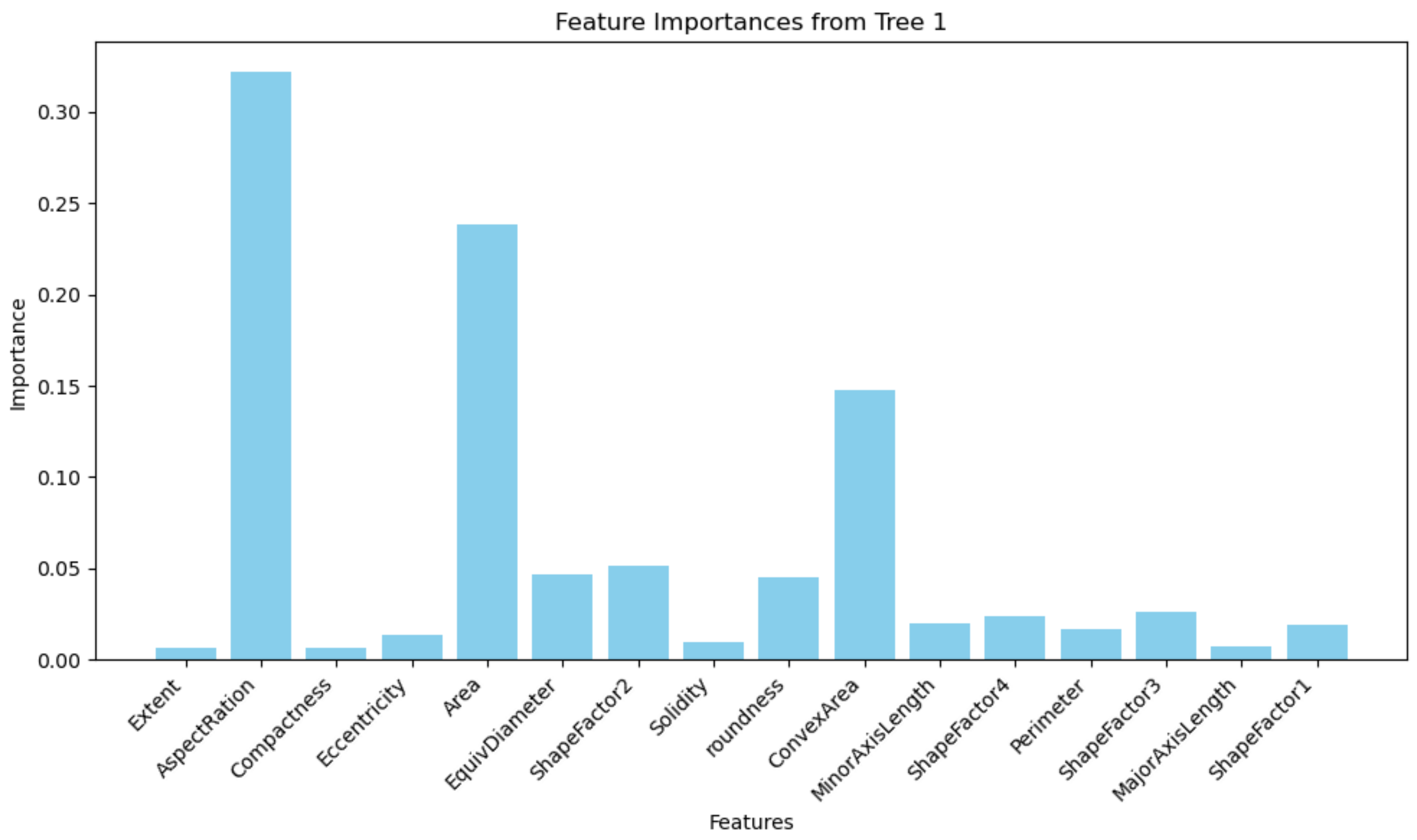
Fortunately for us, fitting a sklearn.ExtraTreeClassifier to a dataset automatically computes the Gini importance for every feature in the extra tree and stores these values in a **feature_importances_** variable. Review the docs for more details on how to access this variable

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 [69]:
```
# 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 [1% 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:

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

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
-2.3	3	-1
-1.5	-1.5	-1
0.5	0.75	-1
1.9	2.25	-1
-2.0	6.75	1
-1.1	6.0	1
0.0	7.5	1
1.5	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. Finally, you can get the needed values using `svm.coef_` and `svm.intercept_`.

In [75]:
```python
import numpy as np
from sklearn.svm import SVC

X = np.array([
    [-2.3, 3.0],
    [-1.5, -1.5],
    [0.5, 0.75],
    [1.9, 2.25],
    [-2.0, 6.75],
    [-1.1, 6.0],
    [0.0, 7.5],
    [1.5, 6.0]
])
y = np.array([-1, -1, -1, -1, 1, 1, 1, 1])

SVM = SVC(kernel='linear')

SVM.fit(X, y)

theta = np.round(SVM.coef_[0], 3)
bias = np.round(SVM.intercept_[0], 3)

print("Theta: ", theta)
print("Bias: ", bias)
```

```
Theta:  [0.111 0.622]
Bias:  -2.611
```

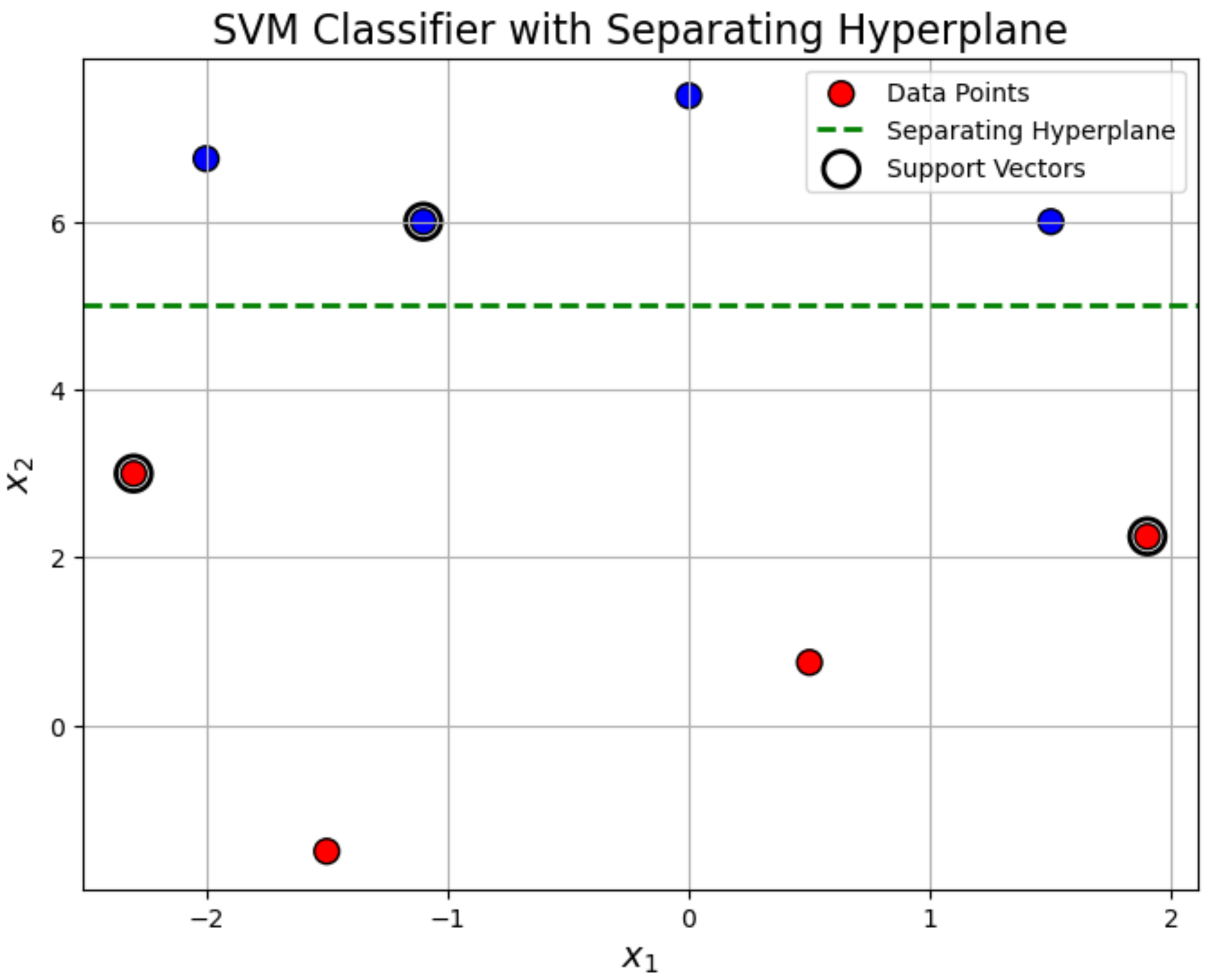
Theta: [0.111 0.622]

```
Bias: -2.611
```

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.

In [76]:
```python
import matplotlib.pyplot as plt

colors = ['red' if label == -1 else 'blue' for label in y]

plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=colors, s=100, edgecolors='k', label='Data Points')

plt.axhline(y=5, color='green', linestyle='--', linewidth=2, label='Separating Hyperplane')

plt.scatter(SVM.support_vectors_[:, 0], SVM.support_vectors_[:, 1],
            s=200, facecolors='none', edgecolors='k', linewidths=2, label='Support Vectors')

plt.xlabel('$x_1$', fontsize=14)
plt.ylabel('$x_2$', fontsize=14)
plt.title('SVM Classifier with Separating Hyperplane', fontsize=16)

plt.legend()

plt.grid(True)

plt.show()
```

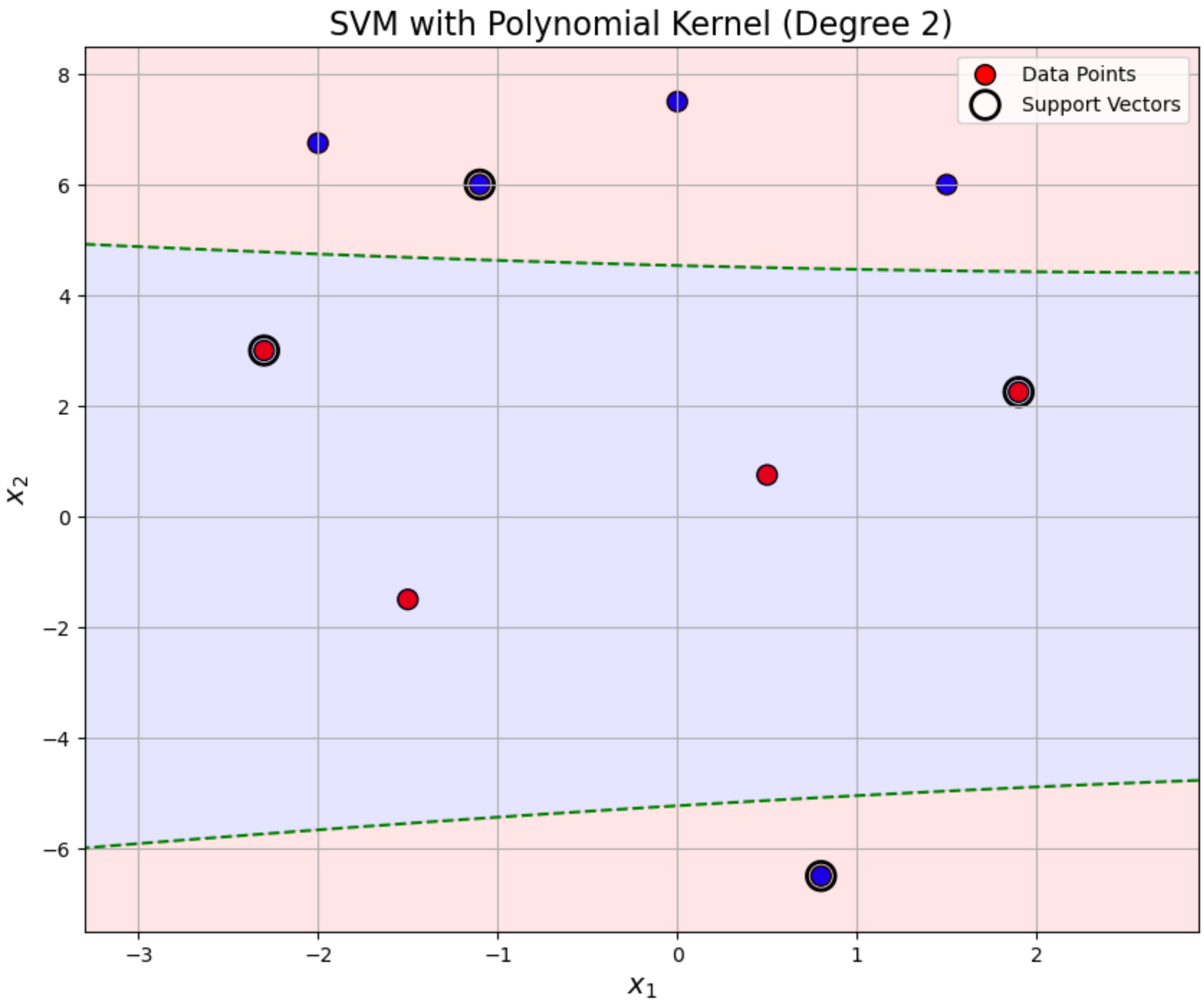


4.2 Using Kernels [5 pts] **[W]**

Suppose we added another point, (0.8, -6.5) with label 1, which causes the dataset to be non-linearly separable. This can be seen by plotting the new set of points, which is impossible to split by a hyperplane that achieves 100% accuracy. To solve this issue, SVM uses kernels, which transform the points in a dataset to a higher-dimensional space making them linearly separable. In section 4.1, you used a linear kernel, which didn't transform the dataset since it was already linearly separable.

Create a specific kernel (transformation function) that could be applied to the dataset with the new point that makes the data linearly separable again.

In [79]:
```
X = np.array([
    [-2.3, 3.0],
    [-1.5, -1.5],
    [0.5, 0.75],
    [1.9, 2.25],
    [-2.0, 6.75],
    [-1.1, 6.0],
    [0.0, 7.5],
    [1.5, 6.0],
    [0.8, -6.5]
])
y = np.array([-1, -1, -1, -1, 1, 1, 1, 1, 1])

svm_poly = SVC(kernel='poly', degree=2, coef0=1, C=100)
svm_poly.fit(X, y)

colors = ['red' if label == -1 else 'blue' for label in y]

plt.figure(figsize=(10, 8))
plt.scatter(X[:,0], X[:,1], c=colors, s=100, edgecolors='k', label='Data Points')

xx, yy = np.meshgrid(np.linspace(X[:,0].min()-1, X[:,0].max()+1, 500),
                     np.linspace(X[:,1].min()-1, X[:,1].max()+1, 500))
Z = svm_poly.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

contour = plt.contour(xx, yy, Z, levels=[0], colors='green', linestyles='--')
plt.contourf(xx, yy, Z > 0, alpha=0.1, colors=['blue', 'red'])


plt.scatter(svm_poly.support_vectors_[:,0], svm_poly.support_vectors_[:,1],
            s=200, facecolors='none', edgecolors='k', linewidths=2, label='Support Vectors')

plt.xlabel('$x_1$', fontsize=14)
plt.ylabel('$x_2$', fontsize=14)
plt.title('SVM with Polynomial Kernel (Degree 2)', fontsize=16)
plt.legend()
plt.grid(True)
plt.show()
```


SVM with Polynomial Kernel (Degree 2)
Data Points
Support Vectors
x_1
x_2
−3
−2
−1
0
1
2
8
6
4
2
0
−2
−4
−6

5: Next Character Prediction using Recurrent Neural Networks (RNNs) [6.8% Bonus for All] [W] [P]

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.

What are Recurrent Neural Networks?

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)

Types of Recurrent Neural Networks

We can divide the applications of RNNs into four main categories:

- **One-to-One**
    - Takes a single input and produces a single output
    - Like feedforward neural networks
    - Example: Image classification (1 image → 1 label)
- **Many-to-One**

    - Takes a sequence as input but produces a single output
    - Example: Sentiment analysis (many words → 1 sentiment)
    - *This is what we'll be building for our character prediction task!*
- **One-to-Many**
    - Takes a single input and produces a sequence
    - Example: Image captioning (1 image → sequence of words)
- **Many-to-Many**
    - Takes a sequence and produces a sequence
    - Examples:
        - Translation (English sentence → French sentence)
        - Video frame prediction (sequence of frames → next frames)

Each type serves different purposes, and understanding which to use depends on your task's input and desired output structure.

In this section, our goal will be a **many-to-one** task: predicting the next character given a sequence of characters. Now let's take a look at the models we can use to accomplish this.

Simple Recurrent Neural Networks

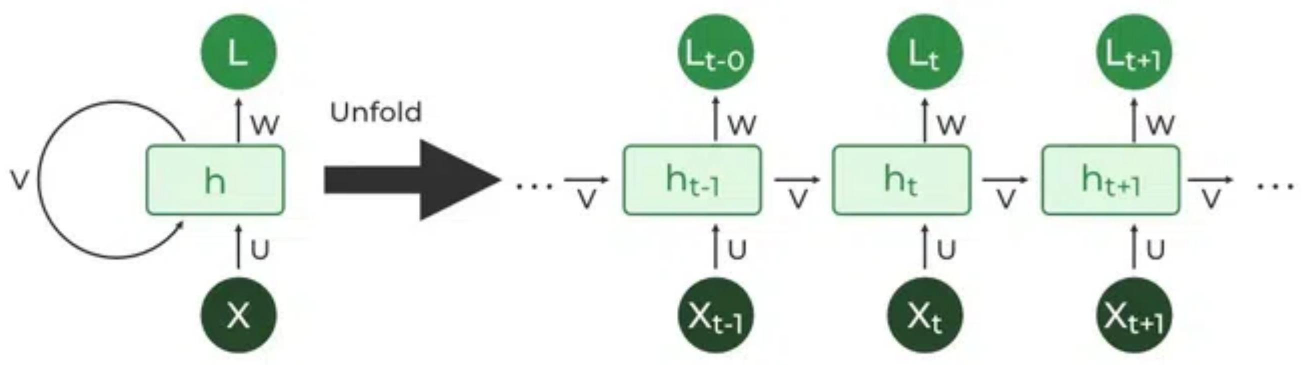
Simple Recurrent Neural Networks (RNNs) are the simplest form of recurrent neural networks. They are similar to feedforward networks, but have connections that loop back on themselves, allowing information from previous steps to influence the current processing step. This looping mechanism enables Simple RNNs to capture dependencies over time, which is especially useful for tasks involving language, where the context of earlier words or characters affects the final prediction. Here's how Simple RNNs generally work:

1. At each timestep, the model:
    - Takes in a new input
    - Uses its current hidden state (memory of previous inputs)
    - Produces both an output and an updated hidden state
2. This process repeats for each element in the sequence, with the hidden state carrying information forward.

For our character prediction task, we'll use a many-to-one configuration:

- Input: We feed in characters one at a time (like "m", "a", "c", "b", "e")
- Hidden State: Gets updated with each new character
- Output: We only use the final prediction to guess the next character ("t")

Architecture



Simple RNNs often suffer from the vanishing gradient problem when performing backpropagation through time. This issue arises because the gradients are repeatedly multiplied by the weight matrix as they propagate through each time step. If the weights are small, these multiplications cause the gradients to decrease exponentially, effectively "vanishing" as they travel backward, while large weights can lead to gradient "explosion." As a result, the network struggles to learn and retain long-term dependencies, meaning that early inputs in a sequence are gradually "forgotten," and training becomes ineffective for tasks requiring the processing of lengthy sequences.

For a detailed explanation of Simple RNNs and their limitations, see:

- An Introduction to Recurrent Neural Networks (RNNs)

- The Vanishing Gradient Problem in RNNs
- Josh Starmer RNNs clearly explained

Note: Simple/Vanilla RNN and RNN are often used interchangeably since Simple RNNs represent the most basic implementation of RNNs. However, RNN is actually a broader term that encompasses any neural network model which incorporates hidden states and uses its previous outputs as inputs.

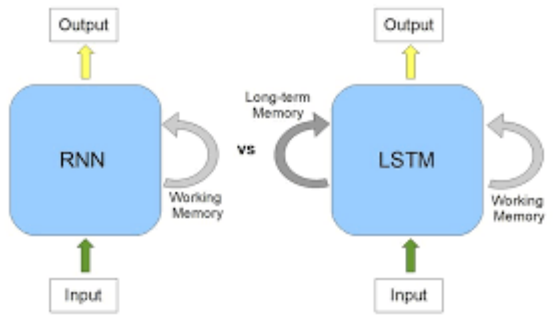
Long Short-Term Memory Networks (LSTMs)

Long Short-Term Memory networks (LSTMs) are an advanced type of recurrent neural network designed to better handle long sequences. Unlike Simple RNNs that have a single memory channel, LSTMs have two memory channels that allows the network to selectively remember or forget information over long sequences. At each timestep, an LSTM:

- Takes in a new input
- Uses two types of state:
  - Cell State: Like a long-term memory that can preserve important information
  - Hidden State: Like a working memory for immediate processing
- Uses three gates to control information flow:
  - Forget gate: Decides what to remove from cell state
  - Input gate: Decides what new information to store from the current input
  - Output gate: Decides what parts of cell state to output

This dual-memory system allows LSTMs to maintain important information for longer periods while still processing new inputs effectively. Think of it like reading a book while taking notes - the cell state is like your notebook where you write down important information, while the hidden state is like your immediate attention to the current word and its context.

Architecture



For our character prediction task, we'll use a many-to-one LSTM configuration:

- Input: Characters fed in one at a time (like "m", "a", "c", "b", "e") as input
- Output: Get the prediced next character ("t") using the last hidden state

Simple RNNs struggle with long sequences because they try to pass all information through a single channel that gets increasingly noisy over time. LSTMs solve this by having two channels and specialized gates that allow the network to selectively update its memory state, rather than forcing all information through a single transformation. The memory cell pathway solves the vanishing gradient problem by providing a direct route for gradients to propagate backwards through time steps without being repeatedly multiplied by small numbers that would cause them to vanish. The gating mechanisms ensure this pathway remains relatively stable while still allowing the network to selectively update its memory when needed.

For a deeper dive into LSTMs and their gate mechanisms, refer to:

- Understanding LSTM Networks
- The Role of Gates in LSTMs
- Josh Starmer LSTM clearly explained

Data Preparation

Loading the text

- We'll use Shakespeare's Macbeth from Project Gutenberg
- The text is first standardized by:
    - Converting to lowercase for consistency

    - Standardizing line endings and spacing
- This standardized text is referred to as the 'corpus' - our primary dataset for training and inference
- We vectorize the text by treating every character in our text as an individual unit (e.g., 'macbeth' -> ['m', 'a', 'c'...])

Character-to-Index Mapping

- Neural networks can't process raw text, so we need to convert characters to numbers.
- First, we find all the unique characters in our text (including alphabets and special characters like spaces and punctuations). This forms our vocabulary, whose size we call the vocabulary size
- Each character in the vocabulary receives a unique integer assignment using the default ASCII value
- 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 use this mapping to convert the vector of text into a vector of numerical values.

Index Representation

- You might be thinking that assigning numbers to letters based on alphabetical order seems rather arbitrary - and you'd be absolutely right! This is precisely why we need smarter index representations

- Using raw indices (like 1, 2, 3) implies an ordering and magnitude that doesn't exist

- For example, 'b' (2) is not "twice as much" as 'a' (1)

- We need a more sophisticated way to represent characters that avoids these misleading numerical relationships

  a) **One-Hot Encoding**: Convert index to binary vector

    - Each index becomes a vector of zeros with a single 1 at the index of the character's value/index
    - Example:
      - index 1 ('a') → [0,1,0,0,...,0] (length 40)
      - index 5 ('e') → [0,0,0,0,1,0,...,0] (length 40)
    - Sparse (mostly zeros)
    - No relationship between characters (both vowels 'a' and 'e' are completely different vectors)

  b) **Embeddings**: Convert index to learned dense vector

  - Each character's index gets transformed into a list of numbers (a dense vector)
  - Think of it like assigning each character a unique point a higher dimension coordinate plane
  - The motivation is that we can position similar characters close to each other and transform discrete data like letters or words into a continous vector space that the model can use to make predictions from
  - What makes embedding layers special is that they're learnable! The model figures out the best way to represent characters based on your specific data and task
  - This is super useful since what makes characters "similar" changes between tasks - the model learns what similarity means for your specific case and groups characters accordingly
  - The embedding layer's dimension parameter determines how many numbers we use to represent each character (the size of the vector space)
  - Let's look at a real example with embedding dimension 4:
    - The letter 'a' might become → [0.2, -0.5, 0.1, 0.8]
    - The letter 'e' might become → [0.3, -0.4, 0.2, 0.7]
  - In this example, model discovered that 'a' and 'e' are both vowels and represents them close to each other
  - Additionally, embeddings are more memory efficient because they are dense (non-sparse) unlike one-hot vectors
- For our model, we will be using an embedding layer in the model for index representations. Since this is a learnable layer, it will be part of our model architecture and not pre-processing.

Creating training data

- When preparing text data, we need to break it into fixed-length sequences for training. Each sequence becomes a training example where we try to predict the next character.
- Using a sliding window approach, we move through the text step by step. The sequence length (context window) is a key hyperparameter - too short and the model lacks context, too long and training becomes computationally intensive. For example, with text "macbeth" (context window=4):
```
Window 1: "macb" → predict "e"
Window 2: "acbe " → predict "t"
Window 3: "cbet" → predict "h"
```
- Our input data is size (`NUM_SEQUENCES` , `SEQUENCE_LEN`), where `NUM_SEQUENCES` is the number of sequences of size `SEQUENCE_LEN` in the text
- Our target data is size (`NUM_SEQUENCES` , `1`) representing all the next characters for all sequences in the text
- NOTE: While we show characters in the examples above for clarity, remember that the actual input and target data are

numerical indices, not the characters themselves

Please refer to **preprocess_text_data** located in utilities 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}")
```

Data Structure

Final preprocessed data includes:

- **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

5.1 Model Architecture [4.4% Bonus for All] [P]

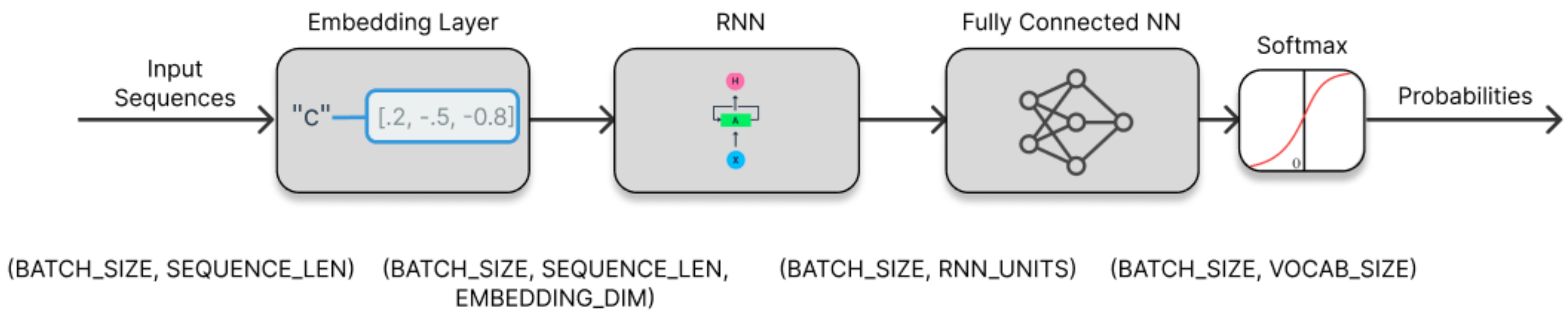
Before diving into the specific architectures, let's understand how data shapes transform through the embedding layer.

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 [2.2% 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

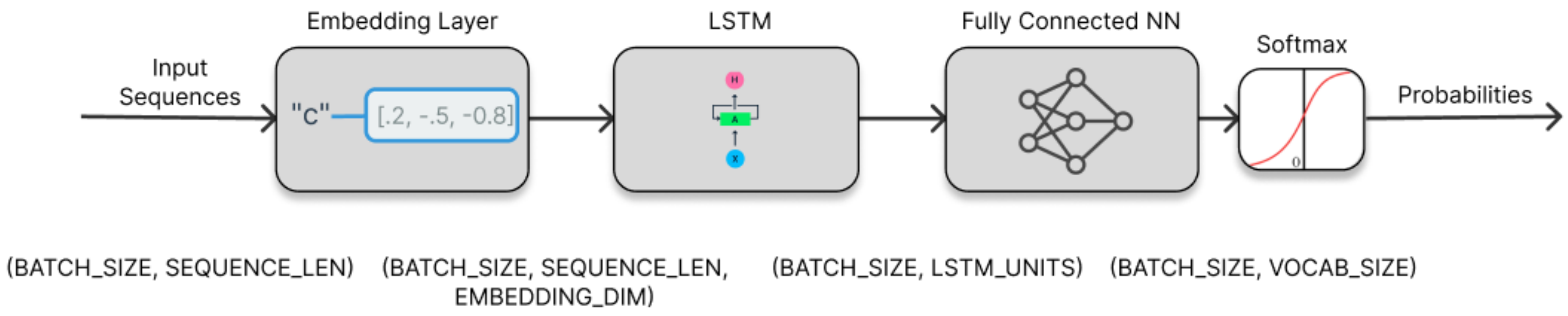
TODO: Implement the **define_model** function in **rnn.py**.

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 [2.2% 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 [2.4% Bonus for All] [W]

Next, we'll train our RNN and LSTM models with their respective hyperparameters for character-level text generation. The model architecture is designed to process fixed-length sequences and predict the next most likely character based on the learned patterns.

During training, we need one final transformation of our target data.

- As shown in both architectures, the model outputs a probability distribution over all possible characters (VOCAB_SIZE)
- To compare this with our target data, we convert our target indices into one-hot vectors
- For example, if our target is 'e' or 5 and VOCAB_SIZE is 34:
    - Target index: `[5]` → One-hot: `[0, 0, 0, 0, 1, 0, ..., 0]` (length 34)
- This transformation happens in the training loop, converting Y from shape (BATCH_SIZE, 1) to (BATCH_SIZE, VOCAB_SIZE)
- The categorical cross-entropy loss can then compare our predicted probability distribution with this one-hot target distribution

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` .

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 `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 `model_weights` directory

Training Visualization

To monitor the training progress:

- A plot showing loss metrics across epochs will be generated
- This helps identify potential overfitting or convergence issues
- The visualization will be displayed automatically after training completes

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()
```

After training both models, we can now use them to generate text in the style of Macbeth. The generation process involves three key components:

1. Seed Selection
    - Start with a random seed sequence from our text corpus
    - This sequence provides initial context for the generation
2. Prediction Loop
    - Model predicts probability over all characters based on current sequence and we sample next character
    - Append generated character to the sequence
    - Slide window forward by one character to maintain context length and predict using generated character
3. Temperature Parameter
    - Controls the randomness of predictions:
        - Lower values (< 0.5): More conservative, deterministic text
        - Higher values (> 1.0): More diverse but potentially less coherent text
        - Default value of 0.5 balances coherence and creativity

Below, we use our TextGenerator class to generate samples from both models:

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

```python
from text_generator import TextGenerator

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

In [ ]:
```python
###############################
### 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)
```

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 of these aspects from your training:

A. Generated Text Quality

- What patterns and quality differences did you observe in the text output?

B. Training Duration

- How long did the model take to train?

C. Loss Convergence

- How did the loss values change across training epochs?

D. Final Loss Achieved

- What was the final loss produced by the model?

You can research and add other considerations to strengthen your argument, but ensure you include atleast 1 observed metric from your training. Some additional considerations include:

- Inference speed
- Memory requirements
- Ability to maintain context
- Simplicity of architecture

You will receive full credit if you use evidence from training and make a sound argument about why one model would be more suitable than the other for both cases.

Sample Response Format

A. Simple RNN Best Use Case: [Application Name]

- Choose 1+ observed metrics:
    - [Metric name]: [What you observed in your training]
- Why this matters: [Explain how your evidence supports using RNN over LSTM for this application]

B. LSTM Best Use Case: [Application Name]

- Choose 1+ observed metrics:
    - [Metric name]: [What you observed in your training]
- Why this matters: [Explain how your evidence supports using LSTM over RNN for this application]

Optional, but recommended: Add theoretical considerations to strengthen your argument

- [Theoretical difference & why it matters for your chosen application]

YOUR ANSWER HERE

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