# HW4

November 28, 2025

## 1 Fall 2025 CS 4641/7641 Homework 4

### 1.1 Instructor: Dr. Mahdi Roozbahani, Dr. Nimisha Roy

### 1.2 Deadline: Tuesday, December 2, 2025 11:59 pm EST

#### 1.2.1 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.

- No unapproved extension of the deadline is allowed. For late submissions, please refer to the course website.
- Discussion is encouraged on Ed as part of the Q/A. We encourage whiteboard-level discussions about the homework. **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, and you must not collaborate with anyone or share your HW content except the ML instructional team.
- Working with a generative-AI platform *may constitute plagiarism*. In line with the Joyner Heuristic being used by many classes at GT, you should treat collaboration with generative-AI as collaboration with a knowledgeable peer. **Sharing the question or your work verbatim with an AI agent so as to generate an answer to the question is considered academic dishonesty and will be treated the same as any other incident of academic dishonesty.** If you find yourself turning to generative-AI for help answering a question, we suggest that you instead make use of Ed or TA office hours.
- Even using generative-AI for formatting your answers is *not permitted*. **While we understand that LaTeX has a learning curve, you may not use any generative-AI platform to improve your writing or LaTeX formatting.** These tools inherently produce output that may include a partial or complete solution to the question, including corrections to work you give it, even if purely prompted for syntactic use. Additionally, you have no custody over the data you give these platforms, which may leak or be used to inform subsequent training. Thus, while you are more than welcome to ask it about LaTeX commands and formatting tips, sharing the question or your answer to a question verbatim with an AI agent, even purely for syntactic use, is considered academic dishonesty and will be treated the same as any other incident of academic dishonesty. If you find yourself turning to generative-AI for help rewording your work to improve the language therein, remember that many of

these questions will not be graded on language, but the content of your work. If you wish to improve it nonetheless, you can make use of the Georgia Tech Communications Lab or TA office hours. If you find yourself turning to generative-AI for help reformatting your LaTeX, we suggest that you instead use the resources in the instructions below or TA office hours. Ed is also an appropriate place to ask about LaTeX formatting, so long as your post doesn't reveal answers to a question (or make a private post if your question necessitates revealing answers).

- *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 the case to OSI, which may, unfortunately, lead to a very harsh outcome, pending review. Consequences can be severe, including academic probation or dismissal, grade penalties, a 0 grade for assignments concerned, and prohibition from withdrawing from the class.**

### 1.3 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.
- We will be using Gradescope for submission and grading of assignments.
- **Unless a question explicitly states that no work is required to be shown, you must provide an explanation, justification, or calculation for your answer.** Basic arithmetic can be combined (it does not need to each have its own step); your work should be at a level of detail that a TA can follow it.
- **For the "Non-programming" turn-in of this assignment, you will need to submit to Gradescope a PDF copy of your Jupyter Notebook with the cells ran.** Please refer to the Deliverables and Point Distribution section for an outline of the non-programming questions.
- When submitting your assignment on Gradescope, **you are required to correctly map pages of your PDF to each question/subquestion to reflect where your solutions appear in your PDF.** For written questions, you should assign every cell relevant to the subquestion, including code cells that demonstrate your code and/or markdown cells where you write an answer. If you're unsure, assign every cell contained in the subquestion. **Improperly mapped questions will receive a 0.** You are permitted to submit a regrade request in this event; however, the review of such a request is at the sole discretion of the instructional staff and is not likely to be accepted.
- **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" (landscape or portrait), otherwise there may be a deduction in points for oversized sheets, up to and including full credit deducted.** Again, you are permitted to submit a regrade request in this event; however, the review of such a request is at the sole discretion of the instructional staff and is not likely to be accepted.

- All assignments should be done individually; each student must write up and submit their own answers.
- **Graduate Students**: You are required to complete any sections marked as Bonus for Undergrads.

### 1.3.1 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. The code you should complete will be contained in marked functions.
- 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
- 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.

### 1.3.2 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.

### 1.3.3 Submission Instructions

Do not add or remove imports from any files, as this will cause the autograder to fail to parse your solution.

For actual submission to Gradescope, upload the following files:

- **Assignment 4 Programming**

  - NN.py
  - cnn_image_transformations.py
  - cnn.py
  - cnn_trainer.py

- **Assignment 4 Programming - Bonus for All**
  - rnn.py
  - lstm.py
  - base_sequential_model.py (you don't need to modify this file but need to submit it)

- **Assignment 4 Programming - Bonus for Undergrad**
  - NN.py
  - cnn_image_transformations.py
  - cnn.py
  - cnn_trainer.py

- **Assignment 4 Non-Programming**
  - You will need to submit to Gradescope a PDF copy of your Jupyter Notebook with the cells ran. Please refer to the **Deliverables and Point Distribution** section for an outline of the non-programming questions. Complete the notebook, then use some tool to convert this notebook into a pdf.
  - It is your responsibility to make sure that all LaTeX renders, all generated matplotlib figures render, none of your answers are clipped, the font is a reasonable size, the pdf is a reasonable resolution, the pdf is the correct size (8.5"x11"). Failure to do so may result in heavy penalties.
  - It is also your responsibility to correctly assign pages to the relevant subquestions. If you are unsure what is being graded, to be safe, you may submit any page containing content for the entire subquestion.

## 1.4 Deliverables and Points Distribution

### 1.4.1 Q1: Classification with Two Layer NN [89pts: 64pts + 25pts Grad / 3.6% Bonus for Undergrad]

**Deliverables: NN.py**

- **1.1 NN Implementation** [57pts: 47pts + 10pts Grad / 1.6% **Bonus for Undergrad**] - *programming*, *non-programming*
  - 1.1.1 Weight Initialization [2pts]
  - 1.1.2 Softmax [2.5pts]
  - 1.1.3 Softsign [2.5pts]
  - 1.1.4 Dropout [5pts]
  - 1.1.5 Cross Entropy loss [2.5pts]
  - 1.1.6 Forward propagation with and without dropout [10pts]
  - 1.1.7 Compute gradients [7.5pts]
  - 1.1.8 Update gradients without Adam [5pts]
  - 1.1.9 Update gradients with Adam [5pts / 0.8% **Bonus for Undergrad**]
  - 1.1.10 Backward [5pt]

  - 1.1.11 Gradient Descent [2.5pt]
  - 1.1.12 Mini-batch Gradient Descent [2.5pts]
  - 1.1.13 Gradient Descent with Adam [2.5pts / 0.4% **Bonus for Undergrad**]
  - 1.1.14 Mini-batch Gradient Descent with Adam [2.5pts / 0.4% **Bonus for Undergrad**]
- **1.2 Training with Gradient Descent** [9.5pts] - *programming*, *non-programming*
  - 1.2.1 Loss plot and cross-entropy (CE) value [7.5pts]
  - 1.2.2 Learning Rates [2pts]
- **1.3 Training with Mini-batch Gradient Descent** [7.5pts] - *programming*
- **1.4 Training with Gradient Descent and Adam** [7.5pts Grad / 1% **Bonus for Undergrad**] - *programming*
- **1.5 Training with Mini-batch Gradient Descent and Adam** [7.5pts Grad / 1% **Bonus for Undergrad**] - *programming*

#### 1.4.2 Q2: Image Classification based on CNNs [26pts: 10pts + 16pts Grad / 2.4% Bonus for Undergrad + 2.5% Bonus for All]

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

- **2.1 Image Classification using Pytorch CNN** [26pts: 10pts + 16pts Grad / 2.4% **Bonus for Undergrad**] - *programming*
  - 2.1.1 Data Augmentation [5pts]
  - 2.1.2 Building the Model [5pts]
  - 2.1.3 Training and Tuning the Model [12pts Grad / 1.8% **Bonus for Undergrad**]
  - 2.1.4 Examining Loss Plots [2pts Grad / 0.3% **Bonus for Undergrad**]
  - 2.1.5 Evaluating Confusion Matrix [2pts Grad / 0.3% **Bonus for Undergrad**]
- **2.2 Exploring Deep CNN Architectures** [2.5% **Bonus for All**] - *non-programming*
  - 2.2.1 Abating Vanishing Gradients [1.5% Bonus for All]
  - 2.2.2 Abating Internal Covariate Shift [1.0% Bonus for All]

#### 1.4.3 Q3: SVM [20 pts]

**Deliverables: svm.py and Written Report**

- **3.1 Picking Performant Constructions** [5pts] - *non-programming*
- **3.2 Custom Feature Engineering** [5pts] - *programming*
- **3.3 Kernel Trick** [10pts]
  - 3.3.1 Build a Kernel [5pts] - *programming*
  - 3.3.2 Build a Known Kernel (RBF) [5pts] - *programming*

#### 1.4.4 Q4: Next Character Prediction using Recurrent Neural Networks (RNNs) [7.5% Bonus for all]

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

- **4.1: Model Architecture** [5% **Bonus for All**] - *programming*
  - 4.1.1: Defining the Simple RNN model [2.5% **Bonus for All**]
  - 4.1.2: Defining the LSTM model [2.5% **Bonus for All**]
- **4.2: Simple RNN vs LSTM Model Text Generation Training Comparison Analysis** [2.5% **Bonus for All**] - *non-programming*

#### 1.4.5 Point Totals

- Total Base: 135pts for grads / 94pts for undergrads
  - Programming: 126pts for grads / 85pts for undergrads
  - Written: 9pts
- Total Undergrad Bonus: 6%
  - Programming: 6%
  - Written: 0%
- Total Bonus for All: 10%
  - Programming: 5%
  - Written: 5%

### 1.5 Environment Setup

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

import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import requests
import torch
from cnn import CNN
from cnn_image_transformations import (
    TransformedDataset,
    create_testing_transformations,
    create_training_transformations,
)
from sklearn.model_selection import train_test_split
from utilities.utils import get_housing_dataset, get_mri_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: 3.12.12 | packaged by conda-forge | (main, Oct 22 2025, 23:13:34) [MSC
v.1944 64 bit (AMD64)]
matplotlib: 3.10.8
numpy: 2.3.4
```

## 1.6 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.

### 1.6.1 Measuring Our Impact:

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

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

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

[2]:
```
from codecarbon import EmissionsTracker

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

```
[codecarbon WARNING @ 14:44:56] Multiple instances of codecarbon are allowed to
run at the same time.
```

## 1.7 Q1: Two Layer Neural Network [89 pts: 64pts + 25pts Grad / 3.6% Undergrad Bonus] [P] | [W]

### 1.7.1 Neural Net Recap

A perceptron can be thought of as a linear hyperplane, as seen in logistic regression and SVM, followed by a non-linear activation function, warping ths space, allowing the final decision to capture non-linear patterns. If we have a $D$-dimensional input vector $x \in \mathbb{R}^D$ (one datapoint with $D$ features), a perceptron is defined by a raw activation $u$, the activation function $\phi : \mathbb{R} \to \mathbb{R}$, and the ouput $o$. To make the linear hyperplane, we weight each feature by some value $\theta_j$ and add on a bias at the end $b$.

$$u = \theta^T x + b$$
$$o = \phi(u) = \phi\left(\sum_{j=1}^{D} \theta_j x_j + b\right)$$

There's nothing stopping us from making more perceptrons, though, each with their own weight vectors. We could have a whole output layer $o$:

$$o_i = \phi(u_i) = \phi\left(\sum_{j=1}^{D} \theta_{ij} x_j + b_i\right)$$

Now, $\theta$ is a matrix [input dims] x [number of output perceptrons], and $b$ is a vector [input dims]. This can indeed be thought of as a matrix operation, where we apply phi to every component of the output:

$$o_i = \phi\left(\theta\vec{x} + \vec{b}\right)$$

The key insight behind neural networks is to use the output layer as a new input layer, and stack this operation! If our single layer can capture a few patterns, multiple layers should be able to capture very complex patterns.

Typically, a modern neural network contains millions of perceptrons. In this part, we describe a fully connected layer in a neural network which comprises multiple perceptrons in every 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 $\theta_i$ 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.

As for the activation function, it really just needs to be non-linear. Some functions are preferred in some places, sometimes due to their gradient, sometimes due to some statistical guarantees for a theoretical analysis, but really, your choice of activation function is just another hyperparameter to throw in the hyperparameter soup.

There are many activation functions that are used for various purposes. For this question, we use softsign and the softmax activation functions. In your project, or personal work, we encourage you to explore the plethora of options.

The one activation function that you need to be really careful with is the last one. If you want your network to output a probability distribution, you should make the activation function map to the range [0,1] and divide by the sum (to make it sum to 1). If you want your network to output a non-negative number, you should make the activation function map to the range (0,inf), maybe exp or relu. This will determine the outputs your network can possibly learn, so make sure all of your training targets are in the range of your final activation.

### 1.7.2 1.1 NN Implementation [57pts: 47pts + 10pts Grad / 1.6% Bonus for Undergrad] [P] | [W]

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 - Softsign and Softmax. You will implement a neural network where the first hidden layer uses a Softsign activation and the output layer uses Softmax.

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

- softmax
- softsign
- derivative_softsign
- _dropout
- cross_entropy_loss
- forward
- compute_gradients
- update_weights
- backward
- gradient_descent
- minibatch_gradient_descent

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

**1.1.1 Weight Initialization** **[W]** Please assign all pages containing your answer to the question.

Weight initialization is the process of setting the initial values of the weights and biases of a neural network prior to training. We have already implemented the weight initialization for this neural network. Specifically, we used a variant of the Xavier initialization, which can be defined as:

$$W = \frac{randn(n_{in}, n_{out})}{\sqrt{n_{in}}}$$

Read these two sources about Xavier initialization. - Geeks for Geeks Summary - Original paper

Explain what the goal of Xavier initialization is and how it helps in neural network training. Then, briefly describe how the variant we provided reflects the same idea, even though it uses a slightly different formula.

**YOUR ANSWER HERE**

---

The goal of Xavier initialization is to prevent gradients from vanishing or exploding by keeping the variance of activations and gradients consistent across all layers of the network. This stability ensures that signals propagate effectively during training without driving activation functions into saturation, which would stall the learning process. Our variant similarly scales the initial random weights by the square root of the number of input units ($1/\sqrt{n_{\text{in}}}$), normalizing the variance to maintain a stable signal during the forward pass.

---

**1.1.2 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.

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

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

```
test_softmax passed!
```

**1.1.3 Softsign** The Softsign activation function, is defined as:

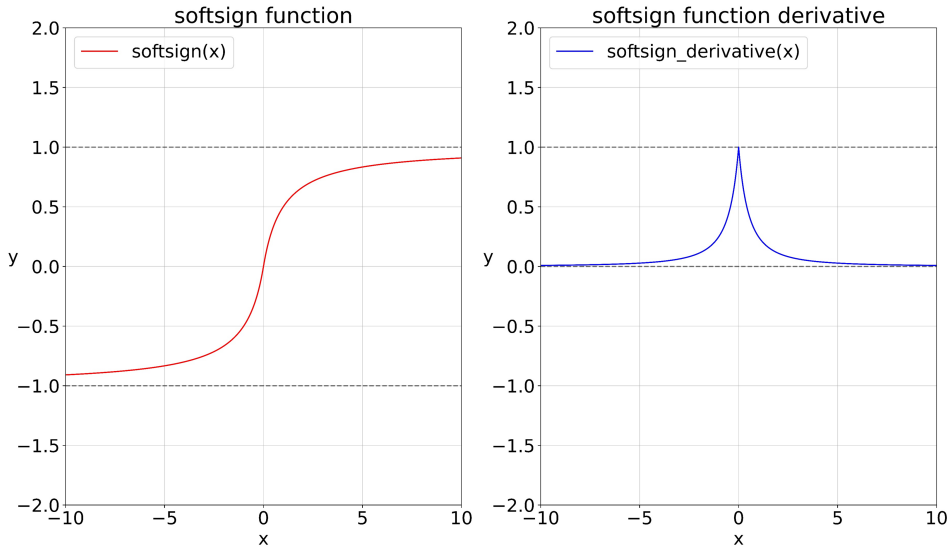
$$o = \phi(u) = \frac{u}{1 + |u|}$$

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

$$\phi'(u) = \frac{1}{(1 + |u|)^2}$$

Unlike ReLU, Softsign is a smooth and non-linear activation function that gradually approaches $-1$ and 1 for large negative and positive inputs, respectively. This smoothness helps stabilize training by reducing sharp gradient changes and improving gradient flow compared to ReLU.

In this homework, we implement Softsign.



TODO: Implement the function softsign and derivative_softsign in NN.py.

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

TestNN("test_softsign").test_softsign()
TestNN("test_d_softsign").test_d_softsign()
```

```
test_softsign passed!
test_d_softsign passed!
```

**1.1.4 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 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 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.

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

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

```
test_dropout passed!
```

**1.1.5 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.

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

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

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

TODO: Implement the cross_entropy_loss function in NN.py.

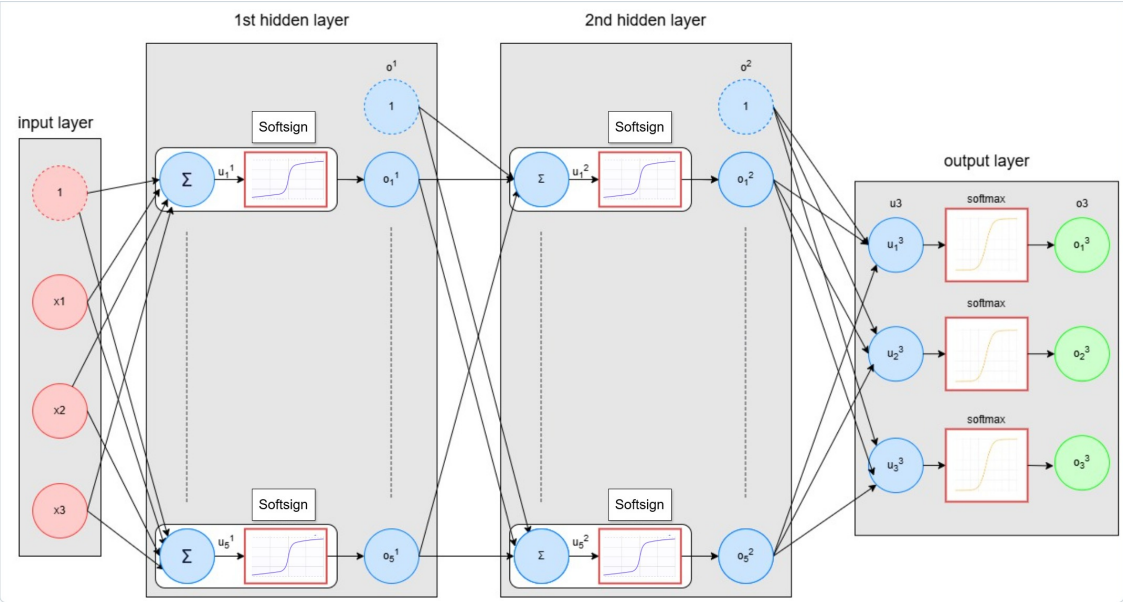
[6]:
```
from utilities.localtests import TestNN

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

```
test_loss passed!
```

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

$$
\begin{aligned}
u_1 &= \theta_1 x + b_1 \\
o_1 &= \text{dropout}(\text{softsign}(u_1), p) \\
u_2 &= \theta_2 o_1 + b_2 \\
o_2 &= \text{softsign}(u_2) \\
u_3 &= \theta_3 o_2 + b_3 \\
o_3 &= \text{softmax}(u_3) \\
l &= \text{cross\_entropy}(o_3)
\end{aligned}
$$

Here is a diagram of the same architecture:




### 1.1.6 Forward Pass TODO: Implement the forward function in NN.py.

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

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

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

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

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

**1.1.7 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.

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

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

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

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

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 $\theta_2$ 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

$$\frac{\partial l}{\partial o_2} = \frac{\partial l}{\partial u_3} \cdot \theta_3$$
$$\frac{\partial l}{\partial u_2} = \frac{\partial l}{\partial o_2} \cdot \frac{\partial \,\text{Softsign}}{\partial u_2}.$$

The same procedure is repeated to obtain the gradients for the upstream parameters $\theta_1$ and $b_1$. We must first perform the intermediate steps of computing the derivative of the loss with respect to $o_1$ and then $u_1$. These are given by

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

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

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

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

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

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

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.

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

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

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

**1.1.8 Update Weights** So, we update the weights and biases using the following formulas

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

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

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

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

[11]:
```
from utilities.localtests import TestNN

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

```
test_update_weights passed!
```

**1.1.9 Update Weights with Adam [Required for grad, Bonus for undergrad]** Gradient descent does a generally good job of facilitating the convergence of the model's parameters to minimize the loss function. However, the process of doing so can be slow and/or noisy.

**Adam (Adaptive Moment Estimation)** is an advanced optimization algorithm that combines the benefits of two other popular optimization techniques: RMSprop and momentum. Introduced

by Kingma and Ba in 2014, Adam has become one of the most widely used optimization algorithms in deep learning due to its efficiency and effectiveness across a wide range of problems. It adapts the learning update for each parameter individually, making it particularly well-suited for problems with sparse gradients or noisy data.

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

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

where $\theta_t$ represents the parameters at time $t$, $\alpha$ represents the learning rate, and $f$ is the loss function.

Adam proposes the following tweak to our parameter update function:

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

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

Here: - t is a single parameter in our network (e.g. theta3 or bias1) - $M_t$ **(First Moment)** - This tracks the exponentially weighted moving average of the gradients of parameter t. It's similar to momentum and helps the optimization continue moving in consistent directions. Essentially, it's an estimate of the mean of the gradients. - Intuition: Think of it as a ball rolling down a hill - it builds up momentum in promising directions and helps overcome small obstacles (local minima) along the way. It remembers where we've been going and helps us stay on course. - $V_t$ **(Second Moment)** - This tracks the exponentially weighted moving average of the squared gradients of parameter t. It captures the variance of gradients, which helps adapt the learning update for each parameter. - Intuition: Imagine driving through varied terrain - you'd slow down on rough roads (high gradient variance) and speed up on smooth highways (low gradient variance). $V_t$ provides this terrain-specific speed control for each parameter. - $\beta_1$ **and** $\beta_2$: These hyperparameters control how much the algorithm relies on information from new gradient vs. previously calculated $M_t$ or $V_t$ for the parameter t: - $\beta_1$ (typically 0.9) controls the decay rate of the moving average of the gradient - $\beta_2$ (typically 0.999) controls the decay rate of the moving average of the squared gradient - The values show we trust our history (90% for direction, 99.9% for variance) more than any single new piece of information. This makes Adam robust to noisy gradients in a single batch.

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

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

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

Now we can have our updating formula for Adam:

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

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

Overall, Adam works really well because we maintain $M_t$ and $V_t$ for every parameter. - Parameters with clear, consistent gradients get larger updates (high $M_t$, low $V_t$) - Parameters with noisy or sparse gradients get smaller, more careful updates (high $V_t$, low $M_t$) - The overall direction benefits from momentum, smoothing the optimization path - Each parameter gets its own custom learning rate, automatically tuned based on its gradient history

Here is an additional resource for learning more about Adam optimization: - Who's Adam and What's He Optimizing? | Deep Dive into Optimizers for Machine Learning!

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

[13]:
```
from utilities.localtests import TestNN

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

```
test_update_weights_with_adam passed!
```

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

TODO: Implement the backward function in NN.py.

[14]:
```
from utilities.localtests import TestNN

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

```
test_backward passed!
```

**1.1.11 Gradient Descent** TODO: Implement the gradient_descent function in NN.py.

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

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

[15]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
from utilities.localtests import TestNN
```

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

```
Loss after iteration 0: 1.086384
Loss after iteration 1: 1.086313
Loss after iteration 2: 1.086242

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

**1.1.12 Mini-batch Gradient Descent** TODO: Implement the minibatch_gradient_descent function in NN.py.

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

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

[16]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from utilities.localtests import TestNN

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

```
Loss after iteration 0: 1.111743
Loss after iteration 1: 1.108553
Loss after iteration 2: 1.084159
Your batch_y works within the expected range: True

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

**1.1.13 Gradient Descent with Adam [Required for grad, Bonus for undergrad]** You may test your implementation of the GD function with adam contained in **NN.py** in the cell below. See Using the Local Tests for more details. Revisit your implementation for update_weights.

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

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

from utilities.localtests import TestNN

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

```
Loss after iteration 0: 1.086384
Loss after iteration 1: 1.077445
Loss after iteration 2: 1.068520

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

**1.1.14 Mini-batch Gradient Descent with Adam [Required for grad, Bonus for undergrad]** You may test your implementation of the mini-batch gradient descent function contained in **NN.py** when use_adam=True in the cell below. See Using the Local Tests for more details.

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

[18]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from utilities.localtests import TestNN

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

```
Loss after iteration 0: 1.079837
Loss after iteration 1: 1.097294
Loss after iteration 2: 1.078121

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

### 1.7.3 1.2 Loss plot, cross-entropy (CE) value, and Learning Rate with Gradient Descent [9.5pts] [P] | [W]

**1.2.1 Loss plot and cross-entropy(CE) value with Gradient Descent [7.5pts]** Now, you can fully train your neural network implementation with gradient descent. The following cells will plot the loss vs epoch graph and calculate the final test cross-entropy(CE).

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

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

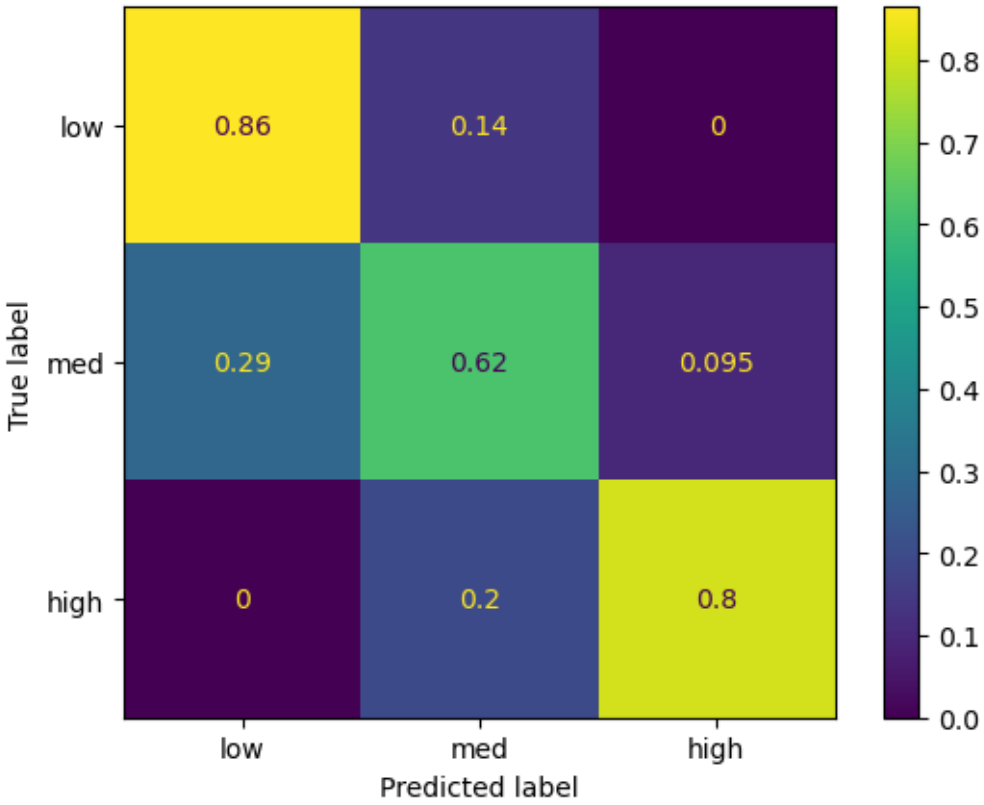
[19]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from NN import NeuralNet
```

```
from sklearn.metrics import ConfusionMatrixDisplay

x_train, y_train, x_test, y_test = get_housing_dataset()

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

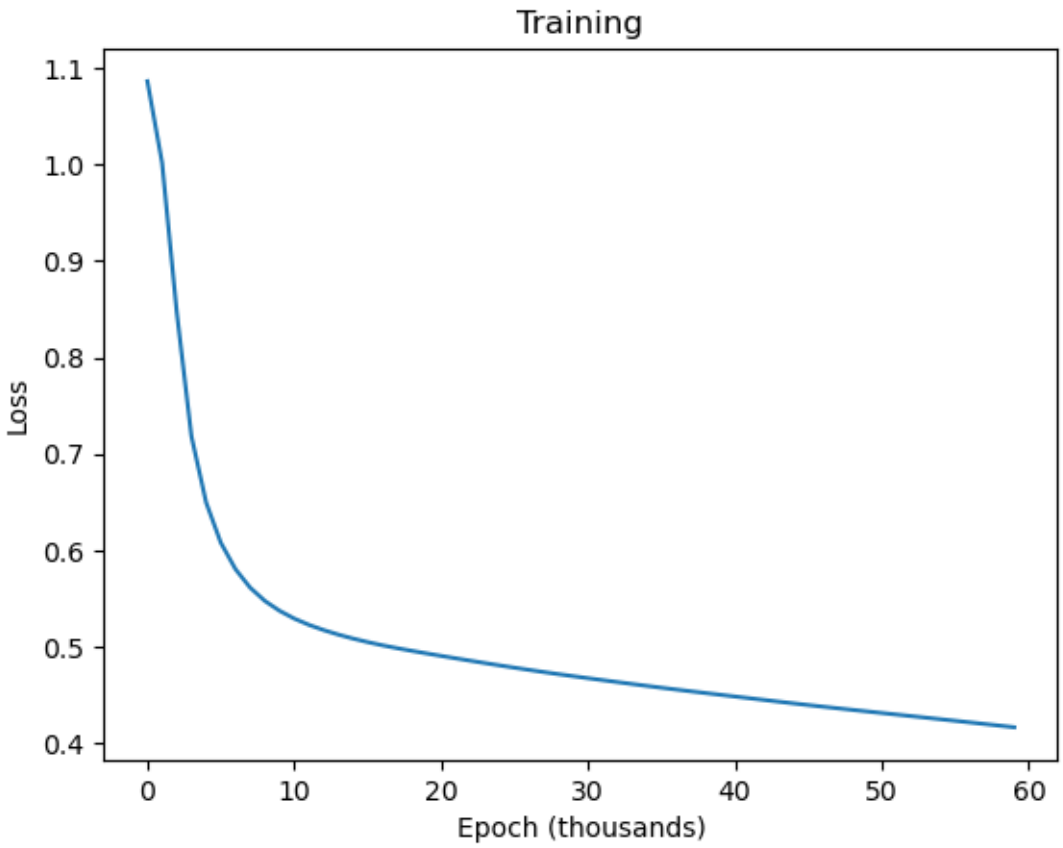
```
Loss after iteration 0: 1.086384
Loss after iteration 1000: 1.001952
Loss after iteration 2000: 0.846899
Loss after iteration 3000: 0.718109
Loss after iteration 4000: 0.649426
Loss after iteration 5000: 0.607674
Loss after iteration 6000: 0.580241
Loss after iteration 7000: 0.561059
Loss after iteration 8000: 0.547331
Loss after iteration 9000: 0.537094
Loss after iteration 10000: 0.529029
Loss after iteration 11000: 0.522520
Loss after iteration 12000: 0.517104
Loss after iteration 13000: 0.512428
Loss after iteration 14000: 0.508310
Loss after iteration 15000: 0.504628
Loss after iteration 16000: 0.501325
Loss after iteration 17000: 0.498342
Loss after iteration 18000: 0.495574
Loss after iteration 19000: 0.492921
Loss after iteration 20000: 0.490341
Loss after iteration 21000: 0.487806
Loss after iteration 22000: 0.485304
Loss after iteration 23000: 0.482841
Loss after iteration 24000: 0.480429
Loss after iteration 25000: 0.478076
Loss after iteration 26000: 0.475778
Loss after iteration 27000: 0.473541
Loss after iteration 28000: 0.471367
Loss after iteration 29000: 0.469261
Loss after iteration 30000: 0.467202
Loss after iteration 31000: 0.465174
Loss after iteration 32000: 0.463173
Loss after iteration 33000: 0.461193
Loss after iteration 34000: 0.459237
Loss after iteration 35000: 0.457314
Loss after iteration 36000: 0.455438
Loss after iteration 37000: 0.453604
```

```
Loss after iteration 38000: 0.451796
Loss after iteration 39000: 0.450005
Loss after iteration 40000: 0.448192
Loss after iteration 41000: 0.446410
Loss after iteration 42000: 0.444626
Loss after iteration 43000: 0.442824
Loss after iteration 44000: 0.441061
Loss after iteration 45000: 0.439332
Loss after iteration 46000: 0.437641
Loss after iteration 47000: 0.435985
Loss after iteration 48000: 0.434350
Loss after iteration 49000: 0.432724
Loss after iteration 50000: 0.431106
Loss after iteration 51000: 0.429496
Loss after iteration 52000: 0.427883
Loss after iteration 53000: 0.426264
Loss after iteration 54000: 0.424637
Loss after iteration 55000: 0.422997
Loss after iteration 56000: 0.421348
Loss after iteration 57000: 0.419686
Loss after iteration 58000: 0.418015
Loss after iteration 59000: 0.416338
```

[20]:
```python
# 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()
```




[21]:
```
# Plot training loss
fig = plt.plot(np.array(nn.loss).squeeze())
plt.title(f"Training")
plt.xlabel("Epoch (thousands)")
plt.ylabel("Loss")
plt.show()
```




[22]:
```
# 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: 0.681
```

**1.2.2 Learning Rate with Gradient Descent [2pts]** Here we have provided three different learning rates to use for training. Run the plots for each learning rate and describe each plot. Which learning rate works best? Explain how you know.

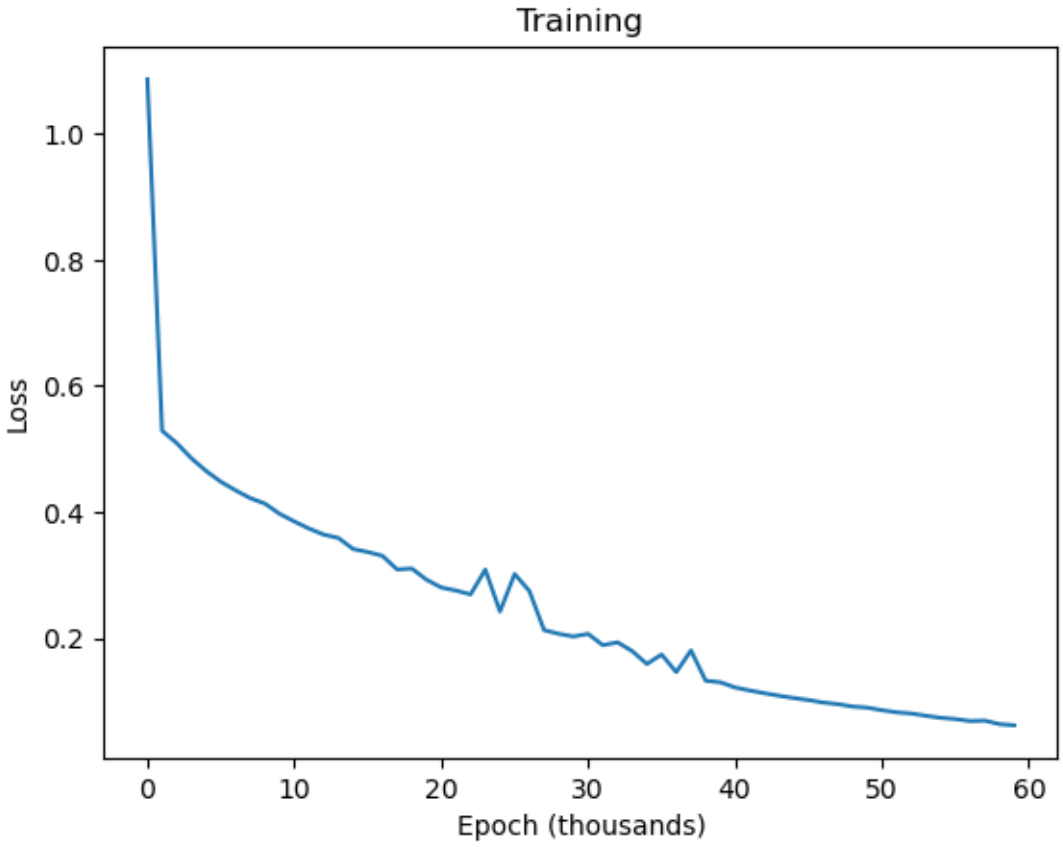
**[W]** Please assign all pages containing your answer to the question. You should assign both your evidence generated from code cells, including the below plots or any you may add. You should also assign any markdown cells containing your ultimate choice and justification.

[23]:
```
from NN import NeuralNet

x_train, y_train, x_test, y_test = get_housing_dataset()

nn = NeuralNet(
    y_train, lr=0.1, use_dropout=False, use_adam=False
)  # initialize neural net class
```

```
nn.gradient_descent(x_train, y_train, iter=60000)  # train

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

```
Loss after iteration 0: 1.086384
Loss after iteration 1000: 0.529056
Loss after iteration 2000: 0.509310
Loss after iteration 3000: 0.485269
Loss after iteration 4000: 0.465143
Loss after iteration 5000: 0.448164
Loss after iteration 6000: 0.434512
Loss after iteration 7000: 0.422164
Loss after iteration 8000: 0.413500
Loss after iteration 9000: 0.397260
Loss after iteration 10000: 0.385149
Loss after iteration 11000: 0.373969
Loss after iteration 12000: 0.364383
Loss after iteration 13000: 0.358991
Loss after iteration 14000: 0.341489
Loss after iteration 15000: 0.336735
Loss after iteration 16000: 0.330647
Loss after iteration 17000: 0.309012
Loss after iteration 18000: 0.310486
Loss after iteration 19000: 0.292806
Loss after iteration 20000: 0.280467
Loss after iteration 21000: 0.275612
Loss after iteration 22000: 0.269508
Loss after iteration 23000: 0.308830
Loss after iteration 24000: 0.242329
Loss after iteration 25000: 0.301847
Loss after iteration 26000: 0.274905
Loss after iteration 27000: 0.212884
Loss after iteration 28000: 0.207031
Loss after iteration 29000: 0.202791
Loss after iteration 30000: 0.206914
Loss after iteration 31000: 0.188975
Loss after iteration 32000: 0.193483
Loss after iteration 33000: 0.179602
Loss after iteration 34000: 0.159240
Loss after iteration 35000: 0.174250
Loss after iteration 36000: 0.146227
Loss after iteration 37000: 0.180497
```

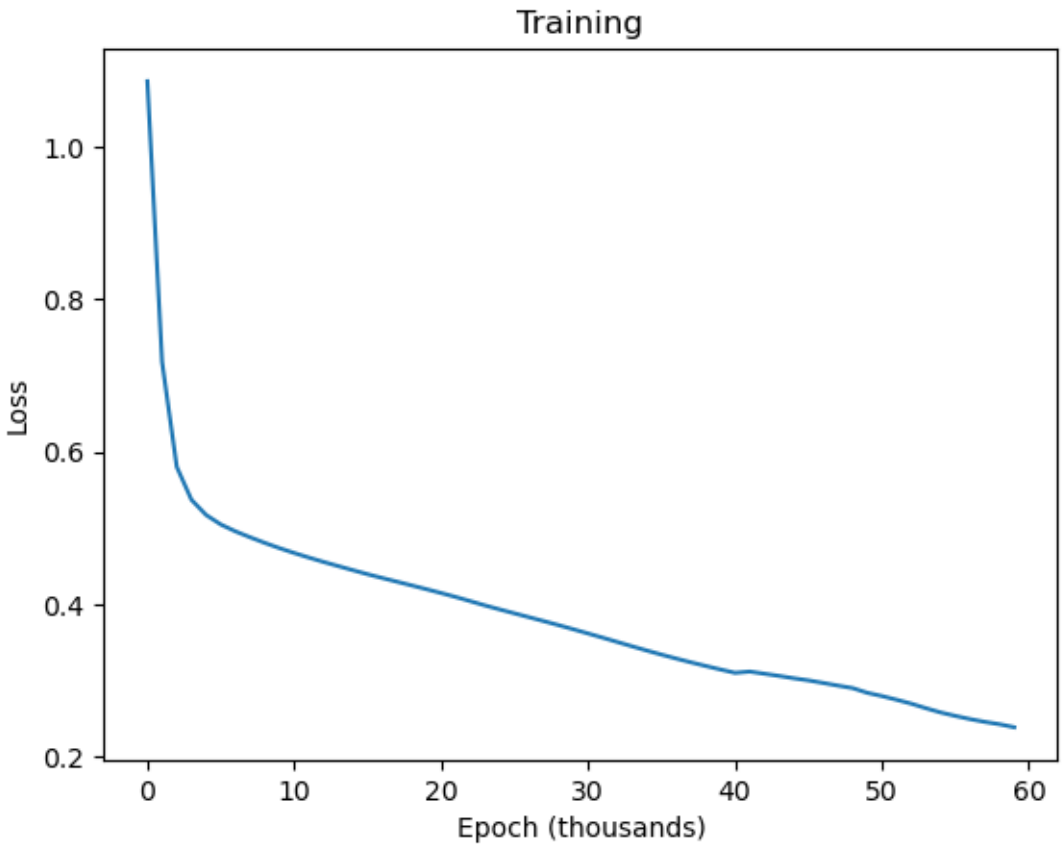
```
Loss after iteration 38000: 0.132637
Loss after iteration 39000: 0.130191
Loss after iteration 40000: 0.121921
Loss after iteration 41000: 0.117122
Loss after iteration 42000: 0.112762
Loss after iteration 43000: 0.108785
Loss after iteration 44000: 0.105160
Loss after iteration 45000: 0.101996
Loss after iteration 46000: 0.098016
Loss after iteration 47000: 0.095523
Loss after iteration 48000: 0.091719
Loss after iteration 49000: 0.089903
Loss after iteration 50000: 0.085827
Loss after iteration 51000: 0.082544
Loss after iteration 52000: 0.080607
Loss after iteration 53000: 0.077049
Loss after iteration 54000: 0.073792
Loss after iteration 55000: 0.071853
Loss after iteration 56000: 0.068696
Loss after iteration 57000: 0.069374
Loss after iteration 58000: 0.063761
Loss after iteration 59000: 0.062024
```



[24]:
```
x_train, y_train, x_test, y_test = get_housing_dataset()

nn = NeuralNet(
    y_train, lr=0.03, use_dropout=False, use_adam=False
)  # initialize neural net class
nn.gradient_descent(x_train, y_train, iter=60000)  # train

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

```
Loss after iteration 0: 1.086384
Loss after iteration 1000: 0.718204
Loss after iteration 2000: 0.580262
Loss after iteration 3000: 0.537101
Loss after iteration 4000: 0.517108
Loss after iteration 5000: 0.504630
Loss after iteration 6000: 0.495575
Loss after iteration 7000: 0.487807
Loss after iteration 8000: 0.480430
Loss after iteration 9000: 0.473541
Loss after iteration 10000: 0.467201
Loss after iteration 11000: 0.461192
Loss after iteration 12000: 0.455436
Loss after iteration 13000: 0.450003
Loss after iteration 14000: 0.444614
Loss after iteration 15000: 0.439320
Loss after iteration 16000: 0.434343
Loss after iteration 17000: 0.429491
Loss after iteration 18000: 0.424633
Loss after iteration 19000: 0.419684
Loss after iteration 20000: 0.414647
Loss after iteration 21000: 0.409434
Loss after iteration 22000: 0.403911
Loss after iteration 23000: 0.398313
Loss after iteration 24000: 0.393064
Loss after iteration 25000: 0.387935
Loss after iteration 26000: 0.382836
Loss after iteration 27000: 0.377706
Loss after iteration 28000: 0.372494
Loss after iteration 29000: 0.367088
Loss after iteration 30000: 0.361588
```

```
Loss after iteration 31000: 0.356047
Loss after iteration 32000: 0.350357
Loss after iteration 33000: 0.344705
Loss after iteration 34000: 0.339192
Loss after iteration 35000: 0.333826
Loss after iteration 36000: 0.328670
Loss after iteration 37000: 0.323721
Loss after iteration 38000: 0.318972
Loss after iteration 39000: 0.314381
Loss after iteration 40000: 0.309958
Loss after iteration 41000: 0.311591
Loss after iteration 42000: 0.308765
Loss after iteration 43000: 0.305905
Loss after iteration 44000: 0.302806
Loss after iteration 45000: 0.300045
Loss after iteration 46000: 0.296678
Loss after iteration 47000: 0.293210
Loss after iteration 48000: 0.289941
Loss after iteration 49000: 0.283736
Loss after iteration 50000: 0.279455
Loss after iteration 51000: 0.274613
Loss after iteration 52000: 0.269466
Loss after iteration 53000: 0.263483
Loss after iteration 54000: 0.257939
Loss after iteration 55000: 0.253394
Loss after iteration 56000: 0.249227
Loss after iteration 57000: 0.245576
Loss after iteration 58000: 0.242489
Loss after iteration 59000: 0.238546
```

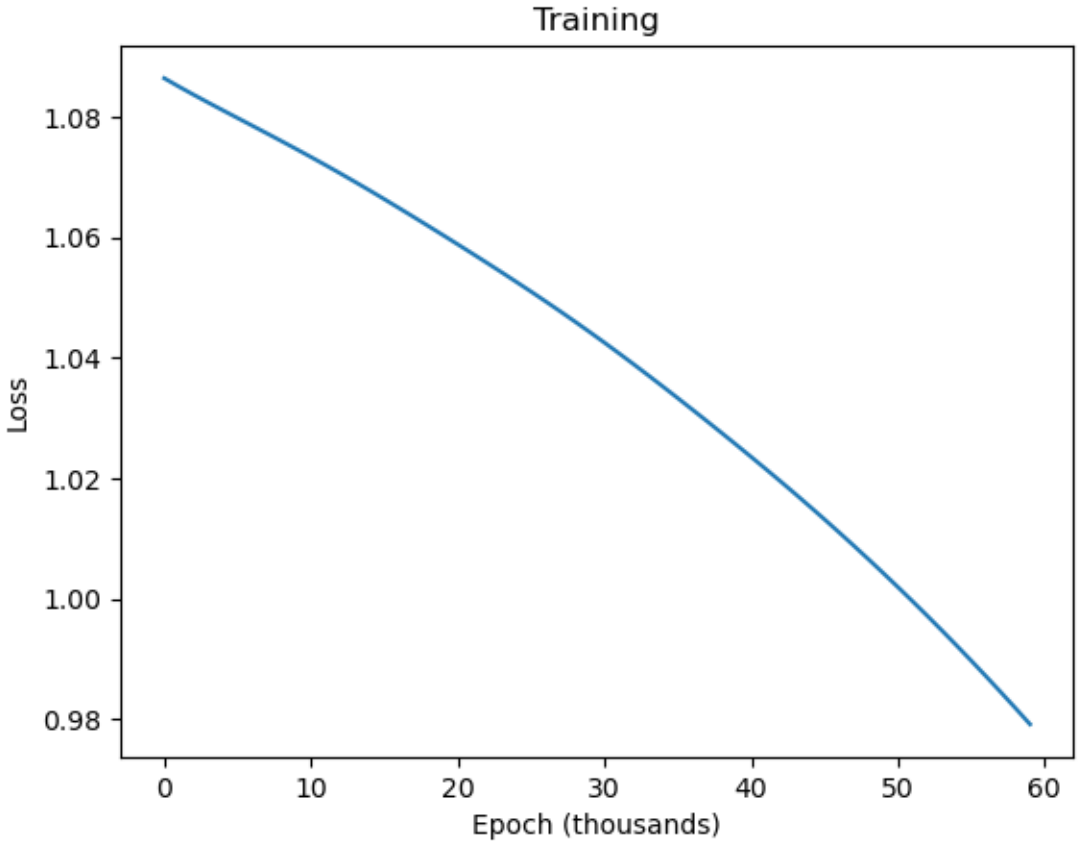



[25]:
```
x_train, y_train, x_test, y_test = get_housing_dataset()

nn = NeuralNet(
    y_train, lr=0.0002, use_dropout=False, use_adam=False
)  # initialize neural net class
nn.gradient_descent(x_train, y_train, iter=60000)  # train

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

```
Loss after iteration 0: 1.086384
Loss after iteration 1000: 1.084990
Loss after iteration 2000: 1.083646
Loss after iteration 3000: 1.082339
Loss after iteration 4000: 1.081052
Loss after iteration 5000: 1.079777
```

```
Loss after iteration 6000: 1.078503
Loss after iteration 7000: 1.077224
Loss after iteration 8000: 1.075934
Loss after iteration 9000: 1.074627
Loss after iteration 10000: 1.073300
Loss after iteration 11000: 1.071951
Loss after iteration 12000: 1.070577
Loss after iteration 13000: 1.069179
Loss after iteration 14000: 1.067754
Loss after iteration 15000: 1.066303
Loss after iteration 16000: 1.064832
Loss after iteration 17000: 1.063344
Loss after iteration 18000: 1.061845
Loss after iteration 19000: 1.060333
Loss after iteration 20000: 1.058808
Loss after iteration 21000: 1.057272
Loss after iteration 22000: 1.055721
Loss after iteration 23000: 1.054153
Loss after iteration 24000: 1.052565
Loss after iteration 25000: 1.050955
Loss after iteration 26000: 1.049321
Loss after iteration 27000: 1.047659
Loss after iteration 28000: 1.045967
Loss after iteration 29000: 1.044243
Loss after iteration 30000: 1.042488
Loss after iteration 31000: 1.040699
Loss after iteration 32000: 1.038880
Loss after iteration 33000: 1.037035
Loss after iteration 34000: 1.035164
Loss after iteration 35000: 1.033271
Loss after iteration 36000: 1.031357
Loss after iteration 37000: 1.029424
Loss after iteration 38000: 1.027474
Loss after iteration 39000: 1.025504
Loss after iteration 40000: 1.023512
Loss after iteration 41000: 1.021497
Loss after iteration 42000: 1.019455
Loss after iteration 43000: 1.017383
Loss after iteration 44000: 1.015278
Loss after iteration 45000: 1.013140
Loss after iteration 46000: 1.010967
Loss after iteration 47000: 1.008758
Loss after iteration 48000: 1.006511
Loss after iteration 49000: 1.004227
Loss after iteration 50000: 1.001904
Loss after iteration 51000: 0.999541
Loss after iteration 52000: 0.997137
Loss after iteration 53000: 0.994691
```

```
Loss after iteration 54000: 0.992204
Loss after iteration 55000: 0.989675
Loss after iteration 56000: 0.987103
Loss after iteration 57000: 0.984489
Loss after iteration 58000: 0.981835
Loss after iteration 59000: 0.979140
```




---

The learning rate `lr=0.1` works best. Over the same number of iterations, it achieves the lowest training loss of ~0.06, much less than ~0.24 and ~0.98 reached by the other learning rates. It also learned much faster than the other two.

---

#### 1.7.4 1.3 Loss plot and CE value for Mini-batch GD [7.5pts] [P]

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

You can test and debug your network here, but your implementation will be tested on gradescope

so there is no partial credit for notebook output.

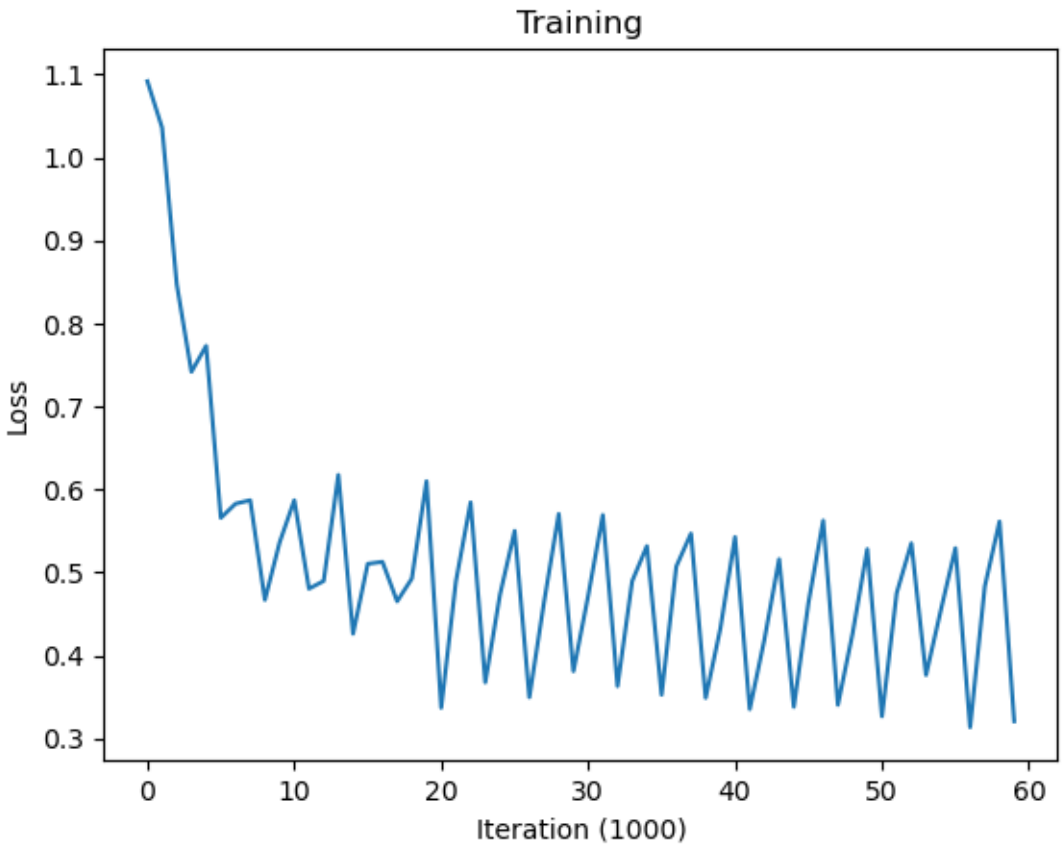
To achieve full credit on gradescope: 1. Your loss trajectory should be decreasing and similar to expected trajectory (2.5 pts) 2. Your final cross entropy loss must be within 0.7 of the expected loss (2.5 pts) 3. The diagonal entries in your confusion matrix must be within 0.15 of the expected values (2.5 pts)

[26]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

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

x_train, y_train, x_test, y_test = get_housing_dataset()

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

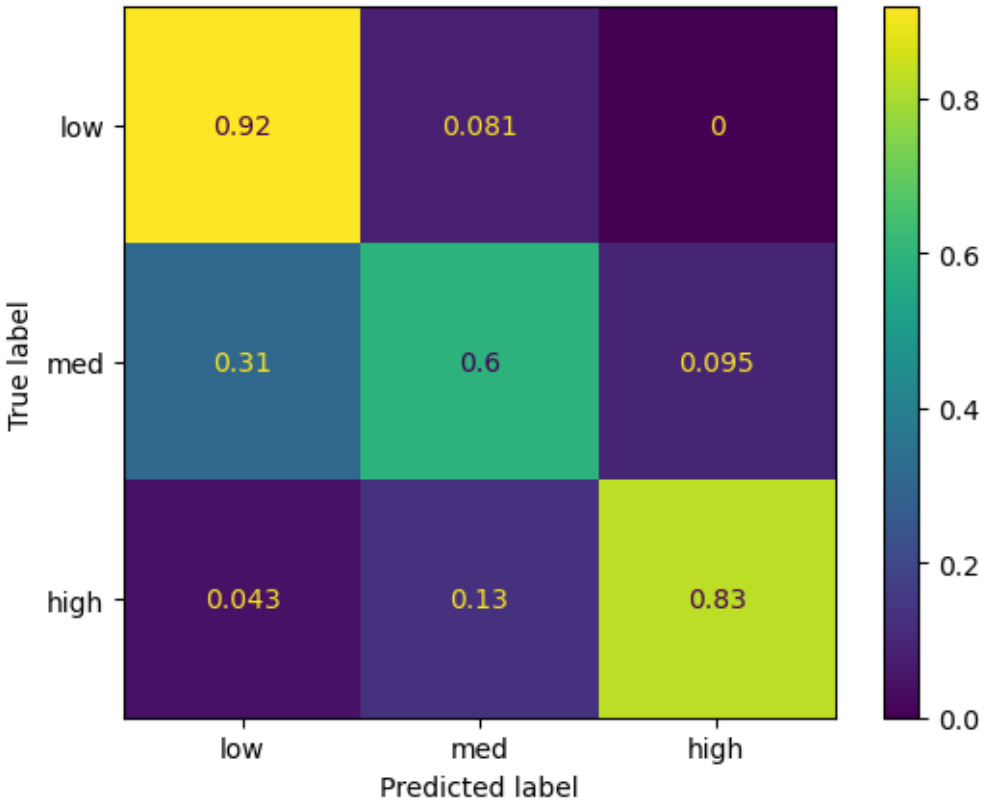
```
Loss after iteration 0: 1.091859
Loss after iteration 1000: 1.035899
Loss after iteration 2000: 0.847171
Loss after iteration 3000: 0.741736
Loss after iteration 4000: 0.773022
Loss after iteration 5000: 0.565779
Loss after iteration 6000: 0.582815
Loss after iteration 7000: 0.587042
Loss after iteration 8000: 0.466858
Loss after iteration 9000: 0.536166
Loss after iteration 10000: 0.587051
Loss after iteration 11000: 0.480393
Loss after iteration 12000: 0.489757
Loss after iteration 13000: 0.617539
Loss after iteration 14000: 0.426187
Loss after iteration 15000: 0.510418
Loss after iteration 16000: 0.512826
Loss after iteration 17000: 0.465207
Loss after iteration 18000: 0.492669
Loss after iteration 19000: 0.609935
Loss after iteration 20000: 0.336917
Loss after iteration 21000: 0.489244
Loss after iteration 22000: 0.584403
Loss after iteration 23000: 0.367593
Loss after iteration 24000: 0.473196
Loss after iteration 25000: 0.550124
Loss after iteration 26000: 0.349599
```

```
Loss after iteration 27000: 0.464375
Loss after iteration 28000: 0.570501
Loss after iteration 29000: 0.381022
Loss after iteration 30000: 0.471606
Loss after iteration 31000: 0.569234
Loss after iteration 32000: 0.363034
Loss after iteration 33000: 0.489637
Loss after iteration 34000: 0.531827
Loss after iteration 35000: 0.352604
Loss after iteration 36000: 0.507195
Loss after iteration 37000: 0.546935
Loss after iteration 38000: 0.348872
Loss after iteration 39000: 0.432399
Loss after iteration 40000: 0.542698
Loss after iteration 41000: 0.335282
Loss after iteration 42000: 0.419321
Loss after iteration 43000: 0.516011
Loss after iteration 44000: 0.338062
Loss after iteration 45000: 0.463816
Loss after iteration 46000: 0.562512
Loss after iteration 47000: 0.340467
Loss after iteration 48000: 0.426473
Loss after iteration 49000: 0.528113
Loss after iteration 50000: 0.326665
Loss after iteration 51000: 0.475154
Loss after iteration 52000: 0.535218
Loss after iteration 53000: 0.376442
Loss after iteration 54000: 0.454066
Loss after iteration 55000: 0.529333
Loss after iteration 56000: 0.313504
Loss after iteration 57000: 0.483299
Loss after iteration 58000: 0.561374
Loss after iteration 59000: 0.320788
```

[27]:
```python
# Plot training loss
fig = plt.plot(np.array(nn.loss).squeeze())
plt.title(f"Training")
plt.xlabel("Iteration (1000)")
plt.ylabel("Loss")
plt.show()
```




[28]:
```
# 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()
```




[29]:
```
# 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: 0.712
```

### 1.7.5 1.4 Loss plot and CE value for Gradient Descent with Adam [7.5pts Grad / 1% Bonus for Undergrad] [P]

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

To achieve full credit on gradescope: 1. Your loss trajectory should be decreasing and similar to expected trajectory (2.5 pts) 2. Your final cross entropy loss must be within 0.7 of the expected loss (2.5 pts) 3. The diagonal entries in your confusion matrix must be within 0.15 of the expected values (2.5 pts)

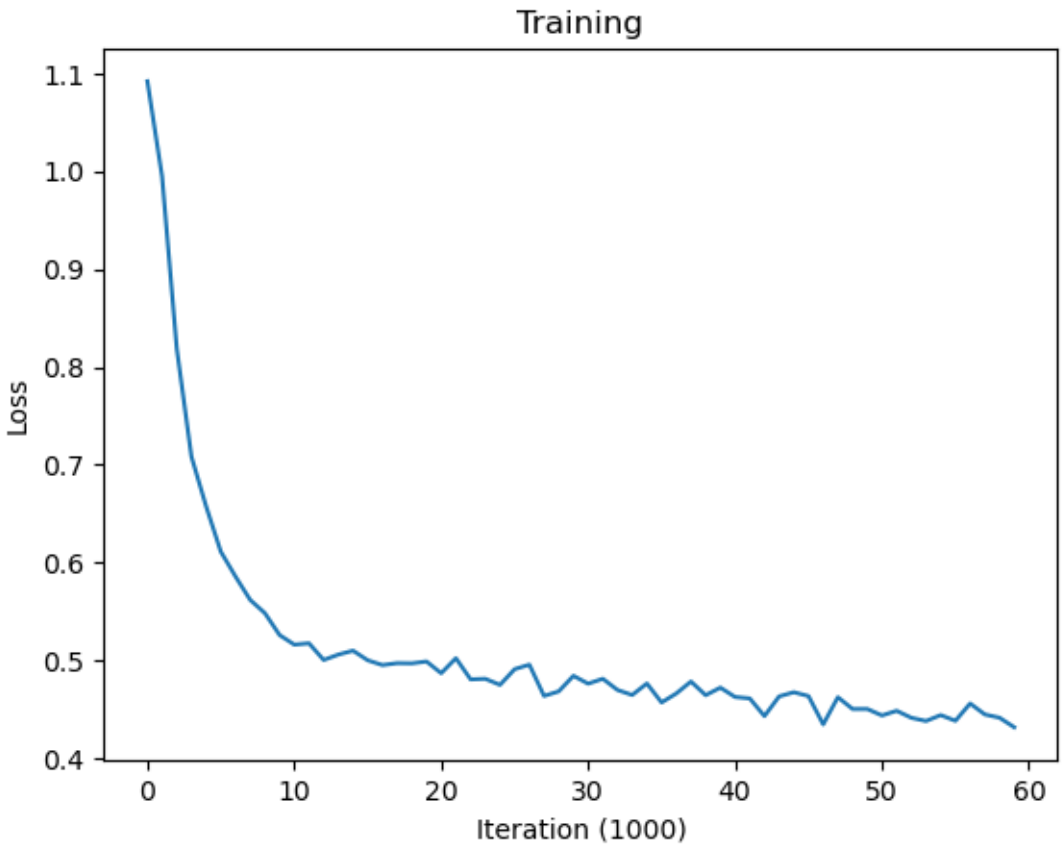
[30]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
```

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

x_train, y_train, x_test, y_test = get_housing_dataset()

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

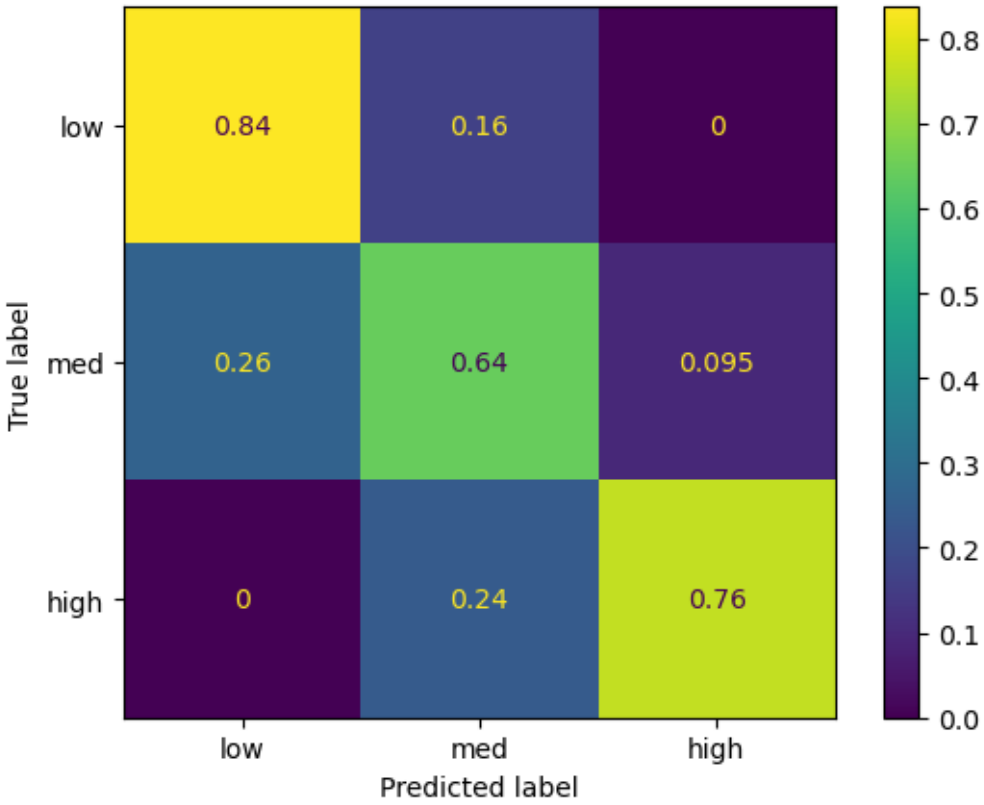
```
Loss after iteration 0: 1.092085
Loss after iteration 1000: 0.994819
Loss after iteration 2000: 0.818885
Loss after iteration 3000: 0.708302
Loss after iteration 4000: 0.657461
Loss after iteration 5000: 0.611148
Loss after iteration 6000: 0.585753
Loss after iteration 7000: 0.561844
Loss after iteration 8000: 0.548353
Loss after iteration 9000: 0.525937
Loss after iteration 10000: 0.516159
Loss after iteration 11000: 0.517531
Loss after iteration 12000: 0.500480
Loss after iteration 13000: 0.505991
Loss after iteration 14000: 0.509984
Loss after iteration 15000: 0.500153
Loss after iteration 16000: 0.495247
Loss after iteration 17000: 0.497214
Loss after iteration 18000: 0.497009
Loss after iteration 19000: 0.498928
Loss after iteration 20000: 0.486947
Loss after iteration 21000: 0.502284
Loss after iteration 22000: 0.480663
Loss after iteration 23000: 0.481239
Loss after iteration 24000: 0.475030
Loss after iteration 25000: 0.491100
Loss after iteration 26000: 0.495675
Loss after iteration 27000: 0.463680
Loss after iteration 28000: 0.468387
Loss after iteration 29000: 0.484296
Loss after iteration 30000: 0.476243
Loss after iteration 31000: 0.481268
Loss after iteration 32000: 0.469781
Loss after iteration 33000: 0.464685
Loss after iteration 34000: 0.476446
Loss after iteration 35000: 0.457027
Loss after iteration 36000: 0.466227
```

```
Loss after iteration 37000: 0.478422
Loss after iteration 38000: 0.464461
Loss after iteration 39000: 0.472205
Loss after iteration 40000: 0.462635
Loss after iteration 41000: 0.461205
Loss after iteration 42000: 0.443126
Loss after iteration 43000: 0.463238
Loss after iteration 44000: 0.467495
Loss after iteration 45000: 0.463566
Loss after iteration 46000: 0.434615
Loss after iteration 47000: 0.462420
Loss after iteration 48000: 0.450356
Loss after iteration 49000: 0.450394
Loss after iteration 50000: 0.443737
Loss after iteration 51000: 0.448532
Loss after iteration 52000: 0.441352
Loss after iteration 53000: 0.438072
Loss after iteration 54000: 0.444127
Loss after iteration 55000: 0.438280
Loss after iteration 56000: 0.455997
Loss after iteration 57000: 0.444804
Loss after iteration 58000: 0.441443
Loss after iteration 59000: 0.431706
```

[31]:
```python
# Plot training loss
fig = plt.plot(np.array(nn.loss).squeeze())
plt.title(f"Training")
plt.xlabel("Iteration (1000)")
plt.ylabel("Loss")
plt.show()
```




[32]:
```
# 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()
```




[33]:
```
# 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: 0.741
```

### 1.7.6 1.5 Loss plot and CE value for Mini-batch GD with Adam [7.5pts Grad / 1% Bonus for Undergrad] [P]

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

To achieve full credit on gradescope: 1. Your loss trajectory should be decreasing and similar to expected trajectory (2.5 pts) 2. Your final cross entropy loss must be within 0.7 of the expected loss (2.5 pts) 3. The diagonal entries in your confusion matrix must be within 0.15 of the expected values (2.5 pts)

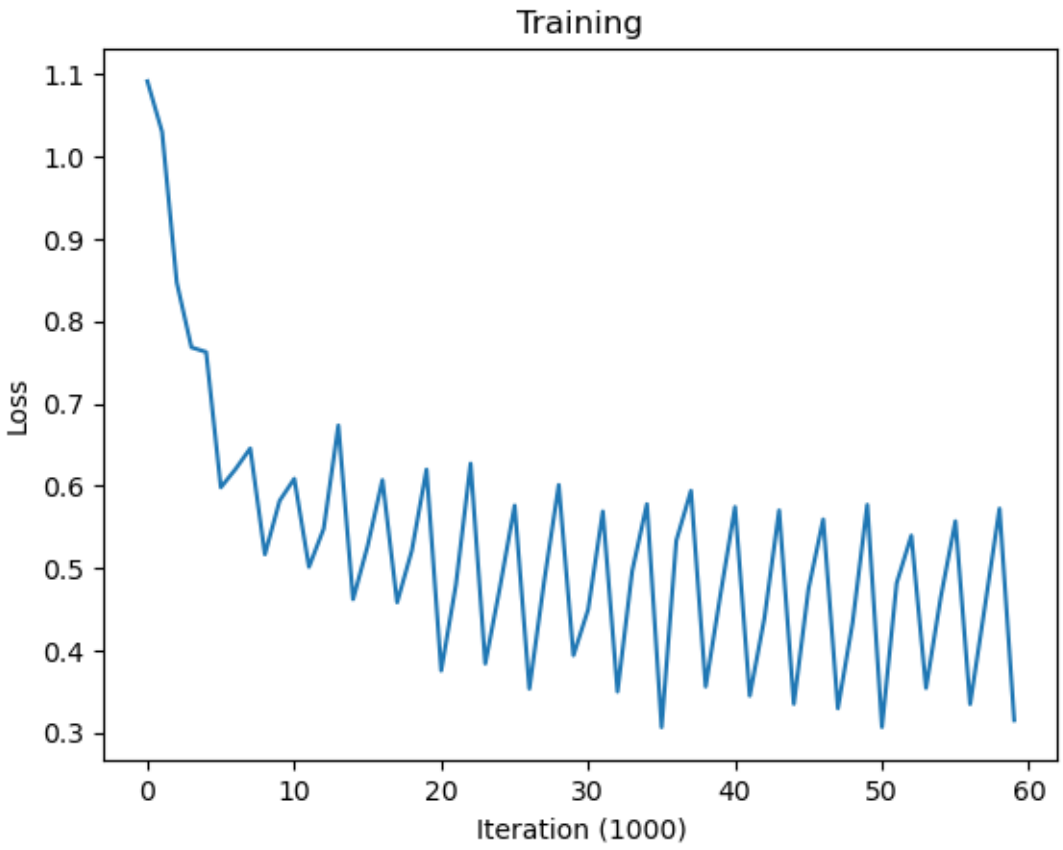
[34]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
```

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

x_train, y_train, x_test, y_test = get_housing_dataset()

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

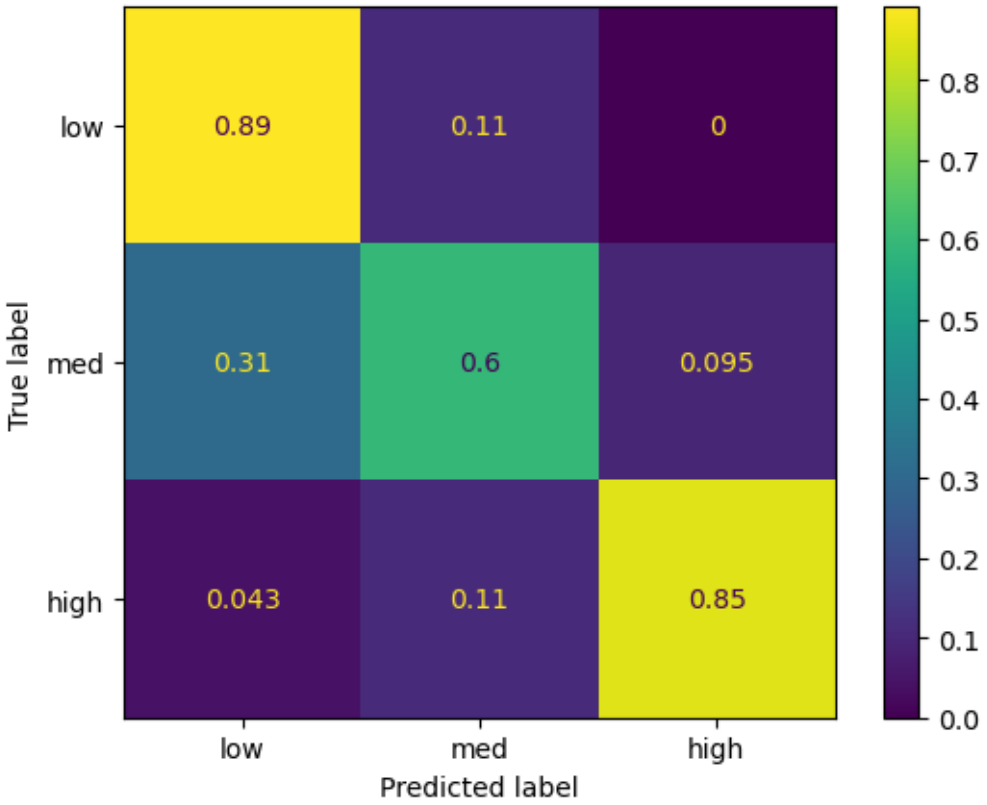
```
Loss after iteration 0: 1.091859
Loss after iteration 1000: 1.030364
Loss after iteration 2000: 0.847108
Loss after iteration 3000: 0.768244
Loss after iteration 4000: 0.762525
Loss after iteration 5000: 0.597977
Loss after iteration 6000: 0.620159
Loss after iteration 7000: 0.645412
Loss after iteration 8000: 0.516455
Loss after iteration 9000: 0.581680
Loss after iteration 10000: 0.608523
Loss after iteration 11000: 0.501214
Loss after iteration 12000: 0.547743
Loss after iteration 13000: 0.673327
Loss after iteration 14000: 0.461862
Loss after iteration 15000: 0.527386
Loss after iteration 16000: 0.607045
Loss after iteration 17000: 0.458089
Loss after iteration 18000: 0.521128
Loss after iteration 19000: 0.619692
Loss after iteration 20000: 0.375154
Loss after iteration 21000: 0.479561
Loss after iteration 22000: 0.626976
Loss after iteration 23000: 0.383667
Loss after iteration 24000: 0.476830
Loss after iteration 25000: 0.575961
Loss after iteration 26000: 0.353128
Loss after iteration 27000: 0.482767
Loss after iteration 28000: 0.600986
Loss after iteration 29000: 0.393712
Loss after iteration 30000: 0.449460
Loss after iteration 31000: 0.568763
Loss after iteration 32000: 0.349636
Loss after iteration 33000: 0.496331
Loss after iteration 34000: 0.577693
Loss after iteration 35000: 0.306311
Loss after iteration 36000: 0.533231
```

```
Loss after iteration 37000: 0.593978
Loss after iteration 38000: 0.355743
Loss after iteration 39000: 0.466800
Loss after iteration 40000: 0.574201
Loss after iteration 41000: 0.344580
Loss after iteration 42000: 0.439001
Loss after iteration 43000: 0.570066
Loss after iteration 44000: 0.334673
Loss after iteration 45000: 0.474704
Loss after iteration 46000: 0.559083
Loss after iteration 47000: 0.329248
Loss after iteration 48000: 0.433405
Loss after iteration 49000: 0.576862
Loss after iteration 50000: 0.306490
Loss after iteration 51000: 0.481590
Loss after iteration 52000: 0.539496
Loss after iteration 53000: 0.353943
Loss after iteration 54000: 0.464240
Loss after iteration 55000: 0.556888
Loss after iteration 56000: 0.334425
Loss after iteration 57000: 0.452438
Loss after iteration 58000: 0.572210
Loss after iteration 59000: 0.314848
```

[35]:
```
# Plot training loss
fig = plt.plot(np.array(nn.loss).squeeze())
plt.title(f"Training")
plt.xlabel("Iteration (1000)")
plt.ylabel("Loss")
plt.show()
```




[36]:
```
# 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()
```




[37]:
```
# 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: 0.695
```

### 1.8 Q2: CNN Image Classification [26pts: 10pts + 16pts Grad / 2.4% Bonus for Undergrad + 2.5% Bonus for all] [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**

The conda environment solver should have correctly installed the correct software for your GPU. If it's being finicky, these directions may be helpful for a manual install.

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.

### 1.8.1 2.1 CNN Image Classification [26pts: 10pts + 16pts Grad / 2.4% Bonus for Undergrad] [P]

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

Data augmentation 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()

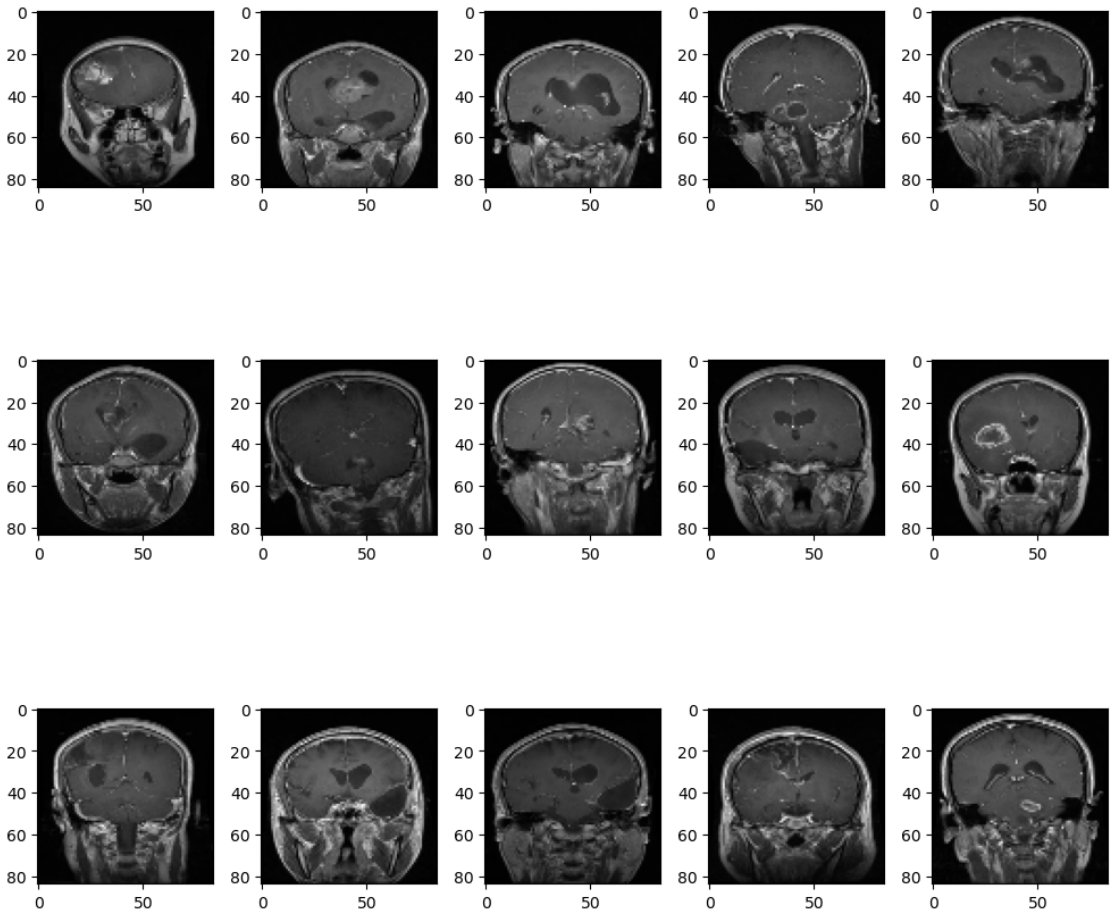
Article about performance regarding transformations

[3]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

# Load data
classes = ["glioma", "meningioma", "notumor", "pituitary"]
```

```
x_train, y_train, x_test, y_test = get_mri_dataset(classes)

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

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

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

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

[4]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

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

# show sample images

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



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

**2.1.2 Build Convolutional Neural Network Model** In this part, you need to build a convolutional neural network as described below. The architecture of the model is outlined.

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

- ___init___: See Defining Variables section
- forward: See Defining Model section

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

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

CONV: Conv. layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to the input volume. In our example architecture, we decide

to set the kernel_size to be $3 \times 3$. For example, the output of the Conv. layer may look like $[8 \times 84 \times 84]$ if we set out_channels to be 8 and use appropriate paddings to maintain shape.

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

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

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 $3 \times 3$ 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 $3 \times 3$ and out_channels to be 64. Appropriate paddings are used to maintain shape.

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

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

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

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

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

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

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

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

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

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

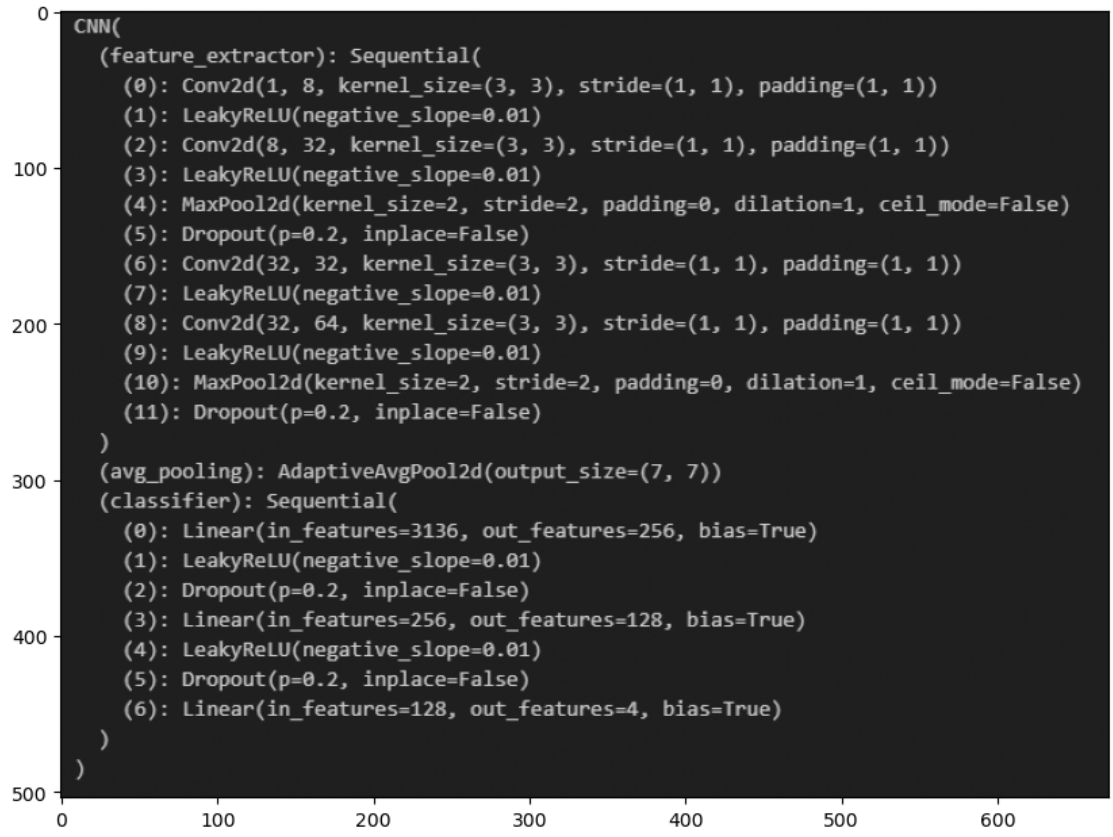
The following links are Pytorch documentation for the layers you are going to use to build the CNN.

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

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

[5]:
```
################################
### 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)
```

[5]: <matplotlib.image.AxesImage at 0x267bf1ed3d0>




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

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

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

[11]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

# You can compare your architecture with the 'Architecture.png'
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, inplace=True)
    (2): Conv2d(8, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): LeakyReLU(negative_slope=0.01, inplace=True)
    (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, inplace=True)
    (8): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): LeakyReLU(negative_slope=0.01, inplace=True)
    (10): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (11): Dropout(p=0.2, inplace=False)
  )
  (avg_pooling): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=3136, out_features=256, bias=True)
    (2): LeakyReLU(negative_slope=0.01, inplace=True)
    (3): Dropout(p=0.1, inplace=False)
    (4): Linear(in_features=256, out_features=128, bias=True)
    (5): LeakyReLU(negative_slope=0.01, inplace=True)
    (6): Dropout(p=0.1, inplace=False)
    (7): Linear(in_features=128, out_features=4, bias=True)
  )
)
```

**Local Test - Model Architecture** The test below tests if your model architecture is compatible with the input.

[12]:
```
from utilities.localtests import TestCNN

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

```
test_model_architecture passed!
```

**2.1.3 Training and Tuning the Model [Required for grad, Bonus for undergrad]** **Training:** You have to implement the function to be used in both the train step and val step of each epoch. Implement `run_epoch()` in cnn_trainer.py.

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

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

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

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

- optim
- Adamax
- CrossEntropyLoss
- ExponentialLR

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

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

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

**Expected Result:**

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

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

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

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

[13]:
```
from cnn import CNN
from cnn_trainer import Trainer

```

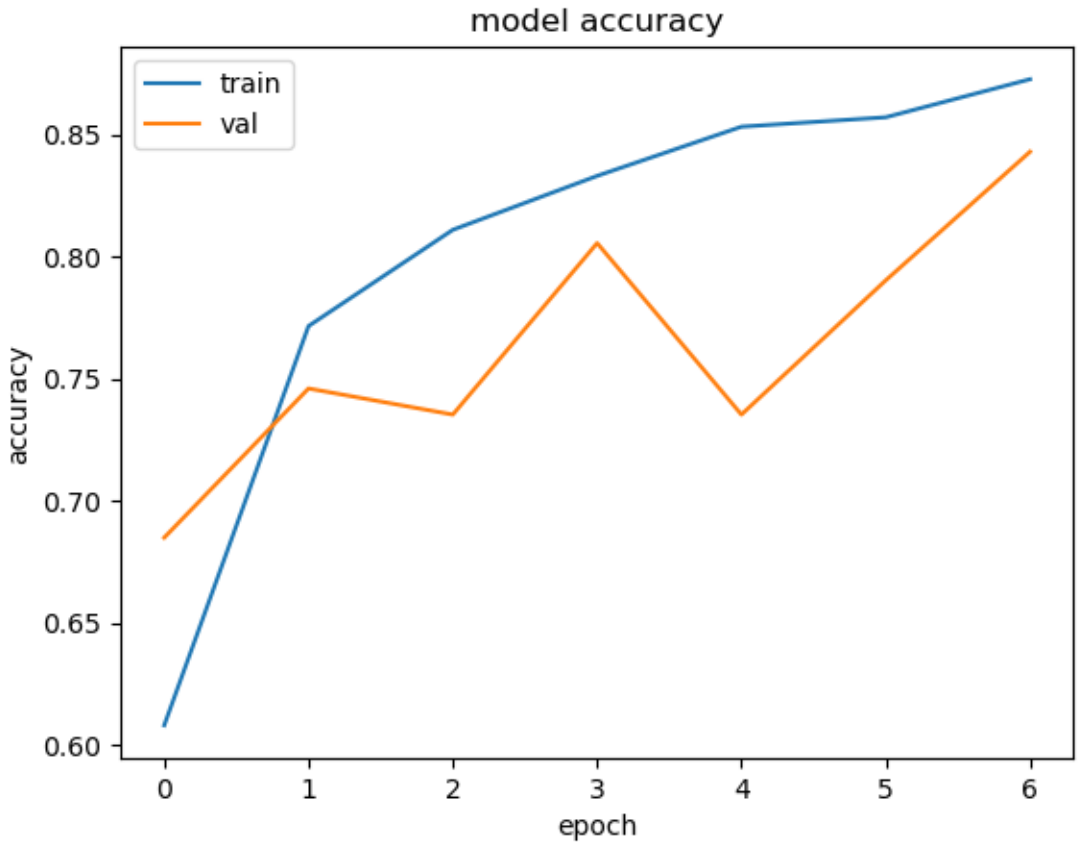
```
net = CNN()

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

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

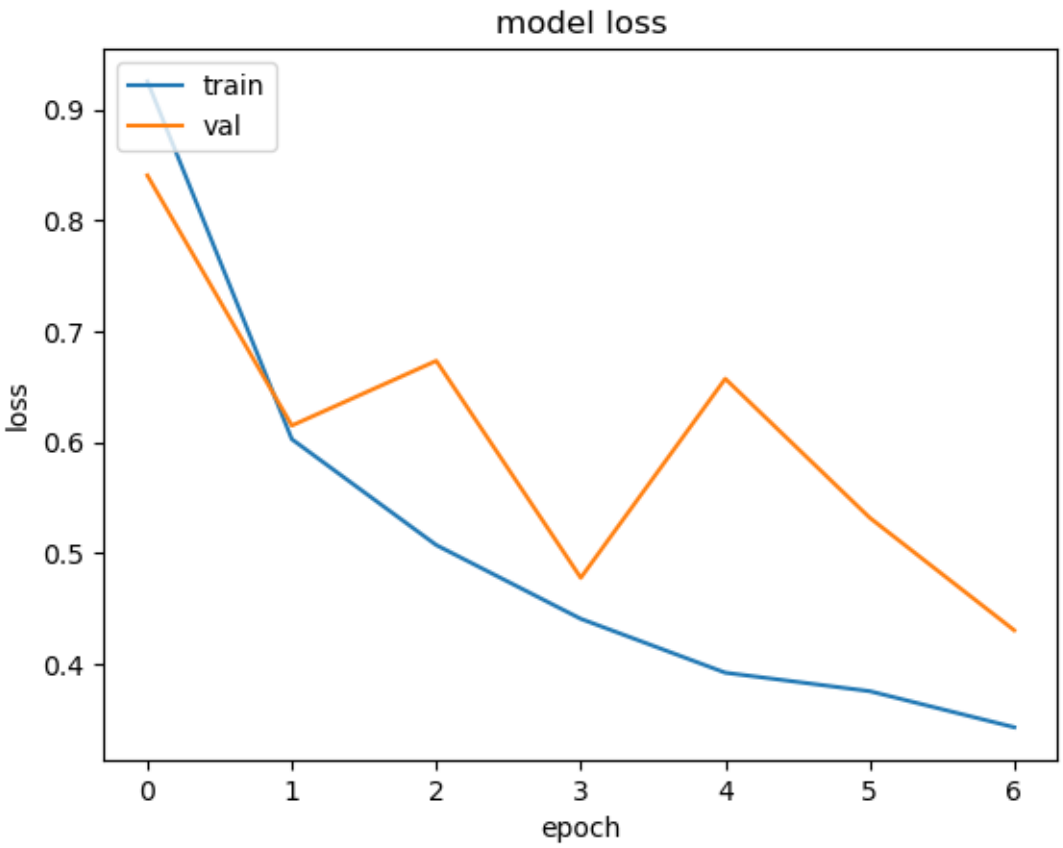
```
Using cpu device
Epoch 1: Validation Loss: 0.84, Validation Accuracy: 0.685
Epoch 2: Validation Loss: 0.61, Validation Accuracy: 0.746
Epoch 3: Validation Loss: 0.67, Validation Accuracy: 0.735
Epoch 4: Validation Loss: 0.48, Validation Accuracy: 0.805
Epoch 5: Validation Loss: 0.66, Validation Accuracy: 0.735
Epoch 6: Validation Loss: 0.53, Validation Accuracy: 0.790
Epoch 7: Validation Loss: 0.43, Validation Accuracy: 0.843
```

**2.1.4 Examine loss plots [Required for grad, Bonus for undergrad]** To achieve full credit, your loss must be decreasing and accuracy must be increasing gradually.

[14]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

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

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

plt.plot(train_loss)
```

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

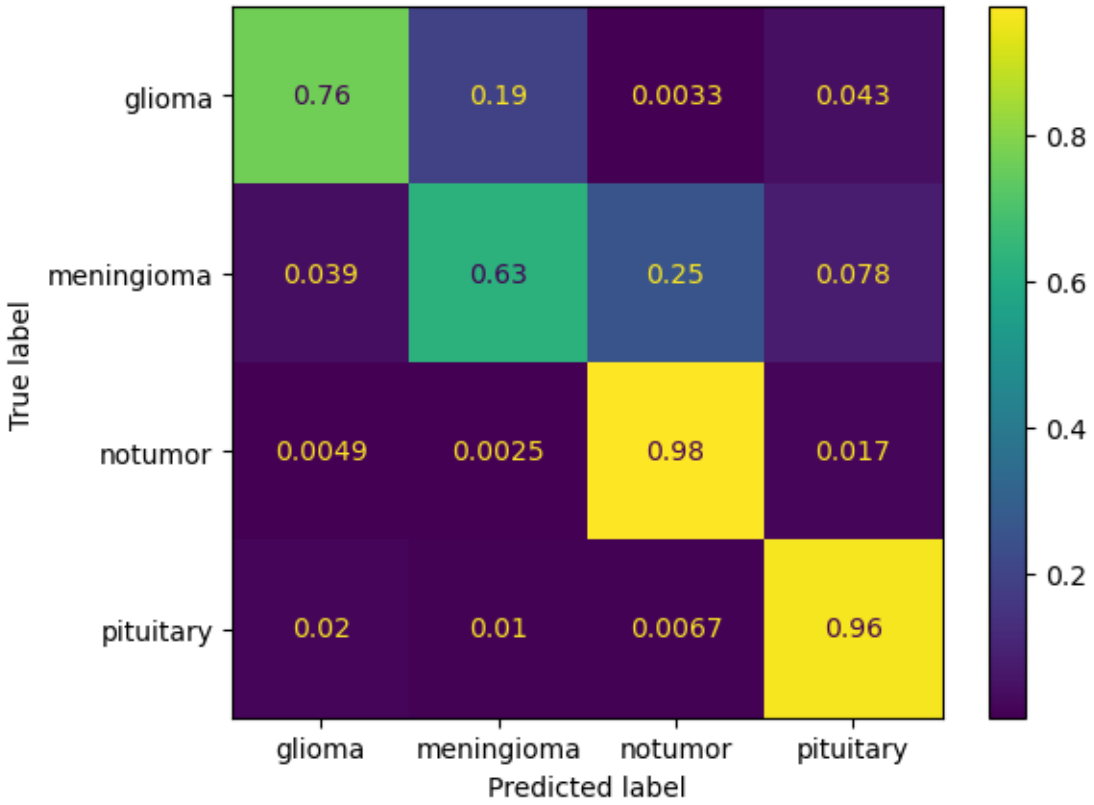





[ ]:
```
from utilities.localtests import TestCNN

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

**2.1.5 Examine Confusion Matrix [Required for grad, Bonus for undergrad]** To get full credit, all the diagonal entries in your confusion matrix should be above 0.5. This ensures our model has reasonable accuracy across all classes. You can use the test below to verify the same.

[15]:
```
TestCNN("test_cnn_confusion_matrix").test_cnn_confusion_matrix(trainer,␣
↪testloader)
```




```
test_cnn_confusion_matrix passed!
```

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

[16]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

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

from sklearn.metrics import ConfusionMatrixDisplay, accuracy_score

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

```
Test accuracy: 0.8428680396643783
```

### 1.8.2 2.2 Exploring Deep CNN Architectures [2.5% Bonus for All] [W]

**2.2.1 Abating Vanishing Gradients [1.5% Bonus for All]** 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.

A common issue with deep CNN architectures is the vanishing gradient problem. Small weights deep into the network can contribute smaller and smaller gradients as backpropagation travels back in the network, and the initial layers may end up seeing very small gradient updates that can encounter floating point accuracy errors depending on the hardware, resulting in frozen weights that are never updated. Vanishing gradients accumulate along the networks layers, which becomes a big problem in deeper architectures.

One of the first successful deep CNNs is known as ResNet (Residual Networks) which uses what are known as skip connections to allow the flow of data through the network to skip layers. Take a moment to explore how ResNet tackles the vanishing gradient problem by reading the original research paper here: https://arxiv.org/abs/1512.03385.

**Question:** In 1-2 sentences, explain how residual 'skip connections' address the vanishing gradient problem.

---

Residual skip connections address the vanishing gradient problem by creating a direct path that allows gradients to flow uninterrupted during backpropagation. This ensures that the gradient signal can effectively reach and update the weights in the earlier layers of deep networks, rather than diminishing to zero as it passes through many transformation layers.

---

**2.2.2 Abating Internal Covariate Shift [1.0% Bonus for All]** Another common strategy to mitigate gradient issues and build deeper networks is to use a learnable Batch Normalization layer after each activation layer, and before the nonlinearity (i.e. $\sigma(BN(Wx+b))$).

$$y = \frac{x - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}}\gamma + \beta}$$

Where the mean $\mu_{\mathcal{B}=\frac{1}{|\mathcal{B}|}\sum_{x\in\mathcal{B}}x}$ and variance $\sigma_{\mathcal{B}^2=\frac{1}{|\mathcal{B}|}\sum_{x\in\mathcal{B}}(x-\mu_{\mathcal{B}})^2}$ are taken over the current training mini-batch $\mathcal{B}$. The learnable parameters $\gamma$ and $\beta$ are updated in backpropagation. Read through the original Batch Normalization paper, particularly sections 2 and 3 for a better grasp of the formulation and motivation behind activation normalization.

**Question:** In 3-5 sentences, explain the problem of internal covariate shift and how Batch Normalization helps to reduce this issue.

---

Internal covariate shift is when the distribution of activations in each layer keeps changing during training as earlier layers' weights are updated. This means deeper layers are constantly having to re-adapt to a new distribution, which slows and destabilizes learning. Batch Normalization addresses this by normalizing the inputs to have a zero mean and unit variance, then re-scaling and shifting them with learned parameters. This stabilizes the input distribution across the network, allowing for faster convergence and the use of higher learning rates.

---

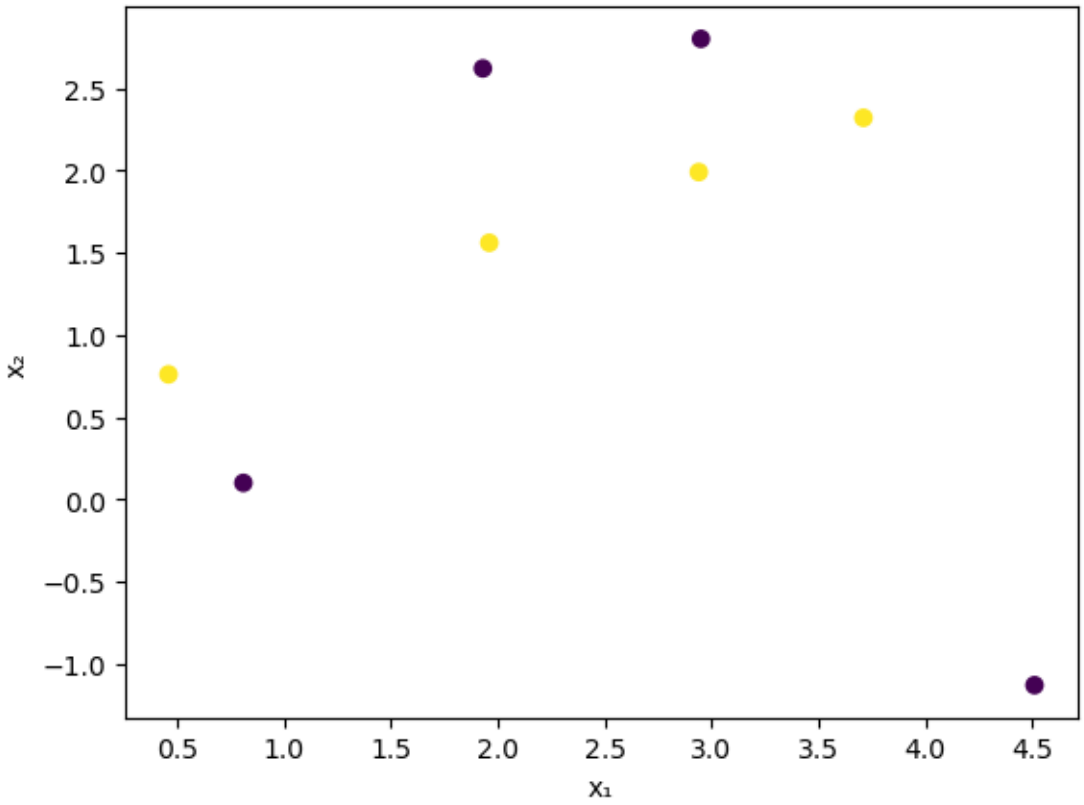
## 1.9 Q3: SVM [20 pts] [P] | [W]

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. If the data has no perfectly separating hyperplane, it uses a regularization constant $C$ to weigh sometimes competing objectives: maximizing train accuracy and maximizing the margin size. SVMs are a pretty powerful tool with some adaptation, but out-of-the-box, they do need some help.

### 1.9.1 3.1 Picking Performant Constructions [5pts] [W]

For this exercise, we will use `SVC` from scikit-learn with a linear kernel. We also have some data that we'd like to classify. Here's a plot of that data:

[30]:
```
x_1 = np.array([0.46, 1.96, 2.94, 3.71, 0.81, 1.93, 2.95, 4.51])  # (N,)
x_2 = np.array([0.76, 1.56, 1.99, 2.32, 0.10, 2.62, 2.80, -1.13])  # (N,)
y = np.array([1, 1, 1, 1, -1, -1, -1, -1])  # (N,)
X = np.column_stack((x_1, x_2))  # (N, 2)

plt.figure()
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.xlabel("x ")
plt.ylabel("x ")
plt.show()
```



As it stands, there is no hyperplane that would separate these points. But surely there's some information here! N=8 is not really enough to get great inductive guarantees about any trends about what this data might be, but let's plug away anyways. We'll use a transformation $f : \mathbb{R}^2 \to \mathbb{R}$ that maps the 2 features into a single combined feature. Even with the dimensionality reduction, if we pick a good function, the data may still be separable.

We have provided several candidate functions that transform the dataset. It's your job to determine which of the candidate functions would result in linear separability (it may be more than 1).

There's a few ways you could investigate this. You could just do the math, you could apply the transforms and run `SVC(kernel='linear')`, you could plot the result, etc. For this question, you do not need to show your work. Any evidence you show might help us grant you extra credit, but you just need to indicate whether each kernel results in linear separability.

[31]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

# These are the candidate transforms that you'll need to evaluate


def f1(feature_1, feature_2):
    return feature_1**2 + feature_2**2


def f2(feature_1, feature_2):
    return feature_1**2


def f3(feature_1, feature_2):
    return np.exp(-(feature_1**2 + feature_2**2) / 2)


def f4(feature_1, feature_2):
    return np.abs(1 - feature_2)


def f5(feature_1, feature_2):
    return feature_2 - feature_1
```

[ ]:
```
#########################################################################################
# OPTIONAL: Investigate the given functions with some code-based approaches      ␣
 ↪      #
#########################################################################################
```

```
from sklearn.svm import SVC

linear_model = SVC(kernel="linear")

candidateFunctions = {'f1': f1, 'f2': f2, 'f3': f3, 'f4': f4, 'f5': f5}

for name, func in candidateFunctions.items():
    xNew = func(x_1, x_2).reshape(-1, 1)
    linear_model.fit(xNew, y)
    acc = linear_model.score(xNew, y)
    print(f"{name:<10} | {acc:<10.2f} | {acc == 1.0}")


##########################################################################################
#                                    END OF YOUR CODE                                    ␣
  ↪      #
##########################################################################################
```

```
f1         | 0.50       | False
f2         | 0.50       | False
f3         | 0.50       | False
f4         | 0.75       | False
f5         | 0.62       | False
```

OPTIONAL: Investigate the given functions with some written approaches

Indicate all of the given functions that cause the provided data to be linearly separable by a hyperplane. (more than 1 function may apply) [5pts]

**Your Answer Here:**

---

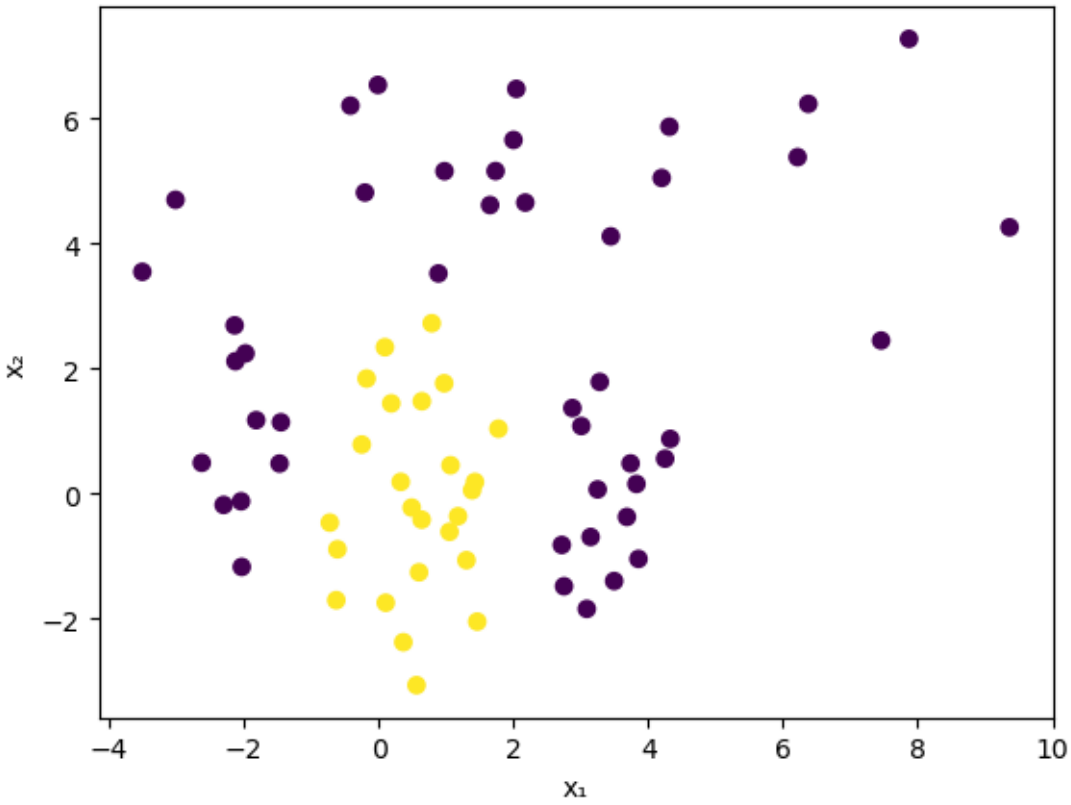
none of these functions achieve perfect linear separability

---

### 1.9.2 3.2 Custom Feature Engineering [5pts] [P]

[24]:
```
df = pd.read_csv("data/artificial_data.csv")

X = df[["x_1", "x_2"]].to_numpy()  # (N, 2)
y = df["y"].to_numpy()  # (N,)
```

[25]:
```
# Plot the data
plt.figure()
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.xlabel("x ")
plt.ylabel("x ")
plt.show()
```




We have the above 2D dataset. Natively, SVM won't be able to handle this with 100% accuracy because the data is not linearly separable. There's no line in the Cartesian plane that would separate these labels.

In `svm.py` complete `feature_construction`. You must take the data here and transform it, so that this data is perfectly linearly separable, and a SVM would be able to fit to the data with 100% training accuracy.

We won't provide a local test for this. If you'd like to try the SVM fit (and find out what's not fitting), you'll need to set up the code yourself. Remember to set the kernel type to 'linear' in the SVC, `SVC(kernel='linear')`. The default kwarg for `kernel` makes the algorithm way more performant, which we'll explore later.

### 1.9.3 3.3 Kernel Trick [10 pts] [P]

In the previous sections, we used feature engineering to make our data linearly separable. In this section, you will explore another, more powerful method to make data linearly separable - "the kernel trick".

The kernel trick makes data linearly separable by projecting it to a higher dimensionality space. This allows for data that might not be linearly separable in one dimensionality to become linearly separable in another. Though, as you'll see, there's never any actual projection happening.

**Justification for the trick** If you're not interested in the derivation, you can collapse and skip this section by clicking on the arrow to the left of this cell.

We have a dataset $(x^{\{1\}}, y^{\{1\}}), (x^{\{2\}}, y^{\{2\}}), \ldots, (x^{\{N\}}, y^{\{N\}})$ where each data-label pair is $(x^{\{i\}}, y^{\{i\}}) \in \mathbb{R}^D \times \{-1, 1\}$.

What we want is a hyperplane of the form $w \cdot x + b = 0$ that separates all data with labels -1 and 1 as purely as possible. So, our decision function is just $f(x) = \mathsf{sign}(w \cdot x + b)$. How can we derive those weights?

A correct classifier would satisfy $y^{\{i\}}(w \cdot x^{\{i\}} + b) > 0$. If the label is negative and the prediction is negative, the product should be positive. If the label is positive and the prediction is positive, the product should be positive. There are two problems with this. 1. We can scale $w$ and $b$ and get equivalent results. 2. We have no margin for error! A perfectly valid classifier might get arbitrarily close to the train points. However, we must assume that the underlying distribution doesn't belong in a convex hull of the train points. We need to keep our target hyperplane as far away from the train points as we can.

We can just strengthen our constraints. We demand $y^{\{i\}}(w \cdot x^{\{i\}} + b) \geq 1$ (widen the required margin) but also seek to maximize that margin.

Because of our choice of widened margin 1, the margin planes are $w \cdot x^{\{i\}} + b = 1$ and $w \cdot x^{\{i\}} + b = -1$. The distance between the two planes can be derived from point to plane distance. Choose $x_+$ on the positive plane. The distance from that point to the negative plane is $\frac{(w \cdot x_+ + b) - (-1)}{\|w\|}$. Being on the positive plane means $w \cdot x_+ + b = 1$, so the distance between those planes is $\frac{(1) - (-1)}{\|w\|} = \frac{2}{\|w\|}$. We need to maximize this margin, but the derivative isn't great, so we need an alternative form. Being careful with constants, signs, and monotonic composition, we get the following form:

$$\max \frac{2}{\|w\|} = \max \frac{1}{\|w\|} = \min \|w\| = \min \|w\|^2 = \min \frac{\|w\|^2}{2}$$

and the derivative is much more helpful, leaving us with the following problem:

$$\begin{cases} \min \frac{\|w\|^2}{2} \\ \text{s.t. } y^{\{i\}}(w \cdot x^{\{i\}} + b) \geq 1 \end{cases}$$

One issue with this is that most data unfortunately won't have any solutions here. We need to permit margin violations and even misclassifications with a slack variable. We should of course minimize this slack, since having misclassifications in training is not ideal. This complicates the problem:

$$\begin{cases} \min \frac{\|w\|^2}{2} + C \cdot \sum_{i=1}^{N} \xi_i \\ \text{s.t.} \quad y^{\{i\}}(w \cdot x^{\{i\}} + b) \geq 1 - \xi_i \\ \qquad \xi_i \geq 0 \end{cases}$$

*Note: $C$ is a hyperparameter that dictates how much you care about misclassifications vs. how much you care about margin.*

To solve, we construct the Lagrangian:

$$\mathcal{L} = \frac{\|w\|^2}{2} + C \cdot \sum_i \xi_i - \sum_i \lambda_i [y^{\{i\}}(w \cdot x^{\{i\}} + b) - 1 + \xi_i] - \sum_i \lambda_i' [\xi_i]$$

Then differentiate:

$$\frac{d\mathcal{L}}{dw} = w - \sum_i \lambda_i y^{\{i\}} x^{\{i\}} := 0 \quad \Longrightarrow \quad w = \sum_i \lambda_i y^{\{i\}} x^{\{i\}}$$

$$\frac{d\mathcal{L}}{db} = -\sum_i \lambda_i y^{\{i\}} := 0 \quad \Longrightarrow \quad \sum_i \lambda_i y^{\{i\}} = 0$$

$$\frac{d\mathcal{L}}{d\xi_i} = C - \lambda_i - \lambda_i' := 0 \quad \Longrightarrow \quad \lambda_i = C - \lambda_i' \quad \Longrightarrow \quad \lambda_i \leq C$$

If we can solve for the Lagrange multipliers in the Lagrangian dual, then we have our $w$ and $b$.

*If you want to exercise your KKT chops, you'll notice we don't actually have b here, only w. The solution for b precipitates from the complementary slackness constraints. You can also derive it after you have w, if you notice what $\lambda_i$ means. If $\lambda_i \neq 0$, then $x^{\{i\}}$ must be a support vector, and we impose a strict margin of $\geq 1/\|w\|$ from any support vector to the hyperplane. That's a linear programming problem.*

Now, back to the dervation. Impose the identities derived from the stationarity conditions back onto the Lagrangian.

$$\mathcal{L} = \frac{1}{2}\|w\|^2 + C\sum_i \xi_i - \sum_i \lambda_i y^{\{i\}}(w \cdot x^{\{i\}}) - \sum_i \lambda_i y^{\{i\}} b + \sum_i \lambda_i - \sum_i \lambda_i \xi_i - \sum_i \lambda_i' \xi_i$$

$$\mathcal{L} = \frac{1}{2}\left\|\left[\sum_i \lambda_i y^{\{i\}} x^{\{i\}}\right]\right\|^2 - \sum_i \lambda_i y^{\{i\}} \left(\left[\sum_j \lambda_j y^{\{j\}} x^{\{j\}}\right] \cdot x^{\{i\}}\right) - b \cdot [0] + \sum_i \lambda_i - \sum_i \lambda_i \xi_i - \sum_i \lambda_i' \xi_i + C \sum_i \xi_i$$

$$\mathcal{L} = \frac{1}{2} \cdot \left(\left[\sum_i \lambda_i y^{\{i\}} x^{\{i\}}\right] \cdot \left[\sum_j \lambda_j y^{\{j\}} x^{\{j\}}\right]\right) - \sum_i \lambda_i y^{\{i\}} \left(\left[\sum_j \lambda_j y^{\{j\}} x^{\{j\}}\right] \cdot x^{\{i\}}\right) + \sum_i \lambda_i - \sum_i \lambda_i \xi_i - \sum_i \lambda_i' \xi_i + \sum_i$$

$$\mathcal{L} = \frac{1}{2} \cdot \left(\sum_i \sum_j \lambda_i \lambda_j y^{\{i\}} y^{\{j\}} (x^{\{i\}} \cdot x^{\{j\}})\right) - \left(\sum_i \sum_j \lambda_i \lambda_j y^{\{i\}} y^{\{j\}} (x^{\{i\}} \cdot x^{\{j\}})\right) + \sum_i \lambda_i + \sum_i \xi_i (C - \lambda_i - \lambda_i')$$

$$\mathcal{L} = -\frac{1}{2} \cdot \left(\sum_i \sum_j \lambda_i \lambda_j y^{\{i\}} y^{\{j\}} (x^{\{i\}} \cdot x^{\{j\}})\right) + \sum_i \lambda_i + \sum_i \xi_i [0]$$

$$\mathcal{L} = -\frac{1}{2} \cdot \left(\sum_i \sum_j \lambda_i \lambda_j y^{\{i\}} y^{\{j\}} (x^{\{i\}} \cdot x^{\{j\}})\right) + \sum_i \lambda_i$$

We can now set up the dual, and solving for the Lagrange multipliers will give a form for $w$, which is precisely what fitting an SVM is.

$$\begin{cases} \max_\lambda \ -\frac{1}{2}\left(\sum_i \sum_j \lambda_i \lambda_j y^{\{i\}} y^{\{j\}} (x^{\{i\}} \cdot x^{\{j\}})\right) + \sum_i \lambda_i \\ \text{s.t.} \quad \sum_i \lambda_i y^{\{i\}} \quad \left(\text{from the derivative } \frac{d\mathcal{L}}{db}\right) \\ \qquad 0 \leq \lambda_i \leq C \quad \left(\text{from the derivative } \frac{d\mathcal{L}}{d\xi_i}\right) \end{cases}$$

**The Kernel Trick** In case you skipped the derivation, this is fitting an SVM looks like:

$$\begin{cases} \max_\lambda \ -\frac{1}{2}\left(\sum_i \sum_j \lambda_i \lambda_j y^{\{i\}} y^{\{j\}} (x^{\{i\}} \cdot x^{\{j\}})\right) + \sum_i \lambda_i \\ \text{s.t.} \quad \sum_i \lambda_i y^{\{i\}} \\ \qquad 0 \leq \lambda_i \leq C \end{cases}$$

We need to find the support parameters $\lambda_i$ that maximize this expression. These $\lambda_i$ will let us solve for $w$ and $b$.

The so-called "kernel trick" is actually a technically wrong extension of the derived formula for a maximum-margin hyperplane. Take a look at the learning objective. Zoom in on one term:

$$(x^{\{i\}} \cdot x^{\{j\}})$$

This functionally measures the similarity of datum $i$ to datum $j$ directly. The trouble with SVM, as we saw in the previous sections, was that the data may just not be linearly separable in the features that we have collected. In the feature engineering, you may have made polynomial or exponential or any number of transformations to try to get the data to be separable. Let's do that here! Make up some mapping $\phi : \mathbb{R}^D \to \mathbb{R}^{D'}$ that maps the data in the $D$ feature space to some new space of dimensionality $D'$ (ideally with $D' \gg D$). Then, we'll just map $x^{\{i\}}$ and $x^{\{j\}}$ into that new space before taking their dot product. If we pick a sensible/continuous/smooth-ish mapping, then data that's similar in the low dimensional space will stay similar in the high dimensional space, but the curse of dimensionality becomes the blessing of dimensionality! We get these tight knit groups of data that are strongly similar in many of the feature pairs.

$$\begin{cases} \max_\lambda \ -\frac{1}{2}\left(\sum_i \sum_j \lambda_i \lambda_j y^{\{i\}} y^{\{j\}} (\phi(x^{\{i\}}) \cdot \phi(x^{\{j\}}))\right) + \sum_i \lambda_i \\ \text{s.t.} \quad \sum_i \lambda_i y^{\{i\}} \\ \qquad 0 \leq \lambda_i \leq C \end{cases}$$

If you have a good function $\phi$ in mind for your problem, that's great! But generally, there's not an excellent closed form solution, so we may just want to make the projected space $D'$ massive so we hit at least a few useful features. If we blow up $D'$, then representing $\phi$ and calculating $(\phi(x^{\{i\}}) \cdot \phi(x^{\{j\}}))$ might become quite expensive. This is the eponymous kernel trick! Replace $(\phi(x^{\{i\}}) \cdot \phi(x^{\{j\}}))$ with a kernel matrix $K[i, j]$ with shape (N, N). $K[i, j]$ describes the similarity of datum $i$ to datum $j$. So, the kernel can be any matrix that looks like pairwise inner products. We only need: - kernel needs to be symmetric. (since $a \cdot b = b \cdot a$) - kernel needs to be positive-semidefinite. (Mercer's Theorem)

$$\begin{cases} \max_\lambda \ -\frac{1}{2}\left(\sum_i \sum_j \lambda_i \lambda_j y^{\{i\}} y^{\{j\}} K[i, j]\right) + \sum_i \lambda_i \\ \text{s.t.} \quad \sum_i \lambda_i y^{\{i\}} \\ \qquad 0 \leq \lambda_i \leq C \end{cases}$$

So, when the $\phi$ you want makes calculating $(\phi(x^{\{i\}}) \cdot \phi(x^{\{j\}}))$ intractible, just find some closed-form for the solution that you can compute quickly and stuff it into a kernel.

**3.3.1 Build a Kernel [5pts] [P]** In `svm.py` complete `kernel_construction`. Given a set of data and a callable function $\phi$, build up the kernel matrix element-by-element. You're fine to use loops here, since the point of this exercise is that this is an inefficient way to build kernels in

general. No need to make your code timely (so we won't give you massive inputs that stall out the Gradescope time).

The kernel that you make can genuinely be passed into scikit-learn's SVC. It will query your matrix and build the corresponding SVM.

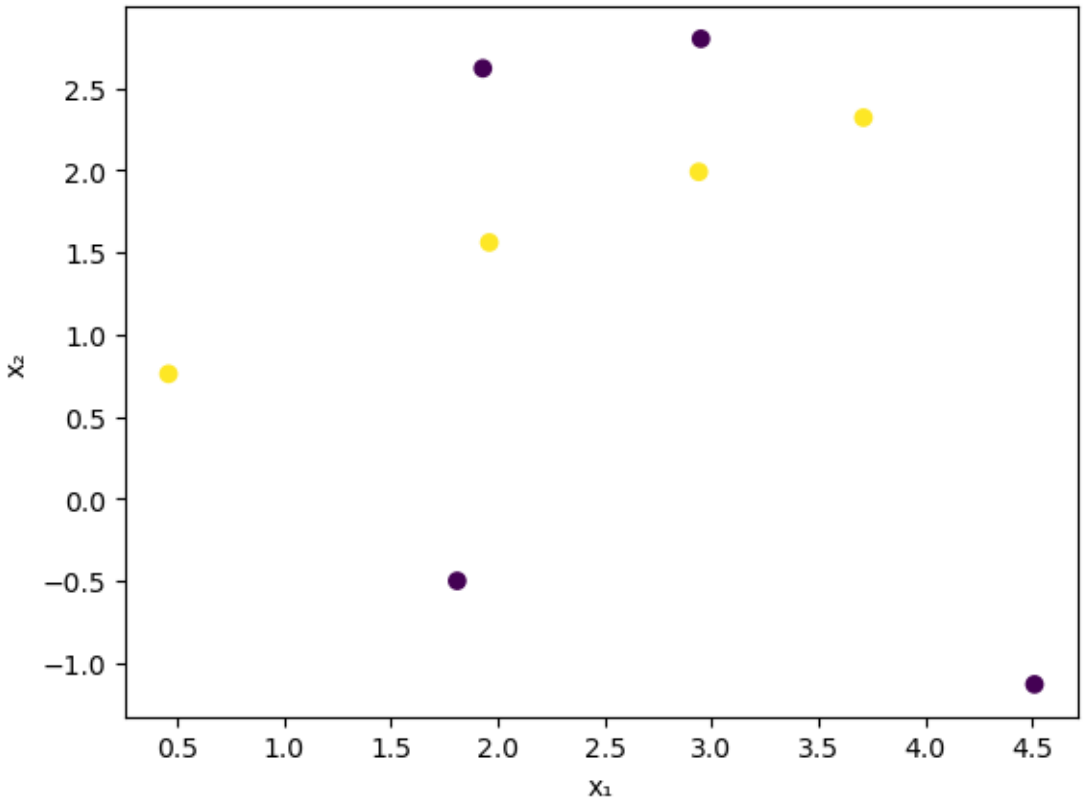
[36]:
```
from utilities.localtests import TestSVM

TestSVM("test_kernel").test_kernel()
```

```
test_kernel passed!
```

Now, the kernel that you make is going to be questionably useful. It's tricky to define a good map for a general problem. Let's see if a phi generating degree 2 polynomials works for the following problem:

[37]:
```
import svm
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC

# Example data
x_1 = np.array([0.46, 1.96, 2.94, 3.71, 1.81, 1.93, 2.95, 4.51])
x_2 = np.array([0.76, 1.56, 1.99, 2.32, -0.50, 2.62, 2.80, -1.13])
y = np.array([1, 1, 1, 1, -1, -1, -1, -1])
X = np.column_stack((x_1, x_2))
plt.figure()
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.xlabel("x ")
plt.ylabel("x ")
plt.show()


# Feature construction
def phi(datum):
    projected = np.zeros((5,))
    projected[0] = datum[0]
    projected[1] = datum[1]
    projected[2] = datum[0] * datum[1]
    projected[3] = datum[0] ** 2
    projected[4] = datum[1] ** 2
    return projected


# Fit with kernel
custom_kernel = svm.kernel_construction(X, phi)
svm_model = SVC(kernel="precomputed")
svm_model.fit(custom_kernel, y)
y_pred = svm_model.predict(custom_kernel)
acc = accuracy_score(y, y_pred)
```

```
print("Training accuracy:", acc)
```

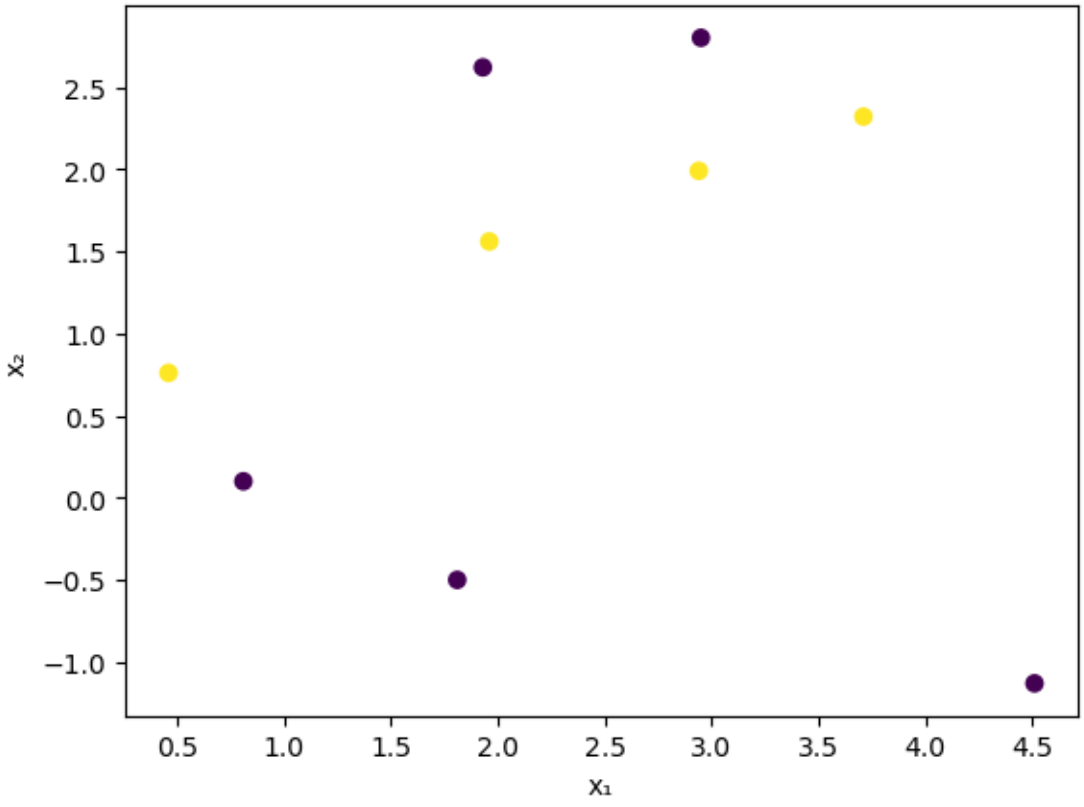



```
Training accuracy: 1.0
```

You should have reached a training accuracy of 100%, but this kernel really does not generalize well. With a new, less fortunately placed data point, the data is no longer separable.

[38]:
```
import svm
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC

# Example data
x_1 = np.array([0.46, 1.96, 2.94, 3.71, 1.81, 0.81, 1.93, 2.95, 4.51])
x_2 = np.array([0.76, 1.56, 1.99, 2.32, -0.50, 0.10, 2.62, 2.80, -1.13])
y = np.array([1, 1, 1, 1, -1, -1, -1, -1, -1])
X = np.column_stack((x_1, x_2))
plt.figure()
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.xlabel("x ")
plt.ylabel("x ")
plt.show()
```

```python
# Feature construction
def phi(datum):
    projected = np.zeros((5,))
    projected[0] = datum[0]
    projected[1] = datum[1]
    projected[2] = datum[0] * datum[1]
    projected[3] = datum[0] ** 2
    projected[4] = datum[1] ** 2
    return projected


# Fit with kernel
custom_kernel = svm.kernel_construction(X, phi)
svm_model = SVC(kernel="precomputed")
svm_model.fit(custom_kernel, y)
y_pred = svm_model.predict(custom_kernel)
acc = accuracy_score(y, y_pred)
print("Training accuracy:", acc)
```




```
Training accuracy: 0.8888888888888888
```

**3.3.2 Build a Known Kernel (RBF) [5pts] [P]** Our degree 2 polynomial kernel clearly is not the greatest tool available. It fails to fully fit on this data, and if this were a larger, more real dataset, it would probably lack significant generalizability. There are better kernels that you can use. We'll explore a kernel that actually projects into infinite dimensions: the radial basis function kernel (RBF).

**What does it mean to project into infinite dimensions?** If you're not interested in the derivation, you can collapse and skip this section by clicking on the arrow to the left of this cell.

The form for the RBF kernel itself is really simple. You define the similarity between two points $x$ and $y$ as the distance between them $\|x - y\|^2$ composed with some decreasing function. A natural choice is $1/x$ or $-x$, but RBF uses the similarly reasonable $\exp(-\gamma x)$, where $\gamma$ is a free hyperparameter. Thus, we have:

$$K[i, j] = \exp\left(-\gamma \left\|x^{\{i\}} - x^{\{j\}}\right\|^2\right)$$

This definition is how you really should intuitively approach RBF. It's a similarity metric based on distance. But we did define $K[i, j] = (\phi(x^{\{i\}}) \cdot \phi(x^{\{j\}}))$, so this begs the question, what would the implicit mapping $\phi$ be for RBF?

$$K[i,j] = \exp\left(-\gamma \left\|x^{\{i\}} - x^{\{j\}}\right\|^2\right)$$

$$K[i,j] = \exp\left(-\gamma \left(\left\|x^{\{i\}}\right\|^2 + \left\|x^{\{j\}}\right\|^2 - 2x^{\{i\}} \cdot x^{\{j\}}\right)\right)$$

$$K[i,j] = \exp\left(-\gamma \left\|x^{\{i\}}\right\|^2\right) \exp\left(-\gamma \left\|x^{\{j\}}\right\|^2\right) \exp\left(2\gamma(x^{\{i\}} \cdot x^{\{j\}})\right)$$

Use the Taylor series for exp

$$K[i,j] = \exp\left(-\gamma \left\|x^{\{i\}}\right\|^2\right) \exp\left(-\gamma \left\|x^{\{j\}}\right\|^2\right) \sum_{m=0}^{\infty} \frac{(2\gamma)^m}{m!} (x^{\{i\}} \cdot x^{\{j\}})^m$$

Note that $(x^{\{i\}} \cdot x^{\{j\}})^m$ is a D-termed multinomial, so we can use the multinomial theorem

Let $\alpha$ be our D-long multi-index (positive integer sequence summing to m)

$$K[i,j] = \exp\left(-\gamma \left\|x^{\{i\}}\right\|^2\right) \exp\left(-\gamma \left\|x^{\{j\}}\right\|^2\right) \sum_{m=0}^{\infty} \left[\frac{(2\gamma)^m}{m!} \cdot \sum_{\forall \alpha,\, \Sigma\alpha = m} \left[\binom{m}{\alpha} \prod_{d=1}^{D} (x_d^{\{i\}} x_d^{\{j\}})^{\alpha_d}\right]\right]$$

$$K[i,j] = \exp\left(-\gamma \left\|x^{\{i\}}\right\|^2\right) \exp\left(-\gamma \left\|x^{\{j\}}\right\|^2\right) \sum_{m=0}^{\infty} \sum_{\forall \alpha,\, \Sigma\alpha = m} \left[\frac{(2\gamma)^m}{m!} \cdot \frac{m!}{\prod_{d=1}^{D} \alpha_d!} \prod_{d=1}^{D} (x_d^{\{i\}} x_d^{\{j\}})^{\alpha_d}\right]$$

$$K[i,j] = \exp\left(-\gamma \left\|x^{\{i\}}\right\|^2\right) \exp\left(-\gamma \left\|x^{\{j\}}\right\|^2\right) \sum_{m=0}^{\infty} \sum_{\forall \alpha,\, \Sigma\alpha = m} \left[\frac{(2\gamma)^m}{\prod_{d=1}^{D} \alpha_d!} \prod_{d=1}^{D} \left(x_d^{\{i\}}\right)^{\alpha_d} \prod_{d=1}^{D} \left(x_d^{\{j\}}\right)^{\alpha_d}\right]$$

Gather terms to make matching $i$ and $j$ parts

$$K[i,j] = \sum_{m=0}^{\infty} \sum_{\forall \alpha,\, \Sigma\alpha = m} \left[\exp\left(-\gamma \left\|x^{\{i\}}\right\|^2\right) \cdot \exp\left(-\gamma \left\|x^{\{j\}}\right\|^2\right) \cdot \frac{(2\gamma)^m}{\prod_{d=1}^{D} \alpha_d!} \cdot \prod_{d=1}^{D} \left(x_d^{\{i\}}\right)^{\alpha_d} \cdot \prod_{d=1}^{D} \left(x_d^{\{j\}}\right)^{\alpha_d}\right]$$

$$K[i,j] = \sum_{m=0}^{\infty} \sum_{\forall \alpha,\, \Sigma\alpha = m} \left[\exp\left(-\gamma \left\|x^{\{i\}}\right\|^2\right) \cdot \sqrt{\frac{(2\gamma)^m}{\prod_{d=1}^{D} \alpha_d!}} \cdot \prod_{d=1}^{D} \left(x_d^{\{i\}}\right)^{\alpha_d}\right] \cdot \left[\exp\left(-\gamma \left\|x^{\{j\}}\right\|^2\right) \cdot \sqrt{\frac{(2\gamma)^m}{\prod_{d=1}^{D} \alpha_d!}} \cdot \prod_{d=1}^{D} \left(x_d^{\{j\}}\right)^{\alpha_d}\right]$$

Now, WLOG, let's reindex the double-sum for clarity.

Let $A = [\alpha_1, \alpha_2, \cdots]$ be the infinite sequence of all multi-index sequences we need to consider for all $m$

$$K[i,j] = \sum_{\forall \alpha \in A} \left[\exp\left(-\gamma \left\|x^{\{i\}}\right\|^2\right) \cdot \sqrt{\frac{(2\gamma)^{\Sigma\alpha}}{\prod_{d=1}^{D} \alpha_d!}} \cdot \prod_{d=1}^{D} \left(x_d^{\{i\}}\right)^{\alpha_d}\right] \cdot \left[\exp\left(-\gamma \left\|x^{\{j\}}\right\|^2\right) \cdot \sqrt{\frac{(2\gamma)^{\Sigma\alpha}}{\prod_{d=1}^{D} \alpha_d!}} \cdot \prod_{d=1}^{D} \left(x_d^{\{j\}}\right)^{\alpha_d}\right]$$

Let $\phi$ be a map $\mathbb{R}^D \to \mathbb{R}^\infty$ (technically, $\mathbb{R}^D \to \ell^2 \cong \mathcal{H}$), with an infinite number of components, each corres

Let $\phi_\alpha$ denote the scalar component of $\phi$ corresponding to a particular multi-index sequence $\alpha$

$$\phi_\alpha(x) := \exp\left(-\gamma \left\|x\right\|^2\right) \cdot \sqrt{\frac{(2\gamma)^{\Sigma\alpha}}{\prod_{d=1}^{D} \alpha_d!}} \cdot \prod_{d=1}^{D} (x_d)^{\alpha_d}$$

Plugging in this definition,

$$K[i,j] = \sum_{\forall \alpha \in A} \phi_\alpha(x^{\{i\}}) \cdot \phi_\alpha(x^{\{j\}})$$

$$K[i,j] = \phi(x^{\{i\}}) \cdot \phi(x^{\{j\}})$$

**Implementation** So, as shown, the simple kernel

$$K[i,j] = \exp\left(-\gamma \left\|x^{\{i\}} - x^{\{j\}}\right\|^2\right)$$

really does correspond uniquely to an infinite inner-product with the infinite mapping:

$$K[i,j] = \phi(x^{\{i\}}) \cdot \phi(x^{\{j\}})$$

$$\phi(x) = \left[ \exp\left(-\gamma \left\|x\right\|^2\right) \cdot \sqrt{\frac{(2\gamma)^{\Sigma\alpha_1}}{\prod_{d=1}^{D} \alpha_{1,d}!}} \cdot \prod_{d=1}^{D} (x_d)^{\alpha_{1,d}}, \quad \exp\left(-\gamma \left\|x\right\|^2\right) \cdot \sqrt{\frac{(2\gamma)^{\Sigma\alpha_2}}{\prod_{d=1}^{D} \alpha_{2,d}!}} \cdot \prod_{d=1}^{D} (x_d)^{\alpha_{2,d}}, \quad \ldots \right]$$

Since the vector components decrease very quickly as the values in the multi-indices increase (and you can't have infinitely many multi-indices below some bound), up to some precision, you could map out an approximation for $\phi(x^{\{i\}})$ and do that for all pairs and fill out the kernel in the same way you filled out the previous polynomial kernel. But the essence of the kernel trick is that you really never need to deal with the implicit functor $\phi$, you just need to build out the kernel.

In `svm.py` complete `rbf_kernel`.

You're also fine to use loops here. Though RBF tends to be quite good, one major weakness of kernel-based SVM is that it requires $\mathcal{O}(N^2)$ space. While we're getting better volumetric storage density and storage cost, if you have 1 million datapoints, which isn't even close to a lot these days, you need 500 billion kernel entries. No point in getting constant-time speedups if you can't even allocate enough storage to store the output. So, no need to make your code timely (we won't give you massive inputs that stall out the Gradescope time).

[39]:
```
from utilities.localtests import TestSVM

TestSVM("test_rbf_kernel").test_rbf_kernel()
```

```
test_rbf_kernel passed!
```

### 1.10 Q4: Next Character Prediction using Recurrent Neural Networks (RNNs) [7.5% Bonus for All] [P] | [W]

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

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

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

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

### 1.10.1 Data Preparation

- We'll use Shakespeare's Macbeth from Project Gutenberg
- We vectorize the text by treating every character in our text as an individual unit (e.g., 'macbeth' -> ['m', 'a', 'c'...])
- We use a dictionary to store this mapping: {'a':1, 'b':2, 'c':3, ...}
- This mapping enables bidirectional conversion between characters and integers for model input and output interpretation
- We assign each each character a learnable embedding vector
- Create fixed-sized batches of characters using sliding window approach
- For example, with text "macbeth" (context window=4):

```
Window 1: "macb" → predict "e"
Window 2: "acbe" → predict "t"
Window 3: "cbet" → predict "h"
```

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

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

[40]:
```
################################
### 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}")
```

```
Length of Corpus:  102176
Vocabulary Map:  {' ': 0, '!': 1, ',': 2, '-': 3, '.': 4, '1': 5, '3': 6, '5':
7, ':': 8, ';': 9, '?': 10, 'a': 11, 'b': 12, 'c': 13, 'd': 14, 'e': 15, 'f':
16, 'g': 17, 'h': 18, 'i': 19, 'j': 20, 'k': 21, 'l': 22, 'm': 23, 'n': 24, 'o':
25, 'p': 26, 'q': 27, 'r': 28, 's': 29, 't': 30, 'u': 31, 'v': 32, 'w': 33, 'x':
34, 'y': 35, 'z': 36}
Vocabulary Size:  37
X shape: (102146, 30)
Y shape: (102146, 1)
```

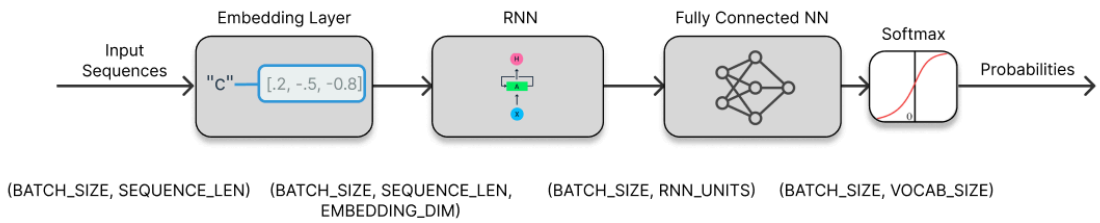
### 1.10.2 4.1 Model Architecture [5% Bonus for All] [P]

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

**4.1.1 Defining the Simple RNN Model** In this part, you need to build a simple recurrent neural network using PyTorch. The architecture of the model is outlined below:




[**EMBEDDING - RNN - ADAPTER - FC**] > **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

> **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
>
> **RNNOutputAdapter**: Helper function implemented to ensure that the pass to the next layer is of correct dimentionality - Input shape (as a tuple): (full_sequence_output of shape (batch_size, sequence_length, rnn_units), - final_hidden_state of shape (1, batch_size, rnn_units)) - Output shape: (batch_size, rnn_units)
>
> **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

You can refer to the following documentation on PyTorch layers for more details: - Embedding - RNN - Dense

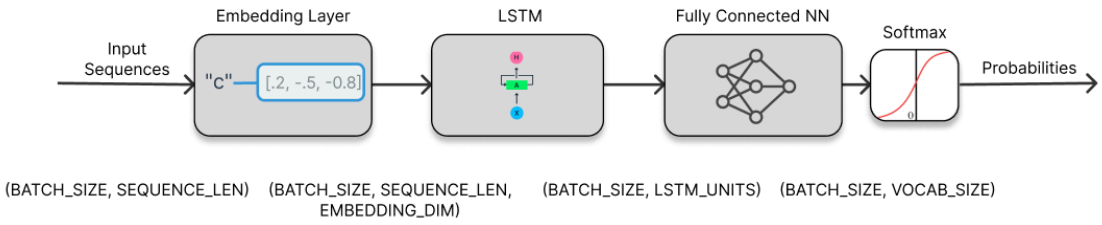
TODO: Implement the define_model function in rnn.py.

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

from rnn import RNN

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

[42]:
```
from utilities.localtests import TestRNN

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

```
test_rnn_architecture passed!
```

**4.1.2 Defining the LSTM Model** 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 - ADAPTER - FC**] > **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
>
> **LSTMOutputAdapter**: Helper function implemented to ensure that the pass to the next layer is of correct dimentionality - Input shape (as a tuple): (full_sequence_output of shape (batch_size, sequence_length, rnn_units), - final_hidden_state of shape (1, batch_size, rnn_units)) - Output shape: (batch_size, rnn_units)
>
> **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

You can refer to the following documentation on PyTorch layers for more details: - Embedding - LSTM - Dense

TODO: Implement the define_model function in lstm.py.

[43]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from lstm import LSTM

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

[44]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from utilities.localtests import TestLSTM

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

```
test_lstm_architecture passed!
```

### 1.10.3 4.2 RNN vs LSTM Model Text Generation Training Comparison Analysis [2.5% Bonus for All] [W]

**Training Configuration**

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

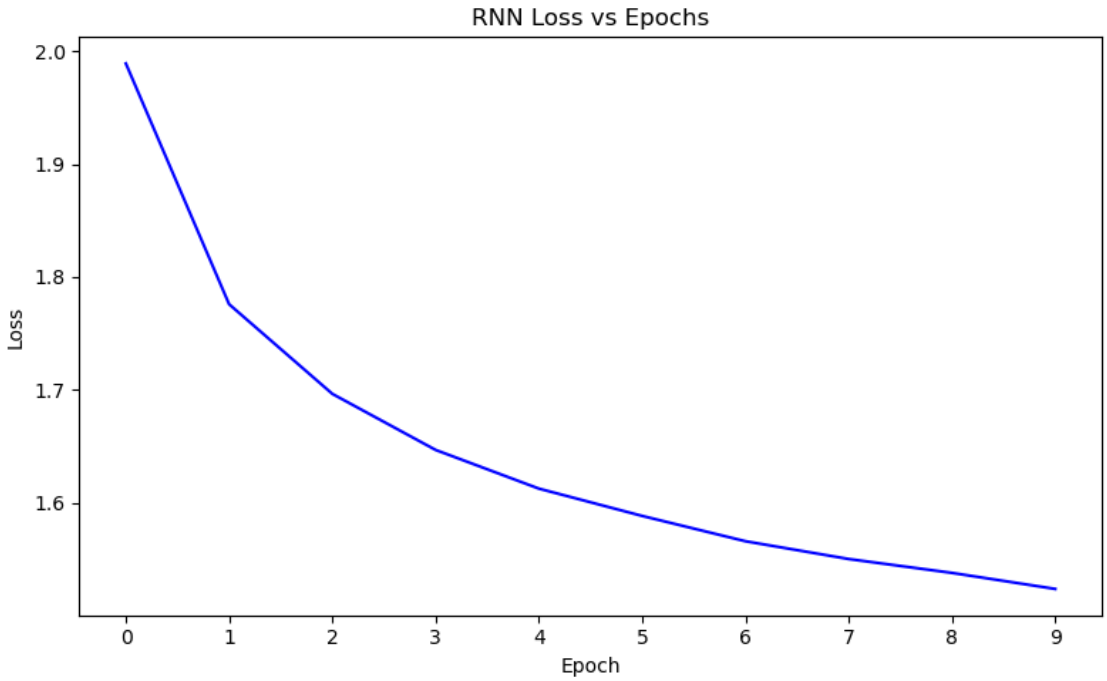
The core training loop implementation can be found in `base_sequential_model.py`. A plot showing loss metrics across epochs will be generated after training completes.

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

[45]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

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

```
Training RNN model from scratch…
Epoch 1/10, Loss: 1.9891
  Saved best model to rnn_model_weights/RNN_weights.pth
Epoch 2/10, Loss: 1.7758
  Saved best model to rnn_model_weights/RNN_weights.pth
Epoch 3/10, Loss: 1.6964
  Saved best model to rnn_model_weights/RNN_weights.pth
Epoch 4/10, Loss: 1.6467
  Saved best model to rnn_model_weights/RNN_weights.pth
Epoch 5/10, Loss: 1.6124
  Saved best model to rnn_model_weights/RNN_weights.pth
Epoch 6/10, Loss: 1.5882
```

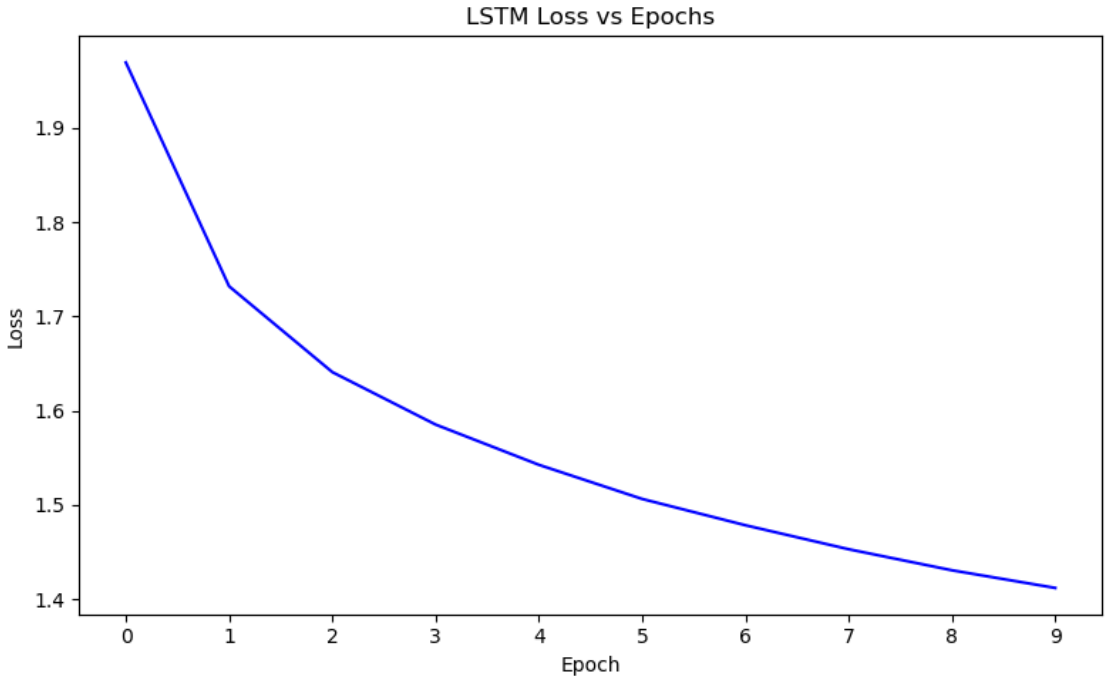
```
  Saved best model to rnn_model_weights/RNN_weights.pth
Epoch 7/10, Loss: 1.5658
  Saved best model to rnn_model_weights/RNN_weights.pth
Epoch 8/10, Loss: 1.5501
  Saved best model to rnn_model_weights/RNN_weights.pth
Epoch 9/10, Loss: 1.5378
  Saved best model to rnn_model_weights/RNN_weights.pth
Epoch 10/10, Loss: 1.5236
  Saved best model to rnn_model_weights/RNN_weights.pth
Finished training. Best model weights saved to rnn_model_weights/RNN_weights.pth
Saved RNN model loss history to rnn_model_weights/RNN_losses.json
```




[46]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

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

```
Training LSTM model from scratch…
Epoch 1/10, Loss: 1.9696
  Saved best model to rnn_model_weights/LSTM_weights.pth
Epoch 2/10, Loss: 1.7319
  Saved best model to rnn_model_weights/LSTM_weights.pth
Epoch 3/10, Loss: 1.6408
```

```
  Saved best model to rnn_model_weights/LSTM_weights.pth
Epoch 4/10, Loss: 1.5851
  Saved best model to rnn_model_weights/LSTM_weights.pth
Epoch 5/10, Loss: 1.5424
  Saved best model to rnn_model_weights/LSTM_weights.pth
Epoch 6/10, Loss: 1.5062
  Saved best model to rnn_model_weights/LSTM_weights.pth
Epoch 7/10, Loss: 1.4783
  Saved best model to rnn_model_weights/LSTM_weights.pth
Epoch 8/10, Loss: 1.4528
  Saved best model to rnn_model_weights/LSTM_weights.pth
Epoch 9/10, Loss: 1.4305
  Saved best model to rnn_model_weights/LSTM_weights.pth
Epoch 10/10, Loss: 1.4117
  Saved best model to rnn_model_weights/LSTM_weights.pth
Finished training. Best model weights saved to
rnn_model_weights/LSTM_weights.pth
Saved LSTM model loss history to rnn_model_weights/LSTM_losses.json
```




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

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

You can refer to text_generator.py for more details.

[47]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from text_generator import TextGenerator

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

[48]:
```
################################
### 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_wrapper=rnn_model, seed_text=seed_text, length=150, temperature=0.75
)
generator.generate(
    model_wrapper=lstm_model, seed_text=seed_text, length=150, temperature=0.75
)
```

```
-------------------- RNN Model --------------------
Prompt: and tartars lips, finger of bi
Model: ve to the concount of clead eal thou desirpt diest to double the father a
for a plore, witch. i ladid come, and your my deare. but birness of an done.
-------------------- LSTM Model --------------------
Prompt: and tartars lips, finger of bi
Model: rd, and that men. macduff. does your suppress with full and father
pocking two the have such an i have be the tworth, well that within. theres to a
mo
```

[48]:
```
'rd, and that men. macduff. does your suppress with full and father pocking two
 the have such an i have be the tworth, well that within. theres to a mo'
```

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

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

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

Some ideas for metrics you can consider:

- Inference time / Training time
- Final Loss Achieved
- Generated Text Quality
- Memory requirements
- Loss convergence
- Simplicity of architecture

**YOUR ANSWER HERE**

---

The LTSM achieved a lower loss than the RNN: 1.4117 compared to 1.5236, respectively. The output of the LTSM also generated coherent text, while the RNN didn't really. Thus, an RNN would be more suitable for short-context, resource-constrained tasks like. In contrast, LSTMs are more suitable for stuff like language modeling or translation that require long-range context.

---

### 1.11 Carbon Impact

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

[49]:
```
emissions = tracker.stop()

car_emissions_per_km = 0.17  # kg CO2e/km driven
# CO2e is a measure of the amount of CO2 that would be equivalent to actual␣
 ↪emissions in terms of warming
# (some gasses have a higher warming effect over 100 years than CO2, e.g., your␣
 ↪car probably emits a bit of SO2, 1g SO2 = 25g CO2e)
students = 900
equivalent_distance_per_student = emissions / car_emissions_per_km
equivalent_distance_all = emissions * students / car_emissions_per_km

print(f"Total emissions in this session: {emissions} kg CO2e")
print(
    f"This session is equivalent to driving {equivalent_distance_per_student:.
 ↪2f}km in an average car"
)

print(
    f"\nDriving Equivalent for all {students} students:␣
 ↪{equivalent_distance_all:.2f} km driven in an average car"
)
```

```
Total emissions in this session: 0.251613481196112 kg CO2e
This session is equivalent to driving 1.48km in an average car

Driving Equivalent for all 900 students: 1332.07 km driven in an average car
```