Spring 2023 CS 4641\7641 A: Machine Learning Homework 4

Instructor: Dr. Mahdi Roozbahani

Deadline: Friday, April 21, 2022 11:59 pm EST

- No unapproved extension of the deadline is allowed. Submission past our 48-hour penalized acceptance period will lead to 0 credit.
- Discussion is encouraged on Ed as part of the Q/A. However, all assignments should be done individually.
- Plagiarism is a **serious offense**. You are responsible for completing your own work. You are not allowed to copy and paste, or paraphrase, or submit materials created or published by others, as if you created the materials. All materials submitted must be your own.
- All incidents of suspected dishonesty, plagiarism, or violations of the Georgia Tech Honor Code will be subject to the institute's Academic Integrity procedures. If we observe any (even small) similarities/plagiarisms detected by Gradescope or our TAs, **WE WILL DIRECTLY REPORT ALL CASES TO OSI**, which may, unfortunately, lead to a very harsh outcome. **Consequences can be severe, e.g., academic probation or dismissal, grade penalties, a 0 grade for assignments concerned, and prohibition from withdrawing from the class.**

Instructions for the assignment

- This assignment consists of both programming and theory questions.
- Unless a theory question explicitly states that no work is required to be shown, you must provide an explanation, justification, or calculation for your answer.
- To switch between cell for code and for markdown, see the menu -> Cell -> Cell Type
- You can directly type Latex equations into markdown cells.
- If a question requires a picture, you could use this syntax `<img src="" style="width: 300px;"/>` to include them within your ipython notebook.
- Your write up must be submitted in PDF form. You may use either Latex, markdown, or any word processing software. We will **NOT** accept handwritten work. Make sure that your work is formatted correctly, for example submit $\sum_{i=0} x_i$ instead of \text{sum_{i=0} x_i}
- When submitting the non-programming part of your assignment, you must correctly map pages of your PDF to each question/subquestion to reflect where they appear. **Improperly mapped questions may not be graded correctly and/or will result in point deductions for the error.**
- All assignments should be done individually, and each student must write up and submit their own answers.
- **Graduate Students**: You are required to complete any sections marked as Bonus for Undergrads

Using the autograder

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

Using the local tests

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

Deliverables and Points Distribution

Q1: Two Layer NN [80 pts; 55pts + 25pts Undergrad Bonus]

Deliverables: NN.py and Notebook Graphs

- **1.1 NN Implementation** [65pts; 50pts + 15pts **Bonus for Undergrad**] - *programming*
  - Leaky_relu [5pts]
  - tanh [5pts]
  - loss [5pts]
  - dropout [5pts]
  - forward propagation and with and without dropout [5pts + 5pts]
  - compute gradients and update weights [2.5pts + 2.5pts]
  - backward without momentum [5pt]
  - Gradient Descent [10pts]
  - Batch Gradient Descent [10pts **Bonus for Undergrad**]
  - Momentum [5pts **Bonus for Undergrad**]
- **1.2 Loss plot and MSE for Gradient Descent** [5pts] - *non-programming*
- **1.3 Loss plot and MSE for Batch Gradient Descent** [5pts **Bonus for Undergrad**] - *non-programming*
- **1.4 Loss plot and MSE value for NN with Gradient Descent with Momentum** [5pts **Bonus for Undergrad**] - *non-programming*

Q2: CNN [25pts; 20pts Bonus for Undergrad + 5pts Bonus for All]

Deliverables: cnn.py and Written Report

- **2.1 Image Classification using Keras CNN** [20pts **Bonus for Undergrad**]
  - 2.1.1 Loading the Model [5pts **Bonus for Undergrad**] - *programming*
  - 2.1.3 Building the Model [5pts **Bonus for Undergrad**] - *non-programming*
  - 2.1.4 Training the Model [8pts **Bonus for Undergrad**] - *non-programming*
  - 2.1.5 Examining Accuracy and Loss [2pts **Bonus for Undergrad**] - *non-programming*

- **2.2 Exploring Deep CNN Architectures** [5pts **Bonus for All**] - *non-programming*

Q3: Random Forest [45pts; 40pts + 5pts Bonus for All]

Deliverables: random_forest.py and Written Report

- 3.1 Random Forest Implementation [35pts] - *programming*
- 3.2 Hyperparameter Tuning with a Random Forest [5pts] - *programming*
- 3.3 Plotting Feature Importance [5pts **Bonus for All**] - *non-programming*

Q4: SVM [30pts Bonus for all]

Deliverables: feature.py and Written Report

- 4.1: Fitting an SVM Classifier by hand [20pts] - *non programming*
- 4.2: Feature Mapping [10pts] - *programming*

Environment Setup

In [ ]:
```
import sys
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_diabetes, fetch_california_housing
from sklearn.preprocessing import MinMaxScaler

from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import mean_squared_error

from collections import Counter
from scipy import stats
from math import log2, sqrt
import pandas as pd
import time
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier

from sklearn.datasets import make_moons
from sklearn.metrics import accuracy_score
from sklearn import svm

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: 3.10.10 | packaged by Anaconda, Inc. | (main, Mar 21 2023, 18:39:17) [MSC v.1916
64 bit (AMD64)]
matplotlib: 3.7.1
numpy: 1.23.5
```

1: Two Layer Neural Network [80 pts; 55pts + 25pts Undergrad Bonus] **[P]****[W]**

1.1 NN Implementation [65pts; 50pts + 15pts Bonus for Undergrad] **[P]**

In this section, you will implement a two layer fully connected neural network. You will also experiment with different activation functions and optimization techniques. We provide two activation functions here - Leaky Relu and Tanh. You will implement a neural network where the first hidden layer has a Leaky Relu activation and the second hidden layer has a Tanh layer.

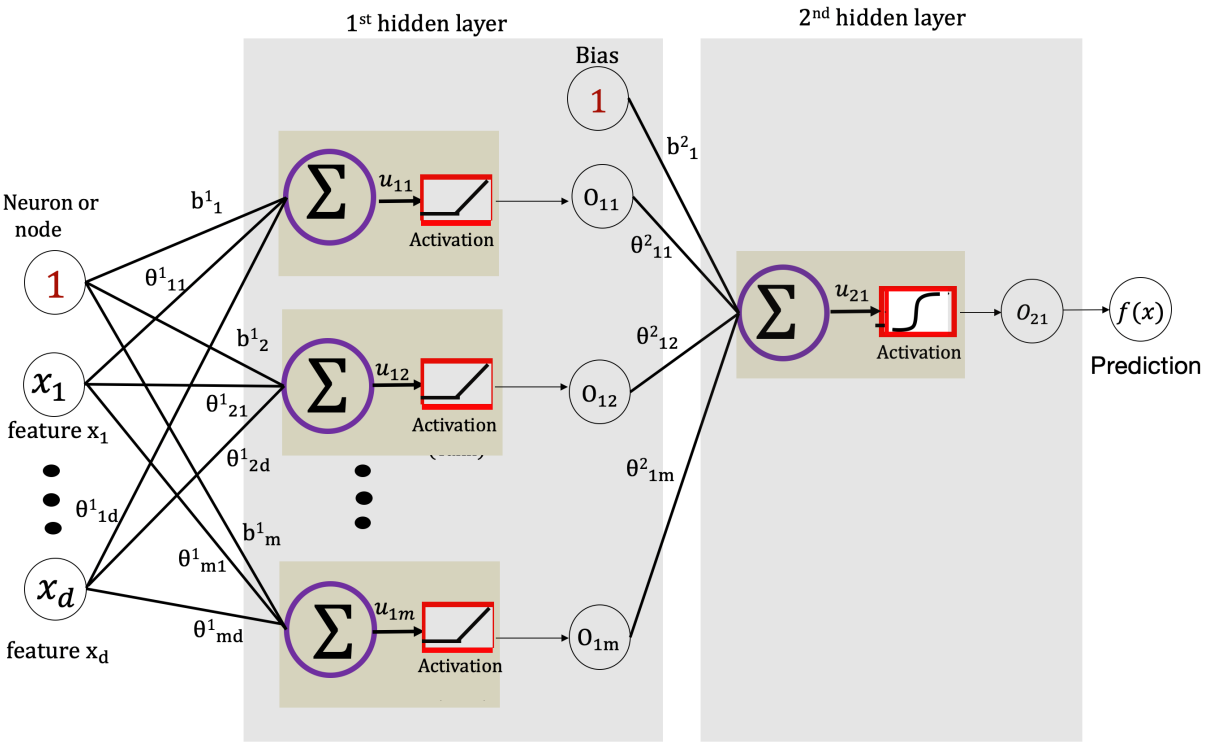
You'll also implement Gradient Descent (GD) and Batch Gradient Descent (BGD) algorithms for training these neural nets. **GD is mandatory for all. BGD is bonus for undergraduate students but mandatory for graduate students.**

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

- **Leaky_Relu**: Note Hint 1
- **Tanh**: Note Hint 1
- **nloss**
- **_dropout**
- **forward**
- **compute_gradients**
- **update_weights**
- **backward**: Note Hint 2, if you still have issues passing the autograder make sure to address Hint 1 as well.
- **gradient_descent**
- **batch_gradient_descent**: **Mandatory for graduate students, bonus for undergraduate students.** Please batch your data in a wraparound manner. For example, given a dataset of 9 numbers, [1, 2, 3, 4, 5, 6, 7, 8, 9], and a batch size of 6, the first iteration batch will be [1, 2, 3, 4, 5, 6], the second iteration batch will be [7, 8, 9, 1, 2, 3], the third iteration batch will be [4, 5, 6, 7, 8, 9], etc...

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

Perceptron



Notation clarification – superscript represents the layer number, subscripts represent the specific units in two adjacent layers being connected by theta
θ^1_{21} - theta of the 1st layer connecting the 2nd hidden unit of the 1st layer and the 1st input unit
θ^2_{12} - theta of the 2nd layer connecting the 1st hidden unit of the 2nd layer and the 2nd hidden unit of the 1st layer
b^1_1 – bias of the 1st hidden unit of the 1st layer

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

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

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

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

Fully connected Layer

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

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

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

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

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

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

Activation Function

There are many activation functions in the literature but for this question we are going to use Leaky Relu and Tanh only.

HINT 1: When calculating the tanh and leaky relu function, make sure you are not modifying the values in the original passed in matrix. You may find np.copy() helpful (`u` should not be modified in the method.) .

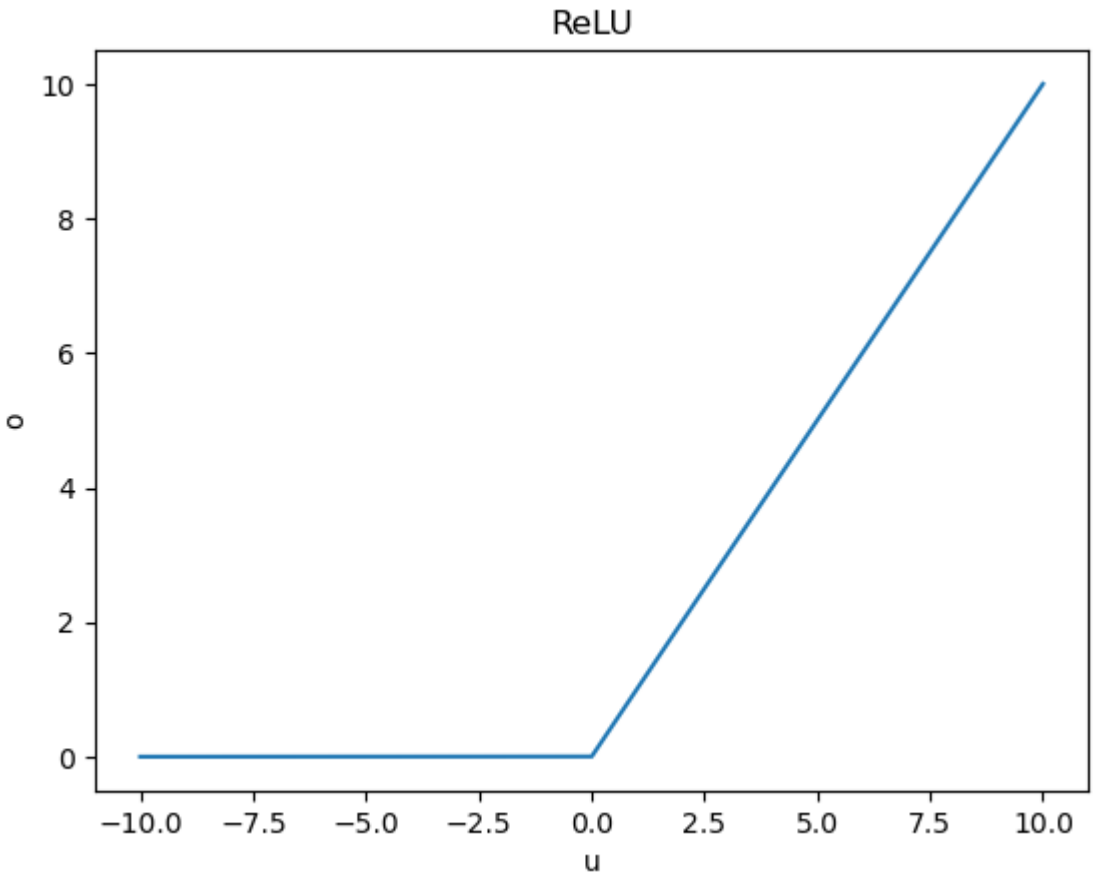
ReLU and Leaky ReLU

The rectified linear unit (ReLU) is one of the most commonly used activation functions in deep learning models. The mathematical form is

$$o = \phi(u) = max(0, u)$$

One of the advantages of Relu is that it is a fast nonlinear activation function.

The derivative of relu function is given as $o' = \phi'(u) = \begin{cases} 0 & u \leq 0 \\ 1 & u > 0 \end{cases}$



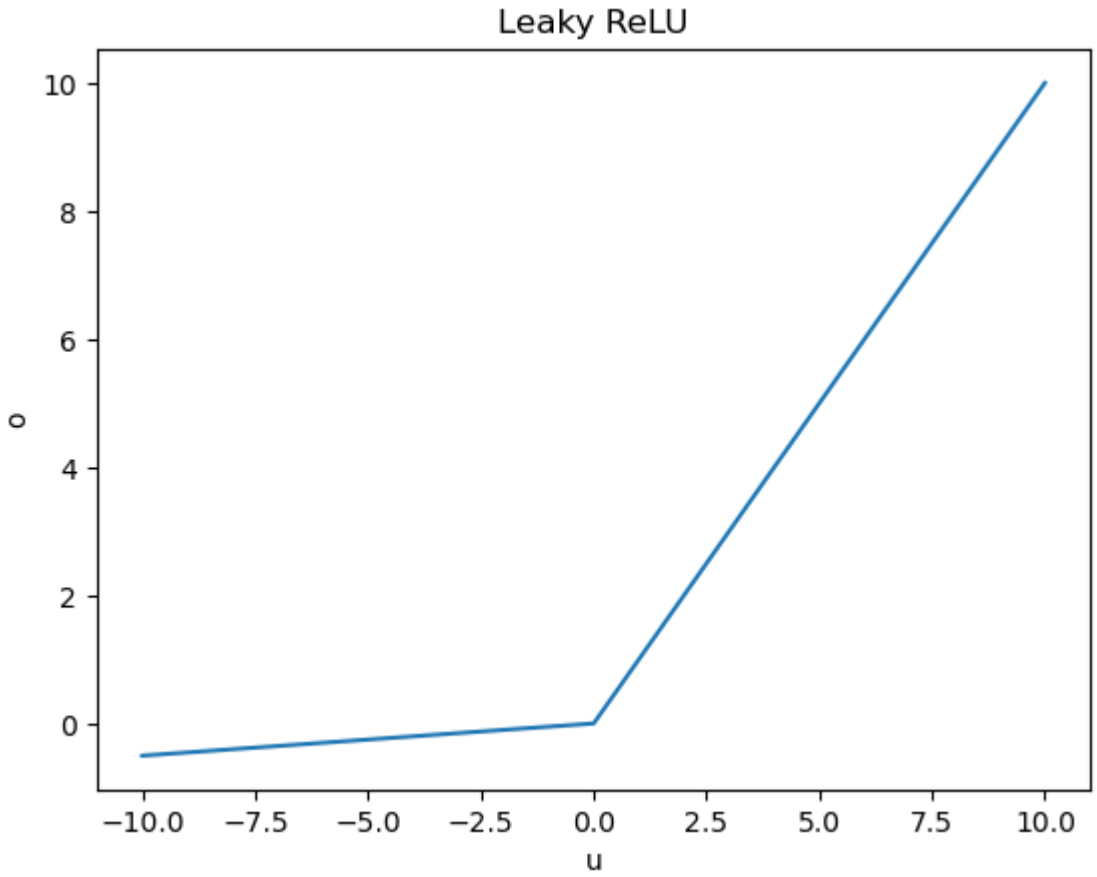
Leaky ReLU is a type of activation function based on a ReLU. The difference is that it has a small slope (such as $\alpha = 0.05$) for negative values instead of a flat slope. The slope coefficient is determined before training. Leaky relu is a popular solution for sparse gradients, for example

training generative adversarial networks.

It takes the form

$$o = \phi(u) = \begin{cases} \alpha u & u \leq 0 \\ u & u > 0 \end{cases}$$

Here in our homework, we are going to implement Leaky ReLU.



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

- **Leaky Relu**: Recall Hint 1

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

TestNN('test_leaky_relu').test_leaky_relu()
TestNN('test_d_leaky_relu').test_d_leaky_relu()
```

```
test_leaky_relu passed!
test_d_leaky_relu passed!
```

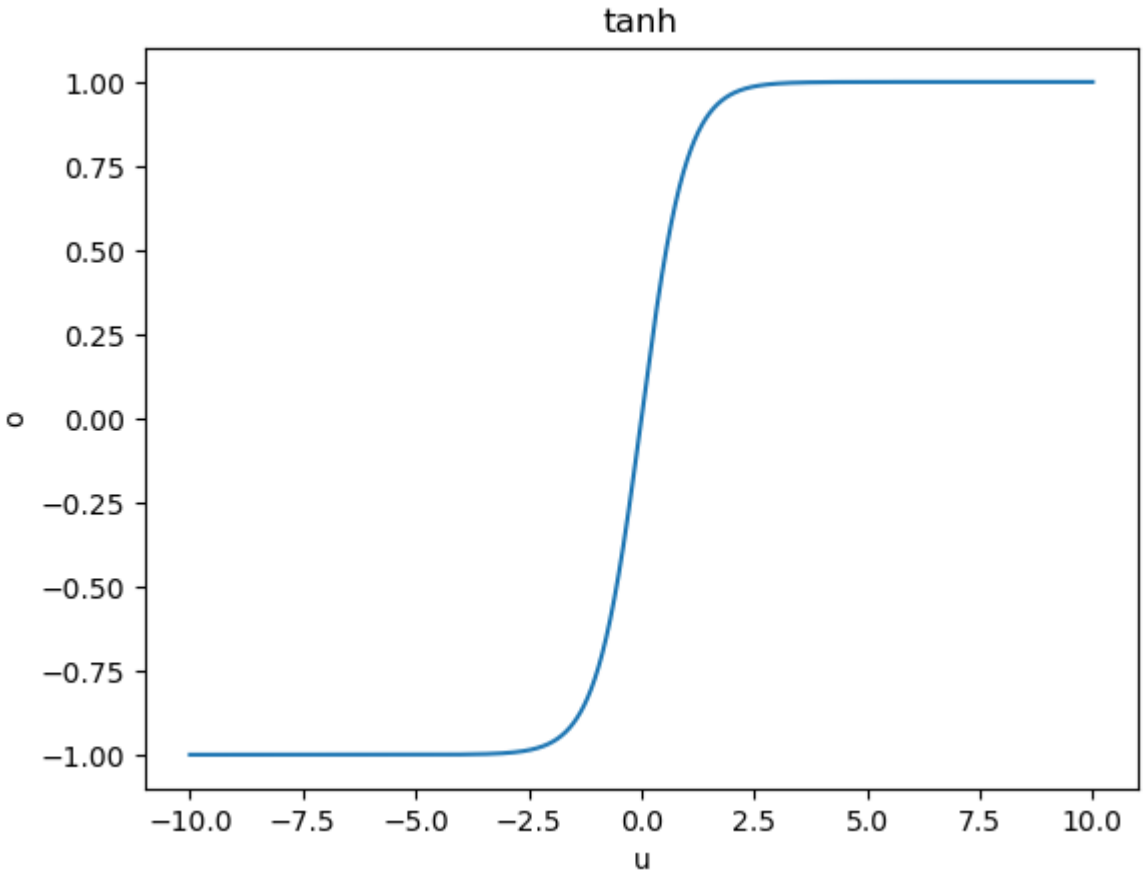
Tanh

Tanh also known as hyperbolic tangent is like a shifted version of sigmoid activation function with its range going from -1 to 1. Tanh almost always proves to be better than the sigmoid function since the mean of the activations are closer to zero. Tanh has an effect of centering data that makes learning for the next layer a bit easier. The mathematical form of tanh is given as

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

The derivative of tanh is given as

$$o' = \phi'(u) = 1 - \left(\frac{e^u - e^{-u}}{e^u + e^{-u}}\right)^2 = 1 - o^2$$



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

- **Tanh**: Recall Hint 1

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

TestNN('test_tanh').test_tanh()
TestNN('test_d_tanh').test_d_tanh()
```

```
test_tanh passed!
test_d_tanh passed!
```

Sigmoid

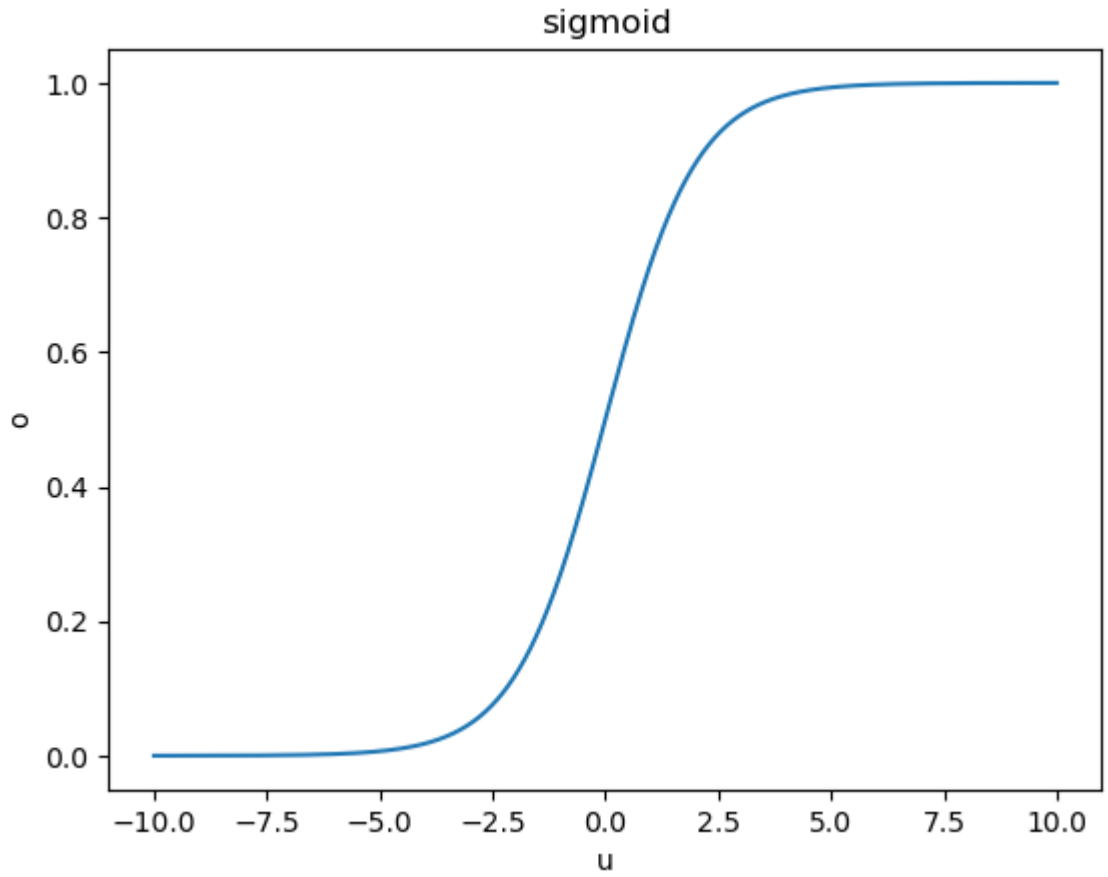
The sigmoid function is another non-linear function with S-shaped curve. This function is useful in the case of binary classification as its output is between 0 and 1. The mathematical form of the function is

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

The derivation of the sigmoid function has a nice form and is given as

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

Note: We will not be using sigmoid activation function for this assignment. This is included only for the sake of completeness.



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 drop (i.e., sets to zero) the input unit. The random decisions are made independently for each unit. Formally, given N x K input units (N is the number of data and K is the width of the layer), it samples N x K i.i.d. from Bernoulli(p). This forces the network to learn more robust and generalizable features, since it cannot rely too much on any one input unit. During inference, the dropout layer is turned off, and the full network is used to make predictions. Usually, p is passed to the neural network as a hyperparameter. Setting $p = 0$ is equivalent to no dropout.

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

Use this as a reference for your implementation.

Hint: Use np.random.choice() to determine which nodes to drop. Use this function to generate a mask consisting 0s (drop) and 1s (keep) to apply to the input. Apply the mask, and then scale the result by a factor of 1/(1-p). Why do we do this? The dropout mask may also be helpful for the backward method.

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

- **_dropout**: Use the hint above.

In [ ]:
```
from utilities.localtests import TestNN
TestNN('test_dropout').test_dropout()
```

```
test_dropout passed!
```

Mean Squared Error

Mean squared error (MSE) is an estimator that measures the average of the squares of the errors i.e. the average squared difference between the actual and the estimated values. MSE estimates the quality of the learnt hypothesis between the actual and the predicted values. Note that MSE is non-negative. If it is closer to zero, the better the learnt function is.

Implementation details

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

$$MSE = \frac{1}{2N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where y_i is the true label and $\hat{y}_i$ is the estimated label. We use a factor of $\frac{1}{2N}$ instead of $\frac{1}{N}$ to simply the derivative of loss function.

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

- **nloss**

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

TestNN('test_loss').test_loss()
```

```
test_loss passed!
```

Initialization

We start by initializing the weights of the fully connected layer using Xavier initialization Xavier initialization (At a high level, we are using a uniform distribution for weight initialization). This is already implemented for you.

Forward Propagation

During training, we pass all the data points through the network layer by layer using forward propagation. The main equations for forward prop have been described below.

$$
\begin{aligned}
u^{[0]} &= x \\
u^{[1]} &= \theta^{[1]} u^{[0]} + b^{[1]} \\
o^{[1]} &= \text{Dropout}(\text{LeakyRelu}(u^{[1]})) \\
u^{[2]} &= \theta^{[2]} o^{[1]} + b^{[2]} \\
\hat{y} = o^{[2]} &= \text{Tanh}(u^{[2]})
\end{aligned}
$$

Then we get the output and compute the loss

$$l = \frac{1}{2N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

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

- **forward**: Be sure to check the value of the *use_dropout* parameter and calculate the output accordingly.

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

TestNN('test_forward').test_forward()
TestNN('test_forward_without_dropout').test_forward_without_dropout()
```

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

Backward Propagation: Update Weights and Compute Gradients

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

Update Weights

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

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

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

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

- **update_weights**: the following local test will test when *use_momentum* is False

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

TestNN('test_update_weights').test_update_weights()
```

```
test_update_weights passed!
```

Update Weights with Momentum [Bonus for Undergrad]

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

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

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

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

Momentum proposes the following tweak to our parameter update function:

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

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

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

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

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

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

- **update_weights**: Make to sure to update the weights using momentum

HINT: z is stored in self.change

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

TestNN('test_update_weights_with_momentum').test_update_weights_with_momentum()
```

```
test_update_weights_with_momentum passed!
```

Compute Gradients

To compute the terms $\frac{\partial l}{\partial \theta^{[i]}}$ and $\frac{\partial l}{\partial b^{[i]}}$ we use chain rule for differentiation as follows:

$$\begin{aligned}\frac{\partial l}{\partial \theta^{[2]}} &= \frac{\partial l}{\partial o^{[2]}}\frac{\partial o^{[2]}}{\partial u^{[2]}}\frac{\partial u^{[2]}}{\partial \theta^{[2]}} \\ \frac{\partial l}{\partial b^{[2]}} &= \frac{\partial l}{\partial o^{[2]}}\frac{\partial o^{[2]}}{\partial u^{[2]}}\frac{\partial u^{[2]}}{\partial b^{[2]}}\end{aligned}$$

So, $\frac{\partial l}{\partial o^{[2]}}$ is the differentiation of the loss function at point $o^{[2]}$

$\frac{\partial o^{[2]}}{\partial u^{[2]}}$ is the differentiation of the Tanh function at point $u^{[2]}$

$\frac{\partial u^{[2]}}{\partial \theta^{[2]}}$ is equal to $o^{[1]}$

$\frac{\partial u^{[2]}}{\partial b^{[2]}}$ is equal to 1.

To compute $\frac{\partial l}{\partial \theta^{[2]}}$, we need $o^{[2]}, u^{[2]} \& o^{[1]}$ which are calculated during forward propagation. So we need to store these values in cache variables during forward propagation to be able to access them during backward propagation. Similarly for calculating other partial derivatives, we store the values we'll be needing for chain rule in cache. These values are obtained from the forward propagation and used in backward propagation. The cache is implemented as a dictionary here where the keys are the variable names and the values are the variables values.

Also, the functional form of the MSE differentiation and Leaky Relu differentiation are given by

$$\begin{aligned}\frac{\partial l}{\partial o^{[2]}} &= \left(o^{[2]} - y\right) \\ \frac{\partial l}{\partial u^{[2]}} &= \frac{\partial l}{\partial o^{[2]}} * (1 - (tanh(u^{[2]}))^2) \\ \frac{\partial u^{[2]}}{\partial \theta^{[2]}} &= o^{[1]} \\ \frac{\partial u^{[2]}}{\partial b^{[2]}} &= 1\end{aligned}$$

On vectorization, the above equations become:

$$\begin{aligned}\frac{\partial l}{\partial \theta^{[2]}} &= \frac{1}{n}\frac{\partial l}{\partial u^{[2]}}o^{[1]} \\ \frac{\partial l}{\partial b^{[2]}} &= \frac{1}{n}\sum\frac{\partial l}{\partial u^{[2]}}\end{aligned}$$

!!!!! IMPORTANT !!!!! HINT 2: Division by N only needs to occur ONCE for any derivative that requires a division by N. Make sure you avoid cascading divisions by N where you might accidentally divide your derivative by N^2 or greater.

This completes the differentiation of loss function w.r.t to parameters in the second layer. We now move on to the first layer, the equations for which are given as follows:

$$\frac{\partial l}{\partial \theta^{[1]}} = \frac{\partial l}{\partial o^{[2]}}\frac{\partial o^{[2]}}{\partial u^{[2]}}\frac{\partial u^{[2]}}{\partial o^{[1]}}\frac{\partial o^{[1]}}{\partial u^{[1]}}\frac{\partial u^{[1]}}{\partial \theta^{[1]}}$$
$$\frac{\partial l}{\partial b^{[1]}} = \frac{\partial l}{\partial o^{[2]}}\frac{\partial o^{[2]}}{\partial u^{[2]}}\frac{\partial u^{[2]}}{\partial o^{[1]}}\frac{\partial o^{[1]}}{\partial u^{[1]}}\frac{\partial u^{[1]}}{\partial b^{[1]}}$$

Where

$$\frac{\partial u^{[2]}}{\partial o^{[1]}} = \theta^{[2]}$$
$$\frac{\partial o^{[1]}}{\partial u^{[1]}} = \begin{cases} \text{DropoutMask} \cdot \text{ScalingFactor} \cdot [1 * (u^{[1]} > 0) \text{ and } \alpha * (u^{[1]} <= 0)] & \text{dropout=true} \\ 1 * (u^{[1]} > 0) \text{ and } \alpha * (u^{[1]} <= 0) & \text{dropout=false} \end{cases}$$
$$\frac{\partial u^{[1]}}{\partial \theta^{[1]}} = x$$
$$\frac{\partial u^{[1]}}{\partial b^{[1]}} = 1$$

Note that $\frac{\partial o^{[1]}}{\partial u^{[1]}}$ is the derivative of the compostion of leaky ReLU and dropout at $u^{[1]}$. The dropout mask you stored at forward may be helpful.

The above equations outline the forward and backward propagation process for a 2-layer fully connected neural net with leaky Relu as the first activation layer and Tanh has the second one. The same process can be extended to different neural networks with different activation layers.

Code Implementation:

$$dLoss\_o2 = \frac{\partial l}{\partial o^{[2]}} \implies dim = (1, 375)$$
$$dLoss\_u2 = dLoss\_o2\frac{\partial o^{[2]}}{\partial u^{[2]}} \implies dim = (1, 375)$$
$$dLoss\_theta2 = dLoss\_u2\frac{\partial u^{[2]}}{\partial \theta^{[2]}} \implies dim = (1, 15)$$
$$dLoss\_b2 = dLoss\_u2\frac{\partial u^{[2]}}{\partial b^{[2]}} \implies dim = (1, 1)$$
$$dLoss\_o1 = dLoss\_u2\frac{\partial u^{[2]}}{\partial o^{[1]}} \implies dim = (15, 375)$$
$$dLoss\_u1 = dLoss\_o1\frac{\partial o^{[1]}}{\partial u^{[1]}} \implies dim = (15, 375)$$
$$dLoss\_theta1 = dLoss\_u1\frac{\partial u^{[1]}}{\partial \theta^{[1]}} \implies dim = (15, 8)$$
$$dLoss\_b1 = dLoss\_u1\frac{\partial u^{[1]}}{\partial b^{[1]}} \implies dim = (15, 1)$$

$\partial O^{[2]}$

One common confusion is when to use matrix multiplication versus element-wise multiplication. A good rule of thumb here is that whenever you use matrix multiplication in forward propagation, you'll likely have to use it for the backward step as well. The easiest way to make sure you're on the right track is to check the shapes of the arrays you're working with.

Note: Training set has 375 examples.

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

- **compute_gradients**: Be sure to account for the different values of *use_dropout* and calculate accordingly.

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

TestNN('test_compute_gradients_without_dropout').test_compute_gradients_without_dropout()
TestNN('test_compute_gradients').test_compute_gradients()
```

```
test_compute_gradients_withou_dropout passed!
test_compute_gradients passed!
```

1.1.1 Local Test: Helper Functions and Compute Gradient

You may test your implementation of the hepler functions and compute_gradient contained in **NN.py** in the cell below. See Using the Local Tests for more details.

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

TestNN('test_leaky_relu').test_leaky_relu()
TestNN('test_d_leaky_relu').test_d_leaky_relu()
TestNN('test_tanh').test_tanh()
TestNN('test_d_tanh').test_d_tanh()
TestNN('test_dropout').test_dropout()
TestNN('test_loss').test_loss()
TestNN('test_forward').test_forward()
TestNN('test_forward_without_dropout').test_forward_without_dropout()
TestNN('test_update_weights').test_update_weights()
TestNN('test_update_weights_with_momentum').test_update_weights_with_momentum()
TestNN('test_compute_gradients_without_dropout').test_compute_gradients_without_dropout()
TestNN('test_compute_gradients').test_compute_gradients()
```

```
test_leaky_relu passed!
test_d_leaky_relu passed!
test_tanh passed!
test_d_tanh passed!
test_dropout passed!
test_loss passed!
test_forward passed!
test_forward_without_dropout passed!
test_update_weights passed!
test_update_weights_with_momentum passed!
test_compute_gradients_withou_dropout passed!
test_compute_gradients passed!
```

1.1.2 Local Test: Gradient Descent

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

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

from utilities.localtests import TestNN
TestNN('test_compute_gradients').test_gradient_descent()
```

```
Loss after iteration 0: 15.794846
Loss after iteration 1: 14.874647
Loss after iteration 2: 14.128740

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

1.1.3 Local Test: Batch Gradient Descent [No Points]

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

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

from utilities.localtests import TestNN
TestNN('test_batch_gradient_descent').test_batch_gradient_descent()
```

```
Loss after iteration 0: 7.642902
Loss after iteration 1: 16.555975
Loss after iteration 2: 20.684735

y_train input: [[1. 2. 3. 4. 5. 6. 7. 8. 9.]]
batch_y at iteration 0:  [[1. 2. 3. 4. 5. 6.]]
batch_y at iteration 1:  [[7. 8. 9. 1. 2. 3.]]
batch_y at iteration 2:  [[4. 5. 6. 7. 8. 9.]]

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

1.1.4 Local Test: Gradient Descent with Momentum

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

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

from utilities.localtests import TestNN
TestNN("test_gradient_descent_with_momentum").test_gradient_descent_with_momentum()
```

```
Loss after iteration 0: 15.794846
Loss after iteration 1: 14.874647
Loss after iteration 2: 13.762403

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

1.2 Loss plot and MSE value for NN with Gradient Descent [5pts] **[W]**

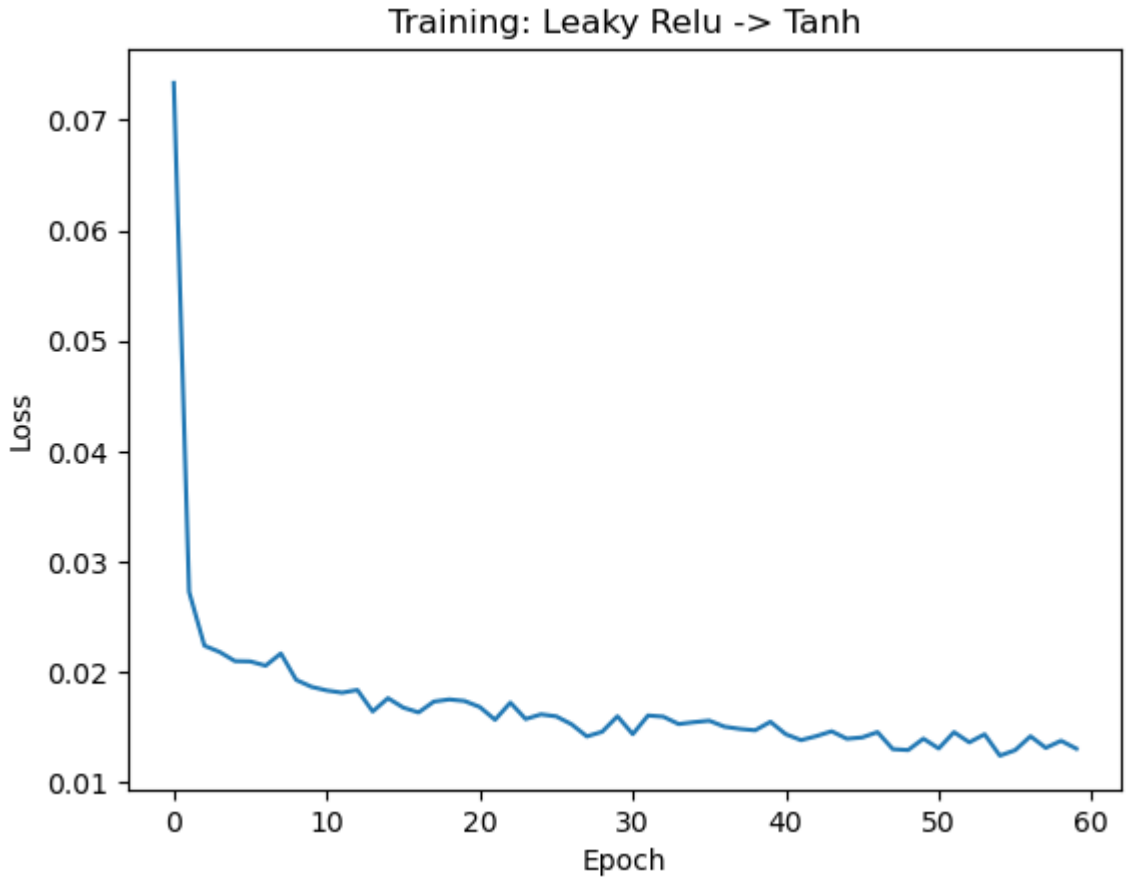
Train your neural net implementation with gradient descent and print out the loss at every 1000th iteration (starting at iteration 0). The following cells will plot the loss vs epoch graph and calculate the final test MSE.

In [ ]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################

from NN import dlnet

dataset = fetch_california_housing() # load the dataset
x, y = dataset.data, dataset.target
y = y.reshape(-1,1)
perm = np.random.RandomState(seed=3).permutation(x.shape[0])[:500]
x = x[perm]
y = y[perm]



x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=1) #split data


x_scale = MinMaxScaler()
x_train = x_scale.fit_transform(x_train) #normalize data/
x_test = x_scale.transform(x_test)

y_scale = MinMaxScaler()
y_train = y_scale.fit_transform(y_train)
y_test = y_scale.transform(y_test)

x_train, x_test, y_train, y_test = x_train.T, x_test.T, y_train.reshape(1,-1), y_test #cor

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

# create figure
fig = plt.plot(np.array(nn.loss).squeeze())
plt.title(f'Training: {nn.neural_net_type}')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```

```
Loss after iteration 0: 0.073367
Loss after iteration 1000: 0.027270
Loss after iteration 2000: 0.022392
Loss after iteration 3000: 0.021815
Loss after iteration 4000: 0.020991
Loss after iteration 5000: 0.020962
Loss after iteration 6000: 0.020578
Loss after iteration 7000: 0.021684
Loss after iteration 8000: 0.019288
Loss after iteration 9000: 0.018663
Loss after iteration 10000: 0.018332
Loss after iteration 11000: 0.018132
Loss after iteration 12000: 0.018375
Loss after iteration 13000: 0.016428
Loss after iteration 14000: 0.017634
Loss after iteration 15000: 0.016780
Loss after iteration 16000: 0.016345
Loss after iteration 17000: 0.017325
Loss after iteration 18000: 0.017520
Loss after iteration 19000: 0.017381
Loss after iteration 20000: 0.016836
Loss after iteration 21000: 0.015676
Loss after iteration 22000: 0.017232
Loss after iteration 23000: 0.015746
Loss after iteration 24000: 0.016166
Loss after iteration 25000: 0.015988
Loss after iteration 26000: 0.015268
Loss after iteration 27000: 0.014172
Loss after iteration 28000: 0.014592
Loss after iteration 29000: 0.015996
Loss after iteration 30000: 0.014374
Loss after iteration 31000: 0.016066
Loss after iteration 32000: 0.015958
Loss after iteration 33000: 0.015288
Loss after iteration 34000: 0.015461
Loss after iteration 35000: 0.015595
Loss after iteration 36000: 0.015049
Loss after iteration 37000: 0.014851
Loss after iteration 38000: 0.014726
Loss after iteration 39000: 0.015492
Loss after iteration 40000: 0.014372
Loss after iteration 41000: 0.013835
Loss after iteration 42000: 0.014194
Loss after iteration 43000: 0.014633
Loss after iteration 44000: 0.013956
Loss after iteration 45000: 0.014078
Loss after iteration 46000: 0.014560
Loss after iteration 47000: 0.013015
Loss after iteration 48000: 0.012933
Loss after iteration 49000: 0.013964
Loss after iteration 50000: 0.013078
Loss after iteration 51000: 0.014556
Loss after iteration 52000: 0.013634
Loss after iteration 53000: 0.014363
Loss after iteration 54000: 0.012426
Loss after iteration 55000: 0.012926
Loss after iteration 56000: 0.014185
Loss after iteration 57000: 0.013132
Loss after iteration 58000: 0.013775
```

```
Loss after iteration 59000: 0.013058
```



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

y_predicted = nn.predict(x_test) # predict
y_test = y_test.reshape(1,-1)
print("Mean Squared Error (MSE)", mean_squared_error(y_test, y_predicted))
```

```
Mean Squared Error (MSE) 0.018511096794416558
```

1.3 Loss plot and MSE value for NN with BGD [5pts Bonus for Undergrad] **[W]**

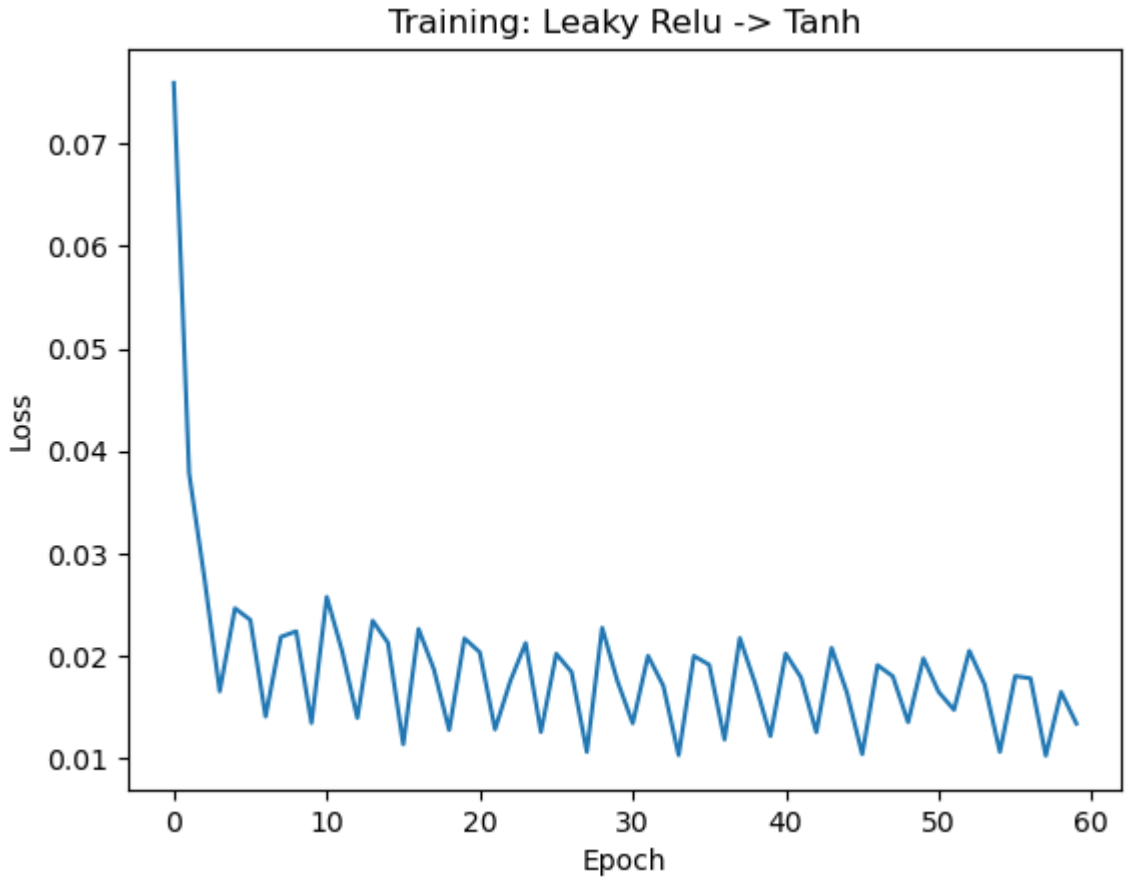
Train your neural net implementation with batch gradient descent and print out the loss at every 1000th iteration (starting at iteration 0). The following cells will plot the loss vs epoch graph and calculate the final test MSE.

In [ ]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################

from NN import dlnet

dataset = fetch_california_housing() # load the dataset
x, y = dataset.data, dataset.target
y = y.reshape(-1,1)
perm = np.random.RandomState(seed=3).permutation(x.shape[0])[:500]
x = x[perm]
y = y[perm]

x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=1) #split data

x_scale = MinMaxScaler()
x_train = x_scale.fit_transform(x_train) #normalize data
x_test = x_scale.transform(x_test)

y_scale = MinMaxScaler()
y_train = y_scale.fit_transform(y_train)
y_test = y_scale.transform(y_test)

x_train, x_test, y_train, y_test = x_train.T, x_test.T, y_train.reshape(1,-1), y_test #cor

nn = dlnet(y_train,lr=0.01,use_dropout=True, use_momentum=False) # initalize neural net cl
nn.batch_gradient_descent(x_train, y_train, iter = 60000, use_momentum=False) #train


# create figure
fig = plt.plot(np.array(nn.loss).squeeze())
plt.title(f'Training: {nn.neural_net_type}')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```

```
Loss after iteration 0: 0.075947
Loss after iteration 1000: 0.037840
Loss after iteration 2000: 0.027605
Loss after iteration 3000: 0.016528
Loss after iteration 4000: 0.024636
Loss after iteration 5000: 0.023500
Loss after iteration 6000: 0.014060
Loss after iteration 7000: 0.021838
Loss after iteration 8000: 0.022388
Loss after iteration 9000: 0.013432
Loss after iteration 10000: 0.025752
Loss after iteration 11000: 0.020464
Loss after iteration 12000: 0.013923
Loss after iteration 13000: 0.023421
Loss after iteration 14000: 0.021260
Loss after iteration 15000: 0.011353
Loss after iteration 16000: 0.022616
Loss after iteration 17000: 0.018678
Loss after iteration 18000: 0.012742
Loss after iteration 19000: 0.021689
Loss after iteration 20000: 0.020347
Loss after iteration 21000: 0.012803
Loss after iteration 22000: 0.017539
Loss after iteration 23000: 0.021243
Loss after iteration 24000: 0.012547
Loss after iteration 25000: 0.020185
Loss after iteration 26000: 0.018413
Loss after iteration 27000: 0.010610
Loss after iteration 28000: 0.022724
Loss after iteration 29000: 0.017517
Loss after iteration 30000: 0.013402
Loss after iteration 31000: 0.019991
Loss after iteration 32000: 0.017062
Loss after iteration 33000: 0.010287
Loss after iteration 34000: 0.019977
Loss after iteration 35000: 0.019117
Loss after iteration 36000: 0.011813
Loss after iteration 37000: 0.021730
Loss after iteration 38000: 0.017288
Loss after iteration 39000: 0.012154
Loss after iteration 40000: 0.020205
Loss after iteration 41000: 0.017872
Loss after iteration 42000: 0.012524
Loss after iteration 43000: 0.020768
Loss after iteration 44000: 0.016404
Loss after iteration 45000: 0.010378
Loss after iteration 46000: 0.019081
Loss after iteration 47000: 0.017990
Loss after iteration 48000: 0.013506
Loss after iteration 49000: 0.019744
Loss after iteration 50000: 0.016456
Loss after iteration 51000: 0.014715
Loss after iteration 52000: 0.020484
Loss after iteration 53000: 0.017198
Loss after iteration 54000: 0.010605
Loss after iteration 55000: 0.018005
Loss after iteration 56000: 0.017798
Loss after iteration 57000: 0.010237
Loss after iteration 58000: 0.016456
```

```
Loss after iteration 59000: 0.013375
```



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

y_predicted = nn.predict(x_test) # predict
y_test = y_test.reshape(1,-1)
print("Mean Squared Error (MSE)", mean_squared_error(y_test, y_predicted))
```

```
Mean Squared Error (MSE) 0.01851442644000594
```

1.4 Loss plot and MSE value for NN with Gradient Descent with Momentum [5pts Bonus for Undergrad] **[W]**

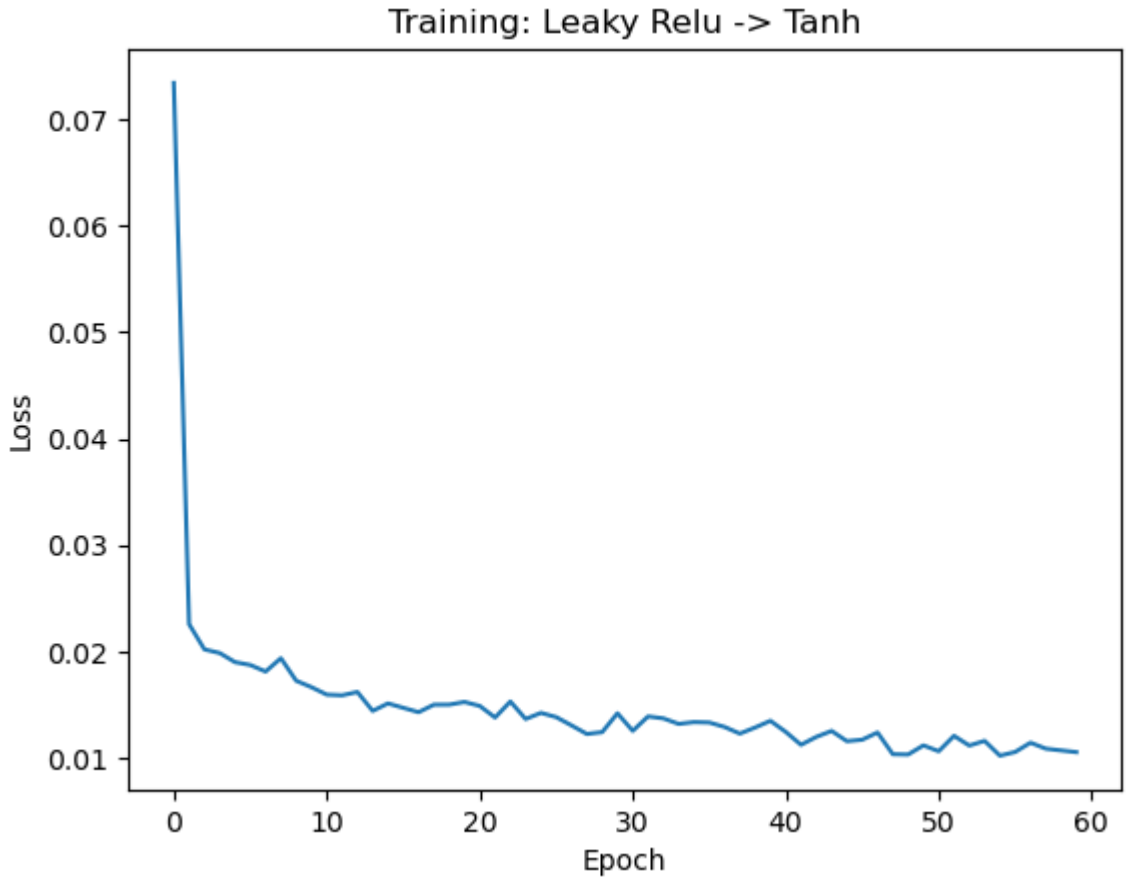
Train your neural net implementation with gradient descent with momentum and print out the loss at every 1000th iteration (starting at iteration 0). The following cells will plot the loss vs epoch graph and calculate the final test MSE.

In [ ]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################

from NN import dlnet

dataset = fetch_california_housing() # load the dataset
x, y = dataset.data, dataset.target
y = y.reshape(-1,1)
perm = np.random.RandomState(seed=3).permutation(x.shape[0])[:500]
x = x[perm]
y = y[perm]



x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=1) #split data


x_scale = MinMaxScaler()
x_train = x_scale.fit_transform(x_train) #normalize data
x_test = x_scale.transform(x_test)

y_scale = MinMaxScaler()
y_train = y_scale.fit_transform(y_train)
y_test = y_scale.transform(y_test)

x_train, x_test, y_train, y_test = x_train.T, x_test.T, y_train.reshape(1,-1), y_test #cor

nn = dlnet(y_train,lr=0.01,use_dropout=True, use_momentum=True) # initalize neural net cla
nn.gradient_descent(x_train, y_train, iter = 60000, use_momentum=True) #train

# create figure
fig = plt.plot(np.array(nn.loss).squeeze())
plt.title(f'Training: {nn.neural_net_type}')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```

```
Loss after iteration 0: 0.073367
Loss after iteration 1000: 0.022590
Loss after iteration 2000: 0.020221
Loss after iteration 3000: 0.019878
Loss after iteration 4000: 0.019025
Loss after iteration 5000: 0.018756
Loss after iteration 6000: 0.018141
Loss after iteration 7000: 0.019393
Loss after iteration 8000: 0.017279
Loss after iteration 9000: 0.016674
Loss after iteration 10000: 0.015974
Loss after iteration 11000: 0.015911
Loss after iteration 12000: 0.016224
Loss after iteration 13000: 0.014447
Loss after iteration 14000: 0.015151
Loss after iteration 15000: 0.014737
Loss after iteration 16000: 0.014328
Loss after iteration 17000: 0.015032
Loss after iteration 18000: 0.015036
Loss after iteration 19000: 0.015289
Loss after iteration 20000: 0.014914
Loss after iteration 21000: 0.013829
Loss after iteration 22000: 0.015331
Loss after iteration 23000: 0.013691
Loss after iteration 24000: 0.014263
Loss after iteration 25000: 0.013865
Loss after iteration 26000: 0.013085
Loss after iteration 27000: 0.012286
Loss after iteration 28000: 0.012457
Loss after iteration 29000: 0.014226
Loss after iteration 30000: 0.012576
Loss after iteration 31000: 0.013932
Loss after iteration 32000: 0.013748
Loss after iteration 33000: 0.013229
Loss after iteration 34000: 0.013398
Loss after iteration 35000: 0.013362
Loss after iteration 36000: 0.012944
Loss after iteration 37000: 0.012318
Loss after iteration 38000: 0.012892
Loss after iteration 39000: 0.013519
Loss after iteration 40000: 0.012480
Loss after iteration 41000: 0.011268
Loss after iteration 42000: 0.012011
Loss after iteration 43000: 0.012570
Loss after iteration 44000: 0.011607
Loss after iteration 45000: 0.011746
Loss after iteration 46000: 0.012421
Loss after iteration 47000: 0.010386
Loss after iteration 48000: 0.010357
Loss after iteration 49000: 0.011214
Loss after iteration 50000: 0.010664
Loss after iteration 51000: 0.012109
Loss after iteration 52000: 0.011196
Loss after iteration 53000: 0.011636
Loss after iteration 54000: 0.010249
Loss after iteration 55000: 0.010597
Loss after iteration 56000: 0.011467
Loss after iteration 57000: 0.010906
Loss after iteration 58000: 0.010742
```

```
Loss after iteration 59000: 0.010581
```



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

y_predicted = nn.predict(x_test) # predict
y_test = y_test.reshape(1,-1)
print("Mean Squared Error (MSE)", mean_squared_error(y_test, y_predicted))
```

```
Mean Squared Error (MSE) 0.01822903192336678
```

2: Image Classification based on Convolutional Neural Networks [25pts; 20pts Bonus for Undergrad + 5pts Bonus for all] **[P]****[W]**

2.1 Image Classification using Keras and CNN

- TransorFlow is a popular platform for machine learning.
- Keras is a deep learning API written in Python that runs on top of the machine learning platform TensorFlow.

TensorFlow and Keras

Tensorflow is a machine learning package developed by Google. In 2019, Google integrated Keras into Tensorflow. It creates a simple, layer-centric interface to Tensorflow.

Helpful Links

- Install Keras
- CS231n Tutorial(Layers used to build ConvNets)

Setup TensorFlow

You can see here to install TensorFlow. Make sure you have upgraded to the latest `pip` to install the TensorFlow 2 package. Alternatively, you can train your model by uploading HW4 notebook to Colab directly. Colab contains all packages you need for this section.

Please also see TensorFlow 2 quickstart for beginners to see how to load a data set, build a `tf.keras.Sequential` model, and train and evaluate the model.

For Mac M1 users, please see this EdStem Post

Environment Setup

In [ ]:
```
from __future__ import print_function
import tensorflow as tf
from keras.datasets import cifar10
from sklearn.model_selection import train_test_split
# # from tensorflow.keras.models import Sequential
# # from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Activation,
# # from tensorflow.keras.layers import LeakyReLU
# # from tensorflow.keras.layers import BatchNormalization

# VS Code was not letting me import the way it was originally implemented
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Activation, Dropout
from keras.layers import LeakyReLU
from keras.layers import BatchNormalization

from sklearn.utils import shuffle
import numpy as np
import matplotlib.pyplot as plt
```

2.1.1 Load CIFAR-10 Dataset and Data Augmentation [5pts - Bonus for Undergrad]**[P]**

We use CIFAR-10 dataset to train our model. This is a dataset of 60,000 32x32 colour images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images. We provide code for you to download CIFAR-10 dataset below.

Data Augmentation [5pts]

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

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

- **data_preprocessing**
- **data_augmentation**
    - HORIZONTAL AND VERTICAL FLIP: We flip the images horizontally or vertically based on the mode attribute which can be horizontal, vertical or both.
    - RANDOM ROTATION: We apply random rotations to each image, filling empty space according to fill_mode. The fill mode can be nearest (using nearest pixels to determine the fill value), reflect, constant or wrap.
    - RANDOM ZOOM: We apply zooming on images by a factor which represents lower and upper bound for zooming. It could be applied along either height or width or both, which positive factor values mean zooming out, and negetive values mean zooming in.

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

You can go through the Tensorflow documentation to see the different image processing techniques we can use for our computer vision projects.

References

tf.data Module

tf.data.Dataset

tf.data: Build TensorFlow input pipelines

Better performance with the tf.data API

Random Rotation

Random Flip

Random Zoom

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
from image_preprocessing import data_augmentation,data_preprocessing
# split data between train and test sets

(x_train, y_train),(x_test, y_test)=cifar10.load_data()
# input image dimensions
img_rows, img_cols = 32, 32
number_channels = 3
#set num of classes
num_classes = 10

#create augmented data
augmented_x=np.zeros((10000,32,32,3))
augmented_y=np.zeros((10000,1),dtype=y_train.dtype)
y=y_train.squeeze()
i=0
for c in range(10):
    augmented_x[i:i+1000,:,:,:]=x_train[y==c][:1000]
    augmented_y[i:i+1000,:]=c
    i=i+1000

# Create a tf.data pipeline of train images (and their labels)
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
train_dataset = train_dataset.map(lambda x, y: (data_preprocessing()(x), y))
aug_dataset=tf.data.Dataset.from_tensor_slices((augmented_x, augmented_y))
aug_dataset=aug_dataset.shuffle(5000)
aug_dataset = aug_dataset.map(lambda x, y: (data_preprocessing()(x), y))
aug_dataset = aug_dataset.map(lambda x, y: (data_augmentation()(x), y))
train_dataset=train_dataset.concatenate(aug_dataset)

# Create a tf.data pipeline of test images (and their labels)
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
test_dataset = test_dataset.map(lambda x, y: (data_preprocessing()(x), y))

x_train, y_train = zip(*train_dataset)
x_train = np.array(x_train)
y_train = np.array(y_train)
x_test, y_test = zip(*test_dataset)
x_test = np.array(x_test)
y_test = np.array(y_test)


if tf.keras.backend.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], number_channels, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], number_channels, img_rows, img_cols)
    input_shape = (number_channels, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, number_channels)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, number_channels)
    input_shape = (img_rows, img_cols, number_channels)

print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
class_names=['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'shi
# convert class vectors to binary class matrices
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
```

```
x_train shape: (60000, 32, 32, 3)
x_test shape: (10000, 32, 32, 3)
60000 train samples
10000 test samples
```

2.1.2 Load some sample images from CIFAR-10 [Setup - No points]

In [ ]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################

# Show some images from CIFAR-10

fig = plt.figure(figsize=(10, 10))
for i in range(9):
    random_index = np.random.randint(0, len(y_train))
    ax = fig.add_subplot(3, 3, i+1)
    ax.imshow(x_train[random_index, :])
plt.show()
```

As you can see from above, the CIFAR-10 dataset contains different types of objects. The images have been size-normalized and objects remain centered in fixed-size images.

2.1.3 Build convolutional neural network model [5pts] **[W]**

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

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

- **__init__**: See Defining Variables section
- **create_net**: See Defining Model section
- **compile_net**: See Compiling Model section

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

> INPUT: [$32 \times 32 \times 3$] will hold the raw pixel values of the image, in this case, an image of width 32, height 32. 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. We decide to set the kernel_size 3×3 for both Conv. layers. For example, the output of the Conv. layer may look like $[32 \times 32 \times 32]$ if we use 32 filters. Again, we use padding to maintain shape.
>
> MAXPOOL: MAXPOOL layer will perform a downsampling operation along the spatial dimensions (width, height). With pool size of 2×2, resulting shape takes form 16×16.
>
> DROPOUT: DROPOUT layer with the dropout rate of 0.30 to prevent overfitting.
>
> CONV: Additonal Conv. layer takes outputs from above layers and applies more filters. The Conv. layer may look like $[16 \times 16 \times 32]$. We set the kernel_size 3×3 and use padding to maintain shape for both Conv. layers.
>
> CONV: Additonal Conv. layer takes outputs from above layers and applies more filters. The Conv. layer may look like $[16 \times 16 \times 64]$.
>
> MAXPOOL: MAXPOOL layer will perform a downsampling operation along the spatial dimensions (width, height).
>
> DROPOUT: Dropout layer with the dropout rate of 0.30 to prevent overfitting.
>
> 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 0.5 to prevent overfitting.
>
> FC2： Dense layer which takes output from above layers, and has 128 neurons.
>
> DROPOUT: Dropout layer with the dropout rate of 0.5 to prevent overfitting.

> FC3: Dense layer with 10 neurons, and Softmax activation, is the final layer. The dimension of the output space is the number of classes.

Activation function: Use LeakyReLU with negative_slope 0.1 as the activation function for Conv. layers and Dense layers unless otherwise indicated to build you model architecture

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

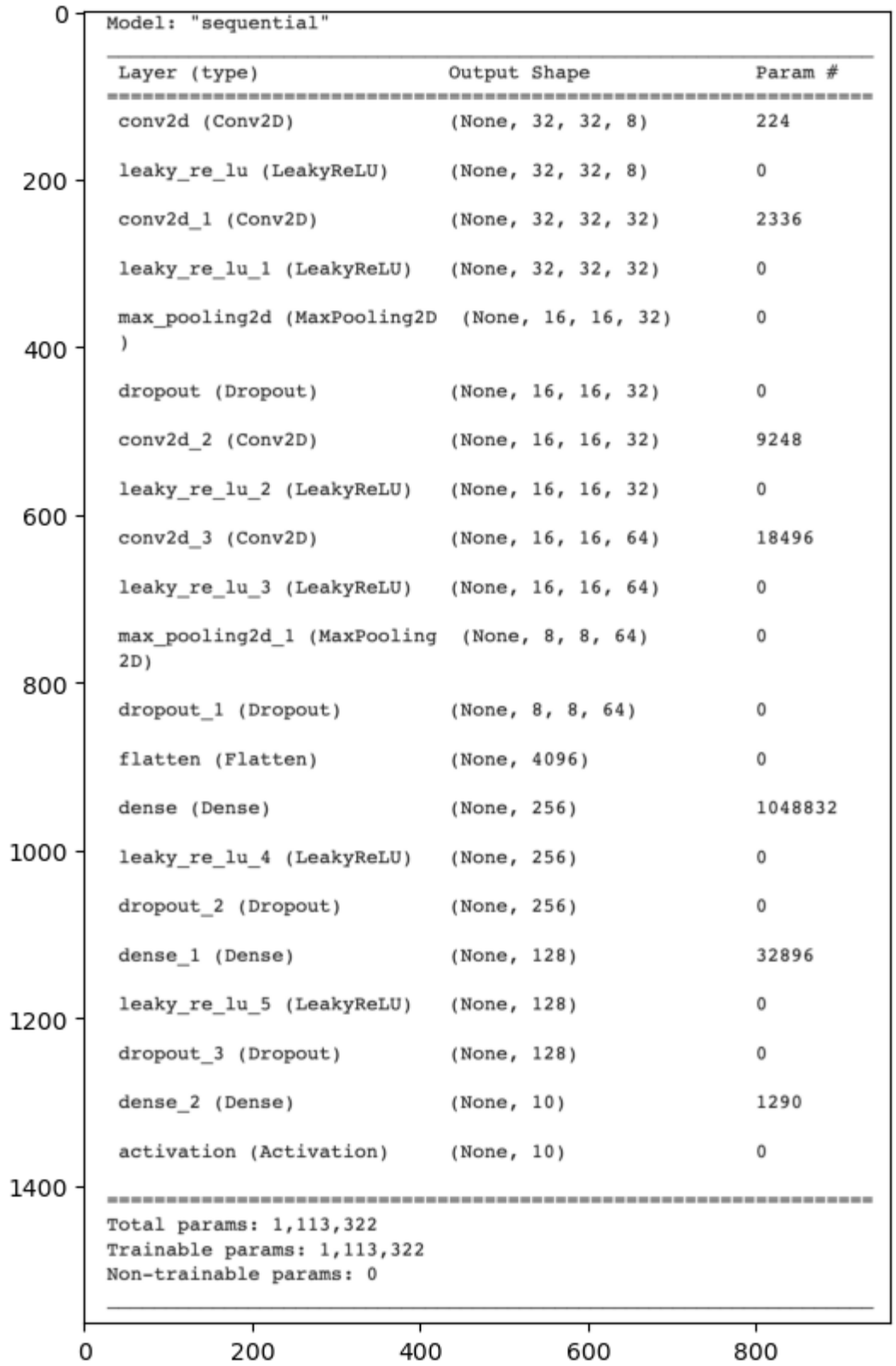
Use the following keras links to reference crucial layers of the model in keras API:

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

Lastly, if you would like to experiment with additional layers, explore the keras layers API.

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

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

Out[ ]:
```
<matplotlib.image.AxesImage at 0x237fc62e0b0>
```



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 8)         224

 leaky_re_lu (LeakyReLU)     (None, 32, 32, 8)         0

 conv2d_1 (Conv2D)           (None, 32, 32, 32)        2336

 leaky_re_lu_1 (LeakyReLU)   (None, 32, 32, 32)        0

 max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
 )

 dropout (Dropout)           (None, 16, 16, 32)        0

 conv2d_2 (Conv2D)           (None, 16, 16, 32)        9248

 leaky_re_lu_2 (LeakyReLU)   (None, 16, 16, 32)        0

 conv2d_3 (Conv2D)           (None, 16, 16, 64)        18496

 leaky_re_lu_3 (LeakyReLU)   (None, 16, 16, 64)        0

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 dropout_1 (Dropout)         (None, 8, 8, 64)          0

 flatten (Flatten)           (None, 4096)              0

 dense (Dense)               (None, 256)               1048832

 leaky_re_lu_4 (LeakyReLU)   (None, 256)               0

 dropout_2 (Dropout)         (None, 256)               0

 dense_1 (Dense)             (None, 128)               32896

 leaky_re_lu_5 (LeakyReLU)   (None, 128)               0

 dropout_3 (Dropout)         (None, 128)               0

 dense_2 (Dense)             (None, 10)                1290

 activation (Activation)     (None, 10)                0

=================================================================
Total params: 1,113,322
Trainable params: 1,113,322
Non-trainable params: 0
_________________________________________________________________
```


Defining Variables [No Points]

You now need to set training variables in the `__init__()` function in **cnn.py**. Once you have defined variables you may use the cell below to see them.

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

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

# You can adjust parameters to train your model in __init__() in cnn.py

from cnn import CNN

net = CNN()
batch_size, epochs, init_lr = net.get_vars()
print(f'Batch Size\t: {batch_size} \nEpochs\t\t: {epochs} \nLearning Rate\t: {init_lr} \n'
```

```
Batch Size      : 64
Epochs          : 5
Learning Rate   : 0.001
```

Defining model [5pts Bonus for Undergrad]**[W]**

You now need to complete the `create_net()` function in **cnn.py** to define your model structure. Once you have defined a model structure you may use the cell below to examine your architecture.

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.

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

# model.summary() gives you details of your architecture.
#You can compare your architecture with the 'Architecture.png'

from cnn import CNN
net = CNN()

s = tf.keras.backend.clear_session()
model=net.create_net()
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 8)         224

 leaky_re_lu (LeakyReLU)     (None, 32, 32, 8)         0

 conv2d_1 (Conv2D)           (None, 32, 32, 32)        2336

 leaky_re_lu_1 (LeakyReLU)   (None, 32, 32, 32)        0

 max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
 )

 dropout (Dropout)           (None, 16, 16, 32)        0

 conv2d_2 (Conv2D)           (None, 16, 16, 32)        9248

 leaky_re_lu_2 (LeakyReLU)   (None, 16, 16, 32)        0

 conv2d_3 (Conv2D)           (None, 16, 16, 64)        18496

 leaky_re_lu_3 (LeakyReLU)   (None, 16, 16, 64)        0

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 dropout_1 (Dropout)         (None, 8, 8, 64)          0

 flatten (Flatten)           (None, 4096)              0

 dense (Dense)               (None, 256)               1048832

 dropout_2 (Dropout)         (None, 256)               0

 dense_1 (Dense)             (None, 128)               32896

 dropout_3 (Dropout)         (None, 128)               0

 dense_2 (Dense)             (None, 10)                1290

 softmax (Softmax)           (None, 10)                0

=================================================================
Total params: 1,113,322
Trainable params: 1,113,322
Non-trainable params: 0
_________________________________________________________________
```

Compiling model [No Points]

Next prepare the model for training by completing `compile_model()` in **cnn.py**. Remember we are performing 10-way clasification when selecting a loss function. Loss function can be categorical crossentropy.

Note that the compile method configures the model for training. You can get more insights into this method here.

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

# Complete compile_model() in cnn.py.
from cnn import CNN
net = CNN()
model = net.compile_net(model)
print(model)
```

```
<keras.engine.sequential.Sequential object at 0x00000237DF0A9240>
```

2.1.4 Train the network [8pts total (3pts, 3pts, 2pts) Bonus for Undergrad] **[W]**

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

Expected Result: You should be able to achieve more than 79% accuracy on the test set to get full points. If you achieve accuracy between 60% to 69%, you will only get 3 points. An accuracy between 69% to 79% will earn an additional 3pts.

Note: If you would like to automate the tuning process, you can use a nested for loop to search for the hyperparameter that achieves the accuracy.

- 60% to 69% earns 3pts
- 69% to 79% earns 3pts more (6pts total)
- 79%+ earns 2pts more (8pts total)

Train your own CNN model

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


from cnn import CNN

net = CNN()
batch_size, epochs, init_lr = net.get_vars()

def lr_scheduler(epoch):
    new_lr = init_lr * 0.9 ** epoch
    print("Learning rate:", new_lr)
    return new_lr

history = model.fit(
    x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
    callbacks=[tf.keras.callbacks.LearningRateScheduler(lr_scheduler)],
    shuffle=True,
    verbose=1,
    initial_epoch=0,
    validation_data=(x_test, y_test)
)
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

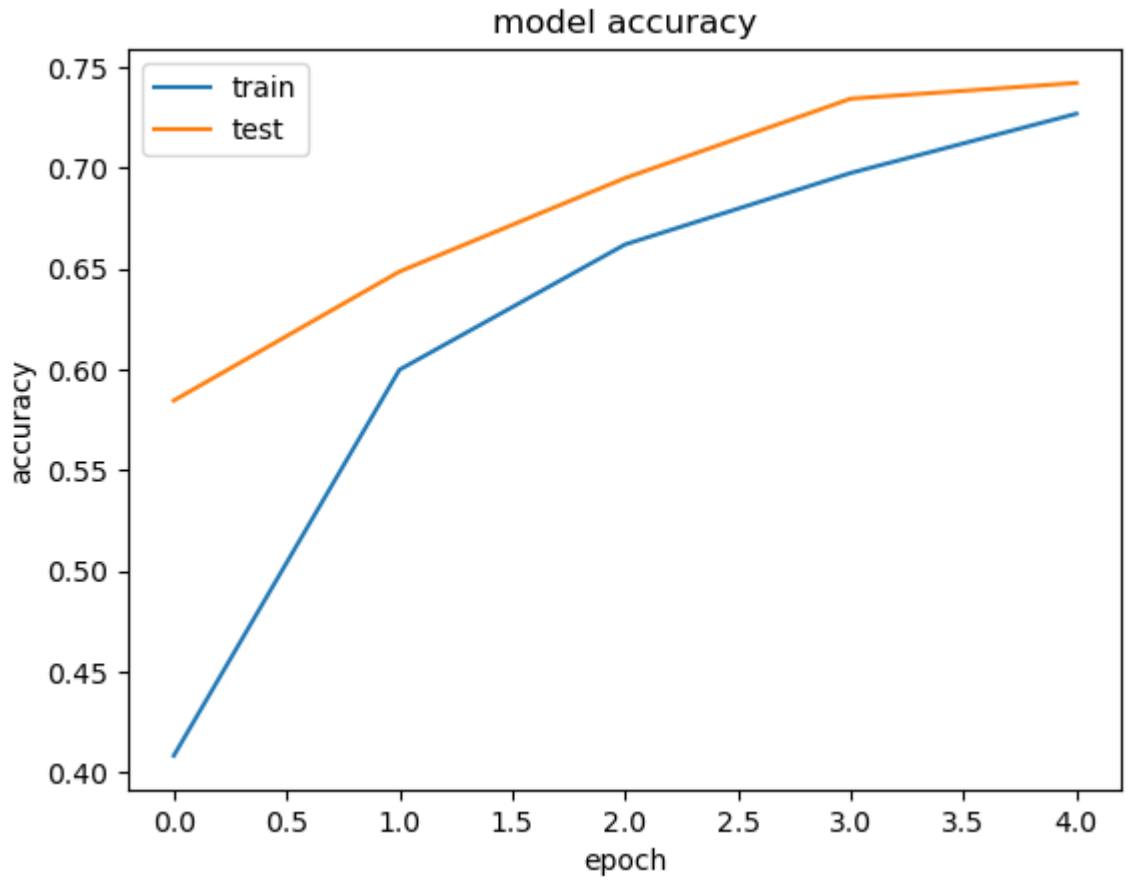
```
Learning rate: 0.001
Epoch 1/5
938/938 [==============================] - 121s 125ms/step - loss: 1.5974 - categorical_a
ccuracy: 0.4083 - val_loss: 1.1697 - val_categorical_accuracy: 0.5845 - lr: 0.0010
Learning rate: 0.0009000000000000001
Epoch 2/5
938/938 [==============================] - 100s 106ms/step - loss: 1.1326 - categorical_a
ccuracy: 0.5998 - val_loss: 0.9903 - val_categorical_accuracy: 0.6485 - lr: 9.0000e-04
Learning rate: 0.0008100000000000001
Epoch 3/5
938/938 [==============================] - 111s 118ms/step - loss: 0.9678 - categorical_a
ccuracy: 0.6620 - val_loss: 0.8594 - val_categorical_accuracy: 0.6949 - lr: 8.1000e-04
Learning rate: 0.0007290000000000002
Epoch 4/5
938/938 [==============================] - 121s 129ms/step - loss: 0.8669 - categorical_a
ccuracy: 0.6974 - val_loss: 0.7586 - val_categorical_accuracy: 0.7343 - lr: 7.2900e-04
Learning rate: 0.0006561000000000001
Epoch 5/5
938/938 [==============================] - 110s 117ms/step - loss: 0.7896 - categorical_a
ccuracy: 0.7268 - val_loss: 0.7425 - val_categorical_accuracy: 0.7421 - lr: 6.5610e-04
Test loss: 0.7425193190574646
Test accuracy: 0.7421000003814697
```

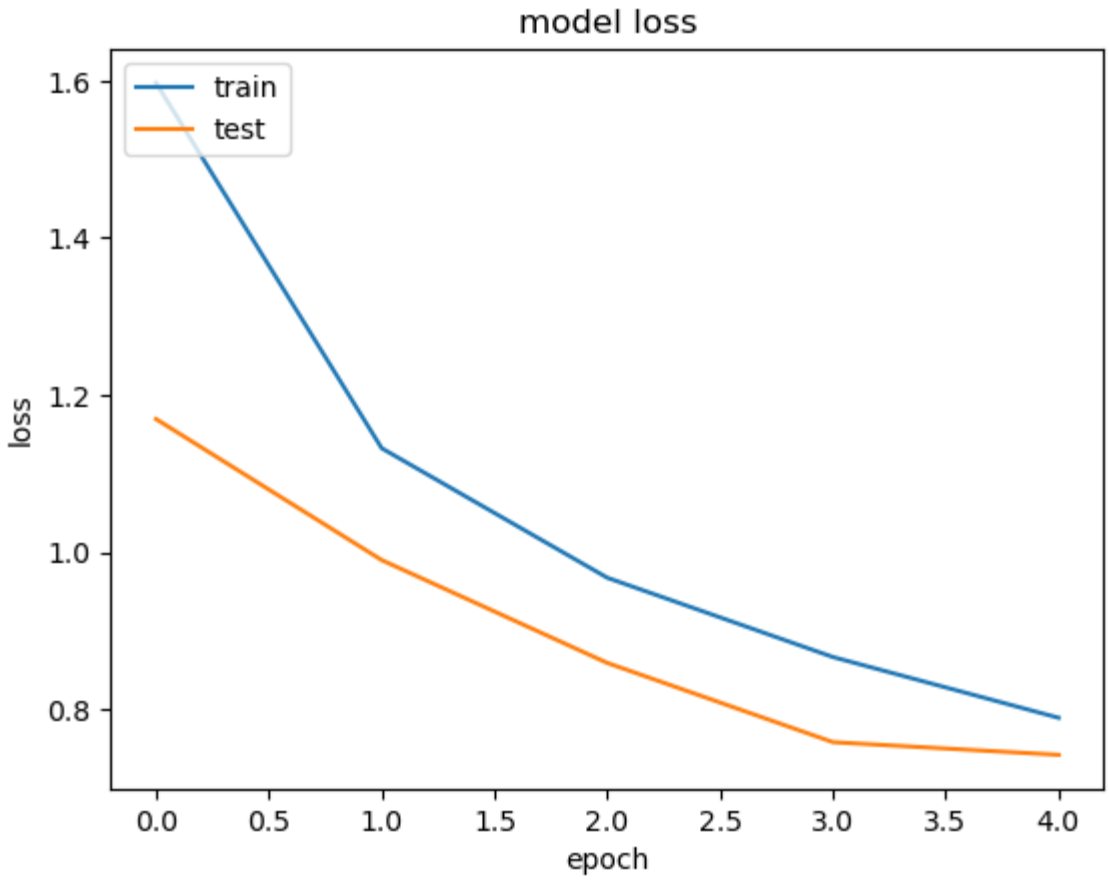
2.1.5 Examine accuracy and loss [2pts Bonus for Undergrad] **[W]**

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

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

# list all data in history
print(history.history.keys())

# summarize history for accuracy and loss
plt.plot(history.history['categorical_accuracy'])
plt.plot(history.history['val_categorical_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

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

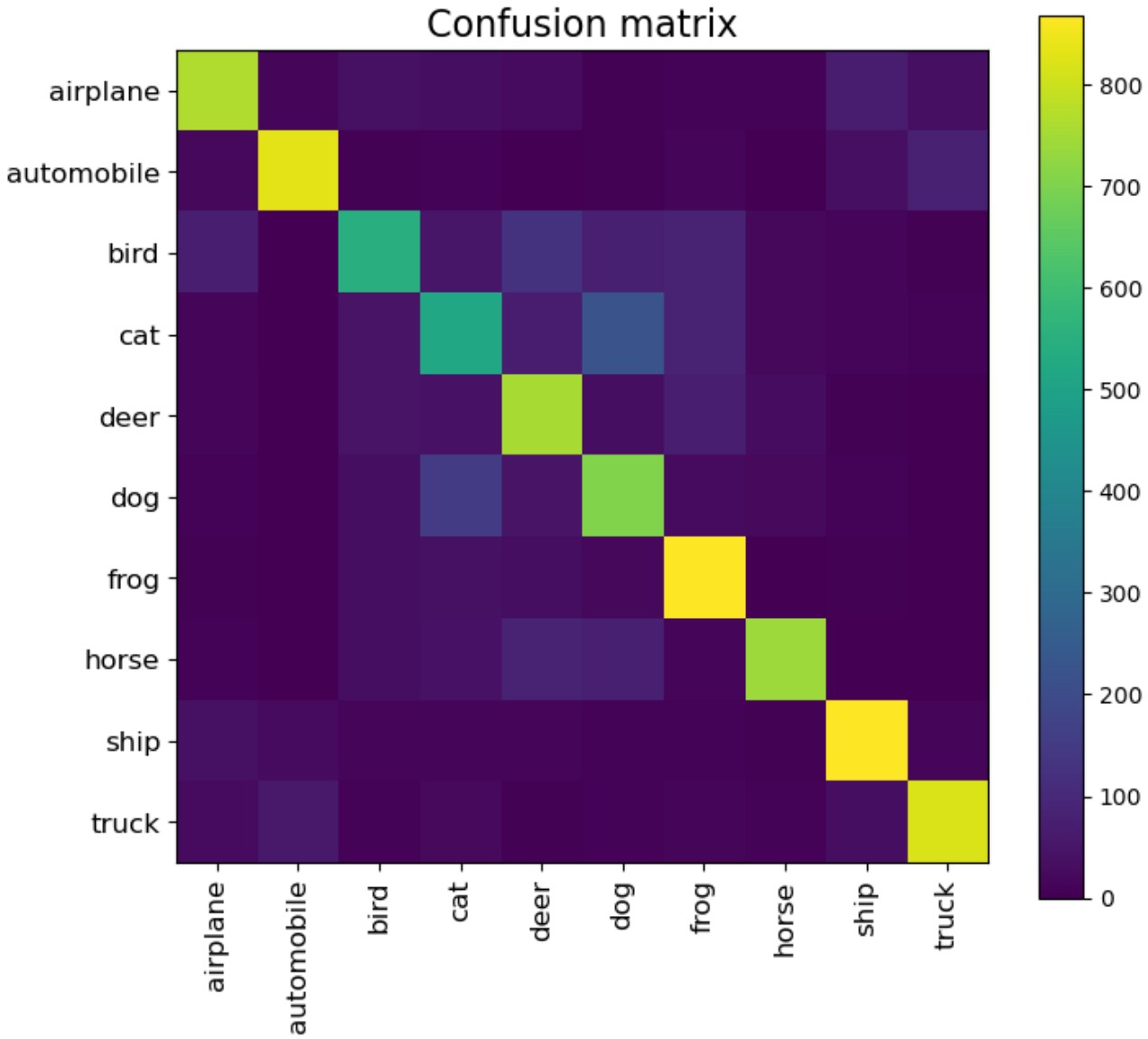
```
dict_keys(['loss', 'categorical_accuracy', 'val_loss', 'val_categorical_accuracy', 'lr'])
```





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

# make predictions
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_pred_prob = np.max(y_pred, axis=1)
y_gt_classes = np.argmax(y_test, axis=1)

from sklearn.metrics import confusion_matrix, accuracy_score
plt.figure(figsize=(8, 7))
plt.imshow(confusion_matrix(y_gt_classes, y_pred_classes))
plt.title('Confusion matrix', fontsize=16)
plt.xticks(np.arange(10), class_names, rotation=90, fontsize=12)
plt.yticks(np.arange(10), class_names, fontsize=12)
plt.colorbar()
plt.show()
```

```
313/313 [==============================] - 4s 10ms/step
```



2.2 Exploring Deep CNN Architectures [5pts Bonus for All] **[W]**

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

One such problem is the problem of the vanishing gradient. The weights of a neural network are updated using the backpropagation algorithm. The backpropagation algorithm makes a small change to each weight in such a way that the loss of the model decreases. Using the chain rule, we can find this gradient for each weight. But, as this gradient keeps flowing backwards to the initial layers, this value keeps getting multiplied by each local gradient. Hence, the gradient becomes smaller and smaller, making the updates to the initial layers very small, increasing the training time considerably.

Many tactics have been used in an effort to solve this problem. One architecture, named ResNet, solves the vanishing gradient problem in a unique way. ResNet was developed at Microsoft Research to find better ways to train deep networks. Take a moment to explore how ResNet tackles the vanishing gradient problem by reading the original research paper here: https://arxiv.org/pdf/1512.03385.pdf (also included as PDF in papers directory).

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

Answer:

...

3: Random Forests [45pts; 40pts + 5pts Bonus for All] **[P]** **[W]**

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

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

3.1 Random Forest Implementation [35pts] **[P]**

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

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

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

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

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

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

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

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

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

NOTES:

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

3.2 Hyperparameter Tuning with a Random Forest [5pts] **[P]**

In machine learning, hyperparameters are parameters that are set before the learning process begins. The max_depth, num_estimators, or max_features variables from 3.1 are examples of different hyperparameters for a random forest model. Let's first review the dataset in a bit more detail.

Dataset Objective

Imagine that we are a team of researchers in the US Census Bureau working to track and document various information pertaining to income for a machine learning model that predicts whether or not a person makes over 50K a year. We know that there are multiple things to track such as hours worked per week in order to determine how much a person makes in the year. We will use the information we track and document in order to publish it for the general public, which will be helpful for people interested in diversifying their incomes and making more money.

After much deliberation amongst the team of researchers, you come to a conclusion that we can use the past observations of how much people have earned in the past to create a model.

We will use our random forest algorithm from Q3.1 to predict if person earns above 50K or not a year.

You can find more information on the dataset here.

Loading the dataset

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

There were 13 features used in this dataset.

Inputs:

1. age: continuous.
2. workclass: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked.
3. fnlwgt: continuous.
4. education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool.
5. education-num: continuous.
6. marital-status: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse.
7. occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces.
8. relationship: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried.
9. sex at birth: Female, Male.
10. capital-gain: continuous.
11. capital-loss: continuous.
12. hours-per-week: continuous.
13. native-country: United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holand-Netherlands.

Output:

1. (high income) target value:
    - h (aka > 50K) means the someone's income is high
    - l (aka <=50K) means the someone's income is low

Your random forest model will try to predict this variable.

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
from sklearn import preprocessing
import pandas as pd
import numpy as np
preprocessor = preprocessing.OrdinalEncoder()

data_train = pd.read_csv("./data/income_train_clean.csv")
data_test = pd.read_csv("./data/income_test_clean.csv")
numerical_features = ["age", "fnlwgt", "education-num", "capital-gain", "capital-loss", "h
data_train_numerical = data_train[numerical_features]
data_test_numerical = data_test[numerical_features]
data_train_categorical = data_train.drop(numerical_features, axis=1)
data_test_categorical = data_test.drop(numerical_features, axis=1)

preprocessor.fit(data_train_categorical)
data_train_categorical = pd.DataFrame(preprocessor.transform(data_train_categorical), colu
data_test_categorical = pd.DataFrame(preprocessor.transform(data_test_categorical), columr
data_train = pd.concat([data_train_categorical, data_train_numerical], axis=1)
data_test = pd.concat([data_test_categorical, data_test_numerical], axis=1)

X_train = data_train.drop(columns = 'income')
y_train = data_train['income']
y_train = y_train.to_numpy()
X_test = data_test.drop(columns = 'income')
X_test = np.array(X_test)
y_test = data_test['income']
y_test = y_test.to_numpy()
X_train, y_train, X_test, y_test = np.array(X_train), np.array(y_train), np.array(X_test),
```

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
assert X_train.shape == (30162, 13)
assert y_train.shape == (30162,)
assert X_test.shape == (12966, 13)
assert y_test.shape == (12966,)
```

```
(30162, 13) (30162,) (12966, 13) (12966,)
```

In the following codeblock, train your random forest model with different values for max_depth, n_estimators, or max_features and evaluate each model on the held-out test set. Try to choose a combination of hyperparameters that maximizes your prediction accuracy on the test set (aim for 80%+).

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

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

Submit this file to Gradescope. You must achieve at least a **80% accuracy** against the test set in Gradescope to receive full credit for this section.

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

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

```
test_bootstrapping passed!
```

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

Tune these three parameters to achieve a better accuracy. n_estimators and max_depth must
be at least 3 in value for moderately reliable answers. While you can use the provided tes
to evaluate your implementation, you will need to obtain 80% on the test set to receive fu
credit for this section.
"""
from random_forest import RandomForest
from sklearn import preprocessing
import sklearn.ensemble

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

################## CHANGE THESE VALUES ################################
n_estimators = 6 #Hint: Consider values between 3-15.
max_depth = 6 # Hint: Consider values betweeen 3-15.
max_features = 0.6 # Hint: Consider values betweeen 0.3-1.0.
######################################################################
random_forest = RandomForest(n_estimators, max_depth, max_features, random_seed=student_ra
random_forest.fit(X_train, y_train)
accuracy=random_forest.OOB_score(X_test, y_test)
print("accuracy: %.4f" % accuracy)
```

```
accuracy: 0.7548
```

In [ ]:
```
## Grid search
student_random_seed = 4641 + 7641

# # Just odd hyperparams - done already
# n_estimators_vec = np.arange(3, 16, 2)
# max_depth_vec = np.arange(3, 16, 2)
# max_features_vec = np.arange(3, 11, 2)

# # Just even hyperparams
# n_estimators_vec = np.arange(4, 16, 2)
# max_depth_vec = np.arange(4, 16, 2)
# max_features_vec = np.arange(4, 11, 2)

# # All hyperparams, but k = [0.3 0.4] only
# n_estimators_vec = np.arange(3, 16)
# max_depth_vec = np.arange(3, 16)
# max_features_vec = np.arange(3, 5)

# Best range of hyperparams
n_estimators_vec = np.arange(5, 16)
max_depth_vec = np.arange(3, 10)
max_features_vec = np.arange(3, 11)

accuracy = np.zeros((16, 16, 11))
acc, max_acc, i_max, j_max, k_max, i2, j2, k2, max2 = 0, 0, 0, 0, 0, 0, 0, 0, 0
for i in n_estimators_vec:
    for j in max_depth_vec:
        for k in max_features_vec:
            random_forest = RandomForest(i, j, k/10, random_seed=student_random_seed)
            random_forest.fit(X_train, y_train)
            acc=random_forest.OOB_score(X_test, y_test)
            accuracy[i,j,k] = acc
            # print("accuracy: %.4f" % accuracy)

            if acc > max_acc:
                max2, i2, j2, k2 = max_acc, i_max, j_max, k_max
                max_acc = acc
                i_max = i
                j_max = j
                k_max = k

            print(acc, i, j, k/10, "max of", max_acc, "at", i_max, j_max, k_max/10)


print("Best accuracy:", max_acc, "at [", i_max, j_max, k_max/10, "]")
print("Second best accuracy:", max2, "at [", i2, j2, k2/10, "]")
# print(n_estimators_vec[i_max])
# print(max_depth_vec[j_max])
# print(max_features_vec[k_max])

print("From previous runs, the best result was 0.7559 at [15 , 3, 0.3]") #Enter values mar
print("From previous runs, the 2nd best result was 0.7548 at [6 , 6, 0.6]") #Enter values
```

```
0.7535496084674297 5 3 0.3 max of 0.7535496084674297 at 5 3 0.3
0.7535496084674297 5 3 0.4 max of 0.7535496084674297 at 5 3 0.3
0.7535496084674297 5 3 0.5 max of 0.7535496084674297 at 5 3 0.3
0.7535496084674297 5 3 0.6 max of 0.7535496084674297 at 5 3 0.3
0.7535496084674297 5 3 0.7 max of 0.7535496084674297 at 5 3 0.3
0.7535496084674297 5 3 0.8 max of 0.7535496084674297 at 5 3 0.3
0.7535496084674297 5 3 0.9 max of 0.7535496084674297 at 5 3 0.3
0.7535496084674297 5 3 1.0 max of 0.7535496084674297 at 5 3 0.3
0.7533846771649025 5 4 0.3 max of 0.7535496084674297 at 5 3 0.3
0.7533846771649025 5 4 0.4 max of 0.7535496084674297 at 5 3 0.3
0.7533846771649025 5 4 0.5 max of 0.7535496084674297 at 5 3 0.3
0.7533846771649025 5 4 0.6 max of 0.7535496084674297 at 5 3 0.3
0.7533846771649025 5 4 0.7 max of 0.7535496084674297 at 5 3 0.3
0.7533846771649025 5 4 0.8 max of 0.7535496084674297 at 5 3 0.3
0.7533846771649025 5 4 0.9 max of 0.7535496084674297 at 5 3 0.3
0.7533846771649025 5 4 1.0 max of 0.7535496084674297 at 5 3 0.3
0.7535998049508074 5 5 0.3 max of 0.7535998049508074 at 5 5 0.3
0.7535998049508074 5 5 0.4 max of 0.7535998049508074 at 5 5 0.3
0.7535998049508074 5 5 0.5 max of 0.7535998049508074 at 5 5 0.3
0.7535998049508074 5 5 0.6 max of 0.7535998049508074 at 5 5 0.3
0.7535998049508074 5 5 0.7 max of 0.7535998049508074 at 5 5 0.3
0.7535998049508074 5 5 0.8 max of 0.7535998049508074 at 5 5 0.3
0.7535998049508074 5 5 0.9 max of 0.7535998049508074 at 5 5 0.3
0.7535998049508074 5 5 1.0 max of 0.7535998049508074 at 5 5 0.3
0.7533559934601153 5 6 0.3 max of 0.7535998049508074 at 5 5 0.3
0.7533559934601153 5 6 0.4 max of 0.7535998049508074 at 5 5 0.3
0.7533559934601153 5 6 0.5 max of 0.7535998049508074 at 5 5 0.3
0.7533559934601153 5 6 0.6 max of 0.7535998049508074 at 5 5 0.3
0.7533559934601153 5 6 0.7 max of 0.7535998049508074 at 5 5 0.3
0.7533559934601153 5 6 0.8 max of 0.7535998049508074 at 5 5 0.3
0.7533559934601153 5 6 0.9 max of 0.7535998049508074 at 5 5 0.3
0.7533559934601153 5 6 1.0 max of 0.7535998049508074 at 5 5 0.3
0.7528655021082522 5 7 0.3 max of 0.7535998049508074 at 5 5 0.3
0.7528655021082522 5 7 0.4 max of 0.7535998049508074 at 5 5 0.3
0.7528655021082522 5 7 0.5 max of 0.7535998049508074 at 5 5 0.3
0.7528655021082522 5 7 0.6 max of 0.7535998049508074 at 5 5 0.3
0.7528655021082522 5 7 0.7 max of 0.7535998049508074 at 5 5 0.3
0.7528655021082522 5 7 0.8 max of 0.7535998049508074 at 5 5 0.3
0.7528655021082522 5 7 0.9 max of 0.7535998049508074 at 5 5 0.3
0.7528655021082522 5 7 1.0 max of 0.7535998049508074 at 5 5 0.3
0.7534061899434932 5 8 0.3 max of 0.7535998049508074 at 5 5 0.3
0.7534061899434932 5 8 0.4 max of 0.7535998049508074 at 5 5 0.3
0.7534061899434932 5 8 0.5 max of 0.7535998049508074 at 5 5 0.3
0.7534061899434932 5 8 0.6 max of 0.7535998049508074 at 5 5 0.3
0.7534061899434932 5 8 0.7 max of 0.7535998049508074 at 5 5 0.3
0.7534061899434932 5 8 0.8 max of 0.7535998049508074 at 5 5 0.3
0.7534061899434932 5 8 0.9 max of 0.7535998049508074 at 5 5 0.3
0.7534061899434932 5 8 1.0 max of 0.7535998049508074 at 5 5 0.3
0.7519117689240743 5 9 0.3 max of 0.7535998049508074 at 5 5 0.3
0.7519117689240743 5 9 0.4 max of 0.7535998049508074 at 5 5 0.3
0.7519117689240743 5 9 0.5 max of 0.7535998049508074 at 5 5 0.3
0.7519117689240743 5 9 0.6 max of 0.7535998049508074 at 5 5 0.3
0.7519117689240743 5 9 0.7 max of 0.7535998049508074 at 5 5 0.3
0.7519117689240743 5 9 0.8 max of 0.7535998049508074 at 5 5 0.3
0.7519117689240743 5 9 0.9 max of 0.7535998049508074 at 5 5 0.3
0.7519117689240743 5 9 1.0 max of 0.7535998049508074 at 5 5 0.3
0.7548211636723257 6 3 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7548211636723257 6 3 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7548211636723257 6 3 0.5 max of 0.7548211636723257 at 6 3 0.3
```

```
0.7548211636723257 6 3 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7548211636723257 6 3 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7548211636723257 6 3 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7548211636723257 6 3 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7548211636723257 6 3 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7547799571452118 6 4 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7547799571452118 6 4 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7547799571452118 6 4 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7547799571452118 6 4 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7547799571452118 6 4 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7547799571452118 6 4 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7547799571452118 6 4 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7547799571452118 6 4 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7544709081918576 6 5 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7544709081918576 6 5 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7544709081918576 6 5 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7544709081918576 6 5 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7544709081918576 6 5 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7544709081918576 6 5 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7544709081918576 6 5 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7544709081918576 6 5 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7547744629415966 6 6 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7547744629415966 6 6 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7547744629415966 6 6 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7547744629415966 6 6 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7547744629415966 6 6 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7547744629415966 6 6 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7547744629415966 6 6 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7547744629415966 6 6 1.0 max of 0.7548211636723257 at 6 3 0.3
0.754198945112906 6 7 0.3 max of 0.7548211636723257 at 6 3 0.3
0.754198945112906 6 7 0.4 max of 0.7548211636723257 at 6 3 0.3
0.754198945112906 6 7 0.5 max of 0.7548211636723257 at 6 3 0.3
0.754198945112906 6 7 0.6 max of 0.7548211636723257 at 6 3 0.3
0.754198945112906 6 7 0.7 max of 0.7548211636723257 at 6 3 0.3
0.754198945112906 6 7 0.8 max of 0.7548211636723257 at 6 3 0.3
0.754198945112906 6 7 0.9 max of 0.7548211636723257 at 6 3 0.3
0.754198945112906 6 7 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7537923740453821 6 8 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7537923740453821 6 8 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7537923740453821 6 8 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7537923740453821 6 8 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7537923740453821 6 8 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7537923740453821 6 8 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7537923740453821 6 8 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7537923740453821 6 8 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7539214878303389 6 9 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7539214878303389 6 9 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7539214878303389 6 9 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7539214878303389 6 9 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7539214878303389 6 9 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7539214878303389 6 9 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7539214878303389 6 9 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7539214878303389 6 9 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7536365828176484 7 3 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7536365828176484 7 3 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7536365828176484 7 3 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7536365828176484 7 3 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7536365828176484 7 3 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7536365828176484 7 3 0.8 max of 0.7548211636723257 at 6 3 0.3
```

```
0.7536365828176484 7 3 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7536365828176484 7 3 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7535696107583916 7 4 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7535696107583916 7 4 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7535696107583916 7 4 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7535696107583916 7 4 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7535696107583916 7 4 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7535696107583916 7 4 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7535696107583916 7 4 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7535696107583916 7 4 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7536700688472768 7 5 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7536700688472768 7 5 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7536700688472768 7 5 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7536700688472768 7 5 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7536700688472768 7 5 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7536700688472768 7 5 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7536700688472768 7 5 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7536700688472768 7 5 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7533847678748427 7 6 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7533847678748427 7 6 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7533847678748427 7 6 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7533847678748427 7 6 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7533847678748427 7 6 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7533847678748427 7 6 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7533847678748427 7 6 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7533847678748427 7 6 1.0 max of 0.7548211636723257 at 6 3 0.3
0.753464368950988 7 7 0.3 max of 0.7548211636723257 at 6 3 0.3
0.753464368950988 7 7 0.4 max of 0.7548211636723257 at 6 3 0.3
0.753464368950988 7 7 0.5 max of 0.7548211636723257 at 6 3 0.3
0.753464368950988 7 7 0.6 max of 0.7548211636723257 at 6 3 0.3
0.753464368950988 7 7 0.7 max of 0.7548211636723257 at 6 3 0.3
0.753464368950988 7 7 0.8 max of 0.7548211636723257 at 6 3 0.3
0.753464368950988 7 7 0.9 max of 0.7548211636723257 at 6 3 0.3
0.753464368950988 7 7 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7529722199898202 7 8 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7529722199898202 7 8 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7529722199898202 7 8 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7529722199898202 7 8 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7529722199898202 7 8 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7529722199898202 7 8 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7529722199898202 7 8 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7529722199898202 7 8 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7521216748372579 7 9 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7521216748372579 7 9 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7521216748372579 7 9 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7521216748372579 7 9 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7521216748372579 7 9 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7521216748372579 7 9 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7521216748372579 7 9 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7521216748372579 7 9 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7533422988687604 8 3 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7533422988687604 8 3 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7533422988687604 8 3 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7533422988687604 8 3 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7533422988687604 8 3 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7533422988687604 8 3 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7533422988687604 8 3 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7533422988687604 8 3 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7533726234738812 8 4 0.3 max of 0.7548211636723257 at 6 3 0.3
```

```
0.7533726234738812 8 4 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7533726234738812 8 4 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7533726234738812 8 4 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7533726234738812 8 4 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7533726234738812 8 4 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7533726234738812 8 4 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7533726234738812 8 4 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7533172481080084 8 5 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7533172481080084 8 5 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7533172481080084 8 5 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7533172481080084 8 5 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7533172481080084 8 5 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7533172481080084 8 5 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7533172481080084 8 5 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7533172481080084 8 5 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7533040634970862 8 6 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7533040634970862 8 6 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7533040634970862 8 6 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7533040634970862 8 6 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7533040634970862 8 6 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7533040634970862 8 6 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7533040634970862 8 6 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7533040634970862 8 6 1.0 max of 0.7548211636723257 at 6 3 0.3
0.752912480552699 8 7 0.3 max of 0.7548211636723257 at 6 3 0.3
0.752912480552699 8 7 0.4 max of 0.7548211636723257 at 6 3 0.3
0.752912480552699 8 7 0.5 max of 0.7548211636723257 at 6 3 0.3
0.752912480552699 8 7 0.6 max of 0.7548211636723257 at 6 3 0.3
0.752912480552699 8 7 0.7 max of 0.7548211636723257 at 6 3 0.3
0.752912480552699 8 7 0.8 max of 0.7548211636723257 at 6 3 0.3
0.752912480552699 8 7 0.9 max of 0.7548211636723257 at 6 3 0.3
0.752912480552699 8 7 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7528676528755637 8 8 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7528676528755637 8 8 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7528676528755637 8 8 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7528676528755637 8 8 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7528676528755637 8 8 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7528676528755637 8 8 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7528676528755637 8 8 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7528676528755637 8 8 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7515636948553648 8 9 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7515636948553648 8 9 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7515636948553648 8 9 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7515636948553648 8 9 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7515636948553648 8 9 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7515636948553648 8 9 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7515636948553648 8 9 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7515636948553648 8 9 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7536467137595575 9 3 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7536467137595575 9 3 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7536467137595575 9 3 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7536467137595575 9 3 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7536467137595575 9 3 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7536467137595575 9 3 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7536467137595575 9 3 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7536467137595575 9 3 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7536243241125303 9 4 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7536243241125303 9 4 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7536243241125303 9 4 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7536243241125303 9 4 0.6 max of 0.7548211636723257 at 6 3 0.3
```

```
0.7536243241125303 9 4 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7536243241125303 9 4 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7536243241125303 9 4 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7536243241125303 9 4 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7537321675790448 9 5 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7537321675790448 9 5 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7537321675790448 9 5 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7537321675790448 9 5 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7537321675790448 9 5 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7537321675790448 9 5 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7537321675790448 9 5 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7537321675790448 9 5 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7536648120575712 9 6 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7536648120575712 9 6 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7536648120575712 9 6 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7536648120575712 9 6 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7536648120575712 9 6 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7536648120575712 9 6 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7536648120575712 9 6 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7536648120575712 9 6 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7534285390879701 9 7 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7534285390879701 9 7 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7534285390879701 9 7 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7534285390879701 9 7 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7534285390879701 9 7 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7534285390879701 9 7 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7534285390879701 9 7 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7534285390879701 9 7 1.0 max of 0.7548211636723257 at 6 3 0.3
0.75329942545678 9 8 0.3 max of 0.7548211636723257 at 6 3 0.3
0.75329942545678 9 8 0.4 max of 0.7548211636723257 at 6 3 0.3
0.75329942545678 9 8 0.5 max of 0.7548211636723257 at 6 3 0.3
0.75329942545678 9 8 0.6 max of 0.7548211636723257 at 6 3 0.3
0.75329942545678 9 8 0.7 max of 0.7548211636723257 at 6 3 0.3
0.75329942545678 9 8 0.8 max of 0.7548211636723257 at 6 3 0.3
0.75329942545678 9 8 0.9 max of 0.7548211636723257 at 6 3 0.3
0.75329942545678 9 8 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7527393733137797 9 9 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7527393733137797 9 9 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7527393733137797 9 9 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7527393733137797 9 9 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7527393733137797 9 9 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7527393733137797 9 9 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7527393733137797 9 9 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7527393733137797 9 9 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7540380417835983 10 3 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7540380417835983 10 3 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7540380417835983 10 3 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7540380417835983 10 3 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7540380417835983 10 3 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7540380417835983 10 3 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7540380417835983 10 3 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7540380417835983 10 3 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7540424963250033 10 4 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7540424963250033 10 4 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7540424963250033 10 4 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7540424963250033 10 4 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7540424963250033 10 4 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7540424963250033 10 4 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7540424963250033 10 4 0.9 max of 0.7548211636723257 at 6 3 0.3
```

```
0.7540424963250033 10 4 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7539341024841493 10 5 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7539341024841493 10 5 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7539341024841493 10 5 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7539341024841493 10 5 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7539341024841493 10 5 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7539341024841493 10 5 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7539341024841493 10 5 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7539341024841493 10 5 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7539263070366906 10 6 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7539263070366906 10 6 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7539263070366906 10 6 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7539263070366906 10 6 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7539263070366906 10 6 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7539263070366906 10 6 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7539263070366906 10 6 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7539263070366906 10 6 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7537001462574429 10 7 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7537001462574429 10 7 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7537001462574429 10 7 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7537001462574429 10 7 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7537001462574429 10 7 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7537001462574429 10 7 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7537001462574429 10 7 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7537001462574429 10 7 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7534529501437827 10 8 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7534529501437827 10 8 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7534529501437827 10 8 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7534529501437827 10 8 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7534529501437827 10 8 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7534529501437827 10 8 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7534529501437827 10 8 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7534529501437827 10 8 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7527364804668359 10 9 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7527364804668359 10 9 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7527364804668359 10 9 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7527364804668359 10 9 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7527364804668359 10 9 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7527364804668359 10 9 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7527364804668359 10 9 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7527364804668359 10 9 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7539322209287285 11 3 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7539322209287285 11 3 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7539322209287285 11 3 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7539322209287285 11 3 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7539322209287285 11 3 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7539322209287285 11 3 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7539322209287285 11 3 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7539322209287285 11 3 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7539542103220799 11 4 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7539542103220799 11 4 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7539542103220799 11 4 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7539542103220799 11 4 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7539542103220799 11 4 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7539542103220799 11 4 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7539542103220799 11 4 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7539542103220799 11 4 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7537666537317294 11 5 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7537666537317294 11 5 0.4 max of 0.7548211636723257 at 6 3 0.3
```

```
0.7537666537317294 11 5 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7537666537317294 11 5 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7537666537317294 11 5 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7537666537317294 11 5 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7537666537317294 11 5 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7537666537317294 11 5 1.0 max of 0.7548211636723257 at 6 3 0.3
0.753945155865994 11 6 0.3 max of 0.7548211636723257 at 6 3 0.3
0.753945155865994 11 6 0.4 max of 0.7548211636723257 at 6 3 0.3
0.753945155865994 11 6 0.5 max of 0.7548211636723257 at 6 3 0.3
0.753945155865994 11 6 0.6 max of 0.7548211636723257 at 6 3 0.3
0.753945155865994 11 6 0.7 max of 0.7548211636723257 at 6 3 0.3
0.753945155865994 11 6 0.8 max of 0.7548211636723257 at 6 3 0.3
0.753945155865994 11 6 0.9 max of 0.7548211636723257 at 6 3 0.3
0.753945155865994 11 6 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7538758615592144 11 7 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7538758615592144 11 7 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7538758615592144 11 7 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7538758615592144 11 7 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7538758615592144 11 7 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7538758615592144 11 7 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7538758615592144 11 7 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7538758615592144 11 7 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7539007459147157 11 8 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7539007459147157 11 8 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7539007459147157 11 8 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7539007459147157 11 8 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7539007459147157 11 8 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7539007459147157 11 8 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7539007459147157 11 8 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7539007459147157 11 8 1.0 max of 0.7548211636723257 at 6 3 0.3
0.753487105099445 11 9 0.3 max of 0.7548211636723257 at 6 3 0.3
0.753487105099445 11 9 0.4 max of 0.7548211636723257 at 6 3 0.3
0.753487105099445 11 9 0.5 max of 0.7548211636723257 at 6 3 0.3
0.753487105099445 11 9 0.6 max of 0.7548211636723257 at 6 3 0.3
0.753487105099445 11 9 0.7 max of 0.7548211636723257 at 6 3 0.3
0.753487105099445 11 9 0.8 max of 0.7548211636723257 at 6 3 0.3
0.753487105099445 11 9 0.9 max of 0.7548211636723257 at 6 3 0.3
0.753487105099445 11 9 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7541440743609605 12 3 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7541440743609605 12 3 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7541440743609605 12 3 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7541440743609605 12 3 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7541440743609605 12 3 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7541440743609605 12 3 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7541440743609605 12 3 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7541440743609605 12 3 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7541408468887169 12 4 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7541408468887169 12 4 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7541408468887169 12 4 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7541408468887169 12 4 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7541408468887169 12 4 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7541408468887169 12 4 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7541408468887169 12 4 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7541408468887169 12 4 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7539231308325034 12 5 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7539231308325034 12 5 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7539231308325034 12 5 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7539231308325034 12 5 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7539231308325034 12 5 0.7 max of 0.7548211636723257 at 6 3 0.3
```

```
0.7539231308325034 12 5 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7539231308325034 12 5 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7539231308325034 12 5 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7540595145881746 12 6 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7540595145881746 12 6 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7540595145881746 12 6 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7540595145881746 12 6 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7540595145881746 12 6 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7540595145881746 12 6 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7540595145881746 12 6 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7540595145881746 12 6 1.0 max of 0.7548211636723257 at 6 3 0.3
0.753606008631183 12 7 0.3 max of 0.7548211636723257 at 6 3 0.3
0.753606008631183 12 7 0.4 max of 0.7548211636723257 at 6 3 0.3
0.753606008631183 12 7 0.5 max of 0.7548211636723257 at 6 3 0.3
0.753606008631183 12 7 0.6 max of 0.7548211636723257 at 6 3 0.3
0.753606008631183 12 7 0.7 max of 0.7548211636723257 at 6 3 0.3
0.753606008631183 12 7 0.8 max of 0.7548211636723257 at 6 3 0.3
0.753606008631183 12 7 0.9 max of 0.7548211636723257 at 6 3 0.3
0.753606008631183 12 7 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7534407005766417 12 8 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7534407005766417 12 8 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7534407005766417 12 8 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7534407005766417 12 8 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7534407005766417 12 8 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7534407005766417 12 8 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7534407005766417 12 8 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7534407005766417 12 8 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7531094082352796 12 9 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7531094082352796 12 9 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7531094082352796 12 9 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7531094082352796 12 9 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7531094082352796 12 9 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7531094082352796 12 9 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7531094082352796 12 9 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7531094082352796 12 9 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7541183294663573 13 3 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7541183294663573 13 3 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7541183294663573 13 3 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7541183294663573 13 3 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7541183294663573 13 3 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7541183294663573 13 3 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7541183294663573 13 3 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7541183294663573 13 3 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7541376643464811 13 4 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7541376643464811 13 4 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7541376643464811 13 4 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7541376643464811 13 4 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7541376643464811 13 4 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7541376643464811 13 4 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7541376643464811 13 4 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7541376643464811 13 4 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7540682122294652 13 5 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7540682122294652 13 5 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7540682122294652 13 5 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7540682122294652 13 5 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7540682122294652 13 5 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7540682122294652 13 5 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7540682122294652 13 5 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7540682122294652 13 5 1.0 max of 0.7548211636723257 at 6 3 0.3
```

```
0.7540911685633264 13 6 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7540911685633264 13 6 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7540911685633264 13 6 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7540911685633264 13 6 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7540911685633264 13 6 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7540911685633264 13 6 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7540911685633264 13 6 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7540911685633264 13 6 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7538020660700475 13 7 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7538020660700475 13 7 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7538020660700475 13 7 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7538020660700475 13 7 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7538020660700475 13 7 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7538020660700475 13 7 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7538020660700475 13 7 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7538020660700475 13 7 1.0 max of 0.7548211636723257 at 6 3 0.3
0.753655826857683 13 8 0.3 max of 0.7548211636723257 at 6 3 0.3
0.753655826857683 13 8 0.4 max of 0.7548211636723257 at 6 3 0.3
0.753655826857683 13 8 0.5 max of 0.7548211636723257 at 6 3 0.3
0.753655826857683 13 8 0.6 max of 0.7548211636723257 at 6 3 0.3
0.753655826857683 13 8 0.7 max of 0.7548211636723257 at 6 3 0.3
0.753655826857683 13 8 0.8 max of 0.7548211636723257 at 6 3 0.3
0.753655826857683 13 8 0.9 max of 0.7548211636723257 at 6 3 0.3
0.753655826857683 13 8 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7532183117217921 13 9 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7532183117217921 13 9 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7532183117217921 13 9 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7532183117217921 13 9 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7532183117217921 13 9 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7532183117217921 13 9 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7532183117217921 13 9 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7532183117217921 13 9 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7538882822065588 14 3 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7538882822065588 14 3 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7538882822065588 14 3 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7538882822065588 14 3 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7538882822065588 14 3 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7538882822065588 14 3 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7538882822065588 14 3 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7538882822065588 14 3 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7538349676987621 14 4 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7538349676987621 14 4 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7538349676987621 14 4 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7538349676987621 14 4 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7538349676987621 14 4 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7538349676987621 14 4 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7538349676987621 14 4 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7538349676987621 14 4 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7538196474379009 14 5 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7538196474379009 14 5 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7538196474379009 14 5 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7538196474379009 14 5 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7538196474379009 14 5 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7538196474379009 14 5 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7538196474379009 14 5 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7538196474379009 14 5 1.0 max of 0.7548211636723257 at 6 3 0.3
0.753837051254239 14 6 0.3 max of 0.7548211636723257 at 6 3 0.3
0.753837051254239 14 6 0.4 max of 0.7548211636723257 at 6 3 0.3
0.753837051254239 14 6 0.5 max of 0.7548211636723257 at 6 3 0.3
```

```
0.753837051254239 14 6 0.6 max of 0.7548211636723257 at 6 3 0.3
0.753837051254239 14 6 0.7 max of 0.7548211636723257 at 6 3 0.3
0.753837051254239 14 6 0.8 max of 0.7548211636723257 at 6 3 0.3
0.753837051254239 14 6 0.9 max of 0.7548211636723257 at 6 3 0.3
0.753837051254239 14 6 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7536655562541594 14 7 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7536655562541594 14 7 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7536655562541594 14 7 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7536655562541594 14 7 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7536655562541594 14 7 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7536655562541594 14 7 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7536655562541594 14 7 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7536655562541594 14 7 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7534328164221609 14 8 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7534328164221609 14 8 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7534328164221609 14 8 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7534328164221609 14 8 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7534328164221609 14 8 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7534328164221609 14 8 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7534328164221609 14 8 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7534328164221609 14 8 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7532309510940505 14 9 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7532309510940505 14 9 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7532309510940505 14 9 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7532309510940505 14 9 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7532309510940505 14 9 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7532309510940505 14 9 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7532309510940505 14 9 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7532309510940505 14 9 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7538843442918145 15 3 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7538843442918145 15 3 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7538843442918145 15 3 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7538843442918145 15 3 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7538843442918145 15 3 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7538843442918145 15 3 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7538843442918145 15 3 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7538843442918145 15 3 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7538793478734056 15 4 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7538793478734056 15 4 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7538793478734056 15 4 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7538793478734056 15 4 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7538793478734056 15 4 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7538793478734056 15 4 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7538793478734056 15 4 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7538793478734056 15 4 1.0 max of 0.7548211636723257 at 6 3 0.3
0.753861008827053 15 5 0.3 max of 0.7548211636723257 at 6 3 0.3
0.753861008827053 15 5 0.4 max of 0.7548211636723257 at 6 3 0.3
0.753861008827053 15 5 0.5 max of 0.7548211636723257 at 6 3 0.3
0.753861008827053 15 5 0.6 max of 0.7548211636723257 at 6 3 0.3
0.753861008827053 15 5 0.7 max of 0.7548211636723257 at 6 3 0.3
0.753861008827053 15 5 0.8 max of 0.7548211636723257 at 6 3 0.3
0.753861008827053 15 5 0.9 max of 0.7548211636723257 at 6 3 0.3
0.753861008827053 15 5 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7538424766141235 15 6 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7538424766141235 15 6 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7538424766141235 15 6 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7538424766141235 15 6 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7538424766141235 15 6 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7538424766141235 15 6 0.8 max of 0.7548211636723257 at 6 3 0.3
```

```
0.7538424766141235 15 6 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7538424766141235 15 6 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7536479244632509 15 7 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7536479244632509 15 7 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7536479244632509 15 7 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7536479244632509 15 7 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7536479244632509 15 7 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7536479244632509 15 7 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7536479244632509 15 7 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7536479244632509 15 7 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7535331745221389 15 8 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7535331745221389 15 8 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7535331745221389 15 8 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7535331745221389 15 8 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7535331745221389 15 8 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7535331745221389 15 8 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7535331745221389 15 8 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7535331745221389 15 8 1.0 max of 0.7548211636723257 at 6 3 0.3
0.7532694074888087 15 9 0.3 max of 0.7548211636723257 at 6 3 0.3
0.7532694074888087 15 9 0.4 max of 0.7548211636723257 at 6 3 0.3
0.7532694074888087 15 9 0.5 max of 0.7548211636723257 at 6 3 0.3
0.7532694074888087 15 9 0.6 max of 0.7548211636723257 at 6 3 0.3
0.7532694074888087 15 9 0.7 max of 0.7548211636723257 at 6 3 0.3
0.7532694074888087 15 9 0.8 max of 0.7548211636723257 at 6 3 0.3
0.7532694074888087 15 9 0.9 max of 0.7548211636723257 at 6 3 0.3
0.7532694074888087 15 9 1.0 max of 0.7548211636723257 at 6 3 0.3
Best accuracy: 0.7548211636723257 at [ 6 3 0.3 ]
Second best accuracy: 0.7535998049508074 at [ 5 5 0.3 ]
From previous runs, the best result was 0.7559 at [15 , 3, 0.3]
From previous runs, the 2nd best result was 0.7548 at [6 , 6, 0.6]
```

In [ ]:
```
print(max_acc, i_max, j_max, k_max)
# np.argmax(accuracy, axis=1, keepdims=True)
np.argmax(accuracy, axis=0)
np.argmax(accuracy, axis=1)
np.argmax(accuracy, axis=2)
```

```
0.7540380417835983 10 3 3
```

Out[ ]:
```
array([[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],

       [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],

       [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],

       [[0, 0, 0, 5, 5, 5, 5, 5, 5, 5, 5]],

       [[0, 0, 0, 4, 4, 4, 4, 4, 4, 4, 4]],

       [[0, 0, 0, 5, 5, 5, 5, 5, 5, 5, 5]],

       [[0, 0, 0, 3, 3, 3, 3, 3, 3, 3, 3]],

       [[0, 0, 0, 5, 5, 5, 5, 5, 5, 5, 5]],

       [[0, 0, 0, 4, 4, 4, 4, 4, 4, 4, 4]],

       [[0, 0, 0, 5, 5, 5, 5, 5, 5, 5, 5]],

       [[0, 0, 0, 3, 3, 0, 0, 0, 0, 0, 0]],

       [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],

       [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],

       [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],

       [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],

       [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]], dtype=int64)
```

DON'T FORGET: Once you are satisfied with your chosen parameters, change the values for `max_depth`, `n_estimators`, and `max_features` in the `select_hyperparameters()` function of your RandomForest class in `random_forest.py` to your chosen values, and then submit this file to Gradescope. You must achieve at least a **80% accuracy** against the test set in Gradescope to receive full credit for this section.

3.3 Plotting Feature Importance [5pts Bonus for All] **[W]**

While building tree-based models, it's common to quantify how well splitting on a particular feature in an extra tree helps with predicting the target label in a dataset. Machine learning practicioners typically use "Gini importance", or the (normalized) total reduction in entropy brought by that feature to evaluate how important that feature is for predicting the target variable.

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

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

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

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

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

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

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

random_forest.plot_feature_importance(data_train)
```

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

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

4: (Bonus for All) SVM [30 pts] **[W]** **[P]**

4.1 Fitting an SVM classifier by hand [20 pts] **[W]**

Consider a dataset with the following points in 2-dimensional space:

x_1	x_2	y
0	0	-1
0	2	-1
2	0	-1
3	3	1
3	4	1
2	4	1

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

The max margin classifier has the formulation,

$$\min ||\theta||^2$$

$$s.t.\ y_i(\mathbf{x_i}\theta + b) \geq 1 \quad \forall\, i$$

Hint: $\mathbf{x_i}$ are the suppport vectors. Margin is equal to $\frac{1}{||\theta||}$ and full margin is equal to $\frac{2}{||\theta||}$. You might find it useful to plot the points in a 2D plane. When calculating the θ you don't need to consider the bias term.

(1) Are the points linearly separable? Does adding the point $\mathbf{x} = (3, 2)$, $y = 1$ change the separability? (2 pts)

(2) According to the max-margin formulation, find the separating hyperplane. Do not consider the new point from part 1 in your calculations for this current question or subsequent parts. (You should give some kind of explanation or calculation on how you found the hyperplane, you may solve this question graphically.) (4 pts)

(3) Find a vector parallel to the optimal vector θ. (Hint: Recall whether the optimal vector is parallel or perpendicular to the separating hyperplane.) (4 pts)

(4) Calculate the value of the margin (single-sided) achieved by this θ? (4 pts)

(5) Solve for θ, given that the margin is equal to $1/||\theta||$. (4 pts)

(6) If we remove one of the points from the original data the SVM solution might change. Find all such points which change the solution. (2 pts)

(7) Consider the optimization formulation stated above. Why do we want to optimzie $||\theta||^2$ instead of $||\theta||$? (2 pts)

4.2 Feature Mapping [10 pts] **[P]**

Let's look at a dataset where the datapoint can't be classified with a good accuracy using a linear classifier. Run the below cell to generate the dataset.

We will also see what happens when we try to fit a linear classifier to the dataset.

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

random_state = 1

np.random.seed(0)
theta = np.linspace(0, 2*np.pi, 1000)
r = np.random.uniform(0.8, 1.2, 1000)
X = np.column_stack([r * np.cos(theta), r * np.sin(theta)])
y = np.logical_or(theta < np.pi, theta >= 2 * np.pi)
X[y == 0, 0] += 1
X[y == 0, 1] += 0.5

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.20,
                                                    random_state=random_state)


f, ax = plt.subplots(nrows=1, ncols=1,figsize=(5,5))
plt.scatter(X[:, 0], X[:, 1], c = y, marker = 'o', s=12)
plt.xlabel('X_1')
plt.ylabel('X_2')
plt.show()
```

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

def visualize_decision_boundary(X, y, feature_new=None, h=0.02):
    '''
    You don't have to modify this function

    Function to vizualize decision boundary

    feature_new is a function to get X with additional features
    '''
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx_1, xx_2 = np.meshgrid(np.arange(x1_min, x1_max, h),
                         np.arange(x2_min, x2_max, h))

    if X.shape[1] == 2:
        Z = svm_cls.predict(np.c_[xx_1.ravel(), xx_2.ravel()])
    else:
        X_conc = np.c_[xx_1.ravel(), xx_2.ravel()]
        X_new = feature_new(X_conc)
        Z = svm_cls.predict(X_new)
    Z = Z.reshape(xx_1.shape)

    f, ax = plt.subplots(nrows=1, ncols=1,figsize=(5,5))
    plt.contourf(xx_1, xx_2, Z, cmap=plt.cm.coolwarm, alpha=0.8)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm)
    plt.xlabel('X_1')
    plt.ylabel('X_2')
    plt.xlim(xx_1.min(), xx_1.max())
    plt.ylim(xx_2.min(), xx_2.max())
    plt.xticks(())
    plt.yticks(())

    plt.show()
```

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
# Try to fit a linear classifier to the dataset

svm_cls = svm.LinearSVC()
svm_cls.fit(X_train, y_train)
y_test_predicted = svm_cls.predict(X_test)

print("Accuracy on test dataset: {}".format(accuracy_score(y_test,
                                                           y_test_predicted)))

visualize_decision_boundary(X_train, y_train)
```

We can see that we need a non-linear boundary to be able to successfully classify data in this dataset. By mapping the current feature x to a higher space with more features, linear SVM could be performed on the features in the higher space to learn a non-linear decision boundary. In feature.py, modify create_nl_feature() to add additional features which can help classify in the above dataset. After creating the additional features use code in the further cells to see how well the features perform on the test set.

Note: You should get a test accuracy above 90%

Hint: Think of the shape of the decision boundary that would best separate the above points. What additional features could help map the linear boundary to the non-linear one? Look at this for a detailed analysis of doing the same for points separable with a circular boundary

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
from feature import create_nl_feature

X_new = create_nl_feature(X)
X_train, X_test, y_train, y_test = train_test_split(X_new, y,
                                                    test_size=0.20,
                                                    random_state=random_state)
```

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
# Fit to the new features and vizualize the decision boundary
# You should get more than 90% accuracy on test set

svm_cls = svm.LinearSVC()
svm_cls.fit(X_train, y_train)
y_test_predicted = svm_cls.predict(X_test)

print("Accuracy on test dataset: {}".format(accuracy_score(y_test, y_test_predicted)))

visualize_decision_boundary(X_train, y_train, create_nl_feature)
```