Fall 2022 CS 4641\7641 A: Machine Learning Homework 4

Instructor: Dr. Mahdi Roozbahani

Deadline: Friday, December 2, 2022 11:59 pm AOE

- No unapproved extension of the deadline is allowed. Late submission 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 download your Jupyter Notebook as html and submit it as a PDF on Gradescope. To download the notebook as html, click on "File" on the top left corner of this page and select "Download as > html". Then, open the html file and print to PDF.** 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 .py files in the **local_tests_folder**
- 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 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 [20pts; 17pts Bonus for Undergrad + 3pts Bonus for All]

Deliverables: cnn.py and Written Report

- **2.1 Image Classification using Keras CNN** [17pts **Bonus for Undergrad**]
  - 2.1.1 Loading the Model and Data Augmentation [5pts **Bonus for Undergrad**] - *programming*
  - 2.1.3 Building the Model [2pts **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** [3pts **Bonus for All**] - *non-programming*

Q3: Random Forest [50pts; 40pts + 10pts 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*
- 3.4 Improvement [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 [1]:
```
import sys
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_diabetes
from sklearn.preprocessing import MinMaxScaler

from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import plot_confusion_matrix
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
%matplotlib inline
```

```
Version information
python: 3.10.8 | packaged by conda-forge | (main, Nov  4 2022, 13:42:51) [MSC v.191
6 64 bit (AMD64)]
matplotlib: 3.5.3
numpy: 1.23.4
```

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

Perceptron

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

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

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

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

Fully connected Layer

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

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

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

where $u^{[l]} \in R^m$ is a m-dimensional vector pertaining to the hidden units of the l^{th} layer of the neural network after applying linear operations. Similarly, $o^{[l-1]}$ 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 nonlinearity.
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 [2]: from local_tests.local_test_nn import TestNN

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

```
test_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 [3]:
```
from local_tests.local_test_nn import TestNN

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

```
test_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

The dropout layer randomly drops nodes of a specified layer of the neural network so that the information is not propogated to the next layer. Formally, the dropout layer takes a batch of activations and for each activation it sets it to zero with a specified probability p. After this, we scale the result by $1/(1-p)$. Usually, p is passed to the neural network as a hyperparameter. Setting $p = 0$ means 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. The returned dropout mask may be helpful for the backward method.

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

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

In [4]:
```
from local_tests.local_test_nn 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 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:

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

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 [5]:
```
from local_tests.local_test_nn 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.

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

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 [6]: from local_tests.local_test_nn 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

$$\begin{aligned}\theta^{[2]} &:= \theta^{[2]} - lr \times \frac{\partial l}{\partial \theta^{[2]}} \\ b^{[2]} &:= b^{[2]} - lr \times \frac{\partial l}{\partial b^{[2]}} \\ \theta^{[1]} &:= \theta^{[1]} - lr \times \frac{\partial l}{\partial \theta^{[1]}} \\ b^{[1]} &:= b^{[1]} - lr \times \frac{\partial l}{\partial b^{[1]}}\end{aligned} \tag{1}$$

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 [7]:
```
from local_tests.local_test_nn 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:

$$\theta_{t+1} = \theta_t - \alpha \nabla f(\theta_t) \tag{2}$$

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

Momentum proposes the following tweak to our parameter update function:

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

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

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

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

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

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 [8]:
```
from local_tests.local_test_nn 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:

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

$\partial b^{[2]} \quad n \quad \partial u^{[2]}$

!!!!! 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

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

$$\frac{\partial o^{[1]}}{\partial u^{[1]}} = \begin{cases} \text{dropout\_mask} \cdot \text{scaling\_factor} \cdot [1 * (o^{[1]} > 0) \text{ and } \alpha * (o^{[1]} <= 0)] & \text{dropout=} \\ 1 * (o^{[1]} > 0) \text{ and } \alpha * (o^{[1]} <= 0) & \text{dropout=} \end{cases}$$

$$\frac{\partial u^{[1]}}{\partial \theta^{[1]}} = x$$

$$\frac{\partial u^{[1]}}{\partial b^{[1]}} = 1$$

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:

$$dLoss\_o2 = \frac{\partial l}{\partial o^{[2]}} \implies dim = (1, 331)$$

$$dLoss\_u2 = dLoss\_o2 \frac{\partial o^{[2]}}{\partial u^{[2]}} \implies dim = (1, 331)$$

$$dLoss\_theta2 = dLoss\_u2 \frac{\partial u^{[2]}}{\partial \theta^{[2]}} \implies dim = (1, 15)$$

$$dLoss\_b2 = dLoss\_u2 \frac{\partial u^{[2]}}{\partial b^{[2]}} \implies dim = (1, 1)$$

$$dLoss\_o1 = dLoss\_u2 \frac{\partial u^{[2]}}{\partial o^{[1]}} \implies dim = (15, 331)$$

$\partial o^{[1]}$

$$dLoss\_u1 = \quad dLoss\_o1\frac{}{\partial u^{[1]}} \quad \Longrightarrow \quad dim = (15, 331)$$
$$dLoss\_theta1 = \quad dLoss\_u1\frac{\partial u^{[1]}}{\partial \theta^{[1]}} \quad \Longrightarrow \quad dim = (15, 10)$$
$$dLoss\_b1 = \quad dLoss\_u1\frac{\partial u^{[1]}}{\partial b^{[1]}} \quad \Longrightarrow \quad dim = (15, 1)$$

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 331 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 [9]:
```
from local_tests.local_test_nn import TestNN

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

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

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 that would have Leaky Relu activation followed by 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**: Recall Hint 1
- **Tanh**: Recall Hint 1
- **nloss**
- **_dropout**
- **forward**
- **compute_gradients**
- **update_weights**
- **backward**: Recall 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 diabetes dataset.

1.1.1 Local Test: Helper Functions [No Points]

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

In [10]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
from local_tests.local_test_nn import TestNN

TestNN('test_leaky_relu').test_leaky_relu()
TestNN('test_tanh').test_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_dro
TestNN('test_compute_gradients').test_compute_gradients()
```

```
test_leaky_relu passed!
test_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 [No Points]

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

from NN import dlnet
from local_tests.nn_test import NN_Test

np.random.seed(0)

test_nn = NN_Test()
nn = dlnet(test_nn.x_train, test_nn.y_train, lr=0.01, batch_size=6, use_dropout=Fal

# Local test gradient descent
nn.gradient_descent(test_nn.x_train, test_nn.y_train, iter=3, local_test=True)

gd_loss_test = np.allclose(np.array(nn.loss), test_nn.gd_loss, rtol=1e-2)
print('\nYour GD losses works within the expected range:', gd_loss_test)
```

```
Loss @ Iteration 0: 15.825729795711393

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

from NN import dlnet
from local_tests.nn_test import NN_Test

np.random.seed(0)

test_nn = NN_Test()
nn = dlnet(test_nn.x_train, test_nn.y_train, lr=0.01, batch_size=6, use_dropout=Fal

# Local test batch gradient descent
nn.batch_gradient_descent(test_nn.x_train, test_nn.y_train, iter=3, local_test=True

batch_str = 'batch_y at iteration %i: '
print('\ny_train input:', test_nn.y_train)
[print(batch_str %(i), batch_y) for i, batch_y in enumerate(nn.batch_y)]

bgd_loss_test = np.allclose(np.array(nn.loss), test_nn.bgd_loss, rtol=1e-2)
print('\nYour BGD losses works within the expected range:', bgd_loss_test)
batch_y_test = np.allclose(np.array(nn.batch_y), test_nn.batch_y, rtol=1e-2)
print('Your batch_y works within the expected range:', batch_y_test)

print(nn.loss)
print(test_nn.bgd_loss)
```

```
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
[7.5828152010012335, 17.072581859100293, 21.846386861623053]
[7.573928445695169, 17.12996447900625, 21.86319152295093]
```

1.1.4 Local Test: Gradient Descent with Momentum [No Points - Bonus for Undergrad]

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 [13]:
```
###############################
### DO NOT CHANGE THIS CELL ###
###############################

from NN import dlnet
from local_tests.nn_test import NN_Test

np.random.seed(0)

test_nn = NN_Test()
nn = dlnet(test_nn.x_train, test_nn.y_train)

# Local test batch gradient descent
nn.gradient_descent(test_nn.x_train, test_nn.y_train, iter=3, use_momentum=True, lo

gd_loss_test_with_momentum = np.allclose(np.array(nn.loss), test_nn.gd_loss_with_mo
print('\nYour GD losses works within the expected range:', gd_loss_test_with_moment
```

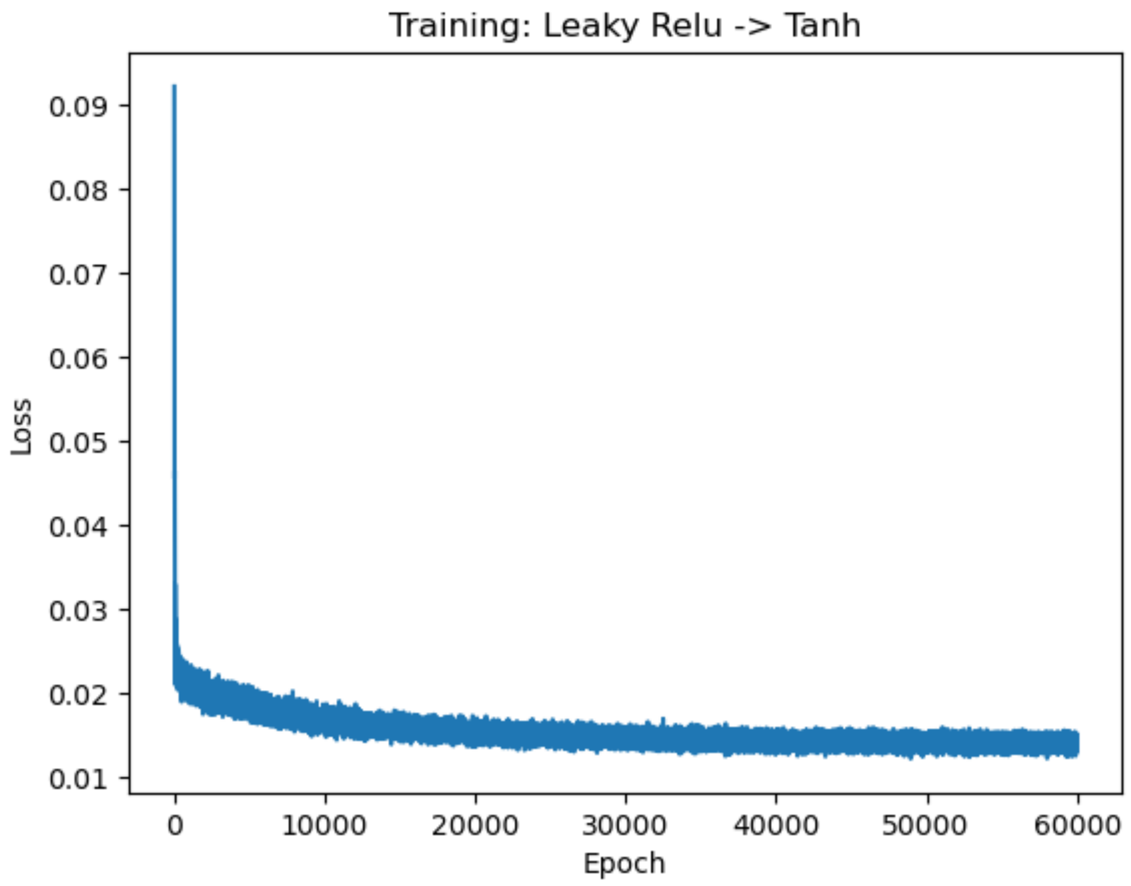
```
Loss @ Iteration 0: 15.825729795711393

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

from NN import dlnet

dataset = load_diabetes() # 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 da


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_te

nn = dlnet(x_train,y_train,lr=0.01,use_dropout=True) # initalize neural net class
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 @ Iteration 0: 0.09214244110372626
Loss @ Iteration 1000: 0.02092549010211555
Loss @ Iteration 2000: 0.01857182141765113
Loss @ Iteration 3000: 0.02012889160816321
Loss @ Iteration 4000: 0.01909246102810032
Loss @ Iteration 5000: 0.017540358827724638
Loss @ Iteration 6000: 0.01929102366358697
Loss @ Iteration 7000: 0.01808499940325032
Loss @ Iteration 8000: 0.016862983558407253
Loss @ Iteration 9000: 0.016855303469317226
Loss @ Iteration 10000: 0.0174738836314212
Loss @ Iteration 11000: 0.016024668199721682
Loss @ Iteration 12000: 0.015886306173312548
Loss @ Iteration 13000: 0.015299815370436343
Loss @ Iteration 14000: 0.016080412681819166
Loss @ Iteration 15000: 0.016456110585323556
Loss @ Iteration 16000: 0.015502216612250767
Loss @ Iteration 17000: 0.015264993228348834
Loss @ Iteration 18000: 0.015552613447406799
Loss @ Iteration 19000: 0.015076963766242953
Loss @ Iteration 20000: 0.014955195404057842
Loss @ Iteration 21000: 0.014816662182044306
Loss @ Iteration 22000: 0.014434217928686459
Loss @ Iteration 23000: 0.014414633228135137
Loss @ Iteration 24000: 0.01492197181197708
Loss @ Iteration 25000: 0.014463317827054194
Loss @ Iteration 26000: 0.01452263032789862
Loss @ Iteration 27000: 0.015501644172845603
Loss @ Iteration 28000: 0.015293427613163742
Loss @ Iteration 29000: 0.0139143869232449
Loss @ Iteration 30000: 0.014072110381700592
Loss @ Iteration 31000: 0.014610569457867918
Loss @ Iteration 32000: 0.01524556628899365
Loss @ Iteration 33000: 0.014826169251328899
Loss @ Iteration 34000: 0.014676577911274281
Loss @ Iteration 35000: 0.013793545400839278
Loss @ Iteration 36000: 0.014692060012758723
Loss @ Iteration 37000: 0.013247985063645433
Loss @ Iteration 38000: 0.01459572755602545
Loss @ Iteration 39000: 0.014283803568755025
Loss @ Iteration 40000: 0.013408031149854848
Loss @ Iteration 41000: 0.01433515781245455
Loss @ Iteration 42000: 0.014136085762589063
Loss @ Iteration 43000: 0.013875992456857349
Loss @ Iteration 44000: 0.01452814305953369
Loss @ Iteration 45000: 0.014634486263366615
Loss @ Iteration 46000: 0.014584144569420749
Loss @ Iteration 47000: 0.0145844190141403
Loss @ Iteration 48000: 0.013913338149231144
Loss @ Iteration 49000: 0.014655166898068818
Loss @ Iteration 50000: 0.013640429777679852
Loss @ Iteration 51000: 0.01396226465828882
Loss @ Iteration 52000: 0.013625931673359312
Loss @ Iteration 53000: 0.014062393354827443
Loss @ Iteration 54000: 0.013522579810177448
Loss @ Iteration 55000: 0.013060826959819633
Loss @ Iteration 56000: 0.013882072743058003
Loss @ Iteration 57000: 0.013865494937748291
Loss @ Iteration 58000: 0.013995741272743036
Loss @ Iteration 59000: 0.013904943998040677
```



```
In [15]: ################################
         ### 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))
```

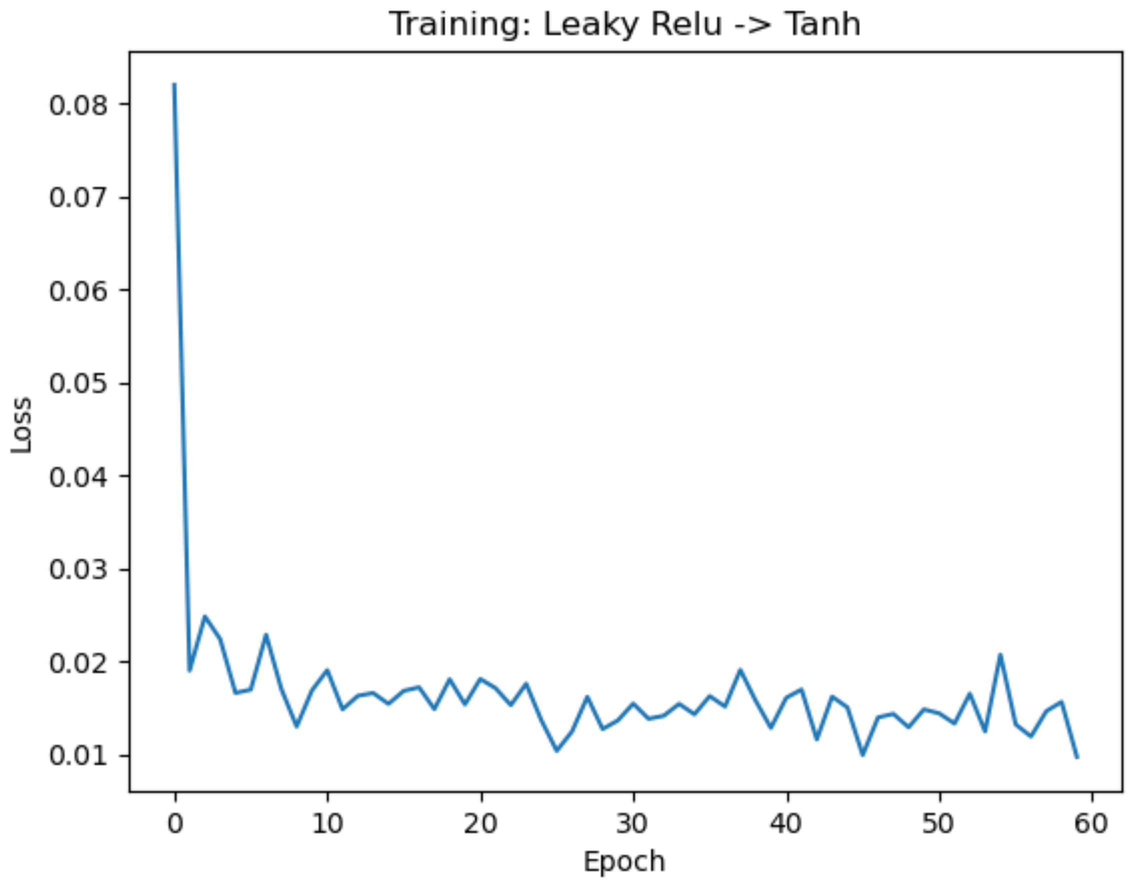
```
Mean Squared Error (MSE) 0.03574491460289309
```

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

from NN import dlnet

dataset = load_diabetes() # 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 da

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_te

nn = dlnet(x_train,y_train,lr=0.01,use_dropout=True) # initalize neural net class
nn.batch_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 @ Iteration 0: 0.08196159124660582
Loss @ Iteration 0: 0.08196159124660582
Loss @ Iteration 1000: 0.018995260742341077
Loss @ Iteration 1000: 0.018995260742341077
Loss @ Iteration 2000: 0.02481663682064045
Loss @ Iteration 2000: 0.02481663682064045
Loss @ Iteration 3000: 0.022373645195696734
Loss @ Iteration 3000: 0.022373645195696734
Loss @ Iteration 4000: 0.016593663515483408
Loss @ Iteration 4000: 0.016593663515483408
Loss @ Iteration 5000: 0.016951125112706675
Loss @ Iteration 5000: 0.016951125112706675
Loss @ Iteration 6000: 0.022850644195473575
Loss @ Iteration 6000: 0.022850644195473575
Loss @ Iteration 7000: 0.017004891492986764
Loss @ Iteration 7000: 0.017004891492986764
Loss @ Iteration 8000: 0.012981456221305946
Loss @ Iteration 8000: 0.012981456221305946
Loss @ Iteration 9000: 0.016868616048378967
Loss @ Iteration 9000: 0.016868616048378967
Loss @ Iteration 10000: 0.019041000324734405
Loss @ Iteration 10000: 0.019041000324734405
Loss @ Iteration 11000: 0.01483111018322387
Loss @ Iteration 11000: 0.01483111018322387
Loss @ Iteration 12000: 0.016282847547250436
Loss @ Iteration 12000: 0.016282847547250436
Loss @ Iteration 13000: 0.016585113694218466
Loss @ Iteration 13000: 0.016585113694218466
Loss @ Iteration 14000: 0.015416242208559389
Loss @ Iteration 14000: 0.015416242208559389
Loss @ Iteration 15000: 0.016815561519564867
Loss @ Iteration 15000: 0.016815561519564867
Loss @ Iteration 16000: 0.017200506751022204
Loss @ Iteration 16000: 0.017200506751022204
Loss @ Iteration 17000: 0.014854624406993507
Loss @ Iteration 17000: 0.014854624406993507
Loss @ Iteration 18000: 0.018062543085091712
Loss @ Iteration 18000: 0.018062543085091712
Loss @ Iteration 19000: 0.015361537342032662
Loss @ Iteration 19000: 0.015361537342032662
Loss @ Iteration 20000: 0.018091240567104365
Loss @ Iteration 20000: 0.018091240567104365
Loss @ Iteration 21000: 0.01710677503367733
Loss @ Iteration 21000: 0.01710677503367733
Loss @ Iteration 22000: 0.015281109341825302
Loss @ Iteration 22000: 0.015281109341825302
Loss @ Iteration 23000: 0.01757992312098966
Loss @ Iteration 23000: 0.01757992312098966
Loss @ Iteration 24000: 0.01362669191806784
Loss @ Iteration 24000: 0.01362669191806784
Loss @ Iteration 25000: 0.010368050221862286
Loss @ Iteration 25000: 0.010368050221862286
Loss @ Iteration 26000: 0.012455015776863466
Loss @ Iteration 26000: 0.012455015776863466
Loss @ Iteration 27000: 0.016173774498345454
Loss @ Iteration 27000: 0.016173774498345454
Loss @ Iteration 28000: 0.012709400372432782
Loss @ Iteration 28000: 0.012709400372432782
Loss @ Iteration 29000: 0.013641219436992265
Loss @ Iteration 29000: 0.013641219436992265
Loss @ Iteration 30000: 0.015447892418995305
Loss @ Iteration 30000: 0.015447892418995305
Loss @ Iteration 31000: 0.01381576730340833
Loss @ Iteration 31000: 0.01381576730340833
Loss @ Iteration 32000: 0.01414034718411921
Loss @ Iteration 32000: 0.01414034718411921
Loss @ Iteration 33000: 0.015417384727304282
Loss @ Iteration 33000: 0.015417384727304282
Loss @ Iteration 34000: 0.014294448682855487
Loss @ Iteration 34000: 0.014294448682855487
Loss @ Iteration 35000: 0.016253461406481755
Loss @ Iteration 35000: 0.016253461406481755
Loss @ Iteration 36000: 0.015124978915789246
Loss @ Iteration 36000: 0.015124978915789246
Loss @ Iteration 37000: 0.019071043820276305
Loss @ Iteration 37000: 0.019071043820276305
Loss @ Iteration 38000: 0.015759758741596363
Loss @ Iteration 38000: 0.015759758741596363
Loss @ Iteration 39000: 0.012832447115654875
Loss @ Iteration 39000: 0.012832447115654875
Loss @ Iteration 40000: 0.01606738772479608
Loss @ Iteration 40000: 0.01606738772479608
Loss @ Iteration 41000: 0.016963908499623138
Loss @ Iteration 41000: 0.016963908499623138
Loss @ Iteration 42000: 0.011590649550016742
Loss @ Iteration 42000: 0.011590649550016742
Loss @ Iteration 43000: 0.016199410774873663
Loss @ Iteration 43000: 0.016199410774873663
Loss @ Iteration 44000: 0.015035936356034927
Loss @ Iteration 44000: 0.015035936356034927
Loss @ Iteration 45000: 0.00991647704328322
Loss @ Iteration 45000: 0.00991647704328322
Loss @ Iteration 46000: 0.013957381070692432
Loss @ Iteration 46000: 0.013957381070692432
Loss @ Iteration 47000: 0.014342296687368028
Loss @ Iteration 47000: 0.014342296687368028
Loss @ Iteration 48000: 0.012896992422363704
Loss @ Iteration 48000: 0.012896992422363704
Loss @ Iteration 49000: 0.014822851397797931
Loss @ Iteration 49000: 0.014822851397797931
Loss @ Iteration 50000: 0.01439609058180117
Loss @ Iteration 50000: 0.01439609058180117
Loss @ Iteration 51000: 0.013299192379217863
Loss @ Iteration 51000: 0.013299192379217863
Loss @ Iteration 52000: 0.016512389455764637
Loss @ Iteration 52000: 0.016512389455764637
Loss @ Iteration 53000: 0.012452372922730484
Loss @ Iteration 53000: 0.012452372922730484
Loss @ Iteration 54000: 0.020715806210451843
Loss @ Iteration 54000: 0.020715806210451843
Loss @ Iteration 55000: 0.013207187959891174
Loss @ Iteration 55000: 0.013207187959891174
Loss @ Iteration 56000: 0.011908882007207092
Loss @ Iteration 56000: 0.011908882007207092
Loss @ Iteration 57000: 0.014631865984476971
Loss @ Iteration 57000: 0.014631865984476971
Loss @ Iteration 58000: 0.015632772804392077
Loss @ Iteration 58000: 0.015632772804392077
Loss @ Iteration 59000: 0.009723813385549637
Loss @ Iteration 59000: 0.009723813385549637
```



In [17]:

```
################################
### 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))
```

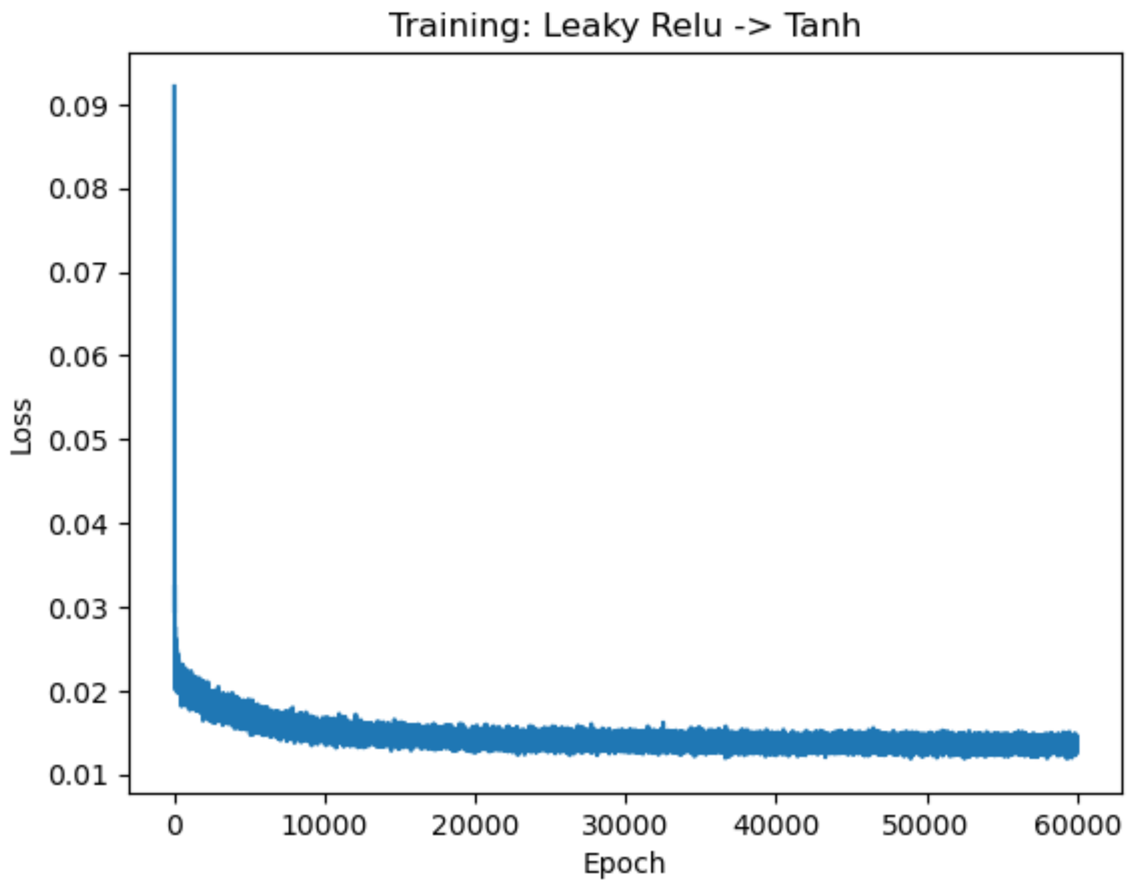
```
Mean Squared Error (MSE) 0.035769090251426924
```

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

from NN import dlnet

dataset = load_diabetes() # 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 da


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_te

nn = dlnet(x_train,y_train,lr=0.01,use_dropout=True) # initalize neural net class
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 @ Iteration 0: 0.09214244110372626
Loss @ Iteration 1000: 0.02012290643203884
Loss @ Iteration 2000: 0.017492902608867947
Loss @ Iteration 3000: 0.018674213264374875
Loss @ Iteration 4000: 0.017391513479854844
Loss @ Iteration 5000: 0.01581500251185056
Loss @ Iteration 6000: 0.017600402645329607
Loss @ Iteration 7000: 0.016483962820533243
Loss @ Iteration 8000: 0.015311016327625241
Loss @ Iteration 9000: 0.01519971206090329
Loss @ Iteration 10000: 0.0157188348056399
Loss @ Iteration 11000: 0.014635885921616418
Loss @ Iteration 12000: 0.014753790250080454
Loss @ Iteration 13000: 0.014186431966936896
Loss @ Iteration 14000: 0.01502276816850926
Loss @ Iteration 15000: 0.015007216338809434
Loss @ Iteration 16000: 0.014303563809573155
Loss @ Iteration 17000: 0.014212155761941072
Loss @ Iteration 18000: 0.014448660555639668
Loss @ Iteration 19000: 0.01409800440011177
Loss @ Iteration 20000: 0.013999668346550049
Loss @ Iteration 21000: 0.01390315969876507
Loss @ Iteration 22000: 0.013743979616114352
Loss @ Iteration 23000: 0.013579541181997983
Loss @ Iteration 24000: 0.014230961493179813
Loss @ Iteration 25000: 0.013708853393951646
Loss @ Iteration 26000: 0.013725415735483783
Loss @ Iteration 27000: 0.014702256098086768
Loss @ Iteration 28000: 0.014561235251171195
Loss @ Iteration 29000: 0.013297766432599526
Loss @ Iteration 30000: 0.013274311015964584
Loss @ Iteration 31000: 0.01396729614454
Loss @ Iteration 32000: 0.014458113252931443
Loss @ Iteration 33000: 0.01421865463352451
Loss @ Iteration 34000: 0.014096620459873472
Loss @ Iteration 35000: 0.013211759236021636
Loss @ Iteration 36000: 0.013951769365349065
Loss @ Iteration 37000: 0.012708085838327922
Loss @ Iteration 38000: 0.014128905375962581
Loss @ Iteration 39000: 0.013757269605425719
Loss @ Iteration 40000: 0.012967823225590555
Loss @ Iteration 41000: 0.013842650180136161
Loss @ Iteration 42000: 0.013840133383111271
Loss @ Iteration 43000: 0.013557180952672339
Loss @ Iteration 44000: 0.014001294572099685
Loss @ Iteration 45000: 0.014073069775567845
Loss @ Iteration 46000: 0.014172362784768317
Loss @ Iteration 47000: 0.014121504889490894
Loss @ Iteration 48000: 0.013557691285704745
Loss @ Iteration 49000: 0.014231807332293029
Loss @ Iteration 50000: 0.013243445703400024
Loss @ Iteration 51000: 0.013493876623888235
Loss @ Iteration 52000: 0.013144330349976287
Loss @ Iteration 53000: 0.013754673760029461
Loss @ Iteration 54000: 0.013262752892783131
Loss @ Iteration 55000: 0.012670912395208733
Loss @ Iteration 56000: 0.013597873019220576
Loss @ Iteration 57000: 0.013587340481449177
Loss @ Iteration 58000: 0.013467598013333656
Loss @ Iteration 59000: 0.013360437393285825
```



In [19]:

```
###############################
### 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) 0.03582861614574747
```

2: Image Classification based on Convolutional Neural Networks [20pts; 17pts Bonus for Undergrad + 3pts Bonus for all] **[P]****[W]**

2.1 Image Classification using Keras and CNN

Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result as fast as possible is key to doing good research. In this part, you will build a convolutional neural network based on TF/Keras to solve the image classification task for the CIFAR-10 dataset. If you haven't installed TensorFlow, you can install the package by **pip** command or train your model by uploading HW4 notebook to Colab directly. Colab contains all packages you need for this section.

Hint1: First contact with Keras

Hint2: How to Install Keras

Hint3: CS231n Tutorial (Layers used to build ConvNets)

Environment Setup

In [20]:
```
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, Activatio
from tensorflow.keras.layers import LeakyReLU
from tensorflow.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. You will learn how to apply data augmentation using the Keras preprocessing layers

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_CROP: We randomly crop the image to a given size.

It would definitely be useful to explore the Tensorflow documentation to see the different image processing techniques we can use for our computer vision projects.

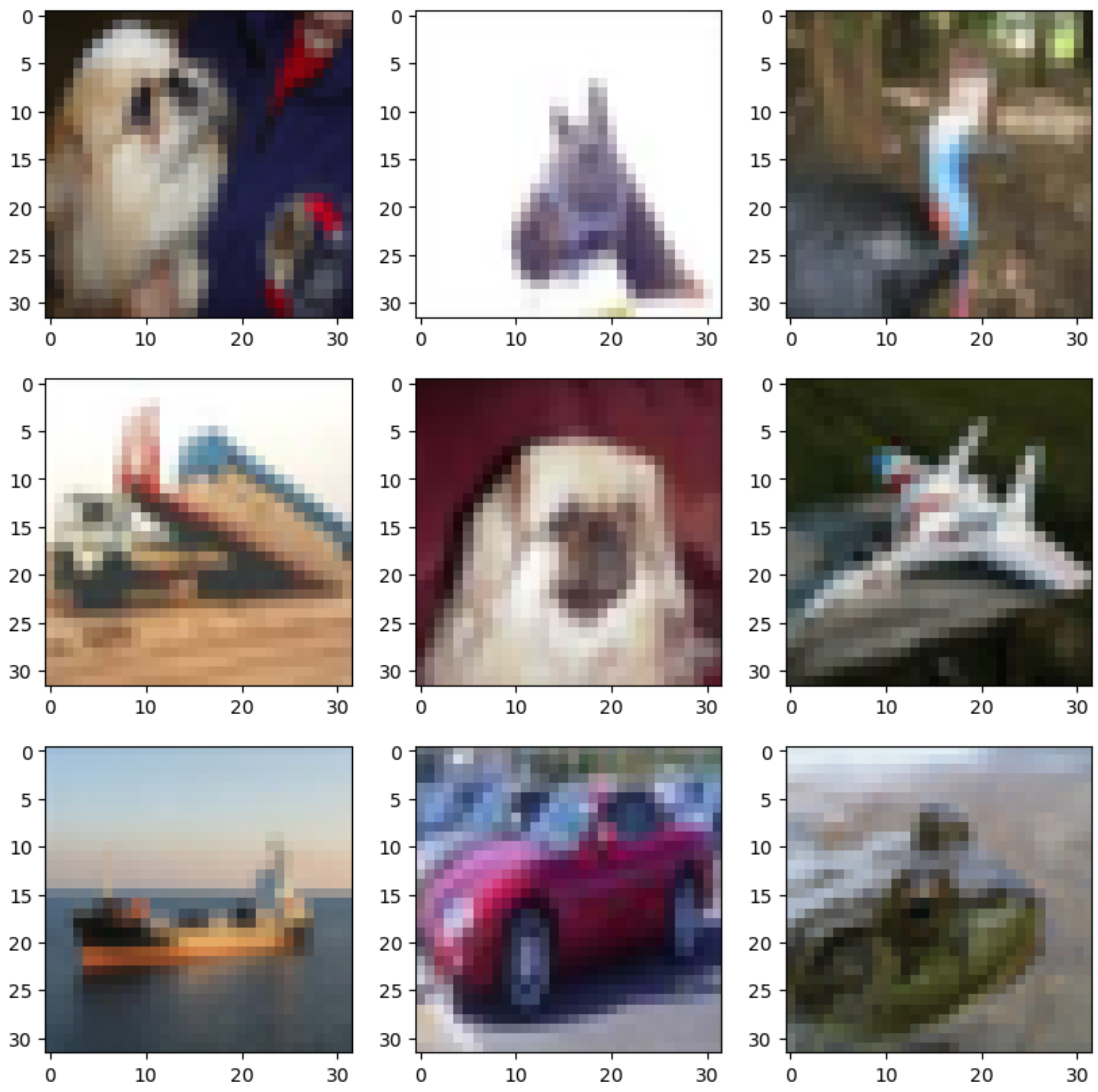
In [21]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
from image_preprocessing import data_augmentation,data_preprocessing
# split data between train and test sets

(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
```

```
class_names=['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse
# 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 [22]:
```
###############################
### 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 [2pts Bonus for Undergrad] **[W]**

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

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

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

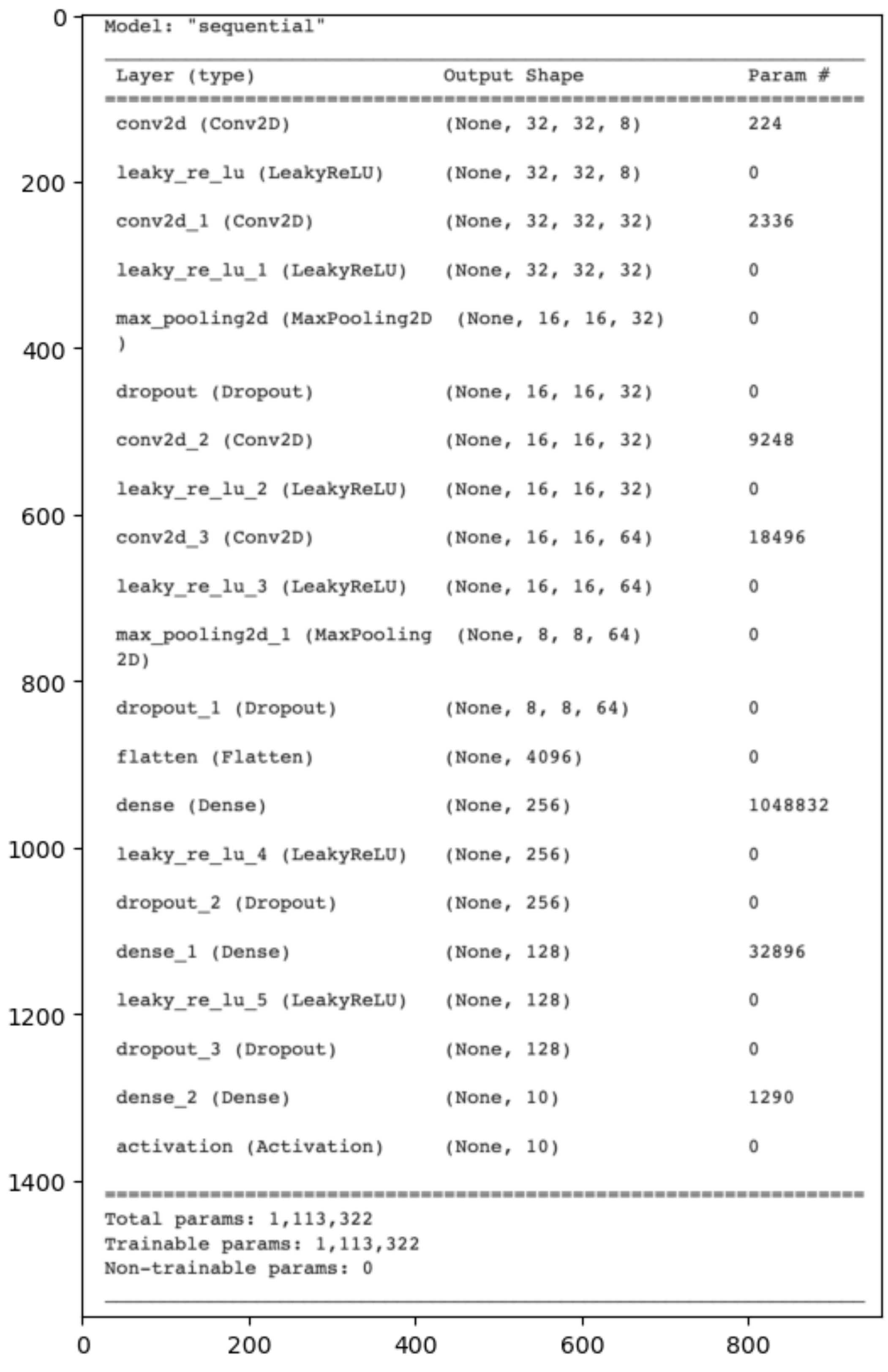
Use the following keras links to reference crucial layers of the model in keras API:

- Conv2d: https://keras.io/api/layers/convolution_layers/convolution2d/
- Dense: https://keras.io/api/layers/core_layers/dense/
- Flatten: https://keras.io/api/layers/reshaping_layers/flatten/
- MaxPool: https://keras.io/api/layers/pooling_layers/max_pooling2d/
- Dropout: https://keras.io/api/layers/regularization_layers/dropout/
- LeakyReLU: https://keras.io/api/layers/activation_layers/leaky_relu/

And explore the keras layers API if you would like to experiment with additional layers: https://keras.io/api/layers/

In [23]:
```
###############################
### 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[23]:
```
<matplotlib.image.AxesImage at 0x1d2a49f8fd0>
```



```
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 [24]:
```
###############################
### 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_
```

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

Defining model [2pts 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 2pts.

In [25]:
```
###############################
### 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   
                                                                 
 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      
                                                                 
 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 [26]:
```
################################
### 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 0x000001D26418FA00>
```

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.

- 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 [27]:
```
################################
### 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(score)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

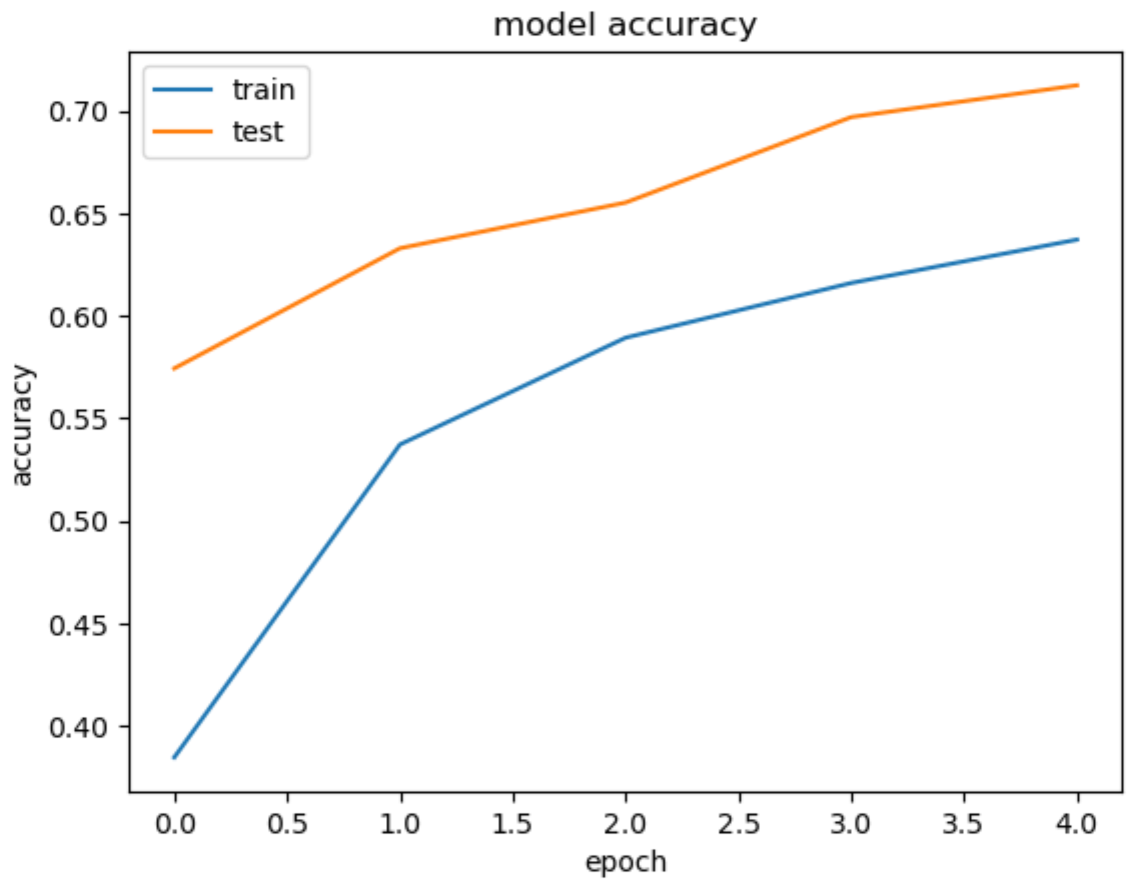
```
Learning rate: 0.001
Epoch 1/5
938/938 [==============================] - 82s 86ms/step - loss: 1.7141 - accuracy:
0.3845 - val_loss: 1.2189 - val_accuracy: 0.5743 - lr: 0.0010
Learning rate: 0.0009000000000000001
Epoch 2/5
938/938 [==============================] - 83s 88ms/step - loss: 1.3283 - accuracy:
0.5372 - val_loss: 1.0789 - val_accuracy: 0.6329 - lr: 9.0000e-04
Learning rate: 0.0008100000000000001
Epoch 3/5
938/938 [==============================] - 88s 94ms/step - loss: 1.1943 - accuracy:
0.5893 - val_loss: 1.0045 - val_accuracy: 0.6552 - lr: 8.1000e-04
Learning rate: 0.0007290000000000002
Epoch 4/5
938/938 [==============================] - 80s 85ms/step - loss: 1.1264 - accuracy:
0.6160 - val_loss: 0.8939 - val_accuracy: 0.6969 - lr: 7.2900e-04
Learning rate: 0.0006561000000000001
Epoch 5/5
938/938 [==============================] - 83s 89ms/step - loss: 1.0742 - accuracy:
0.6371 - val_loss: 0.8623 - val_accuracy: 0.7124 - lr: 6.5610e-04
[0.862256646156311, 0.7124000191688538]
Test loss: 0.862256646156311
Test accuracy: 0.7124000191688538
```

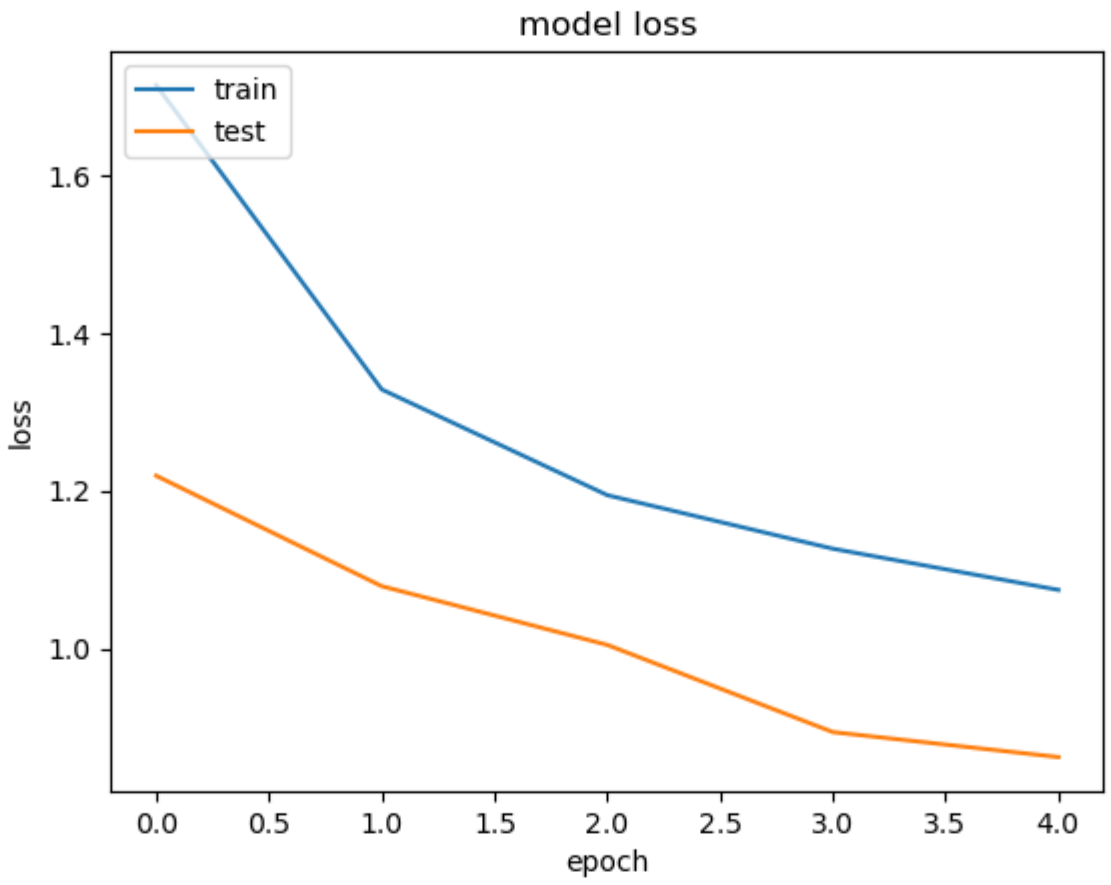
2.1.5 Examine accuracy and loss [2pts Bonus for Undergrad] **[W]**

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.

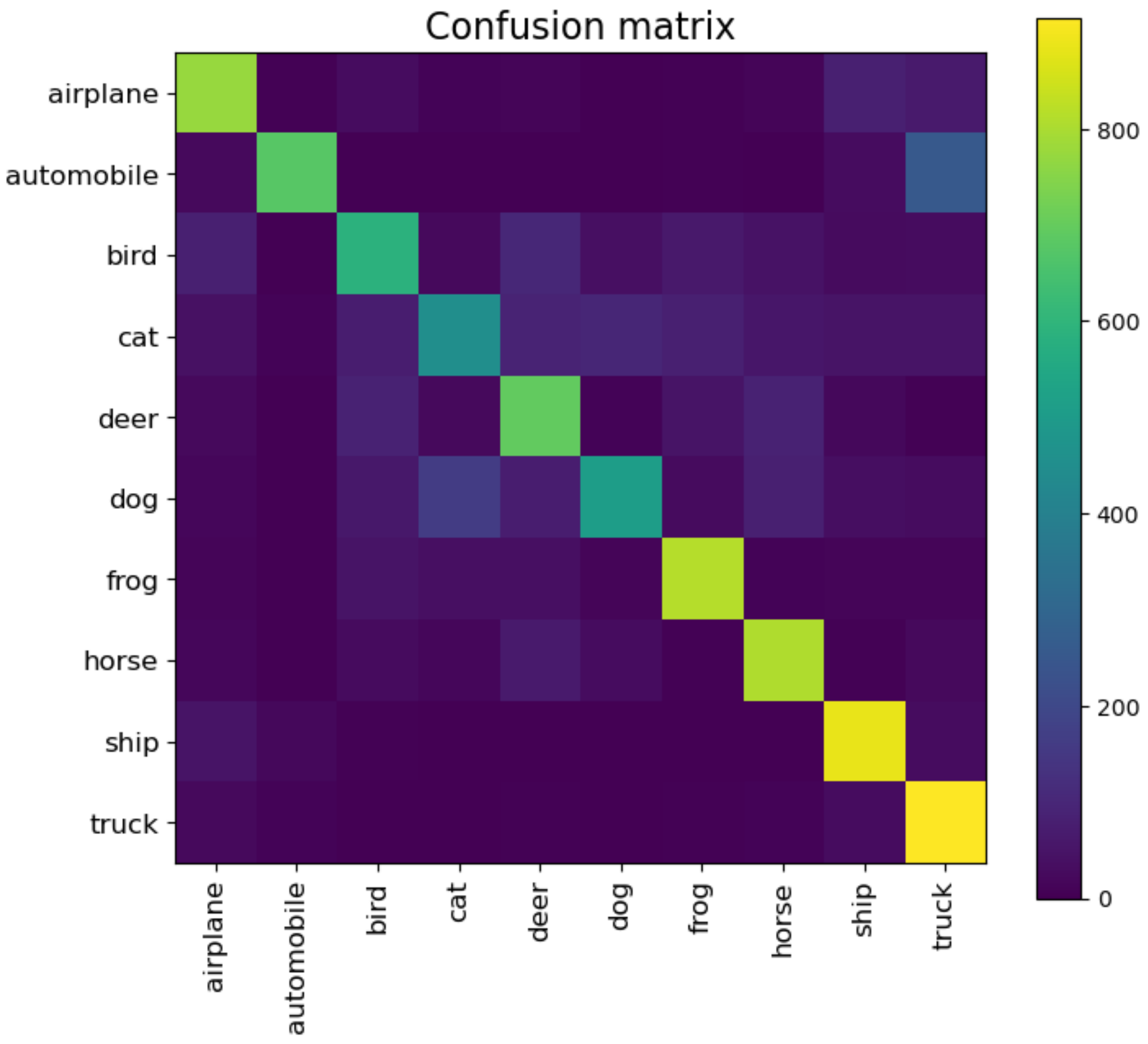
In [28]:
```
###############################
### DO NOT CHANGE THIS CELL ###
###############################

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

# summarize history for accuracy and loss
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_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', 'accuracy', 'val_loss', 'val_accuracy', 'lr'])
```


model accuracy
train
test
accuracy
0.40
0.45
0.50
0.55
0.60
0.65
0.70
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
epoch


model loss
train
test
loss
1.0
1.2
1.4
1.6
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
epoch

In [29]:
```
###############################
### 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()
```



2.2 Exploring Deep CNN Architectures [3pts 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 [50pts; 40pts + 10pts 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.

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 herbologists and mycologists working to document the various properties of mushrooms for a machine learning model so that an amateur can safely determine whether or not a mushroom is usable in the kitchen for cooking. We know that mushrooms exhibit a variety of properties that we can track and associate with their edibility. We are tasked with the responsibility of coming up with a method for determining the likelihood of any given mushroom, exhibiting a series of traits, is edible or poisonous. We will then use this information to document what traits a person should look for when inspecting a mushroom for consumption.

After much deliberation amongst the team, you come to a conclusion that we can use the past observations of other mushroom specimens to create a model.

We will use our random forest algorithm from Q3.1 to predict if a mushroom is edible or not.

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 22 features used in this dataset, with the values converted from letter values to numerical ones (a==>1, b==>2, ...,z==>26).The features observed

Inputs:

1. cap-shape: bell=b,conical=c,convex=x,flat=f, knobbed=k,sunken=s
2. cap-surface: fibrous=f,grooves=g,scaly=y,smooth=s
3. cap-color: brown=n,buff=b,cinnamon=c,gray=g,green=r, pink=p,purple=u,red=e,white=w,yellow=y
4. bruises: bruises=t,no=f
5. odor: almond=a,anise=l,creosote=c,fishy=y,foul=f, musty=m,none=n,pungent=p,spicy=s
6. gill-attachment: attached=a,descending=d,free=f,notched=n
7. gill-spacing: close=c,crowded=w,distant=d
8. gill-size: broad=b,narrow=n
9. gill-color: black=k,brown=n,buff=b,chocolate=h,gray=g, green=r,orange=o,pink=p,purple=u,red=e, white=w,yellow=y
10. stalk-shape: enlarging=e,tapering=t
11. stalk-root: bulbous=b,club=c,cup=u,equal=e, rhizomorphs=z,rooted=r,missing=?
12. stalk-surface-above-ring: fibrous=f,scaly=y,silky=k,smooth=s
13. stalk-surface-below-ring: fibrous=f,scaly=y,silky=k,smooth=s
14. stalk-color-above-ring: brown=n,buff=b,cinnamon=c,gray=g,orange=o, pink=p,red=e,white=w,yellow=y
15. stalk-color-below-ring: brown=n,buff=b,cinnamon=c,gray=g,orange=o, pink=p,red=e,white=w,yellow=y
16. veil-type: partial=p,universal=u
17. veil-color: brown=n,orange=o,white=w,yellow=y
18. ring-number: none=n,one=o,two=t
19. ring-type: cobwebby=c,evanescent=e,flaring=f,large=l, none=n,pendant=p,sheathing=s,zone=z
20. spore-print-color: black=k,brown=n,buff=b,chocolate=h,green=r, orange=o,purple=u,white=w,yellow=y
21. population: abundant=a,clustered=c,numerous=n, scattered=s,several=v,solitary=y
22. habitat: grasses=g,leaves=l,meadows=m,paths=p, urban=u,waste=w,woods=d

Output:

1. (edibility) target value:
    - p (aka 16) means the mushroom is poisonous
    - e (aka 5) means the mushroom is edible

Your random forest model will try to predict this variable.

In [30]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
from sklearn import preprocessing
import pandas as pd
preprocessor = preprocessing.LabelEncoder()

data_train = pd.read_csv("./data/mushroom_train_clean.csv")
data_test = pd.read_csv("./data/mushroom_test_clean.csv")

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

In [31]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
assert X_train.shape == (7124, 22)
assert y_train.shape == (7124,)
assert X_test.shape == (1000, 22)
assert y_test.shape == (1000,)
```

```
(7124, 22) (7124,) (1000, 22) (1000,)
```

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 95%+).

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 **95% accuracy** against the test set in Gradescope to receive full credit for this section.

In [32]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
from random_forest import RandomForest

'''
Once you have implemented Random forest, you can run this cell. If you implemented
then this cell should execute without any errors.
'''
test_seed = 1
num_feats = 40
max_feats = 0.65
rf_test = RandomForest(4, 5, max_feats)

row_idx, col_idx = rf_test._bootstrapping(15, num_feats, test_seed)
assert np.array_equal(row_idx, np.array([5, 11, 12,  8,  9, 11,  5,  0,  0,  1, 12,
assert np.array_equal(col_idx,
                      np.array([30,  2, 16, 32, 31,  5, 34,  6, 15, 19, 10,  3, 21,
                                8, 39, 12, 24, 1,  7, 35, 26, 13, 22,  0, 27, 17]))
```

In [33]:
```
"""
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 tr

Tune these three parameters to achieve a better accuracy. n_estimators and max_dept
be at least 3 in value for moderately reliable answers. While you can use the provi
to evaluate your implementation, you will need to obtain 95% on the test set to rec
credit for this section.
"""
from random_forest import RandomForest

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

################# CHANGE THESE VALUES ###############################
n_estimators = 12 #Hint: Consider values between 3-12.
max_depth = 12 # Hint: Consider values betweeen 3-12.
max_features = 1.0 # Hint: Consider values between 0.3-1.0.
#####################################################################
random_forest = RandomForest(n_estimators, max_depth, max_features, random_seed=stu
random_forest.fit(X_train, y_train)
accuracy=random_forest.OOB_score(X_test, y_test)
print("accuracy: %.4f" % accuracy)
```

```
accuracy: 0.9923
```

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 **95% 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 "Cap Color" in an extra tree trained on our mushroom dataset (many samples will reach this split for a decision) and a large reduction in entropy from splitting on "Cap Color" will result in a high feature importance value for "Cap Color". This could mean "Cap Color" is a very important feature for predicting the probability of a mushroom being edible. On the other hand, a low probability of reaching a split on "population" category in an extra tree and a low reduction in entropy from splitting on "population" will result in a low feature importance value. This could mean "population" is not a very informative feature for predicting a mushroom's probability of edibility in our extra tree. **Thus, the higher the feature importance value, the more important the feature is to predicting the target label.**

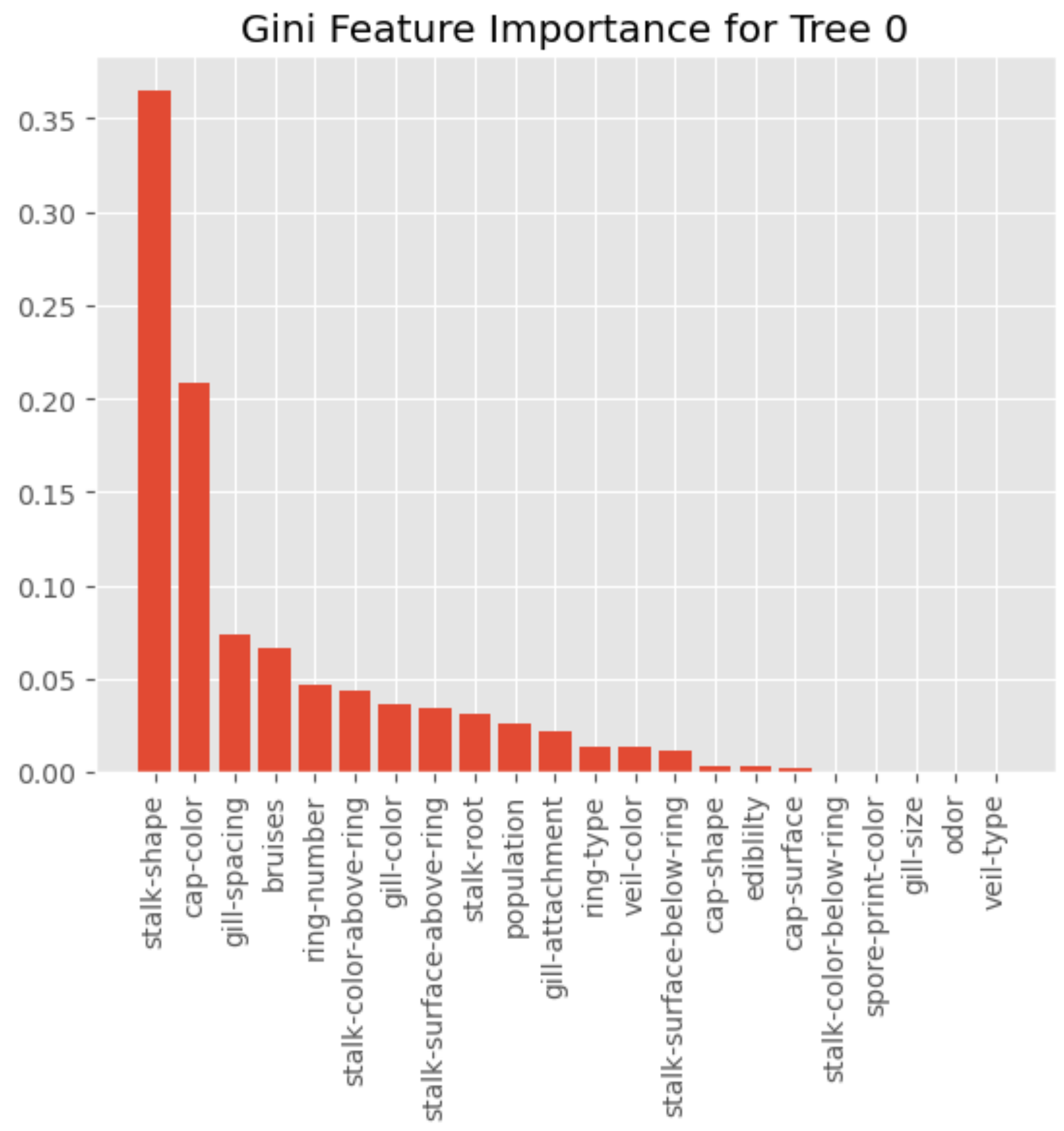
Fortunately for us, fitting a sklearn.ExtraTreeClassifier to a dataset auomatically computes the Gini importance for every feature in the extra tree and stores these values in a **feature_importances_** variable. Review the docs for more details on how to access this variable

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 [34]:
```python
# TODO: Complete plot_feature_importance() in random_forest.py

random_forest.plot_feature_importance(data_train)
```



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

3.4 Improvement [5 pts Bonus for All] **[W]**

Please answer the following two questions:

1. In case we do not get a high accuracy, what are some potential causes for why a Random Forest extra tree model may not produce high accuracies?
2. What are some ways to improve on these potential causes?

Answers:

1.
2.

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	4	-1
4	0	-1
4	4	1
8	0	1
8	8	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 $\mathbf{x} = (8, 4)$, $y = -1$ 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.) (4 pts)

(3) Find a vector parallel to the optimal vector θ. (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)

Answers:

(1)

(2)

(3)

(4)

(5)

(6)

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

X_1 = np.random.uniform(size=(100, 2))
y_1 = np.zeros((100,)) - 1

X_2 = np.random.uniform(size=(100, 2))
X_2[:, 0] = X_2[:, 0] + 1.0
y_2 = np.ones((100,))

X_3 = np.random.uniform(size=(100, 2))
X_3[:, 1] = X_3[:, 1] + 1.0
y_3 = np.ones((100,))

X_4 = np.random.uniform(size=(100, 2))
X_4[:, 0] = X_4[:, 0] + 1.0
X_4[:, 1] = X_4[:, 1] + 1.0
y_4 = np.zeros((100,)) - 1

X = np.concatenate([X_1, X_2, X_3, X_4], axis=0)
y = np.concatenate([y_1, y_2, y_3, y_4], axis=0)
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 = '.')
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)
```