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 Keras to solve the image classification task for CIFAR10. If you haven't installed TensorFlow, you can install the package by pip command or train your model by uploading HW4 notebook to Colab (https://colab.research.google.com/) directly. Colab contains all packages you need for this section.

Hint1: First contact with Keras (https://keras.io/about/)

Hint2: How to Install Keras (https://www.pyimagesearch.com/2016/07/18/installing-keras-for-deep-learning/)

Hint3： CS231n Tutorial (Layers used to build ConvNets) (https://cs231n.github.io/convolutional-networks/)

Environment Setup

In [5]:
```
from __future__ import print_function
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
, Activation, Dropout
from tensorflow.keras.layers import LeakyReLU
from sklearn.utils import shuffle
import numpy as np
import matplotlib.pyplot as plt
```

Load CIFAR10 dataset

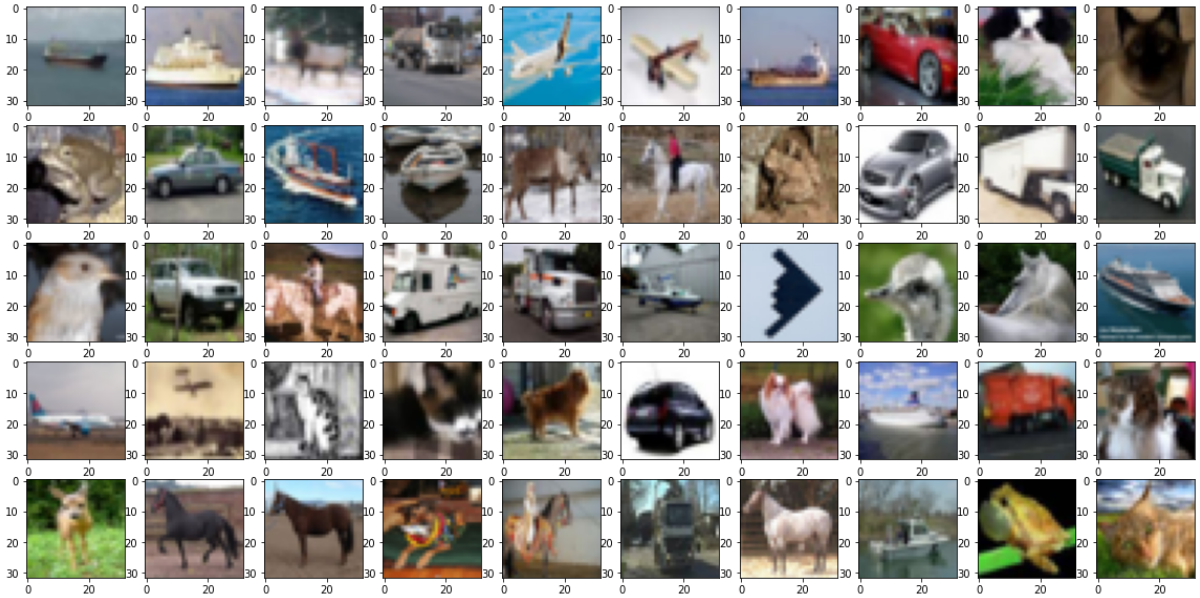
We use CIFAR10 dataset to train our model. This is a dataset of 50,000 32x32 color training images and 10,000 test images, labeled over 10 categories. Each example is 32×32 pixel color image of various objects.

In [6]:
```
# Helper function, You don't need to modify it
# 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

if tf.keras.backend.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], number_channels, img_row
s, 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, numb
er_channels)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, number_
channels)
    input_shape = (img_rows, img_cols, number_channels)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
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')

cifar10_classes = ["airplane", "automobile", "bird", "cat", "deer", "do
g", "frog", "horse", "ship", "truck"]
# 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)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.
tar.gz
170500096/170498071 [==============================] - 2s 0us/step
x_train shape: (50000, 32, 32, 3)
x_test shape: (10000, 32, 32, 3)
50000 train samples
10000 test samples
```

Load some images from CIFAR10

In [7]:
```
# Helper function, You don't need to modify it
# Show some images from CIFAR10

fig = plt.figure(figsize=(20, 10))
for i in range(50):
    random_index = np.random.randint(0, len(y_train))
    ax = fig.add_subplot(5, 10, i+1)
    ax.imshow(x_train[random_index, :])
plt.show()
```



As you can see from above, the CIFAR10 dataset contains selection of objects. The images have been size-normalized and objects remain centered in fixed-size images.

Build convolutional neural network model

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

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

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, and with 3 color channels. This layer should give 16 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 the 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 0.25, to prevent overfitting.

CONV: Additonal Conv. layer take 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 take 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 0.25, to prevent overfitting.

FC1: Dense layer which takes input 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 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 unless otherwise indicated to build you model architecture. We suggest starting with an alpha value of 0.1, but you are free to experiment.

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

In [4]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Implement CNN

In the cell below, implement the **create_net** and **compile_net** function. Set your parameters in the **init** function.

See below sections for more information on implementing these functions.

In [26]:
```
class CNN(object):
    def __init__(self):
        # change these to appropriate values
        self.batch_size = 32
        self.epochs = 20 # best to run for 20 epochs
        self.init_lr= 0.001 #learning rate
        self.dropout = 0.16

        # No need to modify these
        self.model = None

    def get_vars(self):
        return self.batch_size, self.epochs, self.init_lr

    def create_net(self):
        '''
        In this function you are going to build a convolutional neural n
etwork based on TF Keras.
        First, use Sequential() to set the inference features on this mo
del.
        Then, use model.add() to build layers in your own model
        Return: model
        '''

        #TODO: implement this



        self.model = Sequential()
        #Layer 1
        #Conv Layer 1
        self.model.add(Conv2D(filters = 16,
                              padding = 'same',
                              kernel_size = (3,3),
                              strides = 1,
                              activation = tf.keras.layers.LeakyReLU(alpha=0.
1),
                              input_shape = (32,32,3)))
        #Layer 2
        #Conv Layer 2
        self.model.add(Conv2D(filters = 32,
                              padding = 'same',
                              kernel_size = (3,3),
                              strides = 1,
                              activation = tf.keras.layers.LeakyReLU(alpha=0.
1),
                              input_shape = (32,32,16)))
        #Pooling layer 2
        self.model.add(MaxPooling2D(pool_size = (2,2)))
        # dropout added as regularizer
        self.model.add(Dropout(self.dropout))
        #Layer 3
        #Conv Layer 3
        self.model.add(Conv2D(filters = 32,
                              padding = 'same',
                              kernel_size = (3,3),
```

```
                        strides = 1,
                        activation = tf.keras.layers.LeakyReLU(alpha=0.
1),
                        input_shape = (16,16,32)))
        #Layer 4
        #Conv Layer 4
        self.model.add(Conv2D(filters = 64,
                        padding = 'same',
                        kernel_size = (3,3),
                        strides = 1,
                        activation = tf.keras.layers.LeakyReLU(alpha=0.
1),
                        input_shape = (16,16,32)))
        #Pooling Layer 4
        self.model.add(MaxPooling2D(pool_size = (2,2) ))
        # dropout added as regularizer
        self.model.add(Dropout(self.dropout))
        #Flatten
        self.model.add(Flatten())
        #Layer 5
        #Fully connected layer 1
        self.model.add(Dense(units = 256, activation = tf.keras.layers.L
eakyReLU(alpha=0.1)))
        #Layer 6
        # dropout added as regularizer
        self.model.add(Dropout(self.dropout))

        #Fully connected layer 2
        self.model.add(Dense(units = 10, activation = tf.keras.layers.Le
akyReLU(alpha=0.1)))

        self.model.add(Activation(tf.nn.leaky_relu))

        return self.model

    def compile_net(self, model):
        '''
        In this function you are going to compile the model you've creat
ed.
        Use model.compile() to build your model.
        '''
        self.model = model
        self.model.compile(optimizer='Adam',
              loss= 'MSE',
              metrics=['accuracy'])

        #TODO: implement this



        return self.model
```

Defining Variables

You now need to set training variables in the **init**() function in **CNN() class** in the above cell. Once you have defined variables you may use the cell below to see them.

In [27]:
```
# Helper function, You don't need to modify it
# You can adjust parameters to train your model in __init__() in cnn.py

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

```
Batch Size      : 32
Epochs          : 20
Learning Rate   : 0.001
```

Defining model

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

In [28]:
```
# Helper function, You don't need to modify it
# model.summary() gives you details of your architecture.
#You can compare your architecture with the 'Architecture.png'

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, 16)        448
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 32)        4640
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 32)        9248
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
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, 10)                2570
_________________________________________________________________
activation (Activation)      (None, 10)                0
=================================================================
Total params: 1,084,234
Trainable params: 1,084,234
Non-trainable params: 0
_________________________________________________________________
```

Compile model

Next prepare the model for training by completing **compile_model()** in **CNN() class** in the above cell. Remember we are performing 10-way clasification when selecting an appropriate loss function.

You are free to experiment with different optimizers (https://keras.io/api/optimizers/) available in the Keras library.

In [29]:
```
# Helper function, You don't need to modify it
# Complete compile_model() in cnn.py.

net = CNN()
model = net.compile_net(model)
print(model)
```

```
<tensorflow.python.keras.engine.sequential.Sequential object at 0x7f760
1894090>
```

Train the network

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 30 minutes to train your model. We suggest tuning for 20 epochs.

Expected Result: With the given architecture and hyperparameter tuning, you should be able to achieve 80% accuracy on the test set to get full 15 points. If you achieve accuracy between 75% to 79%, you will only get half points of this part.

Train your own CNN model

In [30]:
```
# Helper function, You don't need to modify it
# Train the model

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])
```

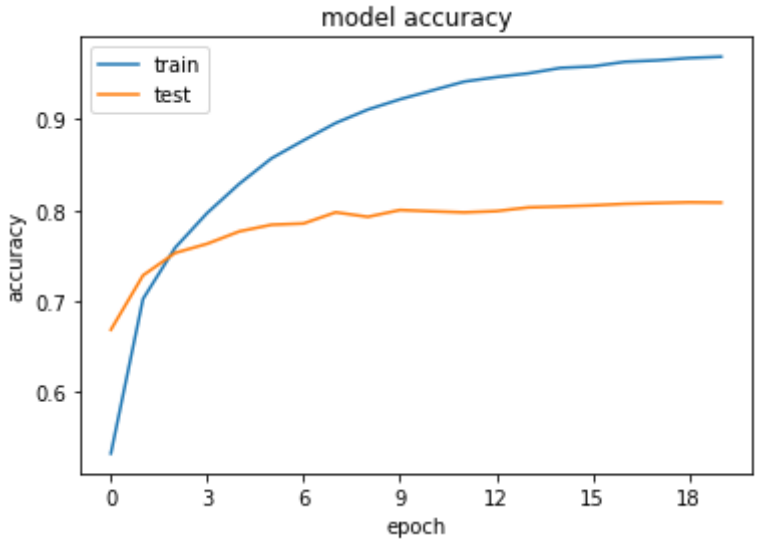
```
Epoch 1/20
Learning rate: 0.001
1563/1563 [==============================] - 209s 133ms/step - loss: 0.
0619 - accuracy: 0.5323 - val_loss: 0.0478 - val_accuracy: 0.6685
Epoch 2/20
Learning rate: 0.0009000000000000001
1563/1563 [==============================] - 209s 134ms/step - loss: 0.
0442 - accuracy: 0.7024 - val_loss: 0.0404 - val_accuracy: 0.7283
Epoch 3/20
Learning rate: 0.0008100000000000001
1563/1563 [==============================] - 211s 135ms/step - loss: 0.
0375 - accuracy: 0.7587 - val_loss: 0.0379 - val_accuracy: 0.7530
Epoch 4/20
Learning rate: 0.0007290000000000002
1563/1563 [==============================] - 210s 135ms/step - loss: 0.
0330 - accuracy: 0.7970 - val_loss: 0.0360 - val_accuracy: 0.7631
Epoch 5/20
Learning rate: 0.0006561000000000001
1563/1563 [==============================] - 210s 134ms/step - loss: 0.
0294 - accuracy: 0.8291 - val_loss: 0.0343 - val_accuracy: 0.7767
Epoch 6/20
Learning rate: 0.00059049
1563/1563 [==============================] - 210s 135ms/step - loss: 0.
0262 - accuracy: 0.8571 - val_loss: 0.0338 - val_accuracy: 0.7840
Epoch 7/20
Learning rate: 0.000531441
1563/1563 [==============================] - 209s 134ms/step - loss: 0.
0237 - accuracy: 0.8769 - val_loss: 0.0342 - val_accuracy: 0.7854
Epoch 8/20
Learning rate: 0.0004782969000000001
1563/1563 [==============================] - 209s 134ms/step - loss: 0.
0214 - accuracy: 0.8960 - val_loss: 0.0325 - val_accuracy: 0.7977
Epoch 9/20
Learning rate: 0.0004304672100000001
1563/1563 [==============================] - 210s 135ms/step - loss: 0.
0196 - accuracy: 0.9108 - val_loss: 0.0326 - val_accuracy: 0.7927
Epoch 10/20
Learning rate: 0.0003874204890000001
1563/1563 [==============================] - 211s 135ms/step - loss: 0.
0181 - accuracy: 0.9221 - val_loss: 0.0319 - val_accuracy: 0.8002
Epoch 11/20
Learning rate: 0.0003486784401000001
1563/1563 [==============================] - 212s 135ms/step - loss: 0.
0168 - accuracy: 0.9318 - val_loss: 0.0321 - val_accuracy: 0.7988
Epoch 12/20
Learning rate: 0.0003138105960900001
1563/1563 [==============================] - 211s 135ms/step - loss: 0.
0156 - accuracy: 0.9416 - val_loss: 0.0320 - val_accuracy: 0.7976
Epoch 13/20
Learning rate: 0.0002824295364810001
1563/1563 [==============================] - 212s 135ms/step - loss: 0.
0148 - accuracy: 0.9465 - val_loss: 0.0316 - val_accuracy: 0.7990
Epoch 14/20
Learning rate: 0.0002541865828329001
1563/1563 [==============================] - 212s 136ms/step - loss: 0.
0141 - accuracy: 0.9507 - val_loss: 0.0315 - val_accuracy: 0.8031
Epoch 15/20
```

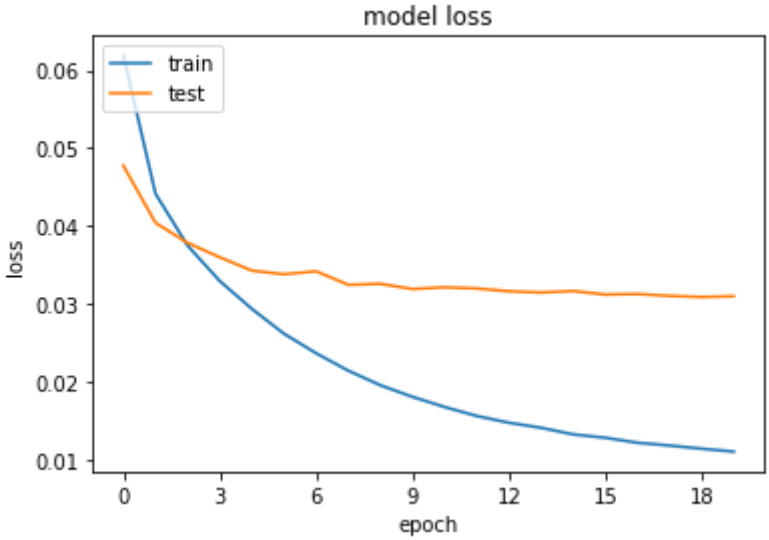
```
Learning rate: 0.0002287679245496101
1563/1563 [==============================] - 212s 136ms/step - loss: 0.
0133 - accuracy: 0.9566 - val_loss: 0.0317 - val_accuracy: 0.8041
Epoch 16/20
Learning rate: 0.0002058911320946491
1563/1563 [==============================] - 213s 136ms/step - loss: 0.
0128 - accuracy: 0.9583 - val_loss: 0.0312 - val_accuracy: 0.8053
Epoch 17/20
Learning rate: 0.00018530201888518417
1563/1563 [==============================] - 211s 135ms/step - loss: 0.
0122 - accuracy: 0.9634 - val_loss: 0.0313 - val_accuracy: 0.8070
Epoch 18/20
Learning rate: 0.00016677181699666576
1563/1563 [==============================] - 211s 135ms/step - loss: 0.
0118 - accuracy: 0.9650 - val_loss: 0.0310 - val_accuracy: 0.8079
Epoch 19/20
Learning rate: 0.00015009463529699917
1563/1563 [==============================] - 211s 135ms/step - loss: 0.
0114 - accuracy: 0.9675 - val_loss: 0.0309 - val_accuracy: 0.8087
Epoch 20/20
Learning rate: 0.00013508517176729931
1563/1563 [==============================] - 211s 135ms/step - loss: 0.
0110 - accuracy: 0.9690 - val_loss: 0.0310 - val_accuracy: 0.8084
Test loss: 0.031006069853901863
Test accuracy: 0.8083999752998352
```

In [ ]:

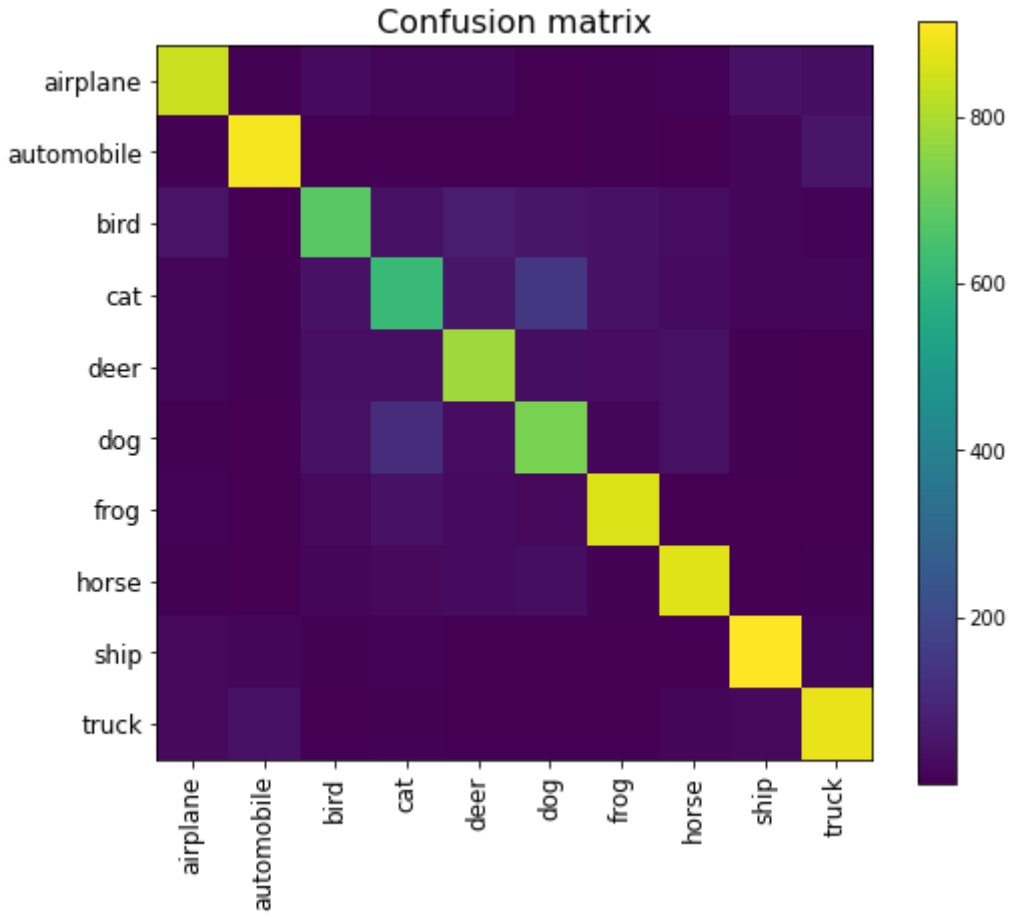
In [31]:
```
# Helper function, You don't need to modify it
# list all data in history
print(history.history.keys())
from matplotlib.ticker import MaxNLocator

# summarize history for accuracy and loss
ax1 = plt.figure().gca()
ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
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()

ax2 = plt.figure().gca()
ax2.xaxis.set_major_locator(MaxNLocator(integer=True))
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'])
```





In [32]:
```
# 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), cifar10_classes, rotation=90, fontsize=12)
plt.yticks(np.arange(10), cifar10_classes, fontsize=12)
plt.colorbar()
plt.show()
```



3: SVM (30 Pts) **[W]** **[P]**

Summer 2021 CS 4641\7641 A: Machine Learning Homework 4

Instructor: Dr. Mahdi Roozbahani

Deadline: July 28, Wednesday, AOE

- No unapproved extension of the deadline is allowed. Late submission will lead to 0 credit.
- Discussion is encouraged on EdStem as part of the Q/A. However, all assignments should be done individually.

Instructions for the assignment

- In this assignment, we have programming and theory questions.
- To switch between cell for code and for markdown, see the menu -> Cell -> Cell Type
- You could directly type the Latex equations in the markdown cell.
- Typing with LaTeX is required for all the written questions, and can be done in markdown cell types. Handwritten answers will not be accepted.
- If a question requires a picture, you could use this syntax "$<imgsrc=""style="width:300px;" />$" to include them within your ipython notebook.
- Questions marked with **[P]** are programming only and should be submitted to the autograder. Questions marked with **[W]** may require that you code, but should NOT be submitted to the autograder. It should be submitted on the written portion of the assignment on gradescope
- The outline of the assignment is as follows:
  - **Q1** [55 + (10 bonus-for-undergrads)] | **Neural Network** **[P]**|**[W]**
  - **Q2** [15 pts bonus-for-all] | **Image Classification based on Convolutional Neural Network** | **[W]**
  - **Q3** [30 pts] | **SVM** **[P]**|**[W]**
  - **Q4** [35 pts bonus-for-all] | **Random Forest** **[P]** 4.1, 4.2 | **[W]** 4.3

Using the autograder

Undergrad students will find four assignments on Gradescope that correspond to HW4: "HW4 - Programming", "HW4 - Programming (Bonus)", "HW4 - Non-Programming (Bonus for all)", and "HW4 - Non-programming". Graduate students will find three assignments on Gradescope that correspond to HW4: "HW4 - Programming", "HW4 - Non-Programming (Bonus for all)", and "HW4 - Non-programming".

For graduate students submit the following files "HW4 - Programming":

- nn.py
- feature.py

For graduate students submit the following files "HW4 - Programming (Bonus)":

- random_forest.py

We have provided you different .py files and added libraries in those .py 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 may not add additional libraries.

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.

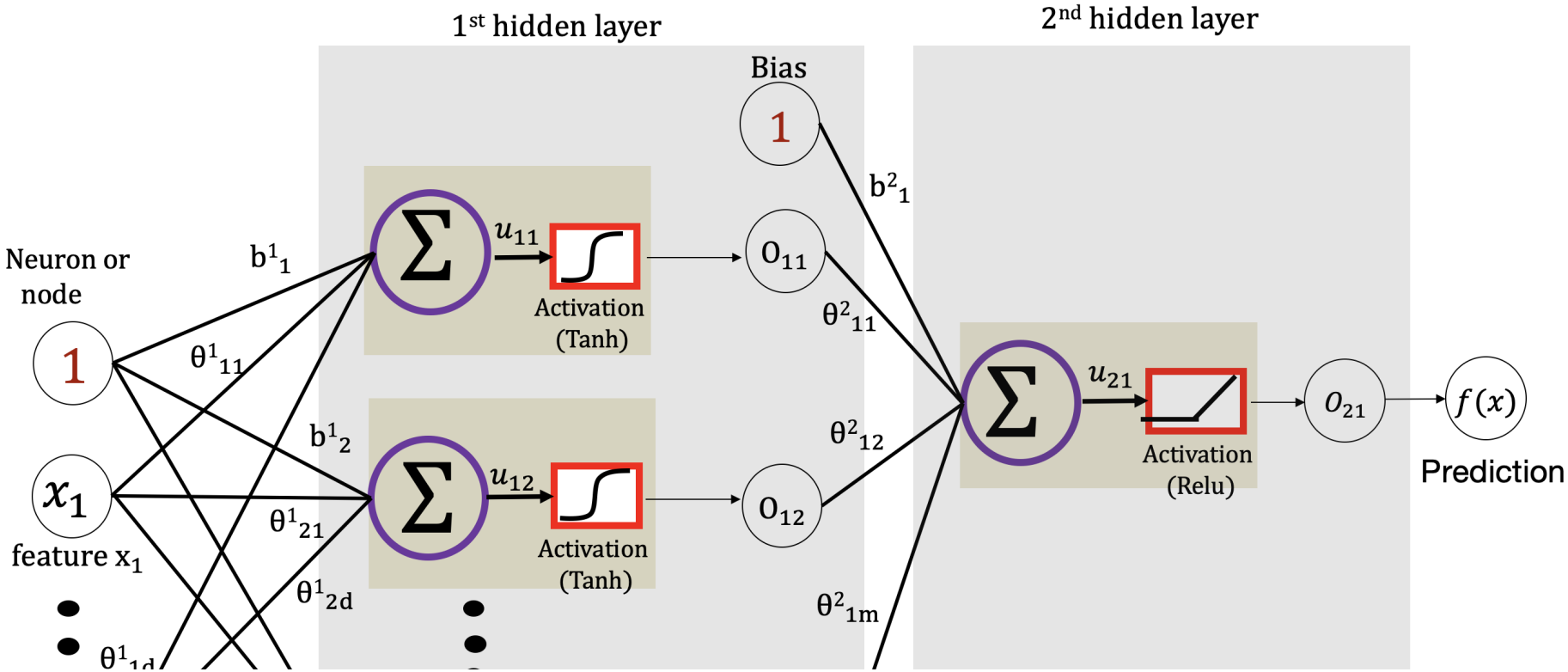
For the "HW4 - Non-programming" part, you will download your jupyter notebook as HTML, print it as a PDF from your browser and submit it on Gradescope. To download the notebook as html, click on "File" on the top left corner of this page and select "Download as > HTML". The non-programming part corresponds to Q1. For questions that include images include both your response and the generated images in your submission

Environment Setup

In [ ]:
```
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 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_circles
from sklearn.metrics import accuracy_score
from sklearn import svm

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

1. Two Layer Neural Network [65pts] **[P]****[W]**

Perceptron


1st hidden layer
2nd hidden layer
Bias
1
Neuron or node
1
x_1
feature x_1
b^1_1
b^1_2
θ^1_{11}
θ^1_{21}
θ^1_{2d}
θ^1_{1d}
u_{11}
u_{12}
Activation (Tanh)
Activation (Tanh)
O_{11}
O_{12}
b^2_1
θ^2_{11}
θ^2_{12}
θ^2_{1m}
u_{21}
Activation (Relu)
O_{21}
$f(x)$
Prediction

Fully connected Layer

Typically, a modern neural network contains millions of perceptrons as the one previously discussed. 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 Relu and Tanh only.

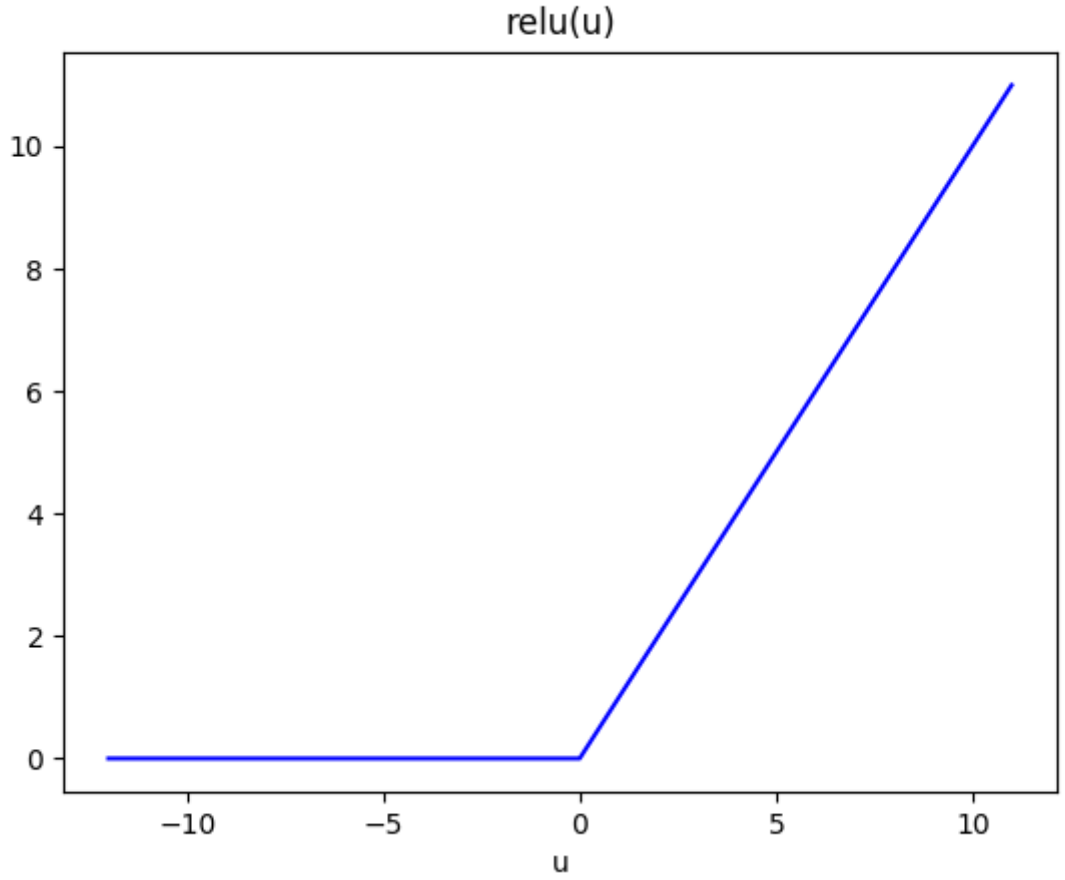
HINT 1: When calculating the tanh and relu function, make sure you are not modifying the values in the original passed in matrix. You may find np.copy() helpful.

Relu

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

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

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



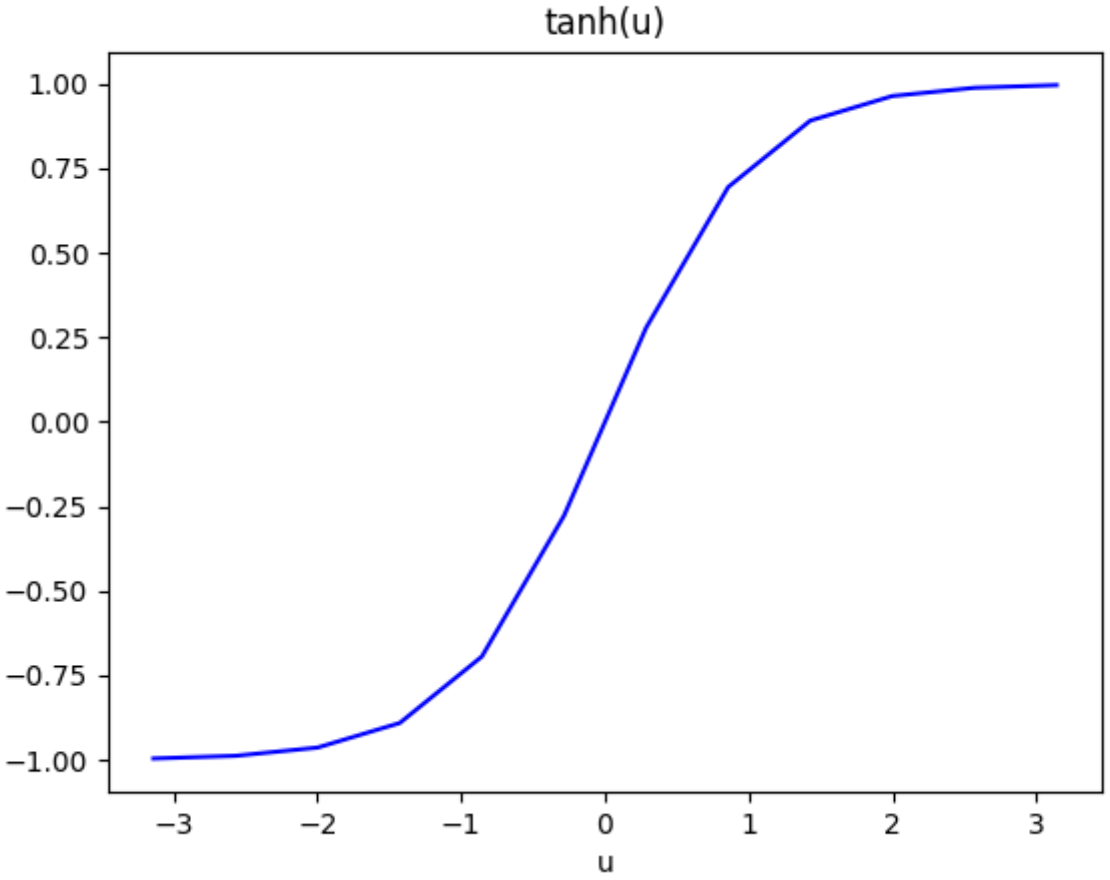
Tanh

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

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



Sigmoid

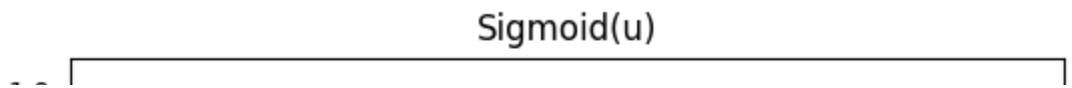
The sigmoid function is another non-linear function with S-shaped curve. This function is useful in the case of binary classification as its output is between 0 and 1. The mathematical form of the function is

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

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.



Mean Squared Error

It 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. It estimates the quality of the learnt hypothesis between the actual and the predicted values. It's non-negative and 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 simplify the derivative of loss function.

Forward Propagation

We start by initializing the weights of the fully connected layer using Xavier initialization Xavier initialization (http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf). 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.

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

Then we get the output and compute the loss

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

Backward propagation

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. 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.

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:

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

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 Relu 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 Relu differentiation are given by

$$\frac{\partial l}{\partial o^{[2]}} = \left(o^{[2]} - y\right)$$
$$\frac{\partial l}{\partial u^{[2]}} = \frac{\partial l}{\partial o^{[2]}} * 1(u^{[2]} > 0)$$
$$\frac{\partial u^{[2]}}{\partial \theta^{[2]}} = o^{[1]}$$
$$\frac{\partial u^{[2]}}{\partial b^{[2]}} = 1$$

On vectorization, the above equations become:

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

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

$$\frac{\partial u^{[2]}}{\partial o^{[1]}} = \theta^{[2]}$$
$$\frac{\partial o^{[1]}}{\partial u^{[1]}} = 1 - (o^{[1]})^2$$
$$\frac{\partial u^{[1]}}{\partial \theta^{[1]}} = x$$
$$\frac{\partial u^{[1]}}{\partial b^{[1]}} = 1$$

Note that $\frac{\partial o^{[1]}}{\partial u^{[1]}}$ is the differentiation of the Tanh function at $u^{[1]}$.

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

Code Implementation:

$$dLoss\_o2 = \frac{\partial l}{\partial o^{[2]}} \implies dim = (1, 331)$$
$$dLoss\_u2 = dLoss\_o2 \frac{\partial o^{[2]}}{\partial u^{[2]}} \implies dim = (1, 331)$$
$$dLoss\_theta2 = dLoss\_u2 \frac{\partial u^{[2]}}{\partial \theta^{[2]}} \implies dim = (1, 15)$$
$$dLoss\_b2 = dLoss\_u2 \frac{\partial u^{[2]}}{\partial b^{[2]}} \implies dim = (1, 1)$$
$$dLoss\_o1 = dLoss\_u2 \frac{\partial u^{[2]}}{\partial o^{[1]}} \implies dim = (15, 331)$$
$$dLoss\_u1 = dLoss\_o1 \frac{\partial o^{[1]}}{\partial u^{[1]}} \implies dim = (15, 331)$$
$$dLoss\_theta1 = dLoss\_u1 \frac{\partial u^{[1]}}{\partial \theta^{[1]}} \implies dim = (15, 10)$$
$$dLoss\_b1 = dLoss\_u1 \frac{\partial u^{[1]}}{\partial b^{[1]}} \implies dim = (15, 1)$$

Note: Training set has 331 examples.

Question

In this question, you will implement a two layer fully connected neural network. You will also experiment with different activation functions and optimization techniques. Functions with comments "TODO: implement this" are for you to implement. We provide two activation functions here - Relu and Tanh. You will implement a neural network that would have tanh activation followed by relu 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:

- **Relu**: Recall Hint 1
- **Tanh**: Recall Hint 1
- **nloss**
- **forward**
- **backward**: Recall Hint 2
- **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. Graduate students have to use both GD and BGD to optimize their neural net. Undergraduate students have to implement GD while BGD is bonus for them. Note: it is possible you'll run into nan or negative values for loss. This happens because of the small dataset we're using and some numerical stability issues that arise due to division by zero, natural log of zeros etc. You can experiment with the total number of iterations to mitigate this.

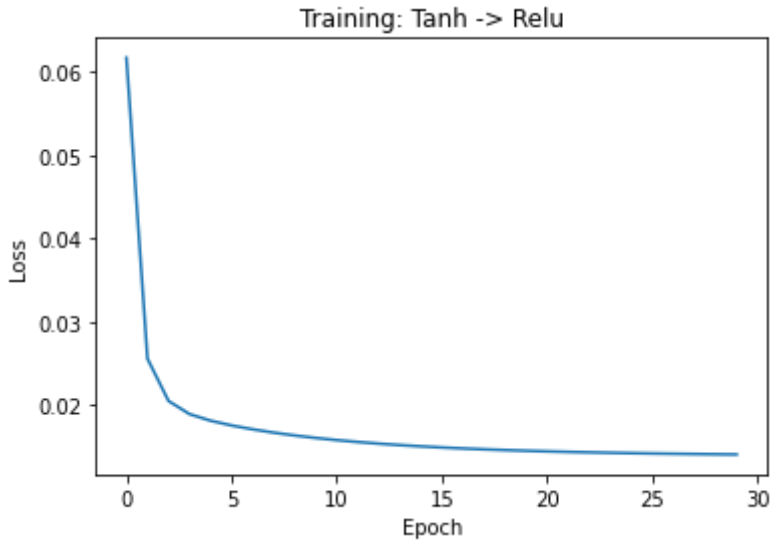
You're free to tune hyperparameters like the batch size, number of hidden units in each layer etc. if that helps you in achieving the desired MSE values to pass the autograder tests. However, you're advised to try out the default values first.

Deliverables for this question:

1. Loss plot and MSE value for neural net with gradient descent
2. Loss plot and MSE value for neural net with batch gradient descent (mandatory for graduate students, bonus for undergraduate students)

In [25]:
```
'''
Training the Neural Network with Gradient Descent, you do not need to modify this cell.
'''
#from template_files.NN import dlnet
from NN import dlnet

# load dataset
dataset = load_diabetes() # load the dataset
x, y = dataset.data, dataset.target
y = y.reshape(-1,1)

x = MinMaxScaler().fit_transform(x) #normalize data
y = MinMaxScaler().fit_transform(y)

x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=1) #split data
x_train, x_test, y_train, y_test = x_train.T, x_test.T, y_train.reshape(1,-1), y_test #condition data

nn = dlnet(x_train,y_train,lr=0.001) # 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")
```

Out[25]: Text(0, 0.5, 'Loss')

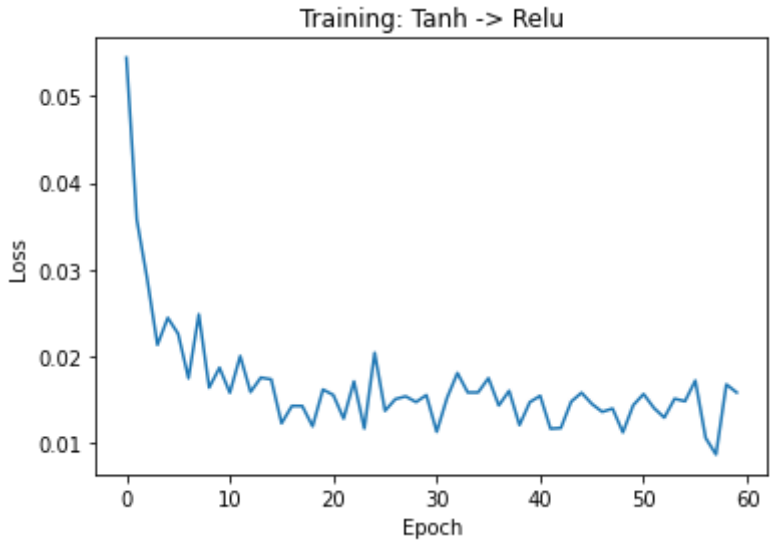


In [26]:
```
'''
Testing Neural Network with Gradient Descent, you do not need to modify this cell.
'''
y_predicted = nn.predict(x_test) # predict
y_test = y_test.reshape(1,-1)
print("Mean Squared Error (MSE)", (np.sum((y_predicted-y_test)**2)/y_test.shape[1]))
```

```
Mean Squared Error (MSE) 0.02773885780084753
```

In [27]:
```
'''
Training the Neural Network with Batch Gradient Descent, you do not need to modify this cell.
'''
from NN import dlnet

# load dataset
dataset = load_diabetes() # load the dataset
x, y = dataset.data, dataset.target
y = y.reshape(-1,1)

x = MinMaxScaler().fit_transform(x) #normalize data
y = MinMaxScaler().fit_transform(y)

x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=1) #split data
x_train, x_test, y_train, y_test = x_train.T, x_test.T, y_train.reshape(1,-1), y_test #condition data

nn = dlnet(x_train,y_train,lr=0.001) # 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")
```

Out[27]: Text(0, 0.5, 'Loss')



In [28]:
```
'''
Testing Neural Network with Batch Gradient Descent, you do not need to modify this cell.
'''
y_predicted = nn.predict(x_test) # predict
y_test = y_test.reshape(1,-1)
print("Mean Squared Error (MSE)", (np.sum((y_predicted-y_test)**2)/y_test.shape[1]))
```

Mean Squared Error (MSE) 0.02776878539608432

2: (Bonus for all) Image Classification based on Convolutional Neural Networks [15pts] ** [W]**

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 Keras to solve the image classification task for CIFAR10. If you haven't installed TensorFlow, you can install the package by pip command or train your model by uploading HW4 notebook to Colab (https://colab.research.google.com/) directly. Colab contains all packages you need for this section.

Hint1: First contact with Keras (https://keras.io/about/)

Hint2: How to Install Keras (https://www.pyimagesearch.com/2016/07/18/installing-keras-for-deep-learning/)

Hint3: CS231n Tutorial (Layers used to build ConvNets) (https://cs231n.github.io/convolutional-networks/)

Environment Setup

In [ ]:
```
from __future__ import print_function
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Activation, Dropout
from tensorflow.keras.layers import LeakyReLU
from sklearn.utils import shuffle
import numpy as np
import matplotlib.pyplot as plt
```

Load CIFAR10 dataset

We use CIFAR10 dataset to train our model. This is a dataset of 50,000 32x32 color training images and 10,000 test images, labeled over 10 categories. Each example is 32×32 pixel color image of various objects.

In [ ]:
```
# Helper function, You don't need to modify it
# 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

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)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
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')

cifar10_classes = ["airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse", "ship",
"truck"]
# 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)
```

Load some images from CIFAR10

In [ ]:
```
# Helper function, You don't need to modify it
# Show some images from CIFAR10

fig = plt.figure(figsize=(20, 10))
for i in range(50):
    random_index = np.random.randint(0, len(y_train))
    ax = fig.add_subplot(5, 10, i+1)
    ax.imshow(x_train[random_index, :])
plt.show()
```

As you can see from above, the CIFAR10 dataset contains selection of objects. The images have been size-normalized and objects remain centered in fixed-size images.

Build convolutional neural network model

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

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

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, and with 3 color channels. This layer should give 16 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 the 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 0.25, to prevent overfitting.

CONV: Additonal Conv. layer take 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 take 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 0.25, to prevent overfitting.

FC1: Dense layer which takes input 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 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 unless otherwise indicated to build you model architecture. We suggest starting with an alpha value of 0.1, but you are free to experiment.

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

In [ ]:
```
# Helper function, You don't need to modify it
# Show the architecture of the model
achi=plt.imread('data/Architecture.png')
fig = plt.figure(figsize=(10,10))
plt.imshow(achi)
```

Implement CNN

In the cell below, implement the **create_net** and **compile_net** function. Set your parameters in the **init** function.

See below sections for more information on implementing these functions.

In [ ]:
```
class CNN(object):
    def __init__(self):
        # change these to appropriate values
        self.batch_size = 16
        self.epochs = 2 # best to run for 20 epochs
        self.init_lr= 0.001 #learning rate
        dropout = 0.25

        # No need to modify these
        self.model = None

    def get_vars(self):
        return self.batch_size, self.epochs, self.init_lr

    def create_net(self):
        '''
        In this function you are going to build a convolutional neural network based on TF Keras.
        First, use Sequential() to set the inference features on this model.
        Then, use model.add() to build layers in your own model
        Return: model
        '''

        #TODO: implement this



        model = Sequential()
        #Layer 1
        #Conv Layer 1
        model.add(Conv2D(filters = 16,
                         kernel_size = 3,
                         strides = 1,
                         activation = 'leakyrelu',
                         input_shape = (32,32,3)))
        #Layer 2
        #Conv Layer 2
        model.add(Conv2D(filters = 32,
                         kernel_size = 3,
                         strides = 1,
                         activation = 'leakyrelu',
                         input_shape = (32,32,16)))
        #Pooling layer 2
```

```
        model.add(MaxPooling2D(pool_size = 2, strides = 2))
        # dropout added as regularizer
        model.add(Dropout(dropout))
        #Layer 3
        #Conv Layer 3
        model.add(Conv2D(filters = 32,
                         kernel_size = 3,
                         strides = 1,
                         activation = 'leakyrelu',
                         input_shape = (16,16,32)))
        #Layer 4
        #Conv Layer 4
        model.add(Conv2D(filters = 64,
                         kernel_size = 3,
                         strides = 1,
                         activation = 'leakyrelu',
                         input_shape = (16,16,32)))
        #Pooling Layer 4
        model.add(MaxPooling2D(pool_size = 2, strides = 2))
        # dropout added as regularizer
        model.add(Dropout(dropout))
        #Flatten
        model.add(Flatten())
        #Layer 5
        #Fully connected layer 1
        model.add(Dense(units = 256, activation = 'leakyrelu'))
        #Layer 6
        # dropout added as regularizer
        model.add(Dropout(dropout))

        #Fully connected layer 2
        model.add(Dense(units = 10, activation = 'leakyrelu'))



        return self.model

    def compile_net(self, model):
        '''
        In this function you are going to compile the model you've created.
        Use model.compile() to build your model.
        '''
        self.model = model
```

```
        #TODO: implement this


        return self.model
```

Defining Variables

You now need to set training variables in the **init**() function in **CNN() class** in the above cell. Once you have defined variables you may use the cell below to see them.

In [ ]:
```
# Helper function, You don't need to modify it
# You can adjust parameters to train your model in __init__() in cnn.py

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')
```

Defining model

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

In [ ]:
```
# Helper function, You don't need to modify it
# model.summary() gives you details of your architecture.
#You can compare your architecture with the 'Architecture.png'

net = CNN()

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

Compile model

Next prepare the model for training by completing **compile_model()** in **CNN() class** in the above cell. Remember we are performing 10-way clasification when selecting an appropriate loss function.

You are free to experiment with different optimizers (https://keras.io/api/optimizers/) available in the Keras library.

In [ ]:
```
# Helper function, You don't need to modify it
# Complete compile_model() in cnn.py.

net = CNN()
model = net.compile_net(model)
print(model)
```

Train the network

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 30 minutes to train your model. We suggest tuning for 20 epochs.

Expected Result: With the given architecture and hyperparameter tuning, you should be able to achieve 80% accuracy on the test set to get full 15 points. If you achieve accuracy between 75% to 79%, you will only get half points of this part.

Train your own CNN model

In [ ]:
```
# Helper function, You don't need to modify it
# Train the model

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])
```

In [ ]:
```
# Helper function, You don't need to modify it
# list all data in history
print(history.history.keys())
from matplotlib.ticker import MaxNLocator

# summarize history for accuracy and loss
ax1 = plt.figure().gca()
ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
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()

ax2 = plt.figure().gca()
ax2.xaxis.set_major_locator(MaxNLocator(integer=True))
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()
```

In [ ]:
```
# 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), cifar10_classes, rotation=90, fontsize=12)
plt.yticks(np.arange(10), cifar10_classes, fontsize=12)
plt.colorbar()
plt.show()
```

3: SVM (30 Pts) **[W]** **[P]**

3.1 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	-1	-1
0	3	-1
1	1	-1
2	-1	1
3	1	1
4	2	1

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

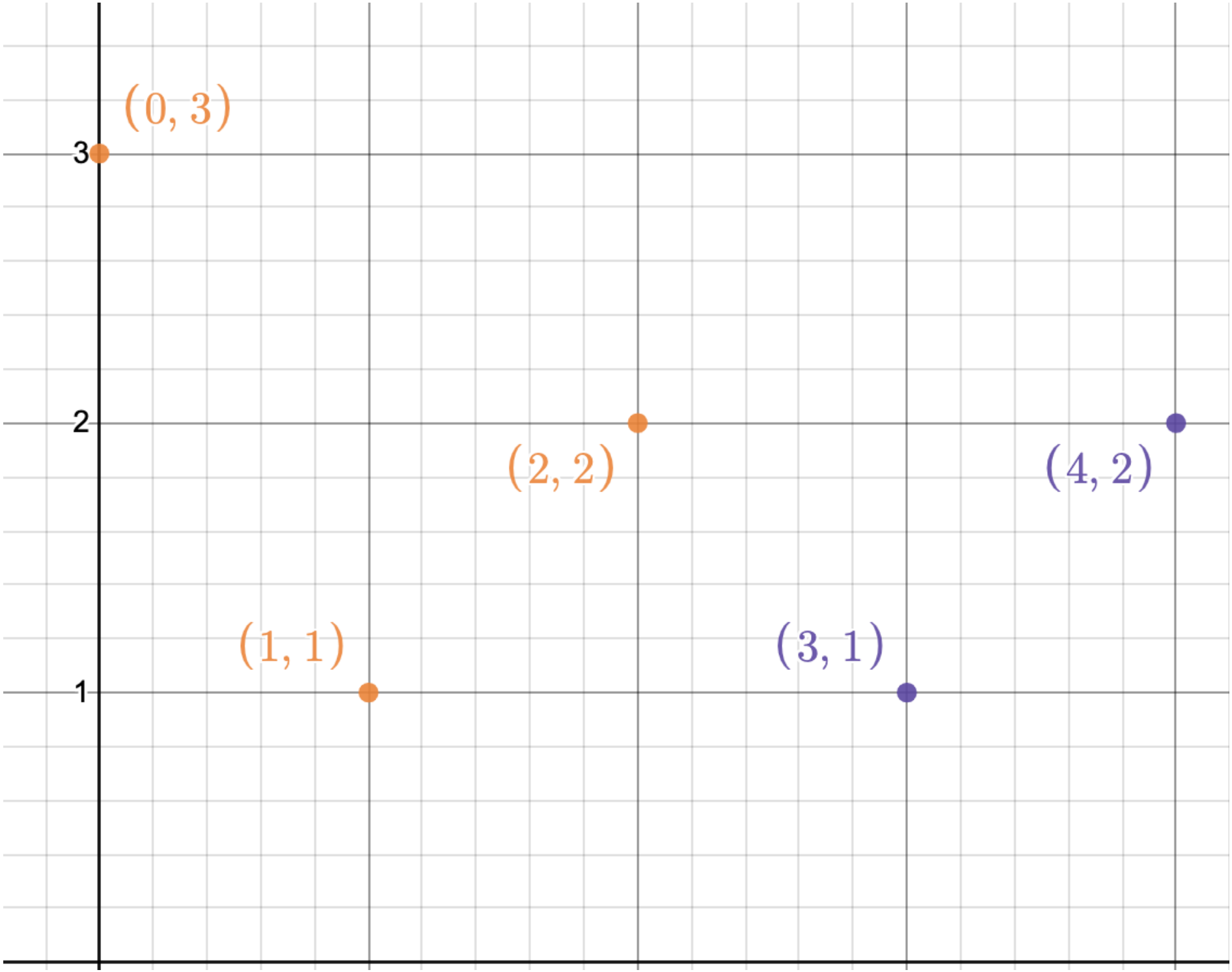
The max margin classifier has the formulation,

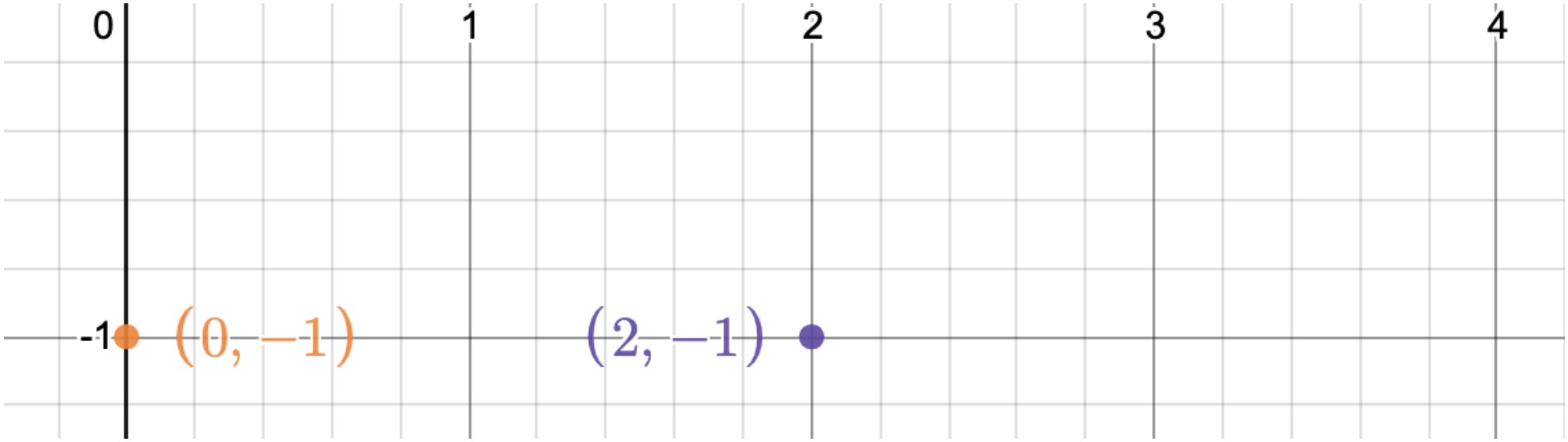
$$\min ||\theta||^2$$
$$s.t.\ \ y_i(\mathbf{x_i}\theta + b) \geq 1 \quad \forall\, i$$

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.

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

Answer:


(0, 3)
3
2
(2, 2)
(4, 2)
(1, 1)
(3, 1)
1



I have plotted points as you can see above (class1 in orange and calss-1 in purple). Yes, they are linearly seperable. For example the line y=x-2 can seperate the points linearly. By adding, $\mathbf{x} = (2, 2)$, $y = -1$ they still can be seperated linearly as it will be in the above classifer line in class1.

(2) According to the max-margin formulation, find the separating hyperplane. (4 pts)

Answer:

The plane that maximizes the distance to the closest data points in both side is the hyperplane with maximum margin.

we know that we have to find:

minα $\frac{1}{2}\sum{i=1}^{N} \sum_{j=1}^{N} y_iy_j \alpha_i \alpha_j x_i xj^T - \sum{i=1}^{N} \alpha$i with respect to$\sum{i=1}^{N} \alpha${i}y{i} = 0$

So, we have:

$$\sum_{i=1}^{6} \alpha_i y_i = 0 \longrightarrow \alpha_1(-1) + \alpha_2(-1) + \alpha_3(-1) + \alpha_4(1) + \alpha_5(1) + \alpha_6(1) = 0 \longrightarrow \alpha_1 + \alpha_2 + \alpha_3 = \alpha_4 + \alpha_5 + \alpha_6$$

On the other hand, we know that:

$$\theta = \sum_{i=1}^{6} \alpha_i y_i x_i \longrightarrow \alpha_1(-1)\begin{bmatrix} 0 & -1 \end{bmatrix} + \alpha_2(-1)\begin{bmatrix} 0 & 3 \end{bmatrix} + \alpha_3(-1)\begin{bmatrix} 1 & 1 \end{bmatrix} + \alpha_4(1)\begin{bmatrix} 2 & -1 \end{bmatrix} + \alpha_5(1)\begin{bmatrix} 3 & 1 \end{bmatrix} + \alpha_6(1)\begin{bmatrix} 4 & 2 \end{bmatrix}$$

so we have:

$$\theta = \begin{bmatrix} -\alpha_3 + 2\alpha_4 + 3\alpha_5 + 4\alpha_6 & \alpha_1 - 3\alpha_2 - \alpha_3 - \alpha_4 + \alpha_5 + 2\alpha_6 \end{bmatrix}$$

The quadratic function is a 6x6 matrix. we have :

$$Quadratic-coefficients = \begin{bmatrix} y_1 y_1 x_1 x_1^T & y_1 y_2 x_1 x_2^T & y_1 y_3 x_1 x_3^T & y_1 y_4 x_1 x_4^T & y_1 y_5 x_1 x_5^T & y_1 y_6 x_1 x_6^T \\ y_2 y_1 x_2 x_1^T & y_2 y_2 x_2 x_2^T & y_2 y_3 x_2 x_3^T & y_2 y_4 x_2 x_4^T & y_2 y_5 x_2 x_5^T & y_2 y_6 x_2 x_6^T \\ y_3 y_1 x_3 x_1^T & y_3 y_2 x_3 x_3^T & y_3 y_3 x_2 x_3^T & y_3 y_4 x_3 x_4^T & y_3 y_5 x_3 x_5^T & y_3 y_6 x_3 x_6^T \\ y_4 y_1 x_4 x_1^T & y_4 y_2 x_4 x_2^T & y_4 y_3 x_4 x_3^T & y_4 y_4 x_4 x_4^T & y_4 y_5 x_4 x_5^T & y_4 y_6 x_4 x_6^T \\ y_5 y_1 x_5 x_1^T & y_5 y_2 x_5 x_2^T & y_5 y_3 x_5 x_3^T & y_5 y_4 x_5 x_4^T & y_5 y_5 x_5 x_5^T & y_5 y_6 x_5 x_6^T \\ y_6 y_1 x_6 x_1^T & y_6 y_2 x_6 x_2^T & y_6 y_3 x_6 x_3^T & y_6 y_4 x_6 x_4^T & y_6 y_5 x_6 x_5^T & y_6 y_6 x_6 x_6^T \end{bmatrix}$$

$$Quadratic-coefficients = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -1 & -4 & -6 \\ 0 & 0 & -1 & 5 & 5 & 6 \\ 0 & 0 & -4 & 7 & 8 & 10 \\ 0 & 0 & -2 & 10 & 10 & 12 \end{bmatrix}$$

Accordginly we have:

$$min_\alpha \frac{1}{2} \begin{bmatrix} \alpha_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \alpha_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \alpha_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & \alpha_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & \alpha_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & \alpha_6 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -1 & -4 & -6 \\ 0 & 0 & -1 & 5 & 5 & 6 \\ 0 & 0 & -4 & 7 & 8 & 10 \\ 0 & 0 & -2 & 10 & 10 & 12 \end{bmatrix} \begin{bmatrix} \alpha_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \alpha_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \alpha_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & \alpha_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & \alpha_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & \alpha_6 \end{bmatrix} - \begin{bmatrix} \alpha_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \alpha_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \alpha_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & \alpha_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & \alpha_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & \alpha_6 \end{bmatrix}$$

$$= min_\alpha \frac{1}{2} \begin{bmatrix} -\alpha_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\alpha_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2\alpha_3\alpha_3 - \alpha_3 & -1\alpha_3\alpha_4 & -4\alpha_3\alpha_5 & -6\alpha_3\alpha_6 \\ 0 & 0 & -1\alpha_4\alpha_3 & 5\alpha_4\alpha_4 - \alpha_4 & 5\alpha_4\alpha_5 & 6\alpha_4\alpha_6 \\ 0 & 0 & -4\alpha_5\alpha_3 & 7\alpha_5\alpha_4 & 8\alpha_5\alpha_5 - \alpha_5 & 10\alpha_5\alpha_6 \\ 0 & 0 & -2\alpha_6\alpha_3 & 10\alpha_6\alpha_4 & 10\alpha_6\alpha_5 & 12\alpha_6\alpha_6 - \alpha_6 \end{bmatrix}$$

subject to:

$$\sum_{i=1}^{6} \alpha_i y_i = 0 \longrightarrow \alpha_1(-1) + \alpha_2(-1) + \alpha_3(-1) + \alpha_4(1) + \alpha_5(1) + \alpha_6(1) = 0 \longrightarrow \alpha_1 + \alpha_2 + \alpha_3 = \alpha_4 + \alpha_5 + \alpha_6$$

and

$lower - bound(0) \leq \alpha \leq upper - bound(inf)$ We can use quadratic programming to find α and by that θ is obtained.

On the other had, if we plot them, based 0n what TA told us, we can find the max distance between two classes approximately by drawing line. The line that lies between two classes with equal distance from both side is $y = 2x - 3$. Considering the fact that the line (3,1) and (2, -1) are on support vector line. So, we the have the line $2x_1 - x_2 - 3 = 0$

(3) Find a vector parallel to the optimal vector θ. (4 pts)

A line that is parallel to θ is prepandicular to classifier line. Let say is passes one point of class2 (e.g., (2,-1). So its equation is is $y = \frac{-1}{2}x$.

A vector parallel to $\theta = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$

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

Answer:

Here is the distance of each point to the line:

$$(0,-1) \longrightarrow \frac{2}{\sqrt{5}}$$
$$(0,3) \longrightarrow \frac{6}{\sqrt{5}}$$
$$(0,3) \longrightarrow \frac{6}{\sqrt{5}}$$
$$(4,2) \longrightarrow \frac{2}{\sqrt{5}}$$
$$(2,-1) \longrightarrow \frac{2}{\sqrt{5}}$$
$$(3,-1) \longrightarrow \frac{4}{\sqrt{5}}$$

the min is $\frac{2}{\sqrt{5}}$

$\frac{1}{||\theta||} = \frac{2}{\sqrt{5}}$ and full margin is $\frac{2}{||\theta||} = \frac{4}{\sqrt{5}}$

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

We know the classifer line is $x_2 = x_1 - 3$. This is equavalant to $[x_1 x_2][2 - 1]^T - 3 = 0$. For the point on support vector lines we need to make sure that $|x_i\theta + b| = 1$. Thus, we have $[x_1, x_2][1, -0.5]^T - 1.5 = 0$. Hence, we have:

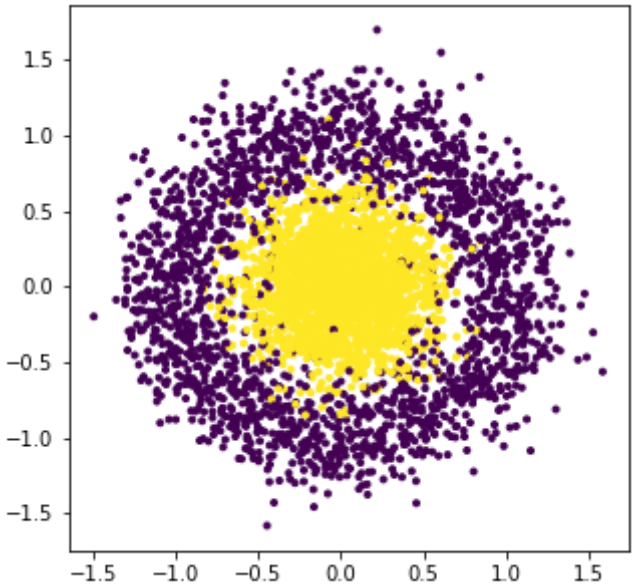
$$\theta = \begin{bmatrix} 1 \\ -0.5 \end{bmatrix}$$

3.2 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.

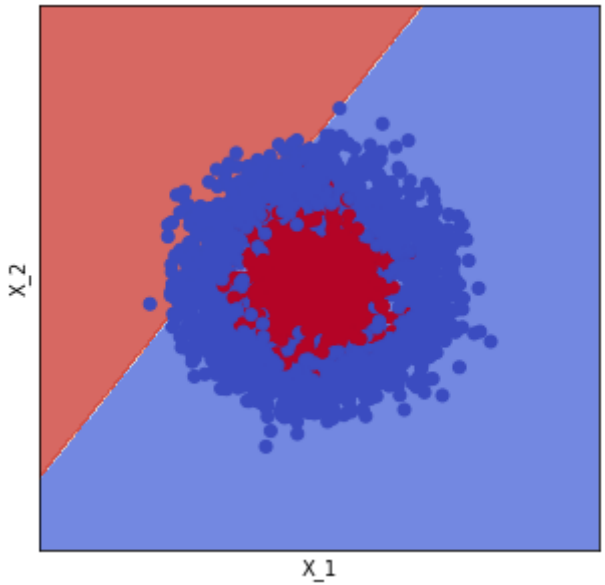
In [29]:
```
# DO NOT CHANGE
# Generate dataset

random_state = 1

X,y = make_circles(n_samples = 4000, noise = 0.2, factor = .3, random_state = random_state)
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 [30]:
```
# DO NOT CHANGE

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 [31]:
```
# DO NOT CHANGE
# 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)
```

```
Accuracy on test dataset: 0.395
```



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 the function below 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.

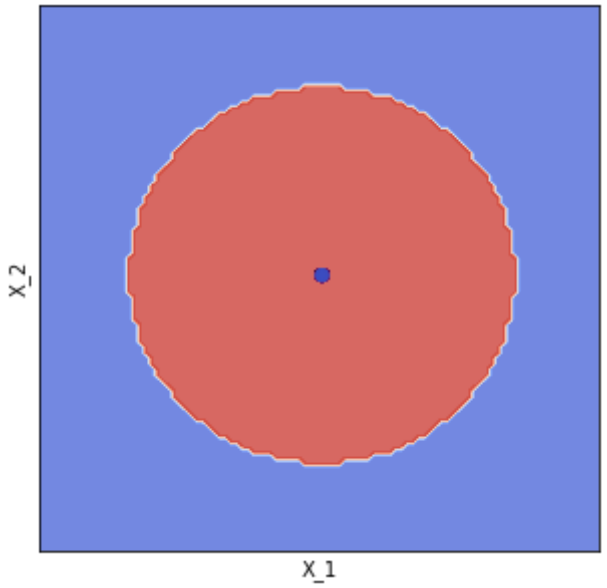
Note: You should get an accuracy above 95%

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 (https://xavierbourretsicotte.github.io/Kernel_feature_map.html) for a detailed analysis of doing the same for points separable with a circular boundary

In [32]:
```
# DO NOT CHANGE
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 [33]:
```
# DO NOT CHANGE
# Fit to the new features and vizualize the decision boundary
# You should get more than 75% 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)
```

```
Accuracy on test dataset: 0.95625
```



4: (Bonus for All) Random Forests [35pts] **[P]** **[W]**

NOTE: We will use sklearn's DecisionTreeClassifier in your Random Forest implementation. [You can find more details about this classifier here. (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier_)](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier_)

NOTE: Be careful not to confuse the use of the word "sample", which could mean either the number of samples/observations in the data, or a random sample/subset of the original data. Understanding the contex is part of the homework, as serveral ML articles will use both meanings of sample.

Jargon	Variable Representing Size-of
observations/samples	N
features/attributes	D

4.1 Random Forest Implementation (30 pts) **[P]**

Implement the functions in random_forest.py that are not already implemented. You need not modify the functions already implemented.

The decision boundaries drawn by decision trees are very sharp, and fitting a decision tree of unbounded depth to a list of examples almost inevitably leads to **overfitting**. In an attempt to decrease the variance of a decision tree, we're going to use a technique called '*bootstrap aggregating*', or in shorthand '*bagging*'. **The idea is that a collection of weak learners (many randomized imperfect decision trees) can learn decision boundaries as well as a strong learner (single best decision tree).**

Hence as a collection of randomized imperfect decision trees makes a Random Forest. 🌳🌲🌳🌲🌳🌲

Summary of Our Algorithm for Generating a Random Forest:

> **For Each Decision Tree in the Random Forest:**
>
> *Note, the number of decision trees is equal to the user-entered hyperparameter num_estimators*
>
> 1. Randomly sample N observations/samples with replacement from the inputed NxD dataset. For our implementation — the number of observations/samples is equal to the original datasets number of observations/samples. For examples, if we had 100 samples before, then we will still have 100 samples for this tree, but some samples will be duplicated or removed.
> 2. Randomly sample a fraction of the D features/attributes from the inputed NxD dataset. For our implementation — the fraction of features is equal a user-entered hyperparameter denoted as max_features. For examples, if we had 10 features before, then we will use 8 random features for this tree if max_features=0.8.
> 3. Fit a decision tree to the subset of data we selected during the bootstrapping. Each decision tree is restricted to a max depth equal to the user-entered hyperparameter max_depth.

Now that we have a bunch of random decision trees how do we classify a new sample from the testing data? Classification for a random forest is then done by taking a simple majority vote of the classifications yielded by all of the generated decision trees in the forest after they all have classified the new sample.

4.2 Hyperparameter Tuning with a Random Forest (2.5pts) **[P]**

Hyperparameters are input parameters that are set before a learning process begins. One example of a hyperparameter is λ from Ridge Regression and Lasso. For λ we simply test a handful of values, e.g. {0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000}, and select the value for λ that results in the highest testing accuracy. So, we must ***tune*** the value of our hyperparameter to the maximize accuracy on testing data.

However, random forest is a little more challenging as there are three hyperparameters: **max_depth**, **num_estimators**, and **max_features**. Documentation on these three hyperparameters located in the init of random_forest.py. How you cope with this will be tested here. We must create an iterative method/algorithm to test which values for the hyperparameters above result in the highest testing accuracy. The process of finding the optimal hyperparameters is called ***tunning our hyperparameters***.

Understanding Our Dataset Objectives:

We will be examining the dataset found in data/titanic.csv.

We will use our random forest algorithm to predict if a passager survived or did not survive given the other features.

Consider reviewing the table found under "Survivors and victims" hear to get a feeling for what features may predict survival: https://en.wikipedia.org/wiki/Titanic#Survivors_and_victims (https://en.wikipedia.org/wiki/Titanic#Survivors_and_victims)

Features:

Feature	Description
Survival	Survival (0 = No; 1 = Yes)
Pclass	Passenger Class (1 = 1st; 2 = 2nd; 3 = 3rd)
Name	Name
Married	Married or Non-Married
Age	Age
SibSp	Number of Siblings/Spouses Aboard
Parch	Number of Parents/Children Aboard
Ticket	Ticket Number
Fare	Passenger Fare (British pound)
Cabin	Room
Embarked	Port of Embarkation (C = Cherbourg; Q = Queenstown; S = Southampton)

Our response vairable (y) is the survival. Our random forest model will attempt to predict this variable.

In [13]:
```
from random_forest import RandomForest
```

In [14]:
```
#This is a Helper cell. DO NOT MODIFY CODE IN THIS CELL
from sklearn import preprocessing
import pandas as pd
le = preprocessing.LabelEncoder()

# Loading data into a pandas dataframe
data = pd.read_csv("data/titanic.csv")
N, D = data.shape

# Dealing with NA data.
data = data.fillna(data.mean())

data = data.drop(columns = ['Cabin'])

data.dropna(inplace=True)




# Converting strings to integers: https://stats.stackexchange.com/questions/238530/why-does-decisiont
reeclassifier-require-numeric-variables-for-inputs
# Review the code and print out below.
columns_type_string = list(data.select_dtypes(include=['object']).columns)
print("Column-wise Data Types Before Pre-processing:\n", data.dtypes, sep='')
print("\nColumns with String Type:\n", columns_type_string, sep='')
for column_name in columns_type_string:
    data[column_name] = le.fit_transform(data[column_name].values)
print("\nColumn-wise Data Types After Pre-processing:\n", data.dtypes, sep='')

# Split into X and y.
X = data.drop(columns = ['Survived'])
y = data['Survived']

# Spliting half training half testing.
X_train = np.array(X.iloc[0:int(N * 0.8),])
X_test = np.array(X.iloc[int(N * 0.8):N,])
y_train = np.array(y.iloc[0:int(N * 0.8),])
y_test = np.array(y.iloc[int(N * 0.8):N,])
```

```
Column-wise Data Types Before Pre-processing:
PassengerId      int64
Survived         int64
Pclass           int64
Name            object
Married         object
Age            float64
SibSp            int64
Parch            int64
Ticket          object
Fare           float64
Embarked        object
dtype: object

Columns with String Type:
['Name', 'Married', 'Ticket', 'Embarked']

Column-wise Data Types After Pre-processing:
PassengerId      int64
Survived         int64
Pclass           int64
Name             int64
Married          int64
Age            float64
SibSp            int64
Parch            int64
Ticket           int64
Fare           float64
Embarked         int64
dtype: object
```

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%+). **Once you are satisfied with your chosen parameters, change the default values for max_depth, n_estimators, and max_features in the init 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.**

In [23]:
```
"""
TODO:
n_estimators defines how many decision 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 decision tree.

Tune these three parameters to achieve a better accuracy. While you can use the provided test set to
evaluate your implementation, you will need to obtain 80% on a test set to receive full credit
for this section.
"""
from random_forest import RandomForest
import sklearn.ensemble
n_estimators = 2000 # Only values between 500-2000.
max_depth = 2          # Only values betweeen 2-10.
max_features = 1     # Only values betweeen 0.5-1.0.

random_forest = RandomForest(n_estimators, max_depth, max_features)
random_forest.fit(X_train, y_train)
accuracy=random_forest.OOB_score(X_test, y_test)

print("Random Forest Accuracy: %.4f" % accuracy)
```

```
Random Forest Accuracy: 0.8096
```

4.3 Plotting Feature Importance (2.5pts) **[W]**

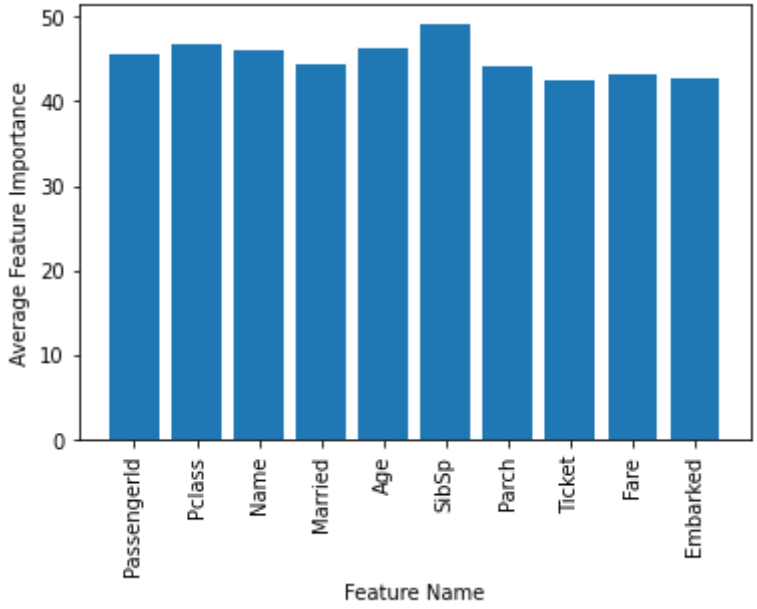
While building tree-based models, it's common to quantify how well splitting on a particular feature in a decision tree helps with predicting the target label in a dataset. Machine learning practicioners typically use "Gini importance", or the (normalized) total reduction in entropy brought by that feature 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 a decision tree weighted by the probability of reaching that split in the decision 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. **The higher the feature importance value, the more important the feature is to predicting the target label.**

Fortunately for us, fitting a sklearn.DecisionTreeClassifier to a dataset auomatically computes the Gini importance for every feature in the decision tree and stores these values in a **feature*importances*** variable. Review the docs for more details on how to access this variable (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier.feature_importances_)

Below we will get the graph the average feature importance for each feature in the random forest model. [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].

In [24]:
```
importance_features = random_forest.average_feature_importance()

plt.figure()
plt.bar(range(len(importance_features)), importance_features, align="center")
plt.xticks(range(len(importance_features)), X.columns.values, rotation=90)
plt.xlabel('Feature Name')
plt.ylabel('Average Feature Importance')
plt.show()
```



In [ ]: