

CS4641 - Summer 2022 Assignment 4 Non-Programming

Shihui Liu

TOTAL POINTS

55 / 55

QUESTION 1

Q1: Neural Nets 10 pts

1.1 Loss Plot and MSE for GD 5 / 5

✓ - 0 pts Correct

- 5 pts Incorrect/No Loss
- 2.5 pts No MSE / Incorrect MSE
- 1 pts MSE is close, but incorrect
- 2.5 pts No plot / Incorrect plot
- 1 pts Plot is close, but incorrect
- 1 pts Incorrectly assigned pages

1.2 Loss Plot and MSE for BGD (BONUS FOR UG) 5 / 5

✓ - 0 pts Correct

- 5 pts No answer / Incorrect
- 1 pts minor error
- 4 pts wrong loss/plot
- 2.5 pts No plot / Incorrect plot
- 2.5 pts No MSE / Incorrect MSE
- 1 pts Plot is close, but incorrect
- 1 pts MSE is close, but incorrect
- 1 pts Incorrectly assigned pages

QUESTION 2

Q2: CNN (BONUS for UG and ALL) 15 pts

2.1 2.1.3 Building the Model (BONUS FOR UG) 2 / 2

✓ - 0 pts Correct

- 2 pts Missing

2.2 2.1.4 CNN Accuracy (BONUS FOR UG) 8 / 8

✓ + 8 pts \$\$\geq 80\% \text{ accuracy}

- + 6 pts 70-80% accuracy
- + 3 pts 60-70% accuracy
- + 1 pts \$\$\leq 60\% \text{ accuracy}
- + 0 pts Not Completed

2.3 2.1.5 Accuracy & Loss Plots (BONUS FOR UG) 2 / 2

✓ - 0 pts All 3 graphs plotted

- 1 pts Incorrect Plots
- 1 pts Incorrect Confusion Matrix
- 2 pts Incorrect Plots and Confusion Matrix
- 1 pts Missing Confusion Matrix
- 2 pts Missing

2.4 2.2 ResNet Paper (BONUS FOR ALL) 3 / 3

✓ + 3 pts Correct

- + 1.5 pts Unclear explanation
- + 0 pts No answer

QUESTION 3

Q3: Random Forest (BONUS FOR ALL) 10 pts

3.1 3.3 Sorted Feature Importance (BONUS FOR ALL) 5 / 5

✓ - 0 pts Correct

- 5 pts No graph present
- 1 pts No explanation/comment

3.2 3.4 Improvement (BONUS FOR ALL) 5 / 5

✓ - 0 pts Correct

- 5 pts No answer
- 2 pts Missing possible improvements
- 3 pts Missing potential causes
- 1 pts Need more clarification for 3.4.1

- 1 pts Need more clarification for 3.4.2

QUESTION 4

4 Q4.1: SVM by hand (BONUS FOR ALL) 20 /

20

✓ - 0 pts Correct

- 20 pts 4.1 No answer

4.1 One or more questions wrong or unanswered.

- 2 pts 1 incorrect or missing

- 4 pts 2 incorrect or missing

- 4 pts 3 incorrect or missing

- 4 pts 4 incorrect or missing

- 4 pts 5 incorrect or missing

- 2 pts 6 incorrect or missing

- 4 pts (4) Incorrect

- 2 pts Partial Credit 1

- 2 pts Partial Credit 2

- 2 pts Partial Credit 4

- 2 pts Partial Credit 5

```
# 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)
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)
print('Your batch_y works within the expected range:', batch_y_test)

Loss after iteration 0 : 7.573928445695169
Loss after iteration 1 : 17.098338048204653
Loss after iteration 2 : 21.75725773432647

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

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

1.2 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 [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]

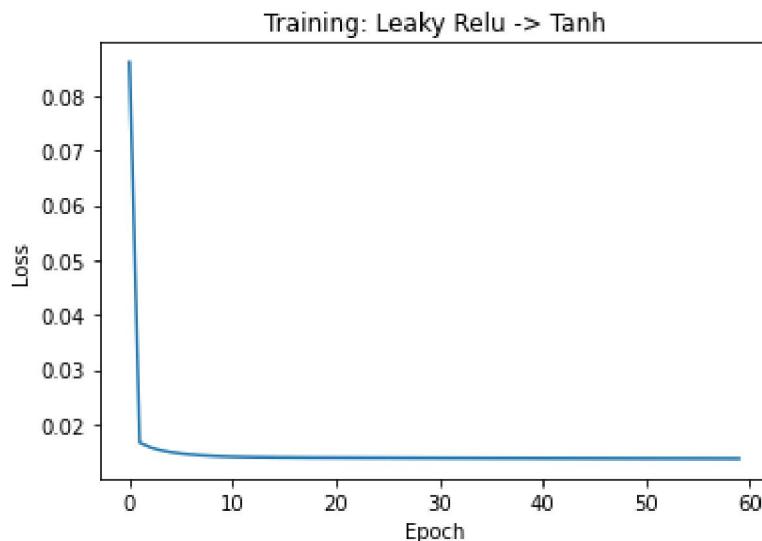
x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=1) #split

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),  
  
nn = dlnet(x_train,y_train,lr=0.01) # initialize 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[18]: Text(0, 0.5, 'Loss')



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) 0.03620883633320266

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 [282...]: #####
DO NOT CHANGE THIS CELL ###
#####

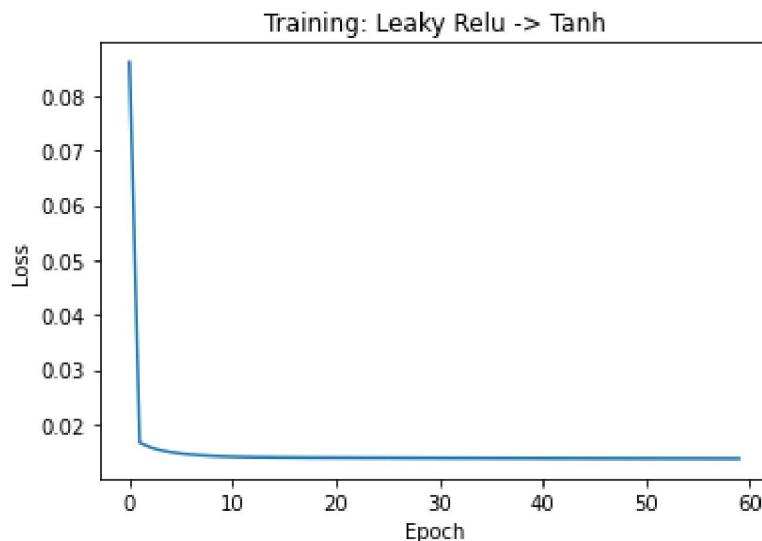
```
from NN import dlnet  
  
dataset = load_diabetes() # load the dataset  
x, y = dataset.data, dataset.target  
y = y.reshape(-1,1)
```

1.1 Loss Plot and MSE for GD 5 / 5

- ✓ - **0 pts** Correct
- **5 pts** Incorrect/No Loss
- **2.5 pts** No MSE / Incorrect MSE
- **1 pts** MSE is close, but incorrect
- **2.5 pts** No plot / Incorrect plot
- **1 pts** Plot is close, but incorrect
- **1 pts** Incorrectly assigned pages

```
x_train, x_test, y_train, y_test = x_train.T, x_test.T, y_train.reshape(1,-1),  
  
nn = dlnet(x_train,y_train,lr=0.01) # initialize 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[18]: Text(0, 0.5, 'Loss')



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) 0.03620883633320266

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 [282...]: #####
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) #splitting data

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

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

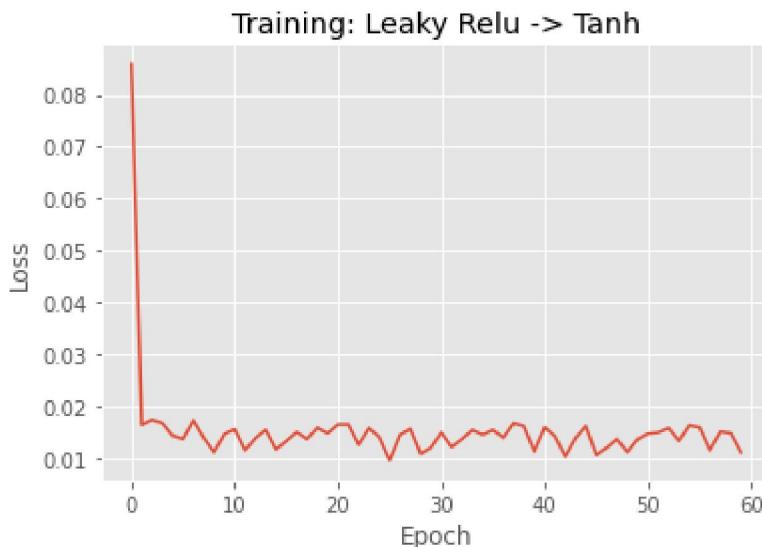
x_train, x_test, y_train, y_test = x_train.T, x_test.T, y_train.reshape(1,-1), y_test.reshape(1,-1)

nn = dlnet(x_train,y_train,lr=0.01) # initialize 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")
```

```
Loss after iteration 0 : 0.08589618997003594
Loss after iteration 1000 : 0.016408346537966557
Loss after iteration 2000 : 0.017377191872833084
Loss after iteration 3000 : 0.01671742017234101
Loss after iteration 4000 : 0.014299328573566396
Loss after iteration 5000 : 0.013670623272153234
Loss after iteration 6000 : 0.01727388842560422
Loss after iteration 7000 : 0.013904127782939469
Loss after iteration 8000 : 0.01112161065927389
Loss after iteration 9000 : 0.01463350207507137
Loss after iteration 10000 : 0.01553485054797452
Loss after iteration 11000 : 0.01151543656687773
Loss after iteration 12000 : 0.013827461223577258
Loss after iteration 13000 : 0.01540299263801511
Loss after iteration 14000 : 0.011658814611871236
Loss after iteration 15000 : 0.01329896807897343
Loss after iteration 16000 : 0.014965115921222858
Loss after iteration 17000 : 0.013651649141060108
Loss after iteration 18000 : 0.015843591998427765
Loss after iteration 19000 : 0.014785872998293759
Loss after iteration 20000 : 0.01646337572247266
Loss after iteration 21000 : 0.016481930473880385
Loss after iteration 22000 : 0.012612394150527378
Loss after iteration 23000 : 0.015772681925901595
Loss after iteration 24000 : 0.014003620201645802
Loss after iteration 25000 : 0.009581157045982537
Loss after iteration 26000 : 0.014434906294412922
Loss after iteration 27000 : 0.015644399708363232
Loss after iteration 28000 : 0.010839396726325098
Loss after iteration 29000 : 0.011996795993274953
Loss after iteration 30000 : 0.014938138321297681
Loss after iteration 31000 : 0.012162912721009054
Loss after iteration 32000 : 0.01364023827661857
Loss after iteration 33000 : 0.015404765608487626
Loss after iteration 34000 : 0.014512317143042195
Loss after iteration 35000 : 0.015420175230938776
Loss after iteration 36000 : 0.01392154992120026
Loss after iteration 37000 : 0.016699342225687665
Loss after iteration 38000 : 0.016165669176525434
Loss after iteration 39000 : 0.011267322767442085
Loss after iteration 40000 : 0.015911910954065602
Loss after iteration 41000 : 0.01421346558734082
Loss after iteration 42000 : 0.010280584053043839
Loss after iteration 43000 : 0.013881268766920637
Loss after iteration 44000 : 0.01617797532795972
Loss after iteration 45000 : 0.010586515862584037
Loss after iteration 46000 : 0.01198802000509644
Loss after iteration 47000 : 0.013655336914950253
Loss after iteration 48000 : 0.011156243398522487
Loss after iteration 49000 : 0.013620986648265488
Loss after iteration 50000 : 0.014682232860550686
Loss after iteration 51000 : 0.014920461224286445
Loss after iteration 52000 : 0.015768308816500164
Loss after iteration 53000 : 0.01329874263969875
Loss after iteration 54000 : 0.01631832492231257
Loss after iteration 55000 : 0.015912420922919036
Loss after iteration 56000 : 0.011519288294398453
Loss after iteration 57000 : 0.015080519251050174
Loss after iteration 58000 : 0.014804697153357279
Loss after iteration 59000 : 0.011133597080553847
```

Out [282]: Text(0, 0.5, 'Loss')



In [283...]

```
#####
### 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) 0.03620392966320903

2: Image Classification based on Convolutional Neural Networks [15pts; 12pts Bonus for Undergrad + 3pts Bonus for all] ** [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 \(Layers used to build ConvNets\)](#)

1.2 Loss Plot and MSE for BGD (BONUS FOR UG) 5 / 5

✓ - 0 pts Correct

- 5 pts No answer / Incorrect

- 1 pts minor error

- 4 pts wrong loss/plot

- 2.5 pts No plot / Incorrect plot

- 2.5 pts No MSE / Incorrect MSE

- 1 pts Plot is close, but incorrect

- 1 pts MSE is close, but incorrect

- 1 pts Incorrectly assigned pages

```
batch_size, epochs, init_lr = net.get_vars()
print(f'Batch Size\t: {batch_size} \nEpochs\t\t: {epochs} \nLearning Rate\t: {i
Batch Size      : 64
Epochs         : 20
Learning Rate   : 0.001
```

Defining model [2pts Bonus for Undergrad]

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 [81]: #####
### 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 (MaxPooling2D)	(None, 8, 8, 64)	0
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

Compiling model [No Points]

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

In [82]: #####

2.1 2.1.3 Building the Model (BONUS FOR UG) 2 / 2

✓ - 0 pts Correct

- 2 pts Missing

```
### DO NOT CHANGE THIS CELL ###
#####
# Complete compile_model() in cnn.py.
from cnn import CNN
net = CNN()
model = net.compile_net(model)
print(model)

<tensorflow.python.keras.engine.sequential.Sequential object at 0x7fda02590f90
>
```

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 80% accuracy on the test set to get full points. If you achieve accuracy between 60% to 70%, you will only get 3 points. An accuracy between 70% to 80% will earn an additional 3pts.

- 60% to 70% earns 3pts
- 70% to 80% earns 3pts more (6pts total)
- 80%+ earns 2pts more (8pts total)

Train your own CNN model

```
In [83]: #####
### DO NOT CHANGE THIS CELL ###
#####

from cnn import CNN

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

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

history = model.fit(
    x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
    callbacks=[tf.keras.callbacks.LearningRateScheduler(lr_scheduler)],
    shuffle=True,
    verbose=1,
    initial_epoch=0,
    validation_data=(x_test, y_test)
)
```

```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Train on 50000 samples, validate on 10000 samples
Learning rate: 0.001
Epoch 1/20
50000/50000 [=====] - 86s 2ms/sample - loss: 1.5864 -
accuracy: 0.4202 - val_loss: 1.1512 - val_accuracy: 0.5896
Learning rate: 0.0009000000000000001
Epoch 2/20
50000/50000 [=====] - 83s 2ms/sample - loss: 1.1470 -
accuracy: 0.5956 - val_loss: 0.9350 - val_accuracy: 0.6731
Learning rate: 0.0008100000000000001
Epoch 3/20
50000/50000 [=====] - 82s 2ms/sample - loss: 0.9834 -
accuracy: 0.6591 - val_loss: 0.8588 - val_accuracy: 0.7011
Learning rate: 0.0007290000000000002
Epoch 4/20
50000/50000 [=====] - 80s 2ms/sample - loss: 0.8809 -
accuracy: 0.6929 - val_loss: 0.8034 - val_accuracy: 0.7196
Learning rate: 0.0006561000000000001
Epoch 5/20
50000/50000 [=====] - 82s 2ms/sample - loss: 0.8194 -
accuracy: 0.7175 - val_loss: 0.7274 - val_accuracy: 0.7486
Learning rate: 0.00059049
Epoch 6/20
50000/50000 [=====] - 86s 2ms/sample - loss: 0.7653 -
accuracy: 0.7343 - val_loss: 0.6960 - val_accuracy: 0.7566
Learning rate: 0.000531441
Epoch 7/20
50000/50000 [=====] - 82s 2ms/sample - loss: 0.7217 -
accuracy: 0.7498 - val_loss: 0.6789 - val_accuracy: 0.7621
Learning rate: 0.0004782969000000001
Epoch 8/20
50000/50000 [=====] - 82s 2ms/sample - loss: 0.6936 -
accuracy: 0.7589 - val_loss: 0.6757 - val_accuracy: 0.7686
Learning rate: 0.000430467210000001
Epoch 9/20
50000/50000 [=====] - 82s 2ms/sample - loss: 0.6557 -
accuracy: 0.7733 - val_loss: 0.6575 - val_accuracy: 0.7764
Learning rate: 0.0003874204890000001
Epoch 10/20
50000/50000 [=====] - 82s 2ms/sample - loss: 0.6327 -
accuracy: 0.7808 - val_loss: 0.6500 - val_accuracy: 0.7803
Learning rate: 0.0003486784401000001
Epoch 11/20
50000/50000 [=====] - 82s 2ms/sample - loss: 0.6080 -
accuracy: 0.7856 - val_loss: 0.6458 - val_accuracy: 0.7796
Learning rate: 0.0003138105960900001
Epoch 12/20
50000/50000 [=====] - 83s 2ms/sample - loss: 0.5909 -
accuracy: 0.7947 - val_loss: 0.6311 - val_accuracy: 0.7832
Learning rate: 0.0002824295364810001
Epoch 13/20
50000/50000 [=====] - 86s 2ms/sample - loss: 0.5684 -
accuracy: 0.8013 - val_loss: 0.6265 - val_accuracy: 0.7872
Learning rate: 0.0002541865828329001
Epoch 14/20
50000/50000 [=====] - 86s 2ms/sample - loss: 0.5615 -
accuracy: 0.8053 - val_loss: 0.6128 - val_accuracy: 0.7888
Learning rate: 0.0002287679245496101
Epoch 15/20
50000/50000 [=====] - 82s 2ms/sample - loss: 0.5420 -
```

```

accuracy: 0.8105 - val_loss: 0.6131 - val_accuracy: 0.7931
Learning rate: 0.0002058911320946491
Epoch 16/20
50000/50000 [=====] - 87s 2ms/sample - loss: 0.5299 -
accuracy: 0.8134 - val_loss: 0.6097 - val_accuracy: 0.7932
Learning rate: 0.00018530201888518417
Epoch 17/20
50000/50000 [=====] - 83s 2ms/sample - loss: 0.5167 -
accuracy: 0.8216 - val_loss: 0.6379 - val_accuracy: 0.7870
Learning rate: 0.00016677181699666576
Epoch 18/20
50000/50000 [=====] - 83s 2ms/sample - loss: 0.5070 -
accuracy: 0.8241 - val_loss: 0.6128 - val_accuracy: 0.7937
Learning rate: 0.00015009463529699917
Epoch 19/20
50000/50000 [=====] - 82s 2ms/sample - loss: 0.5034 -
accuracy: 0.8234 - val_loss: 0.6053 - val_accuracy: 0.7951
Learning rate: 0.0001350851717672993
Epoch 20/20
50000/50000 [=====] - 92s 2ms/sample - loss: 0.4927 -
accuracy: 0.8279 - val_loss: 0.5939 - val_accuracy: 0.8003
Test loss: 0.5939190960407257
Test accuracy: 0.8003

```

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 [84]:

```

#####
### 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'])

```

2.2 2.1.4 CNN Accuracy (BONUS FOR UG) 8 / 8

✓ + 8 pts \$\$\geq 80\% \text{ accuracy}

+ 6 pts 70-80% accuracy

+ 3 pts 60-70% accuracy

+ 1 pts < 60% accuracy

+ 0 pts Not Completed

```

accuracy: 0.8105 - val_loss: 0.6131 - val_accuracy: 0.7931
Learning rate: 0.0002058911320946491
Epoch 16/20
50000/50000 [=====] - 87s 2ms/sample - loss: 0.5299 -
accuracy: 0.8134 - val_loss: 0.6097 - val_accuracy: 0.7932
Learning rate: 0.00018530201888518417
Epoch 17/20
50000/50000 [=====] - 83s 2ms/sample - loss: 0.5167 -
accuracy: 0.8216 - val_loss: 0.6379 - val_accuracy: 0.7870
Learning rate: 0.00016677181699666576
Epoch 18/20
50000/50000 [=====] - 83s 2ms/sample - loss: 0.5070 -
accuracy: 0.8241 - val_loss: 0.6128 - val_accuracy: 0.7937
Learning rate: 0.00015009463529699917
Epoch 19/20
50000/50000 [=====] - 82s 2ms/sample - loss: 0.5034 -
accuracy: 0.8234 - val_loss: 0.6053 - val_accuracy: 0.7951
Learning rate: 0.0001350851717672993
Epoch 20/20
50000/50000 [=====] - 92s 2ms/sample - loss: 0.4927 -
accuracy: 0.8279 - val_loss: 0.5939 - val_accuracy: 0.8003
Test loss: 0.5939190960407257
Test accuracy: 0.8003

```

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 [84]:

```

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

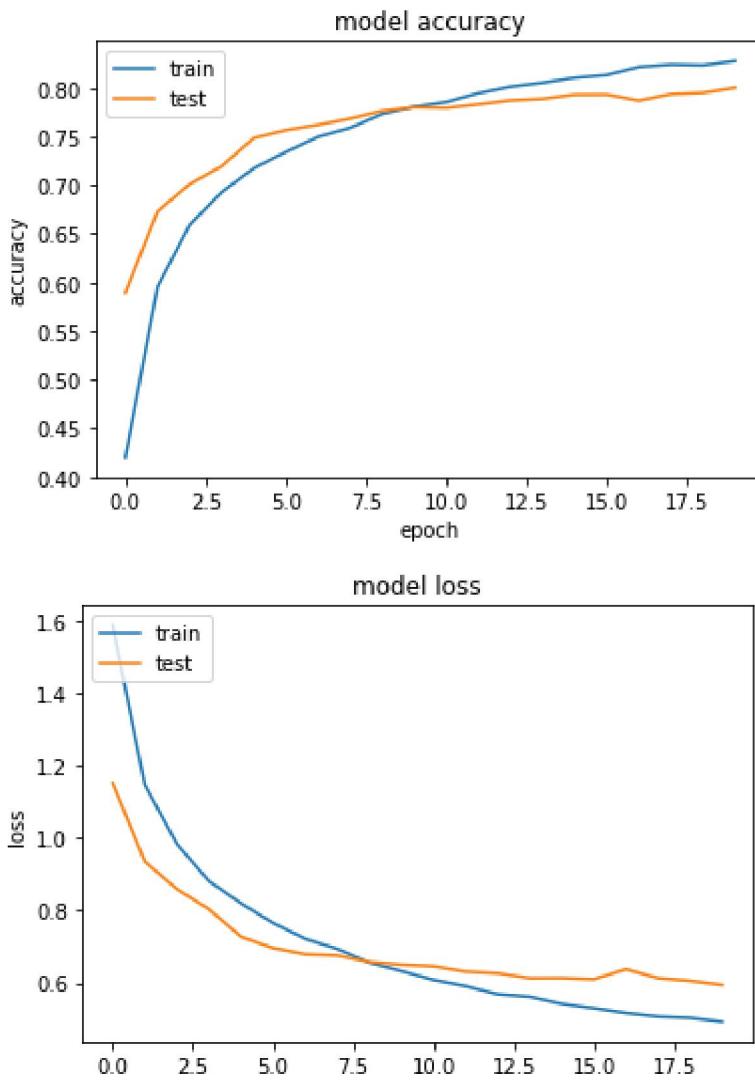
# list all data in history
print(history.history.keys())

# 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'])

```

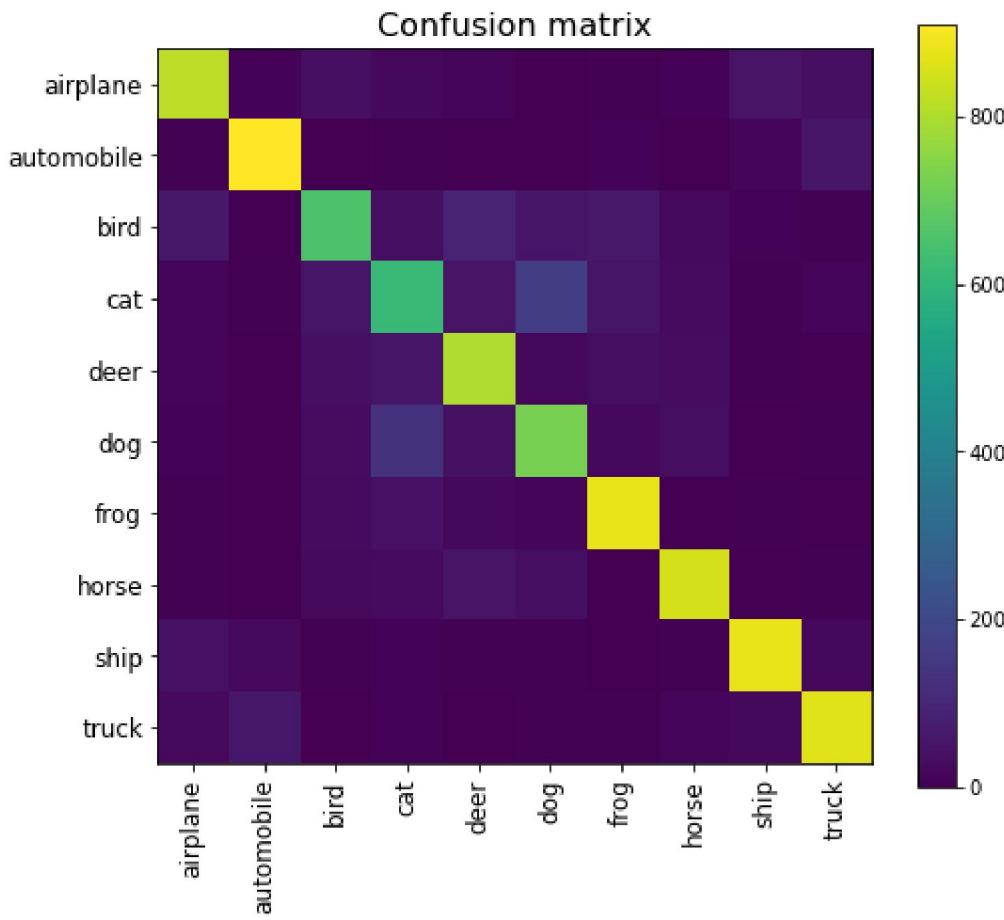


In [85]:

```
#####
### 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).

2.3 2.1.5 Accuracy & Loss Plots (BONUS FOR UG) 2 / 2

✓ - **0 pts** All 3 graphs plotted

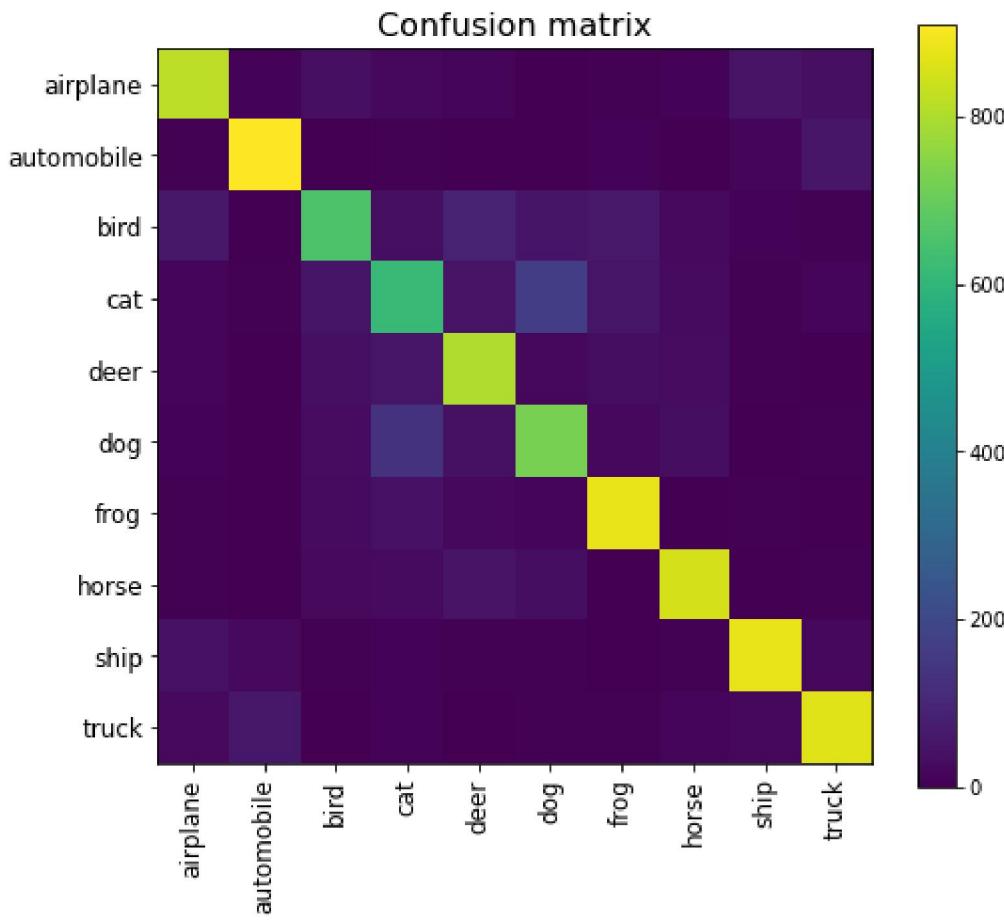
- **1 pts** Incorrect Plots

- **1 pts** Incorrect Confusion Matrix

- **2 pts** Incorrect Plots and Confusion Matrix

- **1 pts** Missing Confusion Matrix

- **2 pts** Missing



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: \ Vanishing gradient occurs during the back propagation process when the partial derivative of the error functions are multiplied by chain rule. For a deep model with a lot of layers, the product of the chain rule could become so small that it approaches zero. \ Resnet uses shortcut connections("highway networks") which allows one or more layers to be skipped during the back propagation process. This method reduces the number of chain rules that are performed on the gradient, which avoids the vanishing gradient problem.

3: Random Forests [50pts; 40pts + 10pts Bonus for All] **[P]** **[W]**

NOTE: Please use sklearn's DecisionTreeClassifier in your Random Forest implementation.
[You can find more details about this classifier here.](#)

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

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' (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 decision 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 (70% 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 a decision 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.

2.4 2.2 ResNet Paper (BONUS FOR ALL) 3 / 3

✓ + 3 pts Correct

+ 1.5 pts Unclear explanation

+ 0 pts No answer

```
accuracy=random_forest.oob_score(x_test, y_test)

print("accuracy: %.4f" % accuracy)

accuracy: 0.7536
```

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 **75% 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 a decision tree helps with predicting the target label in a dataset. Machine learning practitioners typically use "Gini importance", or the (normalized) total reduction in entropy brought by that feature to evaluate how important that feature is for predicting the target variable.

Gini importance is typically calculated as the reduction in entropy from reaching a split in 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.

Let's think about what this metric means with an example. A high probability of reaching a split on "Age" in a decision tree trained on our patient dataset (many samples will reach this split for a decision) and a large reduction in entropy from splitting on "Age" will result in a high feature importance value for "Age". This could mean "Age" is a very important feature for predicting a patients probability of disease. On the other hand, a low probability of reaching a split on "Cholesterol (chol)" in a decision tree (few samples will reach this split for a decision) and a low reduction in entropy from splitting on "Cholesterol (chol)" will result in a low feature importance value. This could mean "Cholesterol (chol)" is not a very informative feature for predicting a patients probability of disease in our decision 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.DecisionTreeClassifier` to a dataset automatically 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](#)

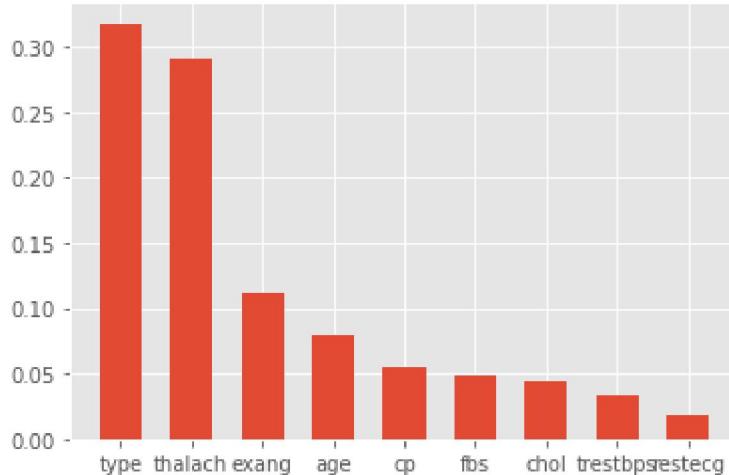
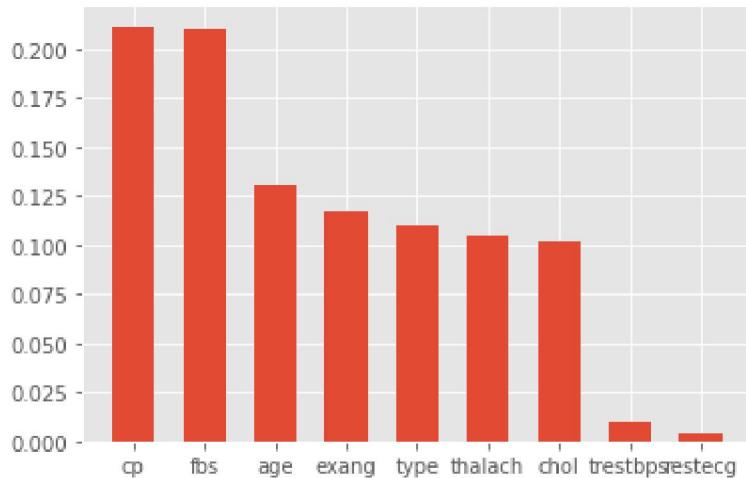
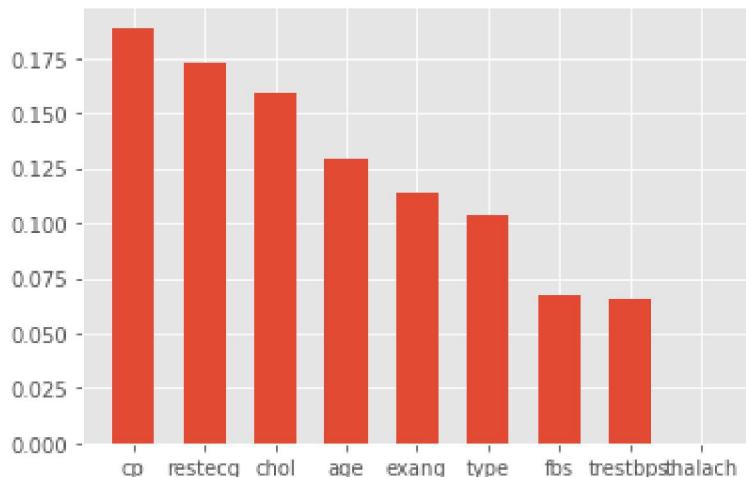
In the `random_forest.py` file, complete the following function:

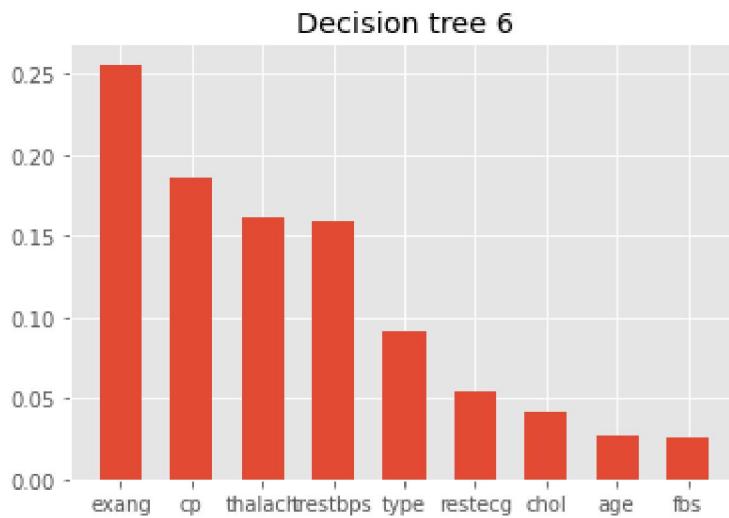
- **plot_feature_importance**: Make sure to sort the bars in descending order and remove any features with feature importance of 0

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 decision tree in your tuned random forest from Q3.2.

```
In [315]: # TODO: Complete plot_feature_importance() in random_forest.py  
random_forest.plot_feature_importance(data_train)
```



Decision tree 2**Decision tree 3****Decision tree 4**



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]**

For this question, we only ask that the Random Forest model have an accuracy of at least 75%. In the real world, we would not be satisfied with this accuracy and would most likely not immediately deploy this model to identify narrowing arteries.

Please answer the following two questions:

1. What are some potential causes for why a Random Forest decision tree model may not produce high accuracies?
2. What are some ways to improve on these potential causes?

Answer:

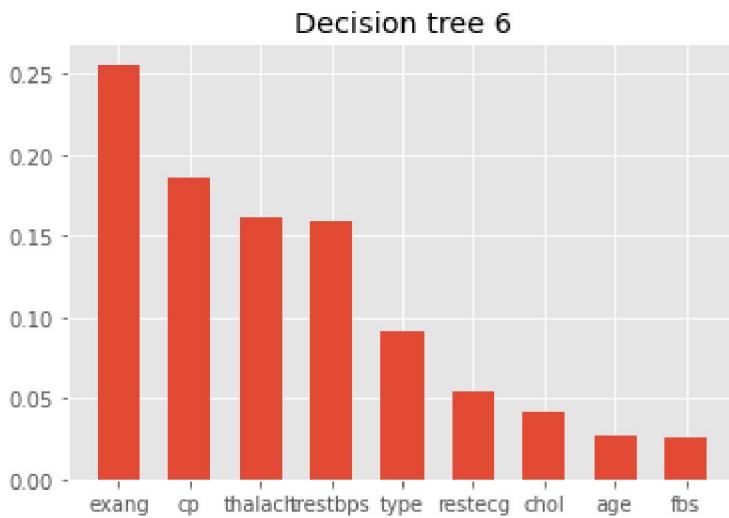
1. What are some potential causes for why a Random Forest decision tree model may not produce high accuracies?

3.1 3.3 Sorted Feature Importance (BONUS FOR ALL) 5 / 5

✓ - **0 pts** Correct

- **5 pts** No graph present

- **1 pts** No explanation/comment



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]**

For this question, we only ask that the Random Forest model have an accuracy of at least 75%. In the real world, we would not be satisfied with this accuracy and would most likely not immediately deploy this model to identify narrowing arteries.

Please answer the following two questions:

1. What are some potential causes for why a Random Forest decision tree model may not produce high accuracies?
2. What are some ways to improve on these potential causes?

Answer:

1. What are some potential causes for why a Random Forest decision tree model may not produce high accuracies?

The first potential cause is the extrapolation problem. Random forest builds its trees based on the feature space of the training data. Therefore, when making a prediction, the Random forest model is unable to make correct predictions that requires extrapolating outside of the value range of its training data. This problem tends to show up when dealing with linear datasets.

A second potential cause is when the data is sparse. Sparse data is data that contain mostly zeros. When there are sparse features, zero features could be randomly chosen when splitting, which could lead to errors and inaccurate predictions when making predictions.

1. What are some ways to improve on these potential causes?

To solve the extrapolation problem, we could limit the predictions to only the feature space of the training data, or we could use models that are capable of extrapolating, like linear regression for instance.

To solve the sparse data problem, we could set our random forest to only make splits on features that are of importance/informative. This eliminates the splitting on zero feature problem.

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

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

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

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

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

The max margin classifier has the formulation,

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

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

3.2 3.4 Improvement (BONUS FOR ALL) 5 / 5

✓ - 0 pts Correct

- 5 pts No answer

- 2 pts Missing possible improvements

- 3 pts Missing potential causes

- 1 pts Need more clarification for 3.4.1

- 1 pts Need more clarification for 3.4.2

The first potential cause is the extrapolation problem. Random forest builds its trees based on the feature space of the training data. Therefore, when making a prediction, the Random forest model is unable to make correct predictions that requires extrapolating outside of the value range of its training data. This problem tends to show up when dealing with linear datasets.

A second potential cause is when the data is sparse. Sparse data is data that contain mostly zeros. When there are sparse features, zero features could be randomly chosen when splitting, which could lead to errors and inaccurate predictions when making predictions.

1. What are some ways to improve on these potential causes?

To solve the extrapolation problem, we could limit the predictions to only the feature space of the training data, or we could use models that are capable of extrapolating, like linear regression for instance.

To solve the sparse data problem, we could set our random forest to only make splits on features that are of importance/informative. This eliminates the splitting on zero feature problem.

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

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

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

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

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

The max margin classifier has the formulation,

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

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

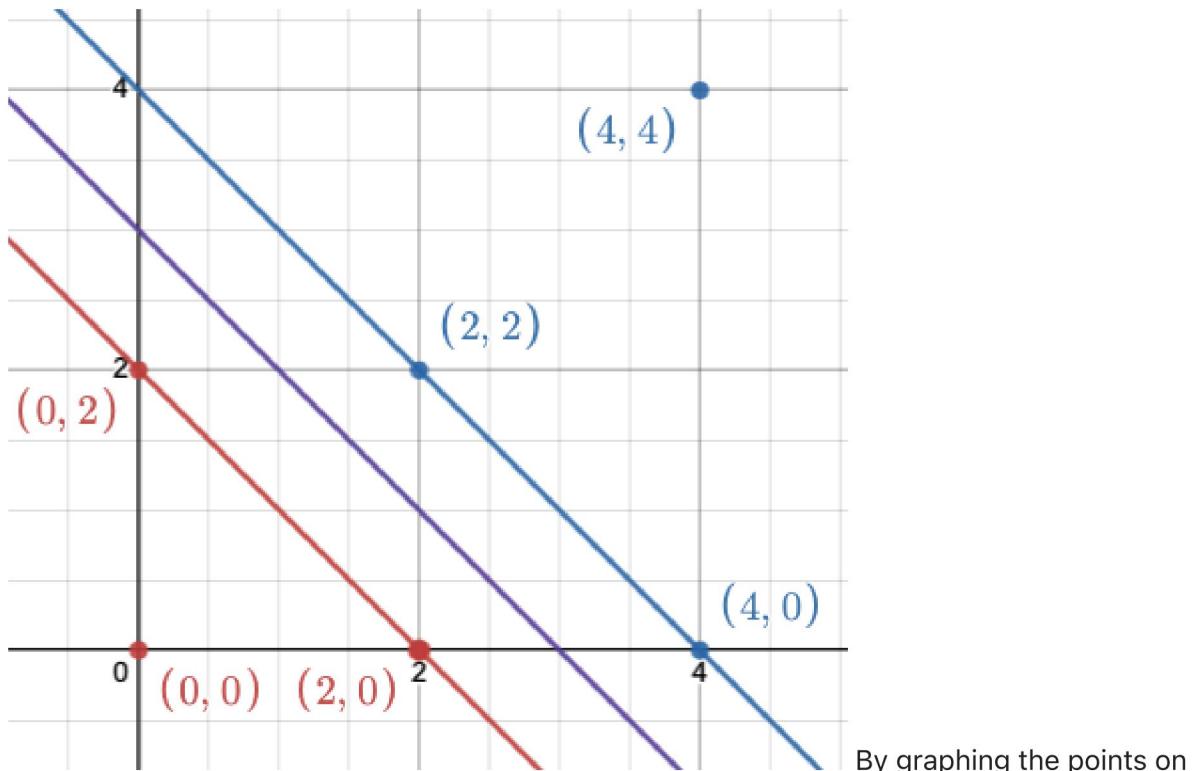
Hint: \mathbf{x}_i are the support vectors. Margin is equal to $\frac{1}{\|\theta\|}$ and full margin is equal to $\frac{2}{\|\theta\|}$.

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 $\mathbf{x} = (4, 2)$, $y = -1$ change the separability? (2 pts)

Yes, the points are linearly separable. Adding the point $\mathbf{x} = (4, 2)$, $y = -1$ will make the points **not** linearly separable

- (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)



By graphing the points on a 2-D plane, we observe that the points with label $y = -1$ has a constraint of $g(x) = x_1 + x_2 - 2 = 0$ that passes through $(0, 2)$ and $(2, 0)$, and the points with label $y = 1$ has a constraint of $g(x) = x_1 + x_2 - 4 = 0$ that passes through $(2, 2)$ and $(4, 0)$. The separating hyperlane using max-margin formulation, therefore, should have a equal distance from the two constraint for the margin to be maximized. The separating hyperplane is therefore is the midway between the two constraints: $x_1 + x_2 - 3 = 0$

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

From the answer above, we know that $\theta = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. A vector parallel to θ is its unit vector:

$$\begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

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

$$\frac{1}{||\theta||} = \frac{1}{\sqrt{1^2 + 1^2}} = \frac{1}{\sqrt{2}}$$

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

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

(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) \ The following points affect the SVM solution, since they define the constraints as demonstrated in (2):

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

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 [287]: #####
### 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,))
```

4 Q4.1: SVM by hand (BONUS FOR ALL) 20 / 20

✓ - 0 pts Correct

- 20 pts 4.1 No answer

4.1 One or more questions wrong or unanswered.

- 2 pts 1 incorrect or missing

- 4 pts 2 incorrect or missing

- 4 pts 3 incorrect or missing

- 4 pts 4 incorrect or missing

- 4 pts 5 incorrect or missing

- 2 pts 6 incorrect or missing

- 4 pts (4) Incorrect

- 2 pts Partial Credit 1

- 2 pts Partial Credit 2

- 2 pts Partial Credit 4

- 2 pts Partial Credit 5

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

Instructor: Dr. Mahdi Roodzabani

Deadline: Monday, July 25, 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 `` 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.**

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 [65 pts; 50pts + 15pts Undergrad Bonus]

Deliverables: **NN.py and Notebook Graphs**

- **1.1 NN Implementation** [55pts; 45pts + 10pts **Bonus for Undergrad**] - *programming*
 - Leaky relu [5pts]
 - tanh [5pts]
 - loss [5pts]
 - forward [10pts]
 - backward [10pts]
 - Gradient Descent [10pts]
 - Batch Gradient Descent [10pts **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*

Q2: CNN [15pts; 12pts Bonus for Undergrad + 3pts Bonus for All]

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

- **2.1 Image Classification using Keras CNN** [12pts **Bonus for Undergrad**]
 - 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

Version information
python: 3.7.13 (default, Mar 28 2022, 07:24:34)
[Clang 12.0.0 ]
matplotlib: 3.5.1
numpy: 1.21.5

```

1: Two Layer Neural Network [65pts; 50pts + 15pts 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 $\phi(\cdot)$ 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 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

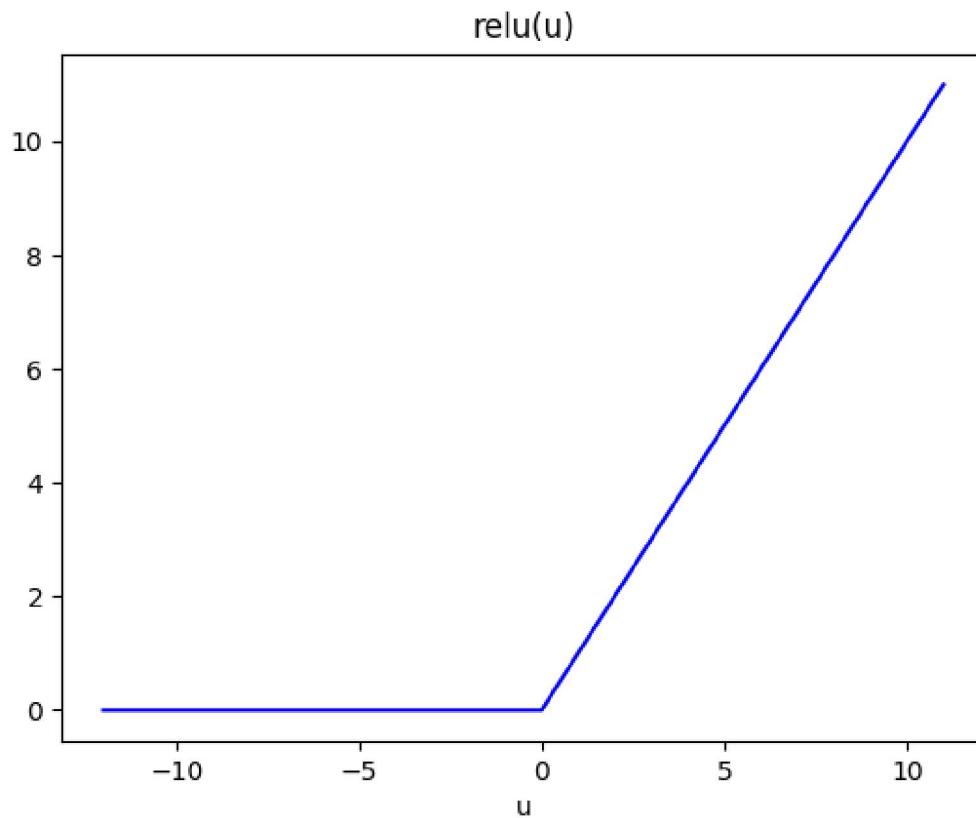
$$o = \phi(u) = \max(0, u)$$

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 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.

Here in our homework, we are going to implement leaky relu.



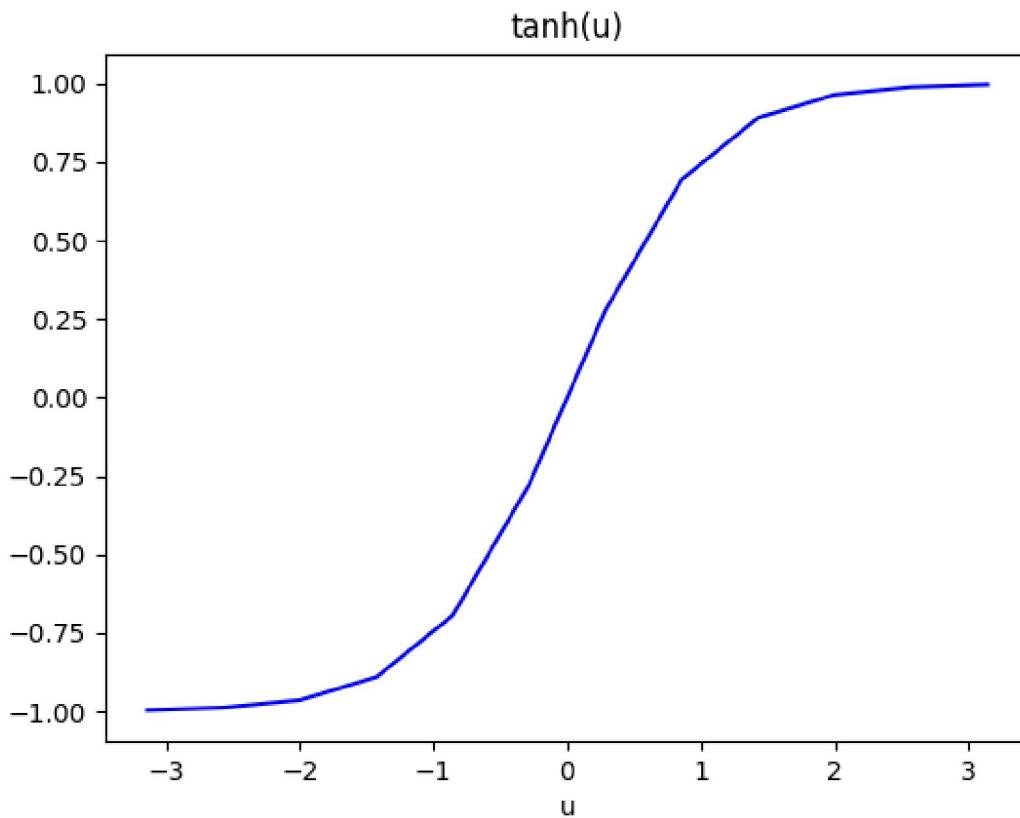
Tanh

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

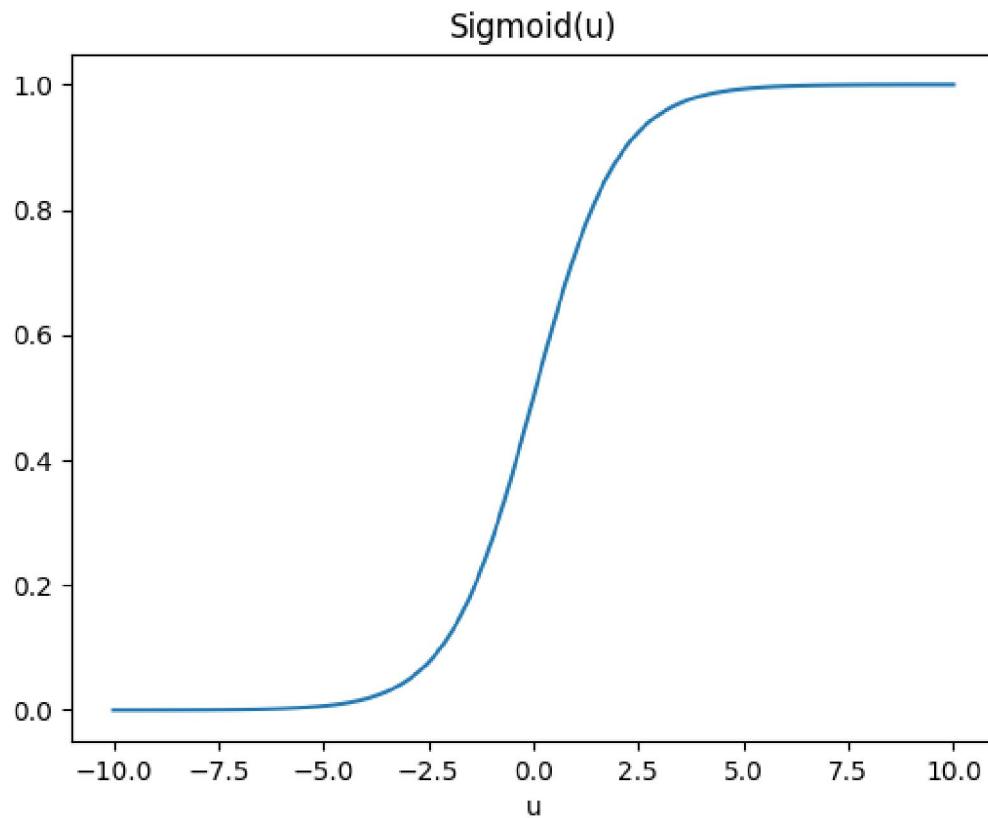
The sigmoid function is another non-linear function with S-shaped curve. This function is useful in the case of binary classification as its output is between 0 and 1. The mathematical form of the function is

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

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:

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

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

Forward Propagation

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). 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.

$$u^{[0]} = x \quad (1)$$

$$u^{[1]} = \theta^{[1]} u^{[0]} + b^{[1]} \quad (2)$$

$$o^{[1]} = \text{LeakyRelu}(u^{[1]}) \quad (3)$$

$$u^{[2]} = \theta^{[2]} o^{[1]} + b^{[2]} \quad (4)$$

$$\hat{y} = o^{[2]} = \text{Tanh}(u^{[2]}) \quad (5)$$

Then we get the output and compute the loss

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

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

$$\theta^{[2]} := \theta^{[2]} - lr \times \frac{\partial l}{\partial \theta^{[2]}} \quad (6)$$

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

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

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

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:

$$\frac{\partial l}{\partial \theta^{[2]}} = \frac{\partial l}{\partial o^{[2]}} \frac{\partial o^{[2]}}{\partial u^{[2]}} \frac{\partial u^{[2]}}{\partial \theta^{[2]}} \quad (7)$$

$$\frac{\partial l}{\partial b^{[2]}} = \frac{\partial l}{\partial o^{[2]}} \frac{\partial o^{[2]}}{\partial u^{[2]}} \frac{\partial u^{[2]}}{\partial b^{[2]}} \quad (8)$$

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

$$\frac{\partial l}{\partial o^{[2]}} = (o^{[2]} - y) \quad (9)$$

$$\frac{\partial l}{\partial u^{[2]}} = \frac{\partial l}{\partial o^{[2]}} * (1 - (\tanh(u^{[2]}))^2) \quad (10)$$

$$\frac{\partial u^{[2]}}{\partial \theta^{[2]}} = o^{[1]} \quad (11)$$

$$\frac{\partial u^{[2]}}{\partial b^{[2]}} = 1 \quad (12)$$

On vectorization, the above equations become:

$$\frac{\partial l}{\partial o^{[2]}} = \frac{1}{n} (o^{[2]} - y) \quad (13)$$

$$\frac{\partial l}{\partial \theta^{[2]}} = \frac{1}{n} \frac{\partial l}{\partial u^{[2]}} o^{[1]} \quad (14)$$

$$\frac{\partial l}{\partial b^{[2]}} = \frac{1}{n} \sum \frac{\partial l}{\partial u^{[2]}} \quad (15)$$

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:

$$\frac{\partial l}{\partial \theta^{[1]}} = \frac{\partial l}{\partial o^{[2]}} \frac{\partial o^{[2]}}{\partial u^{[2]}} \frac{\partial u^{[2]}}{\partial o^{[1]}} \frac{\partial o^{[1]}}{\partial u^{[1]}} \frac{\partial u^{[1]}}{\partial \theta^{[1]}} \quad (16)$$

$$\frac{\partial l}{\partial b^{[1]}} = \frac{\partial l}{\partial o^{[2]}} \frac{\partial o^{[2]}}{\partial u^{[2]}} \frac{\partial u^{[2]}}{\partial o^{[1]}} \frac{\partial o^{[1]}}{\partial u^{[1]}} \frac{\partial u^{[1]}}{\partial b^{[1]}} \quad (17)$$

Where

$$\frac{\partial u^{[2]}}{\partial o^{[1]}} = \theta^{[2]} \quad (18)$$

$$\frac{\partial o^{[1]}}{\partial u^{[1]}} = 1(o^{[1]} > 0) \text{ and } \alpha(o^{[1]} \leq 0) \quad (19)$$

$$\frac{\partial u^{[1]}}{\partial \theta^{[1]}} = x \quad (20)$$

$$\frac{\partial u^{[1]}}{\partial b^{[1]}} = 1 \quad (21)$$

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

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:

$$dLoss_o2 = \frac{\partial l}{\partial o^{[2]}} \implies \text{dim} = (1, 331) \quad (22)$$

$$dLoss_u2 = dLoss_o2 \frac{\partial o^{[2]}}{\partial u^{[2]}} \implies \text{dim} = (1, 331) \quad (23)$$

$$dLoss_theta2 = dLoss_u2 \frac{\partial u^{[2]}}{\partial \theta^{[2]}} \implies \text{dim} = (1, 15) \quad (24)$$

$$dLoss_b2 = dLoss_u2 \frac{\partial u^{[2]}}{\partial b^{[2]}} \implies \text{dim} = (1, 1) \quad (25)$$

$$dLoss_o1 = dLoss_u2 \frac{\partial u^{[2]}}{\partial o^{[1]}} \implies \text{dim} = (15, 331) \quad (26)$$

$$dLoss_u1 = dLoss_o1 \frac{\partial o^{[1]}}{\partial u^{[1]}} \implies \text{dim} = (15, 331) \quad (27)$$

$$dLoss_theta1 = dLoss_u1 \frac{\partial u^{[1]}}{\partial \theta^{[1]}} \implies \text{dim} = (15, 10) \quad (28)$$

$$dLoss_b1 = dLoss_u1 \frac{\partial u^{[1]}}{\partial b^{[1]}} \implies \text{dim} = (15, 1) \quad (29)$$

Note: Training set has 331 examples.

1.1 NN Implementation [55pts; 45pts + 10pts 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**
- **forward**
- **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 Tests: Neural Network [No Points]

You may test your implementation of the functions contained in **NN.py** in the cell below. Feel free to comment out tests for functions that have not been completed yet. See [Using the Local Tests](#) for more details.

In [266...]

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

from NN import dlnet
from local_tests.nn_test import NN_Test

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

print('Local Tests Neural Network \n')
# Local test for LeakyRelu
output = nn.Leaky_Relu(0.05, test_nn.x_train)
leaky_relu_test = np.allclose(output, test_nn.leaky_relu)
print('Your Leaky_Relu works within the expected range:', leaky_relu_test)

# Local test for Tanh
output = nn.Tanh(test_nn.x_train)
tanh_test = np.allclose(output, test_nn.tanh)
print('Your Tanh works within the expected range:', tanh_test)
```

```

# Local test for nloss
output = nn.nloss(test_nn.y_train, test_nn.yh)
nloss_test = np.allclose(output, test_nn.nloss)
print('Your nloss works within the expected range:', nloss_test)

# Local test for forward
# Values for u1, o1, u2 are also saved in nn_test.py if further debugging is needed
nn.nInit()
output = nn.forward(test_nn.x_train)
forward_test = np.allclose(output, test_nn.forward_o2)
print('Your forward works within the expected range:', forward_test)

# Local test for backward
print('\nTest Losses in backward function:')
output = nn.backward(test_nn.y_train, test_nn.forward_o2)
dLoss_theta2_test = np.allclose(output[0], test_nn.dLoss_theta2)
print('Your dLoss_theta2 works within the expected range:', dLoss_theta2_test)
dLoss_b2_test = np.allclose(output[1], test_nn.dLoss_b2)
print('Your dLoss_b2 works within the expected range:', dLoss_b2_test)
dLoss_theta1_test = np.allclose(output[2], test_nn.dLoss_theta1)
print('Your dLoss_theta1 works within the expected range:', dLoss_theta1_test)
dLoss_b1_test = np.allclose(output[3], test_nn.dLoss_b1)
print('Your dLoss_b1 works within the expected range:', dLoss_b1_test)

```

Local Tests Neural Network

Your Leaky_Relu works within the expected range: True
 Your Tanh works within the expected range: True
 Your nloss works within the expected range: True
 Your forward works within the expected range: True

Test Losses in backward function:

Your dLoss_theta2 works within the expected range: True
 Your dLoss_b2 works within the expected range: True
 Your dLoss_theta1 works within the expected range: True
 Your dLoss_b1 works within the expected range: True

1.1.2 Local Test: Gradient Descent [No Points]

There is no local test for the Gradient Descent function, however section 1.2 may help with local debugging.

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 [281...]

```

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

from NN import dlnet
from local_tests.nn_test import NN_Test

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

```

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, Activation
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
```

Using TensorFlow backend.

2.1.1 Load CIFAR-10 dataset [Setup - No points]

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.

```
In [21]: #####
### DO NOT CHANGE THIS CELL ####
#####

# 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')

class_names = ['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)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [=====] - 49s 0us/step
x_train shape: (50000, 32, 32, 3)
x_test shape: (10000, 32, 32, 3)
50000 train samples
10000 test samples
```

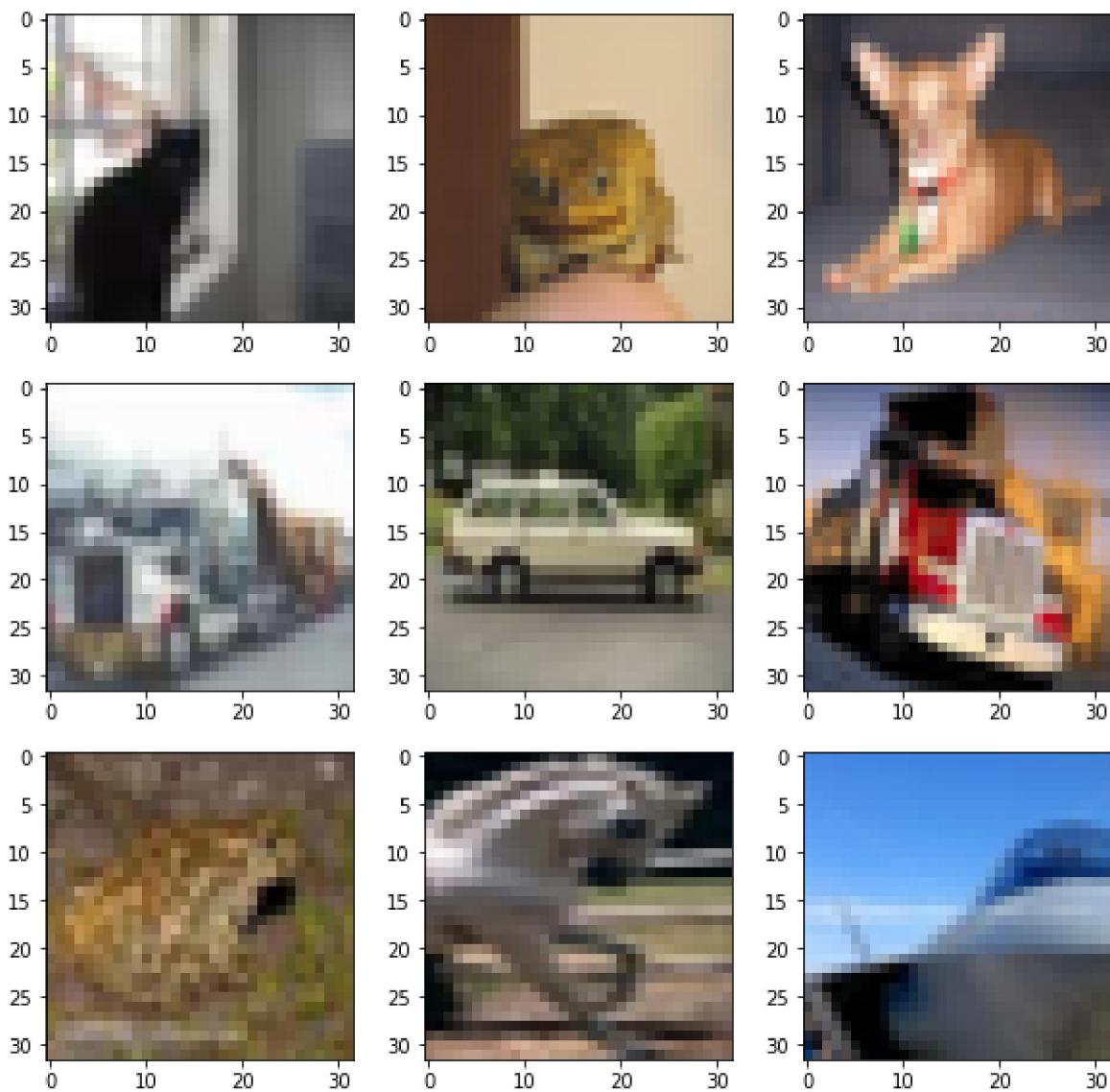
2.1.2 Load some sample images from CIFAR-10 [Setup - No points]

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] **[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: Additional 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: Additional 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 your model architecture

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

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

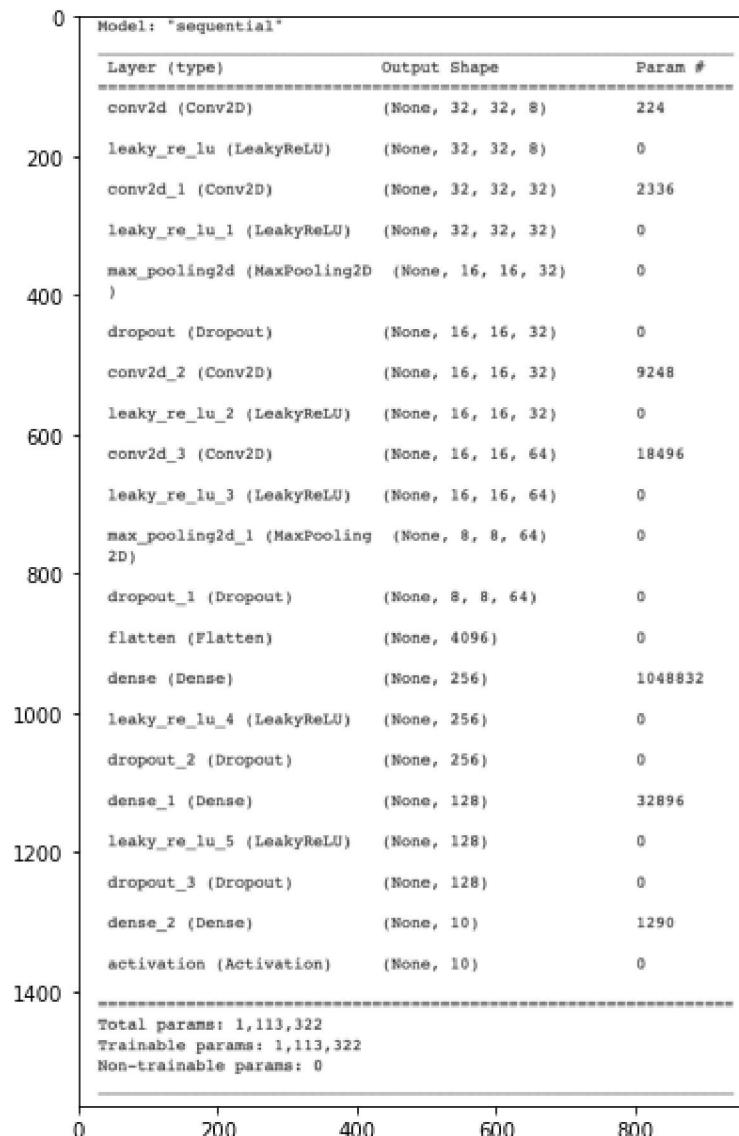
- Conv2d: https://keras.io/api/layers/convolution_layers/convolution2d/
- Dense: https://keras.io/api/layers/core_layers/dense/
- Flatten: https://keras.io/api/layers/reshaping_layers/flatten/
- MaxPool: https://keras.io/api/layers/pooling_layers/max_pooling2d/
- Dropout: https://keras.io/api/layers/regularization_layers/dropout/
- 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 [26]: #####  
### 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[26]: <matplotlib.image.AxesImage at 0x7fda888e5850>
```



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 [80]:

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

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

from cnn import CNN

net = CNN()
```

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

- **_bootstrapping**: this function will be used in `bootstrapping()`
- **fit**: Fit the decision 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. In this section, you will tune your random forest model on a heart disease to achieve a high accuracy on determining the likelihood of patient having narrowing arteries.

Let's first review the dataset in a bit more detail.

Dataset Objective

Imagine that we are doctors working on a cure for heart disease by using machine learning to categorize patients. We know that narrowing arteries are an early indicator of disease. We are tasked with the responsibility of coming up with a method for determining the likelihood of patient having narrowing arteries. We will then use this information to decide which patients to run further tests on for treatment.

After much deliberation amongst the team, you come to a conclusion that we can use past patient data to predict the future occurrence of disease.

We will use our random forest algorithm from Q3.1 to predict if a patient may have indicators of heart disease.

You can find more information on the dataset [here](#).

Loading the dataset

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

Only 13 features used out of a potential 76 to train. We also use the "num" feature as our label we are trying to predict.

Inputs:

1. (age)
2. (type)
3. (cp) chest pain type
4. (trestbps) resting blood pressure (in mm Hg on admission to the hospital)
5. (chol) serum cholestorol in mg/dl
6. (fbs) (fasting blood sugar > 120 mg/dl) (1 = true; 0 = false)
7. (restecg) resting electrocardiographic results:
 - Value 0: normal
 - Value 1: having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of > 0.05 mV)
 - Value 2: showing probable or definite left ventricular hypertrophy by Estes' criteria
8. (thalach) maximum heart rate achieved
9. (exang) exercise induced angina (1 = yes; 0 = no)
10. (oldpeak) ST depression induced by exercise relative to rest
11. (slope) the slope of the peak exercise ST segment
 - Value 1: upsloping
 - Value 2: flat
 - Value 3: downsloping
12. (ca) number of major vessels (0-3) colored by flourosopy
13. (thal) 3 = normal; 6 = fixed defect; 7 = reversable defect

Output:

1. (num) target value:
 - 0 means <50% chance of narrowing arteries
 - 1 means greater than 50% chance of narrowing arteries

Your random forest model will try to predict this variable.

In [102...]

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

from sklearn import preprocessing
import pandas as pd
preprocessor = preprocessing.LabelEncoder()

data_train = pd.read_csv("./data/heart_disease_cleaveland_train.csv")
data_test = pd.read_csv("./data/heart_disease_cleaveland_test.csv")

X_train = data_train.drop(columns = 'num')
y_train = data_train['num']
y_train = y_train.to_numpy()
y_train[y_train > 1] = 1
X_test = data_test.drop(columns = 'num')
X_test = np.array(X_test)
```

```
y_test = data_test['num']
y_test = y_test.to_numpy()
y_test[y_test > 1] = 1
X_train, y_train, X_test, y_test = np.array(X_train), np.array(y_train), np.array(y_test)
```

In [103...]

```
#####
### DO NOT CHANGE THIS CELL #####
#####
assert X_train.shape == (200, 13)
assert y_train.shape == (200,)
assert X_test.shape == (98, 13)
assert y_test.shape == (98,)
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

Out[103]: ((200, 13), (200,), (98, 13), (98,))

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 75%+).

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 **75% accuracy** against the test set in Gradescope to receive full credit for this section.

In [134...]

```
"""
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
values provided here as a starting point, you will need to obtain 75% on the test set to
receive full credit for this section.
"""

from random_forest import RandomForest
import sklearn.ensemble

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

#####
# CHANGE THESE VALUES #####
n_estimators = 7 # Hint: Consider values between 5-12.
max_depth = 12 # Hint: Consider values between 3-12.
max_features = 0.7 # Hint: Consider values between 0.6-1.0.
#####

random_forest = RandomForest(n_estimators, max_depth, max_features, random_seed=student_random_seed)
random_forest.fit(X_train, y_train)
```

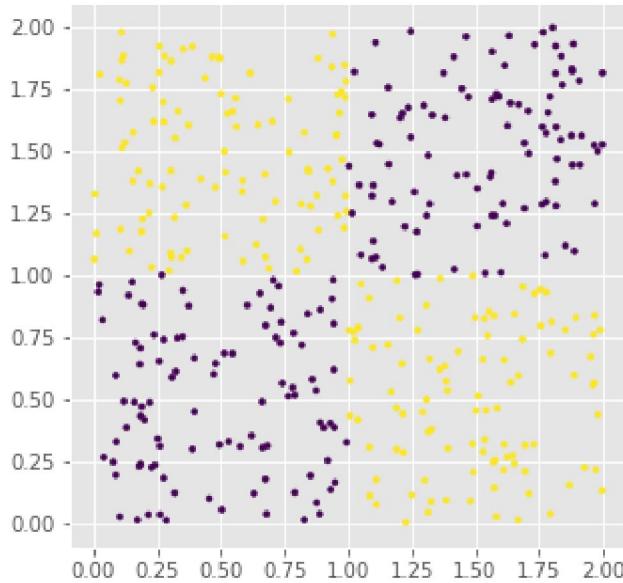
```

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 [248...]

```

#####
### 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 visualize 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 [249...]

```

#####
### DO NOT CHANGE THIS CELL ###
#####
# Try to fit a linear classifier to the dataset

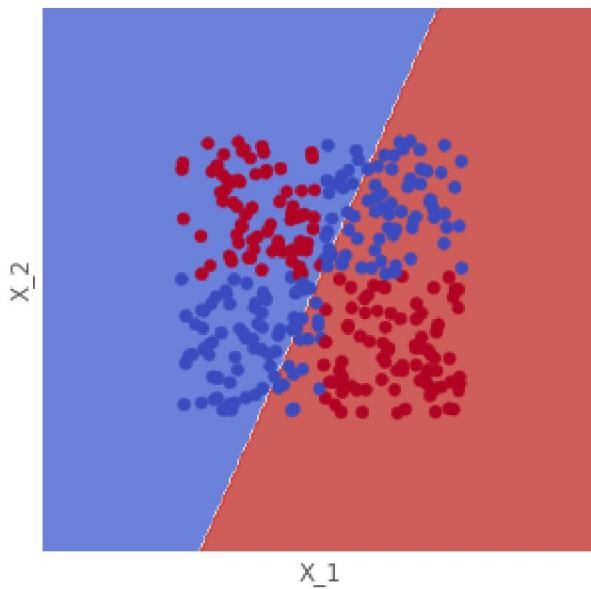
svm_cls = svm.LinearSVC()
svm_cls.fit(X_train, y_train)
y_test_predicted = svm_cls.predict(X_test)

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

visualize_decision_boundary(X_train, y_train)

```

Accuracy on test dataset: 0.425



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 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 [313]:

```
#####
### 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 [314]:

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

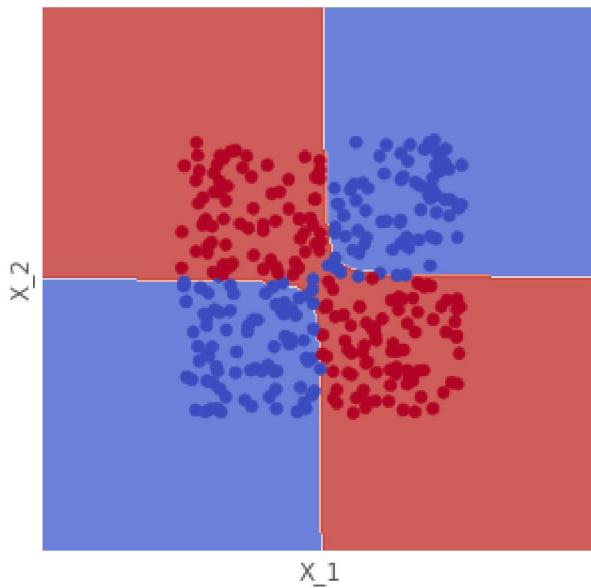
# Fit to the new features and vizualize the decision boundary
# You should get more than 90% accuracy on test set

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)
```

Accuracy on test dataset: 0.9375



In []: