Homework 4 Spring 202

Due Date - 11/23/2022

Your Name - Clarence Jiang

Your UNI - yj2737

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import pprint
pp = pprint.PrettyPrinter(indent=4)
import warnings
warnings.filterwarnings("ignore")
```

PART 2 CIFAR 10 Dataset

CIFAR-10 is a dataset of 60,000 color images (32 by 32 resolution) across 10 classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). The train/test split is 50k/10k.

In [2]:

```
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
from tensorflow.keras.datasets import cifar10
(x_dev, y_dev), (x_test, y_test) = cifar10.load_data()
```

In [3]:

```
LABELS = ['airplane','automobile','bird','cat','deer','dog','frog','horse','ship','truck']
```

In [4]:

```
y_dev_flatten = y_dev.flatten()
```

In [5]:

```
y_dev_flatten.shape
```

Out[5]:

```
(50000,)
```

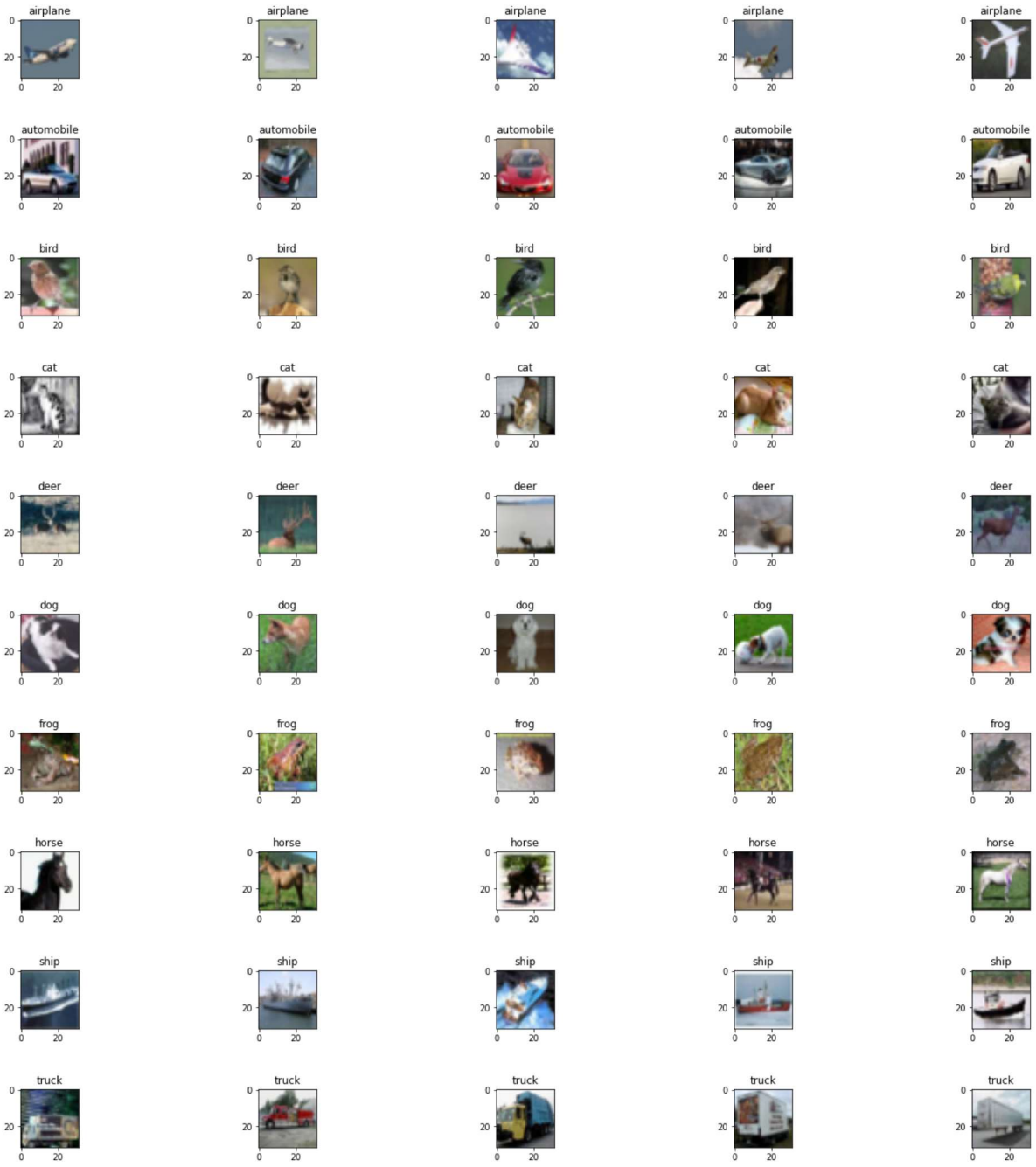
2.1 Plot 5 samples from each class/label from train set on a 10*5 subplot

In [6]:

```
#Your code here

import random
fig, axs = plt.subplots(10, 5 , figsize = (20,20))
fig.tight_layout(pad=5.0)
for i in range(10):
    interest_index_list = np.where(y_dev_flatten == i)[0]
    interest_index = random.choices(list(interest_index_list), k=5)
    x_dev_interest = x_dev[interest_index]
    y_dev_interest = y_dev[interest_index]

    for j in range(5):
        image = x_dev_interest[j]
        axs[i,j].imshow(image)
        axs[i,j].set_title(LABELS[i])
```



2.2 Preparing the dataset for CNN

1) Print the shapes - $x_{dev}, y_{dev}, x_{test}, y_{test}$

2) Flatten the images into one-dimensional vectors and again print the shapes of x_{dev},x_{test}

3) Standardize the development and test sets.

4) Train-test split your development set into train and validation sets (8:2 ratio).

In [7]:

```
#Your code here
print(f"The shape of X_dev is {x_dev.shape}")
print(f"The shape of y_dev is {y_dev.shape}")
print(f"The shape of X_test is {x_test.shape}")
print(f"The shape of y_test is {y_test.shape}")
```

```
The shape of X_dev is (50000, 32, 32, 3)
The shape of y_dev is (50000, 1)
The shape of X_test is (10000, 32, 32, 3)
The shape of y_test is (10000, 1)
```

In [8]:

```
x_dev_rs = x_dev.reshape(50000, 32*32*3)
x_test_rs = x_test.reshape(10000, 32*32*3)

print(f"The shape of X_dev is {x_dev_rs.shape}")
print(f"The shape of X_test is {x_test_rs.shape}")
```

```
The shape of X_dev is (50000, 3072)
The shape of X_test is (10000, 3072)
```

In [9]:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
x_dev_scale = scaler.fit_transform(x_dev_rs)
x_test_scale = scaler.fit_transform(x_test_rs)
```

In [10]:

```
from sklearn.model_selection import train_test_split
from keras.utils.np_utils import to_categorical
y_dev_cat = to_categorical(y_dev, 10)
x_train, x_val,y_train, y_val = train_test_split(
    x_dev_scale, y_dev_cat, test_size=0.2, random_state=42)
```

2.3 Build the feed forward network

First hidden layer size - 128

Second hidden layer size - 64

Third and last layer size - You should know this

In [11]:

```
#Your code here
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
model = Sequential()
model.add(Dense(128, input_shape=(3072,), activation="relu"))
model.add(Dense(64, activation="relu"))
model.add(Dense(10, activation="softmax"))
```

2.4) Print out the model summary. Can show show the calculation for each layer for estimating the number of parameters

In [12]:

```
#Your code here
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               393344

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 10)                650

=================================================================
Total params: 402,250
Trainable params: 402,250
Non-trainable params: 0
_________________________________________________________________
```

2.5) Do you think this number is dependent on the image height and width?

ans: No I do not think it depends on the image height and width. It more depends on the number of neurons at current and previous layers.

Printing out your model's output on first train sample. This will confirm if your dimensions are correctly set up. The sum of this output equal to 1 upto two decimal places?

In [13]:

```
#modify name of X_train based on your requirement

model.compile()
output = model.predict(x_train[0].reshape(1,-1))

# print(output)
print("Output: {:.2f}".format(sum(output[0])))
```

```
1/1 [==============================] - 0s 150ms/step
Output: 1.00
```

2.6) Using the right metric and the right loss function, with Adam as the optimizer, train your model for 20 epochs with batch size 128.

In [14]:

```
#Your code here
model.compile("adam", "categorical_crossentropy", metrics = ["accuracy"])

history = model.fit(x_dev_scale, y_dev_cat, batch_size = 128, epochs = 20, validation_split
```

```
Epoch 1/20
313/313 [==============================] - 2s 4ms/step - loss: 1.7884 - accu
racy: 0.3768 - val_loss: 1.6601 - val_accuracy: 0.4196
Epoch 2/20
313/313 [==============================] - 1s 4ms/step - loss: 1.5409 - accu
racy: 0.4603 - val_loss: 1.6011 - val_accuracy: 0.4499
Epoch 3/20
313/313 [==============================] - 1s 4ms/step - loss: 1.4313 - accu
racy: 0.4960 - val_loss: 1.5267 - val_accuracy: 0.4724
Epoch 4/20
313/313 [==============================] - 1s 4ms/step - loss: 1.3545 - accu
racy: 0.5243 - val_loss: 1.4957 - val_accuracy: 0.4836
Epoch 5/20
313/313 [==============================] - 1s 4ms/step - loss: 1.2847 - accu
racy: 0.5506 - val_loss: 1.5003 - val_accuracy: 0.4818
Epoch 6/20
313/313 [==============================] - 1s 4ms/step - loss: 1.2258 - accu
racy: 0.5681 - val_loss: 1.4882 - val_accuracy: 0.4894
Epoch 7/20
313/313 [==============================] - 1s 4ms/step - loss: 1.1834 - accu
racy: 0.5846 - val_loss: 1.5114 - val_accuracy: 0.4889
Epoch 8/20
313/313 [==============================] - 1s 4ms/step - loss: 1.1397 - accu
racy: 0.5972 - val_loss: 1.4952 - val_accuracy: 0.4934
Epoch 9/20
313/313 [==============================] - 1s 4ms/step - loss: 1.0938 - accu
racy: 0.6173 - val_loss: 1.5316 - val_accuracy: 0.4934
Epoch 10/20
313/313 [==============================] - 1s 4ms/step - loss: 1.0564 - accu
racy: 0.6274 - val_loss: 1.5442 - val_accuracy: 0.5009
Epoch 11/20
313/313 [==============================] - 1s 4ms/step - loss: 1.0149 - accu
racy: 0.6442 - val_loss: 1.5491 - val_accuracy: 0.4938
Epoch 12/20
313/313 [==============================] - 1s 4ms/step - loss: 0.9768 - accu
racy: 0.6564 - val_loss: 1.6038 - val_accuracy: 0.4938
Epoch 13/20
313/313 [==============================] - 1s 3ms/step - loss: 0.9413 - accu
racy: 0.6667 - val_loss: 1.6139 - val_accuracy: 0.4911
Epoch 14/20
313/313 [==============================] - 1s 4ms/step - loss: 0.9158 - accu
racy: 0.6784 - val_loss: 1.6364 - val_accuracy: 0.4870
Epoch 15/20
313/313 [==============================] - 1s 4ms/step - loss: 0.8723 - accu
racy: 0.6924 - val_loss: 1.6774 - val_accuracy: 0.4931
Epoch 16/20
313/313 [==============================] - 1s 4ms/step - loss: 0.8507 - accu
racy: 0.7002 - val_loss: 1.7086 - val_accuracy: 0.4859
Epoch 17/20
313/313 [==============================] - 1s 3ms/step - loss: 0.8161 - accu
racy: 0.7128 - val_loss: 1.7355 - val_accuracy: 0.4863
Epoch 18/20
313/313 [==============================] - 1s 4ms/step - loss: 0.7931 - accu
```

```
racy: 0.7205 - val_loss: 1.7763 - val_accuracy: 0.4914
Epoch 19/20
313/313 [==============================] - 1s 4ms/step - loss: 0.7718 - accu
racy: 0.7272 - val_loss: 1.8222 - val_accuracy: 0.4818
Epoch 20/20
313/313 [==============================] - 1s 4ms/step - loss: 0.7403 - accu
racy: 0.7395 - val_loss: 1.8999 - val_accuracy: 0.4779
```

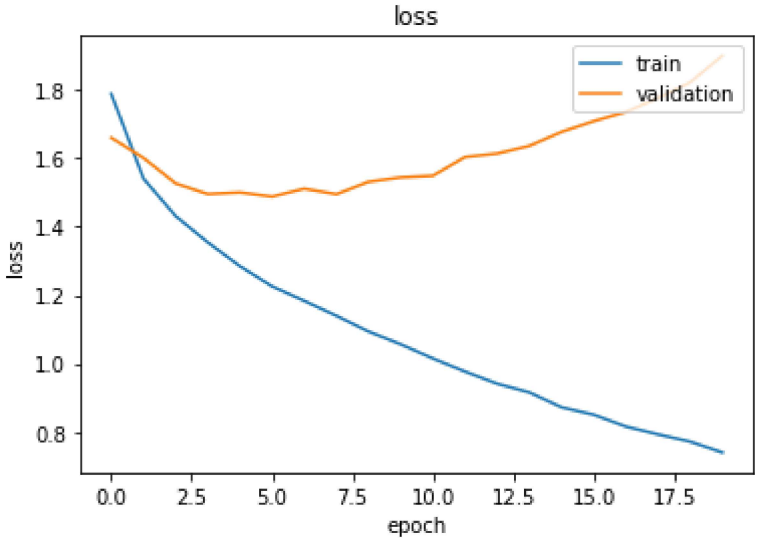
2.7) Plot a separate plots for:

a. displaying train vs validation loss over each epoch

b. displaying train vs validation accuracy over each epoch

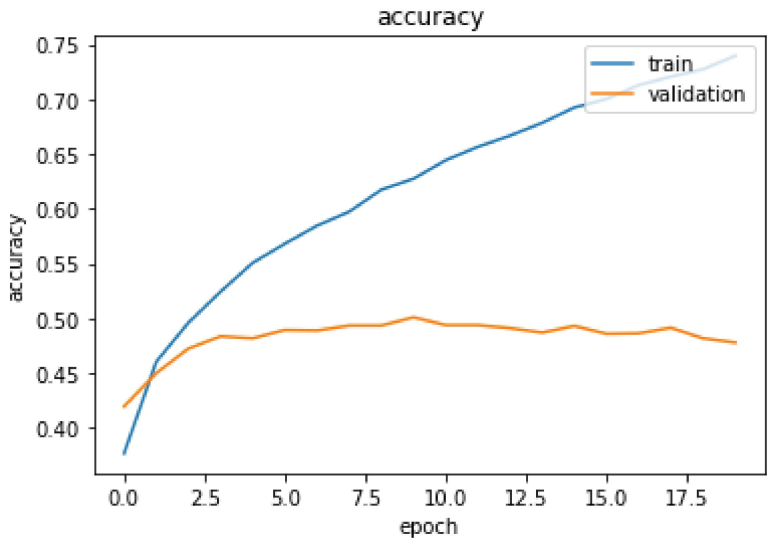
In [15]:

```
#Your code here
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```



In [16]:

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```



2.8) Finally, report the metric chosen on test set.

In [17]:

```
#Your code here
y_test_cat = to_categorical(y_test, 10)
score = model.evaluate(x_test_scale, y_test_cat, verbose=0)
print(score)
print(f"Test loss: {score[0]}")
print(f"Test accuracy: {score[1]}")
```

```
[1.861322283744812, 0.48570001125335693]
Test loss: 1.861322283744812
Test accuracy: 0.48570001125335693
```

2.9 If the accuracy achieved is quite less(<50%), try improve the accuracy [Open ended question, you may try different approaches]

In [52]:

```
#Your code here
from tensorflow.keras.layers import Dropout
model2 = Sequential()
model2.add(Dense(128, input_shape=(3072,), activation="relu"))
model2.add(Dense(64, activation="relu"))
model2.add(Dense(32, activation="relu"))
model2.add(Dense(16, activation="relu"))
model2.add(Dense(10, activation="softmax"))
```

In [53]:

```
model2.compile("adam", "categorical_crossentropy", metrics = ["accuracy"])

history2 = model2.fit(x_dev_scale, y_dev_cat, batch_size = 32, epochs = 13, validation_spli
```

```
Epoch 1/13
1250/1250 [==============================] - 3s 2ms/step - loss: 1.8523 - ac
curacy: 0.3396 - val_loss: 1.7141 - val_accuracy: 0.3928
Epoch 2/13
1250/1250 [==============================] - 2s 2ms/step - loss: 1.6189 - ac
curacy: 0.4270 - val_loss: 1.6000 - val_accuracy: 0.4452
Epoch 3/13
1250/1250 [==============================] - 2s 2ms/step - loss: 1.5164 - ac
curacy: 0.4664 - val_loss: 1.5709 - val_accuracy: 0.4544
Epoch 4/13
1250/1250 [==============================] - 2s 2ms/step - loss: 1.4443 - ac
curacy: 0.4868 - val_loss: 1.5207 - val_accuracy: 0.4664
Epoch 5/13
1250/1250 [==============================] - 2s 2ms/step - loss: 1.3773 - ac
curacy: 0.5138 - val_loss: 1.5213 - val_accuracy: 0.4681
Epoch 6/13
1250/1250 [==============================] - 2s 2ms/step - loss: 1.3201 - ac
curacy: 0.5348 - val_loss: 1.4863 - val_accuracy: 0.4904
Epoch 7/13
1250/1250 [==============================] - 2s 2ms/step - loss: 1.2652 - ac
curacy: 0.5554 - val_loss: 1.4799 - val_accuracy: 0.4864
Epoch 8/13
1250/1250 [==============================] - 2s 2ms/step - loss: 1.2216 - ac
curacy: 0.5687 - val_loss: 1.4721 - val_accuracy: 0.5016
Epoch 9/13
1250/1250 [==============================] - 2s 2ms/step - loss: 1.1779 - ac
curacy: 0.5852 - val_loss: 1.4681 - val_accuracy: 0.5000
Epoch 10/13
1250/1250 [==============================] - 2s 2ms/step - loss: 1.1376 - ac
curacy: 0.5957 - val_loss: 1.4737 - val_accuracy: 0.5002
Epoch 11/13
1250/1250 [==============================] - 2s 2ms/step - loss: 1.1067 - ac
curacy: 0.6119 - val_loss: 1.4990 - val_accuracy: 0.5083
Epoch 12/13
1250/1250 [==============================] - 2s 2ms/step - loss: 1.0722 - ac
curacy: 0.6229 - val_loss: 1.5147 - val_accuracy: 0.5028
Epoch 13/13
1250/1250 [==============================] - 2s 2ms/step - loss: 1.0373 - ac
curacy: 0.6340 - val_loss: 1.5081 - val_accuracy: 0.5131
```

In [55]:

```
score2 = model2.evaluate(x_test_scale, y_test_cat, verbose=0)
print(f"Test loss: {score2[0]}")
print(f"Test accuracy: {score2[1]}")
```

```
Test loss: 1.504762053489685
Test accuracy: 0.5113000273704529
```

2.10 Plot the first 50 samples of test dataset on a 10*5 subplot and this time label the images with both the ground truth (GT) and predicted class (P). (Make sure you predict the class with the improved model)

In [21]:

```
#Your code here
y_prob = model.predict(x_test_scale[:50])
y_classes = y_prob.argmax(axis=-1)
y_classes
```

```
2/2 [==============================] - 0s 2ms/step
```

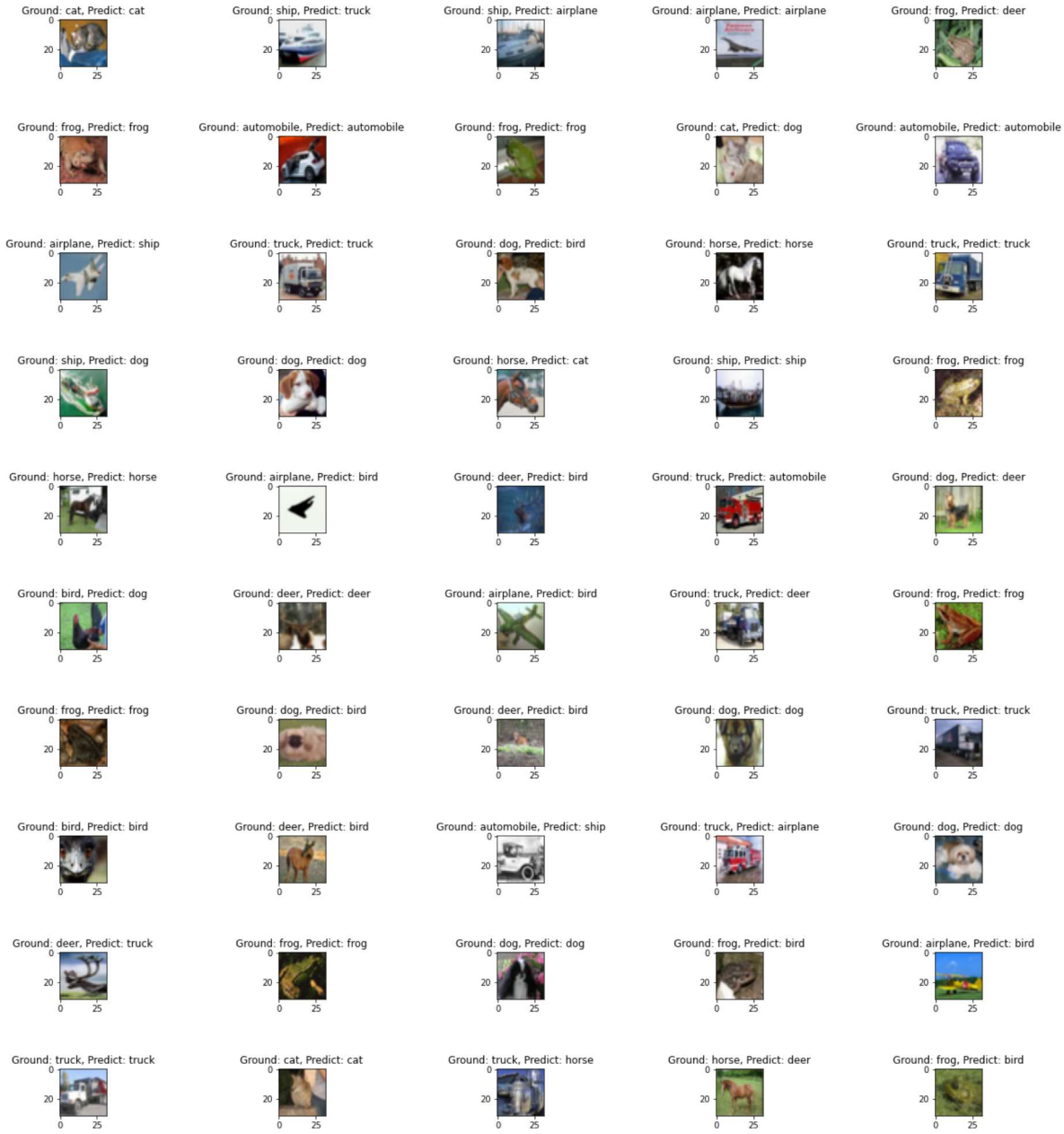
Out[21]:

```
array([3, 9, 0, 0, 4, 6, 1, 6, 5, 1, 8, 9, 2, 7, 9, 5, 5, 3, 8, 6, 7, 2,
       2, 1, 4, 5, 4, 2, 4, 6, 6, 2, 2, 5, 9, 2, 2, 8, 0, 5, 9, 6, 5, 2,
       2, 9, 3, 7, 4, 2], dtype=int64)
```

In [22]:

```
rows = 10
columns = 5
fig = plt.figure(figsize = (20, 20))

for i in range(1, rows*columns+1):
    fig.tight_layout(pad=5.0)
    fig.add_subplot(rows, columns, i).set_title(f"Ground: {LABELS[y_test[i-1][0]]}, Predict
    image = x_test[i-1]
    plt.imshow(image)
```



PART 3 Convolutional Neural Network

In this part of the homework, we will build and train a classical convolutional neural network on the CIFAR Dataset

In [23]:

```
from tensorflow.keras.datasets import cifar10
(x_dev, y_dev), (x_test, y_test) = cifar10.load_data()
print("x_dev: {},y_dev: {},x_test: {},y_test: {}".format(x_dev.shape, y_dev.shape, x_test.s

x_dev, x_test = x_dev.astype('float32'), x_test.astype('float32')
x_dev = x_dev/255.0
x_test = x_test/255.0


from sklearn.model_selection import train_test_split

X_train, X_val, y_train, y_val = train_test_split(x_dev, y_dev,test_size = 0.2, random_stat
```

```
x_dev: (50000, 32, 32, 3),y_dev: (50000, 1),x_test: (10000, 32, 32, 3),y_tes
t: (10000, 1)
```

3.1 We will be implementing the one of the first CNN models put forward by Yann LeCunn, which is commonly refered to as LeNet-5. The network has the following layers:

1) 2D convolutional layer with 6 filters, 5x5 kernel, stride of 1 padded to yield the same size as input, ReLU activation

2) Maxpooling layer of 2x2

3) 2D convolutional layer with 16 filters, 5x5 kernel, 0 padding, ReLU activation

4)Maxpooling layer of 2x2

5) 2D convolutional layer with 120 filters, 5x5 kernel, ReLU activation. Note that this layer has 120 output channels (filters), and each channel has only 1 number. The output of this layer is just a vector with 120 units!

6) A fully connected layer with 84 units, ReLU activation

7) The output layer where each unit respresents the probability of image being in that category. What activation function should you use in this layer? (You should know this)

In [40]:

```
# your code here

# cnn initialize
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten
cnn = Sequential()

#1
cnn.add(Conv2D(6, kernel_size = (5,5), activation='relu', input_shape=(32,32,3), padding =
#2
cnn.add(MaxPooling2D(pool_size = (2,2)))
#3
cnn.add(Conv2D(16, kernel_size = (5,5), activation='relu', padding = "valid"))
#4
cnn.add(MaxPooling2D(pool_size = (2,2)))
#5
cnn.add(Conv2D(120, kernel_size = (5,5), activation='relu'))
# 5.5
cnn.add(Flatten())
#6
cnn.add(Dense(84, activation = 'relu'))
#7
cnn.add(Dense(10, activation = 'softmax'))
```

3.2 Report the model summary

In [41]:

```
#your code here
cnn.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 32, 32, 6)         456

 max_pooling2d_2 (MaxPooling  (None, 16, 16, 6)        0
 2D)

 conv2d_4 (Conv2D)           (None, 12, 12, 16)        2416

 max_pooling2d_3 (MaxPooling  (None, 6, 6, 16)         0
 2D)

 conv2d_5 (Conv2D)           (None, 2, 2, 120)         48120

 flatten_1 (Flatten)         (None, 480)               0

 dense_10 (Dense)            (None, 84)                40404

 dense_11 (Dense)            (None, 10)                850

=================================================================
Total params: 92,246
Trainable params: 92,246
Non-trainable params: 0
_________________________________________________________________
```

3.3 Model Training

1) Train the model for 20 epochs. In each epoch, record the loss and metric (chosen in part 3) scores for both train and validation sets.

2) Plot a separate plots for:

- displaying train vs validation loss over each epoch
- displaying train vs validation accuracy over each epoch

3) Report the model performance on the test set. Feel free to tune the hyperparameters such as batch size and optimizers to achieve better performance.

In [42]:

```
x_dev.reshape(x_dev.shape[0], 32, 32, 3)
x_dev.shape
```

Out[42]:

```
(50000, 32, 32, 3)
```

In [43]:

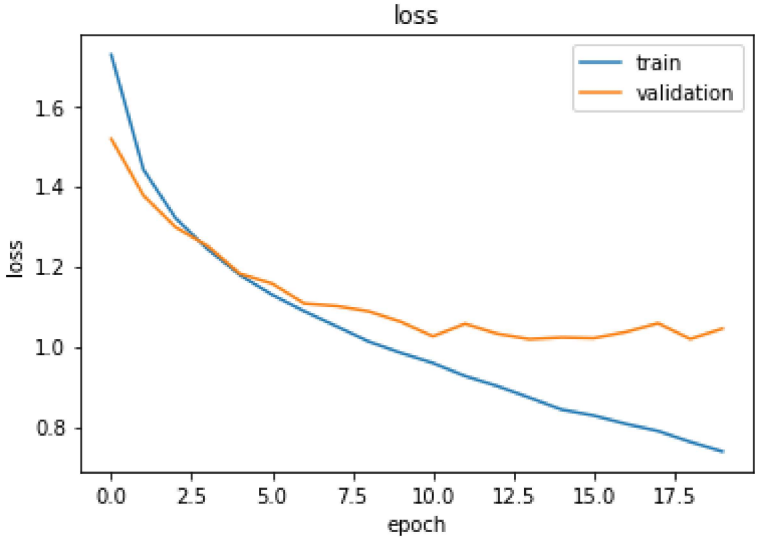
```
# Your code here
cnn.compile("adam", "categorical_crossentropy", metrics=['accuracy'])
history_cnn = cnn.fit(x_dev, to_categorical(y_dev, 10), batch_size = 128, epochs = 20, vali
```

```
Epoch 1/20
352/352 [==============================] - 9s 26ms/step - loss: 1.7273 - acc
uracy: 0.3704 - val_loss: 1.5174 - val_accuracy: 0.4514
Epoch 2/20
352/352 [==============================] - 9s 25ms/step - loss: 1.4407 - acc
uracy: 0.4788 - val_loss: 1.3764 - val_accuracy: 0.5012
Epoch 3/20
352/352 [==============================] - 9s 25ms/step - loss: 1.3199 - acc
uracy: 0.5265 - val_loss: 1.2973 - val_accuracy: 0.5368
Epoch 4/20
352/352 [==============================] - 9s 25ms/step - loss: 1.2420 - acc
uracy: 0.5569 - val_loss: 1.2503 - val_accuracy: 0.5580
Epoch 5/20
352/352 [==============================] - 9s 25ms/step - loss: 1.1782 - acc
uracy: 0.5810 - val_loss: 1.1814 - val_accuracy: 0.5766
Epoch 6/20
352/352 [==============================] - 9s 25ms/step - loss: 1.1288 - acc
uracy: 0.5996 - val_loss: 1.1568 - val_accuracy: 0.5886
Epoch 7/20
352/352 [==============================] - 9s 25ms/step - loss: 1.0878 - acc
uracy: 0.6153 - val_loss: 1.1072 - val_accuracy: 0.6066
Epoch 8/20
352/352 [==============================] - 9s 25ms/step - loss: 1.0503 - acc
uracy: 0.6292 - val_loss: 1.1008 - val_accuracy: 0.6100
Epoch 9/20
352/352 [==============================] - 9s 25ms/step - loss: 1.0123 - acc
uracy: 0.6420 - val_loss: 1.0876 - val_accuracy: 0.6208
Epoch 10/20
352/352 [==============================] - 9s 25ms/step - loss: 0.9839 - acc
uracy: 0.6515 - val_loss: 1.0615 - val_accuracy: 0.6288
Epoch 11/20
352/352 [==============================] - 9s 26ms/step - loss: 0.9588 - acc
uracy: 0.6618 - val_loss: 1.0252 - val_accuracy: 0.6448
Epoch 12/20
352/352 [==============================] - 9s 26ms/step - loss: 0.9263 - acc
uracy: 0.6736 - val_loss: 1.0565 - val_accuracy: 0.6332
Epoch 13/20
352/352 [==============================] - 9s 26ms/step - loss: 0.9013 - acc
uracy: 0.6821 - val_loss: 1.0316 - val_accuracy: 0.6370
Epoch 14/20
352/352 [==============================] - 9s 25ms/step - loss: 0.8723 - acc
uracy: 0.6928 - val_loss: 1.0180 - val_accuracy: 0.6514
Epoch 15/20
352/352 [==============================] - 9s 27ms/step - loss: 0.8426 - acc
uracy: 0.7042 - val_loss: 1.0223 - val_accuracy: 0.6484
Epoch 16/20
352/352 [==============================] - 9s 26ms/step - loss: 0.8277 - acc
uracy: 0.7083 - val_loss: 1.0205 - val_accuracy: 0.6500
Epoch 17/20
352/352 [==============================] - 9s 26ms/step - loss: 0.8069 - acc
uracy: 0.7160 - val_loss: 1.0359 - val_accuracy: 0.6404
Epoch 18/20
352/352 [==============================] - 9s 25ms/step - loss: 0.7890 - acc
uracy: 0.7228 - val_loss: 1.0578 - val_accuracy: 0.6484
```

```
Epoch 19/20
352/352 [==============================] - 9s 25ms/step - loss: 0.7619 - acc
uracy: 0.7314 - val_loss: 1.0185 - val_accuracy: 0.6592
Epoch 20/20
352/352 [==============================] - 9s 24ms/step - loss: 0.7381 - acc
uracy: 0.7401 - val_loss: 1.0444 - val_accuracy: 0.6478
```

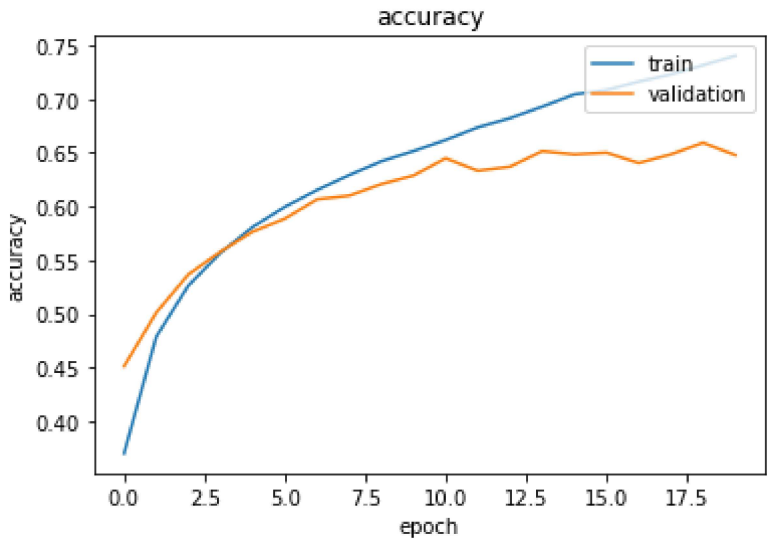
In [44]:

```
plt.plot(history_cnn.history['loss'])
plt.plot(history_cnn.history['val_loss'])
plt.title('loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```



In [45]:

```
plt.plot(history_cnn.history['accuracy'])
plt.plot(history_cnn.history['val_accuracy'])
plt.title('accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```



In [39]:

```
y_test_cat = to_categorical(y_test, 10)
score = cnn.evaluate(x_test, y_test_cat, verbose=0)
print(score)
print(f"Test loss: {score[0]}")
print(f"Test accuracy: {score[1]}")
```

```
[1.40663743019104, 0.6166999936103821]
Test loss: 1.40663743019104
Test accuracy: 0.6166999936103821
```

3.4 Overfitting

1) To overcome overfitting, we will train the network again with dropout this time. For hidden layers use dropout probability of 0.3. Train the model again for 20 epochs. Report model performance on test set.

Plot a separate plots for:

- displaying train vs validation loss over each epoch
- displaying train vs validation accuracy over each epoch

2) This time, let's apply a batch normalization after every hidden layer, train the model for 20 epochs, report model performance on test set as above.

Plot a separate plots for:

- displaying train vs validation loss over each epoch
- displaying train vs validation accuracy over each epoch

3) Compare batch normalization technique with the original model and with dropout, which technique do you think helps with overfitting better?

In [56]:

```
# Your code here

# 1)
cnn2 = Sequential()
cnn2.add(Conv2D(6, kernel_size = (5,5), activation='relu', input_shape=(32,32,3), padding =
cnn2.add(MaxPooling2D(pool_size = (2,2)))
cnn2.add(Dropout(0.3))
cnn2.add(Conv2D(16, kernel_size = (5,5), activation='relu', padding = "valid"))
cnn2.add(MaxPooling2D(pool_size = (2,2)))
cnn2.add(Dropout(0.3))
cnn2.add(Conv2D(120, kernel_size = (5,5), activation='relu'))
cnn2.add(Flatten())
cnn2.add(Dense(84, activation = 'relu'))
#7
cnn2.add(Dense(10, activation = 'softmax'))
```

In [57]:

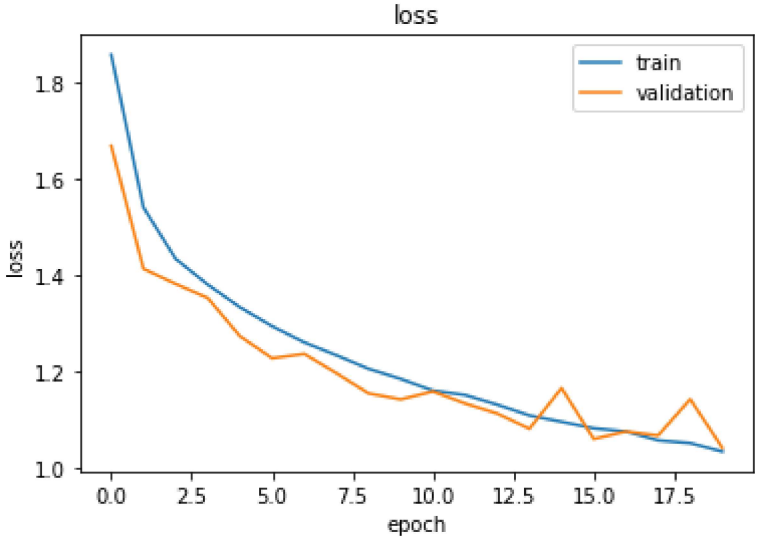
```
cnn2.compile("adam", "categorical_crossentropy", metrics=['accuracy'])
history_cnn2 = cnn2.fit(x_dev, to_categorical(y_dev, 10), batch_size = 128, epochs = 20, va
```

```
Epoch 1/20
313/313 [==============================] - 9s 29ms/step - loss: 1.8578 - a
ccuracy: 0.3167 - val_loss: 1.6685 - val_accuracy: 0.3857
Epoch 2/20
313/313 [==============================] - 9s 28ms/step - loss: 1.5404 - a
ccuracy: 0.4395 - val_loss: 1.4131 - val_accuracy: 0.5052
Epoch 3/20
313/313 [==============================] - 9s 28ms/step - loss: 1.4337 - a
ccuracy: 0.4828 - val_loss: 1.3817 - val_accuracy: 0.5140
Epoch 4/20
313/313 [==============================] - 9s 28ms/step - loss: 1.3803 - a
ccuracy: 0.5034 - val_loss: 1.3524 - val_accuracy: 0.5170
Epoch 5/20
313/313 [==============================] - 9s 28ms/step - loss: 1.3332 - a
ccuracy: 0.5171 - val_loss: 1.2732 - val_accuracy: 0.5464
Epoch 6/20
313/313 [==============================] - 9s 29ms/step - loss: 1.2938 - a
ccuracy: 0.5343 - val_loss: 1.2272 - val_accuracy: 0.5695
Epoch 7/20
313/313 [==============================] - 9s 29ms/step - loss: 1.2602 - a
ccuracy: 0.5489 - val_loss: 1.2363 - val_accuracy: 0.5543
Epoch 8/20
313/313 [==============================] - 9s 30ms/step - loss: 1.2334 - a
ccuracy: 0.5590 - val_loss: 1.1962 - val_accuracy: 0.5762
Epoch 9/20
313/313 [==============================] - 9s 29ms/step - loss: 1.2052 - a
ccuracy: 0.5698 - val_loss: 1.1542 - val_accuracy: 0.5901
Epoch 10/20
313/313 [==============================] - 9s 28ms/step - loss: 1.1843 - a
ccuracy: 0.5781 - val_loss: 1.1418 - val_accuracy: 0.5974
Epoch 11/20
313/313 [==============================] - 9s 27ms/step - loss: 1.1601 - a
ccuracy: 0.5860 - val_loss: 1.1583 - val_accuracy: 0.5840
Epoch 12/20
313/313 [==============================] - 9s 28ms/step - loss: 1.1509 - a
ccuracy: 0.5874 - val_loss: 1.1333 - val_accuracy: 0.5918
Epoch 13/20
313/313 [==============================] - 9s 28ms/step - loss: 1.1308 - a
ccuracy: 0.5964 - val_loss: 1.1123 - val_accuracy: 0.6108
Epoch 14/20
313/313 [==============================] - 9s 27ms/step - loss: 1.1083 - a
ccuracy: 0.6058 - val_loss: 1.0807 - val_accuracy: 0.6243
Epoch 15/20
313/313 [==============================] - 8s 27ms/step - loss: 1.0951 - a
ccuracy: 0.6085 - val_loss: 1.1653 - val_accuracy: 0.5933
Epoch 16/20
313/313 [==============================] - 9s 27ms/step - loss: 1.0817 - a
ccuracy: 0.6145 - val_loss: 1.0594 - val_accuracy: 0.6310
Epoch 17/20
313/313 [==============================] - 8s 27ms/step - loss: 1.0750 - a
ccuracy: 0.6144 - val_loss: 1.0750 - val_accuracy: 0.6154
Epoch 18/20
313/313 [==============================] - 9s 27ms/step - loss: 1.0565 - a
ccuracy: 0.6230 - val_loss: 1.0667 - val_accuracy: 0.6177
Epoch 19/20
```

```
313/313 [==============================] - 8s 27ms/step - loss: 1.0509 - a
ccuracy: 0.6253 - val_loss: 1.1421 - val_accuracy: 0.5957
Epoch 20/20
313/313 [==============================] - 8s 27ms/step - loss: 1.0335 - a
ccuracy: 0.6320 - val_loss: 1.0410 - val_accuracy: 0.6329
```

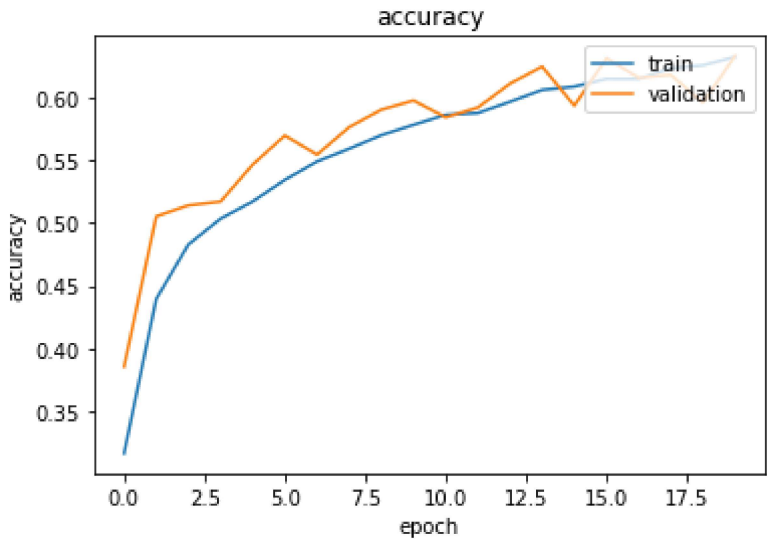
In [58]:

```
plt.plot(history_cnn2.history['loss'])
plt.plot(history_cnn2.history['val_loss'])
plt.title('loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```



In [59]:

```
plt.plot(history_cnn2.history['accuracy'])
plt.plot(history_cnn2.history['val_accuracy'])
plt.title('accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```



In [60]:

```
score = cnn2.evaluate(x_test, y_test_cat, verbose=0)
print(score)
print(f"Test loss: {score[0]}")
print(f"Test accuracy: {score[1]}")
```

```
[1.0528693199157715, 0.6319000124931335]
Test loss: 1.0528693199157715
Test accuracy: 0.6319000124931335
```

In [62]:

```
from tensorflow.keras.layers import BatchNormalization
cnn3 = Sequential()
cnn3.add(Conv2D(6, kernel_size = (5,5), activation='relu', input_shape=(32,32,3), padding =
cnn3.add(MaxPooling2D(pool_size = (2,2)))
cnn3.add(BatchNormalization())
cnn3.add(Conv2D(16, kernel_size = (5,5), activation='relu', padding = "valid"))
cnn3.add(MaxPooling2D(pool_size = (2,2)))
cnn3.add(BatchNormalization())
cnn3.add(Conv2D(120, kernel_size = (5,5), activation='relu'))
cnn3.add(Flatten())
cnn3.add(Dense(84, activation = 'relu'))
cnn3.add(Dense(10, activation = 'softmax'))
```

In [63]:

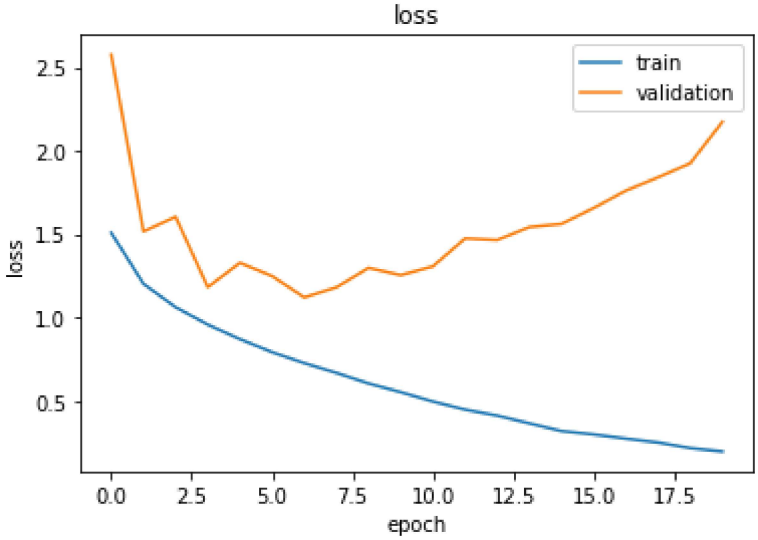
```
cnn3.compile("adam", "categorical_crossentropy", metrics=['accuracy'])
history_cnn3 = cnn3.fit(x_dev, to_categorical(y_dev, 10), batch_size = 128, epochs = 20, va
```

```
Epoch 1/20
313/313 [==============================] - 10s 30ms/step - loss: 1.5082 -
accuracy: 0.4540 - val_loss: 2.5761 - val_accuracy: 0.2157
Epoch 2/20
313/313 [==============================] - 9s 29ms/step - loss: 1.2025 - a
ccuracy: 0.5723 - val_loss: 1.5165 - val_accuracy: 0.4978
Epoch 3/20
313/313 [==============================] - 9s 29ms/step - loss: 1.0620 - a
ccuracy: 0.6246 - val_loss: 1.6045 - val_accuracy: 0.4858
Epoch 4/20
313/313 [==============================] - 9s 29ms/step - loss: 0.9579 - a
ccuracy: 0.6608 - val_loss: 1.1824 - val_accuracy: 0.5945
Epoch 5/20
313/313 [==============================] - 9s 29ms/step - loss: 0.8708 - a
ccuracy: 0.6911 - val_loss: 1.3282 - val_accuracy: 0.5676
Epoch 6/20
313/313 [==============================] - 9s 30ms/step - loss: 0.7926 - a
ccuracy: 0.7211 - val_loss: 1.2491 - val_accuracy: 0.5949
Epoch 7/20
313/313 [==============================] - 9s 29ms/step - loss: 0.7269 - a
ccuracy: 0.7437 - val_loss: 1.1208 - val_accuracy: 0.6317
Epoch 8/20
313/313 [==============================] - 9s 29ms/step - loss: 0.6681 - a
ccuracy: 0.7645 - val_loss: 1.1814 - val_accuracy: 0.6198
Epoch 9/20
313/313 [==============================] - 9s 29ms/step - loss: 0.6053 - a
ccuracy: 0.7880 - val_loss: 1.2980 - val_accuracy: 0.5907
Epoch 10/20
313/313 [==============================] - 9s 29ms/step - loss: 0.5527 - a
ccuracy: 0.8043 - val_loss: 1.2546 - val_accuracy: 0.6226
Epoch 11/20
313/313 [==============================] - 9s 29ms/step - loss: 0.4965 - a
ccuracy: 0.8276 - val_loss: 1.3070 - val_accuracy: 0.6272
Epoch 12/20
313/313 [==============================] - 9s 29ms/step - loss: 0.4482 - a
ccuracy: 0.8421 - val_loss: 1.4742 - val_accuracy: 0.5880
Epoch 13/20
313/313 [==============================] - 9s 29ms/step - loss: 0.4121 - a
ccuracy: 0.8547 - val_loss: 1.4651 - val_accuracy: 0.6117
Epoch 14/20
313/313 [==============================] - 9s 30ms/step - loss: 0.3654 - a
ccuracy: 0.8710 - val_loss: 1.5429 - val_accuracy: 0.6012
Epoch 15/20
313/313 [==============================] - 9s 30ms/step - loss: 0.3192 - a
ccuracy: 0.8909 - val_loss: 1.5619 - val_accuracy: 0.6209
Epoch 16/20
313/313 [==============================] - 9s 29ms/step - loss: 0.2989 - a
ccuracy: 0.8936 - val_loss: 1.6569 - val_accuracy: 0.6108
Epoch 17/20
313/313 [==============================] - 9s 30ms/step - loss: 0.2740 - a
ccuracy: 0.9038 - val_loss: 1.7610 - val_accuracy: 0.6099
Epoch 18/20
313/313 [==============================] - 9s 29ms/step - loss: 0.2509 - a
ccuracy: 0.9117 - val_loss: 1.8412 - val_accuracy: 0.6083
Epoch 19/20
```

```
313/313 [==============================] - 9s 30ms/step - loss: 0.2175 - a
ccuracy: 0.9245 - val_loss: 1.9251 - val_accuracy: 0.6194
Epoch 20/20
313/313 [==============================] - 10s 33ms/step - loss: 0.1970 -
accuracy: 0.9310 - val_loss: 2.1733 - val_accuracy: 0.5978
```

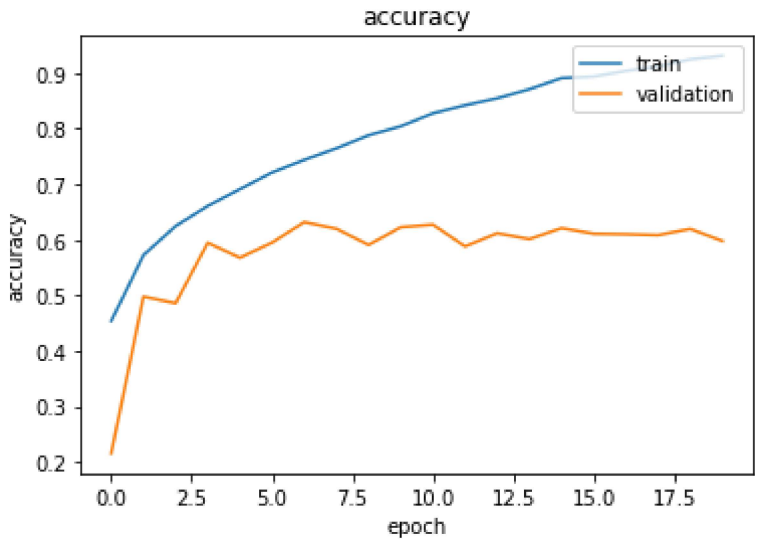
In [64]:

```
plt.plot(history_cnn3.history['loss'])
plt.plot(history_cnn3.history['val_loss'])
plt.title('loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```



In [65]:

```
plt.plot(history_cnn3.history['accuracy'])
plt.plot(history_cnn3.history['val_accuracy'])
plt.title('accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```



In [66]:

```
score = cnn3.evaluate(x_test, y_test_cat, verbose=0)
print(score)
print(f"Test loss: {score[0]}")
print(f"Test accuracy: {score[1]}")
```

```
[2.198857069015503, 0.5889000296592712]
Test loss: 2.198857069015503
Test accuracy: 0.5889000296592712
```

In [ ]:

```

```