Homework 4 Spring 202

Due Date - 11/23/2022

Your Name: Liwen Zhu

Your UNI: lz2512

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.

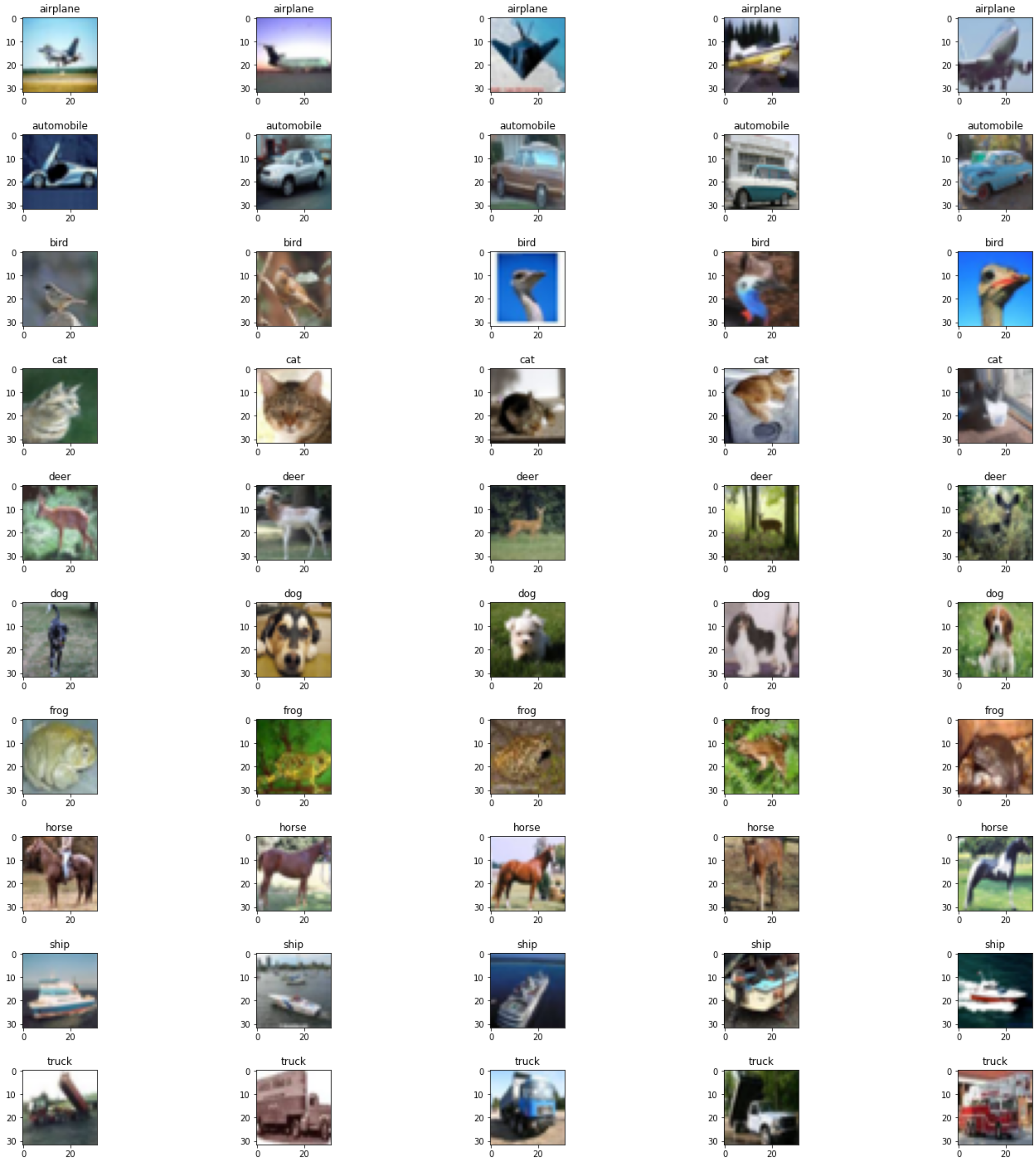
In [2]:
```
from tensorflow.keras.datasets import cifar10
(x_dev, y_dev), (x_test, y_test) = cifar10.load_data()
```

```
2022-11-23 10:21:59.378129: I tensorflow/core/platform/cpu_feature_guard.cc:
193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Lib
rary (oneDNN) to use the following CPU instructions in performance-critical
operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
```

In [3]:
```
LABELS = ['airplane','automobile','bird','cat','deer','dog','frog','horse','
```

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

In [4]:
```
#Your code here
import random
fig,axs = plt.subplots(10,5,figsize=(20,20))
fig.tight_layout(pad=3)
for i in range(10):
    target_list = np.where(y_dev==i)[0]
    index = random.choices(list(target_list),k=5)
    x_dev_list = x_dev[index]

    for j in range(5):
        image = x_dev_list[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 [5]:
```
#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 [6]:
```
x_dev_rs = x_dev.reshape(x_dev.shape[0],32*32*3)
x_test_rs = x_test.reshape(x_test.shape[0],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 [7]:
```
from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
x_dev_std = ss.fit_transform(x_dev_rs)
x_test_std = ss.fit_transform(x_test_rs)
```

In [8]:
```
from sklearn.model_selection import train_test_split
from keras.utils.np_utils import to_categorical
y_dev_tc = to_categorical(y_dev,10)
y_test_tc = to_categorical(y_test,10)
x_train, x_val, y_train, y_val = train_test_split(x_dev_std,y_dev_tc,test_si
```

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 [9]:
```
#Your code here
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
model = Sequential([
    Dense(128, input_shape=(3072,)),
    Activation('relu'),
    Dense(64),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

```
2022-11-23 10:22:35.495472: I tensorflow/core/platform/cpu_feature_guard.cc:
193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Lib
rary (oneDNN) to use the following CPU instructions in performance-critical
operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
```

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

In [10]:
```
#Your code here
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense (Dense)               (None, 128)               393344    
                                                                 
 activation (Activation)     (None, 128)               0         
                                                                 
 dense_1 (Dense)             (None, 64)                8256      
                                                                 
 activation_1 (Activation)   (None, 64)                0         
                                                                 
 dense_2 (Dense)             (None, 10)                650       
                                                                 
 activation_2 (Activation)   (None, 10)                0         
                                                                 
=================================================================
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?

In [11]:
```
# Your text here
print("No, I think the number of parameters depends on the number of neurons
```

```
No, I think the number of parameters depends on the number of neurons at cur
rent 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 [12]:
```
#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 286ms/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 [13]:
```
#Your code here
model.compile(optimizer="adam",loss="categorical_crossentropy",metrics=["acc
history_callback = model.fit(x_dev_std,y_dev_tc,batch_size=128,epochs=20,val
```

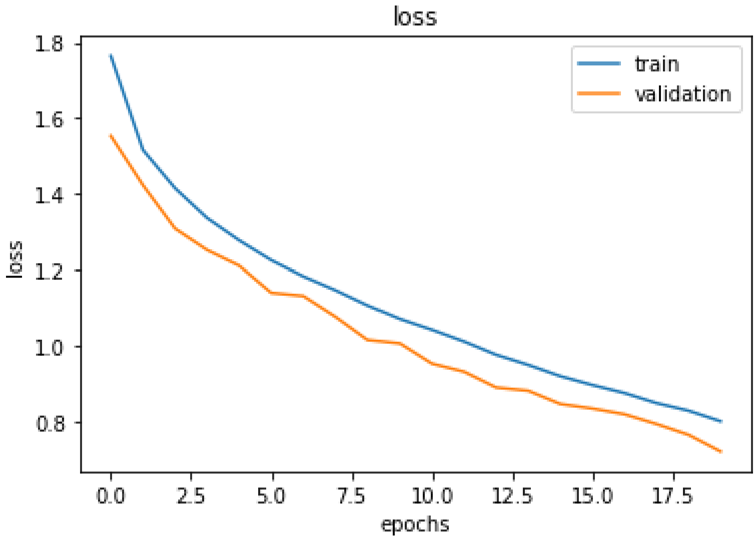
```
Epoch 1/20
391/391 [==============================] - 3s 5ms/step - loss: 1.7644 - accu
racy: 0.3898 - val_loss: 1.5529 - val_accuracy: 0.4553
Epoch 2/20
391/391 [==============================] - 2s 4ms/step - loss: 1.5158 - accu
racy: 0.4703 - val_loss: 1.4228 - val_accuracy: 0.5066
Epoch 3/20
391/391 [==============================] - 1s 3ms/step - loss: 1.4148 - accu
racy: 0.5028 - val_loss: 1.3082 - val_accuracy: 0.5472
Epoch 4/20
391/391 [==============================] - 1s 3ms/step - loss: 1.3359 - accu
racy: 0.5318 - val_loss: 1.2516 - val_accuracy: 0.5699
Epoch 5/20
391/391 [==============================] - 1s 3ms/step - loss: 1.2770 - accu
racy: 0.5544 - val_loss: 1.2103 - val_accuracy: 0.5728
Epoch 6/20
391/391 [==============================] - 1s 3ms/step - loss: 1.2247 - accu
racy: 0.5690 - val_loss: 1.1384 - val_accuracy: 0.6000
Epoch 7/20
391/391 [==============================] - 1s 3ms/step - loss: 1.1818 - accu
racy: 0.5825 - val_loss: 1.1302 - val_accuracy: 0.6050
Epoch 8/20
391/391 [==============================] - 1s 3ms/step - loss: 1.1453 - accu
racy: 0.5961 - val_loss: 1.0755 - val_accuracy: 0.6224
Epoch 9/20
391/391 [==============================] - 1s 3ms/step - loss: 1.1051 - accu
racy: 0.6099 - val_loss: 1.0143 - val_accuracy: 0.6433
Epoch 10/20
391/391 [==============================] - 2s 4ms/step - loss: 1.0697 - accu
racy: 0.6233 - val_loss: 1.0051 - val_accuracy: 0.6446
Epoch 11/20
391/391 [==============================] - 1s 3ms/step - loss: 1.0410 - accu
racy: 0.6311 - val_loss: 0.9515 - val_accuracy: 0.6638
Epoch 12/20
391/391 [==============================] - 1s 3ms/step - loss: 1.0100 - accu
racy: 0.6428 - val_loss: 0.9302 - val_accuracy: 0.6738
Epoch 13/20
391/391 [==============================] - 1s 3ms/step - loss: 0.9750 - accu
racy: 0.6558 - val_loss: 0.8885 - val_accuracy: 0.6836
Epoch 14/20
391/391 [==============================] - 1s 3ms/step - loss: 0.9482 - accu
racy: 0.6645 - val_loss: 0.8800 - val_accuracy: 0.6859
Epoch 15/20
391/391 [==============================] - 1s 3ms/step - loss: 0.9184 - accu
racy: 0.6743 - val_loss: 0.8449 - val_accuracy: 0.6993
Epoch 16/20
391/391 [==============================] - 1s 3ms/step - loss: 0.8951 - accu
racy: 0.6843 - val_loss: 0.8329 - val_accuracy: 0.7069
Epoch 17/20
391/391 [==============================] - 1s 3ms/step - loss: 0.8739 - accu
racy: 0.6899 - val_loss: 0.8178 - val_accuracy: 0.7110
Epoch 18/20
391/391 [==============================] - 1s 3ms/step - loss: 0.8473 - accu
racy: 0.6985 - val_loss: 0.7920 - val_accuracy: 0.7214
Epoch 19/20
391/391 [==============================] - 1s 3ms/step - loss: 0.8271 - accu
racy: 0.7050 - val_loss: 0.7634 - val_accuracy: 0.7293
Epoch 20/20
391/391 [==============================] - 1s 3ms/step - loss: 0.7992 - accu
racy: 0.7155 - val_loss: 0.7194 - val_accuracy: 0.7460
```

2.7) Plot a separate plots for:

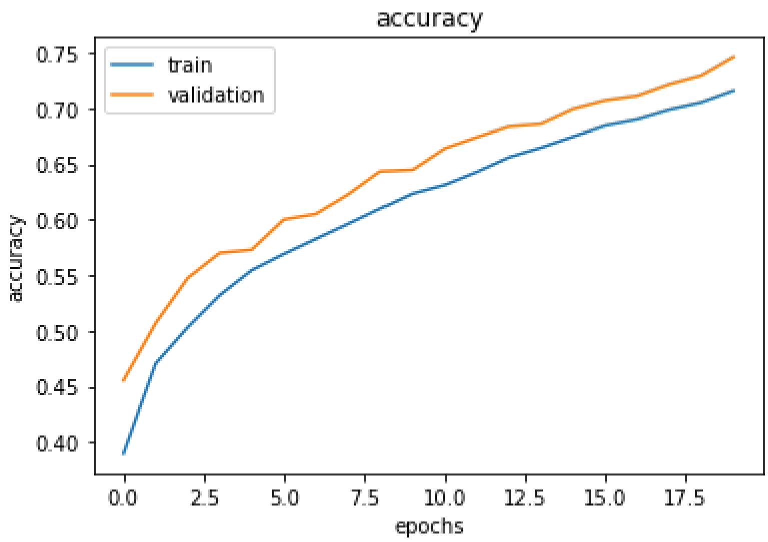
a. displaying train vs validation loss over each epoch

b. displaying train vs validation accuracy over each epoch

In [14]:
```
#Your code here
import pandas as pd
hist=pd.DataFrame(history_callback.history)
plt.plot(hist.index,hist["loss"])
plt.plot(hist.index,hist["val_loss"])
plt.xlabel("epochs")
plt.ylabel("loss")
plt.title("loss")
plt.legend(["train","validation"])
plt.show()
```



In [15]:
```
plt.plot(hist.index,hist["accuracy"])
plt.plot(hist.index,hist["val_accuracy"])
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.title("accuracy")
plt.legend(["train","validation"])
plt.show()
```



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

In [16]:
```
#Your code here
score = model.evaluate(x_test_std,y_test_tc,verbose=0)
print("Test loss: {:.3f}".format(score[0]))
print("Test accuracy: {:.3f}".format(score[1]))
```

```
Test loss: 1.690
Test accuracy: 0.507
```

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

In [17]:
```
#Your code here
from tensorflow.keras import Input
from tensorflow.keras.layers import Dropout, BatchNormalization
print("I will apply dropout and batch normalization to improve the accuracy"
model_dbn = Sequential()
model_dbn.add(Input(shape=(3072,)))
model_dbn.add(BatchNormalization())
model_dbn.add(Dense(128,activation="relu"))
model_dbn.add(Dropout(0.5))
model_dbn.add(BatchNormalization())
model_dbn.add(Dense(64,activation="relu"))
model_dbn.add(Dropout(0.5))
model_dbn.add(Dense(10,activation="softmax"))
```

```
I will apply dropout and batch normalization to improve the accuracy
```

In [18]:
```
model_dbn.compile(optimizer="adam",loss="categorical_crossentropy",metrics=[
history_callback_dropout = model_dbn.fit(x_dev_std,y_dev_tc,batch_size=128,e
```

```
Epoch 1/20
391/391 [==============================] - 4s 8ms/step - loss: 2.1304 - accu
racy: 0.2544 - val_loss: 1.7542 - val_accuracy: 0.3898
Epoch 2/20
391/391 [==============================] - 2s 6ms/step - loss: 1.8630 - accu
racy: 0.3311 - val_loss: 1.6559 - val_accuracy: 0.4271
Epoch 3/20
391/391 [==============================] - 2s 6ms/step - loss: 1.7784 - accu
racy: 0.3639 - val_loss: 1.5956 - val_accuracy: 0.4435
Epoch 4/20
391/391 [==============================] - 3s 7ms/step - loss: 1.7367 - accu
racy: 0.3740 - val_loss: 1.5511 - val_accuracy: 0.4573
Epoch 5/20
391/391 [==============================] - 3s 7ms/step - loss: 1.6982 - accu
racy: 0.3909 - val_loss: 1.5063 - val_accuracy: 0.4736
Epoch 6/20
391/391 [==============================] - 3s 9ms/step - loss: 1.6762 - accu
racy: 0.3987 - val_loss: 1.4871 - val_accuracy: 0.4776
Epoch 7/20
391/391 [==============================] - 4s 9ms/step - loss: 1.6543 - accu
racy: 0.4045 - val_loss: 1.4623 - val_accuracy: 0.4848
Epoch 8/20
391/391 [==============================] - 3s 7ms/step - loss: 1.6384 - accu
racy: 0.4139 - val_loss: 1.4480 - val_accuracy: 0.4925
Epoch 9/20
391/391 [==============================] - 3s 7ms/step - loss: 1.6228 - accu
racy: 0.4183 - val_loss: 1.4296 - val_accuracy: 0.5012
Epoch 10/20
391/391 [==============================] - 3s 7ms/step - loss: 1.6105 - accu
racy: 0.4246 - val_loss: 1.4103 - val_accuracy: 0.5058
Epoch 11/20
391/391 [==============================] - 3s 7ms/step - loss: 1.5954 - accu
racy: 0.4309 - val_loss: 1.3920 - val_accuracy: 0.5122
Epoch 12/20
391/391 [==============================] - 3s 6ms/step - loss: 1.5834 - accu
racy: 0.4356 - val_loss: 1.3912 - val_accuracy: 0.5138
Epoch 13/20
391/391 [==============================] - 3s 7ms/step - loss: 1.5749 - accu
racy: 0.4362 - val_loss: 1.3742 - val_accuracy: 0.5233
Epoch 14/20
391/391 [==============================] - 3s 7ms/step - loss: 1.5652 - accu
racy: 0.4409 - val_loss: 1.3546 - val_accuracy: 0.5244
Epoch 15/20
391/391 [==============================] - 3s 7ms/step - loss: 1.5580 - accu
racy: 0.4433 - val_loss: 1.3493 - val_accuracy: 0.5242
Epoch 16/20
391/391 [==============================] - 3s 7ms/step - loss: 1.5527 - accu
racy: 0.4454 - val_loss: 1.3527 - val_accuracy: 0.5307
Epoch 17/20
391/391 [==============================] - 3s 7ms/step - loss: 1.5473 - accu
racy: 0.4484 - val_loss: 1.3348 - val_accuracy: 0.5363
Epoch 18/20
391/391 [==============================] - 3s 7ms/step - loss: 1.5423 - accu
racy: 0.4487 - val_loss: 1.3249 - val_accuracy: 0.5379
Epoch 19/20
391/391 [==============================] - 3s 7ms/step - loss: 1.5326 - accu
racy: 0.4526 - val_loss: 1.3202 - val_accuracy: 0.5431
Epoch 20/20
391/391 [==============================] - 3s 7ms/step - loss: 1.5254 - accu
racy: 0.4534 - val_loss: 1.3187 - val_accuracy: 0.5407
```

In [19]:

```
score_dbn = model_dbn.evaluate(x_test_std,y_test_tc,verbose=0)
print("Test loss: {:.3f}".format(score_dbn[0]))
print("Test accuracy: {:.3f}".format(score_dbn[1]))
```

```
Test loss: 1.407
Test accuracy: 0.502
```

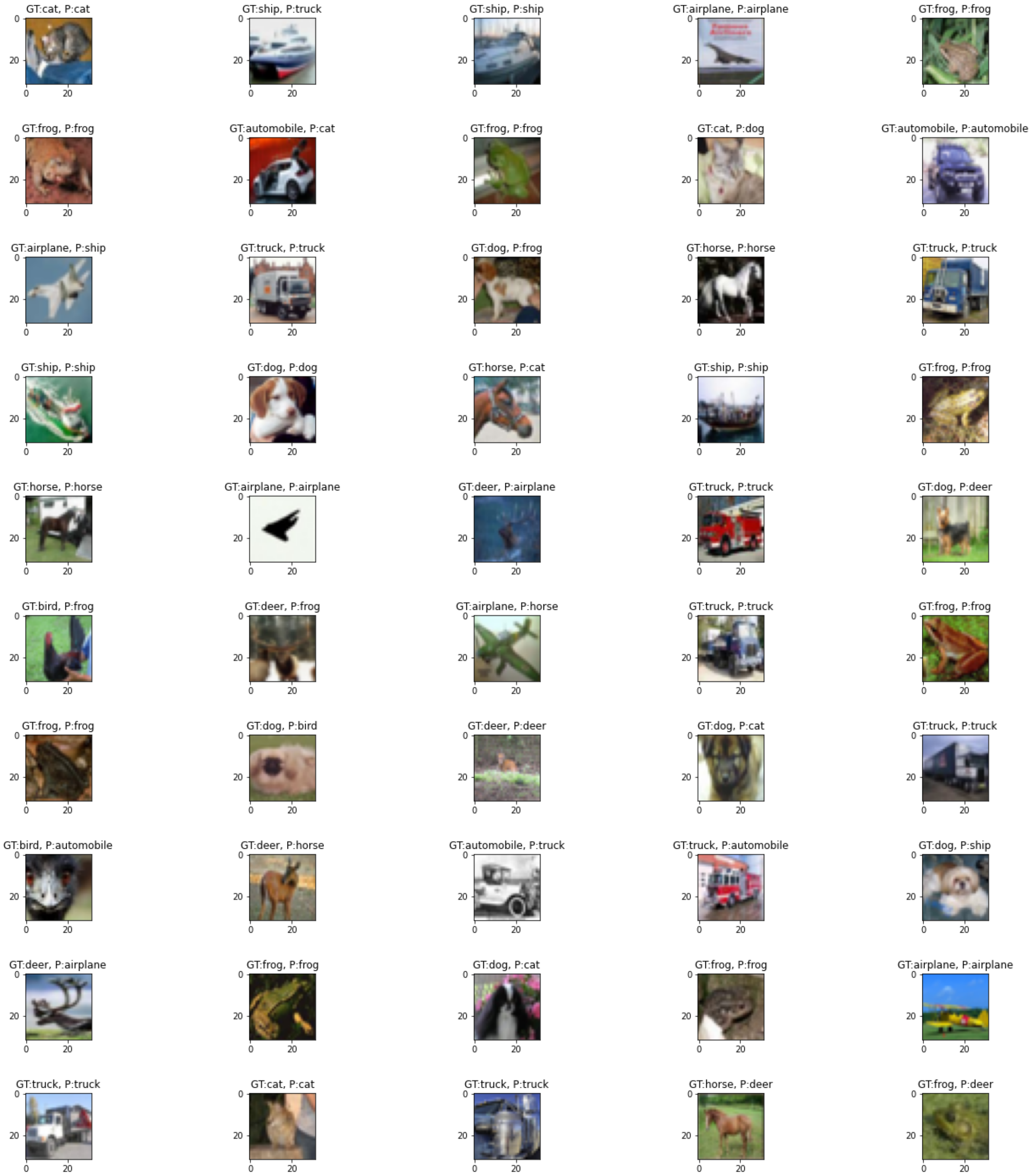
2.10 Plot the first 50 samples of test dataset on a 10*5 subplot and this time label the images with both the ground truth (GT) and predicted class (P). (Make sure you predict the class with the improved model)

In [20]:
```
y_pred = model_dbn.predict(x_test_std[:50])
y_pred_class = y_pred.argmax(axis=-1)
y_pred_class
```

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

Out[20]:
```
array([3, 9, 8, 0, 6, 6, 3, 6, 5, 1, 8, 9, 6, 7, 9, 8, 5, 3, 8, 6, 7, 0,
       0, 9, 4, 6, 6, 7, 9, 6, 6, 2, 4, 3, 9, 1, 7, 9, 1, 8, 0, 6, 3, 6,
       0, 9, 3, 9, 4, 4])
```

In [21]:
```
#Your code here
fig = plt.figure(figsize=(20,20))

for i in range(50):
    fig.tight_layout(pad=3)
    fig.add_subplot(10,5,i+1).set_title(f"GT:{LABELS[y_test[i][0]]}, P:{LABE
    plt.imshow(x_test[i])
```



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 [22]:
```
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.

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
```

```
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 [23]:
```
# your code here
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten
cnn = Sequential()

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

3.2 Report the model summary

In [24]:
```
#your code here
cnn.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 6)         456

 max_pooling2d (MaxPooling2D  (None, 16, 16, 6)        0
 )

 conv2d_1 (Conv2D)           (None, 12, 12, 16)        2416

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 16)         0
 2D)

 conv2d_2 (Conv2D)           (None, 2, 2, 120)         48120

 flatten (Flatten)           (None, 480)               0

 dense_6 (Dense)             (None, 84)                40404

 dense_7 (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.

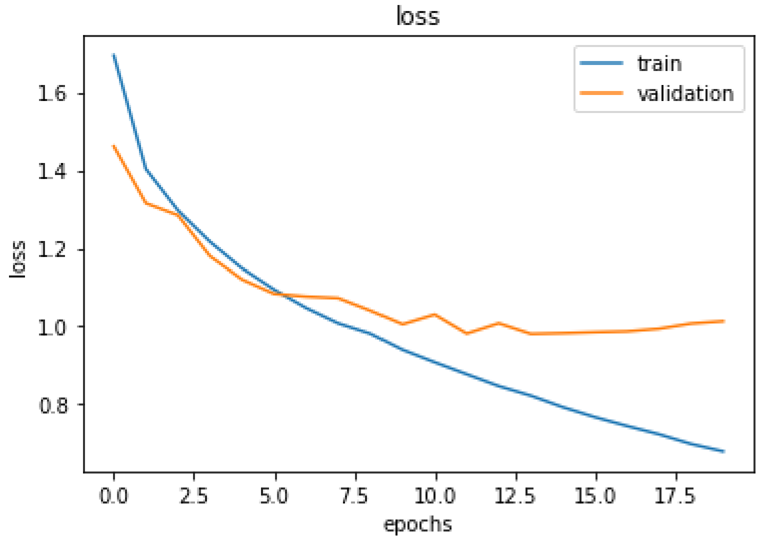
In [25]:
```
# Your code here
cnn.compile("adam","categorical_crossentropy",metrics=['accuracy'])
history_cnn = cnn.fit(x_dev,to_categorical(y_dev,10),batch_size=128,epochs=2
```

```
Epoch 1/20
352/352 [==============================] - 17s 46ms/step - loss: 1.6943 - ac
curacy: 0.3780 - val_loss: 1.4611 - val_accuracy: 0.4636
Epoch 2/20
352/352 [==============================] - 16s 47ms/step - loss: 1.4030 - ac
curacy: 0.4930 - val_loss: 1.3153 - val_accuracy: 0.5274
Epoch 3/20
352/352 [==============================] - 17s 47ms/step - loss: 1.2969 - ac
curacy: 0.5346 - val_loss: 1.2842 - val_accuracy: 0.5376
Epoch 4/20
352/352 [==============================] - 16s 46ms/step - loss: 1.2166 - ac
curacy: 0.5680 - val_loss: 1.1798 - val_accuracy: 0.5816
Epoch 5/20
352/352 [==============================] - 17s 48ms/step - loss: 1.1486 - ac
curacy: 0.5928 - val_loss: 1.1199 - val_accuracy: 0.6060
Epoch 6/20
352/352 [==============================] - 16s 47ms/step - loss: 1.0941 - ac
curacy: 0.6101 - val_loss: 1.0833 - val_accuracy: 0.6230
Epoch 7/20
352/352 [==============================] - 16s 46ms/step - loss: 1.0473 - ac
curacy: 0.6275 - val_loss: 1.0762 - val_accuracy: 0.6220
Epoch 8/20
352/352 [==============================] - 17s 47ms/step - loss: 1.0080 - ac
curacy: 0.6447 - val_loss: 1.0724 - val_accuracy: 0.6210
Epoch 9/20
352/352 [==============================] - 16s 46ms/step - loss: 0.9814 - ac
curacy: 0.6533 - val_loss: 1.0405 - val_accuracy: 0.6422
Epoch 10/20
352/352 [==============================] - 18s 50ms/step - loss: 0.9400 - ac
curacy: 0.6677 - val_loss: 1.0056 - val_accuracy: 0.6552
Epoch 11/20
352/352 [==============================] - 17s 48ms/step - loss: 0.9080 - ac
curacy: 0.6785 - val_loss: 1.0305 - val_accuracy: 0.6456
Epoch 12/20
352/352 [==============================] - 17s 48ms/step - loss: 0.8772 - ac
curacy: 0.6898 - val_loss: 0.9816 - val_accuracy: 0.6568
Epoch 13/20
352/352 [==============================] - 17s 47ms/step - loss: 0.8468 - ac
curacy: 0.7020 - val_loss: 1.0081 - val_accuracy: 0.6580
Epoch 14/20
352/352 [==============================] - 17s 48ms/step - loss: 0.8222 - ac
curacy: 0.7106 - val_loss: 0.9808 - val_accuracy: 0.6648
Epoch 15/20
352/352 [==============================] - 17s 47ms/step - loss: 0.7927 - ac
curacy: 0.7220 - val_loss: 0.9826 - val_accuracy: 0.6700
Epoch 16/20
352/352 [==============================] - 16s 47ms/step - loss: 0.7671 - ac
curacy: 0.7300 - val_loss: 0.9851 - val_accuracy: 0.6702
Epoch 17/20
352/352 [==============================] - 17s 48ms/step - loss: 0.7444 - ac
curacy: 0.7389 - val_loss: 0.9872 - val_accuracy: 0.6728
Epoch 18/20
352/352 [==============================] - 17s 47ms/step - loss: 0.7231 - ac
curacy: 0.7463 - val_loss: 0.9941 - val_accuracy: 0.6692
Epoch 19/20
352/352 [==============================] - 17s 48ms/step - loss: 0.6982 - ac
curacy: 0.7546 - val_loss: 1.0074 - val_accuracy: 0.6700
Epoch 20/20
352/352 [==============================] - 17s 49ms/step - loss: 0.6793 - ac
curacy: 0.7615 - val_loss: 1.0133 - val_accuracy: 0.6682
```

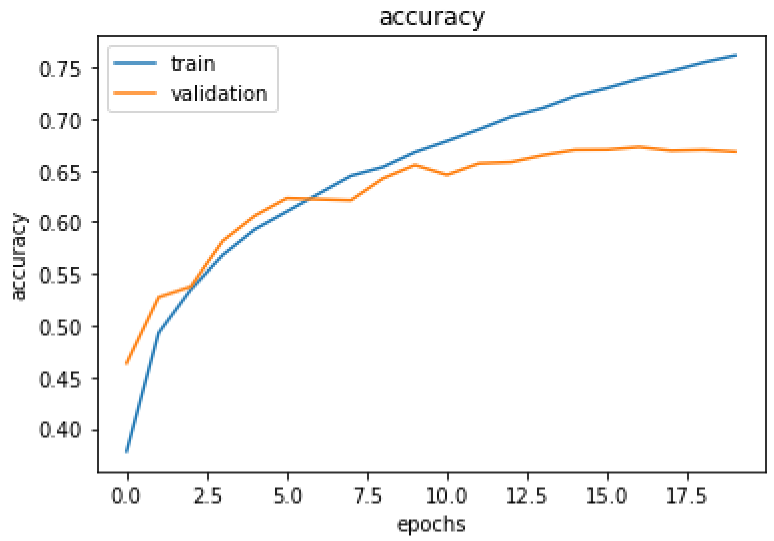
In [26]:

```
hist_cnn=pd.DataFrame(history_cnn.history)
plt.plot(hist_cnn.index,hist_cnn["loss"])
plt.plot(hist_cnn.index,hist_cnn["val_loss"])
```

```
plt.xlabel("epochs")
plt.ylabel("loss")
plt.title("loss")
plt.legend(["train","validation"])
plt.show()
```



In [27]:
```
plt.plot(hist_cnn.index,hist_cnn["accuracy"])
plt.plot(hist_cnn.index,hist_cnn["val_accuracy"])
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.title("accuracy")
plt.legend(["train","validation"])
plt.show()
```



In [28]:
```
score_cnn = cnn.evaluate(x_test,to_categorical(y_test,10),verbose=0)
print("Test loss: {:.3f}".format(score_cnn[0]))
print("Test accuracy: {:.3f}".format(score_cnn[1]))
```

```
Test loss: 1.089
Test accuracy: 0.643
```

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?

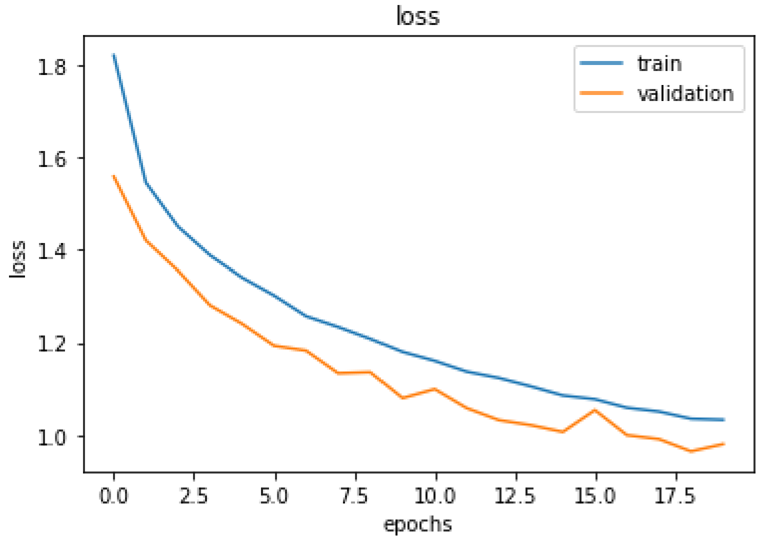
In [29]:
```
# Your code here
cnn_drop = Sequential()

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

In [30]:
```
cnn_drop.compile("adam","categorical_crossentropy",metrics=['accuracy'])
history_cnndp = cnn_drop.fit(x_dev,to_categorical(y_dev,10),batch_size=128,e
```

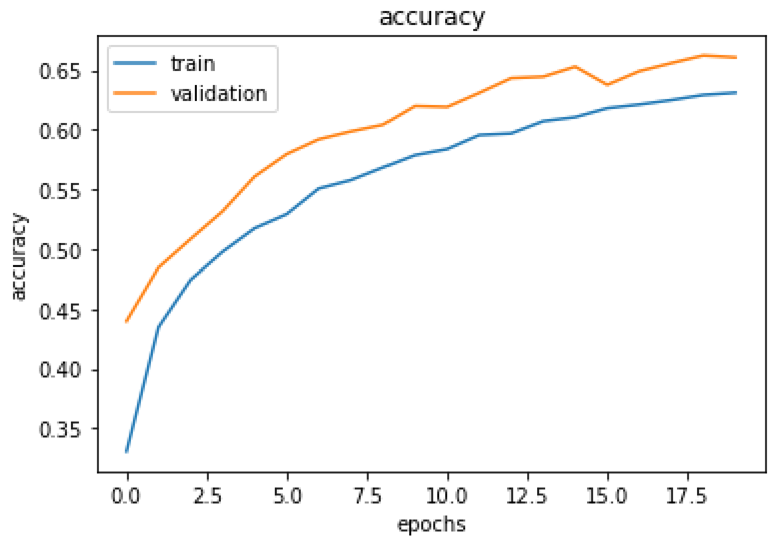
```
Epoch 1/20
352/352 [==============================] - 18s 51ms/step - loss: 1.8210 - ac
curacy: 0.3305 - val_loss: 1.5589 - val_accuracy: 0.4398
Epoch 2/20
352/352 [==============================] - 17s 49ms/step - loss: 1.5460 - ac
curacy: 0.4348 - val_loss: 1.4210 - val_accuracy: 0.4846
Epoch 3/20
352/352 [==============================] - 17s 49ms/step - loss: 1.4510 - ac
curacy: 0.4742 - val_loss: 1.3558 - val_accuracy: 0.5084
Epoch 4/20
352/352 [==============================] - 17s 48ms/step - loss: 1.3890 - ac
curacy: 0.4978 - val_loss: 1.2811 - val_accuracy: 0.5316
Epoch 5/20
352/352 [==============================] - 17s 49ms/step - loss: 1.3408 - ac
curacy: 0.5174 - val_loss: 1.2412 - val_accuracy: 0.5608
Epoch 6/20
352/352 [==============================] - 18s 50ms/step - loss: 1.3021 - ac
curacy: 0.5292 - val_loss: 1.1936 - val_accuracy: 0.5796
Epoch 7/20
352/352 [==============================] - 17s 49ms/step - loss: 1.2575 - ac
curacy: 0.5507 - val_loss: 1.1830 - val_accuracy: 0.5920
Epoch 8/20
352/352 [==============================] - 17s 48ms/step - loss: 1.2342 - ac
curacy: 0.5578 - val_loss: 1.1341 - val_accuracy: 0.5986
Epoch 9/20
352/352 [==============================] - 17s 49ms/step - loss: 1.2084 - ac
curacy: 0.5684 - val_loss: 1.1363 - val_accuracy: 0.6042
Epoch 10/20
352/352 [==============================] - 17s 49ms/step - loss: 1.1803 - ac
curacy: 0.5788 - val_loss: 1.0803 - val_accuracy: 0.6200
Epoch 11/20
352/352 [==============================] - 17s 49ms/step - loss: 1.1609 - ac
curacy: 0.5838 - val_loss: 1.0997 - val_accuracy: 0.6192
Epoch 12/20
352/352 [==============================] - 17s 49ms/step - loss: 1.1375 - ac
curacy: 0.5956 - val_loss: 1.0585 - val_accuracy: 0.6310
Epoch 13/20
352/352 [==============================] - 17s 48ms/step - loss: 1.1238 - ac
curacy: 0.5970 - val_loss: 1.0326 - val_accuracy: 0.6434
Epoch 14/20
352/352 [==============================] - 17s 49ms/step - loss: 1.1057 - ac
curacy: 0.6072 - val_loss: 1.0217 - val_accuracy: 0.6446
Epoch 15/20
352/352 [==============================] - 17s 49ms/step - loss: 1.0860 - ac
curacy: 0.6106 - val_loss: 1.0070 - val_accuracy: 0.6530
Epoch 16/20
352/352 [==============================] - 17s 49ms/step - loss: 1.0778 - ac
curacy: 0.6182 - val_loss: 1.0542 - val_accuracy: 0.6378
Epoch 17/20
352/352 [==============================] - 17s 49ms/step - loss: 1.0593 - ac
curacy: 0.6212 - val_loss: 1.0000 - val_accuracy: 0.6492
Epoch 18/20
352/352 [==============================] - 18s 51ms/step - loss: 1.0511 - ac
curacy: 0.6250 - val_loss: 0.9914 - val_accuracy: 0.6560
Epoch 19/20
352/352 [==============================] - 17s 50ms/step - loss: 1.0355 - ac
curacy: 0.6293 - val_loss: 0.9649 - val_accuracy: 0.6624
Epoch 20/20
352/352 [==============================] - 18s 52ms/step - loss: 1.0333 - ac
curacy: 0.6312 - val_loss: 0.9805 - val_accuracy: 0.6608
```

In [31]:
```
hist_cnndp=pd.DataFrame(history_cnndp.history)
plt.plot(hist_cnndp.index,hist_cnndp["loss"])
plt.plot(hist_cnndp.index,hist_cnndp["val_loss"])
```

```
plt.xlabel("epochs")
plt.ylabel("loss")
plt.title("loss")
plt.legend(["train","validation"])
plt.show()
```



In [32]:
```
plt.plot(hist_cnndp.index,hist_cnndp["accuracy"])
plt.plot(hist_cnndp.index,hist_cnndp["val_accuracy"])
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.title("accuracy")
plt.legend(["train","validation"])
plt.show()
```



In [33]:
```
cnn_bn = Sequential()

cnn_bn.add(Conv2D(6,kernel_size=(5,5),activation='relu',input_shape=(32,32,3
cnn_bn.add(MaxPooling2D(pool_size=(2,2)))
cnn_bn.add(BatchNormalization())
cnn_bn.add(Conv2D(16,kernel_size=(5,5),activation='relu',padding="valid"))
cnn_bn.add(MaxPooling2D(pool_size=(2,2)))
cnn_bn.add(BatchNormalization())
cnn_bn.add(Conv2D(120,kernel_size=(5,5),activation='relu'))
cnn_bn.add(Flatten())
cnn_bn.add(Dense(84,activation='relu'))
cnn_bn.add(Dense(10,activation='softmax'))
```

In [34]:
```
cnn_bn.compile("adam","categorical_crossentropy",metrics=['accuracy'])
```

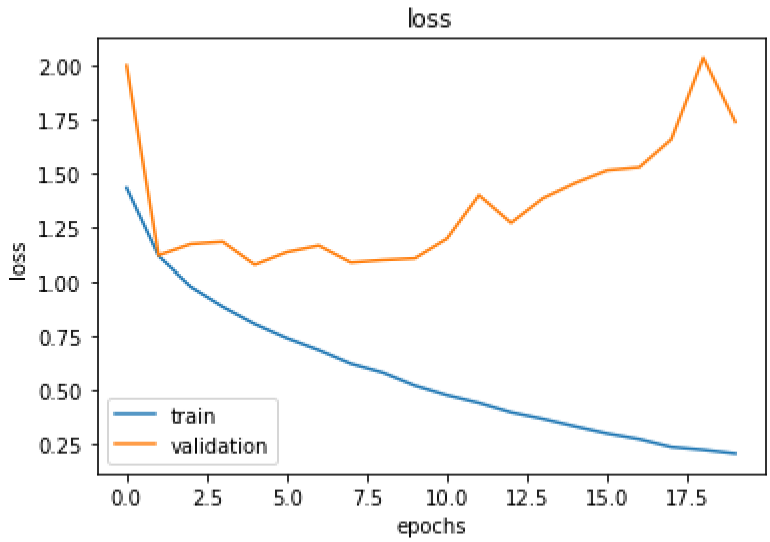
```
history_cnnbn = cnn_bn.fit(x_dev,to_categorical(y_dev,10),batch_size=128,epc
```

```
Epoch 1/20
352/352 [==============================] - 20s 54ms/step - loss: 1.4323 - ac
curacy: 0.4880 - val_loss: 1.9981 - val_accuracy: 0.3508
Epoch 2/20
352/352 [==============================] - 19s 54ms/step - loss: 1.1182 - ac
curacy: 0.6018 - val_loss: 1.1208 - val_accuracy: 0.6060
Epoch 3/20
352/352 [==============================] - 19s 54ms/step - loss: 0.9786 - ac
curacy: 0.6521 - val_loss: 1.1727 - val_accuracy: 0.5894
Epoch 4/20
352/352 [==============================] - 20s 57ms/step - loss: 0.8865 - ac
curacy: 0.6880 - val_loss: 1.1833 - val_accuracy: 0.5890
Epoch 5/20
352/352 [==============================] - 21s 60ms/step - loss: 0.8071 - ac
curacy: 0.7155 - val_loss: 1.0771 - val_accuracy: 0.6254
Epoch 6/20
352/352 [==============================] - 19s 53ms/step - loss: 0.7414 - ac
curacy: 0.7396 - val_loss: 1.1352 - val_accuracy: 0.6098
Epoch 7/20
352/352 [==============================] - 19s 53ms/step - loss: 0.6864 - ac
curacy: 0.7562 - val_loss: 1.1655 - val_accuracy: 0.6124
Epoch 8/20
352/352 [==============================] - 20s 56ms/step - loss: 0.6235 - ac
curacy: 0.7790 - val_loss: 1.0872 - val_accuracy: 0.6510
Epoch 9/20
352/352 [==============================] - 19s 55ms/step - loss: 0.5820 - ac
curacy: 0.7941 - val_loss: 1.0986 - val_accuracy: 0.6536
Epoch 10/20
352/352 [==============================] - 19s 55ms/step - loss: 0.5222 - ac
curacy: 0.8150 - val_loss: 1.1064 - val_accuracy: 0.6572
Epoch 11/20
352/352 [==============================] - 19s 54ms/step - loss: 0.4777 - ac
curacy: 0.8326 - val_loss: 1.1976 - val_accuracy: 0.6326
Epoch 12/20
352/352 [==============================] - 20s 57ms/step - loss: 0.4421 - ac
curacy: 0.8429 - val_loss: 1.3982 - val_accuracy: 0.5932
Epoch 13/20
352/352 [==============================] - 20s 55ms/step - loss: 0.3984 - ac
curacy: 0.8591 - val_loss: 1.2704 - val_accuracy: 0.6414
Epoch 14/20
352/352 [==============================] - 18s 52ms/step - loss: 0.3676 - ac
curacy: 0.8690 - val_loss: 1.3843 - val_accuracy: 0.6522
Epoch 15/20
352/352 [==============================] - 19s 53ms/step - loss: 0.3331 - ac
curacy: 0.8815 - val_loss: 1.4547 - val_accuracy: 0.6150
Epoch 16/20
352/352 [==============================] - 19s 53ms/step - loss: 0.3002 - ac
curacy: 0.8930 - val_loss: 1.5131 - val_accuracy: 0.6454
Epoch 17/20
352/352 [==============================] - 19s 53ms/step - loss: 0.2742 - ac
curacy: 0.9029 - val_loss: 1.5273 - val_accuracy: 0.6300
Epoch 18/20
352/352 [==============================] - 20s 57ms/step - loss: 0.2380 - ac
curacy: 0.9160 - val_loss: 1.6562 - val_accuracy: 0.6352
Epoch 19/20
352/352 [==============================] - 21s 58ms/step - loss: 0.2246 - ac
curacy: 0.9202 - val_loss: 2.0335 - val_accuracy: 0.5960
Epoch 20/20
352/352 [==============================] - 20s 56ms/step - loss: 0.2078 - ac
curacy: 0.9261 - val_loss: 1.7377 - val_accuracy: 0.6166
```

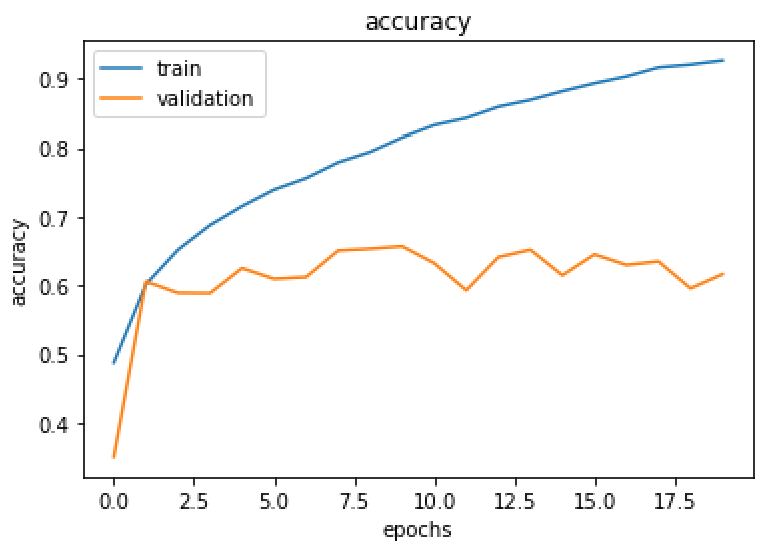
In [35]:

```
hist_cnnbn=pd.DataFrame(history_cnnbn.history)
```

```
plt.plot(hist_cnnbn.index,hist_cnnbn["loss"])
plt.plot(hist_cnnbn.index,hist_cnnbn["val_loss"])
plt.xlabel("epochs")
plt.ylabel("loss")
plt.title("loss")
plt.legend(["train","validation"])
plt.show()
```



In [36]:
```
plt.plot(hist_cnnbn.index,hist_cnnbn["accuracy"])
plt.plot(hist_cnnbn.index,hist_cnnbn["val_accuracy"])
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.title("accuracy")
plt.legend(["train","validation"])
plt.show()
```



In [37]:
```
score_cnndp = cnn_drop.evaluate(x_test,to_categorical(y_test,10),verbose=0)
score_cnnbn = cnn_bn.evaluate(x_test,to_categorical(y_test,10),verbose=0)
print(f"The loss and accuracy with dropout are {score_cnndp[0]} and {score_c
print(f"The loss and accuracy with batch normalization are {score_cnnbn[0]}
print("Dropout helps with overfitting better.")
```

```
The loss and accuracy with dropout are 1.0076313018798828 and 0.648699998855
5908.
The loss and accuracy with batch normalization are 1.7768703699111938 and 0.
6223999857902527.
Dropout helps with overfitting better.
```

In [ ]:

```
```