In [1]:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=94731898
9803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%
3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20https%3a%2f%2fwww.googleapis.co
m%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive%20https%3a%2f%2fww
w.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2fwww.googleapis.com%2fauth
%2fpeopleapi.readonly

Enter your authorization code:
··········
Mounted at /content/drive
```

We unzip the three datasets

In [0]:

```
!unzip -qq "/content/drive/My Drive/SDS Group Stuff/craigslistVehicles.csv.zip"
```

In [0]:

```
!unzip -qq "/content/drive/My Drive/SDS Group Stuff/sorted_car_images.zip"
```

In [0]:

```
!unzip -qq "/content/drive/My Drive/SDS Group Stuff/stanford_cars_dataset.zip"
```

In [4]:

```
# We make the necessary imports
import pandas as pd
import numpy as np
import gdown
from IPython.display import clear_output
from skimage import io
import matplotlib.pyplot as plt
from keras.preprocessing import image
from keras.preprocessing.image import save_img
from sklearn.preprocessing import StandardScaler
from sklearn.utils import random
import seaborn as sns
import matplotlib.pyplot as plt
import httplib2

sns.set()
```

```
Using TensorFlow backend.
```

The default version of TensorFlow in Colab will soon switch to TensorFlow 2.x.
We recommend you upgrade now or ensure your notebook will continue to use TensorFlow 1.x via the `%tensorflow_version 1.x` magic: more info.

In [0]:

```
# We load in our two datasets
data = pd.read_csv("craigslistVehicles.csv")
index_data = pd.read_csv("/content/drive/My Drive/SDS Group Stuff/sorted_image_list.csv")
```

Data Preperation

Here we prepare the data. We start out by removing the rows where the columns we deemed important for determining the price of a car is NaN. Next filter the price. As we can see, the most expensive car in the dataset

determining the price of a car is hard. Next filter the price. As we can see, the most expensive car in the dataset is about 3.7 billion dollars and some of them are free. We therefor set a mininum limit of 200 dollars (199) and find a suitable maximum using the 99.5 percentile, which is about 60.000 dollars

In [0]:

```
data =  data.dropna(subset = ["odometer", "manufacturer", "year", "transmission", "fuel"
, "cylinders", "drive", "type", "condition"])
data = data.reset_index(drop=True)
```

In [7]:

```
print("most expensive car:", data["price"].max(), "dollars")
data = data[data["price"] <= 60000]
data = data[data["price"] >= 199]
print("most expensive car after limiting the data:", data["price"].max(), "dollars")
```

```
most expensive car: 2793940305 dollars
most expensive car after limiting the data: 60000 dollars
```

Here we showcase our price class determination for each car. We spread each price class with a difference of 750 dollars, which make up to 80 price classes, since the most expensive car is 59.995. The reason we do this is to make a more precise and a better estimate of a price for a new car in the future, with our neural network. The price column alone has many different unique values, and is therefore made more compact and more precise.

We end this by mapping the new found price class to a string, that describes the price range.

In [0]:

```
category_span = 750
```

In [9]:

```
floor_price_class = int(data["price"].max() / category_span)
print("class:", (76) * category_span, "to", (76 + 1) * category_span);
```

```
class: 57000 to 57750
```

We save the price class, both as numerical data, and categorical, in order to create indicator variables later with `pd.get_dummies()` .

In [0]:

```
data["price_class"] = data["price"] / category_span
data["price_class"] = data["price_class"].astype(int)
```

In [0]:

```
def price_class_to_string(i):
  return "{0} to {1}".format(i * category_span, (i + 1) * category_span)
```

In [0]:

```
data["price_class_label"] = data["price_class"].apply(lambda x:  price_class_to_string(x
))
```

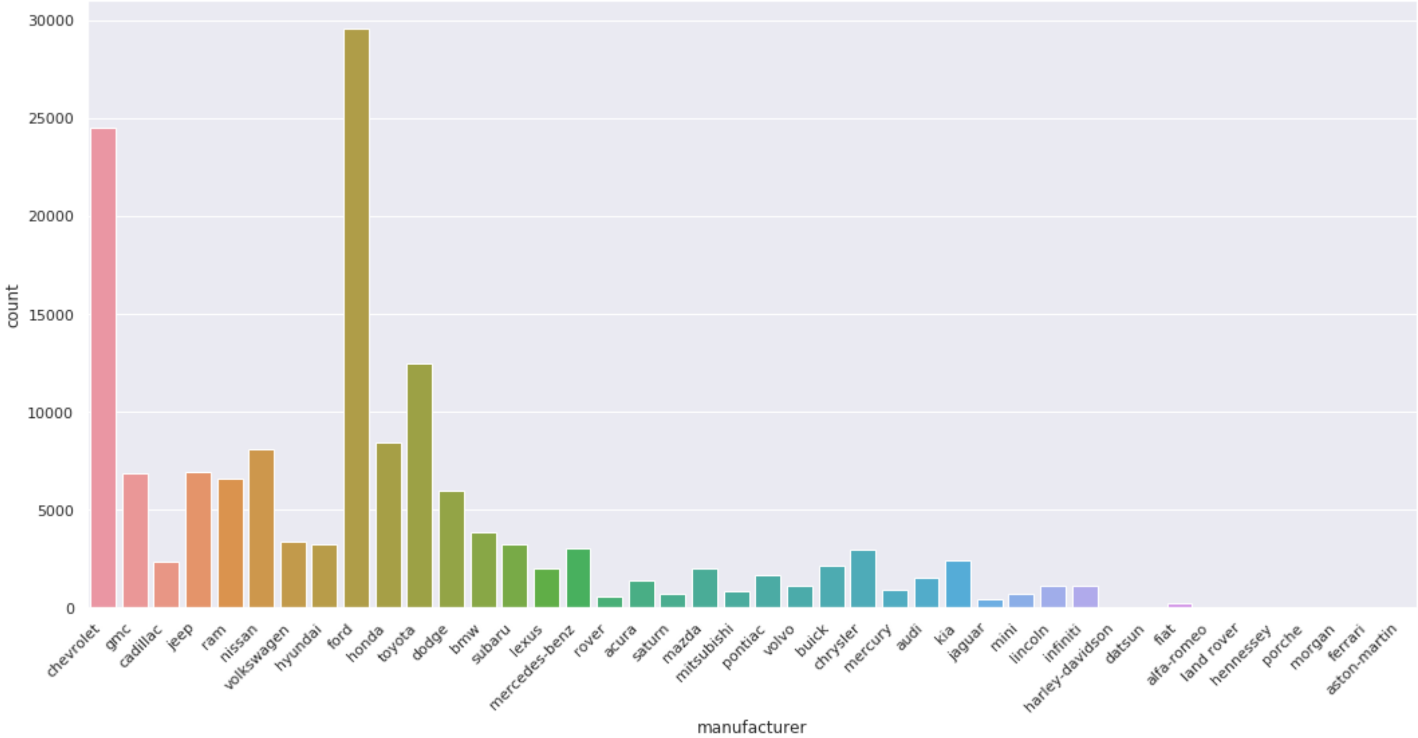
Data Exploration

People seem to like selling Ford and Chevrolet cars in particular it seems.

In [13]:

```
plt.figure(figsize=(15, 8))
chart = sns.countplot(x=data["manufacturer"])
chart.set_xticklabels(chart.get_xticklabels(), rotation=45, ha="right")
plt.tight_layout()
```

```
plt.show()
```



The distribution of the prices in the dataset reveal that about 55% of the cars costs less than 10.000 dollars

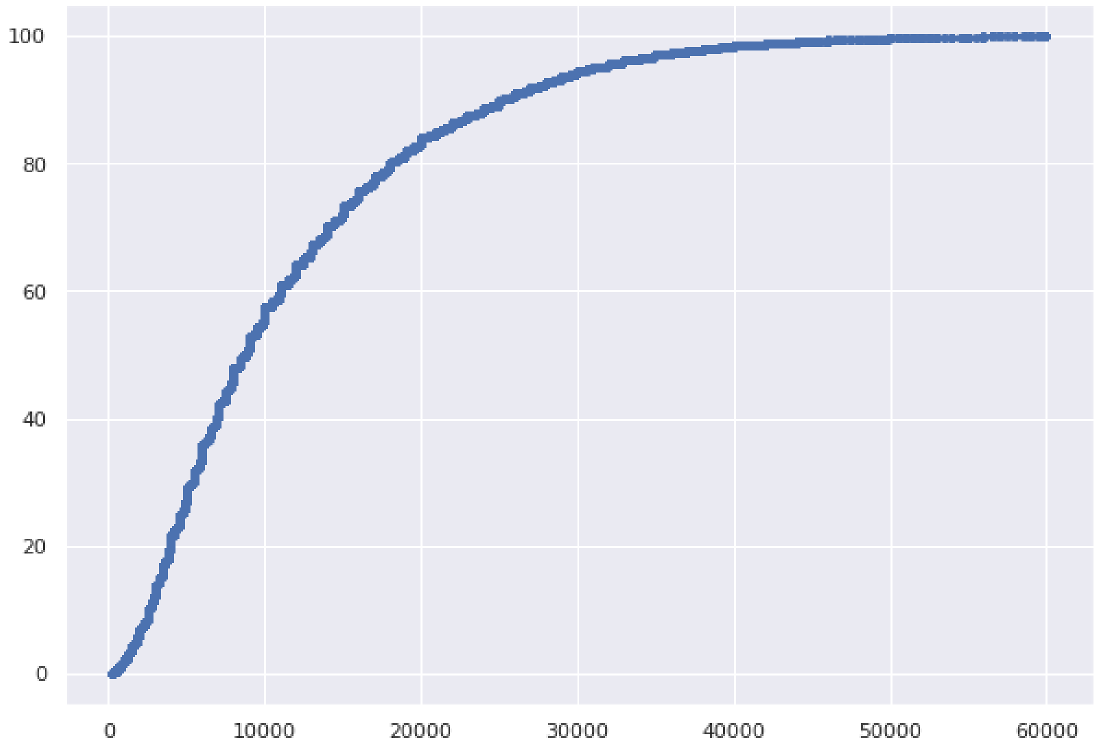
In [14]:

```
# distribution of the price in our dataset
sorted_price = data["price"].sort_values()
percentage = np.linspace(0, 100, num=len(data))

plt.subplots(figsize=(10, 7))
plt.plot(sorted_price, percentage, marker=".")
```

Out[14]:

```
[<matplotlib.lines.Line2D at 0x7f8ad7e0d5f8>]
```

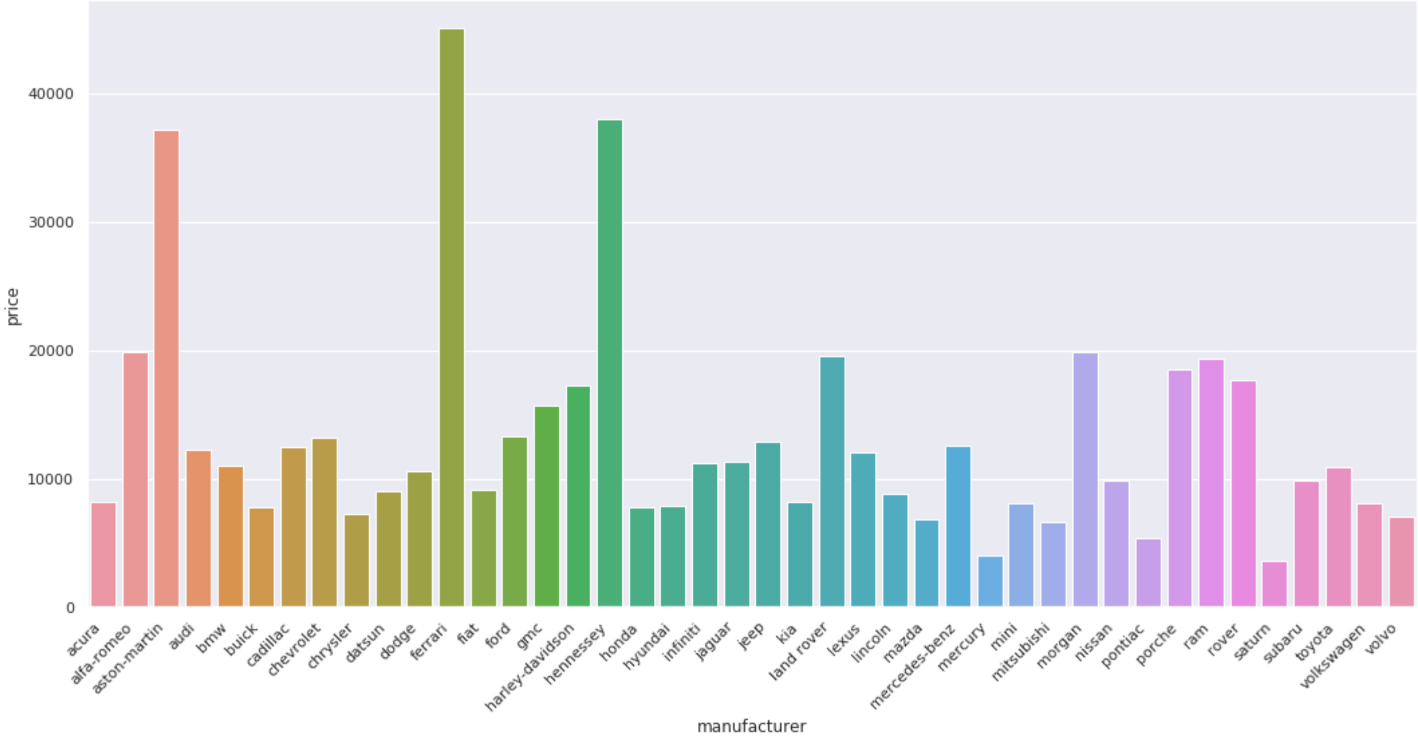


Aston martin, Ferrari and Hennessey are the top 3 most expensive brands an average, where as Mercury, Saturn and Pontiac are the 3 cheapest brands on average, in the dataset.

In [15]:

```
manufacturer_price_mean = data.groupby(["manufacturer"]).mean()["price"]

plt.figure(figsize=(15, 8))
chart = sns.barplot(x = manufacturer_price_mean.index, y = manufacturer_price_mean)
chart.set_xticklabels(chart.get_xticklabels(), rotation=45, ha="right")
plt.tight_layout()
plt.show()
```



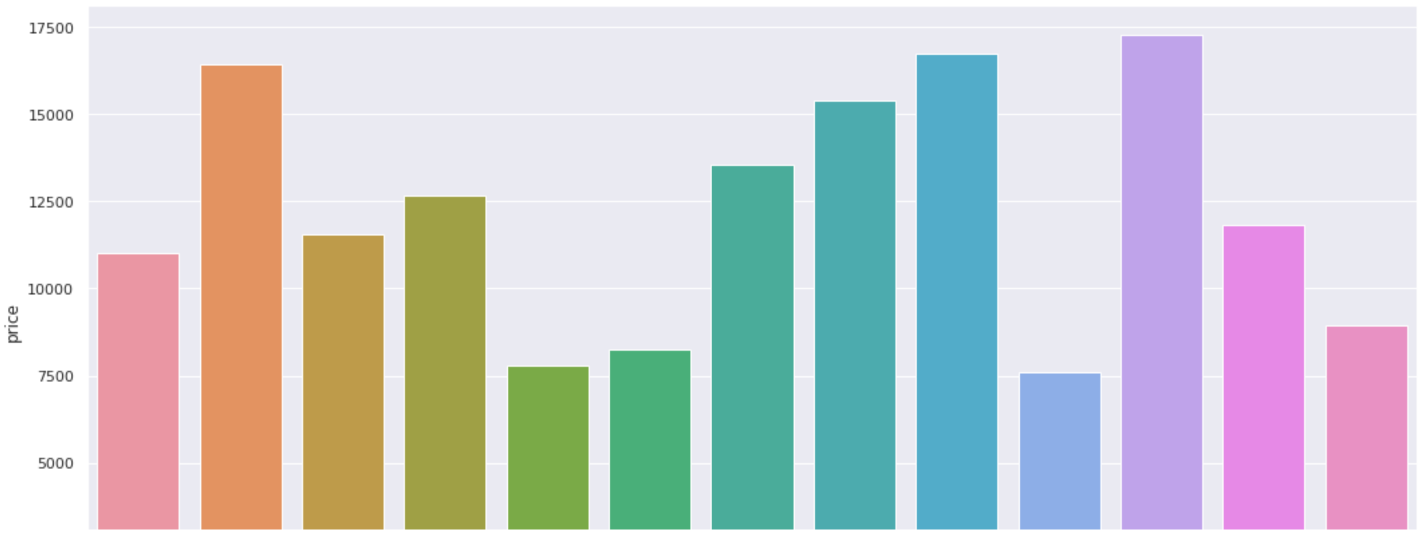
As for the average price per type, we see that trucks pickups and busses are the three most expensive types of cars in the dataset.

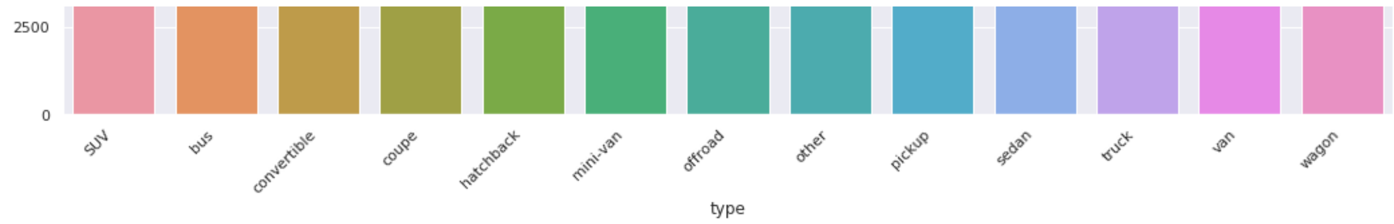
When we look at the earlier graph with price per manufacturer, we saw that Ferrari was the most expensive manufacturer on average, while there isn't all that many of them. This also indicates that although the Ferrari car is expensive, it shares type with a lot of cheap manufacturers, thereby not contributing all that much to the overall price for the type.

In [16]:

```
type_price_mean = data.groupby(["type"]).mean()["price"]

plt.figure(figsize=(15, 8))
chart = sns.barplot(x = type_price_mean.index, y = type_price_mean)
chart.set_xticklabels(chart.get_xticklabels(), rotation=45, ha="right")
plt.tight_layout()
plt.show()
```





As we can see here, the correlation between the year and the price class, which we invented ourselves, is quite high. This suggests that newer cars in general tend to be more expensive.

In [17]:

```
corr_data = pd.get_dummies(data[["price_class", "year", "odometer", "size", "paint_color
", "transmission", "manufacturer", "condition"]]).corr()
corr = corr_data.iloc[1:, 0]
corr.sort_values()
# print("max correlation:", corr.max(), "min correlation:", corr.min())
```

Out[17]:

```
odometer              -0.269581
condition_fair        -0.192782
condition_good        -0.155089
size_mid-size         -0.130272
size_compact          -0.130203
                         ...
size_full-size         0.142521
transmission_other     0.168992
manufacturer_ram       0.170585
condition_like new     0.178035
year                   0.384505
Name: price_class, Length: 69, dtype: float64
```

ML baseline models

In [0]:

```
#We make the necessary imports
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
```

We have a lot of categorical data, which means we can't use a tokenizer, so we use get dummies to convert categorical variables into indicator variables.

We then scale our data and set our y and x values for the train_test_split, where we try to predict the price_class, with the features from our dataframe, but with the price_class dropped.

In [0]:

```
ml_df = pd.get_dummies(data[["year", "odometer", "fuel", "price_class", "transmission",
"cylinders", "drive", "type", "condition", "manufacturer"]])
#ml_df.info()
```

In [0]:

```
ml_df = ml_df.iloc[index_data["0"][:]]
```

In [0]:

```
scaler = StandardScaler()
```

In [0]:

```
y = ml_df["price_class"]
x = scaler.fit_transform(ml_df.drop(columns=["price_class"]))

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42
)
```

We try to use the Decision tree algorithm and the RandomForest algorithm and we get around 22 % accuracy. Seems like we either don't have enough data, or we don't have the right features to measure them.

In [26]:

```
tree = DecisionTreeClassifier()

scores = cross_val_score(tree, x, y, cv=5, scoring="accuracy")
print("Average accuracy score:", scores.mean())
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_split.py:657: Warning: Th
e least populated class in y has only 3 members, which is too few. The minimum number of
members in any class cannot be less than n_splits=5.
  % (min_groups, self.n_splits)), Warning)
```

```
Average accuracy score: 0.2096748913750738
```

In [27]:

```
rf = RandomForestClassifier()

scores = cross_val_score(rf, x, y, cv=5, scoring="accuracy")
print("Average accuracy score:", scores.mean())
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_split.py:657: Warning: Th
e least populated class in y has only 3 members, which is too few. The minimum number of
members in any class cannot be less than n_splits=5.
  % (min_groups, self.n_splits)), Warning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
```

```
Average accuracy score: 0.21862190163862216
```

Neural networks

In [28]:

```
# We make the necessary imports for Deep Learning
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalizati
on, concatenate
import cv2
from keras.applications import VGG16
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
```

```
#Load the VGG model
vgg_conv = VGG16(weights='imagenet', include_top=False, input_shape=(64, 64, 3))
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_b
ackend.py:66: The name tf.get_default_graph is deprecated. Please use tf.compat.v1.get_de
fault_graph instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_b
ackend.py:541: The name tf.placeholder is deprecated. Please use tf.compat.v1.placeholder
instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_b
ackend.py:4432: The name tf.random_uniform is deprecated. Please use tf.random.uniform in
stead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_b
ackend.py:4267: The name tf.nn.max_pool is deprecated. Please use tf.nn.max_pool2d instea
d.

Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/
v0.1/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58892288/58889256 [==============================] - 5s 0us/step
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_b
ackend.py:190: The name tf.get_default_session is deprecated. Please use tf.compat.v1.get
_default_session instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_b
ackend.py:197: The name tf.ConfigProto is deprecated. Please use tf.compat.v1.ConfigProto
instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_b
ackend.py:203: The name tf.Session is deprecated. Please use tf.compat.v1.Session instead
.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_b
ackend.py:207: The name tf.global_variables is deprecated. Please use tf.compat.v1.global
_variables instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_b
ackend.py:216: The name tf.is_variable_initialized is deprecated. Please use tf.compat.v1
.is_variable_initialized instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_b
ackend.py:223: The name tf.variables_initializer is deprecated. Please use tf.compat.v1.v
ariables_initializer instead.
```

Test training_set

We tried another test_set to make sure that both datasets runs well.

We make use of the ImageDataGenerator, and we normalize it by dividing by 255, since RGB runs from 0 to 255. We then get the data via the directory flow, we choose the size to be 64 by 64, to keep it in a managable size.

In [0]:

```
train_datagen = ImageDataGenerator(rescale = 1./255)
test_datagen = ImageDataGenerator(rescale = 1./255)
```

In [0]:

```
training_set = train_datagen.flow_from_directory('stanford_cars_dataset/car_data/train',
                                                 target_size = (64, 64),
                                                 batch_size = 32,
                                                 class_mode = "categorical")

test_set = test_datagen.flow_from_directory('stanford_cars_dataset/car_data/test',
                                            target_size = (64, 64),
                                            batch_size = 32,
```

```
                                          class_mode = 'categorical')
```

```
Found 8144 images belonging to 196 classes.
Found 8041 images belonging to 196 classes.
```

We find out that there are 196 classes in the dataset, so we make sure that our output has the same amount of nodes in the output layer, since we use the softmax activation function. This function distributes a 100 % out on the 196 classes, and the one with the highest percentage, is the class deemed most likely, for the specific car at hand (based on the data).

In [0]:

```
# Create the model
test_model = models.Sequential()

for layer in vgg_conv.layers[:-2]:
  layer.trainable = False

  # Add the vgg convolutional base model
test_model.add(vgg_conv)

test_model.add(layers.Flatten())
test_model.add(layers.Dense(1024, activation='relu'))
test_model.add(layers.Dropout(0.2))
test_model.add(layers.Dense(512, activation='relu'))
test_model.add(layers.Dropout(0.2))
test_model.add(layers.Dense(196, activation='softmax'))
```

In [0]:

```
test_model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy
"])
test_model.fit_generator(training_set, epochs=4, validation_data= test_set)
```

```
Epoch 1/4
255/255 [==============================] - 38s 151ms/step - loss: 5.2801 - acc: 0.0070 -
val_loss: 5.2769 - val_acc: 0.0085
Epoch 2/4
255/255 [==============================] - 26s 102ms/step - loss: 5.2782 - acc: 0.0085 -
val_loss: 5.2760 - val_acc: 0.0085
Epoch 3/4
255/255 [==============================] - 26s 102ms/step - loss: 5.2770 - acc: 0.0083 -
val_loss: 5.2750 - val_acc: 0.0085
Epoch 4/4
255/255 [==============================] - 26s 102ms/step - loss: 5.2762 - acc: 0.0085 -
val_loss: 5.2750 - val_acc: 0.0085
```

Out[0]:

```
<keras.callbacks.History at 0x7f3b1dca0438>
```

Building the neural networks

Datapreperation

Here we index our data, and is a very important part for later to make a neural network that combine textual data and image data. We index our data, so we can feed the same row of data to two different neural networks. The combined network will be useless if we feed price_classes, type, odometer and so on in the ANN to a different car, than the image we are feeding the CNN.

In [0]:

```
data = data.reset_index(drop=True)
index_list = index_data["0"].to_list()
index_list.sort()
df = data.iloc[index_list[:]]
```

In [0]:

```
df = df.reset_index(drop=True)
```

In [0]:

```
images = []

index_list.sort()

for i in index_list:
  img = cv2.imread("sorted_car_images/{0}.jpg".format(i))
  img = cv2.resize(img, (64, 64))
  images.append(img / 255)

images = np.array(images)
```

In [0]:

```
split_df = pd.get_dummies(df[["price_class_label", "year", "odometer", "fuel", "transmis
sion", "cylinders", "drive", "type", "condition", "manufacturer"]])
```

We split the data and make sure that our Y, consist of our 80 price classes, and we make sure to drop our price classes in our X.

In [0]:

```
split = train_test_split(split_df, images, test_size=0.25, random_state=42)
(trainAttrX, testAttrX, trainImagesX, testImagesX) = split

max_price_class = 80

trainY = trainAttrX[trainAttrX.columns[2:83]]
testY = testAttrX[testAttrX.columns[2:83]]

# trainY = trainAttrX["price_class"] / max_price_class
# testY = testAttrX["price_class"] / max_price_class

trainAttrX = trainAttrX.drop(columns= trainAttrX.columns[2:83])
testAttrX = testAttrX.drop(columns= testAttrX.columns[2:83])
```

Model creation functions

In this section we define functions, that can run different setups of layers, which we choose, so we easilier can test them out and later use hyperparameter tuning easily, merely by taking in the parameters.

The first function we define is a regular artifical neural network (ANN), which we use for our textual and more categorical data. We use regular dense layers mixed with some dropouts and some normalization of the data.

In [0]:

```
def create_ANN(setup = 1):
  text_model = Sequential()

  if setup == 1:
    text_model.add(Dense(128, input_shape = (81,), activation="relu"))
    text_model.add(BatchNormalization())
    text_model.add(Dropout(0.2))
    text_model.add(Dense(64, activation="relu"))
    text_model.add(Dropout(0.2))
    text_model.add(BatchNormalization())
    text_model.add(Dense(32, activation="relu"))
  elif setup == 2:
    text_model.add(Dense(256, input_shape = (81,), activation="relu"))
    text_model.add(BatchNormalization())
    text_model.add(Dropout(0.2))
    text_model.add(Dense(128, activation="relu"))
```

```
    text_model.add(Dropout(0.2))
    text_model.add(Dense(64, activation="relu"))
    text_model.add(Dropout(0.2))
    text_model.add(BatchNormalization())
    text_model.add(Dense(32, activation="relu"))

  text_model.add(Dense(81, activation="softmax"))
  return text_model
```

Here we define the function for our VGG16 network, where we instanciate the model sequentially. We make sure to freeze the last two layers, so we can benefit from the training that has been done in the VGG16, and add our own layers on top of that. We create three different setups with different combinations of nodes, using dense layers. VGG16 consist of a lot of layers with Convolutional layers and Maxpooling layers in 2D, and therefore we need to flatten them out to vectors, to fit them in dense layers.

In [0]:

```
def create_VGG_network(setup = 1):

  # Create the model
  image_model = Sequential()

  #We freeze the last two layers
  for layer in vgg_conv.layers[:-2]:
    layer.trainable = False

  # Add the vgg convolutional base model
  image_model.add(vgg_conv)

  if setup == 1:
    image_model.add(Flatten())
    image_model.add(Dense(1024, activation='relu'))
    image_model.add(Dropout(0.2))

    image_model.add(Dense(512, activation='relu'))
    image_model.add(Dropout(0.2))
  elif setup == 2:
    image_model.add(Flatten())
    image_model.add(Dense(1024, activation='relu'))
    image_model.add(Dropout(0.2))
    image_model.add(BatchNormalization())

    image_model.add(Dense(512, activation='relu'))
    image_model.add(Dropout(0.2))
  elif setup == 3:
    image_model.add(Flatten())
    image_model.add(Dense(512, activation='relu'))
    image_model.add(Dropout(0.2))

    image_model.add(Dense(256, activation='relu'))
    image_model.add(Dropout(0.2))

  image_model.add(Dense(81, activation='softmax'))
  return image_model
```

The concept here is the same as the one above, except we don't have any pre-trained model, so no transfer-learning here. We create two setups of convolutional neural networks, which we flatten to fit our output layer, which is a dense layer.

In [0]:

```
def create_CNN(setup = 1):
    # Create the model
  image_model = Sequential()

  if setup == 1:
    image_model.add(Conv2D(32, (3, 3), input_shape=(64, 64, 3), activation = "relu"))
    image_model.add(MaxPooling2D(pool_size=(2,2)))
    image_model.add(Conv2D(64, (3, 3), padding="same", activation = "relu"))
```

```
    image_model.add(MaxPooling2D(pool_size=(2,2)))
    image_model.add(Conv2D(128, (3, 3), padding="same", activation = "relu"))
    image_model.add(MaxPooling2D(pool_size=(2,2)))
  elif setup == 2:
    image_model.add(Conv2D(128, (3, 3), input_shape=(64, 64, 3), activation = "relu"))
    image_model.add(MaxPooling2D(pool_size=(2,2)))
    image_model.add(Conv2D(64, (3, 3), padding="same", activation = "relu"))
    image_model.add(MaxPooling2D(pool_size=(2,2)))
    image_model.add(Conv2D(32, (3, 3), padding="same", activation = "relu"))
    image_model.add(MaxPooling2D(pool_size=(2,2)))

  image_model.add(Flatten())
  image_model.add(Dense(81, activation='softmax'))
  return image_model
```

After we have created the three different functions to run our different networks, ANN, CNN and VGG16, respectively, we try to test them out to see if they work.

We use a regular adam optimizer, and use a categorical_crossentropy loss function. We fit the model and add the values to history, so we can plot the accuracy and the validation accuracy, and the loss and validation loss of the outcome.

We do this for all three functions and hope that all of them works (they do work).

In [0]:

```
image_model = create_VGG_network()
image_model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accurac
y"])
history = image_model.fit(trainImagesX,trainY, batch_size=32, epochs= 3, validation_data
= (testImagesX, testY))
```

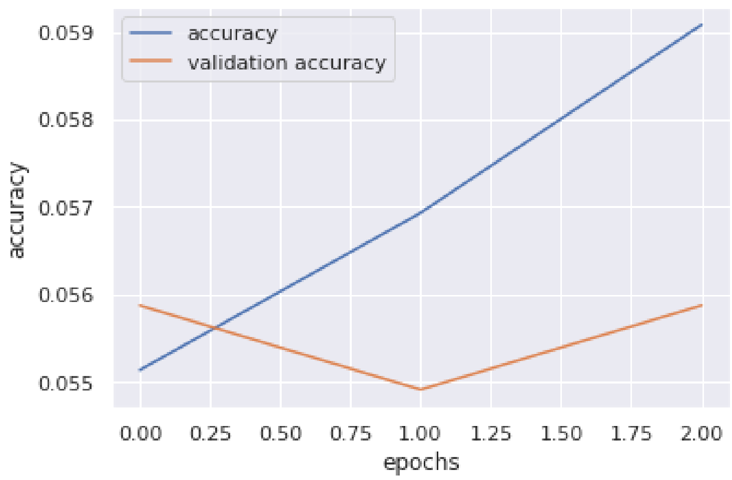
```
Train on 31194 samples, validate on 10398 samples
Epoch 1/3
31194/31194 [==============================] - 36s 1ms/step - loss: 3.6984 - acc: 0.0551
- val_loss: 3.6660 - val_acc: 0.0559
Epoch 2/3
31194/31194 [==============================] - 23s 744us/step - loss: 3.6625 - acc: 0.056
9 - val_loss: 3.6621 - val_acc: 0.0549
Epoch 3/3
31194/31194 [==============================] - 23s 745us/step - loss: 3.6601 - acc: 0.059
1 - val_loss: 3.6633 - val_acc: 0.0559
```

In [0]:

```
plt.plot(history.history["acc"], label="accuracy")
plt.plot(history.history["val_acc"], label="validation accuracy")
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.legend()
```

Out[0]:

```
<matplotlib.legend.Legend at 0x7f3b1ca712b0>
```

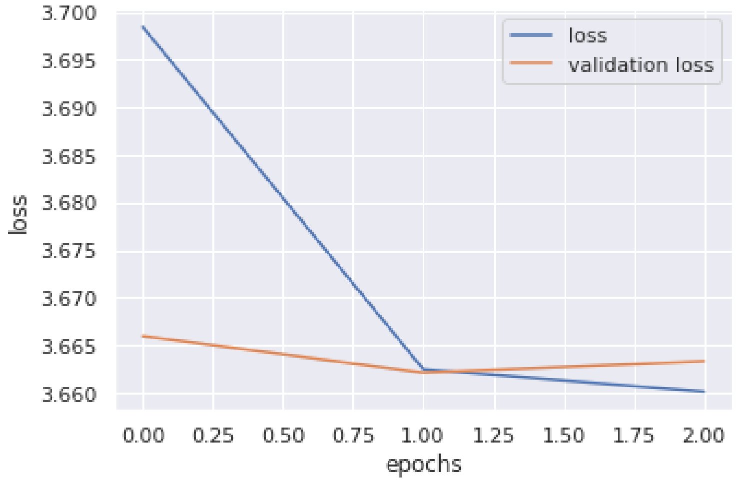


In [0]:

```
plt.plot(history.history["loss"], label="loss")
plt.plot(history.history["val_loss"], label="validation loss")
plt.xlabel("epochs")
plt.ylabel("loss")
plt.legend()
```

Out[0]:

```
<matplotlib.legend.Legend at 0x7f3b1ca4bef0>
```



In [0]:

```
text_model = create_ANN()
text_model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy
"])
history = text_model.fit(trainAttrX, trainY, batch_size=32, epochs=3, validation_data= (
testAttrX, testY))
```

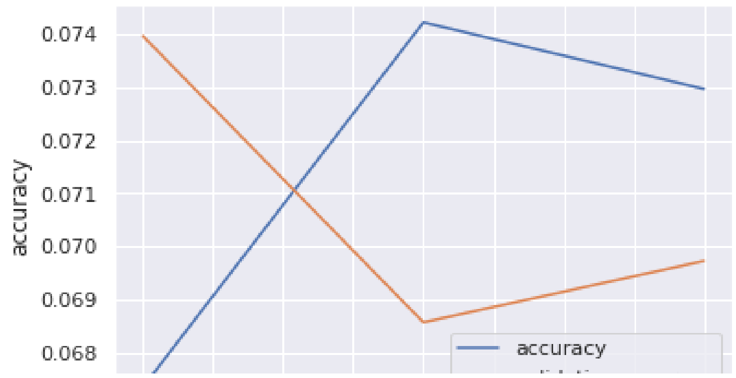
```
Train on 31194 samples, validate on 10398 samples
Epoch 1/3
31194/31194 [==============================] - 25s 802us/step - loss: 3.5915 - acc: 0.067
4 - val_loss: 3.5025 - val_acc: 0.0740
Epoch 2/3
31194/31194 [==============================] - 12s 381us/step - loss: 3.4854 - acc: 0.074
2 - val_loss: 3.5179 - val_acc: 0.0686
Epoch 3/3
31194/31194 [==============================] - 12s 381us/step - loss: 3.4734 - acc: 0.073
0 - val_loss: 3.4888 - val_acc: 0.0697
```

In [0]:

```
plt.plot(history.history["acc"], label="accuracy")
plt.plot(history.history["val_acc"], label="validation accuracy")
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.legend()
```

Out[0]:

```
<matplotlib.legend.Legend at 0x7f3b1c63a080>
```

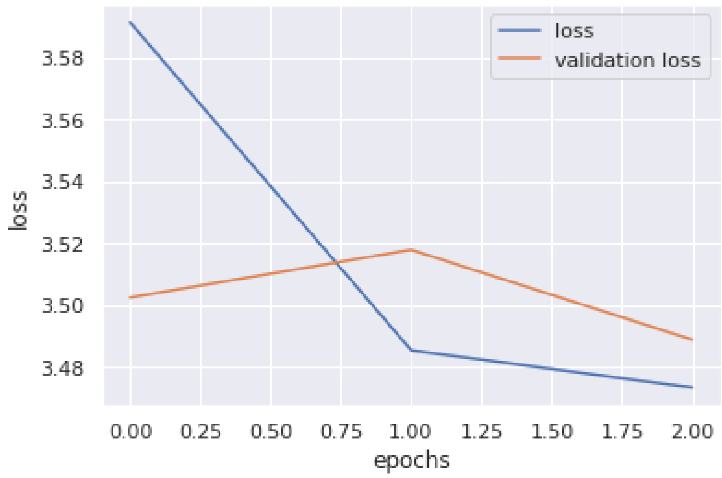


In [0]:

```
plt.plot(history.history["loss"], label="loss")
plt.plot(history.history["val_loss"], label="validation loss")
plt.xlabel("epochs")
plt.ylabel("loss")
plt.legend()
```

Out[0]:

```
<matplotlib.legend.Legend at 0x7f3b1c5d1fd0>
```



Combined model

We made three different functions before, but the intention was to combine both textual data and image data, so we try to make a single network that can take both kinds of data. We simply create a function that concatinates both our textual model and our image model. We make sure that we can choose which image model we want, when we try to run, since we both have our own CNN and the VGG16. Lastly we compile the model in the function as well.

We fit the model and gets no errors, which means that we have created a network that can both take in textual data and images.

The results aren't the best, and that could be because of a lot of factors, such as, the fact that we could let it train for longer, or that the dataset could be much better, since this was a webscrapping from craigslist (not done by us) a lot of pictures might be unusable, since many pictures consist of bad pictures of the cars, or has phonenumbers on them and so on...

In [0]:

```
def create_combined_model(use_vgg = False, ANN_setup = 1, CNN_setup = 1, VGG_setup = 1,
learning_rate = 0.1):
  text_model = create_ANN(ANN_setup)

  if use_vgg:
    image_model = create_VGG_network(VGG_setup)
  else:
    image_model = create_CNN(CNN_setup)

  combinedInput = concatenate([text_model.output, image_model.output])

  x = Dense(128, activation="relu")(combinedInput)

  x = Dense(81, activation="softmax")(x)

  model = Model(inputs=[text_model.input, image_model.input], outputs=x)
```

```
    opt = Adam(lr=learning_rate)
    model.compile(optimizer=opt, loss="categorical_crossentropy", metrics=["accuracy"])
    return model
```

Beneath we test a couple of setups, to make sure it works with both the VGG network and the CNN, hence the parameter in the create_combined_model, which states true or false.

In [60]:

```
model = create_combined_model(False, 1, 2, 1)
monitor_val_acc = EarlyStopping(monitor="val_acc", patience=4)
history = model.fit([trainAttrX, trainImagesX], trainY, batch_size=32, epochs= 30, valid
ation_data=([testAttrX, testImagesX], testY), callbacks=[monitor_val_acc])
```

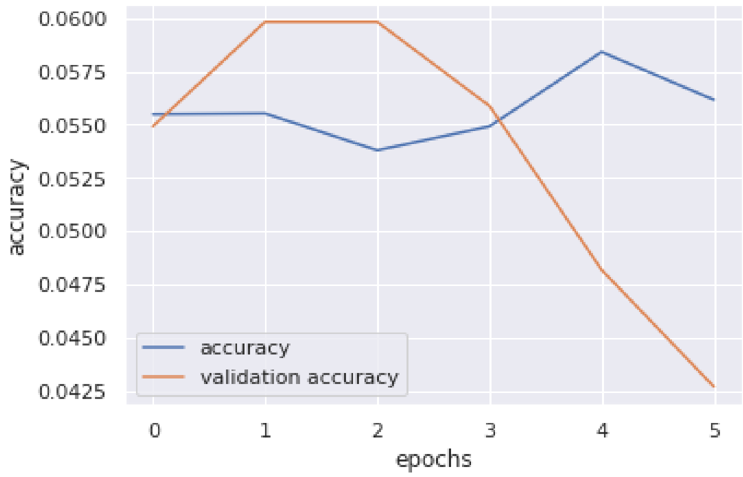
```
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 15s 490us/step - loss: 3.6797 - acc: 0.055
5 - val_loss: 3.6778 - val_acc: 0.0549
Epoch 2/30
31194/31194 [==============================] - 14s 442us/step - loss: 3.6811 - acc: 0.055
5 - val_loss: 3.6770 - val_acc: 0.0598
Epoch 3/30
31194/31194 [==============================] - 14s 455us/step - loss: 3.6800 - acc: 0.053
8 - val_loss: 3.6831 - val_acc: 0.0598
Epoch 4/30
31194/31194 [==============================] - 14s 441us/step - loss: 3.6808 - acc: 0.054
9 - val_loss: 3.6993 - val_acc: 0.0559
Epoch 5/30
31194/31194 [==============================] - 14s 443us/step - loss: 3.6800 - acc: 0.058
4 - val_loss: 3.6821 - val_acc: 0.0482
Epoch 6/30
31194/31194 [==============================] - 14s 438us/step - loss: 3.6821 - acc: 0.056
2 - val_loss: 3.6832 - val_acc: 0.0427
```

In [61]:

```
plt.plot(history.history["acc"], label="accuracy")
plt.plot(history.history["val_acc"], label="validation accuracy")
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.legend()
```

Out[61]:

```
<matplotlib.legend.Legend at 0x7f8ac65e5828>
```

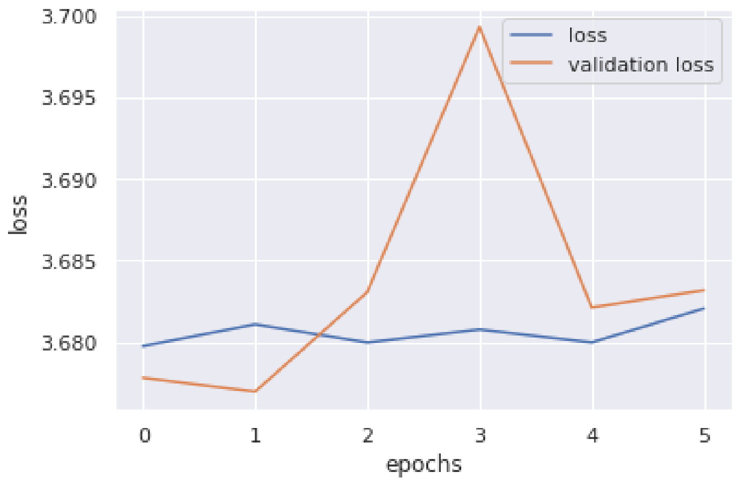


In [62]:

```
plt.plot(history.history["loss"], label="loss")
plt.plot(history.history["val_loss"], label="validation loss")
plt.xlabel("epochs")
plt.ylabel("loss")
plt.legend()
```

Out[62]:

<matplotlib.legend.Legend at 0x7f8ac65b57f0>



Hyperparameter tuning

The tuning of the network is hindered by the lack of sklearn support for the Keras functional API. We have therefor written our own simple brute force function for testing the hyper parameters.

The function simply loops through all the different parameter configuration and tests them all. The history and accuracy of each network configuration is saved and returned.

In [0]:

```
def hyperparameter_tune(params, stop_patience = 4, epochs_to_run = 5):
  histories = []
  accuracies = []

  for i in range(len(params["use_vgg"])):
    for k in range(len(params["ANN_setup"])):
      for j in range(len(params["CNN_setup"])):
        for z in range(len(params["VGG_setup"])):
          model = create_combined_model(use_vgg=params["use_vgg"][i], ANN_setup=params["
ANN_setup"][k], CNN_setup=params["CNN_setup"][j], VGG_setup=params["VGG_setup"][z])
          monitor_val_acc = EarlyStopping(monitor="val_acc", patience=stop_patience)
          histories.append(model.fit([trainAttrX, trainImagesX], trainY, batch_size=32,
epochs=epochs_to_run, validation_data=([testAttrX, testImagesX], testY), callbacks=[moni
tor_val_acc]).history)
          accuracies.append([model.evaluate([testAttrX, testImagesX], testY)[1], {"use_v
gg": params["use_vgg"][i], "ANN_setup": params["ANN_setup"][k], "CNN_setup": params["CNN
_setup"][j], "VGG_setup": params["VGG_setup"][z]}])

  return histories, accuracies
```

In [69]:

```
parameters = {"use_vgg": [False, True], "ANN_setup" : [1, 2], "CNN_setup": [1, 2], "VGG_
setup": [1, 2, 3]}

histories, accuracies = hyperparameter_tune(params = parameters, epochs_to_run=30)
```

```
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 17s 550us/step - loss: 3.6773 - acc: 0.052
5 - val_loss: 3.6830 - val_acc: 0.0427
Epoch 2/30
31194/31194 [==============================] - 12s 394us/step - loss: 3.6803 - acc: 0.054
6 - val_loss: 3.6788 - val_acc: 0.0559
Epoch 3/30
31194/31194 [==============================] - 12s 397us/step - loss: 3.6794 - acc: 0.055
5 - val_loss: 3.6809 - val_acc: 0.0598
Epoch 4/30
31194/31194 [==============================] - 12s 398us/step - loss: 3.6801 - acc: 0.055
0 - val_loss: 3.6775 - val_acc: 0.0598
```

```
Epoch 5/30
31194/31194 [==============================] - 13s 402us/step - loss: 3.6802 - acc: 0.055
9 - val_loss: 3.6856 - val_acc: 0.0598
Epoch 6/30
31194/31194 [==============================] - 12s 394us/step - loss: 3.6801 - acc: 0.056
5 - val_loss: 3.6865 - val_acc: 0.0549
Epoch 7/30
31194/31194 [==============================] - 12s 395us/step - loss: 3.6790 - acc: 0.053
7 - val_loss: 3.6828 - val_acc: 0.0559
10398/10398 [==============================] - 1s 132us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 17s 559us/step - loss: 3.6778 - acc: 0.057
9 - val_loss: 3.6751 - val_acc: 0.0597
Epoch 2/30
31194/31194 [==============================] - 12s 397us/step - loss: 3.6823 - acc: 0.058
6 - val_loss: 3.6832 - val_acc: 0.0549
Epoch 3/30
31194/31194 [==============================] - 12s 398us/step - loss: 3.6797 - acc: 0.056
1 - val_loss: 3.6846 - val_acc: 0.0559
Epoch 4/30
31194/31194 [==============================] - 12s 395us/step - loss: 3.6808 - acc: 0.055
4 - val_loss: 3.6770 - val_acc: 0.0597
Epoch 5/30
31194/31194 [==============================] - 12s 392us/step - loss: 3.6812 - acc: 0.053
4 - val_loss: 3.6876 - val_acc: 0.0549
10398/10398 [==============================] - 1s 134us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 18s 578us/step - loss: 3.6779 - acc: 0.058
0 - val_loss: 3.7043 - val_acc: 0.0598
Epoch 2/30
31194/31194 [==============================] - 13s 409us/step - loss: 3.6807 - acc: 0.056
8 - val_loss: 3.6832 - val_acc: 0.0598
Epoch 3/30
31194/31194 [==============================] - 12s 394us/step - loss: 3.6815 - acc: 0.055
1 - val_loss: 3.6824 - val_acc: 0.0559
Epoch 4/30
31194/31194 [==============================] - 12s 400us/step - loss: 3.6801 - acc: 0.057
1 - val_loss: 3.6819 - val_acc: 0.0597
Epoch 5/30
31194/31194 [==============================] - 12s 398us/step - loss: 3.6810 - acc: 0.056
8 - val_loss: 3.6919 - val_acc: 0.0549
10398/10398 [==============================] - 1s 135us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 21s 677us/step - loss: 3.6823 - acc: 0.054
2 - val_loss: 3.6754 - val_acc: 0.0598
Epoch 2/30
31194/31194 [==============================] - 15s 476us/step - loss: 3.6811 - acc: 0.054
2 - val_loss: 3.6848 - val_acc: 0.0549
Epoch 3/30
31194/31194 [==============================] - 15s 476us/step - loss: 3.6806 - acc: 0.057
9 - val_loss: 3.6878 - val_acc: 0.0549
Epoch 4/30
31194/31194 [==============================] - 15s 480us/step - loss: 3.6809 - acc: 0.054
9 - val_loss: 3.6833 - val_acc: 0.0559
Epoch 5/30
31194/31194 [==============================] - 15s 483us/step - loss: 3.6803 - acc: 0.057
2 - val_loss: 3.6801 - val_acc: 0.0598
10398/10398 [==============================] - 2s 157us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 24s 779us/step - loss: 3.6768 - acc: 0.056
9 - val_loss: 3.6869 - val_acc: 0.0549
Epoch 2/30
31194/31194 [==============================] - 18s 576us/step - loss: 3.6808 - acc: 0.054
9 - val_loss: 3.6821 - val_acc: 0.0559
Epoch 3/30
31194/31194 [==============================] - 18s 578us/step - loss: 3.6803 - acc: 0.054
2 - val_loss: 3.6793 - val_acc: 0.0597
Epoch 4/30
```

```
31194/31194 [==============================] - 18s 579us/step - loss: 3.6805 - acc: 0.055
7 - val_loss: 3.6892 - val_acc: 0.0598
Epoch 5/30
31194/31194 [==============================] - 18s 578us/step - loss: 3.6800 - acc: 0.056
8 - val_loss: 3.6737 - val_acc: 0.0598
Epoch 6/30
31194/31194 [==============================] - 18s 577us/step - loss: 3.6808 - acc: 0.055
5 - val_loss: 3.6922 - val_acc: 0.0598
Epoch 7/30
31194/31194 [==============================] - 18s 577us/step - loss: 3.6819 - acc: 0.055
8 - val_loss: 3.6898 - val_acc: 0.0598
Epoch 8/30
31194/31194 [==============================] - 18s 580us/step - loss: 3.6818 - acc: 0.053
7 - val_loss: 3.6904 - val_acc: 0.0313
10398/10398 [==============================] - 2s 165us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 22s 697us/step - loss: 3.6385 - acc: 0.059
7 - val_loss: 3.6906 - val_acc: 0.0599
Epoch 2/30
31194/31194 [==============================] - 15s 485us/step - loss: 3.6504 - acc: 0.058
0 - val_loss: 3.6488 - val_acc: 0.0598
Epoch 3/30
31194/31194 [==============================] - 15s 484us/step - loss: 3.6797 - acc: 0.057
0 - val_loss: 3.6882 - val_acc: 0.0559
Epoch 4/30
31194/31194 [==============================] - 15s 476us/step - loss: 3.6823 - acc: 0.054
1 - val_loss: 3.6771 - val_acc: 0.0469
Epoch 5/30
31194/31194 [==============================] - 15s 481us/step - loss: 3.6796 - acc: 0.056
5 - val_loss: 3.6803 - val_acc: 0.0414
10398/10398 [==============================] - 2s 163us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 21s 672us/step - loss: 3.6724 - acc: 0.056
6 - val_loss: 3.6896 - val_acc: 0.0597
Epoch 2/30
31194/31194 [==============================] - 17s 541us/step - loss: 3.6798 - acc: 0.056
5 - val_loss: 3.6774 - val_acc: 0.0597
Epoch 3/30
31194/31194 [==============================] - 17s 541us/step - loss: 3.6803 - acc: 0.054
0 - val_loss: 3.6935 - val_acc: 0.0598
Epoch 4/30
31194/31194 [==============================] - 17s 541us/step - loss: 3.6795 - acc: 0.057
3 - val_loss: 3.7013 - val_acc: 0.0427
Epoch 5/30
31194/31194 [==============================] - 17s 540us/step - loss: 3.6829 - acc: 0.055
8 - val_loss: 3.6873 - val_acc: 0.0598
Epoch 6/30
31194/31194 [==============================] - 17s 538us/step - loss: 3.6806 - acc: 0.055
8 - val_loss: 3.6834 - val_acc: 0.0427
Epoch 7/30
31194/31194 [==============================] - 17s 542us/step - loss: 3.6803 - acc: 0.055
9 - val_loss: 3.6819 - val_acc: 0.0598
10398/10398 [==============================] - 1s 142us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 21s 664us/step - loss: 3.6780 - acc: 0.053
2 - val_loss: 3.6962 - val_acc: 0.0597
Epoch 2/30
31194/31194 [==============================] - 13s 417us/step - loss: 3.6803 - acc: 0.055
9 - val_loss: 3.6794 - val_acc: 0.0599
Epoch 3/30
31194/31194 [==============================] - 13s 422us/step - loss: 3.6801 - acc: 0.055
4 - val_loss: 3.6965 - val_acc: 0.0597
Epoch 4/30
31194/31194 [==============================] - 13s 412us/step - loss: 3.6804 - acc: 0.056
9 - val_loss: 3.6753 - val_acc: 0.0599
Epoch 5/30
31194/31194 [==============================] - 13s 422us/step - loss: 3.6811 - acc: 0.055
4 - val_loss: 3.6843 - val_acc: 0.0598
Epoch 6/30
```

```
31194/31194 [==============================] - 14s 447us/step - loss: 3.6813 - acc: 0.055
8 - val_loss: 3.6889 - val_acc: 0.0561
10398/10398 [==============================] - 2s 156us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 23s 725us/step - loss: 3.6868 - acc: 0.055
1 - val_loss: 3.6768 - val_acc: 0.0549
Epoch 2/30
31194/31194 [==============================] - 15s 467us/step - loss: 3.6805 - acc: 0.056
8 - val_loss: 3.6880 - val_acc: 0.0597
Epoch 3/30
31194/31194 [==============================] - 15s 469us/step - loss: 3.6807 - acc: 0.055
1 - val_loss: 3.6901 - val_acc: 0.0549
Epoch 4/30
31194/31194 [==============================] - 14s 462us/step - loss: 3.6809 - acc: 0.055
3 - val_loss: 3.6904 - val_acc: 0.0559
Epoch 5/30
31194/31194 [==============================] - 14s 451us/step - loss: 3.6802 - acc: 0.055
5 - val_loss: 3.7008 - val_acc: 0.0559
Epoch 6/30
31194/31194 [==============================] - 14s 434us/step - loss: 3.6791 - acc: 0.056
7 - val_loss: 3.6813 - val_acc: 0.0559
10398/10398 [==============================] - 2s 147us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 23s 752us/step - loss: 3.6868 - acc: 0.055
3 - val_loss: 3.6801 - val_acc: 0.0598
Epoch 2/30
31194/31194 [==============================] - 15s 480us/step - loss: 3.6804 - acc: 0.056
5 - val_loss: 3.6833 - val_acc: 0.0559
Epoch 3/30
31194/31194 [==============================] - 15s 478us/step - loss: 3.6796 - acc: 0.055
2 - val_loss: 3.6838 - val_acc: 0.0559
Epoch 4/30
31194/31194 [==============================] - 15s 479us/step - loss: 3.6826 - acc: 0.054
7 - val_loss: 3.6798 - val_acc: 0.0598
Epoch 5/30
31194/31194 [==============================] - 15s 480us/step - loss: 3.6796 - acc: 0.056
1 - val_loss: 3.6938 - val_acc: 0.0482
10398/10398 [==============================] - 2s 169us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 25s 789us/step - loss: 3.6875 - acc: 0.055
2 - val_loss: 3.6943 - val_acc: 0.0559
Epoch 2/30
31194/31194 [==============================] - 16s 497us/step - loss: 3.6829 - acc: 0.054
6 - val_loss: 3.6907 - val_acc: 0.0598
Epoch 3/30
31194/31194 [==============================] - 16s 501us/step - loss: 3.6804 - acc: 0.053
2 - val_loss: 3.7010 - val_acc: 0.0598
Epoch 4/30
31194/31194 [==============================] - 15s 495us/step - loss: 3.6820 - acc: 0.056
5 - val_loss: 3.6811 - val_acc: 0.0598
Epoch 5/30
31194/31194 [==============================] - 15s 491us/step - loss: 3.6821 - acc: 0.055
9 - val_loss: 3.6937 - val_acc: 0.0559
Epoch 6/30
31194/31194 [==============================] - 15s 497us/step - loss: 3.6814 - acc: 0.055
5 - val_loss: 3.6900 - val_acc: 0.0469
10398/10398 [==============================] - 2s 174us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 25s 788us/step - loss: 3.6063 - acc: 0.059
9 - val_loss: 3.5703 - val_acc: 0.0661
Epoch 2/30
31194/31194 [==============================] - 15s 475us/step - loss: 3.6620 - acc: 0.056
4 - val_loss: 3.6930 - val_acc: 0.0452
Epoch 3/30
31194/31194 [==============================] - 15s 481us/step - loss: 3.6805 - acc: 0.056
4 - val_loss: 3.6795 - val_acc: 0.0452
Epoch 4/30
31194/31194 [==============================] - 15s 486us/step - loss: 3.6805 - acc: 0.055
```

```
7 - val_loss: 3.6876 - val_acc: 0.0598
Epoch 5/30
31194/31194 [==============================] - 15s 481us/step - loss: 3.6815 - acc: 0.056
0 - val_loss: 3.6843 - val_acc: 0.0559
10398/10398 [==============================] - 2s 173us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 33s 1ms/step - loss: 3.6852 - acc: 0.0587
- val_loss: 3.6687 - val_acc: 0.0598
Epoch 2/30
31194/31194 [==============================] - 23s 747us/step - loss: 3.6796 - acc: 0.055
5 - val_loss: 3.7051 - val_acc: 0.0559
Epoch 3/30
31194/31194 [==============================] - 23s 737us/step - loss: 3.6817 - acc: 0.056
5 - val_loss: 3.6844 - val_acc: 0.0482
Epoch 4/30
31194/31194 [==============================] - 23s 737us/step - loss: 3.6812 - acc: 0.054
1 - val_loss: 3.6975 - val_acc: 0.0549
Epoch 5/30
31194/31194 [==============================] - 23s 744us/step - loss: 3.6798 - acc: 0.056
2 - val_loss: 3.6854 - val_acc: 0.0469
10398/10398 [==============================] - 4s 388us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 35s 1ms/step - loss: 3.6705 - acc: 0.0551
- val_loss: 3.6889 - val_acc: 0.0597
Epoch 2/30
31194/31194 [==============================] - 25s 786us/step - loss: 3.6814 - acc: 0.056
2 - val_loss: 3.6979 - val_acc: 0.0559
Epoch 3/30
31194/31194 [==============================] - 25s 787us/step - loss: 3.6811 - acc: 0.055
7 - val_loss: 3.6852 - val_acc: 0.0559
Epoch 4/30
31194/31194 [==============================] - 24s 782us/step - loss: 3.6808 - acc: 0.055
7 - val_loss: 3.6870 - val_acc: 0.0481
Epoch 5/30
31194/31194 [==============================] - 24s 781us/step - loss: 3.6815 - acc: 0.055
3 - val_loss: 3.6860 - val_acc: 0.0559
10398/10398 [==============================] - 4s 397us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 34s 1ms/step - loss: 3.6843 - acc: 0.0558
- val_loss: 3.6840 - val_acc: 0.0549
Epoch 2/30
31194/31194 [==============================] - 23s 728us/step - loss: 3.6822 - acc: 0.056
6 - val_loss: 3.6909 - val_acc: 0.0559
Epoch 3/30
31194/31194 [==============================] - 23s 731us/step - loss: 3.6795 - acc: 0.056
9 - val_loss: 3.6782 - val_acc: 0.0597
Epoch 4/30
31194/31194 [==============================] - 23s 734us/step - loss: 3.6800 - acc: 0.054
0 - val_loss: 3.6779 - val_acc: 0.0598
Epoch 5/30
31194/31194 [==============================] - 23s 727us/step - loss: 3.6805 - acc: 0.056
6 - val_loss: 3.6954 - val_acc: 0.0597
Epoch 6/30
31194/31194 [==============================] - 23s 729us/step - loss: 3.6803 - acc: 0.057
3 - val_loss: 3.6836 - val_acc: 0.0549
Epoch 7/30
31194/31194 [==============================] - 23s 733us/step - loss: 3.6818 - acc: 0.055
7 - val_loss: 3.6889 - val_acc: 0.0559
Epoch 8/30
31194/31194 [==============================] - 23s 730us/step - loss: 3.6792 - acc: 0.057
5 - val_loss: 3.6873 - val_acc: 0.0598
10398/10398 [==============================] - 4s 391us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 35s 1ms/step - loss: 3.6852 - acc: 0.0543
- val_loss: 3.6866 - val_acc: 0.0469
Epoch 2/30
31194/31194 [==============================] - 23s 750us/step - loss: 3.6784 - acc: 0.055
8 - val_loss: 3.6817 - val_acc: 0.0598
```

```
Epoch 3/30
31194/31194 [==============================] - 23s 744us/step - loss: 3.6797 - acc: 0.055
5 - val_loss: 3.6874 - val_acc: 0.0597
Epoch 4/30
31194/31194 [==============================] - 23s 744us/step - loss: 3.6793 - acc: 0.054
5 - val_loss: 3.6911 - val_acc: 0.0549
Epoch 5/30
31194/31194 [==============================] - 23s 751us/step - loss: 3.6806 - acc: 0.056
5 - val_loss: 3.6911 - val_acc: 0.0598
Epoch 6/30
31194/31194 [==============================] - 23s 744us/step - loss: 3.6780 - acc: 0.056
2 - val_loss: 3.6766 - val_acc: 0.0598
10398/10398 [==============================] - 4s 391us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 36s 1ms/step - loss: 3.6713 - acc: 0.0540
- val_loss: 3.6897 - val_acc: 0.0599
Epoch 2/30
31194/31194 [==============================] - 24s 784us/step - loss: 3.6794 - acc: 0.057
3 - val_loss: 3.6874 - val_acc: 0.0560
Epoch 3/30
31194/31194 [==============================] - 24s 784us/step - loss: 3.6799 - acc: 0.056
1 - val_loss: 3.6886 - val_acc: 0.0559
Epoch 4/30
31194/31194 [==============================] - 24s 783us/step - loss: 3.6807 - acc: 0.056
5 - val_loss: 3.6880 - val_acc: 0.0559
Epoch 5/30
31194/31194 [==============================] - 24s 782us/step - loss: 3.6794 - acc: 0.054
4 - val_loss: 3.6837 - val_acc: 0.0597
10398/10398 [==============================] - 4s 401us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 35s 1ms/step - loss: 3.6798 - acc: 0.0562
- val_loss: 3.6733 - val_acc: 0.0559
Epoch 2/30
31194/31194 [==============================] - 23s 737us/step - loss: 3.6796 - acc: 0.055
5 - val_loss: 3.6833 - val_acc: 0.0598
Epoch 3/30
31194/31194 [==============================] - 23s 738us/step - loss: 3.6833 - acc: 0.054
0 - val_loss: 3.6943 - val_acc: 0.0597
Epoch 4/30
31194/31194 [==============================] - 23s 741us/step - loss: 3.6805 - acc: 0.057
1 - val_loss: 3.6910 - val_acc: 0.0452
Epoch 5/30
31194/31194 [==============================] - 23s 733us/step - loss: 3.6815 - acc: 0.054
6 - val_loss: 3.6944 - val_acc: 0.0414
Epoch 6/30
31194/31194 [==============================] - 23s 748us/step - loss: 3.6818 - acc: 0.054
0 - val_loss: 3.6833 - val_acc: 0.0598
10398/10398 [==============================] - 4s 401us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 36s 1ms/step - loss: 3.6788 - acc: 0.0559
- val_loss: 3.6887 - val_acc: 0.0482
Epoch 2/30
31194/31194 [==============================] - 23s 750us/step - loss: 3.6787 - acc: 0.053
8 - val_loss: 3.6927 - val_acc: 0.0559
Epoch 3/30
31194/31194 [==============================] - 24s 754us/step - loss: 3.6801 - acc: 0.055
9 - val_loss: 3.6858 - val_acc: 0.0559
Epoch 4/30
31194/31194 [==============================] - 23s 753us/step - loss: 3.6805 - acc: 0.056
6 - val_loss: 3.6865 - val_acc: 0.0559
Epoch 5/30
31194/31194 [==============================] - 23s 753us/step - loss: 3.6796 - acc: 0.056
1 - val_loss: 3.6935 - val_acc: 0.0549
Epoch 6/30
31194/31194 [==============================] - 23s 752us/step - loss: 3.6819 - acc: 0.054
1 - val_loss: 3.6995 - val_acc: 0.0559
10398/10398 [==============================] - 4s 404us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
```

```
31194/31194 [==============================] - 38s 1ms/step - loss: 3.6857 - acc: 0.0557
- val_loss: 3.6771 - val_acc: 0.0597
Epoch 2/30
31194/31194 [==============================] - 25s 802us/step - loss: 3.6804 - acc: 0.055
8 - val_loss: 3.6978 - val_acc: 0.0597
Epoch 3/30
31194/31194 [==============================] - 25s 801us/step - loss: 3.6807 - acc: 0.057
3 - val_loss: 3.6838 - val_acc: 0.0549
Epoch 4/30
31194/31194 [==============================] - 25s 799us/step - loss: 3.6809 - acc: 0.055
7 - val_loss: 3.6899 - val_acc: 0.0559
Epoch 5/30
31194/31194 [==============================] - 26s 819us/step - loss: 3.6811 - acc: 0.058
9 - val_loss: 3.6815 - val_acc: 0.0559
10398/10398 [==============================] - 4s 405us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 37s 1ms/step - loss: 3.6861 - acc: 0.0560
- val_loss: 3.6832 - val_acc: 0.0559
Epoch 2/30
31194/31194 [==============================] - 23s 752us/step - loss: 3.6798 - acc: 0.056
3 - val_loss: 3.6860 - val_acc: 0.0482
Epoch 3/30
31194/31194 [==============================] - 24s 754us/step - loss: 3.6814 - acc: 0.054
4 - val_loss: 3.6786 - val_acc: 0.0559
Epoch 4/30
31194/31194 [==============================] - 24s 758us/step - loss: 3.6800 - acc: 0.054
6 - val_loss: 3.6866 - val_acc: 0.0452
Epoch 5/30
31194/31194 [==============================] - 23s 753us/step - loss: 3.6804 - acc: 0.055
4 - val_loss: 3.6831 - val_acc: 0.0598
Epoch 6/30
31194/31194 [==============================] - 23s 753us/step - loss: 3.6823 - acc: 0.054
6 - val_loss: 3.6829 - val_acc: 0.0559
Epoch 7/30
31194/31194 [==============================] - 23s 753us/step - loss: 3.6811 - acc: 0.056
2 - val_loss: 3.6858 - val_acc: 0.0597
Epoch 8/30
31194/31194 [==============================] - 23s 753us/step - loss: 3.6808 - acc: 0.056
2 - val_loss: 3.6839 - val_acc: 0.0559
Epoch 9/30
31194/31194 [==============================] - 24s 756us/step - loss: 3.6805 - acc: 0.054
7 - val_loss: 3.6870 - val_acc: 0.0559
10398/10398 [==============================] - 4s 402us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 38s 1ms/step - loss: 3.6849 - acc: 0.0568
- val_loss: 3.6801 - val_acc: 0.0598
Epoch 2/30
31194/31194 [==============================] - 24s 762us/step - loss: 3.6800 - acc: 0.054
8 - val_loss: 3.6917 - val_acc: 0.0598
Epoch 3/30
31194/31194 [==============================] - 24s 770us/step - loss: 3.6792 - acc: 0.052
8 - val_loss: 3.6854 - val_acc: 0.0597
Epoch 4/30
31194/31194 [==============================] - 24s 763us/step - loss: 3.6812 - acc: 0.053
6 - val_loss: 3.6846 - val_acc: 0.0549
Epoch 5/30
31194/31194 [==============================] - 24s 760us/step - loss: 3.6812 - acc: 0.056
0 - val_loss: 3.6901 - val_acc: 0.0597
10398/10398 [==============================] - 4s 407us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 41s 1ms/step - loss: 3.6789 - acc: 0.0564
- val_loss: 3.6783 - val_acc: 0.0598
Epoch 2/30
31194/31194 [==============================] - 25s 812us/step - loss: 3.6803 - acc: 0.053
8 - val_loss: 3.6864 - val_acc: 0.0549
Epoch 3/30
31194/31194 [==============================] - 25s 814us/step - loss: 3.6806 - acc: 0.054
7 - val_loss: 3.6805 - val_acc: 0.0598
Epoch 4/30
```

```
31194/31194 [==============================] - 25s 810us/step - loss: 3.6818 - acc: 0.056
9 - val_loss: 3.6912 - val_acc: 0.0549
Epoch 5/30
31194/31194 [==============================] - 25s 817us/step - loss: 3.6803 - acc: 0.054
5 - val_loss: 3.6880 - val_acc: 0.0549
10398/10398 [==============================] - 4s 415us/step
Train on 31194 samples, validate on 10398 samples
Epoch 1/30
31194/31194 [==============================] - 39s 1ms/step - loss: 3.6791 - acc: 0.0573
- val_loss: 3.6782 - val_acc: 0.0559
Epoch 2/30
31194/31194 [==============================] - 24s 758us/step - loss: 3.6813 - acc: 0.055
0 - val_loss: 3.6857 - val_acc: 0.0598
Epoch 3/30
31194/31194 [==============================] - 24s 755us/step - loss: 3.6794 - acc: 0.057
7 - val_loss: 3.6987 - val_acc: 0.0452
Epoch 4/30
31194/31194 [==============================] - 24s 763us/step - loss: 3.6818 - acc: 0.054
1 - val_loss: 3.6880 - val_acc: 0.0559
Epoch 5/30
31194/31194 [==============================] - 24s 755us/step - loss: 3.6817 - acc: 0.058
0 - val_loss: 3.6763 - val_acc: 0.0549
Epoch 6/30
31194/31194 [==============================] - 24s 755us/step - loss: 3.6813 - acc: 0.056
6 - val_loss: 3.6863 - val_acc: 0.0598
10398/10398 [==============================] - 4s 401us/step
```

The last bits just illustrates how we would access the results from the hyperparameter tuning. The setup with the highest accuracy, along with it's history can easily be extracted.

In [71]:

```
highest_accuracy_index = 0
for i in range(1, len(accuracies)):
  if accuracies[i][0] > accuracies[highest_accuracy_index][0]:
    highest_accuracy_index = i

accuracies[highest_accuracy_index]
```

Out[71]:

```
[0.0598191960042464,
 {'ANN_setup': 1, 'CNN_setup': 2, 'VGG_setup': 1, 'use_vgg': False}]
```

In [75]:

```
histories[highest_accuracy_index]["acc"]
```

Out[75]:

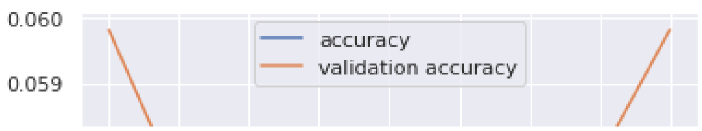
```
[0.05420914278434639,
 0.054209142785062926,
 0.05786369173559018,
 0.05494646406360197,
 0.057190485349746745]
```

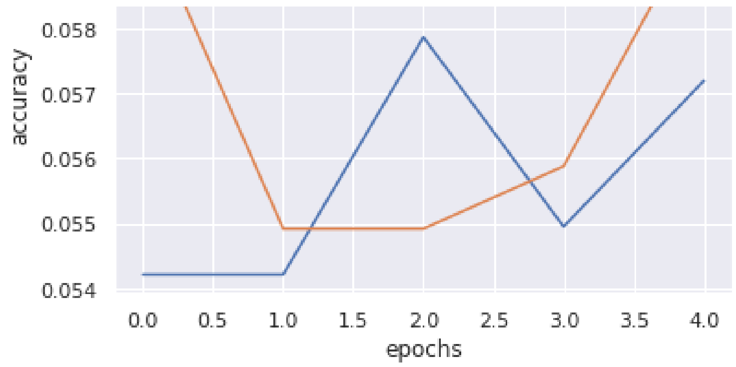
In [76]:

```
plt.plot(histories[highest_accuracy_index]["acc"], label="accuracy")
plt.plot(histories[highest_accuracy_index]["val_acc"], label="validation accuracy")
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.legend()
```

Out[76]:

```
<matplotlib.legend.Legend at 0x7f8710ce4208>
```





In [77]:

```
plt.plot(histories[highest_accuracy_index]["loss"], label="loss")
plt.plot(histories[highest_accuracy_index]["val_loss"], label="validation loss")
plt.xlabel("epochs")
plt.ylabel("loss")
plt.legend()
```

Out[77]:

```
<matplotlib.legend.Legend at 0x7f8710cbcc18>
```