Module 3: Convolutional Neural Network

To write legible answers you will need to be familiar with both Markdown and Latex

Before you turn this problem in, make sure everything runs as expected. First, restart the kernel (in the menubar, select Kernel→→Restart) and then run all cells (in the menubar, select Cell→→Run All).

Make sure you fill in any place that says "YOUR CODE HERE" or "YOUR ANSWER HERE", as well as your name below:

In [ ]:
```
NAME = ""
STUDENT_ID = ""
```

Image Classification

Building a Convolutional Neural Network [CNN]

In this assignment, we will build a classifier model that is able to distinguish between 10 different classes of images - airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. We will follow these steps:

1. Explore the example data
2. Build a small convnet to solve our classification problem
3. Evaluate training and validation accuracy

Data Exploration and Preparation

We'll start by downloading the CIFAR-10 dataset from Keras.

This is a link to the dataset documentation: https://keras.io/datasets/#cifar10-small-image-classification

And a link to the dataset source: https://www.cs.toronto.edu/~kriz/cifar.html

Be sure to set your Runtime environment to include a GPU, as it will speed up the training considerably (this time that's important!).

Loading data into local variables

In [ ]:
```
from keras.datasets import cifar10

# Fetch the data:
(X, y), (_, _) = cifar10.load_data()
```

Import needed functions and libraries

In [ ]:
```
# Ignore the warnings - Otherwise, TensorFlow tends to innundate one with far too m
import warnings
warnings.filterwarnings('always')
warnings.filterwarnings('ignore')

# For matrix operations and dataframes.
import numpy as np

# Data visualizaton.
import matplotlib.pyplot as plt
from matplotlib import style
import seaborn as sns
import random as rn

# Configure some defaults.
%matplotlib inline
style.use('fivethirtyeight')
sns.set(style='whitegrid',color_codes=True)

# Useful deep learning functions.
from tensorflow.keras import backend as K
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam, SGD, Adagrad, Adadelta, RMSprop
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.layers import Dropout, Flatten, Activation
from tensorflow.keras.layers import Conv2D, MaxPooling2D

# Powerful deep learning module.
import tensorflow as tf

# For dealing with data.
import numpy as np
```

Data Preparation & Exploration

Let's take a look at a few of these images. Rerun this cell multiple times to see different images for each class.

You may notice that these images look low fidelity, which is because they are! As we increase our image size, we also increase our model complexity. What's important is that our classes are still distinguishable from each other.

In [ ]:
```
fig, ax = plt.subplots(2, 5)
fig.set_size_inches(10, 6)

for i in range(2):
    for j in range(5):
        c = j + 5*i # Class counter
        l = np.random.choice(np.where(y == c)[0], 1)[0] # Get a random image from c
        ax[i, j].imshow(X[l])
        ax[i, j].set_title('Class: ' + str(y[l]))
        # Hide grid lines
        ax[i, j].grid(False)
        # Hide axes ticks
        ax[i, j].set_xticks([])
        ax[i, j].set_yticks([])

plt.tight_layout()
```

Let's take a look at the format of our data

In [ ]:
```
print('X (images)', X.shape)
print('y (classes)', y.shape)
```

We can see that we have 50,000 samples, where each images is 32 by 32 pixels with 3 color channels: RGB.

For each of these images, we have a single label for which class they each belong to.

One hot encode the labels, and normalize the data

We want to one hot encode our class labels. We also want to normalize our image data similarly to what we did in Assignment 3.

In [ ]:
```
# One-hot encode those integer values of class labels
y = to_categorical(y, 10)

# Normalize all entries to the interval [0, 1]
X = X / 255.
```

Problem 1 (a)

Create your own deep learning architecture, and train it on the dataset above.

One suggestion is to add several convolution layers each followed by a maxpooling layer. Towards the end you can add one or more fully connected layers. Dropout layers are often useful after each fully connected layer for overfitting, and you can try experimenting with that parameter. Your model should be able to reach **70% validation accuracy**.

You are responsible for your model architecture, hyperparameters, and optimizer.

HOWEVER, you are limited to a maximum of 50 epochs and 500,000 model parameters! You will lose points for exceeding these limits.

In [ ]:
```
# This is where we define the architecture of our deep neural network.
model = Sequential()

### YOUR CODE HERE ###

# A dense layer with 10 neurons (one per class).
model.add(Dense(10, activation = "softmax"))
```

In [ ]:
```
# A batch is the size of each training chunk. We're implementing batch gradient des
# stochastic gradient descent and full gradient descent.
batchsize = ### YOUR CODE HERE ###

# Each epoch goes through the entire training set once
epochs = ### YOUR CODE HERE ### MAXIMUM OF 50!
```

In [ ]:
```
opt = ### YOUR CODE HERE ###

model.compile(optimizer = opt,
              loss = 'categorical_crossentropy',
              metrics = ['accuracy'])
```

In [ ]:
```
model.summary()
# MAXIMUM OF 500,000 PARAMETERS!
```

In [ ]:
```
history = model.fit(X,
                    y,
                    batch_size = batchsize,
                    epochs = epochs,
                    validation_split = 0.2, # DON'T CHANGE validation_split!
                    verbose = 1)
```

Problem 1 (b)

Create training and validation loss and accuracy plots as above.

In [ ]:
```
### YOUR CODE HERE ###
```

Transfer Learning

There's a huge shortcut possible in training neural networks for recognition tasks, called transfer learning. The idea is to start with a fully-trained image recognition neural network, off-the-shelf with trained weights. We can repurpose the trained network for your particular recognition task, making use of the days of training that were needed to find those weights. What was learned by the neural network in its early layers are useful features in recognizing various things in images. Keras even has pretrained models built in for this purpose.

Keras Pretrained Models

```
Xception
VGG16
VGG19
ResNet, ResNetV2, ResNeXt
InceptionV3
InceptionResNetV2
MobileNet
MobileNetV2
DenseNet
NASNet
```

Usually one uses the layers of the pretrained model up to some point, and then creates some fully-connected layers to learn the desired recognition task. The earlier layers are "frozen", and only the later layers need to be trained. We'll use VGG16, which was trained to recognize 1000 objects in ImageNet. What we're doing here for our CIFAR-10 classifier may be akin to killing a fly with a shotgun, but the same process can be used to recognize objects the original network couldn't (i.e., you could use this technique to train your computer to recognize family and friends).

In [ ]:
```
# Some stuff we'll need...
from tensorflow.keras.layers import Input
from tensorflow.keras.applications.vgg16 import VGG16, preprocess_input
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import Model
```

Creating this pretrained network is a one line command. Notice we specified that the "top" should not be included. We aren't classifying 1000 different categories like ImageNet, so we don't include that layer. We'll add our own layer more suited to the task at hand.

We choose 224 as our image dimension because the pretrained VGG16 was trained using the ImageNet dataset which has images of this dimension.

In [ ]:
```
# Import the VGG16 trained neural network model, minus it's last (top) neuron layer
base_model = VGG16(weights = 'imagenet',
                   include_top = False,
                   input_shape = (32, 32, 3),
                   pooling = None)
```

Let's take a look at this pretrained model:

In [ ]:
```
base_model.summary()
```

Please do realize, this may be overkill for our toy recognition task. One could use this network with some layers (as we're about to add) to recognize 100 dog breeds or to recognize all your friends. If you wanted to recognize 100 dog breeds, you would use a final 100 neuron softmax for the final layer. We'll need a final softmax layer as before. First let's freeze all these pretrained weights. They are fine as they are.

In [ ]:
```
# This freezes the weights of our VGG16 pretrained model.
for layer in base_model.layers:
    layer.trainable = False
```

Now we'll add a flatten layer, a trainable dense layer, and a final softmax layer. This illustrates another way to create networks besides the sequential method we used for our example model. This is the Keras functional approach to building networks. It's more flexible and more powerful than the sequential method. For example, it allows you to implement transfer learning.

In [ ]:
```
# Now add layers to our pre-trained base model and add classification layers on top
x = base_model.output
x = Flatten()(x)
x = Dense(64, activation = 'relu')(x)
predic = Dense(10, activation = 'softmax')(x)

# And now put this all together to create our new model.
model = Model(inputs = base_model.input, outputs = predic)
model.summary()
```

Initialize Training Parameters

In [ ]:
```
# Compile the model.
model.compile(loss = 'categorical_crossentropy',
              optimizer = RMSprop(lr = 0.001),
              metrics = ['acc'])
```

We can see here in the Keras docs:

https://keras.io/api/applications/vgg/#vgg16-function

that we are required to preprocess our image data in a specific way to use this pretrained model, so let's go ahead and do that.

In [ ]:
```
(X, y), (_, _) = cifar10.load_data()
X = preprocess_input(X)
y = to_categorical(y, 10)
```

In [ ]:
```
# Let's also reduce the number of training epochs.
epochs = 20
batchsize = 200



# Train the model
history = model.fit(X,
                    y,
                    batch_size = batchsize,
                    epochs = epochs,
                    validation_split = 0.2,
                    verbose = 1)
```

Now that's better, about 80% accuracy on the train set, but only about 62% accuracy on the validation set (though each time this is run, a different result is obtained, so your results may vary), and with fewer epochs and trainable parameters than a network from scratch (such as Question 1). And we didn't need to add much to the output of our pretrained VGG16 network.

Do notice that we are heavily overfitting though.

In [ ]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.legend(['train', 'test'])
plt.show()

plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend(['train', 'test'])
plt.show()
```

Problem 2(a)

Add on your own last layers to the pretrained model and train it on the training data. You can increase (or decrease) the number of nodes per layer, increase (or decrease) the number of layers, and add dropout if your model is overfitting, change the hyperparameters, change your optimizer, etc. Try to get the validation accuracy higher than our example transfer learning model (shown above) was able to obtain, and try to minimize the amount of overfitting.

In [ ]:
```
base_model = VGG16(weights = 'imagenet',
                   include_top = False,
                   input_shape = (32, 32, 3),
                   pooling = None)

x = base_model.output
x = Flatten()(x)

### YOUR CODE HERE ###

predic = Dense(10, activation = 'softmax')(x)

# And now put this all together to create our new model.
model = Model(inputs = base_model.input, outputs = predic)
model.summary()
```

In [ ]:
```
# Compile the model.

### YOUR CODE HERE ###
```

In [ ]:
```
epochs = 20 ### Leave the epochs at 20 ###

batchsize = ### YOUR CODE HERE ###

# Train the model

### YOUR CODE HERE ###
```

Problem 2(b)

Create training and validation loss and accuracy plots for this model.

In [ ]:
```
### YOUR CODE HERE ###
```

Problem 2 (c)

- What did you add in your transfer learning model after the VGG16 portion?
- What did you try that didn't work?
- What optimizer and hyperparameters did you go with and why?
- How much did it improve over the example transfer learning model we provided?

YOUR ANSWER

Problem 2 (d)

Does your final model show signs of underfitting or overfitting? Explain why you think so.

YOUR ANSWER