Open in Colab

T81-558: Applications of Deep Neural Networks

Module 6: Convolutional Neural Networks (CNN) for Computer Vision

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 6 Material

- Part 6.1: Image Processing in Python [Video] [Notebook]
- **Part 6.2: Using Convolutional Neural Networks** [Video] [Notebook]
- Part 6.3: Using Pretrained Neural Networks with Keras [Video] [Notebook]
- Part 6.4: Looking at Keras Generators and Image Augmentation [Video] [Notebook]
- Part 6.5: Recognizing Multiple Images with YOLOv5 [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
try:
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False

# Nicely formatted time string
def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return f"{h}:{m:>02}:{s:>05.2f}"
```

```
Note: using Google CoLab
```

Part 6.2: Keras Neural Networks for Digits and Fashion MNIST

This module will focus on computer vision. There are some important differences and similarities with previous neural networks.

- We will usually use classification, though regression is still an option.
- The input to the neural network is now 3D (height, width, color)
- Data are not transformed; no z-scores or dummy variables.
- Processing time is much longer.
- We now have different layer times: dense layers (just like before), convolution layers, and max-pooling layers.
- Data will no longer arrive as CSV files. TensorFlow provides some utilities for going directly from the image to the input for a neural network.

Common Computer Vision Data Sets

There are many data sets for computer vision. Two of the most popular classic datasets are the MNIST digits data set and the CIFAR image data sets. We will not use either of these datasets in this course, but it is important to be familiar with them since neural network texts often refer to them.

The MNIST Digits Data Set is very popular in the neural network research community. You can see a sample of it in Figure 6.MNIST.

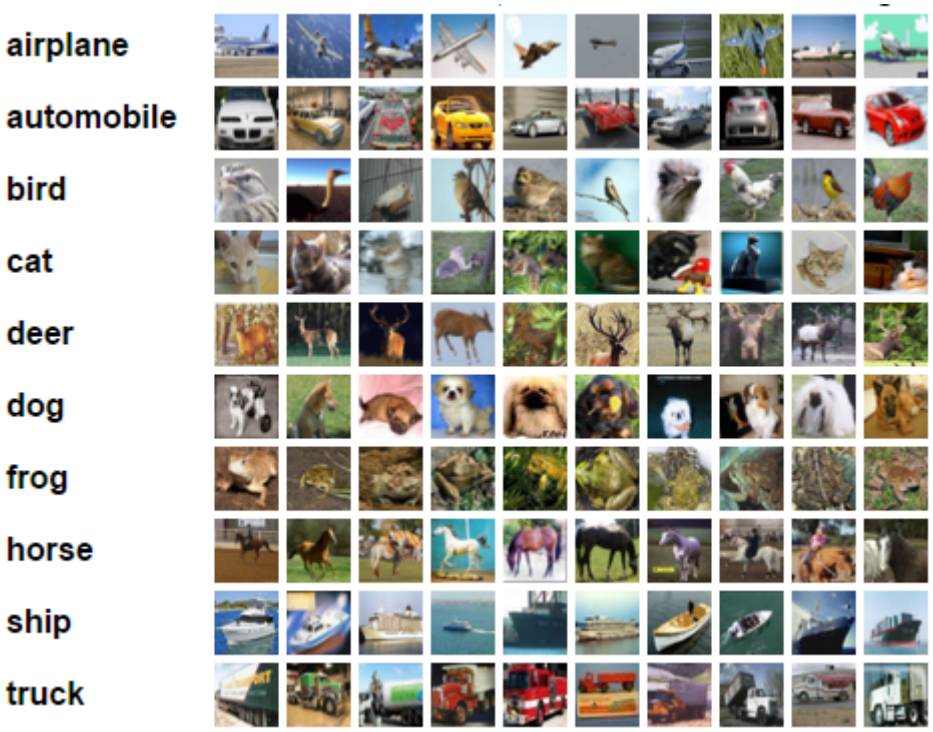
Figure 6.MNIST: MNIST Data Set

Fashion-MNIST is a dataset of Zalando 's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image associated with a label from 10 classes. Fashion-MNIST is a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits. You can see this data in Figure 6.MNIST-FASHION.

Figure 6.MNIST-FASHION: MNIST Fashon Data Set

The CIFAR-10 and CIFAR-100 datasets are also frequently used by the neural network research community.

Figure 6.CIFAR: CIFAR Data Set



The CIFAR-10 data set contains low-rez images that are divided into 10 classes. The CIFAR-100 data set contains 100 classes in a hierarchy.

Convolutional Neural Networks (CNNs)

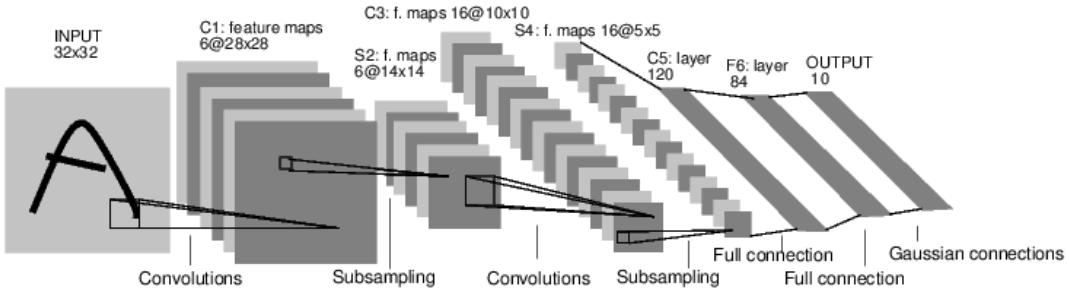
The convolutional neural network (CNN) is a neural network technology that has profoundly impacted the area of computer vision (CV). Fukushima (1980) [Cite:fukushima1980neocognitron] introduced the original concept of a convolutional neural network, and LeCun, Bottou, Bengio & Haffner (1998) [Cite:lecun1995convolutional] greatly improved this work. From this research, Yan LeCun introduced the famous LeNet-5 neural network architecture. This chapter follows the LeNet-5 style of convolutional neural network.
Although computer vision primarily uses CNNs, this technology has some applications outside of the field. You need to realize that if you want to utilize CNNs on non-visual data, you must find a way to encode your data to mimic the properties of visual data.

The order of the input array elements is crucial to the training. In contrast, most neural networks that are not CNNs treat their input data as a long vector of values, and the order in which you arrange the incoming features in this vector is irrelevant. You cannot change the order for these types of neural networks after you have trained the network.

The CNN network arranges the inputs into a grid. This arrangement worked well with images because the pixels in closer proximity to each other are important to each other. The order of pixels in an image is significant. The human body is a relevant example of this type of order. For the design of the face, we are accustomed to eyes being near to each other.

This advance in CNNs is due to years of research on biological eyes. In other words, CNNs utilize overlapping fields of input to simulate features of biological eyes. Until this breakthrough, AI had been unable to reproduce the capabilities of biological vision. Scale, rotation, and noise have presented challenges for AI computer vision research. You can observe the complexity of biological eyes in the example that follows. A friend raises a sheet of paper with a large number written on it. As your friend moves nearer to you, the number is still identifiable. In the same way, you can still identify the number when your friend rotates the paper. Lastly, your friend creates noise by drawing lines on the page, but you can still identify the number. As you can see, these examples demonstrate the high function of the biological eye and allow you to understand better the research breakthrough of CNNs. That is, this neural network can process scale, rotation, and noise in the field of computer vision. You can see this network structure in Figure 6.LENET.

Figure 6.LENET: A LeNET-5 Network (LeCun, 1998)



So far, we have only seen one layer type (dense layers). By the end of this book we will have seen:

- **Dense Layers** - Fully connected layers.
- **Convolution Layers** - Used to scan across images.
- **Max Pooling Layers** - Used to downsample images.
- **Dropout Layers** - Used to add regularization.
- **LSTM and Transformer Layers** - Used for time series data.

Convolution Layers

The first layer that we will examine is the convolutional layer. We will begin by looking at the hyper-parameters that you must specify for a convolutional layer in most neural network frameworks that support the CNN:

- Number of filters
- Filter Size
- Stride
- Padding
- Activation Function/Non-Linearity

The primary purpose of a convolutional layer is to detect features such as edges, lines, blobs of color, and other visual elements. The filters can detect these features. The more filters we give to a convolutional layer, the more features it can see.

A filter is a square-shaped object that scans over the image. A grid can represent the individual pixels of a grid. You can think of the convolutional layer as a smaller grid that sweeps left to right over each image row. There is also a hyperparameter that specifies both the width and height of the square-shaped filter. The following figure shows this configuration in which you see the six convolutional filters sweeping over the image grid:

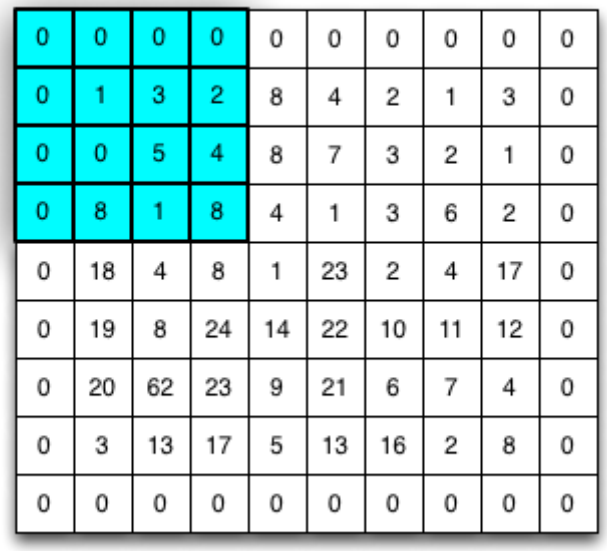
A convolutional layer has weights between it and the previous layer or image grid. Each pixel on each convolutional layer is a weight. Therefore, the number of weights between a convolutional layer and its predecessor layer or image field is the following:

```
[FilterSize] * [FilterSize] * [# of Filters]
```

For example, if the filter size were 5 (5x5) for 10 filters, there would be 250 weights.

You need to understand how the convolutional filters sweep across the previous layer's output or image grid. Figure 6.CNN illustrates the sweep:

Figure 6.CNN: Convolutional Neural Network



The above figure shows a convolutional filter with 4 and a padding size of 1. The padding size is responsible for the border of zeros in the area that the filter sweeps. Even though the image is 8x7, the extra padding provides a virtual image size of 9x8 for the filter to sweep across. The stride specifies the number of positions the convolutional filters will stop. The convolutional filters move to the right, advancing by the number of cells specified in the stride. Once you reach the far right, the convolutional filter moves back to the far left; then, it moves down by the stride amount and continues to the right again.

Some constraints exist concerning the size of the stride. The stride cannot be 0. The convolutional filter would never move if you set the stride. Furthermore, neither the stride nor the convolutional filter size can be larger than the previous grid. There

are additional constraints on the stride (*s*), padding (*p*), and the filter width (*f*) for an image of width (*w*). Specifically, the convolutional filter must be able to start at the far left or top border, move a certain number of strides, and land on the far right or bottom border. The following equation shows the number of steps a convolutional operator must take to cross the image:

$$steps = \frac{w - f + 2p}{s} + 1$$

The number of steps must be an integer. In other words, it cannot have decimal places. The purpose of the padding (*p*) is to be adjusted to make this equation become an integer value.

Max Pooling Layers

Max-pool layers downsample a 3D box to a new one with smaller dimensions. Typically, you can always place a max-pool layer immediately following the convolutional layer. The LENET shows the max-pool layer immediately after layers C1 and C3. These max-pool layers progressively decrease the size of the dimensions of the 3D boxes passing through them. This technique can avoid overfitting (Krizhevsky, Sutskever & Hinton, 2012).

A pooling layer has the following hyper-parameters:

- Spatial Extent (*f*)
- Stride (*s*)

Unlike convolutional layers, max-pool layers do not use padding. Additionally, max-pool layers have no weights, so training does not affect them. These layers downsample their 3D box input. The 3D box output by a max-pool layer will have a width equal to this equation:

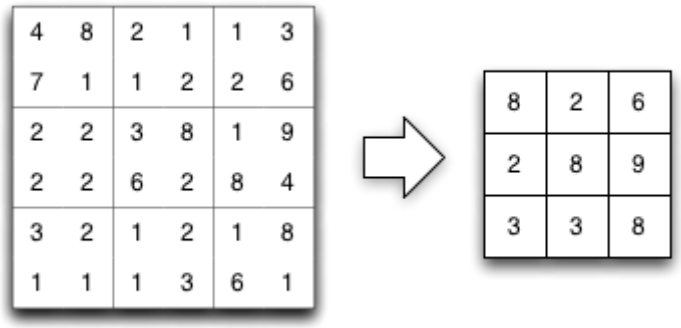
$$w_2 = \frac{w_1 - f}{s} + 1$$

The height of the 3D box produced by the max-pool layer is calculated similarly with this equation:

$$h_2 = \frac{h_1 - f}{s} + 1$$

The depth of the 3D box produced by the max-pool layer is equal to the depth the 3D box received as input. The most common setting for the hyper-parameters of a max-pool layer is f=2 and s=2. The spatial extent (f) specifies that boxes of 2x2 will be scaled down to single pixels. Of these four pixels, the pixel with the maximum value will represent the 2x2 pixel in the new grid. Because squares of size 4 are replaced with size 1, 75% of the pixel information is lost. The following figure shows this transformation as a 6x6 grid becomes a 3x3:

Figure 6.MAXPOOL: Max Pooling Layer



Of course, the above diagram shows each pixel as a single number. A grayscale image would have this characteristic. We usually take the average of the three numbers for an RGB image to determine which pixel has the maximum value.

Regression Convolutional Neural Networks

We will now look at two examples, one for regression and another for classification. For supervised computer vision, your dataset will need some labels. For classification, this label usually specifies what the image is a picture of. For regression, this "label" is some numeric quantity the image should produce, such as a count. We will look at two different means of providing this label.

The first example will show how to handle regression with convolution neural networks. We will provide an image and expect the neural network to count items in that image. We will use a dataset that I created that contains a random number of paperclips. The following code will download this dataset for you.

In [2]:
```
import os

URL = "https://github.com/jeffheaton/data-mirror/releases/"
DOWNLOAD_SOURCE = URL+"download/v1/paperclips.zip"
DOWNLOAD_NAME = DOWNLOAD_SOURCE[DOWNLOAD_SOURCE.rfind('/')+1:]

if COLAB:
  PATH = "/content"
else:
  # I used this locally on my machine, you may need different
  PATH = "/Users/jeff/temp"

EXTRACT_TARGET = os.path.join(PATH,"clips")
SOURCE = os.path.join(EXTRACT_TARGET, "paperclips")
```

Next, we download the images. This part depends on the origin of your images. The following code downloads images from a URL, where a ZIP file contains the images. The code unzips the ZIP file.

In [3]:
```
# HIDE OUTPUT
!wget -O {os.path.join(PATH,DOWNLOAD_NAME)} {DOWNLOAD_SOURCE}
!mkdir -p {SOURCE}
```

```
!mkdir -p {TARGET}
!mkdir -p {EXTRACT_TARGET}
!unzip -o -j -d {SOURCE} {os.path.join(PATH, DOWNLOAD_NAME)} >/dev/null
```

```
--2022-03-01 22:45:29--  https://github.com/jeffheaton/data-mirror/release
s/download/v1/paperclips.zip
Resolving github.com (github.com)... 140.82.121.4
Connecting to github.com (github.com)|140.82.121.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://objects.githubusercontent.com/github-production-release-
asset-2e65be/408419764/25830812-b9e6-4ddf-93b6-7932d9ef5982?X-Amz-Algorith
m=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20220301%2Fus-e
ast-1%2Fs3%2Faws4_request&X-Amz-Date=20220301T224529Z&X-Amz-Expires=300&X-
Amz-Signature=92d63a15fadf8b244e13610e65457b6628f86d8465cb450a763a00f4e70f
de8f&X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=408419764&respon
se-content-disposition=attachment%3B%20filename%3Dpaperclips.zip&response-
content-type=application%2Foctet-stream [following]
--2022-03-01 22:45:29--  https://objects.githubusercontent.com/github-prod
uction-release-asset-2e65be/408419764/25830812-b9e6-4ddf-93b6-7932d9ef598
2?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2
F20220301%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20220301T224529Z&X-Am
z-Expires=300&X-Amz-Signature=92d63a15fadf8b244e13610e65457b6628f86d8465cb
450a763a00f4e70fde8f&X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=
408419764&response-content-disposition=attachment%3B%20filename%3Dpapercli
ps.zip&response-content-type=application%2Foctet-stream
Resolving objects.githubusercontent.com (objects.githubusercontent.com)...
185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to objects.githubusercontent.com (objects.githubusercontent.co
m)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 163590691 (156M) [application/octet-stream]
Saving to: '/content/paperclips.zip'

/content/paperclips 100%[===================>] 156.01M  22.9MB/s    in 6.0
s

2022-03-01 22:45:35 (26.1 MB/s) - '/content/paperclips.zip' saved [1635906
91/163590691]
```

The labels are contained in a CSV file named **train.csv**for regression. This file has just two labels, **id** and **clip_count**. The ID specifies the filename; for example, row id 1 corresponds to the file **clips-1.jpg**. The following code loads the labels for the training set and creates a new column, named **filename**, that contains the filename of each image, based on the **id** column.

In [4]:
```
import pandas as pd

df = pd.read_csv(
    os.path.join(SOURCE,"train.csv"),
    na_values=['NA', '?'])

df['filename']="clips-"+df["id"].astype(str)+".jpg"
```

This results in the following dataframe.

In [5]:
```
df
```

Out[5]:

	id	clip_count	filename
0	30001	11	clips-30001.jpg
1	30002	2	clips-30002.jpg
2	30003	26	clips-30003.jpg
3	30004	41	clips-30004.jpg
4	30005	49	clips-30005.jpg
...	...	...	...
19995	49996	35	clips-49996.jpg
19996	49997	54	clips-49997.jpg
19997	49998	72	clips-49998.jpg
19998	49999	24	clips-49999.jpg
19999	50000	35	clips-50000.jpg

20000 rows × 3 columns

Separate into a training and validation (for early stopping)

In [6]:

```
TRAIN_PCT = 0.9
TRAIN_CUT = int(len(df) * TRAIN_PCT)

df_train = df[0:TRAIN_CUT]
df_validate = df[TRAIN_CUT:]

print(f"Training size: {len(df_train)}")
print(f"Validate size: {len(df_validate)}")
```

```
Training size: 18000
Validate size: 2000
```

We are now ready to create two ImageDataGenerator objects. We currently use a generator, which creates additional training data by manipulating the source material. This technique can produce considerably stronger neural networks. The generator below flips the images both vertically and horizontally. Keras will train the neuron network both on the original images and the flipped images. This augmentation increases the size of the training data considerably. Module 6.4 goes deeper into the transformations you can perform. You can also specify a target size to resize the images automatically.

The function **flow_from_dataframe** loads the labels from a Pandas dataframe connected to our **train.csv** file. When we demonstrate classification, we will use the **flow_from_directory**; which loads the labels from the directory structure rather than a CSV.

In [7]:

```
import tensorflow as tf
import keras_preprocessing
```

```
from keras_preprocessing import image
from keras_preprocessing.image import ImageDataGenerator

training_datagen = ImageDataGenerator(
  rescale = 1./255,
  horizontal_flip=True,
  vertical_flip=True,
  fill_mode='nearest')

train_generator = training_datagen.flow_from_dataframe(
        dataframe=df_train,
        directory=SOURCE,
        x_col="filename",
        y_col="clip_count",
        target_size=(256, 256),
        batch_size=32,
        class_mode='other')

validation_datagen = ImageDataGenerator(rescale = 1./255)

val_generator = validation_datagen.flow_from_dataframe(
        dataframe=df_validate,
        directory=SOURCE,
        x_col="filename",
        y_col="clip_count",
        target_size=(256, 256),
        class_mode='other')
```

```
Found 18000 validated image filenames.
Found 2000 validated image filenames.
```

We can now train the neural network. The code to build and train the neural network is not that different than in the previous modules. We will use the Keras Sequential class to provide layers to the neural network. We now have several new layer types that we did not previously see.

- **Conv2D** - The convolution layers.
- **MaxPooling2D** - The max-pooling layers.
- **Flatten** - Flatten the 2D (and higher) tensors to allow a Dense layer to process.
- **Dense** - Dense layers, the same as demonstrated previously. Dense layers often form the final output layers of the neural network.

The training code is very similar to previously. This code is for regression, so a final linear activation is used, along with mean_squared_error for the loss function. The generator provides both the *x* and *y* matrixes we previously supplied.

In [8]:

```
from tensorflow.keras.callbacks import EarlyStopping
import time

model = tf.keras.models.Sequential([
    # Note the input shape is the desired size of the image 150x150
    # with 3 bytes color.
    # This is the first convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu',
        input_shape=(256, 256, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    # The second convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
```

```
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    # 512 neuron hidden layer
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(1, activation='linear')
])


model.summary()
epoch_steps = 250 # needed for 2.2
validation_steps = len(df_validate)
model.compile(loss = 'mean_squared_error', optimizer='adam')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
        patience=5, verbose=1, mode='auto',
        restore_best_weights=True)

start_time = time.time()
history = model.fit(train_generator,
  verbose = 1,
  validation_data=val_generator, callbacks=[monitor], epochs=25)

elapsed_time = time.time() - start_time
print("Elapsed time: {}".format(hms_string(elapsed_time)))
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 254, 254, 64)      1792

 max_pooling2d (MaxPooling2D  (None, 127, 127, 64)     0
 )

 conv2d_1 (Conv2D)           (None, 125, 125, 64)      36928

 max_pooling2d_1 (MaxPooling  (None, 62, 62, 64)       0
 2D)

 flatten (Flatten)           (None, 246016)            0

 dense (Dense)               (None, 512)               125960704

 dense_1 (Dense)             (None, 1)                 513

=================================================================
Total params: 125,999,937
Trainable params: 125,999,937
Non-trainable params: 0
_________________________________________________________________
Epoch 1/25
563/563 [==============================] - 64s 96ms/step - loss: 193.3214
- val_loss: 25.4486
Epoch 2/25
563/563 [==============================] - 52s 92ms/step - loss: 25.5836 -
val_loss: 13.8235
Epoch 3/25
563/563 [==============================] - 53s 93ms/step - loss: 18.8956 -
val_loss: 12.4469
Epoch 4/25
563/563 [==============================] - 52s 92ms/step - loss: 18.8489 -
val_loss: 20.0634
Epoch 5/25
563/563 [==============================] - 52s 92ms/step - loss: 16.5164 -
val_loss: 17.8989
Epoch 6/25
563/563 [==============================] - 52s 91ms/step - loss: 15.5483 -
val_loss: 16.3132
Epoch 7/25
563/563 [==============================] - 52s 92ms/step - loss: 15.6795 -
val_loss: 16.9717
Epoch 8/25
563/563 [==============================] - 52s 92ms/step - loss: 11.5606 -
val_loss: 12.1518
Epoch 9/25
563/563 [==============================] - 52s 92ms/step - loss: 26.0139 -
val_loss: 34.7480
Epoch 10/25
563/563 [==============================] - 52s 93ms/step - loss: 13.2884 -
val_loss: 12.1712
Epoch 11/25
563/563 [==============================] - 52s 93ms/step - loss: 10.7682 -
val_loss: 12.7467
Epoch 12/25
563/563 [==============================] - 53s 94ms/step - loss: 9.8051 -
val_loss: 9.8815
Epoch 13/25
563/563 [==============================] - 62s 109ms/step - loss: 8.2021 -
```

```
val_loss: 8.1277
Epoch 14/25
563/563 [==============================] - 52s 93ms/step - loss: 7.0807 -
val_loss: 6.8239
Epoch 15/25
563/563 [==============================] - 52s 93ms/step - loss: 6.6521 -
val_loss: 7.0292
Epoch 16/25
563/563 [==============================] - 52s 92ms/step - loss: 5.2696 -
val_loss: 6.5994
Epoch 17/25
563/563 [==============================] - 52s 93ms/step - loss: 5.1749 -
val_loss: 12.0623
Epoch 18/25
563/563 [==============================] - 52s 92ms/step - loss: 27.0990 -
val_loss: 15.1000
Epoch 19/25
563/563 [==============================] - 52s 92ms/step - loss: 10.3702 -
val_loss: 6.4094
Epoch 20/25
563/563 [==============================] - 54s 96ms/step - loss: 5.1338 -
val_loss: 5.1066
Epoch 21/25
563/563 [==============================] - 60s 107ms/step - loss: 4.1413 -
val_loss: 6.3927
Epoch 22/25
563/563 [==============================] - 55s 97ms/step - loss: 3.8659 -
val_loss: 4.6390
Epoch 23/25
563/563 [==============================] - 55s 98ms/step - loss: 3.4385 -
val_loss: 4.1845
Epoch 24/25
563/563 [==============================] - 54s 95ms/step - loss: 3.2399 -
val_loss: 4.0449
Epoch 25/25
563/563 [==============================] - 53s 94ms/step - loss: 3.2823 -
val_loss: 4.4899
Elapsed time: 0:22:22.78
```

This code will run very slowly if you do not use a GPU. The above code takes approximately 13 minutes with a GPU.

Score Regression Image Data

Scoring/predicting from a generator is a bit different than training. We do not want augmented images, and we do not wish to have the dataset shuffled. For scoring, we want a prediction for each input. We construct the generator as follows:

- shuffle=False
- batch_size=1
- class_mode=None

We use a **batch_size** of 1 to guarantee that we do not run out of GPU memory if our prediction set is large. You can increase this value for better performance. The **class_mode** is None because there is no *y*, or label. After all, we are predicting.

In [9]:
```
df_test = pd.read_csv(
```

```
    os.path.join(SOURCE,"test.csv"),
    na_values=['NA', '?'])

df_test['filename']="clips-"+df_test["id"].astype(str)+".jpg"

test_datagen = ImageDataGenerator(rescale = 1./255)

test_generator = validation_datagen.flow_from_dataframe(
        dataframe=df_test,
        directory=SOURCE,
        x_col="filename",
        batch_size=1,
        shuffle=False,
        target_size=(256, 256),
        class_mode=None)
```

```
Found 5000 validated image filenames.
```

We need to reset the generator to ensure we are always at the beginning.

In [10]:
```
test_generator.reset()
pred = model.predict(test_generator,steps=len(df_test))
```

We can now generate a CSV file to hold the predictions.

In [11]:
```
df_submit = pd.DataFrame({'id':df_test['id'],'clip_count':pred.flatten()}
df_submit.to_csv(os.path.join(PATH,"submit.csv"),index=False)
```

Classification Neural Networks

Just like earlier in this module, we will load data. However, this time we will use a dataset of images of three different types of the iris flower. This zip file contains three different directories that specify each image's label. The directories are named the same as the labels:

- iris-setosa
- iris-versicolour
- iris-virginica

In [12]:
```
import os

URL = "https://github.com/jeffheaton/data-mirror/releases"
DOWNLOAD_SOURCE = URL+"/download/v1/iris-image.zip"
DOWNLOAD_NAME = DOWNLOAD_SOURCE[DOWNLOAD_SOURCE.rfind('/')+1:]

if COLAB:
  PATH = "/content"
  EXTRACT_TARGET = os.path.join(PATH,"iris")
  SOURCE = EXTRACT_TARGET # In this case its the same, no subfolder
else:
  # I used this locally on my machine, you may need different
  PATH = "/Users/jeff/temp"
  EXTRACT_TARGET = os.path.join(PATH,"iris")
  SOURCE = EXTRACT_TARGET # In this case its the same, no subfolder
```

Just as before, we unzip the images.

In [13]:
```
# HIDE OUTPUT
!wget -O {os.path.join(PATH,DOWNLOAD_NAME)} {DOWNLOAD_SOURCE}
!mkdir -p {SOURCE}
!mkdir -p {TARGET}
!mkdir -p {EXTRACT_TARGET}
!unzip -o -d {EXTRACT_TARGET} {os.path.join(PATH, DOWNLOAD_NAME)} >/dev/
```

```
--2022-03-01 23:08:29--  https://github.com/jeffheaton/data-mirror/release
s/download/v1/iris-image.zip
Resolving github.com (github.com)... 140.82.121.4
Connecting to github.com (github.com)|140.82.121.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://objects.githubusercontent.com/github-production-release-
asset-2e65be/408419764/d548babd-36c3-414e-add2-a5d9ab941e6e?X-Amz-Algorith
m=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20220301%2Fus-e
ast-1%2Fs3%2Faws4_request&X-Amz-Date=20220301T230829Z&X-Amz-Expires=300&X-
Amz-Signature=f9315f39373d1b8e6ae60a618db5da58714c8bab017e0f55b38c7c0a55d2
cb20&X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=408419764&respon
se-content-disposition=attachment%3B%20filename%3Diris-image.zip&response-
content-type=application%2Foctet-stream [following]
--2022-03-01 23:08:30--  https://objects.githubusercontent.com/github-prod
uction-release-asset-2e65be/408419764/d548babd-36c3-414e-add2-a5d9ab941e6
e?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2
F20220301%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20220301T230829Z&X-Am
z-Expires=300&X-Amz-Signature=f9315f39373d1b8e6ae60a618db5da58714c8bab017e
0f55b38c7c0a55d2cb20&X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=
408419764&response-content-disposition=attachment%3B%20filename%3Diris-ima
ge.zip&response-content-type=application%2Foctet-stream
Resolving objects.githubusercontent.com (objects.githubusercontent.com)...
185.199.109.133, 185.199.110.133, 185.199.108.133, ...
Connecting to objects.githubusercontent.com (objects.githubusercontent.co
m)|185.199.109.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5587253 (5.3M) [application/octet-stream]
Saving to: '/content/iris-image.zip'

/content/iris-image 100%[===================>]   5.33M  6.65MB/s    in 0.8
s

2022-03-01 23:08:31 (6.65 MB/s) - '/content/iris-image.zip' saved [558725
3/5587253]
```

You can see these folders with the following command.

In [14]:
```
!ls /content/iris
```

```
iris-setosa  iris-versicolour  iris-virginica
```

We set up the generator, similar to before. This time we use flow_from_directory to get the labels from the directory structure.

In [15]:
```
import tensorflow as tf
import keras_preprocessing
from keras_preprocessing import image
from keras_preprocessing.image import ImageDataGenerator

training_datagen = ImageDataGenerator(
  rescale = 1./255,
```

```
  horizontal_flip=True,
  vertical_flip=True,
  width_shift_range=[-200,200],
  rotation_range=360,

  fill_mode='nearest')

train_generator = training_datagen.flow_from_directory(
    directory=SOURCE, target_size=(256, 256),
    class_mode='categorical', batch_size=32, shuffle=True)

validation_datagen = ImageDataGenerator(rescale = 1./255)

validation_generator = validation_datagen.flow_from_directory(
    directory=SOURCE, target_size=(256, 256),
    class_mode='categorical', batch_size=32, shuffle=True)
```

```
Found 421 images belonging to 3 classes.
Found 421 images belonging to 3 classes.
```

Training the neural network with classification is similar to regression.

In [16]:

```
from tensorflow.keras.callbacks import EarlyStopping

class_count = len(train_generator.class_indices)

model = tf.keras.models.Sequential([
    # Note the input shape is the desired size of the image
    # 300x300 with 3 bytes color
    # This is the first convolution
    tf.keras.layers.Conv2D(16, (3,3), activation='relu',
        input_shape=(256, 256, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    # The second convolution
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.MaxPooling2D(2,2),
    # The third convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.MaxPooling2D(2,2),
    # The fourth convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # The fifth convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # Flatten the results to feed into a DNN

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.5),
    # 512 neuron hidden layer
    tf.keras.layers.Dense(512, activation='relu'),
    # Only 1 output neuron. It will contain a value from 0-1
    tf.keras.layers.Dense(class_count, activation='softmax')
])

model.summary()

model.compile(loss = 'categorical_crossentropy', optimizer='adam')

model.fit(train_generator, epochs=50, steps_per_epoch=10,
```

```
                    verbose = 1)
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)            (None, 254, 254, 16)      448

 max_pooling2d_2 (MaxPooling  (None, 127, 127, 16)     0
 2D)

 conv2d_3 (Conv2D)            (None, 125, 125, 32)      4640

 dropout (Dropout)            (None, 125, 125, 32)      0

 max_pooling2d_3 (MaxPooling  (None, 62, 62, 32)       0
 2D)

 conv2d_4 (Conv2D)            (None, 60, 60, 64)        18496

 dropout_1 (Dropout)          (None, 60, 60, 64)        0

 max_pooling2d_4 (MaxPooling  (None, 30, 30, 64)       0
 2D)

 conv2d_5 (Conv2D)            (None, 28, 28, 64)        36928

 max_pooling2d_5 (MaxPooling  (None, 14, 14, 64)       0
 2D)

 conv2d_6 (Conv2D)            (None, 12, 12, 64)        36928

 max_pooling2d_6 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 flatten_1 (Flatten)          (None, 2304)              0

 dropout_2 (Dropout)          (None, 2304)              0

 dense_2 (Dense)              (None, 512)               1180160

 dense_3 (Dense)              (None, 3)                 1539

=================================================================
Total params: 1,279,139
Trainable params: 1,279,139
Non-trainable params: 0
_________________________________________________________________
Epoch 1/50
10/10 [==============================] - 6s 486ms/step - loss: 1.0254
Epoch 2/50
10/10 [==============================] - 5s 472ms/step - loss: 0.9060
Epoch 3/50
10/10 [==============================] - 5s 474ms/step - loss: 0.9712
Epoch 4/50
10/10 [==============================] - 5s 520ms/step - loss: 0.9099
Epoch 5/50
10/10 [==============================] - 5s 517ms/step - loss: 0.9061
Epoch 6/50
10/10 [==============================] - 5s 510ms/step - loss: 0.8965
Epoch 7/50
10/10 [==============================] - 5s 458ms/step - loss: 0.8909
Epoch 8/50
10/10 [==============================] - 5s 514ms/step - loss: 0.8941
Epoch 9/50
```

```
10/10 [==============================] - 5s 493ms/step - loss: 0.9248
Epoch 10/50
10/10 [==============================] - 5s 500ms/step - loss: 0.8780
Epoch 11/50
10/10 [==============================] - 5s 453ms/step - loss: 0.8724
Epoch 12/50
10/10 [==============================] - 5s 448ms/step - loss: 0.8901
Epoch 13/50
10/10 [==============================] - 5s 456ms/step - loss: 0.8817
Epoch 14/50
10/10 [==============================] - 5s 465ms/step - loss: 0.9040
Epoch 15/50
10/10 [==============================] - 5s 449ms/step - loss: 0.8779
Epoch 16/50
10/10 [==============================] - 4s 441ms/step - loss: 0.8479
Epoch 17/50
10/10 [==============================] - 5s 499ms/step - loss: 0.8713
Epoch 18/50
10/10 [==============================] - 5s 456ms/step - loss: 0.8432
Epoch 19/50
10/10 [==============================] - 4s 444ms/step - loss: 0.8816
Epoch 20/50
10/10 [==============================] - 5s 508ms/step - loss: 0.8791
Epoch 21/50
10/10 [==============================] - 5s 497ms/step - loss: 0.8553
Epoch 22/50
10/10 [==============================] - 5s 448ms/step - loss: 0.8275
Epoch 23/50
10/10 [==============================] - 5s 502ms/step - loss: 0.8216
Epoch 24/50
10/10 [==============================] - 5s 456ms/step - loss: 0.8739
Epoch 25/50
10/10 [==============================] - 5s 510ms/step - loss: 0.8650
Epoch 26/50
10/10 [==============================] - 5s 456ms/step - loss: 0.8405
Epoch 27/50
10/10 [==============================] - 5s 456ms/step - loss: 0.8729
Epoch 28/50
10/10 [==============================] - 5s 499ms/step - loss: 0.8618
Epoch 29/50
10/10 [==============================] - 5s 500ms/step - loss: 0.8125
Epoch 30/50
10/10 [==============================] - 5s 504ms/step - loss: 0.8813
Epoch 31/50
10/10 [==============================] - 5s 508ms/step - loss: 0.8392
Epoch 32/50
10/10 [==============================] - 5s 449ms/step - loss: 0.8377
Epoch 33/50
10/10 [==============================] - 5s 499ms/step - loss: 0.8509
Epoch 34/50
10/10 [==============================] - 5s 454ms/step - loss: 0.8647
Epoch 35/50
10/10 [==============================] - 5s 466ms/step - loss: 0.8874
Epoch 36/50
10/10 [==============================] - 5s 502ms/step - loss: 0.9221
Epoch 37/50
10/10 [==============================] - 5s 511ms/step - loss: 0.9186
Epoch 38/50
10/10 [==============================] - 5s 496ms/step - loss: 0.8549
Epoch 39/50
10/10 [==============================] - 5s 493ms/step - loss: 0.9194
Epoch 40/50
10/10 [==============================] - 5s 496ms/step - loss: 0.8528
```

```
Epoch 41/50
10/10 [==============================] - 5s 453ms/step - loss: 0.9105
Epoch 42/50
10/10 [==============================] - 5s 454ms/step - loss: 0.8462
Epoch 43/50
10/10 [==============================] - 5s 459ms/step - loss: 0.8858
Epoch 44/50
10/10 [==============================] - 5s 497ms/step - loss: 0.9119
Epoch 45/50
10/10 [==============================] - 5s 458ms/step - loss: 0.8799
Epoch 46/50
10/10 [==============================] - 5s 499ms/step - loss: 0.8582
Epoch 47/50
10/10 [==============================] - 5s 493ms/step - loss: 0.8536
Epoch 48/50
10/10 [==============================] - 5s 490ms/step - loss: 0.8669
Epoch 49/50
10/10 [==============================] - 5s 458ms/step - loss: 0.7957
Epoch 50/50
10/10 [==============================] - 5s 501ms/step - loss: 0.8670
```

The iris image dataset is not easy to predict; it turns out that a tabular dataset of measurements is more manageable. However, we can achieve a 63%.

In [22]:

```
from sklearn.metrics import accuracy_score
import numpy as np

validation_generator.reset()
pred = model.predict(validation_generator)

predict_classes = np.argmax(pred,axis=1)
expected_classes = validation_generator.classes

correct = accuracy_score(expected_classes,predict_classes)
print(f"Accuracy: {correct}")
```

```
Accuracy: 0.6389548693586699
```

Other Resources

- Imagenet:Large Scale Visual Recognition Challenge 2014
- Andrej Karpathy - PhD student/instructor at Stanford.
- CS231n Convolutional Neural Networks for Visual Recognition - Stanford course on computer vision/CNN's.
- CS231n - GitHub
- ConvNetJS - JavaScript library for deep learning.