"DeepLearning.AI TensorFlow Developer" Specialization

Andrii X

1 Introduction to TensorFlow for AI, ML, and DL

1.1 Week 1: First NN

- **Simple example aka "Hello, World!":**
  Define NN (1 layer with 1 neuron):

  ```
  # Build a simple Sequential model
  model = tf.keras.Sequential(
  [keras.layers.Dense(units=1, input_shape=[1])]
  )
  ```

  Compile the model:

  ```
  model.compile(optimizer='sgd', loss='mean_squared_error')
  ```

  Provide the data:

  ```
  # Declare model inputs and outputs for training
  xs = np.array([-1.0,  0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
  ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)
  ```

  Train the NN:

  ```
  # Train the model
  model.fit(xs, ys, epochs=500)
  ```

  Use trained NN for new data:

  ```
  # Make a prediction
  print(model.predict([10.0]))
  ```

1.2 Week 2: Intro to CV

- **A Computer Vision Example: Fashion MNIST dataset**
  The Fashion MNIST dataset is a collection of grayscale 28x28 pixel clothing images.
  1) Load the Fashion MNIST dataset:

  ```
  fmnist = tf.keras.datasets.fashion_mnist
  ```

  2) Load the training and test split of the Fashion MNIST dataset:

  ```
  (training_images, training_labels),
  (test_images, test_labels) = fmnist.load_data()
  ```

  3) Normalize the pixel values of the train and test images:

  ```
  training_images  = training_images / 255.0
  test_images = test_images / 255.0
  ```

  4) Build the classification model:

  ```
  model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation=tf.nn.relu),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
  ```

  Sequential - defines a sequence of layers in the neural network.
  Flatten - converts a 28x28 matrix into a 1-D array.
  Dense - adds a layer of neurons. Activation function **relu** passes values greater than 0 to the next layer.
  Softmax takes a list of values and scales these so the sum of all elements will be equal to 1. When applied to model outputs, you can think of the scaled values as the probability for that class.
  5) Compile and train the model:

  ```
  model.compile(optimizer = tf.optimizers.Adam(),
  loss = 'sparse_categorical_crossentropy',
  metrics=['accuracy'])

  model.fit(training_images, training_labels, epochs=5)
  ```

  6) Evaluate the model on unseen data

  ```
  model.evaluate(test_images, test_labels)
  ```

  Exploration Exercises
  ex1: the below code c**reates a set of classifications for each of the test images**, and then prints the first entry in the classifications.

```
classifications = model.predict(test_images)
print(classifications[0])
# The output of the model is a list of 10 numbers.
# These numbers are a probability that the value
# being classified is the corresponding value
```

ex2: **adding more Neurons** we have to do more calculations, slowing down the process, but in this case they have a good impact – **we do get more accurate**. That doesn't mean it's always a case of 'more is better', **you can hit the law of diminishing returns very quickly**!
ex3: it may seem vague right now, but it reinforces **the rule of thumb that the first layer in your network should be the same shape as your data**. Right now our data is 28x28 images, and 28 layers of 28 neurons would be infeasible, so it makes more sense to 'flatten' that 28,28 into a 784x1.
ex4: **another rule of thumb** – the number of neurons in the last layer should match the number of classes you are classifying for.
ex5: consider the effects of additional layers in the network. There isn't a significant impact – because this is relatively simple data. **For far more complex data** (including color images to be classified as flowers that you'll see in the next lesson), **extra layers are often necessary**.
ex6: consider the **impact of training for more or less epochs**. Try 15 epochs – you'll probably get a model with a much better loss than the one with 5. Try 30 epochs – you might see the loss value decrease more slowly, and sometimes increases. This is a side effect of something called **'overfitting'**.
ex7: If you try to train the model without normalizing the data, you might find that the model takes longer to train, or that it's unable to learn effectively from the training data, leading to poorer performance on the test data. The reason you get different results with and without normalization is because the scale of the inputs can significantly impact the gradient of the loss function, and hence the updates to the weights during training. **Normalization ensures that the scale of the inputs is consistent**, which can make the training process more stable and efficient.
ex8: 'wouldn't it be nice if I could stop the training when I reach a desired value?' – i.e. 85% accuracy might be enough for you, and if you reach that after 3 epochs, why sit around waiting for it to finish a lot more epochs.... you have callbacks!

```
class myCallback(tf.keras.callbacks.Callback):
def on_epoch_end(self, epoch, logs={}):
if logs.get('accuracy') is not None
and logs.get('accuracy') > 0.60:
print("\nReached 60% accuracy so cancelling training!")
self.model.stop_training = True

callbacks = myCallback()
```

```
fmnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels) ,
(test_images, test_labels) = fmnist.load_data()

training_images=training_images/255.0
test_images=test_images/255.0
model = tf.keras.models.Sequential([
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(128, activation=tf.nn.relu),
tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
# model fitting with callback
model.fit(training_images, training_labels,
epochs=5, callbacks=[callbacks])
```

1.3 Week 3: Convolutional NN

- **Improving Computer Vision Accuracy using Convolutions**
  A neural network containing three layers – the input layer (in the shape of the data), the output layer (in the shape of the desired output) and only one hidden layer gives accuracy about 89% on training and 87% on validation. How does one make that even better? One way is to use something called **convolutions**. The ultimate **concept** is that they **narrow down the content of the image to focus on specific parts** and this will likely improve the model accuracy. This is perfect for computer vision because **it often highlights features that distinguish one item from another**. Moreover, the **amount of information needed is then much less** because one will just **train on the highlighted features**. That's the concept of **Convolutional Neural Networks**.

```
# Define the model
model = tf.keras.models.Sequential([

# Add convolutions and max pooling
tf.keras.layers.Conv2D(32, (3,3), activation='relu',
input_shape=(28, 28, 1)),
tf.keras.layers.MaxPooling2D(2, 2),
tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
tf.keras.layers.MaxPooling2D(2,2),

# Add the same layers as before
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(128, activation='relu'),
tf.keras.layers.Dense(10, activation='softmax')
])

# Print the model summary
model.summary()

# Use same settings
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Train the model
print(f'\nMODEL TRAINING:')
model.fit(training_images, training_labels, epochs=5)

# Evaluate on the test set
print(f'\nMODEL EVALUATION:')
test_loss = model.evaluate(test_images, test_labels)
```

- **Lab**:

```
import os
import numpy as np
import tensorflow as tf
from tensorflow import keras

# Load the data
# Get current working directory
current_dir = os.getcwd()
# Append data/mnist.npz to the previous path to get
# the full path
data_path = os.path.join(current_dir, "data/mnist.npz")
# Get only training set
(training_images, training_labels), _ =
tf.keras.datasets.mnist.load_data(path=data_path)
```

Pre-processing the data:
– Reshape the data so that it has an extra dimension. The reason for this is that commonly you will use 3-dimensional arrays (without counting the batch dimension) to represent image data. The third dimension represents the color using RGB values. This data might be in black and white format so the third dimension doesn't really add any additional information for the classification process but it is a good practice regardless.
– Normalize the pixel values so that these are values between 0 and 1. You can achieve this by dividing every value in the array by the maximum.

```
def reshape_and_normalize(images):
### START CODE HERE
```

```
# Reshape the images to add an extra dimension
images = images.reshape(images.shape[0],
images.shape[1],
images.shape[2],
1)
# Normalize pixel values
images = images / 255.0
### END CODE HERE
return images

# rest of the code like previously
```

1.4 Week 4: Usage of real-world images

- **Lab 1: Training with ImageDataGenerator: Building a Small Model from Scratch**

```
import tensorflow as tf

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=(300, 300, 3)),
tf.keras.layers.MaxPooling2D(2, 2),
# The second convolution
tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
tf.keras.layers.MaxPooling2D(2,2),
# The third convolution
tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
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(),
# 512 neuron hidden layer
tf.keras.layers.Dense(512, activation='relu'),
# Only 1 output neuron. It will contain a value
# from 0-1 where 0 for 1 class ('horses') and 1 for
# the other ('humans')
tf.keras.layers.Dense(1, activation='sigmoid')
])
```

The **increasing number of filters** in deeper layers allows the network to **capture a hierarchy of features**, from simple to complex, which is essential for the network to understand and classify intricate patterns in images.

```
from tensorflow.keras.optimizers import RMSprop

model.compile(loss='binary_crossentropy',
optimizer=RMSprop(learning_rate=0.001),
metrics=['accuracy'])
```

In this case, using the **RMSprop optimization algorithm** is preferable to stochastic gradient descent (SGD), because **RMSprop automates learning-rate tuning** for us. (Other optimizers, such as Adam and Adagrad, also automatically adapt the learning rate during training, and would work equally well here.)

Data Preprocessing

Next step is to set up the data generators that will read pictures in the source folders, convert them to float32 tensors, and feed them (with their labels) to the model.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1/255)

# Flow training images in batches of 128 using train_datagen generator
train_generator = train_datagen.flow_from_directory(
'./horse-or-human/',  # This is the source directory for training images
target_size=(300, 300),  # All images will be resized to 300x300
batch_size=128,
# Since we use binary_crossentropy loss, we need binary labels
class_mode='binary')
```

Training

```
history = model.fit(train_generator,
steps_per_epoch=8, epochs=15, verbose=1)
```

As a result, a **convnet** processes images by transforming pixels through layers into abstract representations, **emphasizing key features** and achieving representation sparsity, which refines information about the image's class.

- **Lab 2: ImageDataGenerator with a Validation Set**
  Building a Small Model from Scratch

```
# the same model architecture as before
# in previous lab 1
# the same compile settings as before
```

Data Preprocessing
It will mostly be the same as last time but notice the additional code to also prepare the validation data.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1/255)
validation_datagen = ImageDataGenerator(rescale=1/255)

# Flow training images in batches of 128 using train_datagen generator
train_generator = train_datagen.flow_from_directory(
'./horse-or-human/',  # This is the source directory for training images
target_size=(300, 300),  # All images will be resized to 300x300
batch_size=128,
# Since you use binary_crossentropy loss, you need binary labels
class_mode='binary')

# Flow validation images in batches of 128 using validation_datagen generator
validation_generator = validation_datagen.flow_from_directory(
'./validation-horse-or-human/',  # This is the source directory for validation images
target_size=(300, 300),  # All images will be resized to 300x300
batch_size=32,
# Since you use binary_crossentropy loss, you need binary labels
class_mode='binary')
```

Training
Notice that as you train with more epochs, your training accuracy might go up but your validation accuracy goes down. This can be a sign of overfitting and you need to prevent your model from reaching this point.

```
history = model.fit(
train_generator,
steps_per_epoch=8,
epochs=15,
verbose=1,
validation_data = validation_generator,
validation_steps=8)
```

- **Final Assignment: Happy or Sad**
  The happy or sad dataset, which contains 80 images of emoji-like faces, 40 happy and 40 sad.
  The task is to create a convolutional neural network that trains to 99.9%

accuracy on these images, which cancels training upon hitting this training accuracy threshold.

```
# IMPORTS
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import os

# LOAD AND EXPLORE THE DATA
from tensorflow.keras.preprocessing.image import load_img
from tensorflow.keras.preprocessing.image import img_to_array

base_dir = "./data/"
happy_dir = os.path.join(base_dir, "happy/")
sad_dir = os.path.join(base_dir, "sad/")

print("Sample happy image:")
plt.imshow(load_img(f"{os.path.join(happy_dir, os.listdir(happy_dir)[0])}"))
plt.show()

print("\nSample sad image:")
plt.imshow(load_img(f"{os.path.join(sad_dir, os.listdir(sad_dir)[0])}"))
plt.show()

# Load the first example of a happy face
sample_image  = load_img(f"{os.path.join(happy_dir, os.listdir(happy_dir)[0])}")
# Convert the image into its numpy array representation
sample_array = img_to_array(sample_image)
print(f"Each image has shape: {sample_array.shape}")
print(f"The maximum pixel value used is: {np.max(sample_array)}")
```

Defining the callback
The **EarlyStopping callback** will monitor the validation loss (val_loss) by default. If **the validation loss does not improve** for a specified number of epochs (defined by the patience parameter), **the training will stop**, and the **best weights** (from the epoch with the lowest validation loss) **will be restored** to the model.
Note: To use the EarlyStopping callback effectively, you should have a validation set defined when calling model.fit().

```
from tensorflow.keras.callbacks import EarlyStopping

# Define the EarlyStopping callback
early_stopping = EarlyStopping(monitor='val_loss', # or 'val_accuracy' depending
# on what you want to monitor
patience=10, # Number of epochs with no improvement after which training will
# be stopped
verbose=1,
restore_best_weights=True) # Restore model weights from the
# epoch with the best value of the monitored quantity.
# Now, when you fit the model, you can use this callback:
# model.fit(..., callbacks=[early_stopping])

class myCallback(tf.keras.callbacks.Callback):
def on_epoch_end(self, epoch, logs={}):
if logs.get('accuracy') is not None and logs.get('accuracy') > 0.999:
print("\nReached 99.9% accuracy so cancelling training!")
self.model.stop_training = True
```

Pre-processing the data:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

def image_generator():
        # Instantiate the ImageDataGenerator class.
        train_datagen = ImageDataGenerator(rescale=1/255.0)

        train_generator = train_datagen.flow_from_directory(directory=base_dir,
        target_size=(150, 150),
        batch_size=10,
        class_mode='binary')

        return train_generator

gen = image_generator()
```

Creating and training your model:

```
from tensorflow.keras import optimizers, losses

def train_happy_sad_model(train_generator):

        # Instantiate the callback
        callbacks = myCallback()

        # Define the model
        model = tf.keras.models.Sequential([
        # This is the first convolution
        tf.keras.layers.Conv2D(16, (3,3), activation='relu', input_shape=(150, 150, 3))
        tf.keras.layers.MaxPooling2D(2, 2),
        # The second convolution
        tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
        tf.keras.layers.MaxPooling2D(2,2),
    # The third convolution
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation = 'relu'),
    tf.keras.layers.Dense(1, activation = 'sigmoid')
    ])

    # Compile the model
    model.compile(loss='binary_crossentropy',
        optimizer=optimizers.RMSprop(learning_rate=0.001),
        metrics=['accuracy'])

    # Train the model
    history = model.fit(x=train_generator,
    epochs=20,
    callbacks=[callbacks]
    )

    return history

hist = train_happy_sad_model(gen)
```

*__Congratulations on finishing__ the last assignment of this course!

2 Convolutional Neural Networks in TensorFlow

2.1 Week 1: Larger Dataset

- **Lab: More sophisticated images with CNN**

```
# -> Download the dataset
# !wget --no-check-certificate https://storage.googleapis.com/mledu-datasets/cats_and_

# -> Unzip it
import zipfile
# Unzip the archive
local_zip = './cats_and_dogs_filtered.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall()
zip_ref.close()

# -> Subdirectories
import os
base_dir = 'cats_and_dogs_filtered'
```

```python
print("Contents of base directory:")
print(os.listdir(base_dir))
print("\nContents of train directory:")
print(os.listdir(f'{base_dir}/train'))
print("\nContents of validation directory:")
print(os.listdir(f'{base_dir}/validation'))

# -> Assign each of these directories to a variable
import os
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
# Directory with training cat/dog pictures
train_cats_dir = os.path.join(train_dir, 'cats')
train_dogs_dir = os.path.join(train_dir, 'dogs')
# Directory with validation cat/dog pictures
validation_cats_dir = os.path.join(validation_dir, 'cats')
validation_dogs_dir = os.path.join(validation_dir, 'dogs')

# -> Filenames
train_cat_fnames = os.listdir( train_cats_dir )
train_dog_fnames = os.listdir( train_dogs_dir )
print(train_cat_fnames[:10])
print(train_dog_fnames[:10])

# -> Total number of cat and dog images in the train
# and validation directories
print('total training cat images :', len(os.listdir(      train_cats_dir ) ))
print('total training dog images :', len(os.listdir(      train_dogs_dir ) ))
print('total validation cat images :', len(os.listdir( validation_cats_dir ) ))
print('total validation dog images :', len(os.listdir( validation_dogs_dir ) ))

# -> Building a Small Model from Scratch to get to ~72% Accuracy ->
import tensorflow as tf
model = tf.keras.models.Sequential([
        # Note the input shape is the desired size of the image 150x150
        # with 3 bytes color
        tf.keras.layers.Conv2D(16, (3,3), activation='relu', input_shape=(150, 150, 3))
        tf.keras.layers.MaxPooling2D(2,2),
        tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
        tf.keras.layers.MaxPooling2D(2,2),
        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(),
        # 512 neuron hidden layer
        tf.keras.layers.Dense(512, activation='relu'),
        # Only 1 output neuron. It will contain a value from 0-1
        # where 0 for 1 class ('cats') and 1 for the other ('dogs')
        tf.keras.layers.Dense(1, activation='sigmoid')
])
model.summary()

# -> Model configuration
from tensorflow.keras.optimizers import RMSprop
model.compile(optimizer=RMSprop(learning_rate=0.001),
loss='binary_crossentropy',
metrics = ['accuracy'])

# -> Data Preprocessing
from tensorflow.keras.preprocessing.image import ImageDataGenerator
# All images will be rescaled by 1./255.
train_datagen = ImageDataGenerator( rescale = 1.0/255. )
test_datagen  = ImageDataGenerator( rescale = 1.0/255. )
# --------------------
# Flow training images in batches of 20 using train_datagen generator
# --------------------
train_generator = train_datagen.flow_from_directory(train_dir,
        batch_size=20,
        class_mode='binary',
        target_size=(150, 150))
# --------------------
# Flow validation images in batches of 20 using test_datagen generator
# --------------------
validation_generator =  test_datagen.flow_from_directory(validation_dir,
        batch_size=20,
        class_mode  = 'binary',
        target_size = (150, 150))

# -> Training
history = model.fit(
        train_generator,
        epochs=15,
        validation_data=validation_generator,
        verbose=2
        )

# -> Take a look at actually running a prediction using the model
import numpy as np
from google.colab import files
from tensorflow.keras.utils import load_img, img_to_array
uploaded=files.upload()
for fn in uploaded.keys():
```

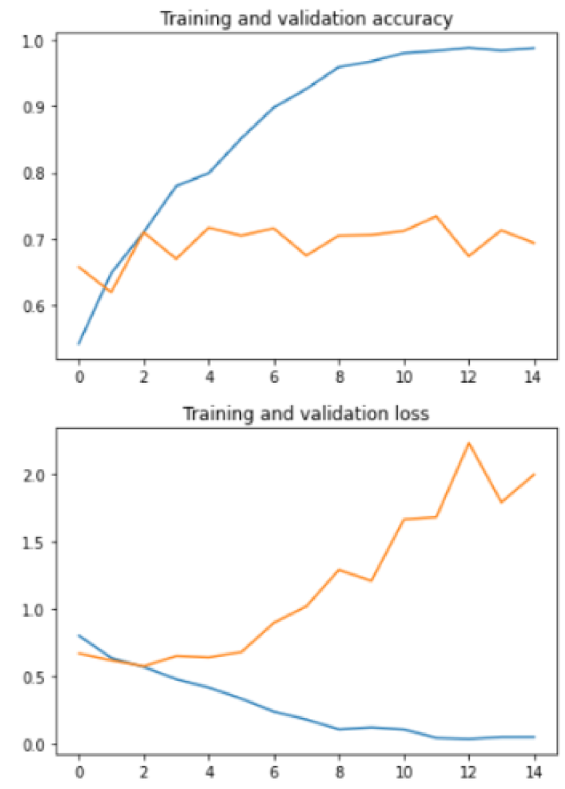
```
# predicting images
path='/content/' + fn
img=load_img(path, target_size=(150, 150))
x=img_to_array(img)
x /= 255
x=np.expand_dims(x, axis=0)
images = np.vstack([x])
classes = model.predict(images, batch_size=10)
print(classes[0])
if classes[0]>0.5:
print(fn + " is a dog")
else:
print(fn + " is a cat")

# -> Visualizing Intermediate Representations
import numpy as np
import random
from tensorflow.keras.utils import img_to_array, load_img
# Define a new Model that will take an image as input, and will output
# intermediate representations for all layers in the previous model
successive_outputs = [layer.output for layer in model.layers]
visualization_model = tf.keras.models.Model(inputs = model.input, outputs = successive_
# Prepare a random input image from the training set.
cat_img_files = [os.path.join(train_cats_dir, f) for f in train_cat_fnames]
dog_img_files = [os.path.join(train_dogs_dir, f) for f in train_dog_fnames]
img_path = random.choice(cat_img_files + dog_img_files)
img = load_img(img_path, target_size=(150, 150))  # this is a PIL image
x   = img_to_array(img)                           # Numpy array with shape (150, 150, 3
x   = x.reshape((1,) + x.shape)                   # Numpy array with shape (1, 150, 150
# Scale by 1/255
x /= 255.0
# Run the image through the network, thus obtaining all
# intermediate representations for this image.
successive_feature_maps = visualization_model.predict(x)
# These are the names of the layers, so you can have them as part of our plot
layer_names = [layer.name for layer in model.layers]
# Display the representations
for layer_name, feature_map in zip(layer_names, successive_feature_maps):
if len(feature_map.shape) == 4:
#-------------------------------------------
# Just do this for the conv / maxpool layers, not the fully-connected layers
#-------------------------------------------
n_features = feature_map.shape[-1]  # number of features in the feature map
size       = feature_map.shape[ 1]  # feature map shape (1, size, size, n_features)
# Tile the images in this matrix
display_grid = np.zeros((size, size * n_features))
```

```
#--------------------------------------------------
# Postprocess the feature to be visually palatable
#--------------------------------------------------
for i in range(n_features):
x  = feature_map[0, :, :, i]
x -= x.mean()
x /= x.std ()
x *=  64
x += 128
x  = np.clip(x, 0, 255).astype('uint8')
display_grid[:, i * size : (i + 1) * size] = x # Tile each filter into a horizontal gri
#-----------------
# Display the grid
#-----------------
scale = 20. / n_features
plt.figure( figsize=(scale * n_features, scale) )
plt.title ( layer_name )
plt.grid  ( False )
plt.imshow( display_grid, aspect='auto', cmap='viridis' )

# -> Evaluating Accuracy and Loss for the Model
#------------------------------------------------------------
# Retrieve a list of list results on training and test data
# sets for each training epoch
#------------------------------------------------------------
acc      = history.history[     'accuracy' ]
val_acc  = history.history[ 'val_accuracy' ]
loss     = history.history[    'loss' ]
val_loss = history.history['val_loss' ]
epochs   = range(len(acc)) # Get number of epochs
#------------------------------------------------
# Plot training and validation accuracy per epoch
#------------------------------------------------
plt.plot  ( epochs,     acc )
plt.plot  ( epochs, val_acc )
plt.title ('Training and validation accuracy')
plt.figure()
#------------------------------------------------
# Plot training and validation loss per epoch
#------------------------------------------------
plt.plot  ( epochs,     loss )
plt.plot  ( epochs, val_loss )
plt.title ('Training and validation loss'   )
```

```
Text(0.5, 1.0, 'Training and validation loss')
```



The model is **overfitting** like it's getting out of fashion. The training accuracy (in blue) gets close to 100% while the validation accuracy (in orange) stalls as 70%. The validation loss reaches its minimum after only five epochs.

Since you have **a relatively small number of training examples (2000), overfitting should be the number one concern**. Overfitting happens when a model exposed to too few examples learns patterns that do not generalize to new data, i.e. when the model starts using irrelevant features for making predictions.

- **Week 1 Assignment: Cats vs Dogs**

```
import os
import zipfile
import random
import shutil
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from shutil import copyfile
```

```python
import matplotlib.pyplot as plt

# -> Download the dataset
!wget --no-check-certificate \
"https://download.microsoft.com/download/3/E/1/3E1C3F21-ECDB-4869-8368-6DEBA77B919F/kag
-O "/tmp/cats-and-dogs.zip"
local_zip = '/tmp/cats-and-dogs.zip'
zip_ref   = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('/tmp')
zip_ref.close()
source_path = '/tmp/PetImages'
source_path_dogs = os.path.join(source_path, 'Dog')
source_path_cats = os.path.join(source_path, 'Cat')
# Deletes all non-image files (there are two .db files bundled into the dataset)
!find /tmp/PetImages/ -type f ! -name "*.jpg" -exec rm {} +
# os.listdir returns a list containing all files under the given path
print(f"There are {len(os.listdir(source_path_dogs))} images of dogs.")
print(f"There are {len(os.listdir(source_path_cats))} images of cats.")
# Define root directory
root_dir = '/tmp/cats-v-dogs'
# Empty directory to prevent FileExistsError is the function is run several times
if os.path.exists(root_dir):
shutil.rmtree(root_dir)

# -> GRADED FUNCTION: create_train_val_dirs
def create_train_val_dirs(root_path):
        """
        Creates directories for the train and test sets
        Args:
        root_path (string) - the base directory path to create subdirectories from
        Returns:
        None
        """
        # Create training and validation directories
        training_dir = os.path.join(root_path, 'training')
        validation_dir = os.path.join(root_path, 'validation')
        # Create subdirectories for cats and dogs under training and validation directo
        os.makedirs(os.path.join(training_dir, 'cats'))
        os.makedirs(os.path.join(training_dir, 'dogs'))
        os.makedirs(os.path.join(validation_dir, 'cats'))
        os.makedirs(os.path.join(validation_dir, 'dogs'))
try:
        create_train_val_dirs(root_path=root_dir)
except FileExistsError:
        print("You should not be seeing this since the upper directory is removed befor
# Test your create_train_val_dirs function
```

```
for rootdir, dirs, files in os.walk(root_dir):
        for subdir in dirs:
                print(os.path.join(rootdir, subdir))

# -> GRADED FUNCTION: split_data
def split_data(SOURCE_DIR, TRAINING_DIR, VALIDATION_DIR, SPLIT_SIZE):
        """
        Splits the data into train and test sets
        Args:
        SOURCE_DIR (string): directory path containing the images
        TRAINING_DIR (string): directory path to be used for training
        VALIDATION_DIR (string): directory path to be used for validation
        SPLIT_SIZE (float): proportion of the dataset to be used for training
        Returns:
        None
        """
        files = os.listdir(SOURCE_DIR)
        # Filter out files with zero length
        files = [f for f in files if os.path.getsize(os.path.join(SOURCE_DIR, f)) > 0]
        # Shuffle the files
        random.shuffle(files)
        # Determine the split index
        split_idx = int(len(files) * SPLIT_SIZE)
        # Split the files into training and validation sets
        training_files = files[:split_idx]
        validation_files = files[split_idx:]
        # Copy the training files
        for file_name in training_files:
                source = os.path.join(SOURCE_DIR, file_name)
                destination = os.path.join(TRAINING_DIR, file_name)
                copyfile(source, destination)
        # Copy the validation files
        for file_name in validation_files:
                source = os.path.join(SOURCE_DIR, file_name)
                destination = os.path.join(VALIDATION_DIR, file_name)
                copyfile(source, destination)
# Test your split_data function
# Define paths
CAT_SOURCE_DIR = "/tmp/PetImages/Cat/"
DOG_SOURCE_DIR = "/tmp/PetImages/Dog/"
TRAINING_DIR = "/tmp/cats-v-dogs/training/"
VALIDATION_DIR = "/tmp/cats-v-dogs/validation/"
TRAINING_CATS_DIR = os.path.join(TRAINING_DIR, "cats/")
VALIDATION_CATS_DIR = os.path.join(VALIDATION_DIR, "cats/")
TRAINING_DOGS_DIR = os.path.join(TRAINING_DIR, "dogs/")
VALIDATION_DOGS_DIR = os.path.join(VALIDATION_DIR, "dogs/")
```

```
# Empty directories in case you run this cell multiple times
if len(os.listdir(TRAINING_CATS_DIR)) > 0:
        for file in os.scandir(TRAINING_CATS_DIR):
                os.remove(file.path)
if len(os.listdir(TRAINING_DOGS_DIR)) > 0:
        for file in os.scandir(TRAINING_DOGS_DIR):
                os.remove(file.path)
if len(os.listdir(VALIDATION_CATS_DIR)) > 0:
        for file in os.scandir(VALIDATION_CATS_DIR):
                os.remove(file.path)
if len(os.listdir(VALIDATION_DOGS_DIR)) > 0:
        for file in os.scandir(VALIDATION_DOGS_DIR):
                os.remove(file.path)
# Define proportion of images used for training
split_size = .9
# Run the function
# NOTE: Messages about zero length images should be printed out
split_data(CAT_SOURCE_DIR, TRAINING_CATS_DIR, VALIDATION_CATS_DIR, split_size)
split_data(DOG_SOURCE_DIR, TRAINING_DOGS_DIR, VALIDATION_DOGS_DIR, split_size)
# Check that the number of images matches the expected output
# Function should perform copies rather than moving images so original directories shou
print(f"\n\nOriginal cat's directory has {len(os.listdir(CAT_SOURCE_DIR))} images")
print(f"Original dog's directory has {len(os.listdir(DOG_SOURCE_DIR))} images\n")
# Training and validation splits
print(f"There are {len(os.listdir(TRAINING_CATS_DIR))} images of cats for training")
print(f"There are {len(os.listdir(TRAINING_DOGS_DIR))} images of dogs for training")
print(f"There are {len(os.listdir(VALIDATION_CATS_DIR))} images of cats for validation"
print(f"There are {len(os.listdir(VALIDATION_DOGS_DIR))} images of dogs for validation"

# -> GRADED FUNCTION: train_val_generators
def train_val_generators(TRAINING_DIR, VALIDATION_DIR):
        """
        Creates the training and validation data generators
        Args:
        TRAINING_DIR (string): directory path containing the training images
        VALIDATION_DIR (string): directory path containing the testing/validation image
        Returns:
        train_generator, validation_generator - tuple containing the generators
        """
        # Instantiate the ImageDataGenerator class
        train_datagen = ImageDataGenerator(rescale=1.0/255.)

        # Pass in the appropriate arguments to the flow_from_directory method
        train_generator = train_datagen.flow_from_directory(directory=TRAINING_DIR,
                batch_size=20,
                class_mode='binary',
                target_size=(150, 150))
        # Instantiate the ImageDataGenerator class
        validation_datagen = ImageDataGenerator(rescale=1.0/255.)
        # Pass in the appropriate arguments to the flow_from_directory method
        validation_generator = validation_datagen.flow_from_directory(directory=VALIDAT
                batch_size=20,
                class_mode='binary',
                target_size=(150, 150))
        return train_generator, validation_generator
# Test generators
train_generator, validation_generator = train_val_generators(TRAINING_DIR, VALIDATION_D

# -> GRADED FUNCTION: create_model
def create_model():
        from tensorflow.keras.optimizers import RMSprop
        model = tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(16, (3,3), activation='relu', input_shape=(150,
                tf.keras.layers.MaxPooling2D(2,2),
                tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
                tf.keras.layers.MaxPooling2D(2,2),
                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(),
                # 512 neuron hidden layer
                tf.keras.layers.Dense(512, activation='relu'),
                # Only 1 output neuron
                tf.keras.layers.Dense(1, activation='sigmoid')
        ])
        model.compile(optimizer=RMSprop(learning_rate=0.001),
                loss='binary_crossentropy',
                metrics=['accuracy'])
        return model
# Get the untrained model
model = create_model()
# Train the model
# Note that this may take some time.
history = model.fit(train_generator,
        epochs=15,
        verbose=1,
        validation_data=validation_generator)

#-----------------------------------------------------------
# Retrieve a list of list results on training and test data
# sets for each training epoch
#-----------------------------------------------------------
```

```python
acc=history.history['accuracy']
val_acc=history.history['val_accuracy']
loss=history.history['loss']
val_loss=history.history['val_loss']
epochs=range(len(acc)) # Get number of epochs
#------------------------------------------------
# Plot training and validation accuracy per epoch
#------------------------------------------------
plt.plot(epochs, acc, 'r', "Training Accuracy")
plt.plot(epochs, val_acc, 'b', "Validation Accuracy")
plt.title('Training and validation accuracy')
plt.show()
print("")
#------------------------------------------------
# Plot training and validation loss per epoch
#------------------------------------------------
plt.plot(epochs, loss, 'r', "Training Loss")
plt.plot(epochs, val_loss, 'b', "Validation Loss")
plt.show()
```

The model is **overfitting**.

2.2 Week 2: Tackle Overfitting with Data Augmentation

- **Image augmentation** is a technique used to **artificially increase the size of a training dataset** by applying various **transformations** to the original images. This helps in making the model more robust and capable of generalizing well to unseen data.
  TensorFlow provides the **ImageDataGenerator** class, which offers a wide range of image augmentation techniques. It can be used to preprocess images, augment them with various transformations, and feed them into a neural network.
  An instance of **ImageDataGenerator** can be created with desired augmentations as follows:

```python
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest'
)
```

Here's a brief description of some common parameters:

  - **rotation_range**: Degree range for random rotations.
  - **width_shift_range**, **height_shift_range**: Range for random horizontal and vertical shifts.
  - **shear_range**: Shear intensity (shear angle in counter-clockwise direction).
  - **zoom_range**: Range for random zoom.
  - **horizontal_flip**: Boolean for randomly flipping half of the images horizontally.
  - **fill_mode**: Points outside the boundaries are filled according to the given mode ('nearest', 'reflect', etc.).

- **Week assignment:**