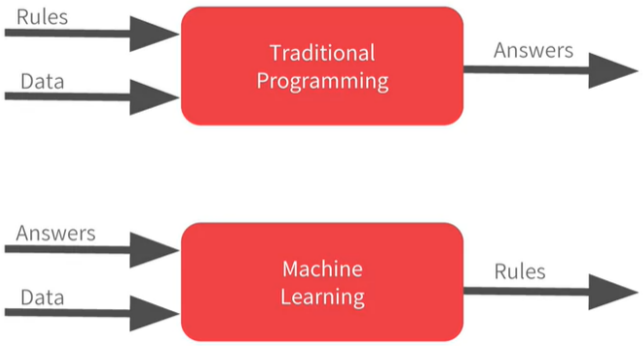
Notes from "Tensorflow for Artificial Intelligence, Machine Learning and Deep Learning" course:

Week 1:

Traditional machines were programmed to output desired answers through ad-hoc programming. The rules were established by the developers and programmers which were put into code along with the data to get consistent desired actions or answers. But, the rules need to be updated upon an encounter with anomalies in the data. Updating rules to keep the consistency is a bit challenging if we are trying to solve complex problems such as in Computer Vision. Instead, if we feed a computer with enough data that we describe (or label) as what we want it to recognize. Given that, computers are really good at processing data and finding patterns that match. Then we could potentially 'train' a system to solve a problem. The following diagram depicts the shift in the thinking paradigm.



Now, to go into depth, let's look at a very simple pipeline of Tensorflow to understand it's basic implementation. Let's go one step at a time.

```
model = keras.models.Sequential([keras.layers.Dense(units=1,
input_shape=[1])])
```

So, this line here implements a single neuron. Tensorflow uses a higher level API called keras. Layers in keras are coded as `keras.layers.Dense` to define a layer of connected neurons. Successive layers are connected in sequence and hence the name Sequential to connect all up. `units` define the number of neurons and `input_shape` defines the input to the neuron units. Here the input is a single value.

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

This line tells the model how to compile the data through a loss function of '`mean_squared_error`' using an optimizer called '`sgd`'. Math is crucial to understand which error to use in which application and which optimizers to use for which data type and application. But, we need our data. Where do we get it from? How to input this data into our model? It is quite simple, use numpy arrays. Let's take an example of a simple linear function `y = 2x-1`. We need to predict this rule through our following data implemented as a numpy array.

```
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)
```

Finally, we need to fit this data using the `model.fit` method. One epoch means the model goes through the whole data once. So, `epochs = 500` means, it goes through the data 500 times. That's huge for this simple problem isn't it ?

```
model.fit(xs,ys, epochs = 500)
print(model.predict([10.0]))
```

Important headers:

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

`print(tf.__version__)` //to check for which version you are running
//
Then the normal code from the beginning will run.

Notes:

Intro to Googlecolab: https://www.youtube.com/watch?v=inN8seMm7UI
https://research.google.com/seedbank/
http://playground.tensorflow.org - check this out for neural networks in action

Exercise: In this exercise you'll try to build a neural network that predicts the price of a house according to a simple formula. So, imagine if house pricing was as easy as a house costs 50k + 50k per bedroom, so that a 1 bedroom house costs 100k, a 2 bedroom house costs 150k etc. How would you create a neural network that learns this relationship so that it would predict a 7 bedroom house as costing close to 400k etc. Try to implement the above simple code for this problem. Think ! It's easy …:)

Week 2:

Computer vision is the hardest problem to solve, even a simple task like oh, that's a shirt and that's a pair of jeans. Humans can do this with great precision and faster than computers. Computers store images in an array of numbers and it is quite hard to manipulate them and find meaningful descriptions of simple recognition tasks of shirts and jeans. Not just this task, in general Computer Vision has been there since we started observing the cosmos. That's the best place to implement such a remarkable field of development and make our understanding of the cosmos more accurate and meaningful through the observations we gather from our telescopes.

Let's get started with gathering the data. The most widely used data used in Computer Vision is the Fashion MNIST dataset. This has become a de facto standard to understand the very early architectures of deep neural networks such as CNN. Let's see how to load that dataset for further training of our neural architecture.

For an example, we try to load this Fashion mnist dataset.

```
fashion_mnist = tf.keras.datasets.fashion_mnist
(x_train, y_train),(x_test, y_test) = fashion_mnist.load_data()
```

This dataset has already been stored as a method under the keras API to load it very easily. Real datasets will be much harder to acquire. Maximum amount of time in developing ML pipelines goes into acquiring data, approximately 70%.

Now further, let's see how we can model our own neural network. It's similar to `keras.Sequential` call but now we add more `Dense` layers to it as shown below.

```
model = keras.Sequential([keras.layers.Flatten(input_shape =[28,28]),
            keras.layers.Dense(units = 128, activation = tf.nn.relu),
            keras.layers.Dense(units= 10, activation = tf.nn.softmax])
```

There are a lot of new introductions. So, let's go through them one by one. `Flatten` is just like `Dense` and it takes a multidimensional numpy array and converts it into a single dimensional numpy array. Look carefully that the next `Dense` layers do not need an `input_size`. It is implicit that the more layers you add to the `keras.Sequential`, the output of the previous layer becomes the input of the next layer. There is an additional argument called `activation` in the `Dense` layers. Math helps us to put in the right function as the activations which go ahead through the network. `relu` is used in the hidden layers. Lastly, there is `tf.nn` which is a dependency you'll have to import. Check out Important headers in the end for the overall dependencies you'll need to train your model. E.g., do an exercise to train a neural network with roman_mnist dataset.

Following is the code to train a neural network for the fashio_mnist dataset. Isn't it just some lines of code ! Tremendously easy to do it now … all thanks to Tensorflow developers.

```
mnist = tf.keras.datasets.fashion_mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train = x_train/255.0
x_test = x_test/255.0

model = tf.keras.models.Sequential([tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(512, activation = tf.nn.relu),
        tf.keras.layers.Dense(10, activation = tf.nn.softmax)
    ])

model.compile(optimizer = 'adam',
               loss = 'sparse_categorial_crossentropy')
model.fit(x_train, y_train, epochs = 5)
```

A question that you might get at this point, in particular when experimenting with different numbers of epochs is, How can I stop training when I reach a point that I want to be at? Why do I always have to hard code it to go for a certain number of epochs? Well, the good news is that the training loop does support callbacks. So in every epoch, you can call back to a code function, having checked the metrics. If they're what you want to say, then you can cancel the training at that point. Let's take a look. Okay, so here's our code for training the neural network to recognize the fashion images. In particular, keep an eye on the model.fit function that executes the training loop. You can see that here.

What we'll now do is write a callback in Python. Here's the code. It's implemented as a separate class, but that can be in-line with your other code.

```
class myCallback(tf.keras.callbacks.Callback):
      def on_epoch_end(self, epoch, logs = {}):
            if (logs.get('loss')<0.4):
                  print("\nLoss is very low so cancelling training")
                  self.model.stop_training = True
```

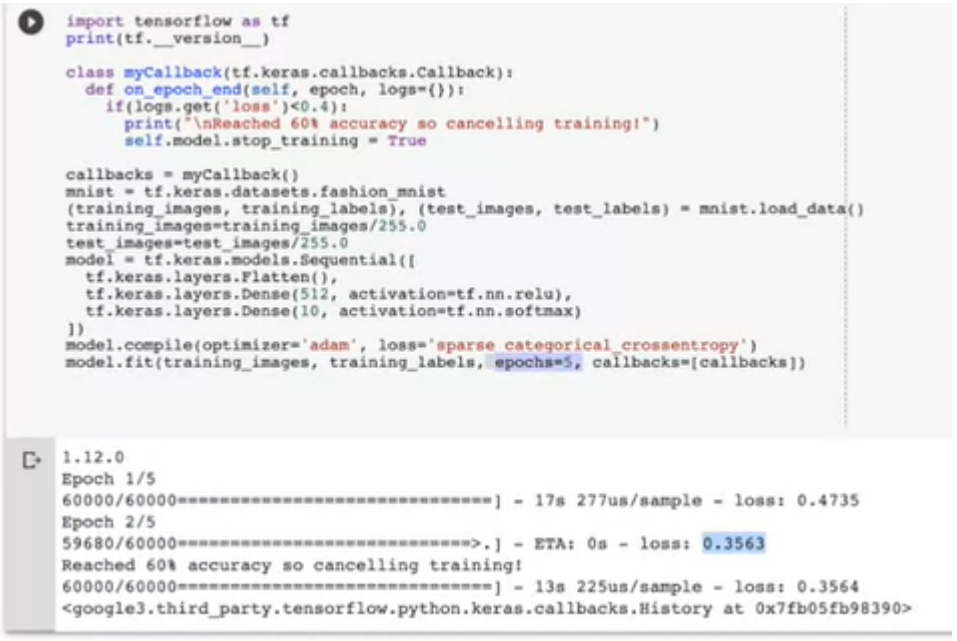
It doesn't need to be in a separate file. In it, we'll implement the on_epoch_end function, which gets called by the callback whenever the epoch ends. It also sends a logs object which contains lots of great information about the current state of training. For example, the current loss is available in the logs, so we can query it for a certain amount. For example, here I'm checking if the loss is less than 0.4 and canceling the training itself. Now that we have our callback, let's return to the rest of the code, and there are two modifications that we need to make. First, we instantiate the class that we just created, we do that with this code. Then, in my model.fit, I used the callbacks parameter and passed it to this instance of the class. Let's see this in action.

```
callbacks = myCallback()

mnist = tf.keras.datasets.fashion_mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train = x_train/255.0
x_test = x_test/255.0

model = tf.keras.models.Sequential([
            tf.keras.layers.Flatten(input_shape = (28,28)),
            tf.keras.layers.Dense(512, activation = tf.nn.relu),
            tf.keras.layers.Dense(10, activation = tf.nn.softmax)
    ])

model.compile(optimizer = 'adam',
               loss = 'sparse_categorial_crossentropy')
model.fit(x_train, y_train, epochs = 5, callbacks = [callbacks])
```

In this image you see that we defined our `myCallback` class and instantiated it before we built our model. And then as in the previous image we use the `callbacks` during the `model.fit` by specifying a `callbacks = [callbacks]` parameter. This stops the training once the error drops below 0.4. If you notice our training error is 0.3563 at the end of the second epoch. So, the callbacks are checked at the end of each epoch and not during the epoch. Isn't it cool, now you don't have to worry and keep looking at the training error. Once it drops after a certain desired threshold the training automatically drops.

```
import tensorflow as tf
class myCallback(tf.keras.callbacks.Callback):
      def on_epoch_end(self, epoch, logs = {}):
            if (logs.get('loss')<0.4):
                  print("\nLoss is very low so cancelling training")
                  self.model.stop_training = True
callbacks = myCallback()
mnist = tf.keras.datasets.fashion_mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
```

```
x_train = x_train/255.0
x_test = x_test/255.0
model = tf.keras.models.Sequential([
            tf.keras.layers.Flatten(input_shape = (28,28)),
            tf.keras.layers.Dense(512, activation = tf.nn.relu),
            tf.keras.layers.Dense(10, activation = tf.nn.softmax)
    ])
model.compile(optimizer = 'adam',
              loss = 'sparse_categorial_crossentropy')
model.fit(x_train, y_train, epochs = 5, callbacks = [callbacks])
```



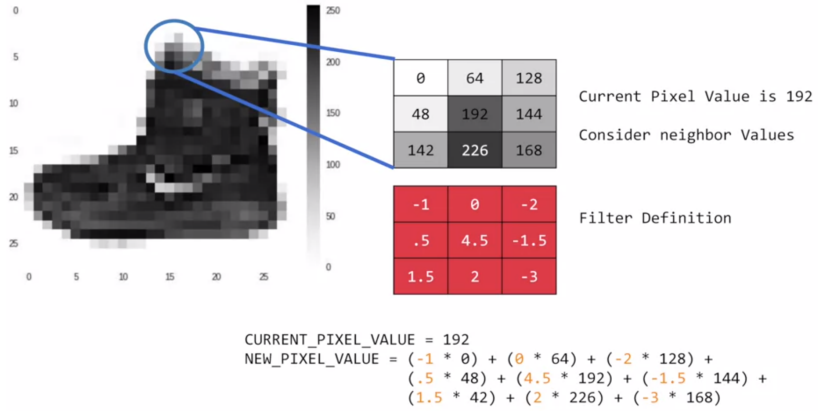
```
import tensorflow as tf
print(tf.__version__)

class myCallback(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs={}):
    if(logs.get('loss')<0.4):
      print("\nReached 60% accuracy so cancelling training!")
      self.model.stop_training = True

callbacks = myCallback()
mnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels), (test_images, test_labels) = mnist.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(512, activation=tf.nn.relu),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
model.fit(training_images, training_labels, epochs=5, callbacks=[callbacks])
```

```
1.12.0
Epoch 1/5
60000/60000==============================] - 17s 277us/sample - loss: 0.4735
Epoch 2/5
59680/60000==============================>.] - ETA: 0s - loss: 0.3563
Reached 60% accuracy so cancelling training!
60000/60000==============================] - 13s 225us/sample - loss: 0.3564
<google3.third_party.tensorflow.python.keras.callbacks.History at 0x7fb05fb98390>
```

Week 3:

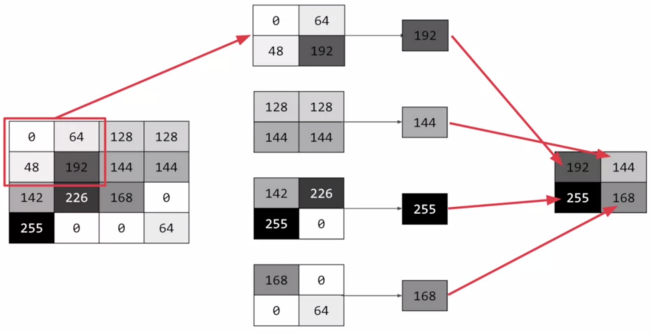
The previous method of using deep neural networks was a naive approach to check for every pixel and map it into a single class number. For e.g., a bag can be thought of having a handle and a rectangular shape in the bottom. These features are unique and can be used to distinguish much more classes and categories within classes. So, a new paradigm shift for image and video analysis is of doing convolutions in 2D space and finding meaningful features. Furthermore convolutions along with max-pooling becomes a powerful tool to capture the most enhanced features due to the convolution of different filters on the input. It's easily implemented in tensorflow -

```
tf.keras.layers.Conv2D(
      filters, kernel_size, strides=(1, 1), padding='valid',
      data_format=None, dilation_rate=(1, 1), groups=1, activation=None,
      use_bias=True, kernel_initializer='glorot_uniform',
      bias_initializer='zeros', kernel_regularizer=None,
      bias_regularizer=None, activity_regularizer=None,
      kernel_constraint=None,
      bias_constraint=None, **kwargs)
```



and

```
tf.keras.layers.MaxPool2D(
    pool_size=(2, 2), strides=None, padding='valid', data_format=None,
    **kwargs)
```



Let's see what changes we have to do with the Fashion MNIST dataset. `Conv2D` takes the image as it is in 2D array. Note the above `tf.keras.layers` methods to understand the parameters inside the method. Same goes for `MaxPool2D`.

```
model = tf.keras.models.Sequential([
            tf.keras.layers.Conv2D(64, (3,3), activations = 'relu',
                                   input_shape = (28,28,1),
            tf.keras.layers.MaxPool2D(2,2),
            tf.keras.layers.Conv2D(64, (3,3), activations = 'relu'),
            tf.keras.layers.MaxPool2D(2,2),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(512, activation = tf.nn.relu),
            tf.keras.layers.Dense(10, activation = tf.nn.softmax)
      ])
```

Furthermore a really useful method is `model.summary()` which allows you to inspect the network and get an overall understanding of how the input evolves through the CNN architecture. Here's how the output looks like.

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_12 (Conv2D)           (None, 26, 26, 64)        640
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 13, 13, 64)        0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 11, 11, 64)        36928
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 5, 5, 64)          0
_________________________________________________________________
flatten_5 (Flatten)          (None, 1600)              0
_________________________________________________________________
dense_10 (Dense)             (None, 128)               204928
_________________________________________________________________
dense_11 (Dense)             (None, 10)                1290
=================================================================
```

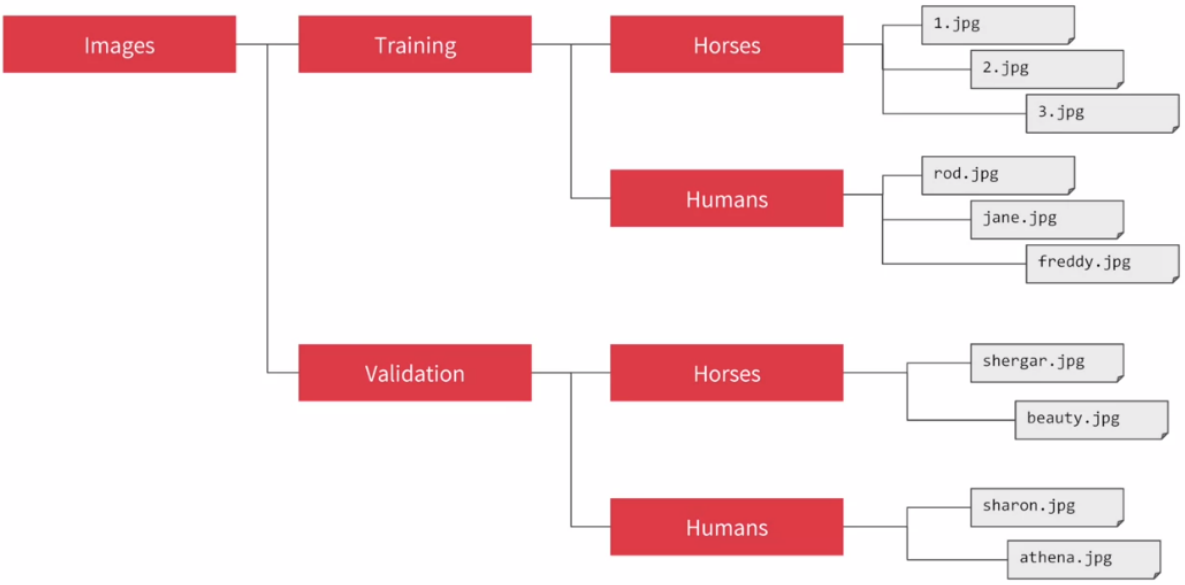
If you see it carefully, notice the output after the first convolution `(26,26,64)`. It is mainly because convolution reduces the size of the input. So, we need to add padding to keep the output to be the same after convolution. How do we do padding ? - Check that out, it's inside the parameter list of `Conv2D`. You've now seen how to turn your Deep Neural Network into a Convolutional Neural Network by adding convolutional layers on top, and having the network train against the results of the convolutions instead of the raw pixels.

Now try to use the visualization notebook where you can see the output after each convolutions trained on the Fashion MNIST dataset. Try tweaking the code with these suggestions, editing the convolutions, removing the final convolution, and adding more, etc. Here's the link to the Notebook: https://github.com/https-deeplearning-ai/tensorflow-1-public/blob/main/C1/W3/ungraded_labs/C1_W3_Lab_1_improving_accuracy_using_convolutions.ipynb

Next try and understand what different filters achieve by going through this Notebook link and a link to commonly used filters in computer graphics.
https://github.com/https-deeplearning-ai/tensorflow-1-public/blob/main/C1/W3/ungraded_labs/C1_W3_Lab_2_exploring_convolutions.ipynb
https://lodev.org/cgtutor/filtering.html

Week 4:

It is good that we were able to improve the performance of the Fashion MNIST dataset by using convolutional neural networks. But the real world data has a lot of variance i.e., they are not 28 x 28 pixel valued images. So, we need something called an image generator that will sort the input images into its desired size and separate them into training and validation sets along with their labels. But this is done during run time and you do not change the input at all. The only thing to take care here is we need two directories - one with training data and the other with validation data. The individual directories should have respective label directories in which the images are stored. E.g., if you are working with handwritten MNIST dataset, you'll have two directories as mentioned before, one for training and the other for validation. Each directory will have 0 to 9 directories in which the respective images are stored.



Once you have this in place, all you need is an API from keras and a few more lines of code,

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(rescale = 1.0/255)

train_generator = train_datagen.flow_from_directory(
                        train_dir,
                        target_size = (300,300),
                        batch_size = 128,
                        class_mode = 'binary')

test_datagen = ImageDataGenerator(rescale = 1.0/255)

validation_generator = test_datagen.flow_from_directory(
                        validation_dir,
                        target_size = (300,300),
                        batch_size = 32, #there are ways to find an optimal
                        class_mode = 'binary') #multi version also exists
```

Now that you've designed the neural network to classify Horses or Humans, the next step is to train it from data that's on the file system, which can be read by generators. To do this, you don't use `model.fit` as earlier, but a new method call: `model.fit_generator`. Why do we do this? The

reason I can make out is because we have used the ImageGenerator and that's why we need a separate method to train our generated images.

Here's a simple example of a binary classification. Before we use model.fit_generator we have to as usual compile the model first.

```
from tensorflow.keras.optimizers import RMSprop

model.compile(loss = 'binary_crossentropy',
              Optimizer = RMSprop(lr = 0.001),
              Metrics = ['acc'])

history = model.fit_generator(
                  train_generator,
                  steps_per_epoch = 8,
                  epochs = 15,
                  validation_data = validation_generator,
                  validation_steps = 8,
                  verbose = 2)
```

`train_generator =` This streams the images from the training directory. Remember the batch size you used when you created it, it was 128, that's important in the next step.

`steps_per_epoch` = Since there are 1024 training images to go through in batches of 128 images. So, this needs to be calculated as dividing the total number of images in the training directory by the number of batches in the train_generator.

`epochs` = as usual the same through the whole training examples

`validation_date` = input the `validation_generator` for testing

`validation_steps` = Similar to steps_per_epoch but for the test set.

verbose =

Now you've gone through the code to define the neural network, train it with on-disk images, and then predict values for new images.

Main Pipeline:

1. Get your data into a local or a temp directory of google colab. (`import os`)
2. Visualize a few images to validate that they are correctly loaded. (`import matplotlib.plt` and `matplotlib.image`)
3. Build a small `tf.keras.model.Sequential()` model from scratch
4. Check the summary to understand your model and parameters to train,=
5. Compile the model using `model.compile`()
6. Create `ImageDataGenerator` for both training and validation data
7. Finally `model.fit_generator`()

8. Test on unseen images to predict from anywhere from the web or other online datasets where there is no overlap with the training and validation set upon which the model is trained. (E.g. pixabay)

This colab notebook depicts the whole pipeline of build a ConvNet for Image processing and this contains only ImageDataGenerator for training set: https://github.com/https-deeplearning-ai/tensorflow-1-public/blob/main/C1/W4/ungraded_labs/C1_W4_Lab_1_image_generator_no_validation.ipynb

And this one with ImageDataGenerator for both training and validation directories:https://github.com/https-deeplearning-ai/tensorflow-1-public/blob/main/C1/W4/ungraded_labs/C1_W4_Lab_2_image_generator_with_validation.ipynb

Last thing to look for is the scaling of nxn pixel images: This is a great example of the importance for measuring your training data against a large validation set, inspecting where it goes wrong and seeing what you can do to fix it. Using this smaller set is much cheaper to train, but then errors like a woman with her back turned and her legs obscured by the dress will be falsely classified, because we don't have that data in the training set. That's a nice hint about how to edit your dataset for the best effect in training.
https://github.com/https-deeplearning-ai/tensorflow-1-public/blob/main/C1/W4/ungraded_labs/C1_W4_Lab_3_compacted_images.ipynb