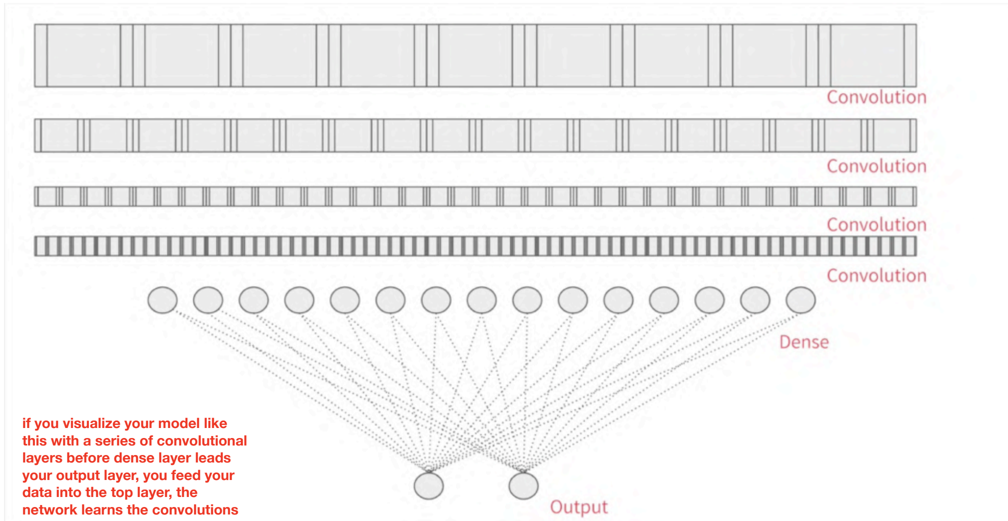
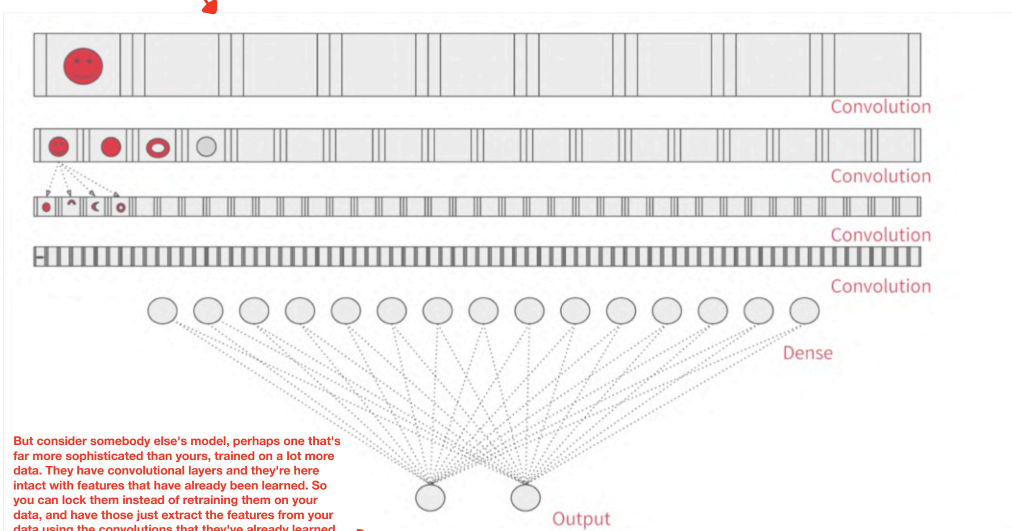
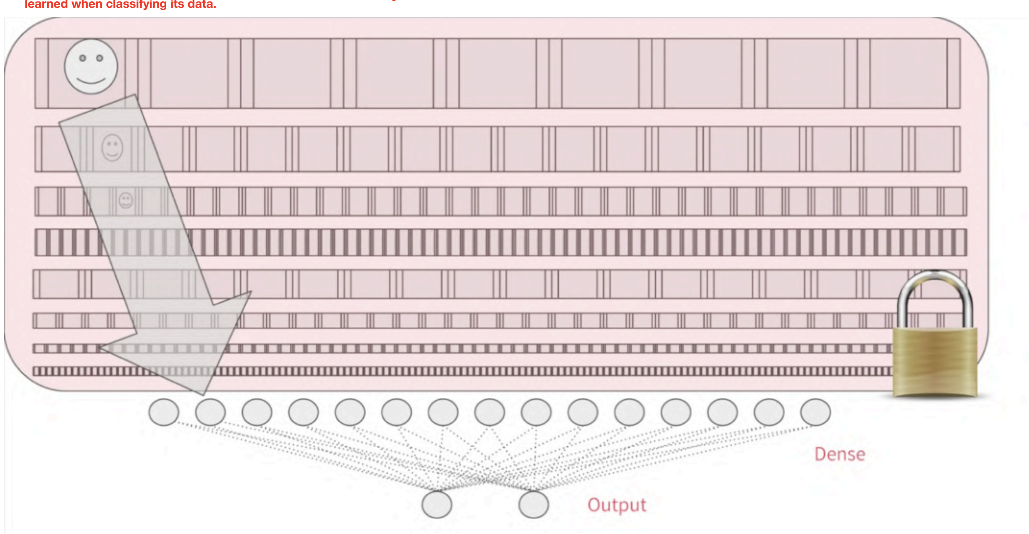


if you visualize your model like this with a series of convolutional layers before dense layer leads your output layer, you feed your data into the top layer, the network learns the convolutions that identify the features in your data and all that.



But consider somebody else's model, perhaps one that's far more sophisticated than yours, trained on a lot more data. They have convolutional layers and they're here intact with features that have already been learned. So you can lock them instead of retraining them on your data, and have those just extract the features from your data using the convolutions that they've already learned. Then you can take a model that has been trained on a very large datasets and use the convolutions that it learned when classifying its data.



Now that we've seen the concepts behind transfer learning, let's dig in and take a look at how to do it for ourselves with TensorFlow and Keras.

In the next few videos you'll be using this notebook to explore transfer learning: https://colab.research.google.com/github/lmoroney/dlaicourse/blob/master/Course%202%20-%20Part%206%20-%20Lesson%203%20-%20Notebook.ipynb

For more on how to freeze/lock layers, explore the documentation, which includes an example using MobileNet architecture: https://www.tensorflow.org/tutorials/images/transfer_learning

```
import os

from tensorflow.keras import layers
from tensorflow.keras import Model
```

Transfer Learning

```
https://storage.googleapis.com/mledu-datasets/
 inception_v3_weights_tf_dim_ordering_tf_kernels
```

A copy of the pretrained weights for the inception neural network is saved at this URL. Think of this as a snapshot of the model after being trained. It's the parameters that can then get loaded into the skeleton of the model, to turn it back into a trained model.

```python
from tensorflow.keras.applications.inception_v3 import InceptionV3

local_weights_file = '/tmp/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5'

pre_trained_model = InceptionV3(input_shape = (150, 150, 3),
                                include_top = False,
                                weights = None)

pre_trained_model.load_weights(local_weights_file)
```

So now if we want to use inception, it's fortunate that keras has the model definition built in. So you instantiate that with the desired input shape for your data, and specify that you don't want to use the built-in weights, but the snapshot that you've just downloaded. The inception V3 has a fully-connected layer at the top. So by setting include_top to false, you're specifying that you want to ignore this and get straight to the convolutions.

```python
for layer in pre_trained_model.layers:
  layer.trainable = False
```

Now that I have my pretrained model instantiated, I can iterate through its layers and lock them, saying that they're not going to be trainable with this code.

```python
pre_trained_model.summary()
```

Adding your DNN

In the previous video you saw how to take the layers from an existing model, and make them so that they don't get retrained -- i.e. you freeze (or lock) the already learned convolutions into your model. Now, you'll need to add your own DNN at the bottom of these, which you can retrain to your data. In the next video you'll see how to do that...

```
last_layer = pre_trained_model.get_layer('mixed7')

last_output = last_layer.output
```

All of the layers have names, so you can look up the name of the last layer that you want to use. If you inspect the summary, you'll see that the bottom layers have convolved to 3 by 3. But I want to use something with a little more information. So I moved up the model description to find mixed7, which is the output of a lot of convolution that are 7 by 7.

```
from tensorflow.keras.optimizers import RMSprop

x = layers.Flatten()(last_output)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dense  (1, activation='sigmoid')(x)

model = Model( pre_trained_model.input, x)
model.compile(optimizer = RMSprop(lr=0.0001),
              loss = 'binary_crossentropy',
              metrics = ['acc'])
```

we'll define our new model, taking the output from the inception model's mixed7 layer. You start by flattening the input, which just happens to be the output from inception

```
from tensorflow.keras.optimizers import RMSprop

x = layers.Flatten()(last_output)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dense  (1, activation='sigmoid')(x)

model = Model( pre_trained_model.input, x)
model.compile(optimizer = RMSprop(lr=0.0001),
              loss = 'binary_crossentropy',
              metrics = ['acc'])
```

```
from tensorflow.keras.optimizers import RMSprop

x = layers.Flatten()(last_output)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dense  (1, activation='sigmoid')(x)

model = Model( pre_trained_model.input, x)
model.compile(optimizer = RMSprop(lr=0.0001),
              loss = 'binary_crossentropy',
              metrics = ['acc'])
```

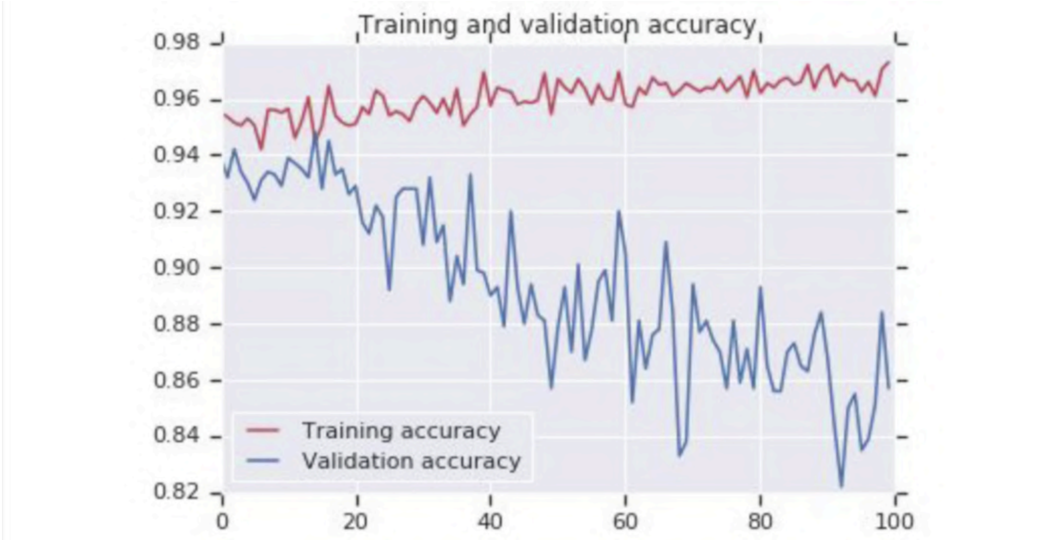
You can then create a model using the Model abstract class. And passing at the input and the layers definition that you've just created.

```
from tensorflow.keras.optimizers import RMSprop

x = layers.Flatten()(last_output)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dense  (1, activation='sigmoid')(x)

model = Model( pre_trained_model.input, x)
model.compile(optimizer = RMSprop(lr=0.0001),
              loss = 'binary_crossentropy',
              metrics = ['acc'])
```

```python
# Add our data-augmentation parameters to 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)
```

```python
train_generator = train_datagen.flow_from_directory(
                    train_dir,
                    batch_size = 20,
                    class_mode = 'binary',
                    target_size = (150, 150))
```

```python
history = model.fit_generator(
            train_generator,
            validation_data = validation_generator,
            steps_per_epoch = 100,
            epochs = 100,
            validation_steps = 50,
            verbose = 2)
```

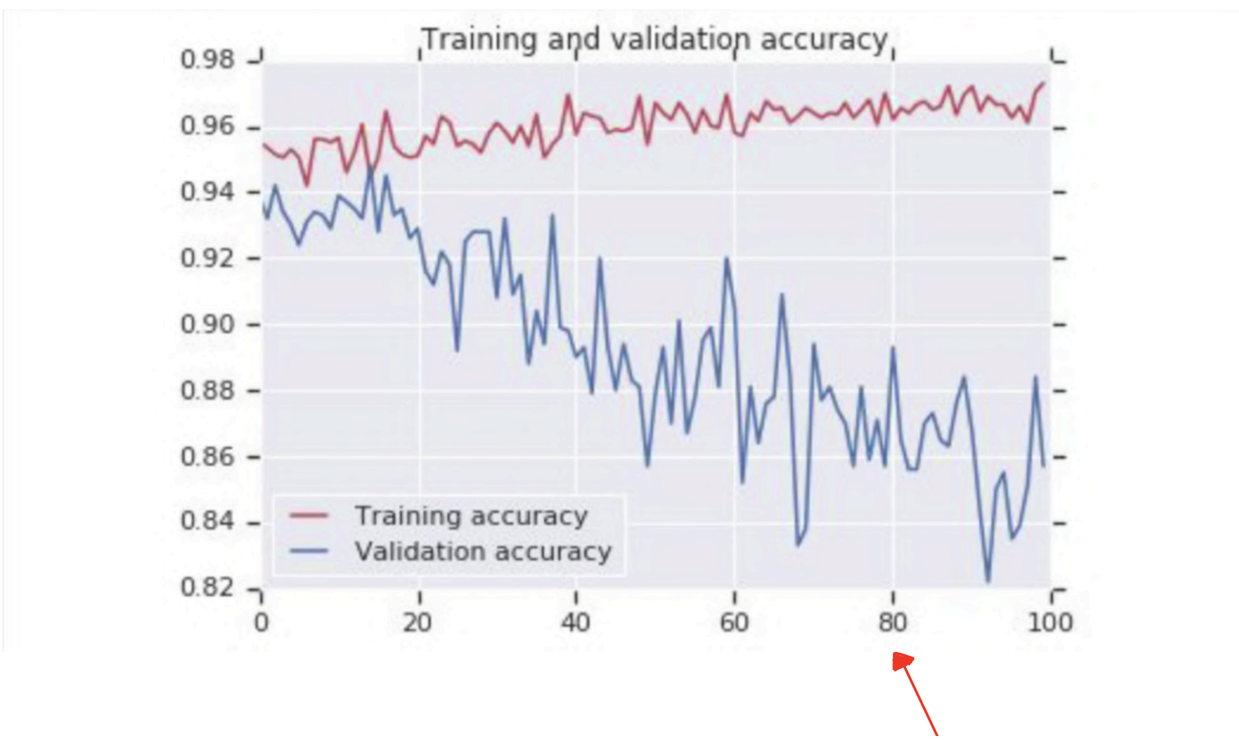


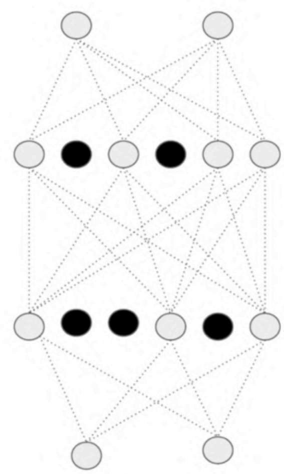
Using dropouts!

Another useful tool to explore at this point is the Dropout.

The idea behind Dropouts is that they remove a random number of neurons in your neural network. This works very well for two reasons: The first is that neighboring neurons often end up with similar weights, which can lead to overfitting, so dropping some out at random can remove this. The second is that often a neuron can over-weigh the input from a neuron in the previous layer, and can over specialize as a result. Thus, dropping out can break the neural network out of this potential bad habit!

Check out Andrew's terrific video explaining dropouts here: https://www.youtube.com/watch?v=ARq74QuavAo



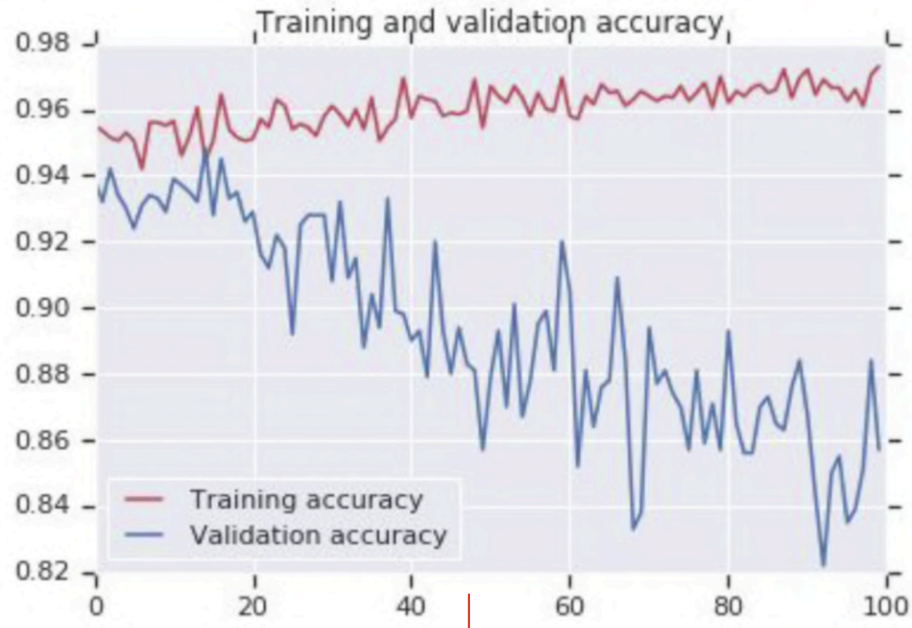


What's interesting if you do this, is that you end up with another but a different overfitting situation. Here is the graph of the accuracy of training versus validation. As you can see, while it started out well, the validation is diverging away from the training in a really bad way. So, how do we fix this?

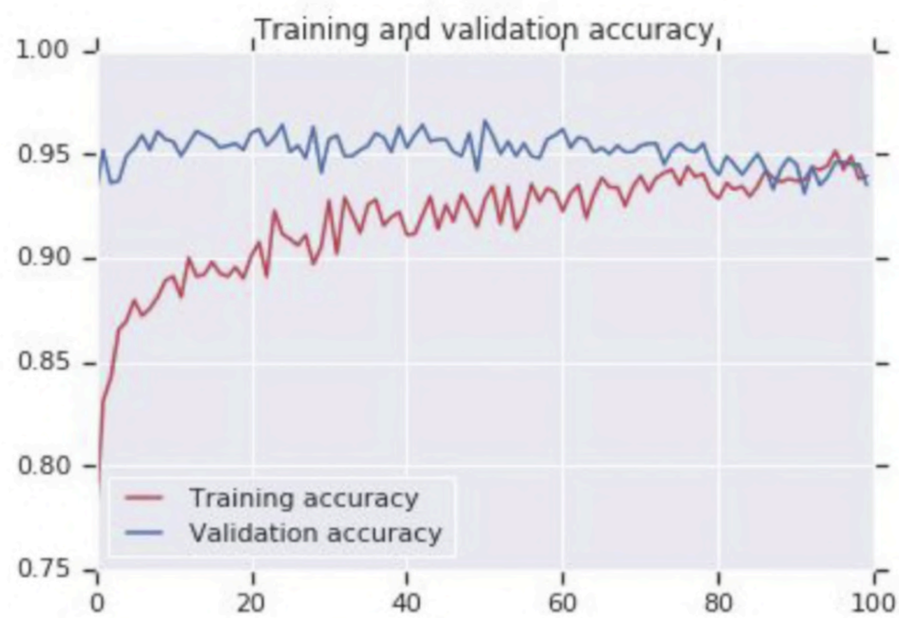
```
from tensorflow.keras.optimizers import RMSprop

x = layers.Flatten()(last_output)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dense  (1, activation='sigmoid')(x)

model = Model( pre_trained_model.input, x)
model.compile(optimizer = RMSprop(lr=0.0001),
              loss = 'binary_crossentropy',
              metrics = ['acc'])
```

```
from tensorflow.keras.optimizers import RMSprop

x = layers.Flatten()(last_output)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dropout(0.2)(x)
x = layers.Dense  (1, activation='sigmoid')(x)

model = Model( pre_trained_model.input, x)
model.compile(optimizer = RMSprop(lr=0.0001),
              loss = 'binary_crossentropy',
              metrics = ['acc'])
```

The parameter is between 0 and 1 and it's the fraction of units to drop. In this case, we're dropping out 20% of our neurons.


Training and validation accuracy
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0.84
0.82
0
20
40
60
80
100
Training accuracy
Validation accuracy

Dropout


Training and validation accuracy
1.00
0.95
0.90
0.85
0.80
0.75
0
20
40
60
80
100
Training accuracy
Validation accuracy