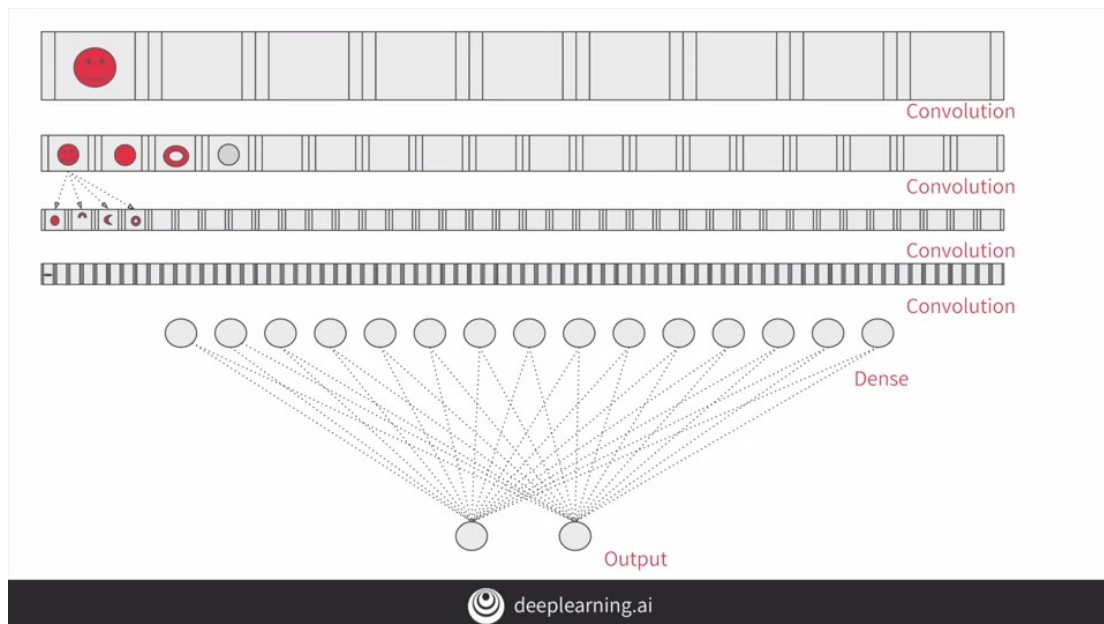
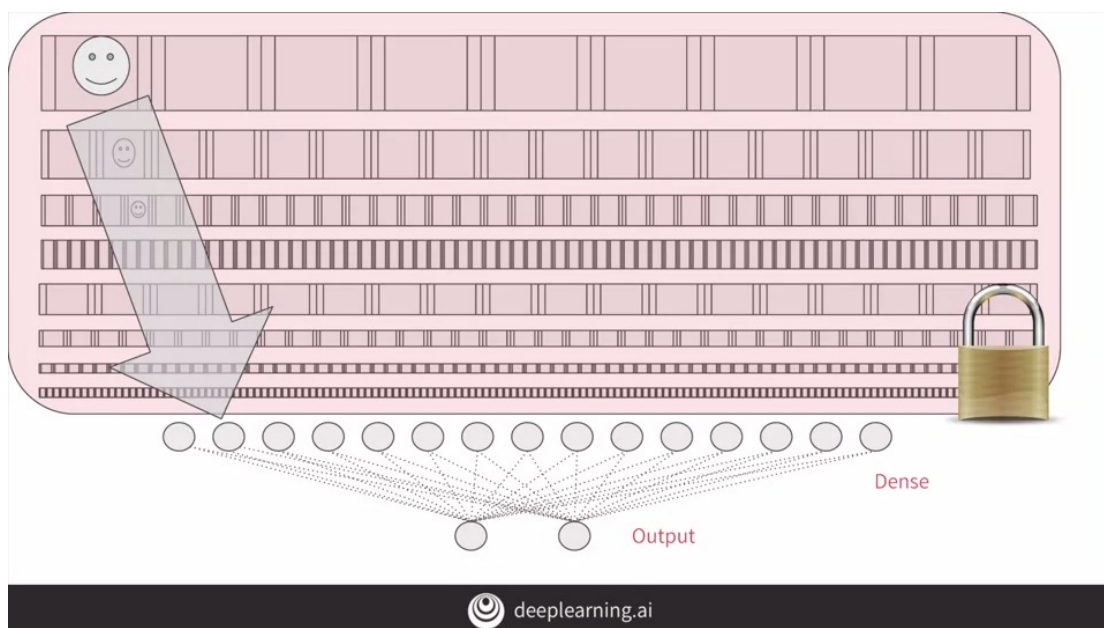


3.1 Understanding transfer learning: the concepts

What if you could take an existing model that's trained on far more data, and use the features that that model learned? That's the concept of transfer learning



So for example, 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. If you recall how convolutions are created and used to identify particular features, and the journey of a feature through the network, it makes sense to just use those, and then retrain the dense layers from that model with your data.

Of course, well, it's typical that you might lock all the convolutions. You don't have to. You can choose to retrain some of the lower ones too because they may be too specialized for the images at hand. It takes some trial and error to discover the right combination.

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.

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

3.2 Coding transfer learning from the inception mode

```
import os

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

We'll start with the inputs. In particular, we'll be using the keras layers API, to pick at the layers, and to understand which ones we want to use, and which ones we want to retrain.

```
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.

```
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.

```
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. You can then print a summary of your pretrained model with this code but be prepared, it's huge.

3.3 Coding your own model with transferred features

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.

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

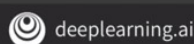
last_output = last_layer.output
```

But with this code, I'm going to grab that layer from inception and take it to output. So now we'll define our new model, taking the output from the inception model's mixed7 layer, which we had called **last_output**. This should look exactly like the dense models that you created way back at the start of this course.

```
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'])
```



The code is a little different, but this is just a different way of using the layers API.

You start by **flattening the input**, which just happens to be the output from inception.

And then **add a Dense hidden layer**.

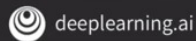
And then your output layer which has just one neuron **activated by a sigmoid** to classify

between two items.

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.

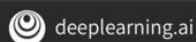
And then you **compile** it as before with an optimizer and a loss function and the metrics that you want to collect.

```
# 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)
```



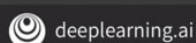
But as before you're going to augment the images with the image generator.

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

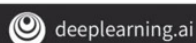
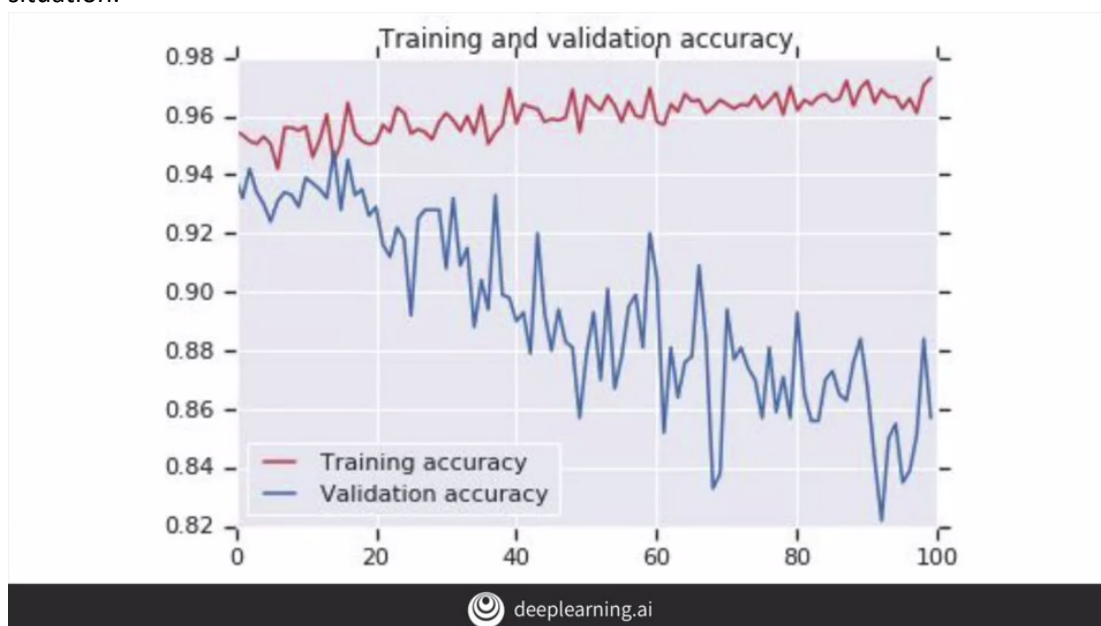


Then, as before, we can get our training data from the generator by flowing from the specified directory and going through all the augmentations.

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



And now we can train as before with `model.fit_generator`. I'm going to run it for 100 epochs. 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? We'll take a look at that in the next lesson.

3.4 Exploring dropouts

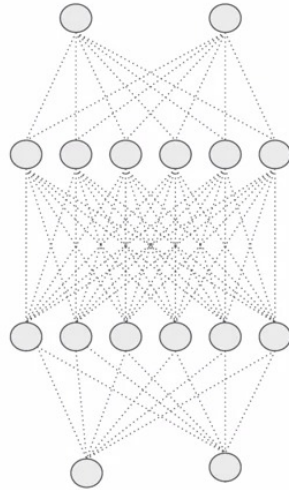
Another useful tool to explore at this point is the **Dropout**.


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**

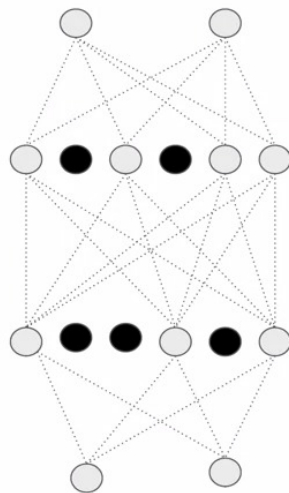
a neuron in the previous layer, and can over specialize as a result. Thus, dropping out can break the neural network out of this potential bad habit!


Check out Andrew's terrific video explaining dropouts

here: <https://www.youtube.com/watch?v=ARq74QuavAo>



 deeplearning.ai



 deeplearning.ai

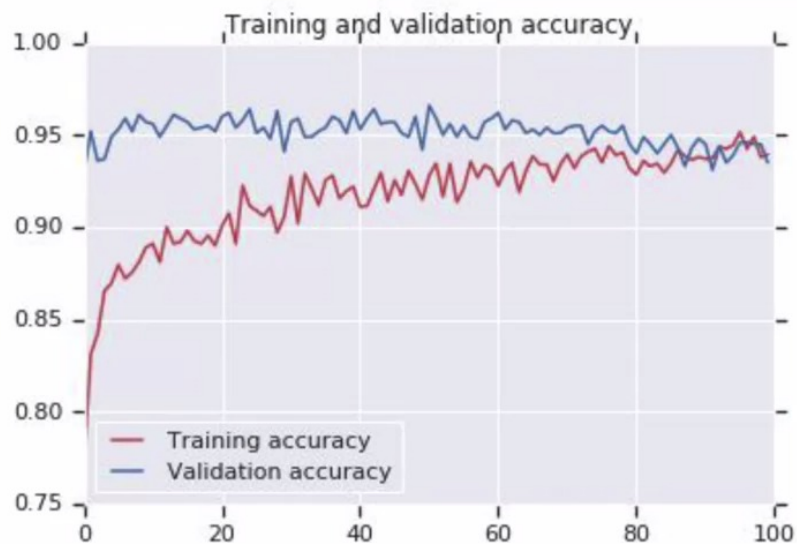
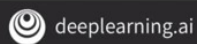
```

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'])

```



3.5 Exploring Transfer Learning with Inception

Transfer Learning.ipynb