

Keras Functional API

Tensors: Mathematics vs TensorFlow

Different notions of tensors:

- Mathematical: N-D grid of values.
 - 1D: vector
 - 2D: matrix
- TensorFlow:
 - Datastructure that can produce a mathematical tensor when “called”
 - Internals: tensor shape + a reference to the object that can produce the value
 - Forms the basis of the data-flow graph that represents the computation that your network performs

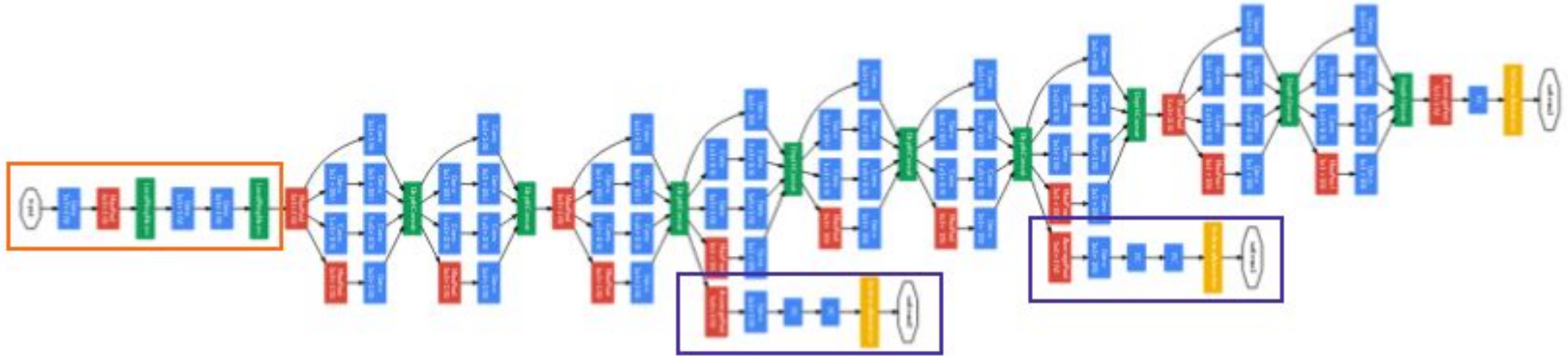
Data Flow Graphs

- Node: performs some computation or stores information
 - Inputs are TF tensors that link to the source of the inputs
 - Output(s) can be referenced by other nodes (as TF tensors)
- Some nodes take input from outside the graph (e.g., the input/output pairs used for training)
- Some nodes only provide output to outside the graph (e.g., an output layer)

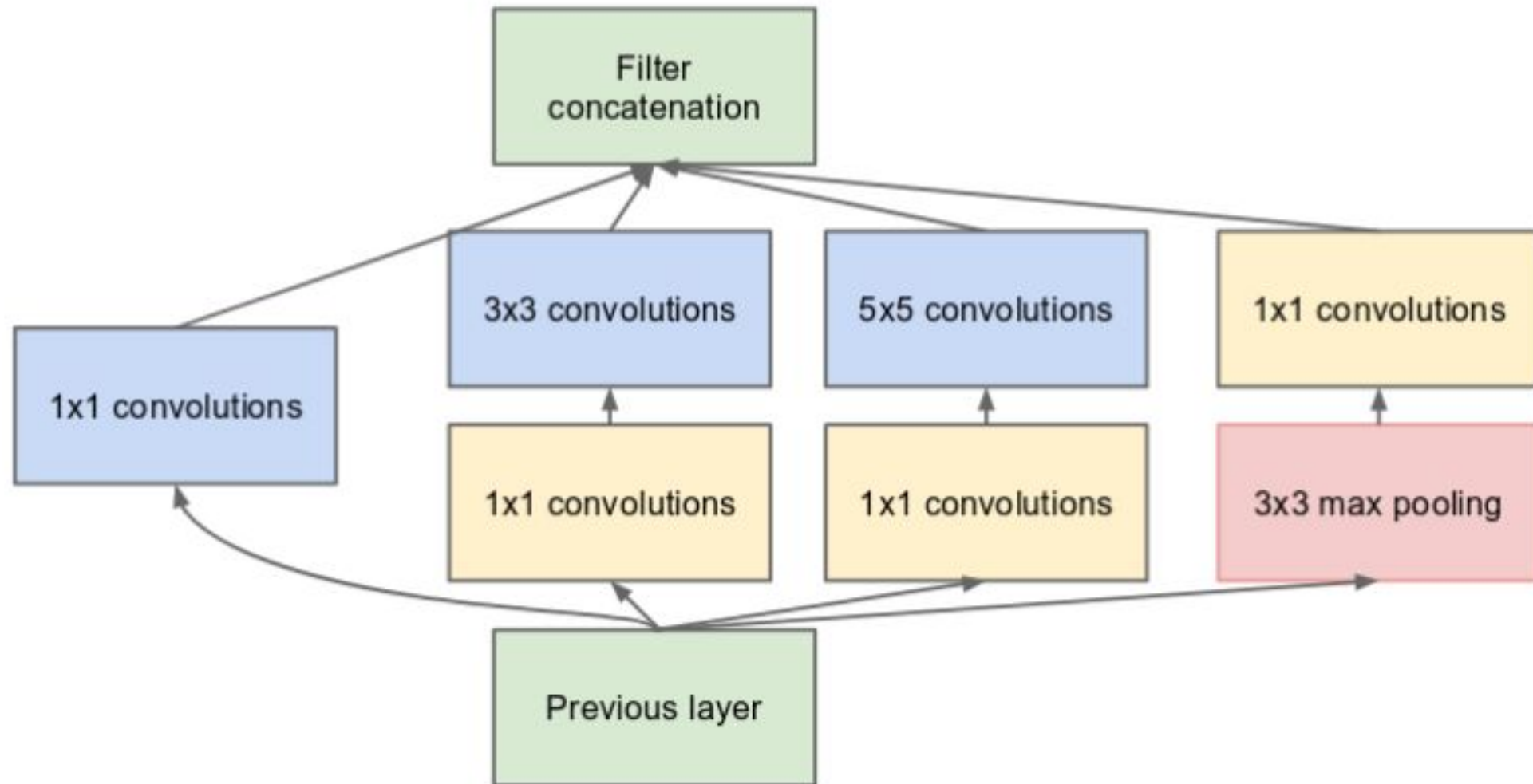
Programming with TensorFlow Graphs

- Sequential class handles a lot of the details implicitly
 - Only allows us to specify particular types of networks
 - We must anticipate the correct order of the operations
- The Keras Model API to build our own networks with somewhat arbitrary topologies

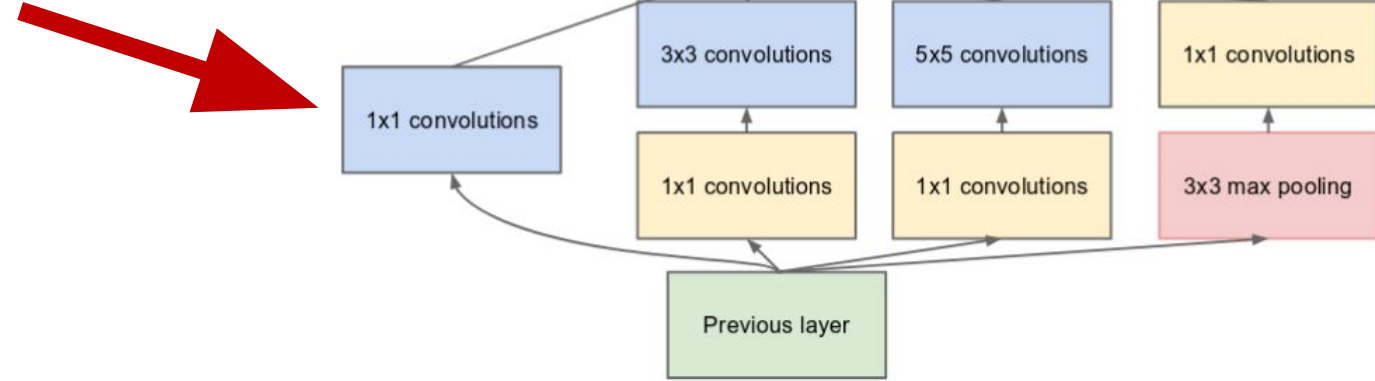
Example: Very Deep Networks (Inception)



Inception Module



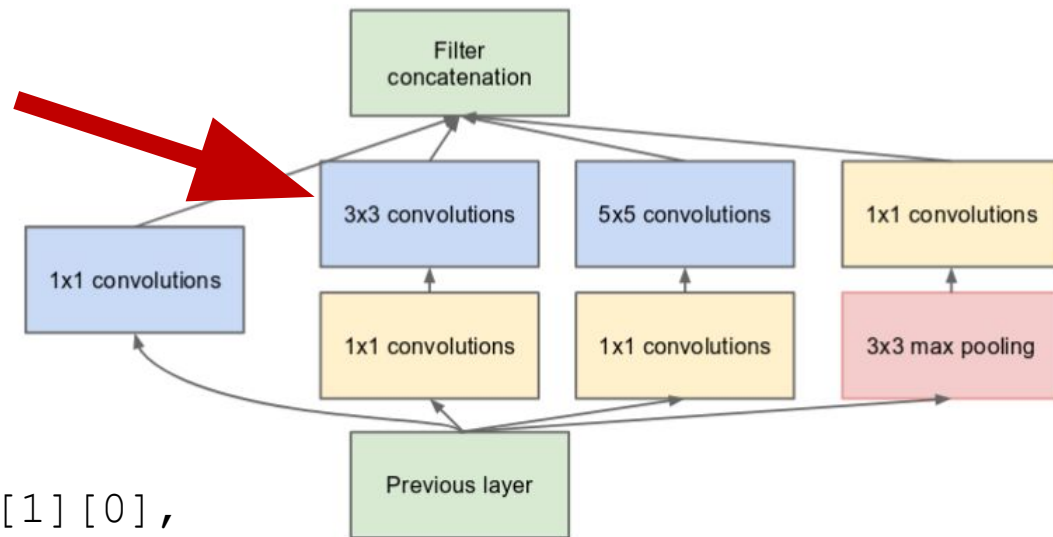
Branch A



```
def inception_module(input_tensor, nfilters, activation,
                     lambda_regularization, name):
```

```
    convA_tensor = Convolution2D(filters=nfilters[0],
                                kernel_size=(1,1),
                                strides=(2,2),
                                padding='same',
                                name = 'convA_'+name,
                                ...) (input_tensor)
```

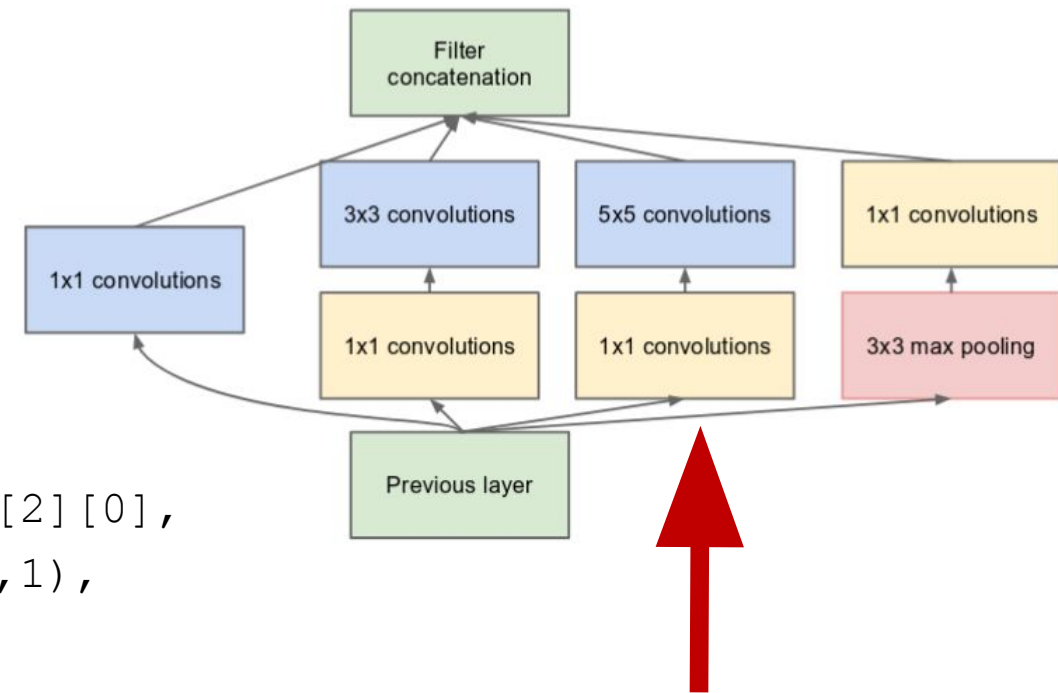
Branch B



```
convB0_tensor = Convolution2D(filters=nfilters[1][0],  
                               kernel_size=(1,1),  
                               strides=(1,1),  
                               padding='same',  
                               name = 'convB0_'+name,  
                               ... )(input_tensor)
```

```
convB1_tensor = Convolution2D(filters=nfilters[1][1],  
                               kernel_size=(3,3),  
                               strides=(2,2),  
                               padding='same',  
                               name = 'convB1_'+name,  
                               activation=activation,  
                               ... )(convB0_tensor)
```

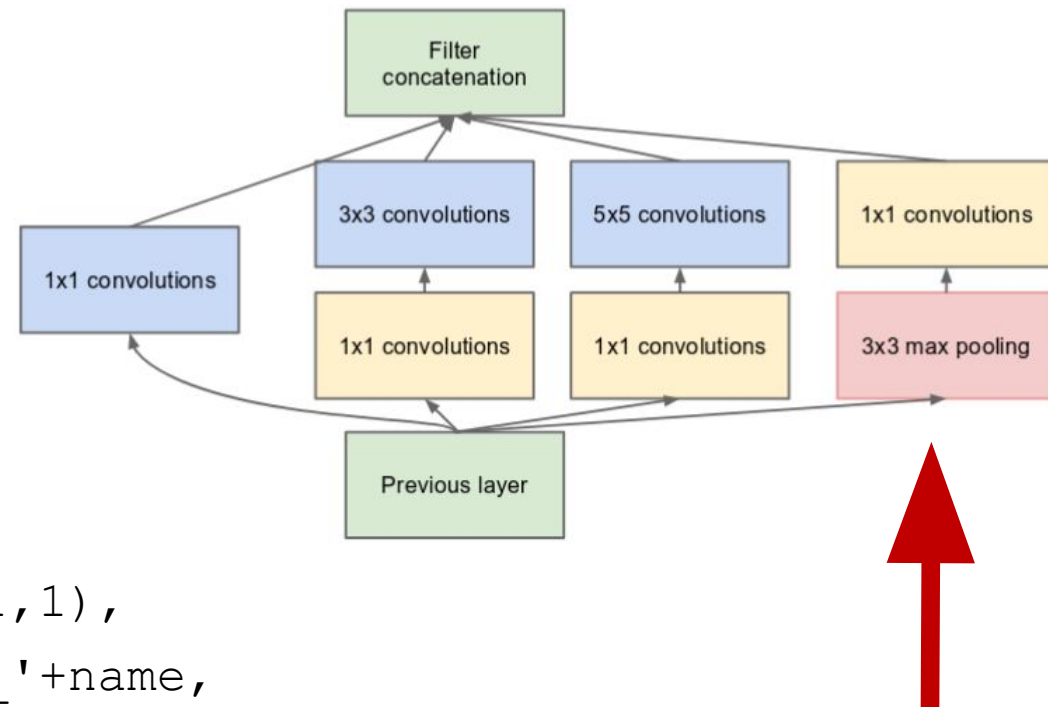

Branch C



```
convC0_tensor = Convolution2D(filters=nfilters[2][0],  
                               kernel_size=(1,1),  
                               strides=(1,1),  
                               padding='same',  
                               name = 'convC0_'+name,  
                               ... )(input_tensor)
```

```
convC1_tensor = Convolution2D(filters=nfilters[2][1],  
                               kernel_size=(5,5),  
                               strides=(2,2),  
                               padding='same',  
                               name = 'convC1_'+name,  
                               activation=activation,  
                               ... )(convC0_tensor)
```

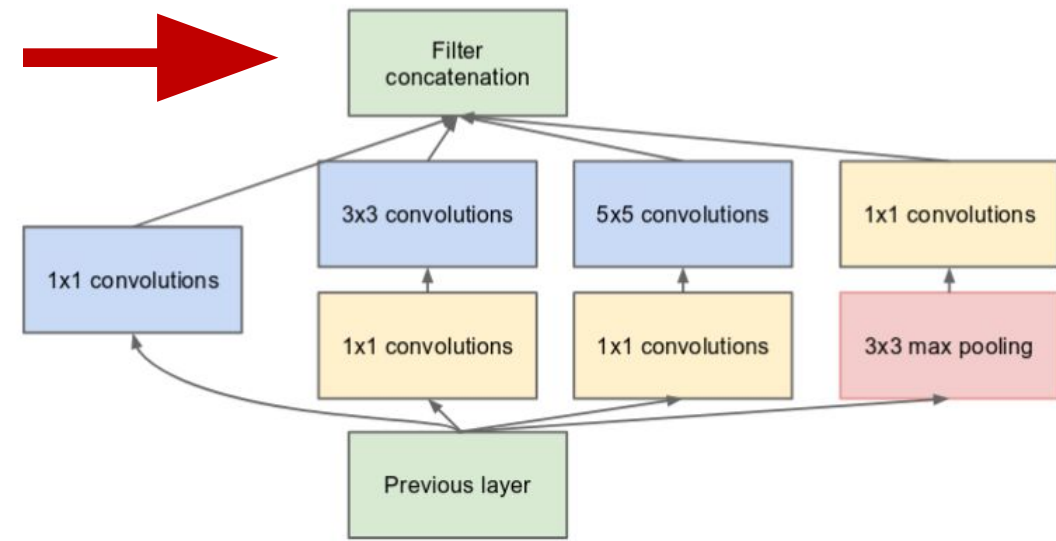
Branch D



```
max_tensor = MaxPooling2D(pool_size=(3,3),  
                           strides=(1,1),  
                           name='MAX_'+name,  
                           padding='same')(input_tensor)
```

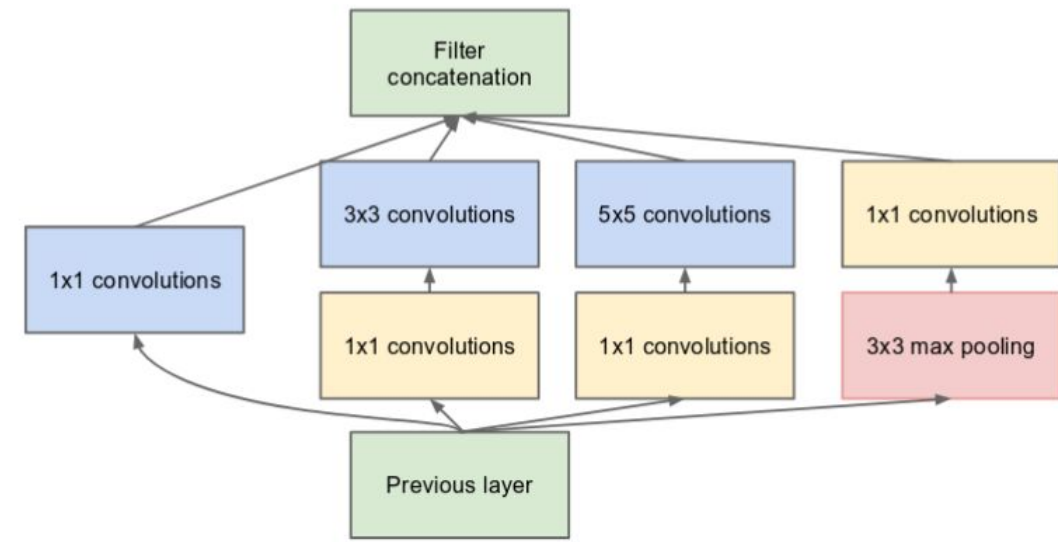
```
convD1_tensor = Convolution2D(filters=nfilters[3],  
                              kernel_size=(1,1),  
                              strides=(2,2),  
                              padding='same',  
                              name = 'convD0_'+name,  
                              activation=activation,  
                              ... )(max_tensor)
```

Concatenation



```
output_tensor = Concatenate()  
    ([convA_tensor, convB1_tensor, convC1_tensor, convD1_tensor])  
  
return output_tensor
```

Building an Image Classifier



```
def create_inception_network(image_size, n_channels,
                             lambda_regularization, activation='elu'):

    input_tensor = Input(shape=(image_size[0], image_size[1], n_channels), name="input")

    i1_tensor = inception_module(input_tensor, (10, (10,10)), (10,10), 10), activation,
                                lambda_regularization, name="i1")

    i2_tensor = inception_module(i1_tensor, (40, (40,40)), (40,40), 40), activation,
                                lambda_regularization, name="i2")

    flatten_tensor = Flatten()(i2_tensor)
```

Building an Image Classifier II

```
dense1_tensor = Dense(units=100, activation=activation, name = "D1", ... ) (flatten_tensor)
dense2_tensor = Dense(units=20, activation=activation, name = "D2", ... ) (dense1_tensor)
output_tensor = Dense(units=1, activation='sigmoid', name = "output", ... ) (dense2_tensor)
```

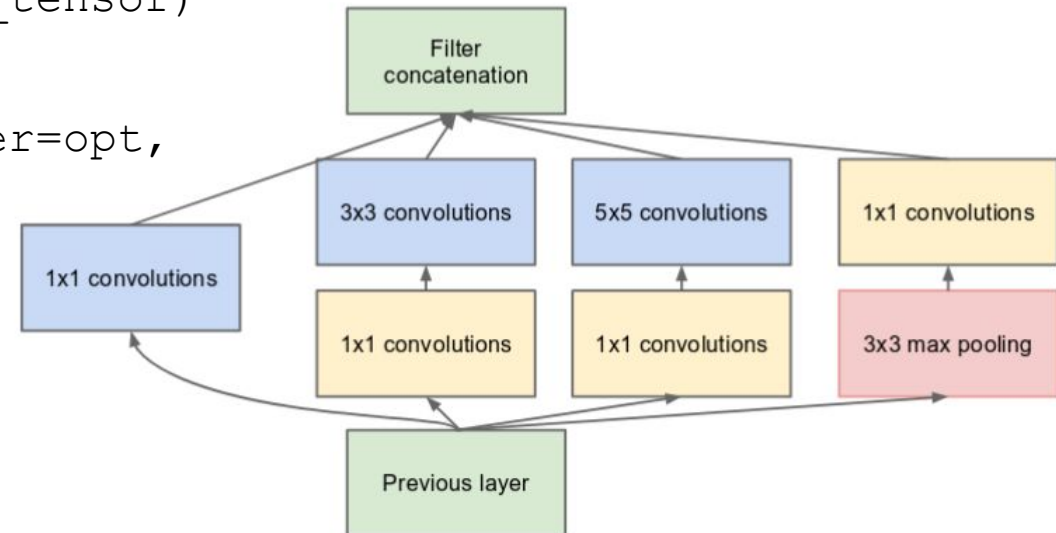
```
opt = keras.optimizers.Adam(lr=0.0001, beta_1=0.9, beta_2=0.999,
                             epsilon=None, decay=0.0, amsgrad=False)
```

```
model = Model(inputs=input_tensor, outputs=output_tensor)
```

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

```
print(model.summary())
```

```
return model
```



Layer (type)	Output Shape	Param #	Connected to
=====			
input (InputLayer)	(None, 32, 32, 3)	0	
convB0_i1 (Conv2D)	(None, 32, 32, 10)	40	input[0][0]
convC0_i1 (Conv2D)	(None, 32, 32, 10)	40	input[0][0]
MAX_i1 (MaxPooling2D)	(None, 32, 32, 3)	0	input[0][0]
convA_i1 (Conv2D)	(None, 16, 16, 10)	40	input[0][0]
convB1_i1 (Conv2D)	(None, 16, 16, 10)	910	convB0_i1[0][0]
convC1_i1 (Conv2D)	(None, 16, 16, 10)	2510	convC0_i1[0][0]
convD0_i1 (Conv2D)	(None, 16, 16, 10)	40	MAX_i1[0][0]
concatenate_14 (Concatenate)	(None, 16, 16, 40)	0	convA_i1[0][0] convB1_i1[0][0] convC1_i1[0][0] convD0_i1[0][0]

Total params: 1,090,761

convB0_i2 (Conv2D)	(None, 16, 16, 40)	1640	concatenate_14[0][0]
convC0_i2 (Conv2D)	(None, 16, 16, 40)	1640	concatenate_14[0][0]
MAX_i2 (MaxPooling2D)	(None, 16, 16, 40)	0	concatenate_14[0][0]
convA_i2 (Conv2D)	(None, 8, 8, 40)	1640	concatenate_14[0][0]
convB1_i2 (Conv2D)	(None, 8, 8, 40)	14440	convB0_i2[0][0]
convC1_i2 (Conv2D)	(None, 8, 8, 40)	40040	convC0_i2[0][0]
convD0_i2 (Conv2D)	(None, 8, 8, 40)	1640	MAX_i2[0][0]
concatenate_15 (Concatenate)	(None, 8, 8, 160)	0	convA_i2[0][0] convB1_i2[0][0] convC1_i2[0][0] convD0_i2[0][0]
flatten_7 (Flatten)	(None, 10240)	0	concatenate_15[0][0]
D1 (Dense)	(None, 100)	1024100	flatten_7[0][0]
D2 (Dense)	(None, 20)	2020	D1[0][0]

Inception Modules in Practice

In my implementation:

- No striding within the inception module
 - Perform striding only after the full module, if it is called for
- Padding='valid'
 - Keeps the dimensions the same across the different branches
- Allow for other parallel sequences
- Can specify all of these details at the command line, too!

Functional API: Multiple Input Tensors

Model construction:

- Create multiple Input objects
- Ideally, these are named

```
input_tensor1 = Input(shape=(image_size[0], image_size[1], n_channels),  
                        name="input1")  
  
input_tensor2 = Input(shape=(image_size[0], image_size[1], n_channels),  
                        name="input2")
```

- Model creation: provide list of Input objects

```
model = Model(inputs=[input_tensor1, input_tensor2],  
              outputs=output_tensor)
```


Functional API: Multiple Input Tensors

Model use:

- Provide list of inputs (in order):

```
model.fit([ins1, ins2], outs)
pred = model.predict([ins1, ins2])
```

- Or provide a dict:

```
ins_dict = {'input1': ins1, 'input2': ins2}
model.fit(ins_dict, outs)
pred = model.predict(ins_dict)
```

Functional API: Multiple Output Tensors

- `model.fit/predict`: mechanics are the same as for multiple Input tensors
 - Provide a list or a dict in place of single numpy arrays
- `model.compile()`:
 - `loss`: one for each output
 - Again, provide as list or a dict
 - `loss_weights`: weights for each loss in computing the aggregate loss. This aggregate loss is what is optimized

Functional API: Sharing Parameters of a Layer

- In some cases, we want to have the same sub-network placed in different locations within a larger network
- If these sub-networks perform the same function, but with different data, it makes sense for us to use the same parameters for both

Sharing Parameters of a Layer

```
input_tensor1 = Input(shape=(1000,), name="input1")
input_tensor2 = Input(shape=(1000,), name="input2")
```

```
# Create a dense layer
dense = Dense(units=100, activation='elu')
```

```
# Use the dense layer for two pathways
dense1_tensor = dense(input_tensor1)
dense2_tensor = dense(input_tensor2)
```

```
# Use dense1_tensor and dense2_tensor together to compute a model output
```

Gradients passing through both dense1/dense2_tensor will result in changes to the parameters of dense

Functional API: Models are Layers!

- Any model can be used as a sub-component of a larger model
- A model takes as input one or more tensors and returns one or more tensors
- During training, error information is propagated through these sub-components and trainable parameters are adjusted

Example: Two-Image Inception

Use our inception model as is, except cut off last dense layers:

- inception -> inception -> flatten -> dense(100)

New model:

- Takes two consecutive images as input
- Each image is passed through the same inception model
- Results are concatenated
- Several dense layers (down to classification)

Example: Modified Inception Model

```
def create_inception_subnetwork(image_size, n_channels, lambda_regularization, activation='elu'):
    input_tensor = Input(shape=(image_size[0], image_size[1], n_channels), name="input")

    i1_tensor = inception_module(input_tensor, (10, (10,10), (10,10), 10), activation,
                                lambda_regularization, name="i1")

    i2_tensor = inception_module(i1_tensor, (40, (40,40), (40,40), 40), activation,
                                lambda_regularization, name="i2")

    flatten_tensor = Flatten()(i2_tensor)
    dense1_tensor = Dense(units=100, name = "D1", ... )(flatten_tensor)
    model = Model(inputs=input_tensor, outputs=dense1_tensor)

    return model
```

Example: Dual-Input Classifier

```
def create_dual_input_network(image_size, n_channels, lambda_regularization, activation='elu'):  
    # Create an instance of the inception model  
    inception_model = create_inception_subnetwork(image_size, n_channels,  
                                                  lambda_regularization, activation)  
  
    input_tensor1 = Input(shape=(image_size[0], image_size[1], n_channels), name="input1")  
    input_tensor2 = Input(shape=(image_size[0], image_size[1], n_channels), name="input2")  
  
    # Use the model twice  
    dense1 = inception_model(input_tensor1)  
    dense2 = inception_model(input_tensor2)  
  
    # Combine the outputs  
    concatenation_tensor = Concatenate()([dense1, dense2])  
  
    :
```


Example: Dual-Input Classifier

:

```
dense3_tensor = Dense(units=20, name = "D3", ... )(concatenation_tensor)
```

```
output_tensor = Dense(units=1, activation='sigmoid', name = "output", ... )(dense3_tensor)
```

```
opt = keras.optimizers.Adam(lr=0.0001, beta_1=0.9, beta_2=0.999,  
                             epsilon=None, decay=0.0, amsgrad=False)
```

```
# Build the object model
```

```
model = Model(inputs=[input_tensor1, input_tensor2], outputs=output_tensor)
```

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

```
return model
```

Layer (type)	Output Shape	Param #	Connected to
input1 (InputLayer)	(None, 32, 32, 3)	0	
input2 (InputLayer)	(None, 32, 32, 3)	0	
model_5 (Model)	(None, 100)	1088720	input1[0][0] input2[0][0]
concatenate_9 (Concatenate)	(None, 200)	0	model_5[1][0] model_5[2][0]
D3 (Dense)	(None, 20)	4020	concatenate_9[0][0]
output (Dense)	(None, 1)	21	D3[0][0]

Total params: 1,092,761

Trainable params: 1,092,761

Non-trainable params: 0