Keras Functional API
Example: Very Deep Networks (Inception)
Inception Module
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)
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

```python
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)
```
output_tensor = Concatenate()

    ([convA_tensor, convB1_tensor, convC1_tensor, convD1_tensor])

return output_tensor
Building an Image Classifier

```python
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

densel\_tensor = Dense(units=100, activation=activation, name = "D1", ... ) (flatten\_tensor)
dense2\_tensor = Dense(units=20, activation=activation, name = "D2", ... ) (densel\_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
<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
<th>Connected to</th>
</tr>
</thead>
<tbody>
<tr>
<td>input (InputLayer)</td>
<td>(None, 32, 32, 3)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>convB0_i1 (Conv2D)</td>
<td>(None, 32, 32, 10)</td>
<td>40</td>
<td>input[0][0]</td>
</tr>
<tr>
<td>convC0_i1 (Conv2D)</td>
<td>(None, 32, 32, 10)</td>
<td>40</td>
<td>input[0][0]</td>
</tr>
<tr>
<td>MAX_i1 (MaxPooling2D)</td>
<td>(None, 32, 32, 3)</td>
<td>0</td>
<td>input[0][0]</td>
</tr>
<tr>
<td>convA_i1 (Conv2D)</td>
<td>(None, 16, 16, 10)</td>
<td>40</td>
<td>input[0][0]</td>
</tr>
<tr>
<td>convB1_i1 (Conv2D)</td>
<td>(None, 16, 16, 10)</td>
<td>910</td>
<td>convB0_i1[0][0]</td>
</tr>
<tr>
<td>convC1_i1 (Conv2D)</td>
<td>(None, 16, 16, 10)</td>
<td>2510</td>
<td>convC0_i1[0][0]</td>
</tr>
<tr>
<td>MAX_i2 (MaxPooling2D)</td>
<td>(None, 16, 16, 40)</td>
<td>0</td>
<td>convA_i1[0][0]</td>
</tr>
<tr>
<td>convA_i2 (Conv2D)</td>
<td>(None, 8, 8, 40)</td>
<td>1640</td>
<td>convB1_i2[0][0]</td>
</tr>
<tr>
<td>convB1_i2 (Conv2D)</td>
<td>(None, 8, 8, 40)</td>
<td>14440</td>
<td>convB0_i2[0][0]</td>
</tr>
<tr>
<td>convC1_i2 (Conv2D)</td>
<td>(None, 8, 8, 40)</td>
<td>40040</td>
<td>convC0_i2[0][0]</td>
</tr>
<tr>
<td>convD0_i2 (Conv2D)</td>
<td>(None, 8, 8, 40)</td>
<td>1640</td>
<td>MAX_i2[0][0]</td>
</tr>
<tr>
<td>concatenate_15 (Concatenate)</td>
<td>(None, 8, 8, 160)</td>
<td>0</td>
<td>convA_i2[0][0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>convB1_i2[0][0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>convC1_i2[0][0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>convD0_i2[0][0]</td>
</tr>
<tr>
<td>flatten_7 (Flatten)</td>
<td>(None, 10240)</td>
<td>0</td>
<td>concatenate_15[0][0]</td>
</tr>
<tr>
<td>D1 (Dense)</td>
<td>(None, 100)</td>
<td>1024100</td>
<td>flatten_7[0][0]</td>
</tr>
<tr>
<td>D2 (Dense)</td>
<td>(None, 20)</td>
<td>2020</td>
<td>D1[0][0]</td>
</tr>
</tbody>
</table>
Performance: Mugs vs Cans

Caveats:
- 32x32 images
- Little training
- No tuning

![ROC Curve with AUC values](image)
Multiple Input or Output Tensors
Functional API: Multiple Input Tensors

Model construction:

• Create multiple Input objects
• Ideally, these are named

```python
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

```python
model = Model(inputs=[input_tensor1, input_tensor2],
              outputs=output_tensor)
```
Functional API: Multiple Input Tensors

Model use:

• Provide list of inputs (in order):
  ```python
  model.fit([ins1, ins2], outs)
  pred = model.predict([ins1, ins2])
  ```

• Or provide a dict:
  ```python
  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

```python
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")

# 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)

# Concatenate dense1_tensor and dense2_tensor and (through multiple layers),
# make a single prediction

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

```python
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
```
<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
<th>Connected to</th>
</tr>
</thead>
<tbody>
<tr>
<td>input1 (InputLayer)</td>
<td>(None, 32, 32, 3)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>input2 (InputLayer)</td>
<td>(None, 32, 32, 3)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>model_5 (Model)</td>
<td>(None, 100)</td>
<td>1088720</td>
<td>input1[0][0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>input2[0][0]</td>
</tr>
<tr>
<td>concatenate_9 (Concatenate)</td>
<td>(None, 200)</td>
<td>0</td>
<td>model_5[1][0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>model_5[2][0]</td>
</tr>
<tr>
<td>D3 (Dense)</td>
<td>(None, 20)</td>
<td>4020</td>
<td>concatenate_9[0][0]</td>
</tr>
<tr>
<td>output (Dense)</td>
<td>(None, 1)</td>
<td>21</td>
<td>D3[0][0]</td>
</tr>
</tbody>
</table>

Total params: 1,092,761
Trainable params: 1,092,761
Non-trainable params: 0
Example: Split Inputs

# 1/2 are consecutive images (making this assumption for simplicity
ins_training1 = ins_training[0::2,:,:,:]
ins_training2 = ins_training[1::2,:,:,:]
# Just take label from the first half
outs_training_new = outs_training[0::2]

ins_validation1 = ins_validation[0::2,:,:,:]
ins_validation2 = ins_validation[1::2,:,:,:]
outs_validation_new = outs_validation[0::2]
Example: Generator with Two Inputs

def training_set_generator_dual_input(ins1, ins2, outs, batch_size=10,
    input_name1='input1',
    input_name2='input2',
    output_name='output'):

    while True:
        example_indices = [random.choice(range(ins1.shape[0]))
                           for k in range(batch_size)]

        yield({input_name1: ins1[example_indices,:,:,:],
               input_name2: ins2[example_indices,:,:,:],
               {output_name: outs[example_indices]})
Performance: Mugs vs Cans

Caveats (again):
• Little training
• No tuning
Model within a Model

A very powerful idea

• Use a single sub-model in multiple ways (we just did this)
  • This effectively increases the training set size that the model has available to it

• Instrumenting a model vs training it
Instrumentation vs Training

For our classifier models:

• Training: input is a model; output is a probability

• But, after training, it is sometimes useful to look inside the different layers to see how they are participating in the computation
  • We did this in a class demo by creating a second model that was a copy of the trained model (including parameters), but with different layers as outputs
  • This allowed us to ask: what does channel k look like when the input is a specific image?
Instrumentation vs Training

An alternative approach with nested models:

• Inner model: instrumentation
  • Input: image
  • Output: all of the output layers that are of interest, including the class probability vector

• Outer model: training
  • Input: image
  • Output: class probability vector
  • In between: include an instance of the Inner model, but output only selects the class probability vector (all other outputs are ignored)

• Our model building function returns both. Training of the outer model selects parameters of both. The inner model can then be queried with new images!
HW 5 Proposal

• Same classification problem
• Two model types are possible
  • Inception-like branching structures
  • Take multiple images of the same object (in sequential order) and predict one class label for the set