Sequential Model Example

```
seq_model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```

Functional API

1 — 2 — 3

Input

Define input to the model

Layers

Define a set of interconnected layers on the input

Model

Define the model using the input and output layers

Defining the Input

```
from tensorflow.keras.layers import Input

...

input = Input(shape=(28, 28))
```

Defining the layers

```
from tensorflow.keras.layers import Dense, Flatten

...

x = Flatten()(input)
x = Dense(128, activation="relu")(x)
predictions = Dense(10, activation="softmax")(x)
```

When you define a layer, you add it to the next one by specifying in parentheses after the declaration of that next layer. For example, earlier you defined inputs. The next layer, will be a flattened and you specify that flattened follows the input layer like this.

The flattened layer is stored in a variable named X. If you want to add a dense layer after the flattened layer, you'll then specify a dense layer and put the variable X in parentheses after it to specify that it's the next one and you continue to finding them like this.

Defining the layers

```
from tensorflow.keras.layers import Dense, Flatten

...

x = Flatten()(input)
x = Dense(128, activation="relu")(x)
predictions = Dense(10, activation="softmax")(x)
```

A feeds into B

Defining the layers

```
from tensorflow.keras.layers import Dense, Flatten

...

x = Flatten()(input)
x = Dense(128, activation="relu")(x)
predictions = Dense(10, activation="softmax")(x)
```

Defining the layers

```
from tensorflow.keras.layers import Dense, Flatten

...

 x = Flatten()(input)
 x = Dense(128, activation="relu")(x)
 predictions = Dense(10, activation="softmax")(x)
```

Defining the layers

```
from tensorflow.keras.layers import Dense, Flatten

...

 x = Flatten()(input)
 x = Dense(128, activation="relu")(x)
 predictions = Dense(10, activation="softmax")(x)
```

Defining the Model

```
from tensorflow.keras.models import Model

...

func_model = Model(inputs=input, outputs=predictions)
```

Defining the Model

```
from tensorflow.keras.models import Model

...

func_model = Model(inputs=input, outputs=predictions)
```

Rewriting Using Functional API

```
input = Input(shape=(28,28))
x = Flatten()(input)
x = Dense(128, activation="relu")(x)
predictions = Dense(10, activation="softmax")(x)

func_model = Model(inputs=input, outputs=predictions)
```

Rewriting Using Functional API

```
input = Input(shape=(28,28))
x = Flatten()(input)
x = Dense(128, activation="relu")(x)
predictions = Dense(10, activation="softmax")(x)

func_model = Model(inputs=input, outputs=predictions)
```

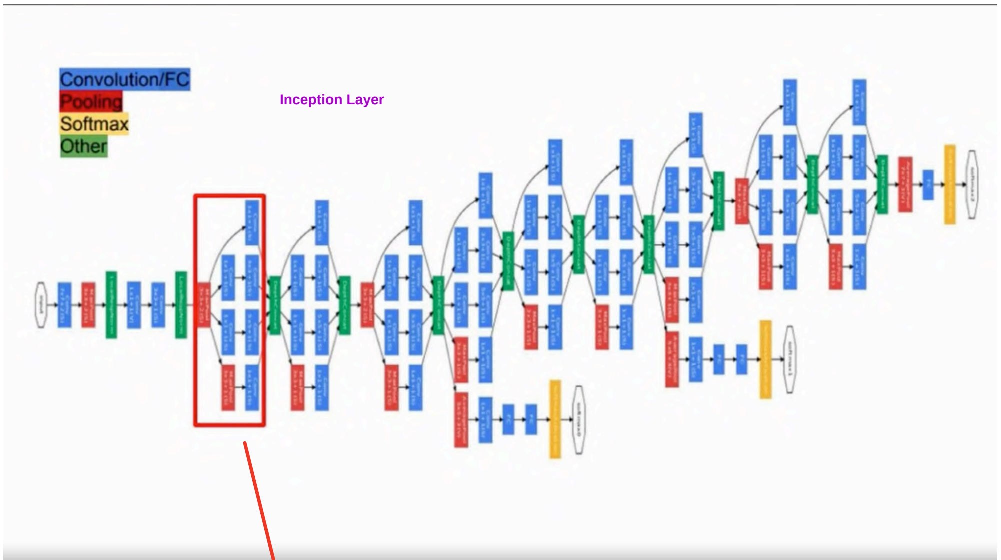
Rewriting Using Functional API

```
input = Input(shape=(28,28))
x = Flatten()(input)
x = Dense(128, activation="relu")(x)
predictions = Dense(10, activation="softmax")(x)

func_model = Model(inputs=input, outputs=predictions)
```

Rewriting Using Functional API

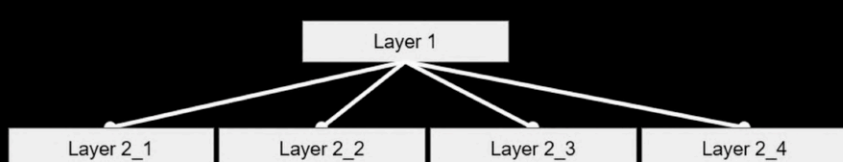
```
input = Input(shape=(28,28))
x = Flatten()(input)
x = Dense(128, activation="relu")(x)
predictions = Dense(10, activation="softmax")(x)

func_model = Model(inputs=input, outputs=predictions)
```

```
def build_model_with_functional():
    from tensorflow.keras.models import Model
    input_layer = tf.keras.Input(shape=(28, 28))
    flatten_layer = tf.keras.layers.Flatten()(input_layer)
    first_dense = tf.keras.layers.Dense(128, activation=tf.nn.relu)(flatten_layer)
    output_layer = tf.keras.layers.Dense(10, activation=tf.nn.softmax)(first_dense)
    func_model = Model(inputs=input_layer, outputs=output_layer)
    return func_model
```

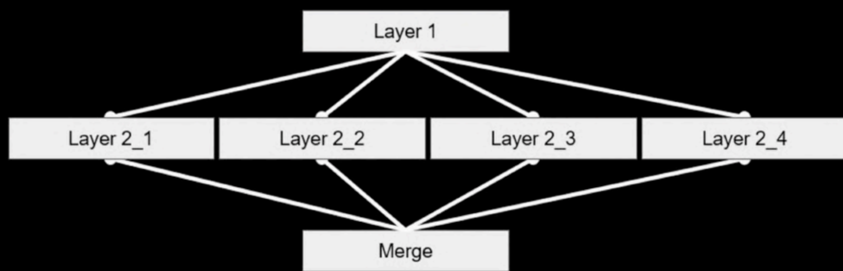



```
layer1 = Dense(32)
layer2_1 = Dense(32)(layer1)
layer2_2 = Dense(32)(layer1)
layer2_3 = Dense(32)(layer1)
layer2_4 = Dense(32)(layer1)
merge = Concatenate([layer2_1, layer2_2, layer2_3, layer2_4])
```

```
layer1 = Dense(32)
layer2_1 = Dense(32)(layer1)
layer2_2 = Dense(32)(layer1)
layer2_3 = Dense(32)(layer1)
layer2_4 = Dense(32)(layer1)
merge = Concatenate([layer2_1, layer2_2, layer2_3, layer2_4])
```



```python
layer1 = Dense(32)
layer2_1 = Dense(32)(layer1)
layer2_2 = Dense(32)(layer1)
layer2_3 = Dense(32)(layer1)
layer2_4 = Dense(32)(layer1)
merge = Concatenate([layer2_1, layer2_2, layer2_3, layer2_4])
```




```python
def build_model_with_functional():
    from tensorflow.keras.models import Model
    input_layer = tf.keras.Input(shape=(28, 28))
    flatten_layer = tf.keras.layers.Flatten()(input_layer)
    first_dense = tf.keras.layers.Dense(128, activation=tf.nn.relu)(flatten_layer)
    output_layer = tf.keras.layers.Dense(10, activation=tf.nn.softmax)(first_dense)
    func_model = Model(inputs=input_layer, outputs=output_layer)
    return func_model
```

Parameters are plural

```python
func_model = Model(inputs=input_layer, outputs=output_layer)
```

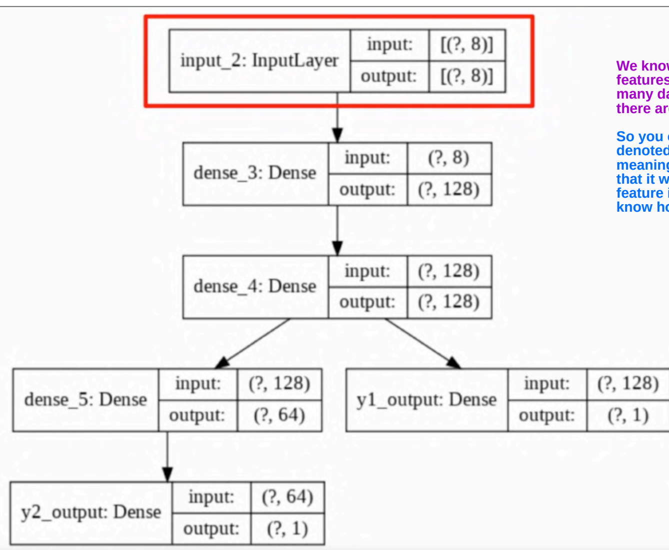
You might be wondering, does that mean that we could have multiple inputs and multiple outputs?

Well, the answer to that, is yes, and you can do them by specifying them as a list, a bit like this.

Note the square bracket syntax, which indicates a python list. You could have defined multiple inputs, so in order to tell the model that you'll be using them, you simply list them out as shown.

```
func_model = Model(inputs=[input1, input2], outputs=[output1, output2])
```

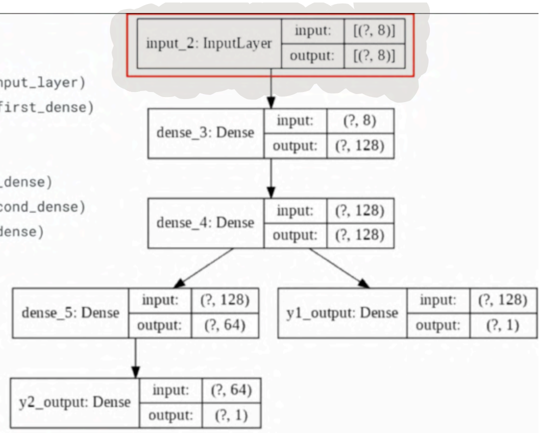
Features	Labels
X1 Relative Compactness	y1 Heating Load
X2 Surface Area	y2 Cooling Load
X3 Wall Area	
X4 Roof Area	
X5 Overall Height	
X6 Orientation	
X7 Glazing Area	
X8 Glazing Area Distribution	



We know that there are eight features but don't know how many data items of 8 features there are.

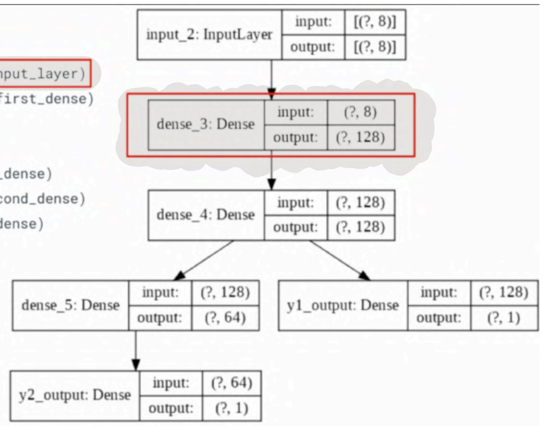
So you can see the shape is denoted by he syntax (?, 8) meaning the network knows that it will have a number 8 feature inputs, but it doesn't know how many.

```
input_layer = Input(shape=(len(train .columns),))
first_dense = Dense(units='128', activation='relu')(input_layer)
second_dense = Dense(units='128', activation='relu')(first_dense)

y1_output = Dense(units='1', name='y1_output')(second_dense)
third_dense = Dense(units='64', activation='relu')(second_dense)
y2_output = Dense(units='1', name='y2_output')(third_dense)
```



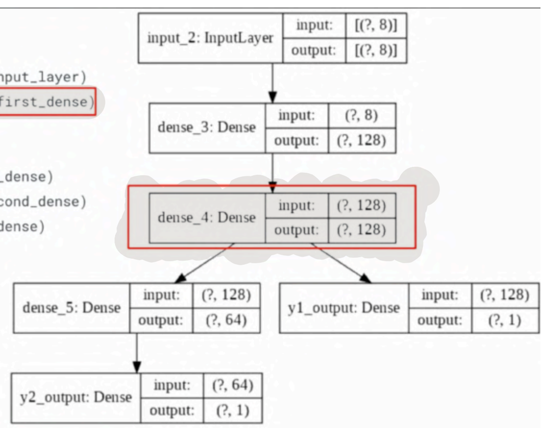

```
# Define the model with the input layer and a list of output layers
model = Model(inputs=input_layer, outputs=[y1_output, y2_output])
```

```
input_layer = Input(shape=(len(train .columns),))
first_dense = Dense(units='128', activation='relu')(input_layer)
second_dense = Dense(units='128', activation='relu')(first_dense)

y1_output = Dense(units='1', name='y1_output')(second_dense)
third_dense = Dense(units='64', activation='relu')(second_dense)
y2_output = Dense(units='1', name='y2_output')(third_dense)
```



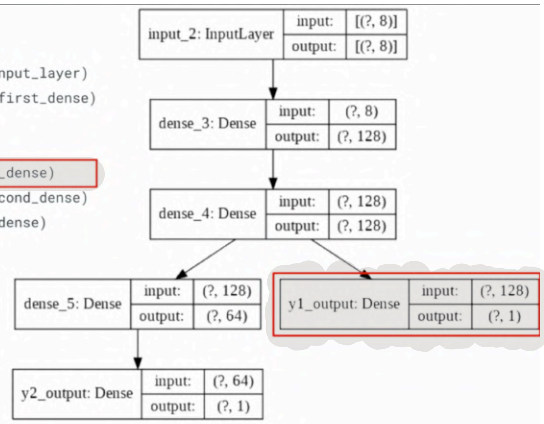

```
# Define the model with the input layer and a list of output layers
model = Model(inputs=input_layer, outputs=[y1_output, y2_output])
```

```
input_layer = Input(shape=(len(train .columns),))
first_dense = Dense(units='128', activation='relu')(input_layer)
second_dense = Dense(units='128', activation='relu')(first_dense)

y1_output = Dense(units='1', name='y1_output')(second_dense)
third_dense = Dense(units='64', activation='relu')(second_dense)
y2_output = Dense(units='1', name='y2_output')(third_dense)
```



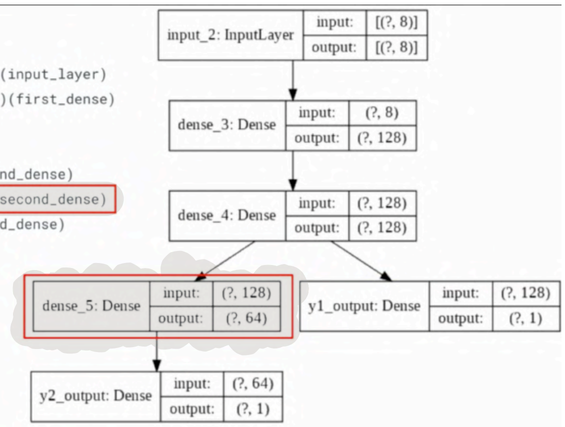

```
# Define the model with the input layer and a list of output layers
model = Model(inputs=input_layer, outputs=[y1_output, y2_output])
```

```python
input_layer = Input(shape=(len(train .columns),))
first_dense = Dense(units='128', activation='relu')(input_layer)
second_dense = Dense(units='128', activation='relu')(first_dense)

y1_output = Dense(units='1', name='y1_output')(second_dense)
third_dense = Dense(units='64', activation='relu')(second_dense)
y2_output = Dense(units='1', name='y2_output')(third_dense)
```



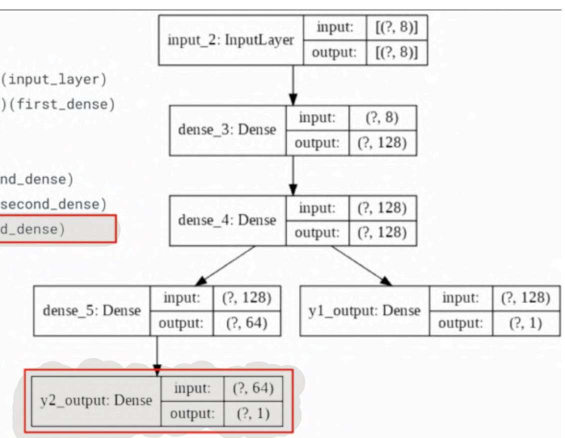

```python
# Define the model with the input layer and a list of output layers
model = Model(inputs=input_layer, outputs=[y1_output, y2_output])
```

```python
input_layer = Input(shape=(len(train .columns),))
first_dense = Dense(units='128', activation='relu')(input_layer)
second_dense = Dense(units='128', activation='relu')(first_dense)

y1_output = Dense(units='1', name='y1_output')(second_dense)
third_dense = Dense(units='64', activation='relu')(second_dense)
y2_output = Dense(units='1', name='y2_output')(third_dense)
```

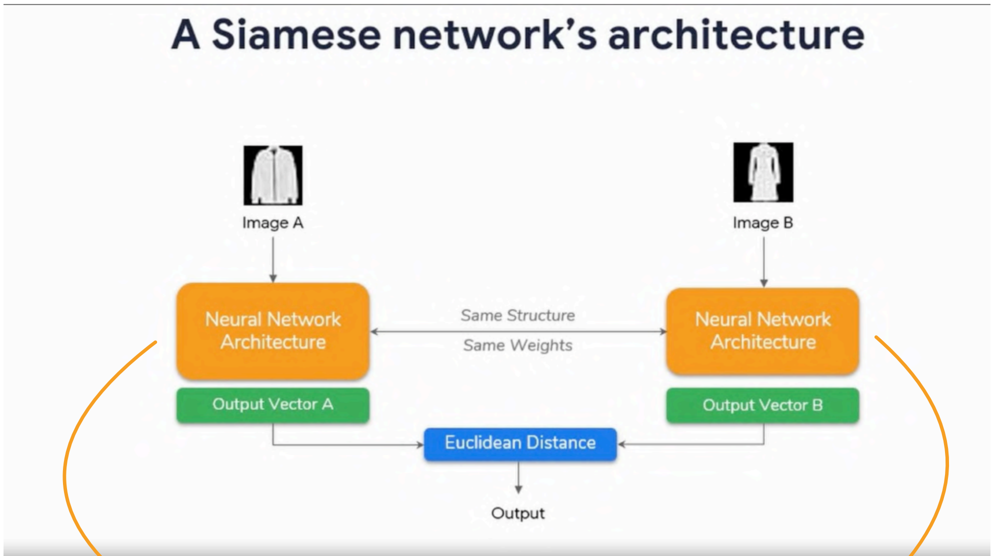



```python
# Define the model with the input layer and a list of output layers
model = Model(inputs=input_layer, outputs=[y1_output, y2_output])
```

```python
input_layer = Input(shape=(len(train .columns),))
first_dense = Dense(units='128', activation='relu')(input_layer)
second_dense = Dense(units='128', activation='relu')(first_dense)

y1_output = Dense(units='1', name='y1_output')(second_dense)
third_dense = Dense(units='64', activation='relu')(second_dense)
y2_output = Dense(units='1', name='y2_output')(third_dense)
```

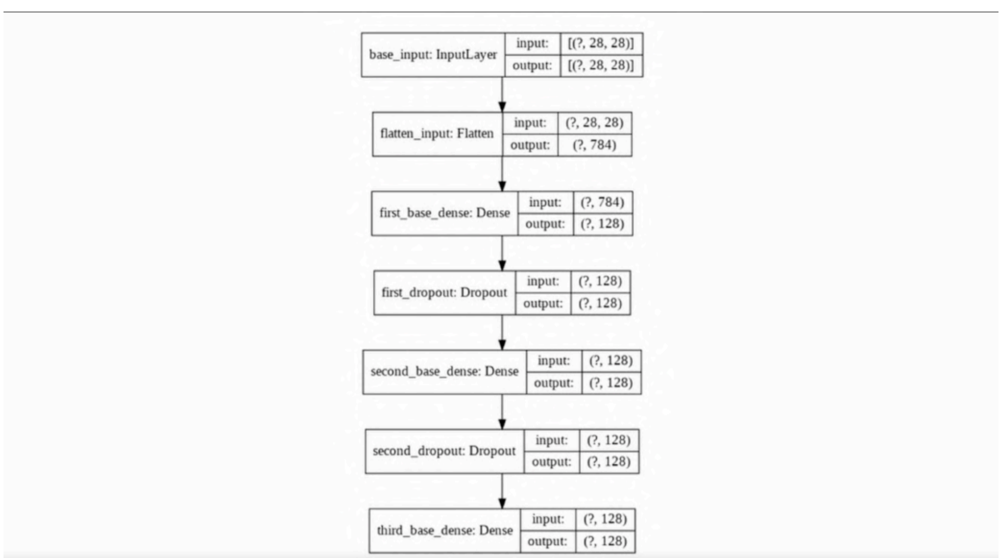


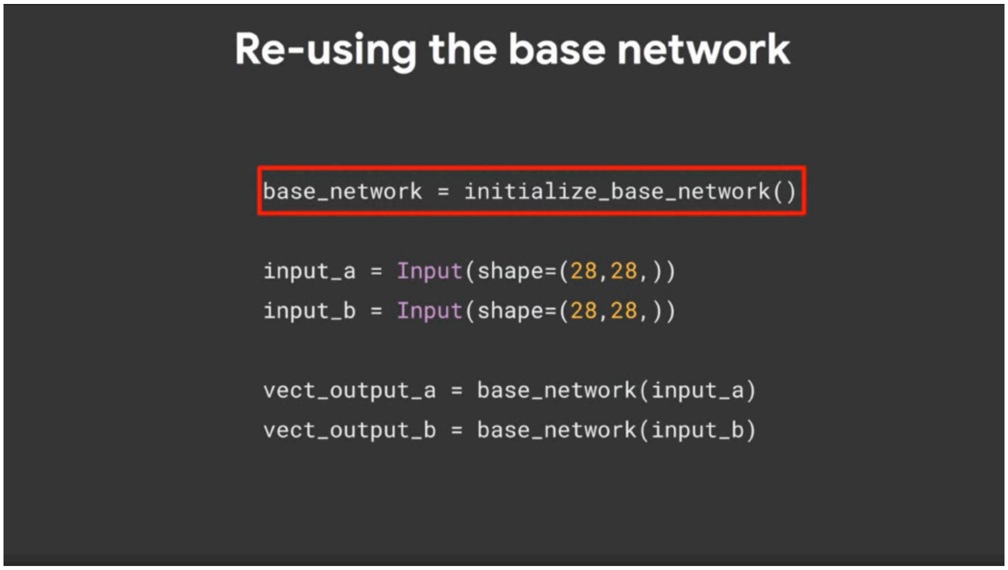

```python
# Define the model with the input layer and a list of output layers
model = Model(inputs=input_layer, outputs=[y1_output, y2_output])
```

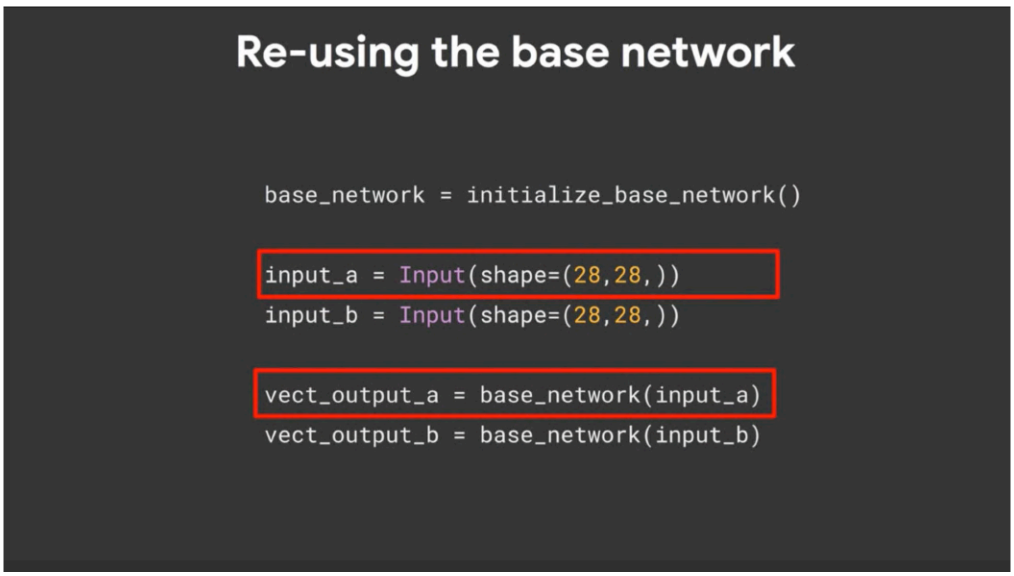


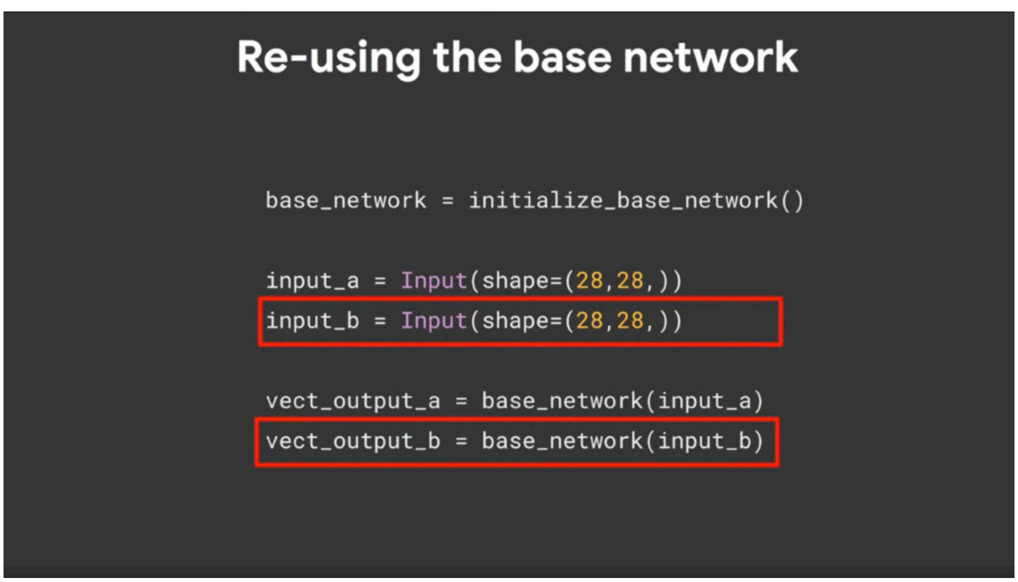
Defining the Base Network

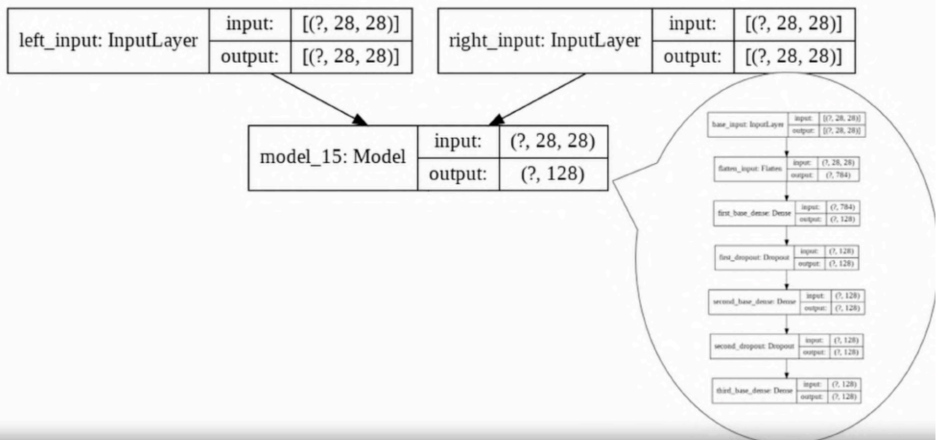
```
def initialize_base_network():
  input = Input(shape=(28,28,))
  x = Flatten()(input)
  x = Dense(128, activation='relu')(x)
  x = Dropout(0.1)(x)
  x = Dense(128, activation='relu')(x)
  x = Dropout(0.1)(x)
  x = Dense(128, activation='relu')(x)
  return Model(inputs=input, outputs=x)
```




Re-using the base network
base_network = initialize_base_network()
input_a = Input(shape=(28,28,))
input_b = Input(shape=(28,28,))
vect_output_a = base_network(input_a)
vect_output_b = base_network(input_b)

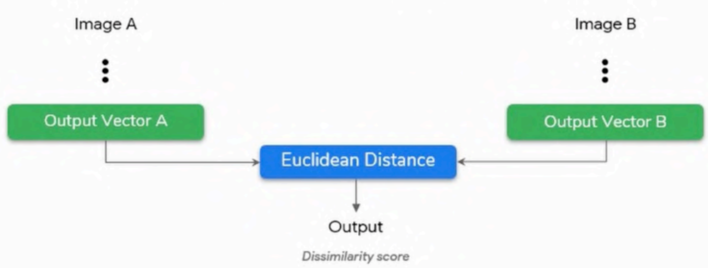

Re-using the base network
base_network = initialize_base_network()
input_a = Input(shape=(28,28,))
input_b = Input(shape=(28,28,))
vect_output_a = base_network(input_a)
vect_output_b = base_network(input_b)


Re-using the base network
base_network = initialize_base_network()
input_a = Input(shape=(28,28,))
input_b = Input(shape=(28,28,))
vect_output_a = base_network(input_a)
vect_output_b = base_network(input_b)



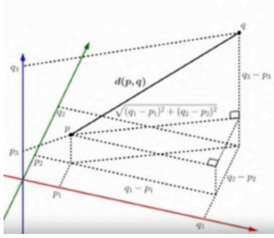
Output of the network

Similarity between two input images



http://mathonline.wikidot.com/the-distance-between-two-vectors

```
def euclidean_distance(vects):
    x, y = vects
    sum_square = K.sum(K.square(x - y), axis=1, keepdims=True)
    return K.sqrt(K.maximum(sum_square, K.epsilon()))


def eucl_dist_output_shape(shapes):
    shape1, shape2 = shapes
    return (shape1[0], 1)
```



Output layer is euclidean distance

We can use a Lambda layer to call the euclidean distance functions. Lambda layers and TensorFlow give you the ability to code custom code, so they're perfectly suited for it.

You can also specify that it follows the two vector outputs from earlier, by putting them into a list, which you can see here. It's still using the functional API syntax.

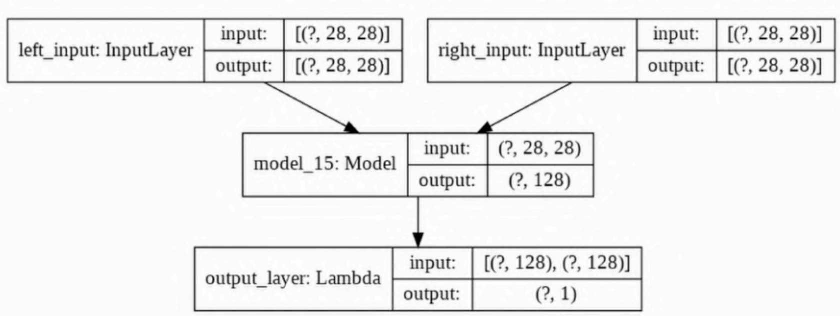
```
output = Lambda(euclidean_distance,
                output_shape=eucl_dist_output_shape)([vect_output_a, vect_output_b])
```

Defining the final model

We can define our models as always by calling the model object, and specifying the inputs and outputs. We created the inputs earlier, and the output is the Lambda layer that you just created.

```
model = Model([input_a, input_b], output)
```

Now the architecture looks like this, with the Lambda layer following the base model.



Defining the final model

```
model = Model([input_a, input_b], output)

rms = RMSprop()
model.compile(loss=contrastive_loss, optimizer=rms)
```

Train the Model

Now we get to training the network. Note how we feed in the training values. The Data has been arranged into pairs of images with a label denoting this similarity.

The syntax tr_pairs[:, 0] will take the first item in the pair and feed it into the left side of the network

```
model.fit([tr_pairs[:,0], tr_pairs[:,1]], tr_y, # Training data
          epochs=20,
          batch_size=128,
          validation_data=([ts_pairs[:,0], ts_pairs[:,1]], ts_y))
```

Train the Model

tr_pairs[:, 1] will take the second item in the pair and feed that into the right side of the network.

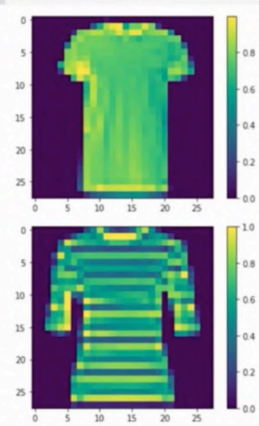
```
model.fit([tr_pairs[:,0], tr_pairs[:,1]], tr_y, # Training data
          epochs=20,
          batch_size=128,
          validation_data=([ts_pairs[:,0], ts_pairs[:,1]], ts_y))
```

Train the Model

tr_y will contain the labels, where it's zero for dissimilar pairs and one for similar ones. If we determine that for the current pair, the two values being fed in are similar

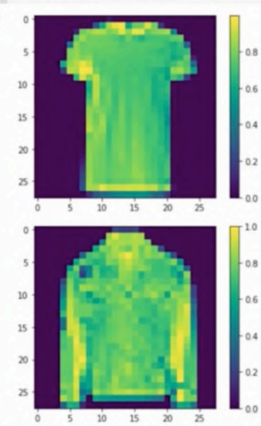
```
model.fit([tr_pairs[:,0], tr_pairs[:,1]], tr_y, # Training data
          epochs=20,
          batch_size=128,
          validation_data=([ts_pairs[:,0], ts_pairs[:,1]], ts_y))
```

```
this_pair = 8
show_image(tr_pairs[this_pair][0])
show_image(tr_pairs[this_pair][1])
print(tr_y[this_pair])
```

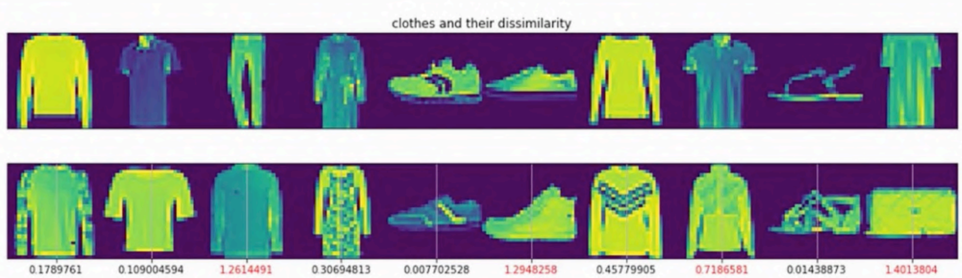


1.0

```
this_pair = 9
show_image(tr_pairs[this_pair][0])
show_image(tr_pairs[this_pair][1])
print(tr_y[this_pair])
```



0.0



Energy efficiency Data Set:

- https://archive.ics.uci.edu/ml/datasets/Energy+efficiency

Learning a Similarity Metric Discriminatively, with Application to Face Verification:

- http://yann.lecun.com/exdb/publis/pdf/chopra-05.pdf

Similarity Learning with (or without) Convolutional Neural Network:

- http://slazebni.cs.illinois.edu/spring17/lec09_similarity.pdf

The Distance Between Two Vectors:

- http://mathonline.wikidot.com/the-distance-between-two-vectors