

We're trying to understand the Functional APIs.

Deep_Input: InputLayer

```
graph TD; A[Deep_Input: InputLayer] --> B[dense_30: Dense]; B --> C[dense_31: Dense]; D[Wide_Input: InputLayer] --> E[concatenate: Concatenate]; C --> E; E --> F[Output: Dense];
```

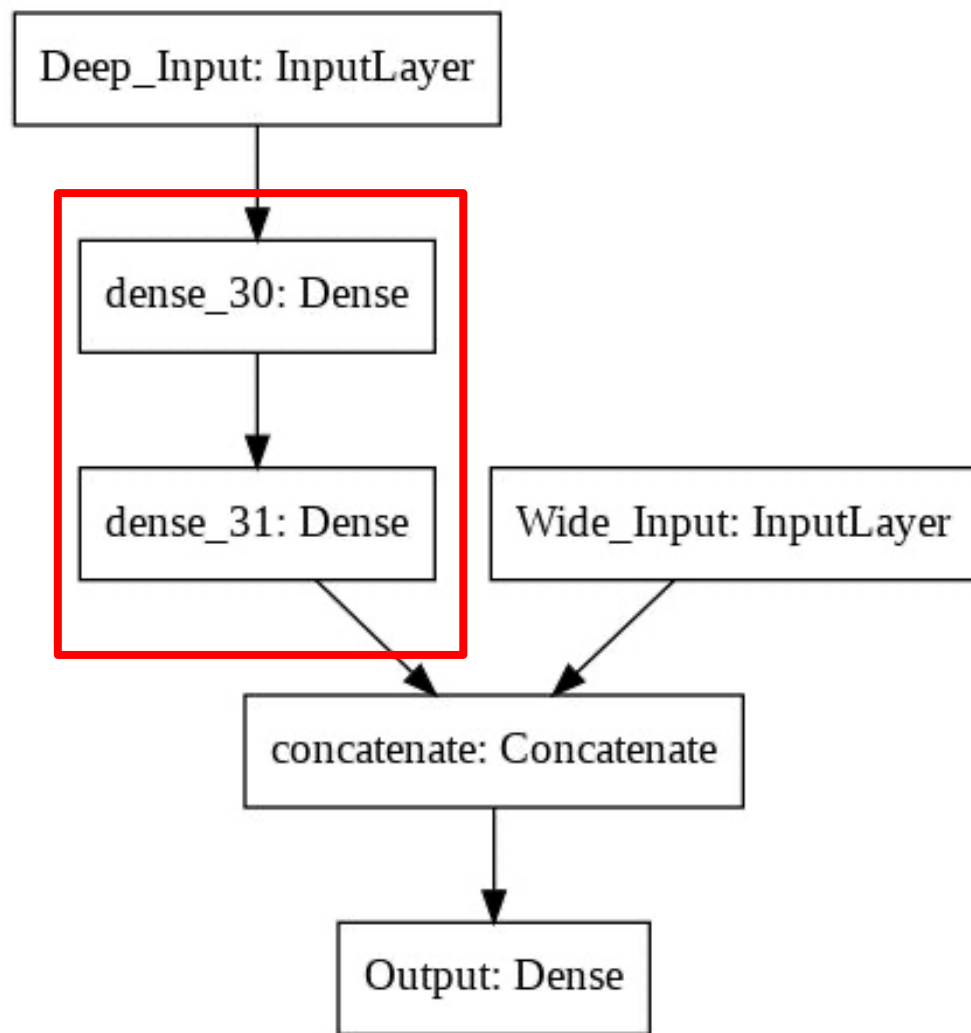
dense_30: Dense

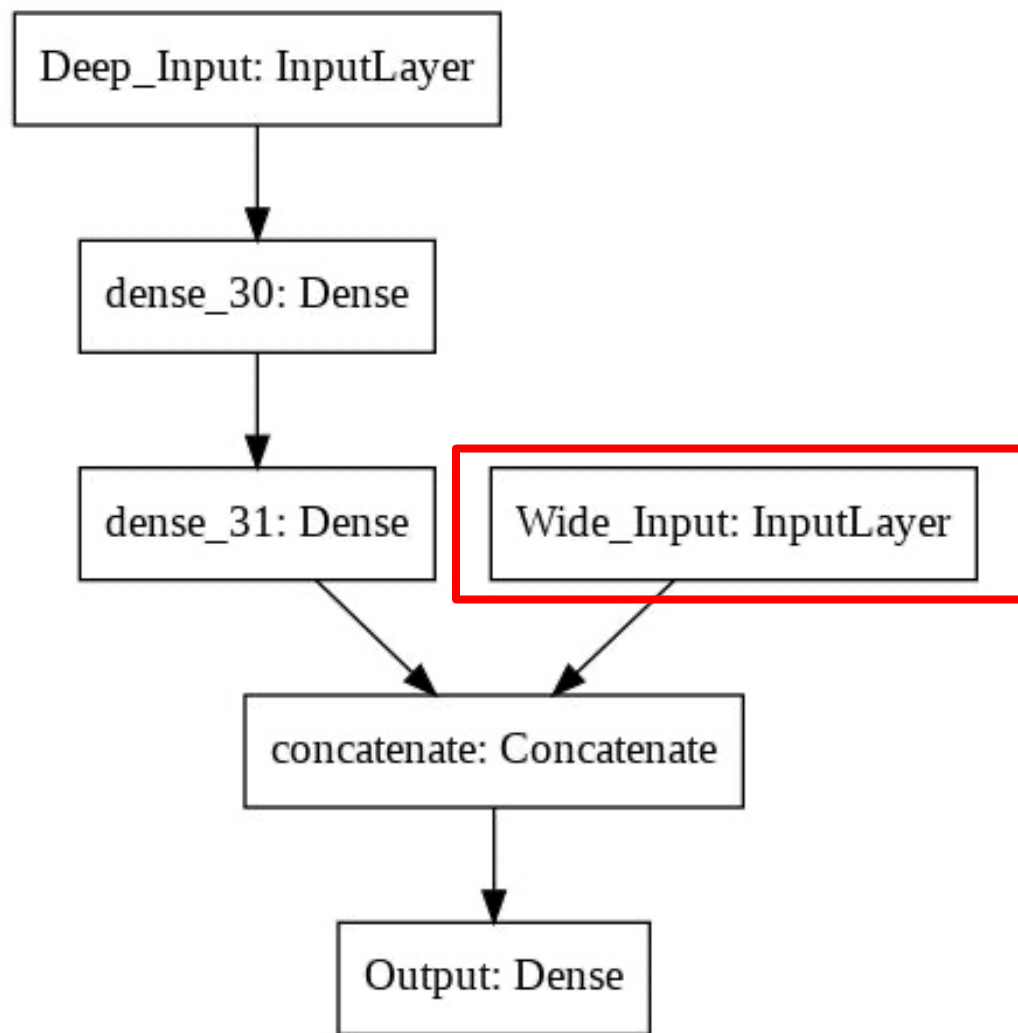
dense_31: Dense

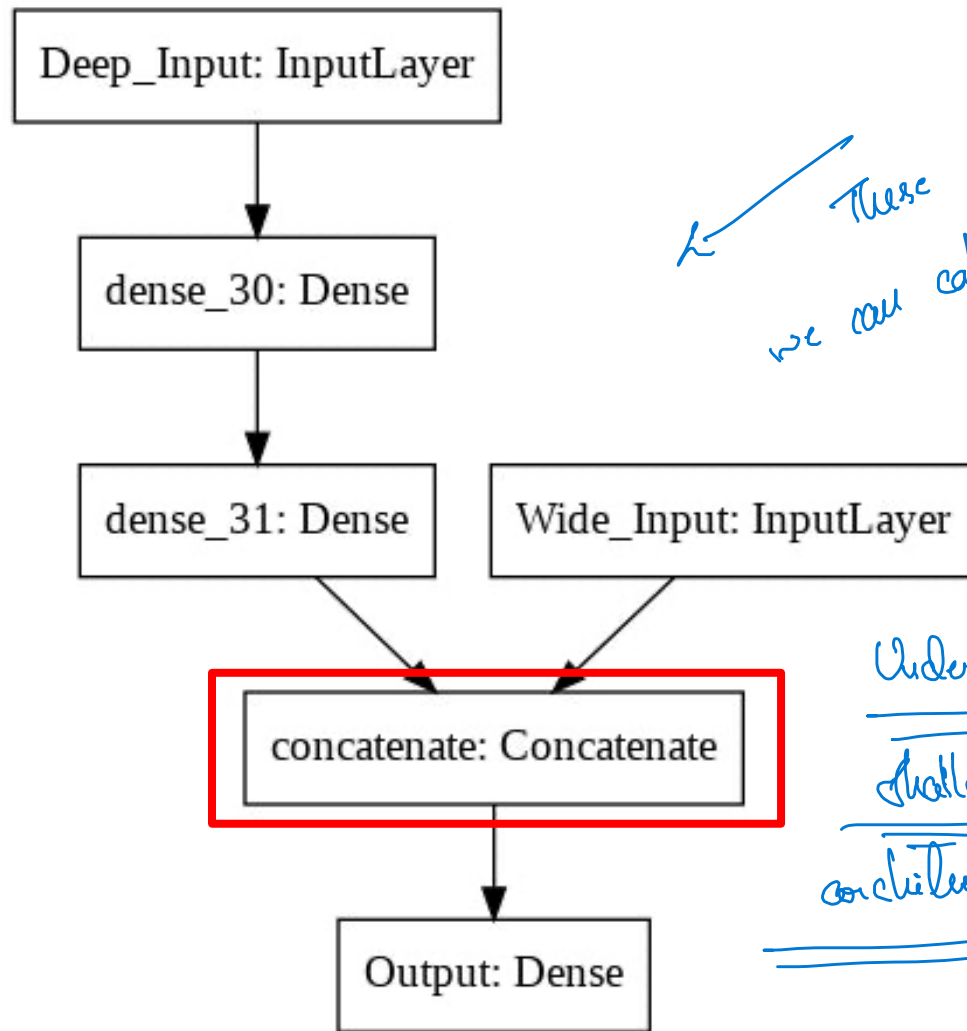
Wide_Input: InputLayer

concatenate: Concatenate

Output: Dense

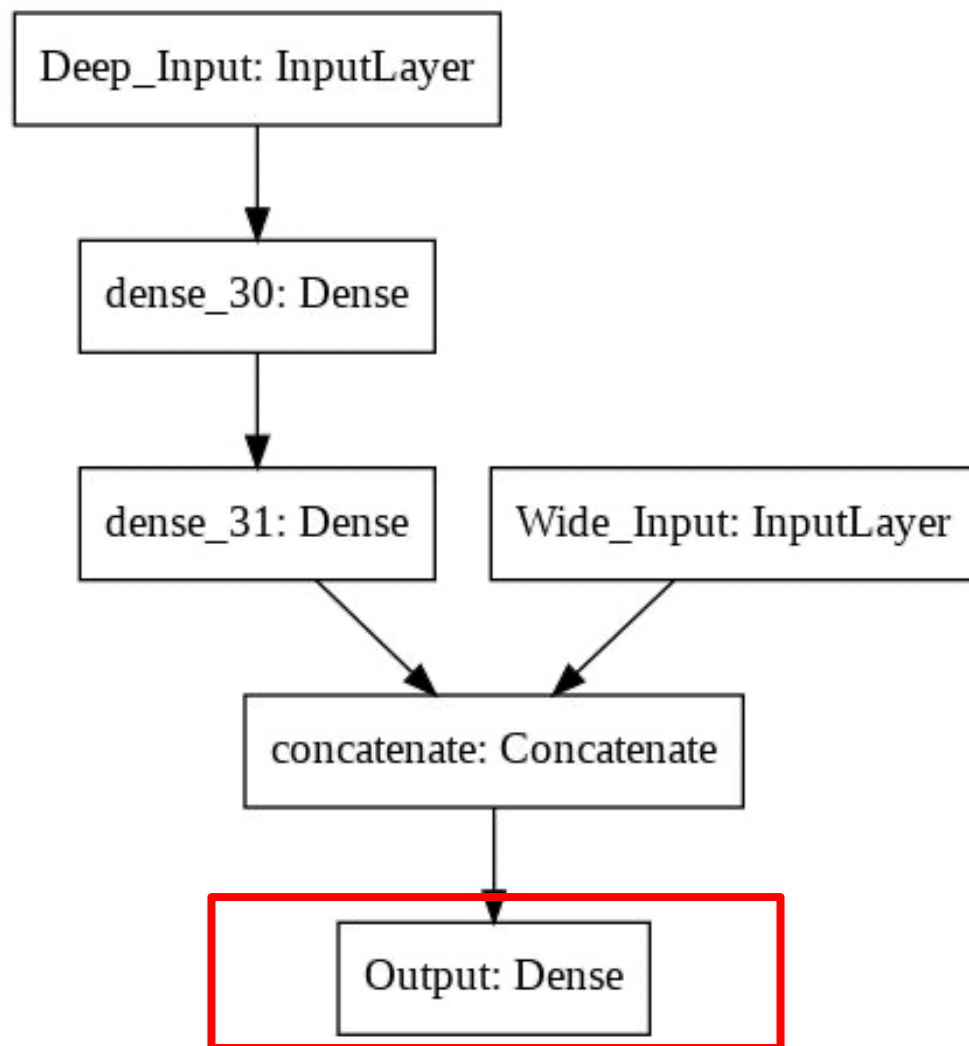




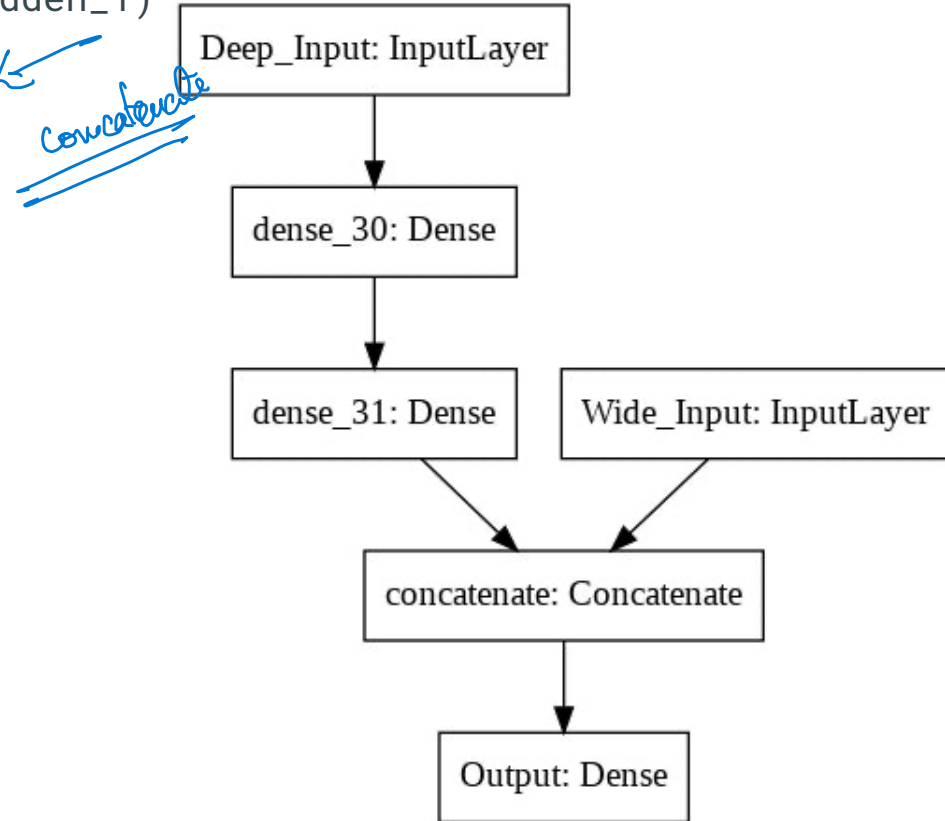


These models something
we can call as deep
widder
modeling
approach.

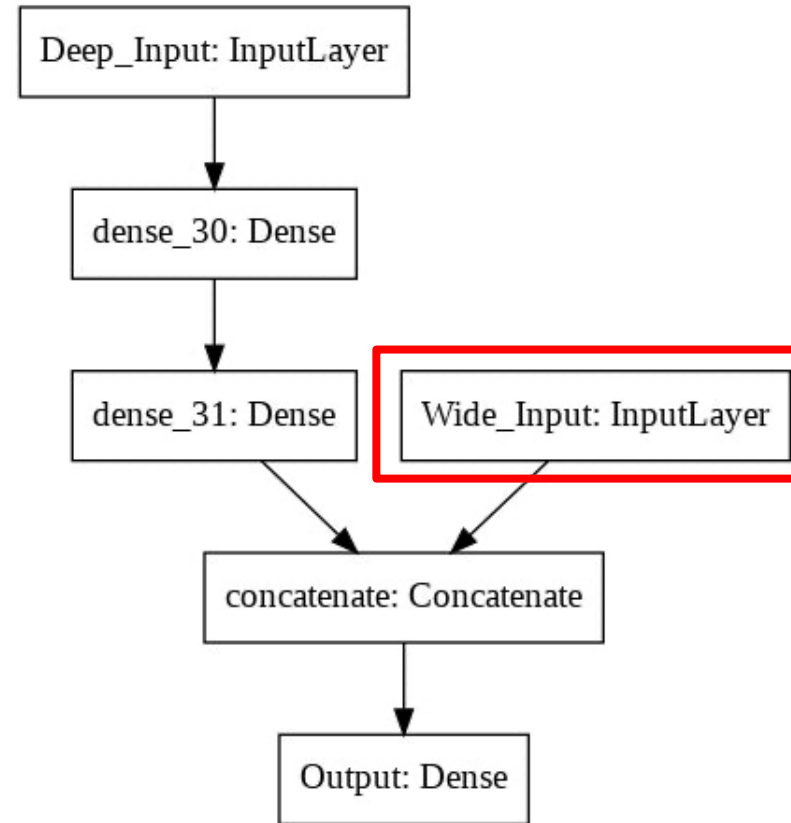
Understand the wide &
shallow deep neural network
architecture



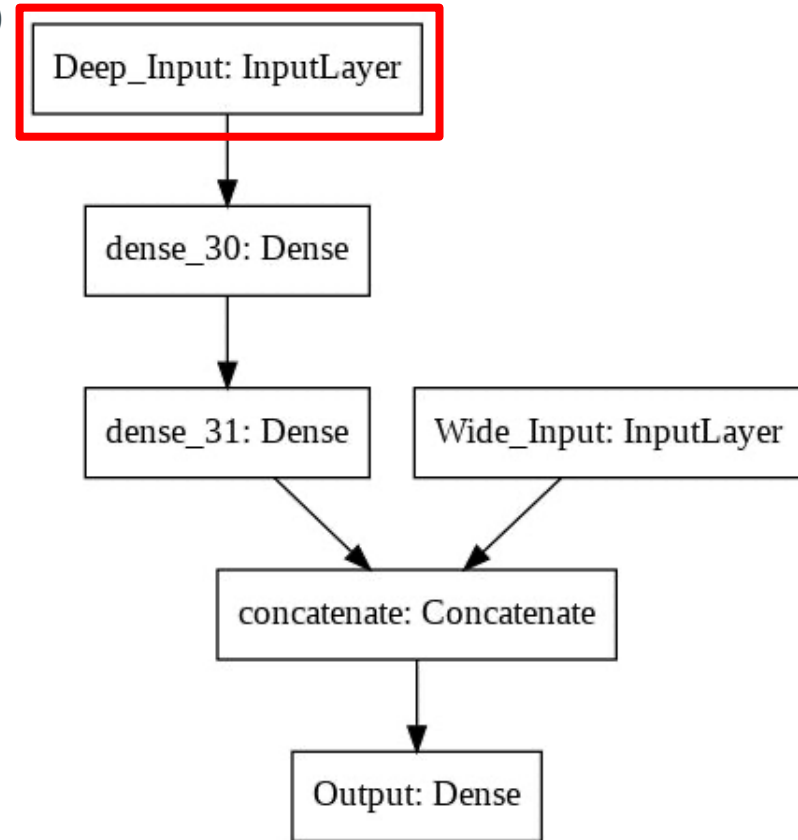
```
input_a = Input(shape=[1], name="Wide_Input")
input_b = Input(shape=[1], name="Deep_Input")
hidden_1 = Dense(30, activation="relu")(input_b)
hidden_2 = Dense(30, activation="relu")(hidden_1)
concat = concatenate([input_a, hidden_2])
output = Dense(1, name="Output")(concat)
model = Model(inputs=[input_a, input_b],
               outputs=[output])
```



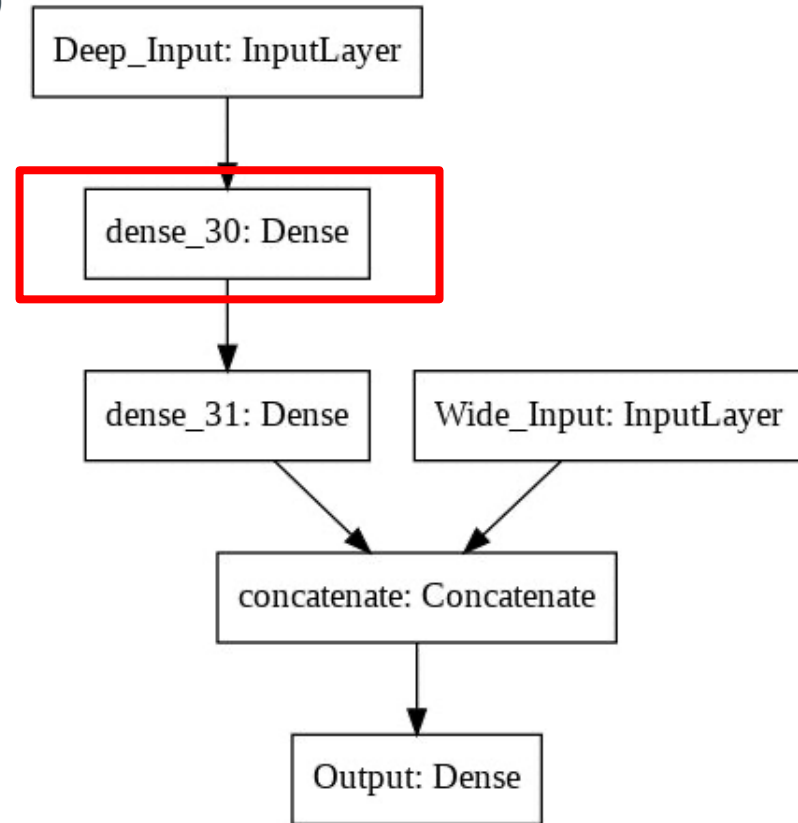
```
input_a = Input(shape=[1], name="Wide_Input")
input_b = Input(shape=[1], name="Deep_Input")
hidden_1 = Dense(30, activation="relu")(input_b)
hidden_2 = Dense(30, activation="relu")(hidden_1)
concat = concatenate([input_a, hidden_2])
output = Dense(1, name="Output")(concat)
model = Model(inputs=[input_a, input_b],
              outputs=[output])
```



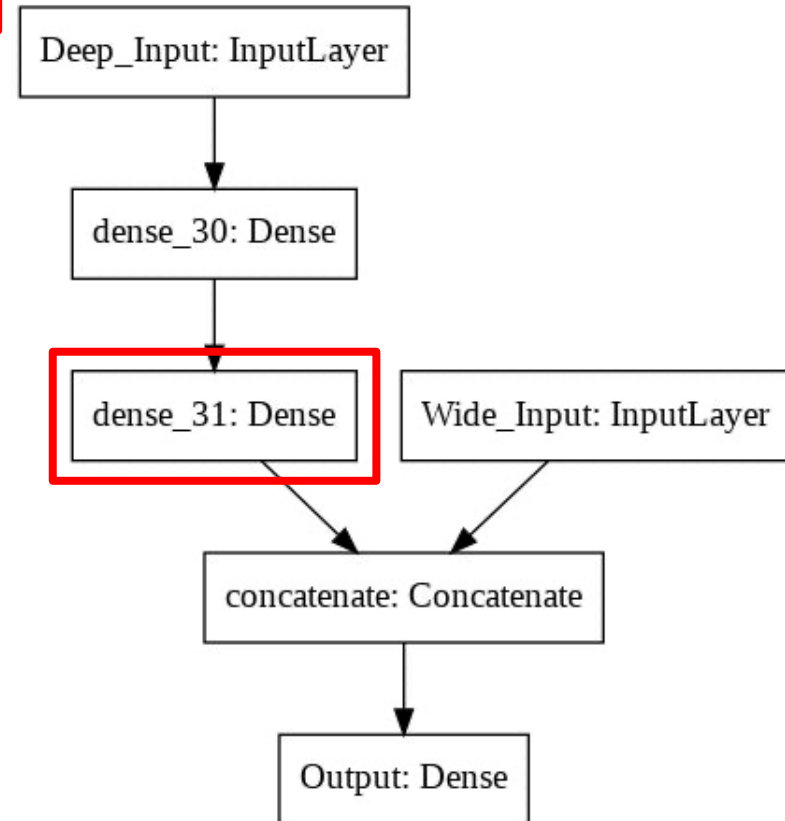

```
input_a = Input(shape=[1], name="Wide_Input")
input_b = Input(shape=[1], name="Deep_Input")
hidden_1 = Dense(30, activation="relu")(input_b)
hidden_2 = Dense(30, activation="relu")(hidden_1)
concat = concatenate([input_a, hidden_2])
output = Dense(1, name="Output")(concat)
model = Model(inputs=[input_a, input_b],
              outputs=[output])
```



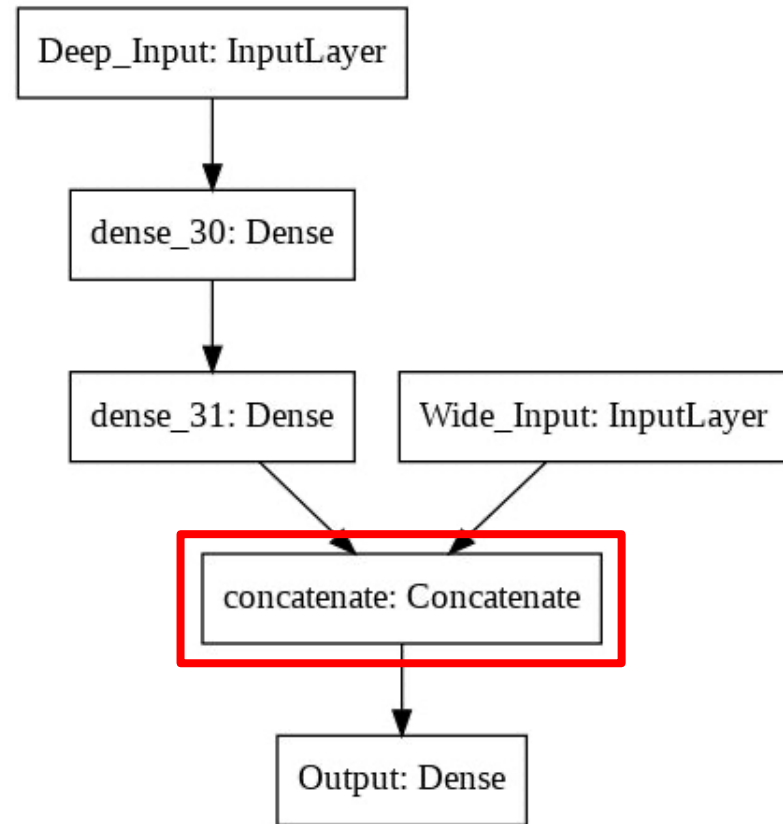
```
input_a = Input(shape=[1], name="Wide_Input")
input_b = Input(shape=[1], name="Deep_Input")
hidden_1 = Dense(30, activation="relu")(input_b)
hidden_2 = Dense(30, activation="relu")(hidden_1)
concat = concatenate([input_a, hidden_2])
output = Dense(1, name="Output")(concat)
model = Model(inputs=[input_a, input_b],
              outputs=[output])
```



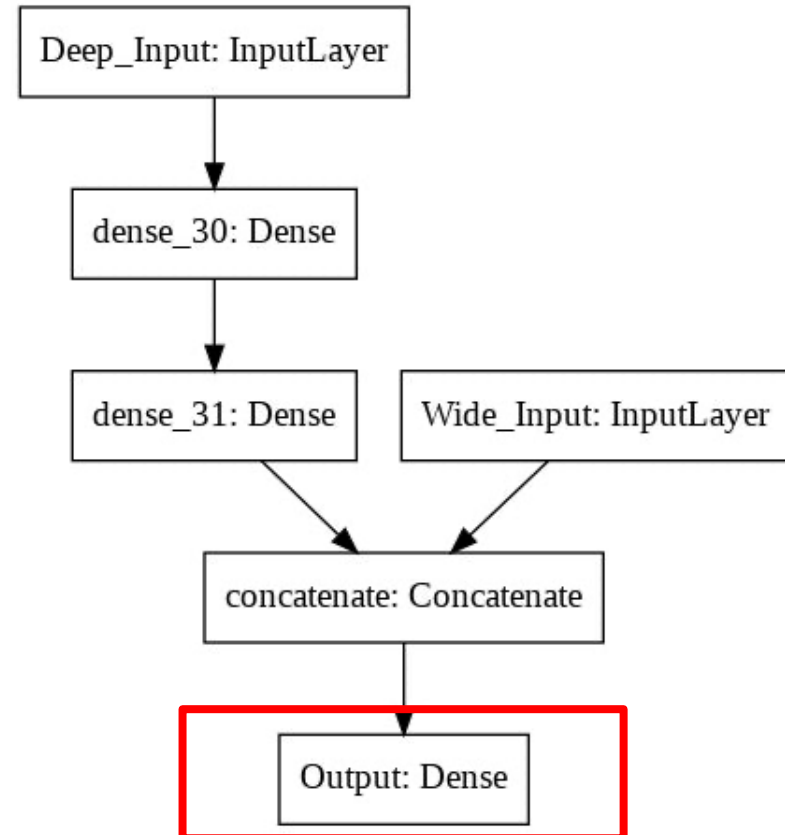
```
input_a = Input(shape=[1], name="Wide_Input")
input_b = Input(shape=[1], name="Deep_Input")
hidden_1 = Dense(30, activation="relu")(input_b)
hidden_2 = Dense(30, activation="relu")(hidden_1)
concat = concatenate([input_a, hidden_2])
output = Dense(1, name="Output")(concat)
model = Model(inputs=[input_a, input_b],
              outputs=[output])
```



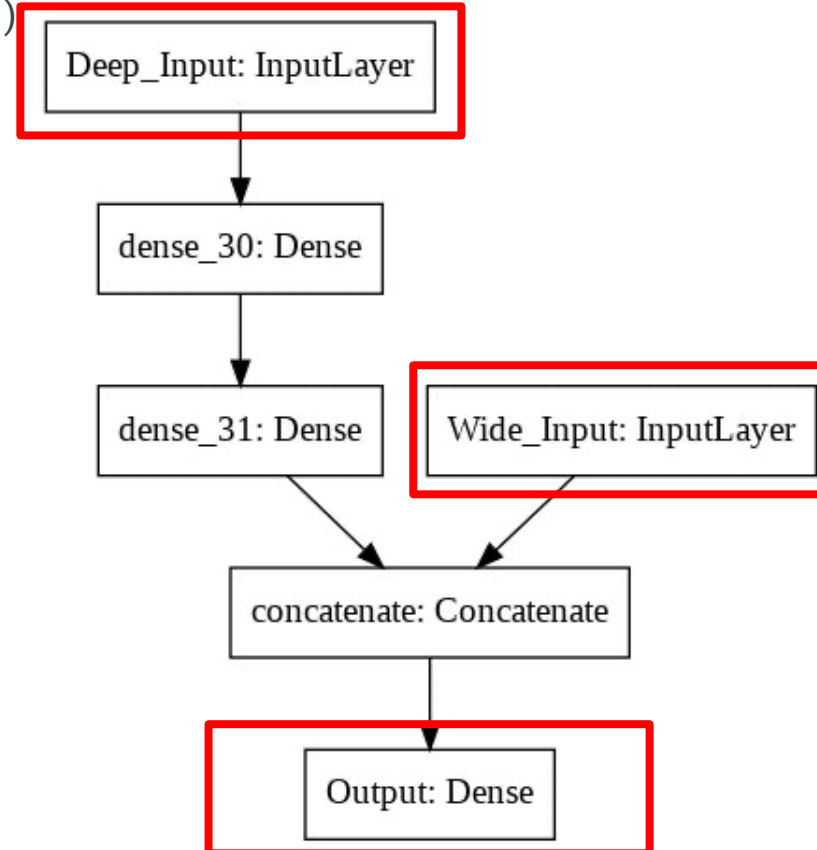
```
input_a = Input(shape=[1], name="Wide_Input")
input_b = Input(shape=[1], name="Deep_Input")
hidden_1 = Dense(30, activation="relu")(input_b)
hidden_2 = Dense(30, activation="relu")(hidden_1)
concat = concatenate([input_a, hidden_2])
output = Dense(1, name="Output")(concat)
model = Model(inputs=[input_a, input_b],
              outputs=[output])
```



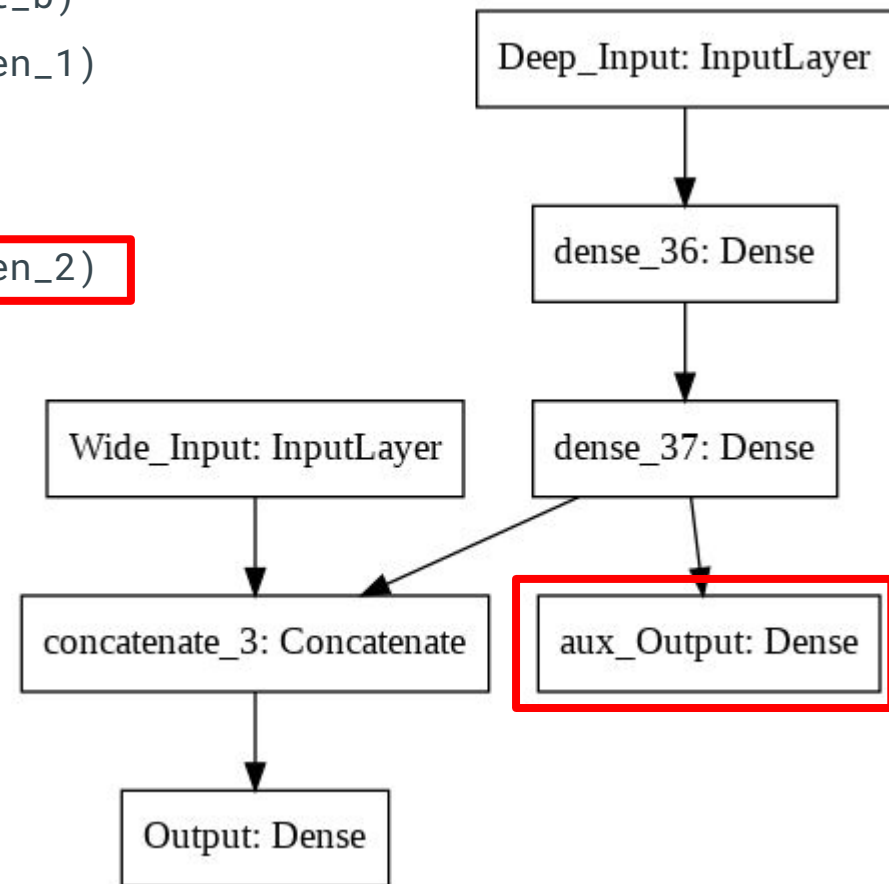
```
input_a = Input(shape=[1], name="Wide_Input")
input_b = Input(shape=[1], name="Deep_Input")
hidden_1 = Dense(30, activation="relu")(input_b)
hidden_2 = Dense(30, activation="relu")(hidden_1)
concat = concatenate([input_a, hidden_2])
output = Dense(1, name="Output")(concat)
model = Model(inputs=[input_a, input_b],
              outputs=[output])
```

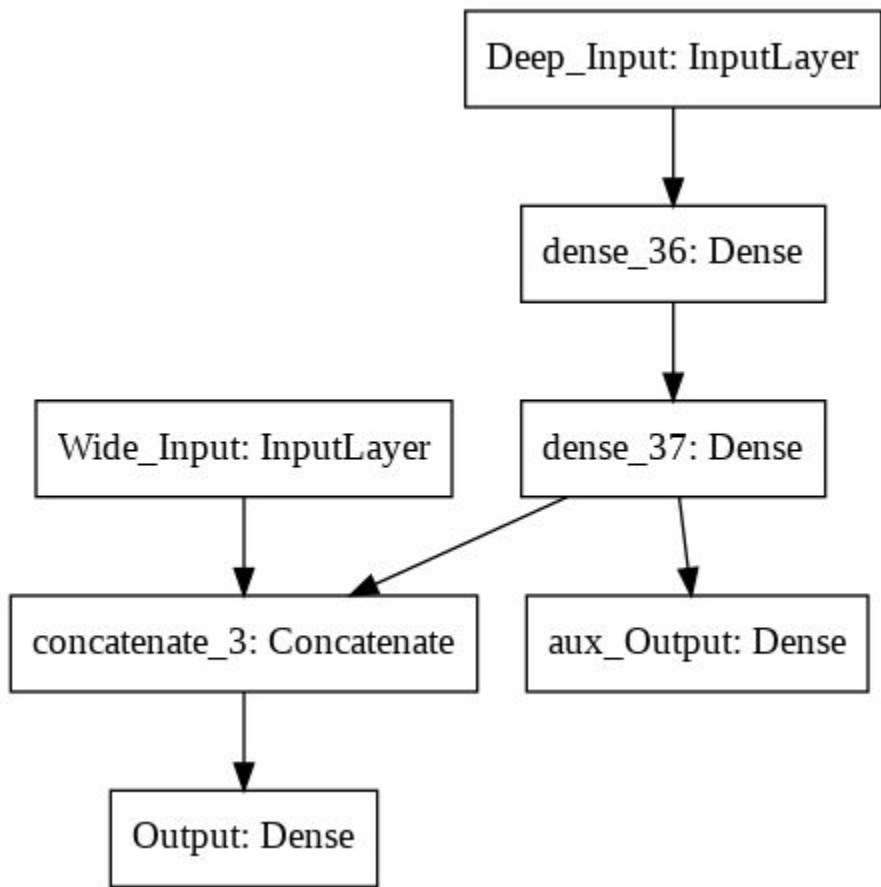


```
input_a = Input(shape=[1], name="Wide_Input")
input_b = Input(shape=[1], name="Deep_Input")
hidden_1 = Dense(30, activation="relu")(input_b)
hidden_2 = Dense(30, activation="relu")(hidden_1)
concat = concatenate([input_a, hidden_2])
output = Dense(1, name="Output")(concat)
model = Model(inputs=[input_a, input_b],
               outputs=[output])
```



```
input_a = Input(shape=[1], name="Wide_Input")
input_b = Input(shape=[1], name="Deep_Input")
hidden_1 = Dense(30, activation="relu")(input_b)
hidden_2 = Dense(30, activation="relu")(hidden_1)
concat = concatenate([input_a, hidden_2])
output = Dense(1, name="Output")(concat)
aux_output = Dense(1, name="aux_Output")(hidden_2)
model = Model(inputs=[input_a, input_b],
              outputs=[output, aux_output])
```





This sort of architecture
can be incorporated in
a class. (python class)


```
class WideAndDeepModel(Model):  
    def __init__(self, units=30, activation='relu', **kwargs):  
        super().__init__(**kwargs)  
        self.hidden1 = Dense(units, activation=activation)  
        self.hidden2 = Dense(units, activation=activation)  
        self.main_output = Dense(1)  
        self.aux_output = Dense(1)
```

```
    def call(self, inputs):  
        input_A, input_B = inputs  
        hidden1 = self.hidden1(input_B)  
        hidden2 = self.hidden2(hidden1)  
        concat = concatenate([input_A, hidden2])  
        main_output = self.main_output(concat)  
        aux_output = self.aux_output(hidden2)  
        return main_output, aux_output
```

Model

core functionality
↓
modules

we should be
definitely
having these two
classes.

```
class WideAndDeepModel(Model):
    def __init__(self, units=30, activation='relu', **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = Dense(units, activation=activation)
        self.hidden2 = Dense(units, activation=activation)
        self.main_output = Dense(1)
        self.aux_output = Dense(1)

    def call(self, inputs):
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output
```

```
class WideAndDeepModel(Model):  
    def __init__(self, units=30, activation='relu', **kwargs):  
        super().__init__(**kwargs)  
        self.hidden1 = Dense(units, activation=activation)  
        self.hidden2 = Dense(units, activation=activation)  
        self.main_output = Dense(1)  
        self.aux_output = Dense(1)  
  
    def call(self, inputs):  
        input_A, input_B = inputs  
        hidden1 = self.hidden1(input_B)  
        hidden2 = self.hidden2(hidden1)  
        concat = concatenate([input_A, hidden2])  
        main_output = self.main_output(concat)  
        aux_output = self.aux_output(hidden2)  
        return main_output, aux_output
```

```
class WideAndDeepModel(Model):
    def __init__(self, units=30, activation='relu', **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = Dense(units, activation=activation)
        self.hidden2 = Dense(units, activation=activation)
        self.main_output = Dense(1)
        self.aux_output = Dense(1)

    def call(self, inputs):
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output
```

```
class WideAndDeepModel(Model):  
    def __init__(self, units=30, activation='relu', **kwargs):  
        super().__init__(**kwargs)  
        self.hidden1 = Dense(units, activation=activation)  
        self.hidden2 = Dense(units, activation=activation)  
        self.main_output = Dense(1)  
        self.aux_output = Dense(1)
```

```
def call(self, inputs):  
    input_A, input_B = inputs  
    hidden1 = self.hidden1(input_B)  
    hidden2 = self.hidden2(hidden1)  
    concat = concatenate([input_A, hidden2])  
    main_output = self.main_output(concat)  
    aux_output = self.aux_output(hidden2)  
    return main_output, aux_output
```

```
class WideAndDeepModel(Model):  
    def __init__(self, units=30, activation='relu', **kwargs):  
        super().__init__(**kwargs)  
        self.hidden1 = Dense(units, activation=activation)  
        self.hidden2 = Dense(units, activation=activation)  
        self.main_output = Dense(1)  
        self.aux_output = Dense(1)
```

```
def call(self, inputs):
```

```
    input_A, input_B = inputs  
    hidden1 = self.hidden1(input_B)  
    hidden2 = self.hidden2(hidden1)  
    concat = concatenate([input_A, hidden2])  
    main_output = self.main_output(concat)  
    aux_output = self.aux_output(hidden2)  
    return main_output, aux_output
```

```
model = WideAndDeepModel()
```

The Model class

Even after creating
custom model, you still have the
properties of
any model
- fit
- evaluate

- Built-in training, evaluation, and prediction loops

e.g., `model.fit()`, `model.evaluate()`, `model.predict()`

- Saving and serialization APIs.

e.g., `model.save()`, `model.save_weights()`

- Summarization and visualization APIs

e.g., `model.summary()`, `tf.keras.utils.plot_model()`

The Model class

- Built-in training, evaluation, and prediction loops

e.g., `model.fit()`, `model.evaluate()`, `model.predict()`

- Saving and serialization APIs.

e.g., `model.save()`, `model.save_weights()`

- Summarization and visualization APIs

e.g., `model.summary()`, `tf.keras.utils.plot_model()`

The Model class

- Built-in training, evaluation, and prediction loops

e.g., `model.fit()`, `model.evaluate()`, `model.predict()`

- Saving and serialization APIs.

e.g., `model.save()`, `model.save_weights()`

- Summarization and visualization APIs

e.g., `model.summary()`, `tf.keras.utils.plot_model()`

The Model class

- Built-in training, evaluation, and prediction loops

e.g., `model.fit()`, `model.evaluate()`, `model.predict()`

- Saving and serialization APIs.

e.g., `model.save()`, `model.save_weights()`

- Summarization and visualization APIs

e.g., `model.summary()`, `tf.keras.utils.plot_model()`

Limitations of Sequential/Functional APIs

- Only suited to models that are Directed Acyclic Graphs of layers

e.g., MobileNet, Inception, etc

- More exotic architectures

e.g., dynamic and recursive networks

This is something we see with the Sequential

← Never loops back.

↙ In this case recursion can be used (or)
the architecture can be played
on the fly

Functional APIs

Limitations of Sequential/Functional APIs

- Only suited to models that are Directed Acyclic Graphs of layers

e.g., MobileNet, Inception, etc

- More exotic architectures

e.g., dynamic and recursive networks

Limitations of Sequential/Functional APIs

- Only suited to models that are Directed Acyclic Graphs of layers

e.g., MobileNet, Inception, etc

- More exotic architectures

e.g., dynamic and recursive networks

Benefits of subclassing models

- Extends how you've been building models
- Continue to use functional and sequential code
- Modular architecture
- Try out experiments quickly
- Control flow in the network

Benefits of subclassing models

- Extends how you've been building models
- Continue to use functional and sequential code
- Modular architecture
- Try out experiments quickly
- Control flow in the network

Benefits of subclassing models

- Extends how you've been building models
- Continue to use functional and sequential code
- Modular architecture
- Try out experiments quickly
- Control flow in the network

Benefits of subclassing models

- Extends how you've been building models
- Continue to use functional and sequential code
- Modular architecture
- Try out experiments quickly
- Control flow in the network

Benefits of subclassing models

- Extends how you've been building models
- Continue to use functional and sequential code
- Modular architecture
- Try out experiments quickly
- Control flow in the network

Benefits of subclassing models

- Extends how you've been building models
- Continue to use functional and sequential code
- Modular architecture
- Try out experiments quickly
- Control flow in the network

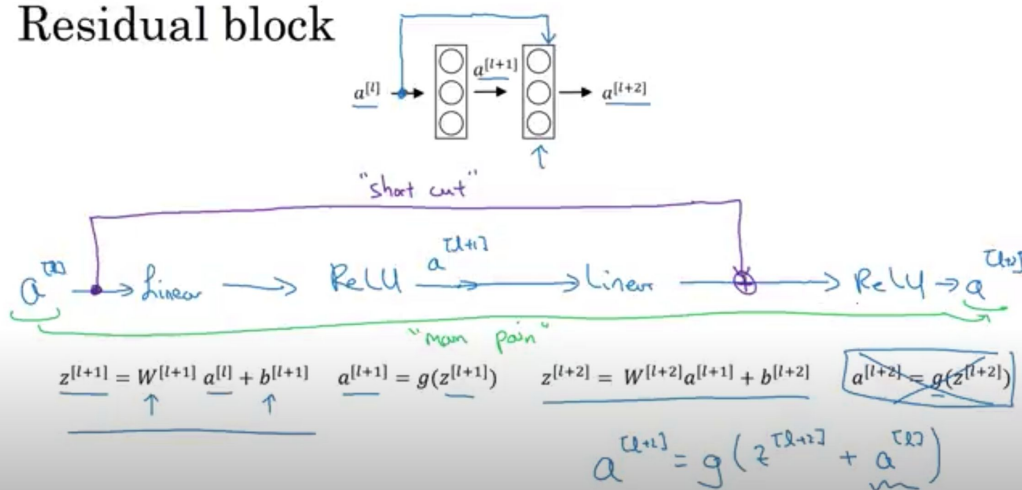
Using the
model's subclassing
you can build
powerful and
flexible model
architecture

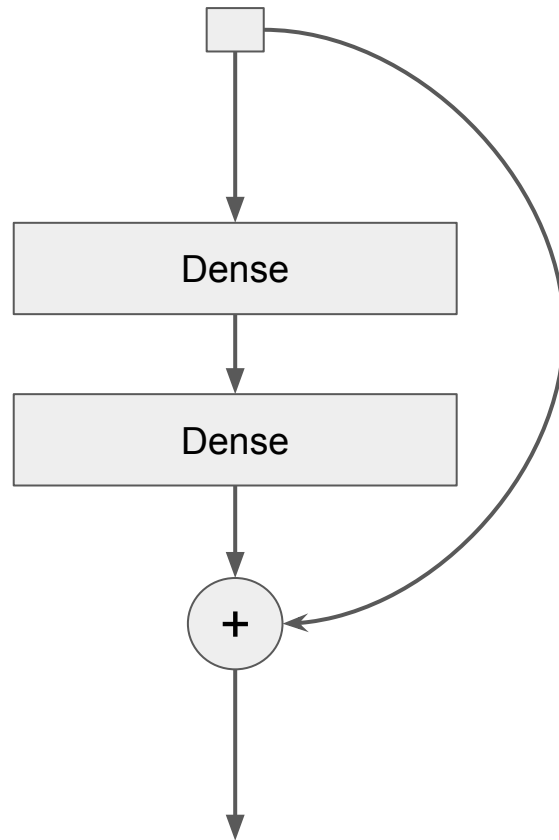
Residual Networks (ResNets)

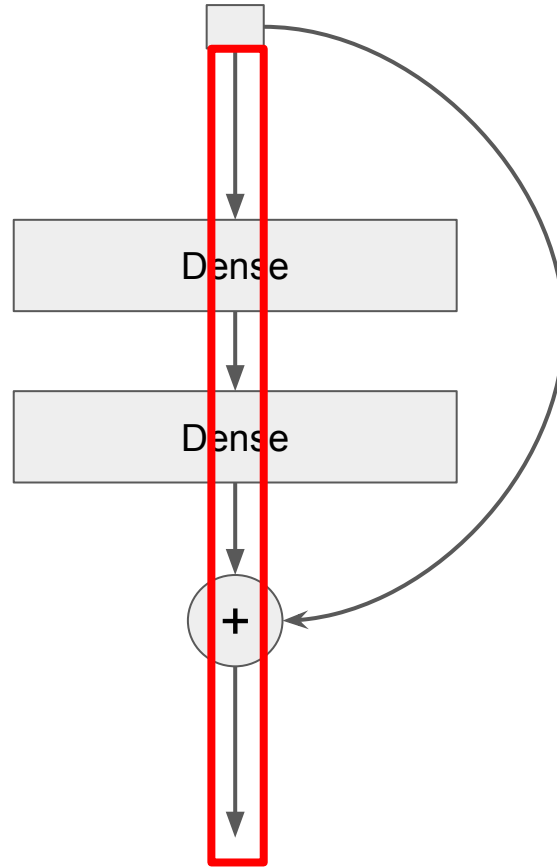
<https://www.coursera.org/lecture/convolutional-neural-networks/resnets-HAhz9>

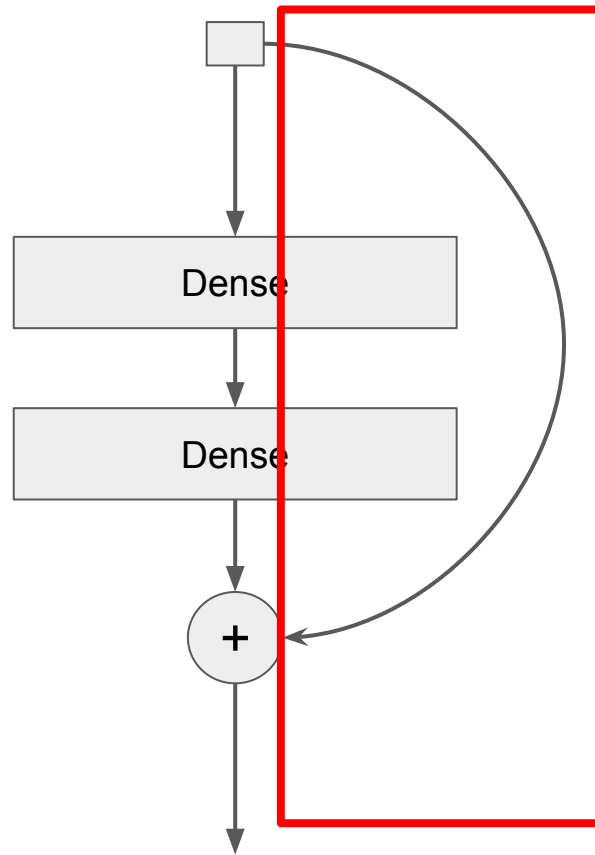


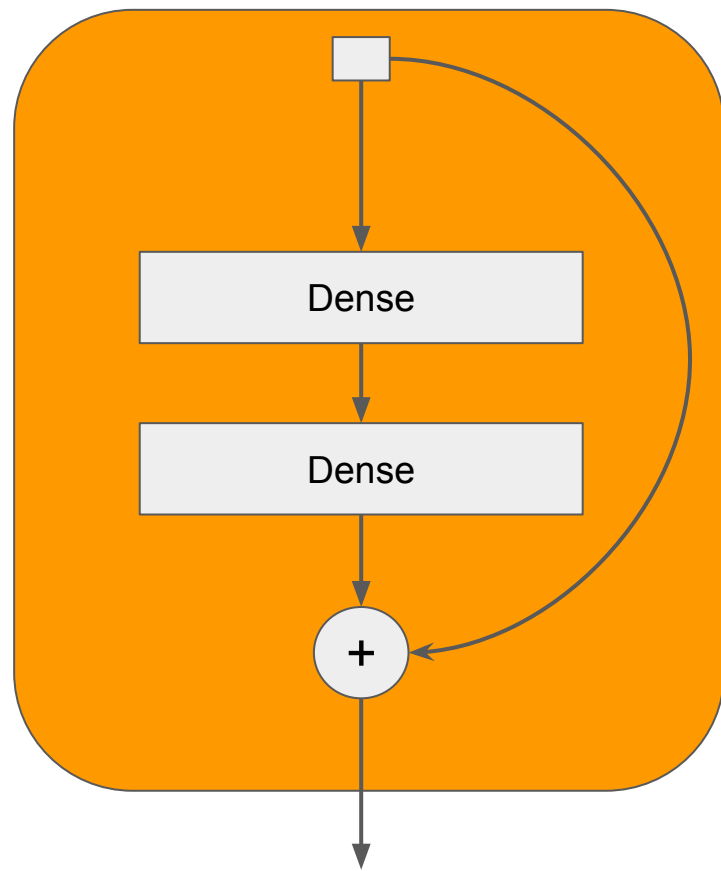
Residual block

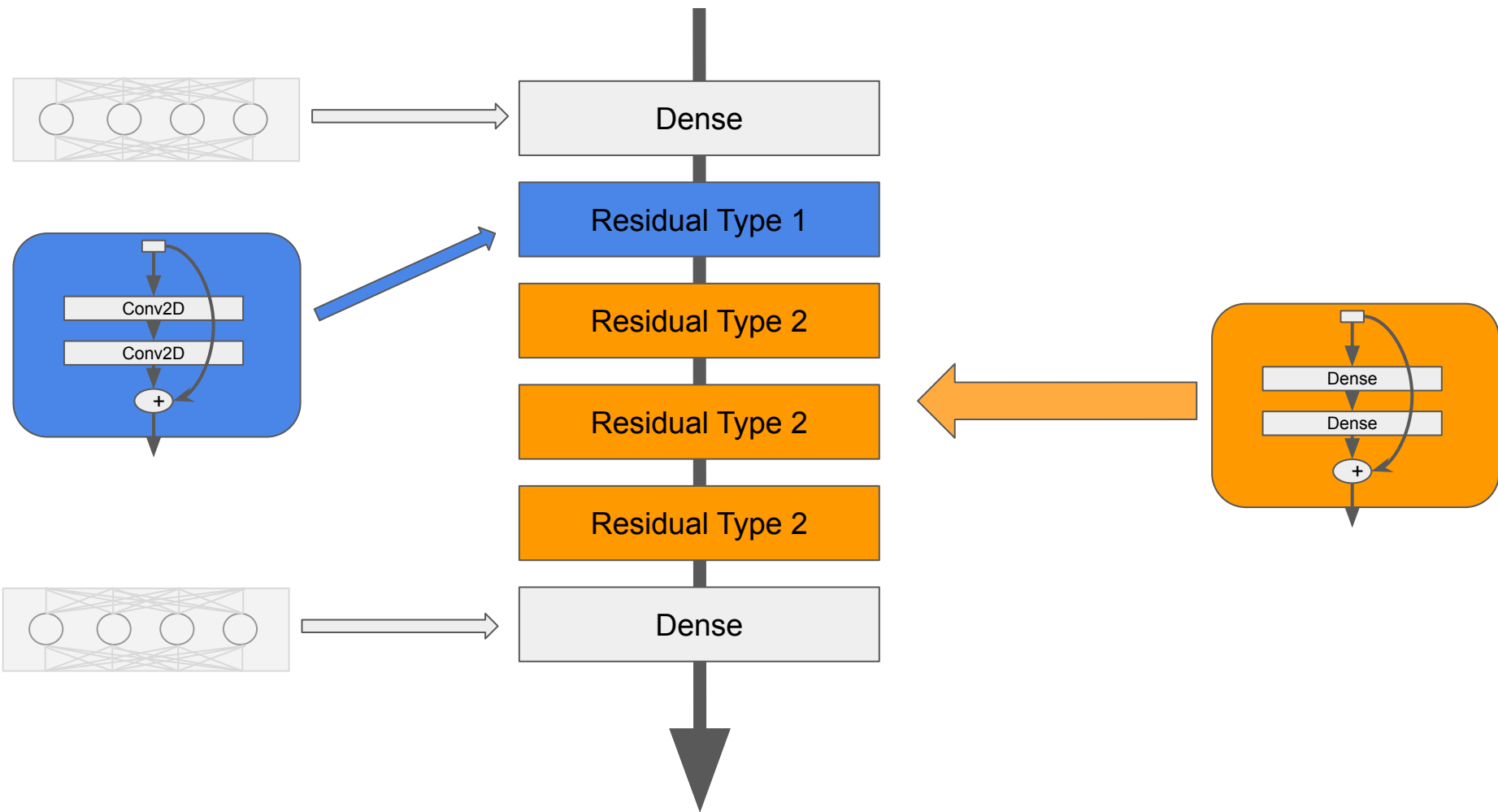


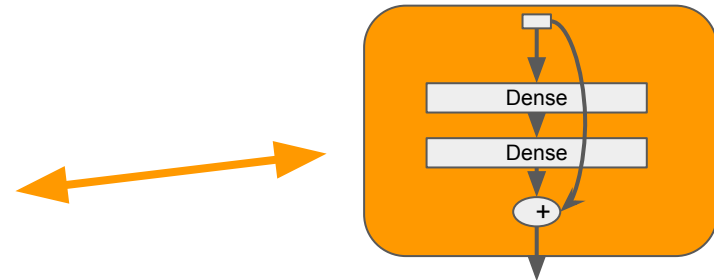
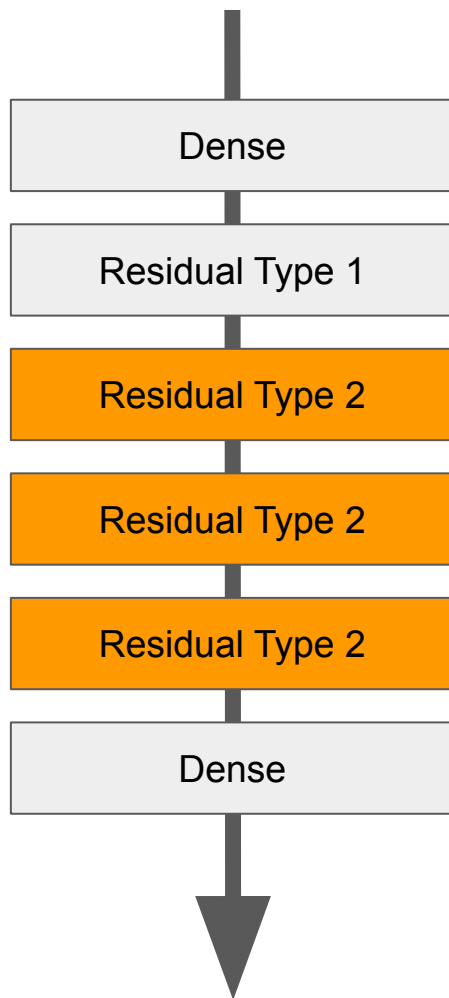


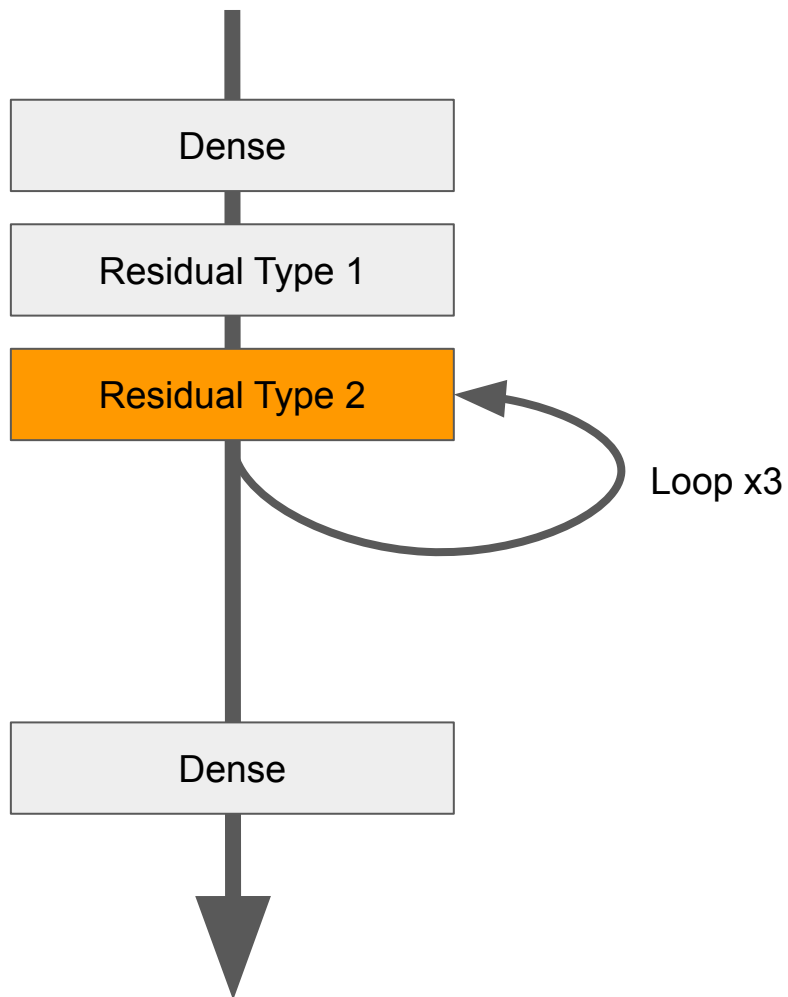












```
class CNNResidual(Layer):
```

```
    def __init__(self, layers, filters, **kwargs):
```

```
        super().__init__(**kwargs)
```

```
        self.hidden = [Conv2D(filters, (3, 3), activation="relu")
```

```
                        for _ in range(layers)]
```

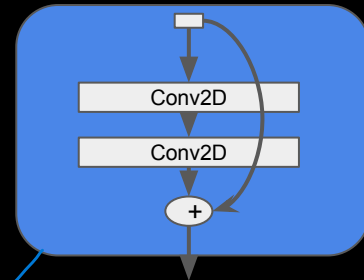
```
    def call(self, inputs):
```

```
        x = inputs
```

```
        for layer in self.hidden:
```

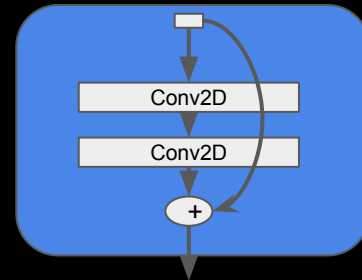
```
            x = layer(x)
```

```
        return inputs + x
```

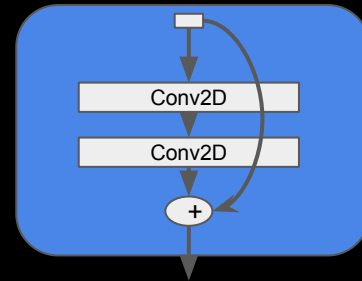


*Here we're trying to create
this architecture*

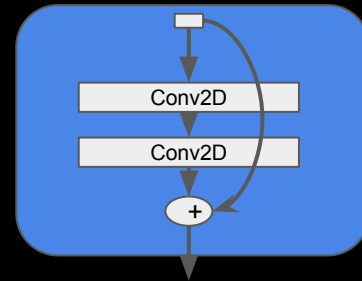
```
class CNNResidual(Layer):  
    def __init__(self, layers, filters, **kwargs):  
        super().__init__(**kwargs)  
        self.hidden = [Conv2D(filters, (3, 3), activation="relu")  
                        for _ in range(layers)]  
  
    def call(self, inputs):  
        x = inputs  
        for layer in self.hidden:  
            x = layer(x)  
        return inputs + x
```



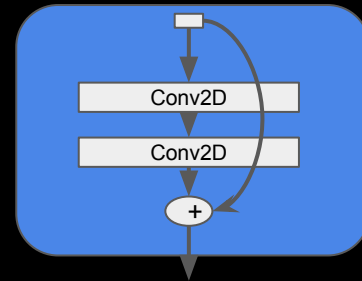
```
class CNNResidual(Layer):  
    def __init__(self, layers, filters, **kwargs):  
        super().__init__(**kwargs)  
        self.hidden = [Conv2D(filters, (3, 3), activation="relu")  
                        for _ in range(layers)]  
  
    def call(self, inputs):  
        x = inputs  
        for layer in self.hidden:  
            x = layer(x)  
        return inputs + x
```



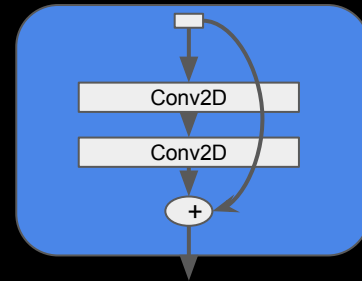
```
class CNNResidual(Layer):  
    def __init__(self, layers, filters, **kwargs):  
        super().__init__(**kwargs)  
        self.hidden = [Conv2D(filters, (3, 3), activation="relu")  
                        for _ in range(layers)]  
  
    def call(self, inputs):  
        x = inputs  
        for layer in self.hidden:  
            x = layer(x)  
        return inputs + x
```



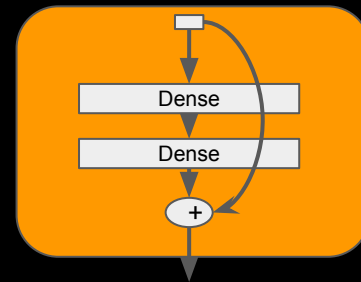

```
class CNNResidual(Layer):  
    def __init__(self, layers, filters, **kwargs):  
        super().__init__(**kwargs)  
        self.hidden = [Conv2D(filters, (3, 3), activation="relu")  
                        for _ in range(layers)]  
  
    def call(self, inputs):  
        x = inputs  
        for layer in self.hidden:  
            x = layer(x)  
        return inputs + x
```



```
class CNNResidual(Layer):  
    def __init__(self, layers, filters, **kwargs):  
        super().__init__(**kwargs)  
        self.hidden = [Conv2D(filters, (3, 3), activation="relu")  
                        for _ in range(layers)]  
  
    def call(self, inputs):  
        x = inputs  
        for layer in self.hidden:  
            x = layer(x)  
        return inputs + x
```



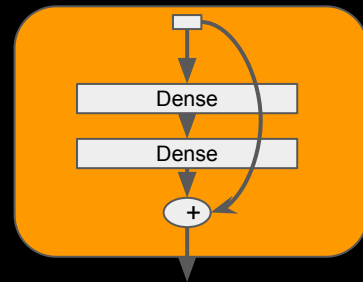
```
class DNNResidual(Layer):  
    def __init__(self, layers, neurons, **kwargs):  
        super().__init__(**kwargs)  
        self.hidden = [Dense(neurons, activation="relu")  
                        for _ in range(layers)]  
  
    def call(self, inputs):  
        x = inputs  
        for layer in self.hidden:  
            x = layer(x)  
        return inputs + x
```



```
class DNNResidual(Layer):
```

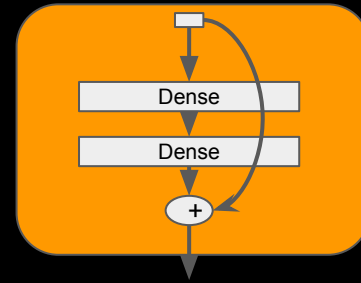
```
def __init__(self, layers, neurons, **kwargs):  
    super().__init__(**kwargs)  
    self.hidden = [Dense(neurons, activation="relu")  
                    for _ in range(layers)]
```

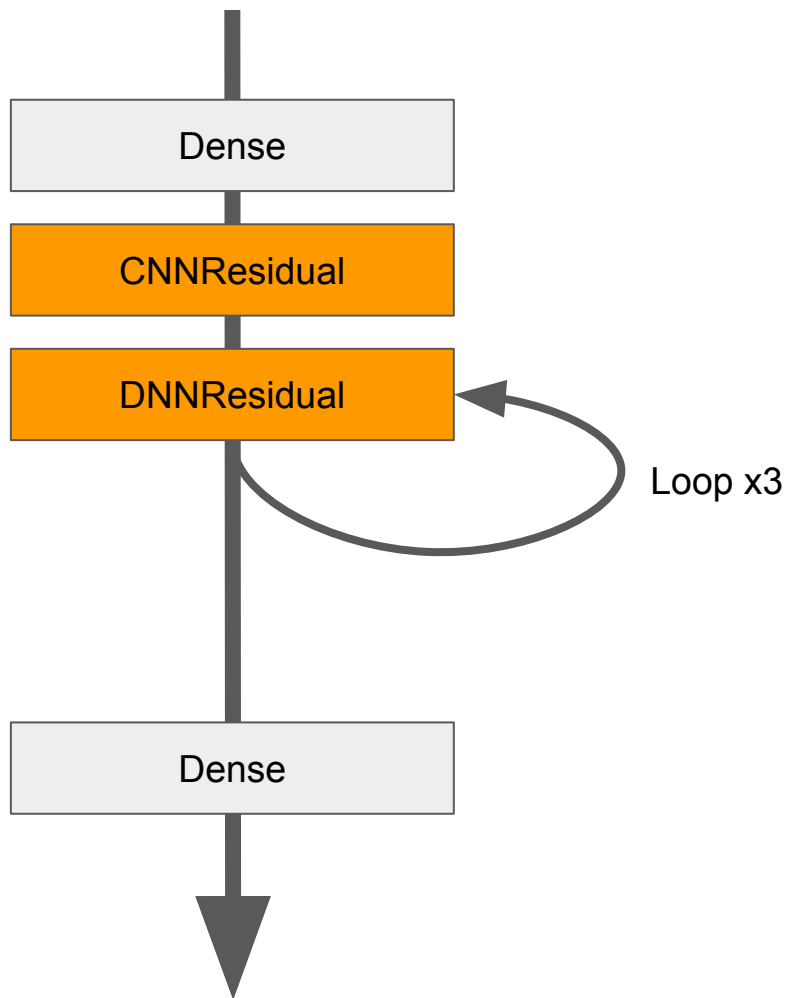
```
def call(self, inputs):  
    x = inputs  
    for layer in self.hidden:  
        x = layer(x)  
    return inputs + x
```



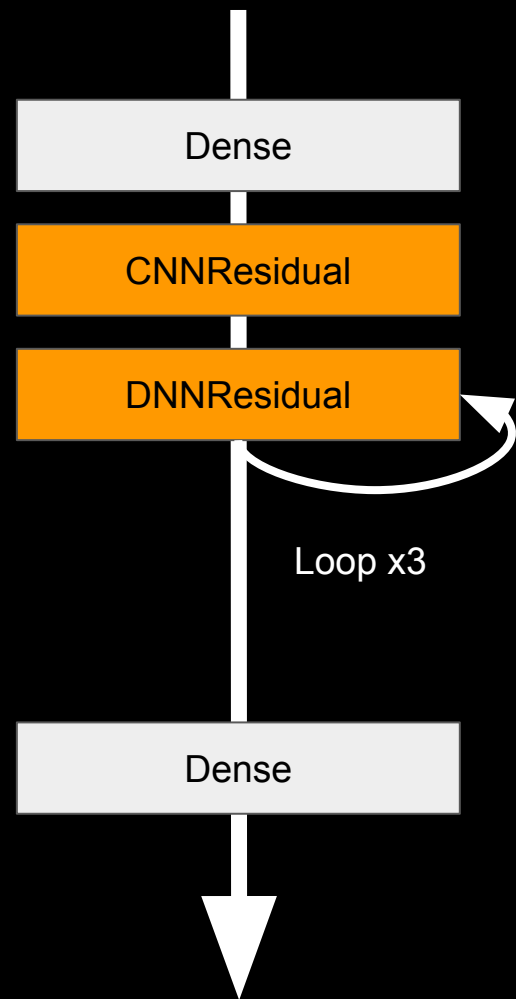
```
class DNNResidual(Layer):  
    def __init__(self, layers, neurons, **kwargs):  
        super().__init__(**kwargs)  
        self.hidden = [Dense(neurons, activation="relu")  
                        for _ in range(layers)]
```

```
def call(self, inputs):  
    x = inputs  
    for layer in self.hidden:  
        x = layer(x)  
    return inputs + x
```

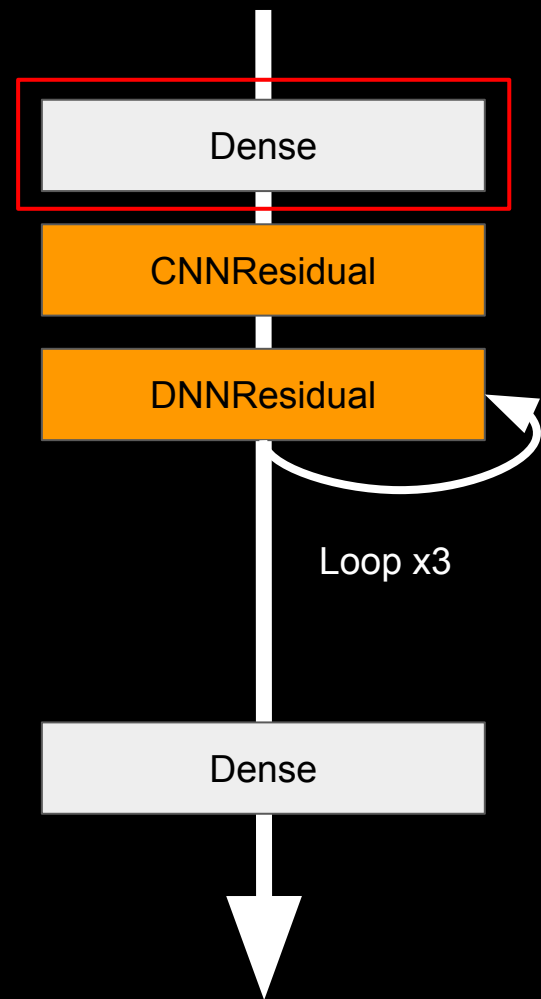




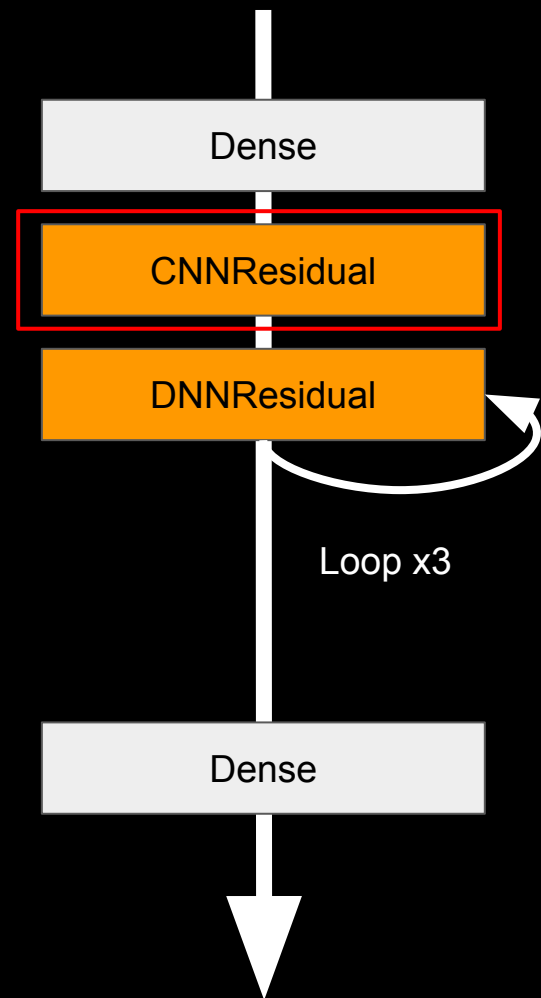
```
class MyResidual(Model):  
    def __init__(self, **kwargs):  
        self.hidden1 = Dense(30, activation="relu")  
        self.block1 = CNNResidual(2, 32)  
        self.block2 = DNNResidual(2, 64)  
        self.out = Dense(1)  
  
    def call(self, inputs):  
        x = self.hidden1(inputs)  
        x = self.block1(x)  
        for _ in range(1, 4):#this will run 3 times  
            x = self.block2(x)  
        return self.out(x)
```



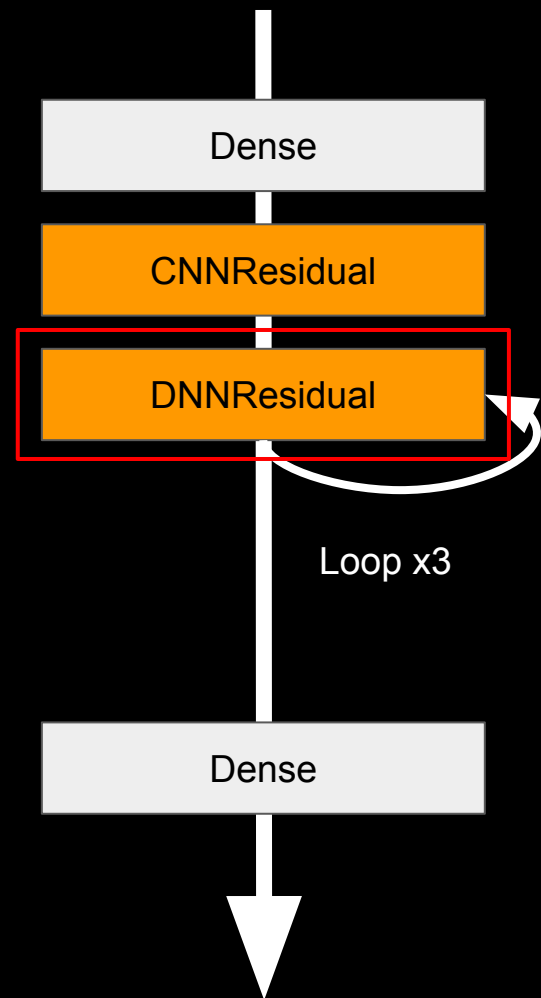
```
class MyResidual(Model):  
    def __init__(self, **kwargs):  
        self.hidden1 = Dense(30, activation="relu")  
        self.block1 = CNNResidual(2, 32)  
        self.block2 = DNNResidual(2, 64)  
        self.out = Dense(1)  
  
    def call(self, inputs):  
        x = self.hidden1(inputs)  
        x = self.block1(x)  
        for _ in range(1, 4):#this will run 3 times  
            x = self.block2(x)  
        return self.out(x)
```



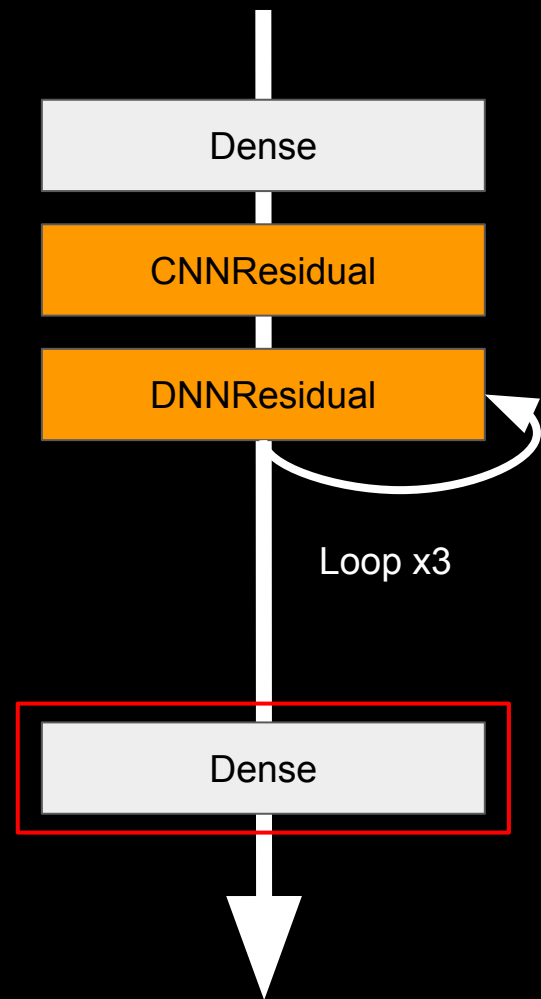

```
class MyResidual(Model):  
    def __init__(self, **kwargs):  
        self.hidden1 = Dense(30, activation="relu")  
        self.block1 = CNNResidual(2, 32)  
        self.block2 = DNNResidual(2, 64)  
        self.out = Dense(1)  
  
    def call(self, inputs):  
        x = self.hidden1(inputs)  
        x = self.block1(x)  
        for _ in range(1, 4):#this will run 3 times  
            x = self.block2(x)  
        return self.out(x)
```



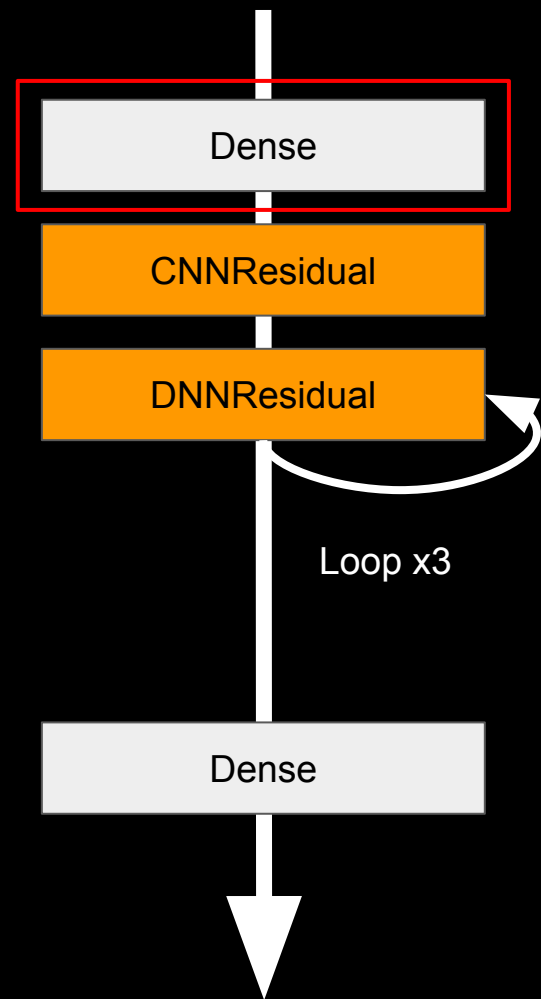
```
class MyResidual(Model):  
    def __init__(self, **kwargs):  
        self.hidden1 = Dense(30, activation="relu")  
        self.block1 = CNNResidual(2, 32)  
        self.block2 = DNNResidual(2, 64)  
        self.out = Dense(1)  
  
    def call(self, inputs):  
        x = self.hidden1(inputs)  
        x = self.block1(x)  
        for _ in range(1, 4):#this will run 3 times  
            x = self.block2(x)  
        return self.out(x)
```



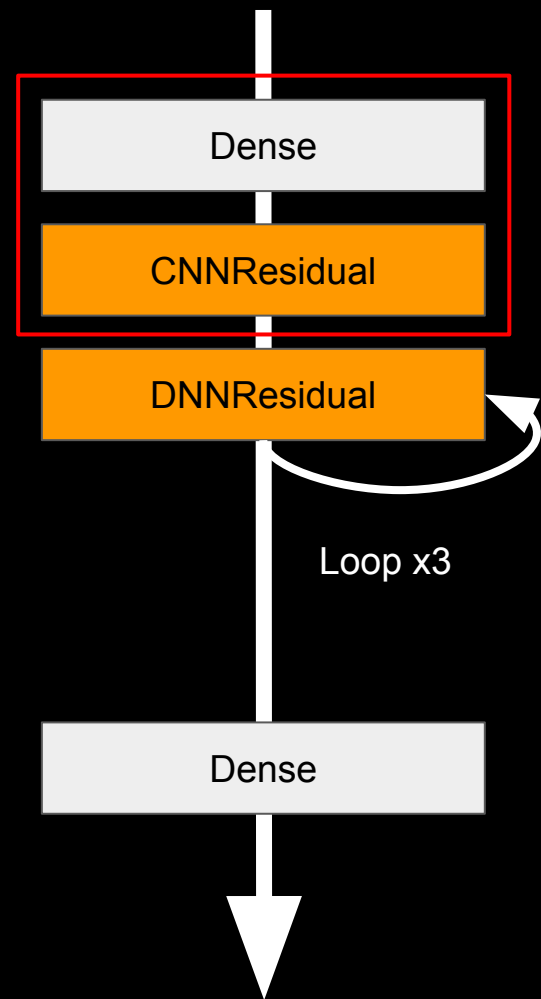
```
class MyResidual(Model):  
    def __init__(self, **kwargs):  
        self.hidden1 = Dense(30, activation="relu")  
        self.block1 = CNNResidual(2, 32)  
        self.block2 = DNNResidual(2, 64)  
        self.out = Dense(1)  
  
    def call(self, inputs):  
        x = self.hidden1(inputs)  
        x = self.block1(x)  
        for _ in range(1, 4):#this will run 3 times  
            x = self.block2(x)  
        return self.out(x)
```



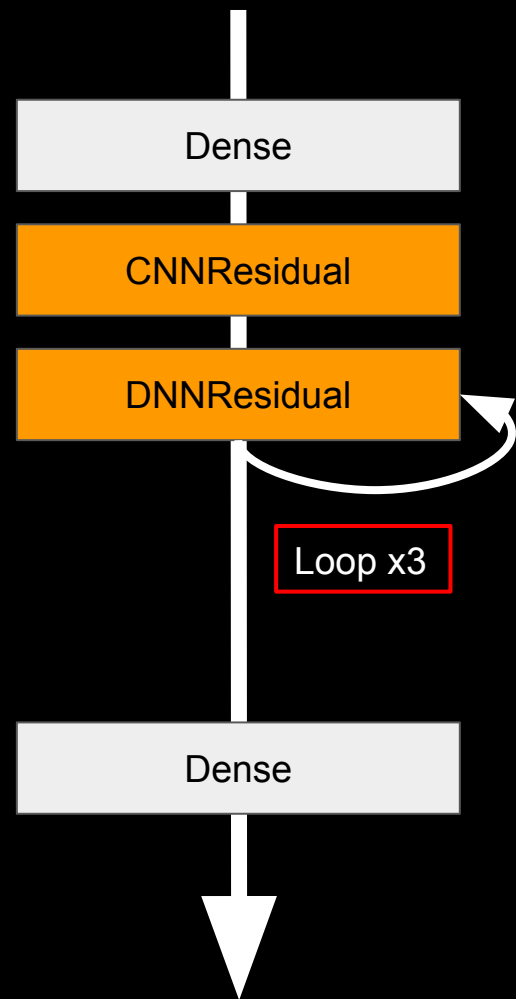
```
class MyResidual(Model):  
    def __init__(self, **kwargs):  
        self.hidden1 = Dense(30, activation="relu")  
        self.block1 = CNNResidual(2, 32)  
        self.block2 = DNNResidual(2, 64)  
        self.out = Dense(1)  
  
    def call(self, inputs):  
        x = self.hidden1(inputs)  
        x = self.block1(x)  
        for _ in range(1, 4):#this will run 3 times  
            x = self.block2(x)  
        return self.out(x)
```



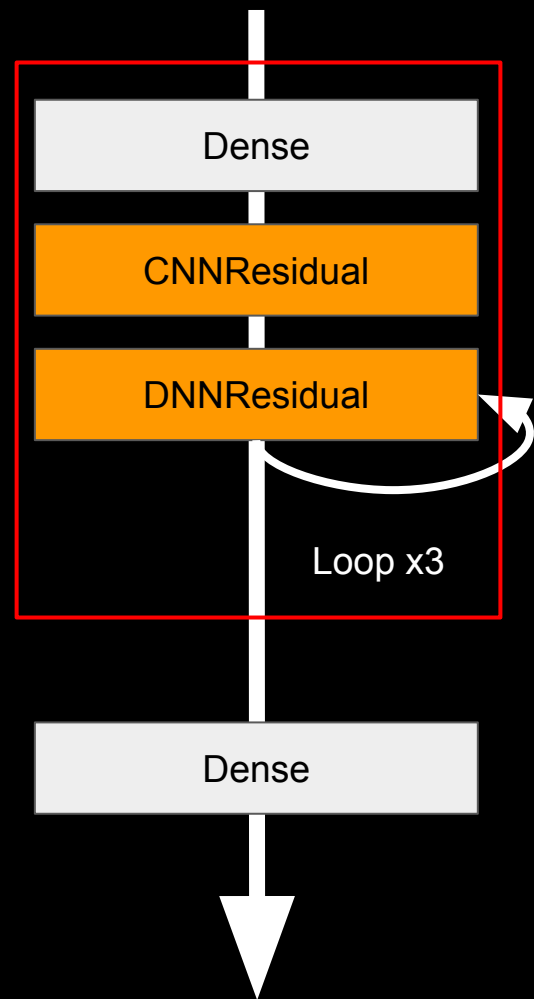
```
class MyResidual(Model):  
    def __init__(self, **kwargs):  
        self.hidden1 = Dense(30, activation="relu")  
        self.block1 = CNNResidual(2, 32)  
        self.block2 = DNNResidual(2, 64)  
        self.out = Dense(1)  
  
    def call(self, inputs):  
        x = self.hidden1(inputs)  
        x = self.block1(x)  
        for _ in range(1, 4):#this will run 3 times  
            x = self.block2(x)  
        return self.out(x)
```



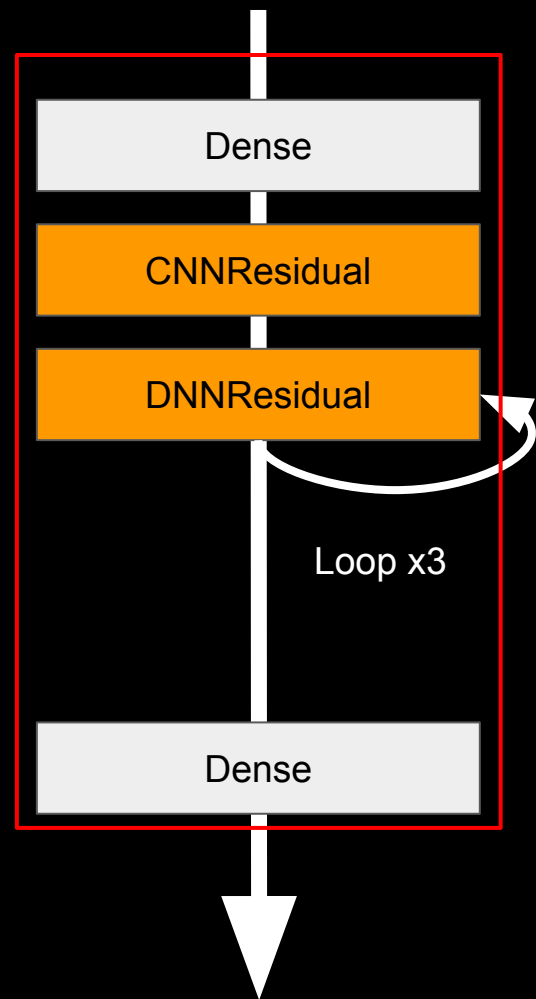
```
class MyResidual(Model):  
    def __init__(self, **kwargs):  
        self.hidden1 = Dense(30, activation="relu")  
        self.block1 = CNNResidual(2, 32)  
        self.block2 = DNNResidual(2, 64)  
        self.out = Dense(1)  
  
    def call(self, inputs):  
        x = self.hidden1(inputs)  
        x = self.block1(x)  
        for _ in range(1, 4):#this will run 3 times  
            x = self.block2(x)  
        return self.out(x)
```



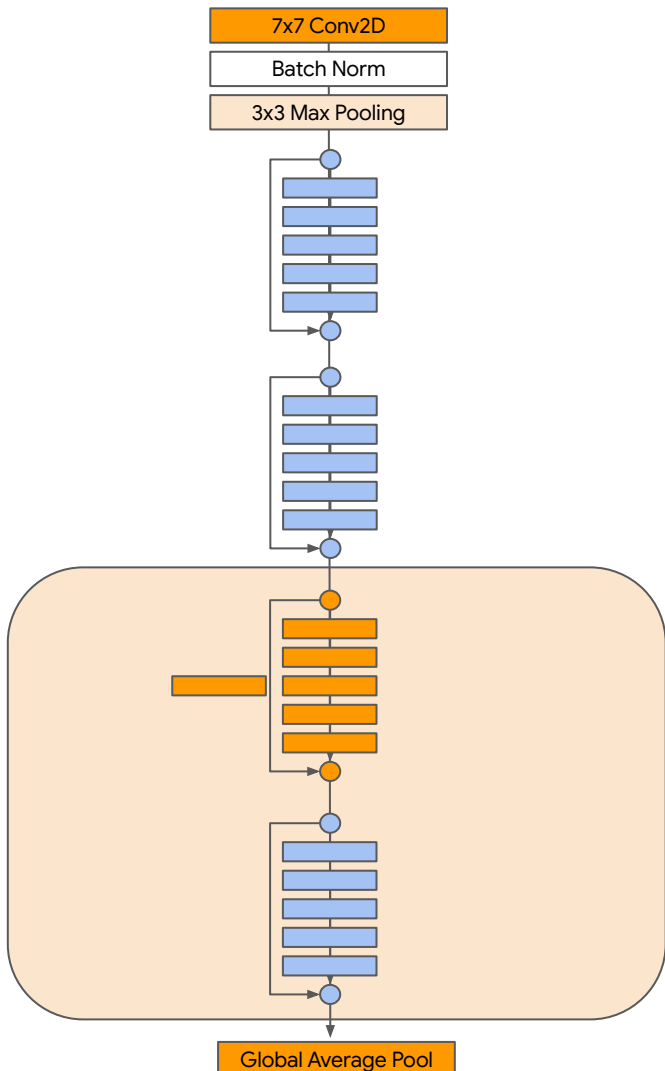
```
class MyResidual(Model):  
    def __init__(self, **kwargs):  
        self.hidden1 = Dense(30, activation="relu")  
        self.block1 = CNNResidual(2, 32)  
        self.block2 = DNNResidual(2, 64)  
        self.out = Dense(1)  
  
    def call(self, inputs):  
        x = self.hidden1(inputs)  
        x = self.block1(x)  
        for _ in range(1, 4):#this will run 3 times  
            x = self.block2(x)  
        return self.out(x)
```



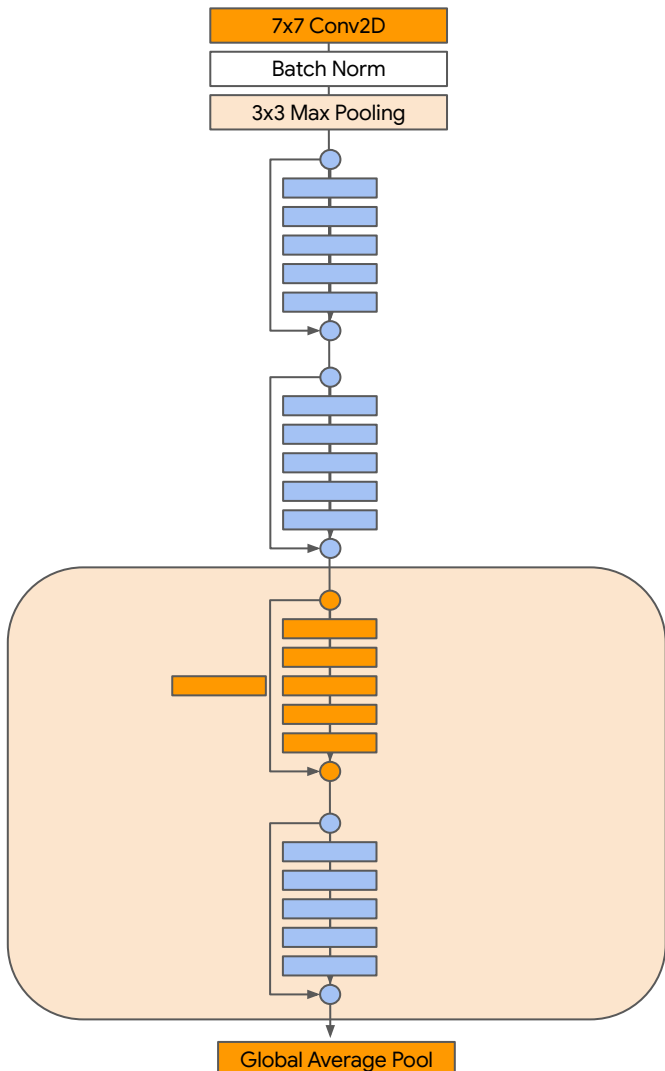
```
class MyResidual(Model):  
    def __init__(self, **kwargs):  
        self.hidden1 = Dense(30, activation="relu")  
        self.block1 = CNNResidual(2, 32)  
        self.block2 = DNNResidual(2, 64)  
        self.out = Dense(1)  
  
    def call(self, inputs):  
        x = self.hidden1(inputs)  
        x = self.block1(x)  
        for _ in range(1, 4):#this will run 3 times  
            x = self.block2(x)  
        return self.out(x)
```



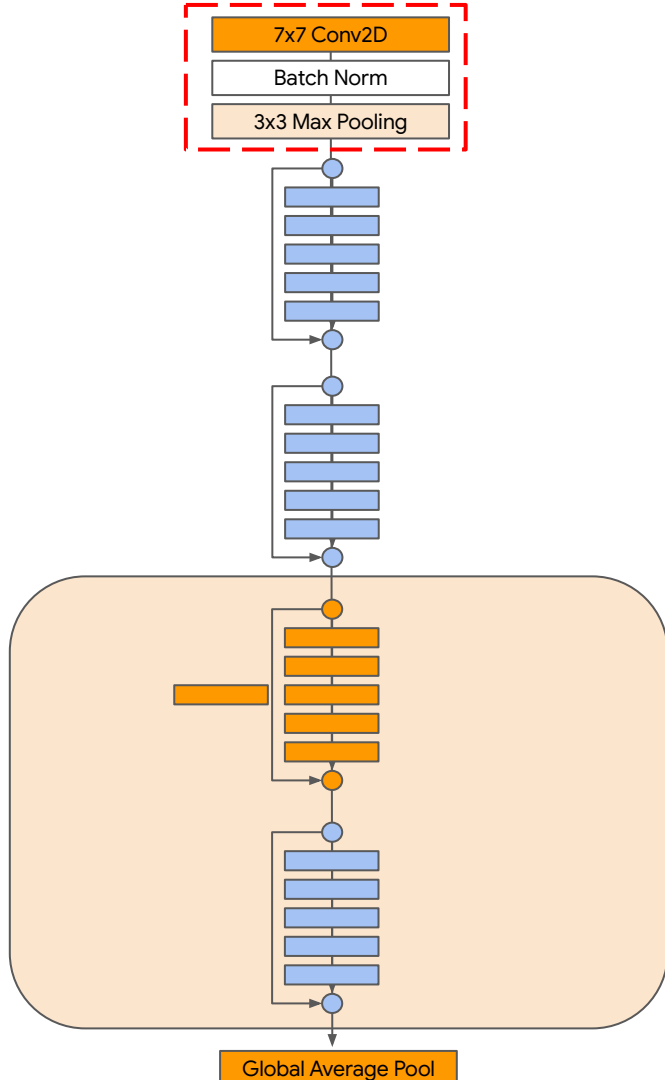
ResNet18



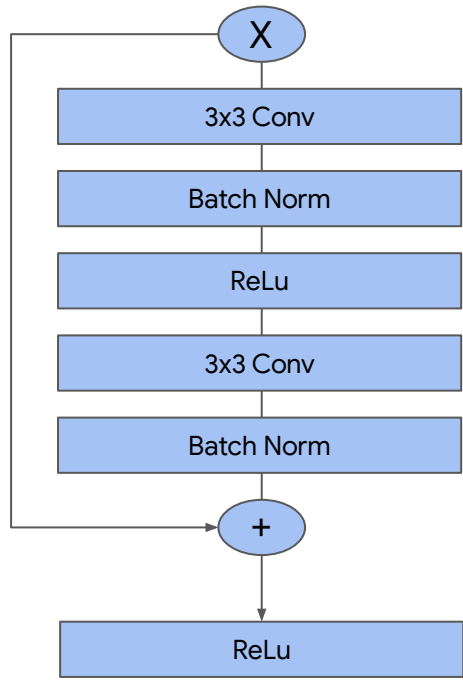
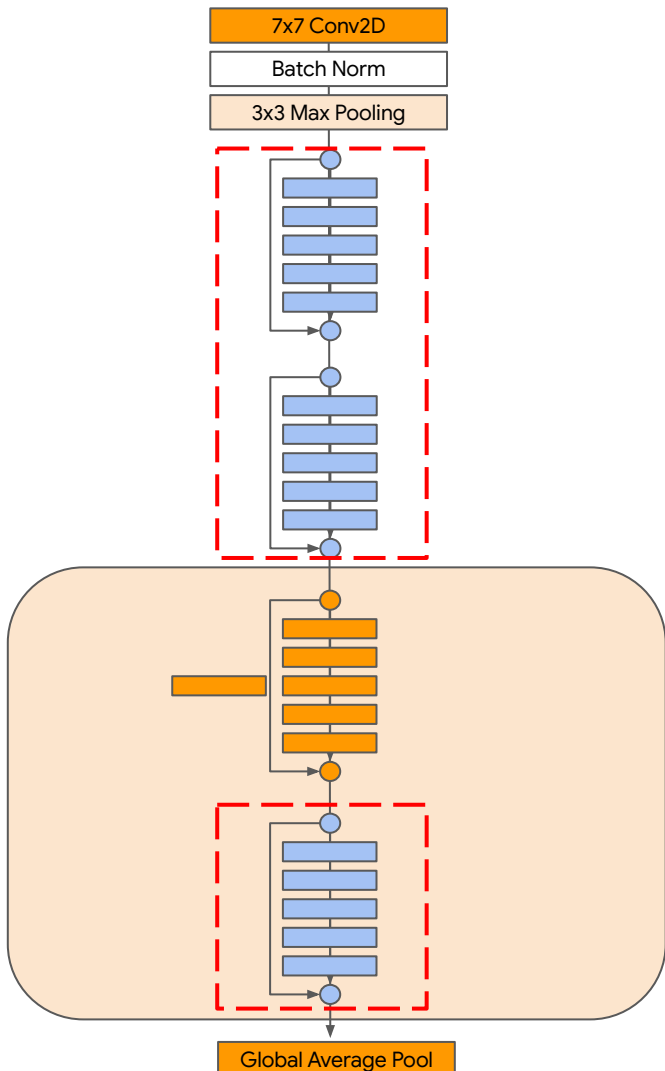
ResNet18



ResNet18

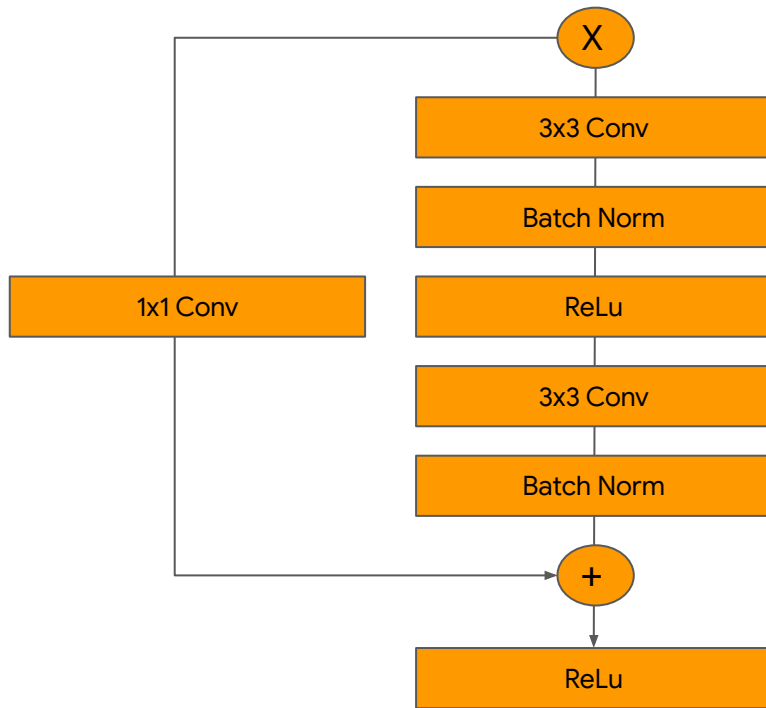
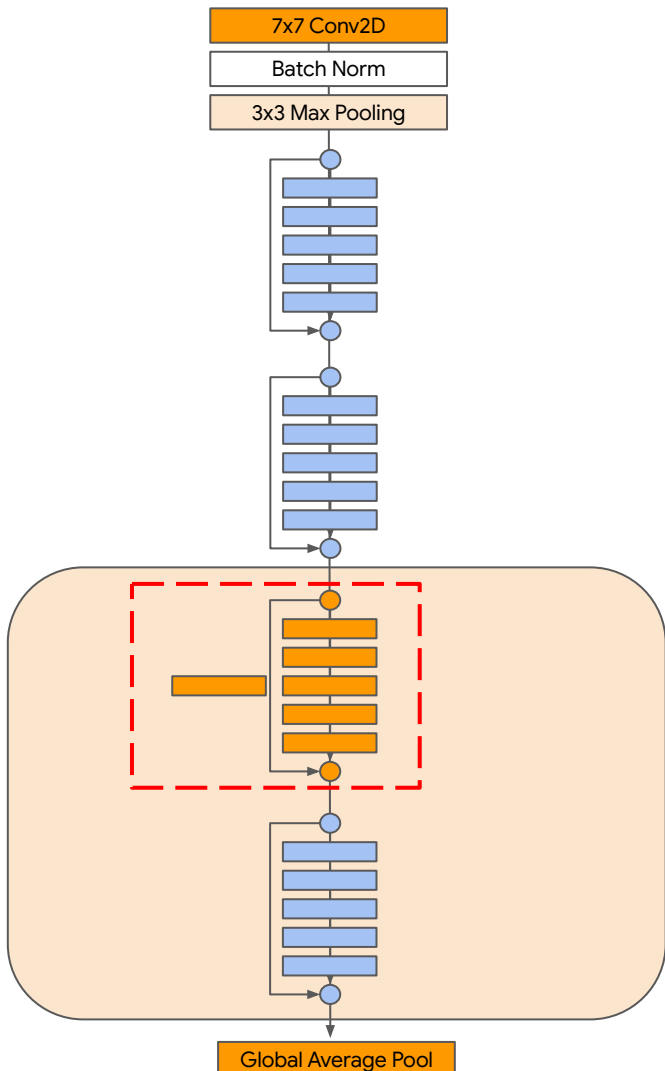


ResNet18



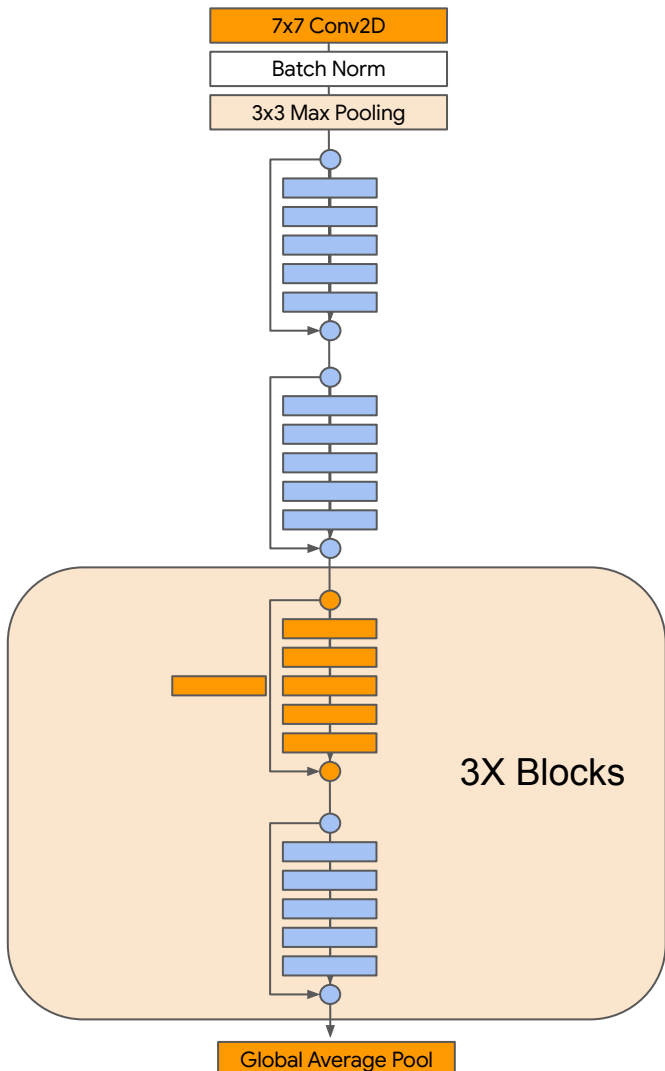
"Identity ResNet Block"

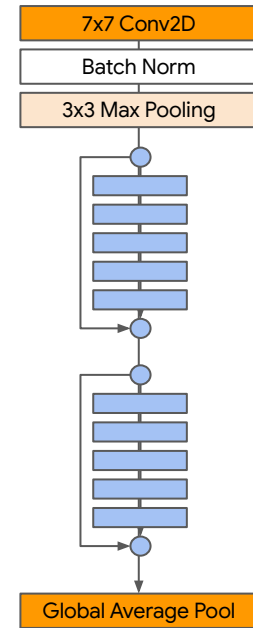
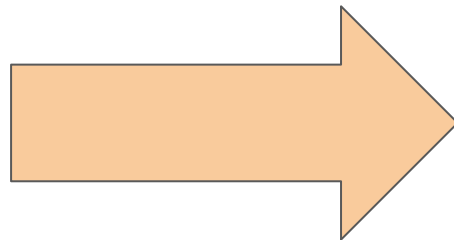
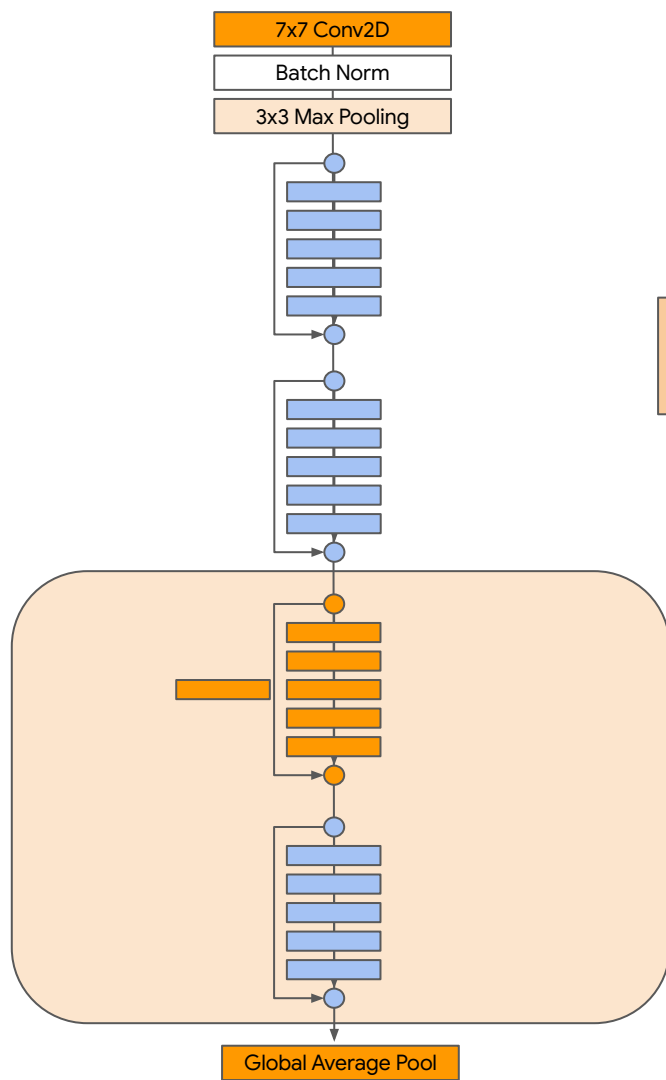
ResNet18



“Identity ResNet Block with 1x1 Convolution”

ResNet18





Mini ResNet

```

class IdentityBlock(tf.keras.Model):
    def __init__(self, filters, kernel_size):
        super(IdentityBlock, self).__init__(name='')

        self.conv1 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
        self.bn1 = tf.keras.layers.BatchNormalization()

        self.conv2 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
        self.bn2 = tf.keras.layers.BatchNormalization()

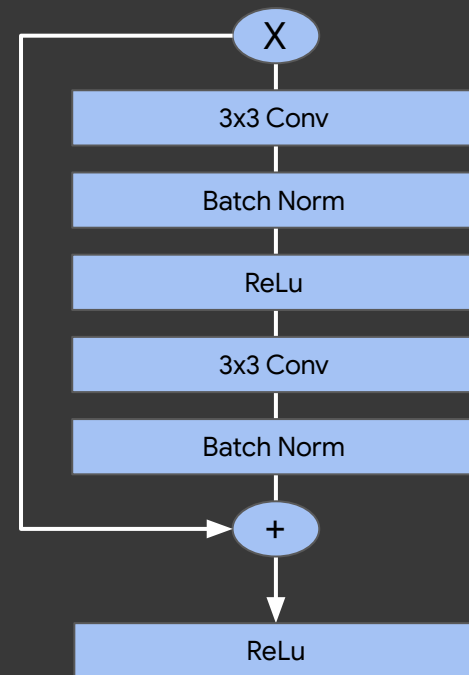
        self.act = tf.keras.layers.Activation('relu')
        self.add = tf.keras.layers.Add()

    def call(self, input_tensor):
        x = self.conv1(input_tensor)
        x = self.bn1(x)
        x = self.act(x)

        x = self.conv2(x)
        x = self.bn2(x)
        x = self.act(x)

        x = self.add([x, input_tensor])
        x = self.act(x)
        return x

```




```

class IdentityBlock(tf.keras.Model):
    def __init__(self, filters, kernel_size):
        super(IdentityBlock, self).__init__(name='')

        self.conv1 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
        self.bn1 = tf.keras.layers.BatchNormalization()

        self.conv2 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
        self.bn2 = tf.keras.layers.BatchNormalization()

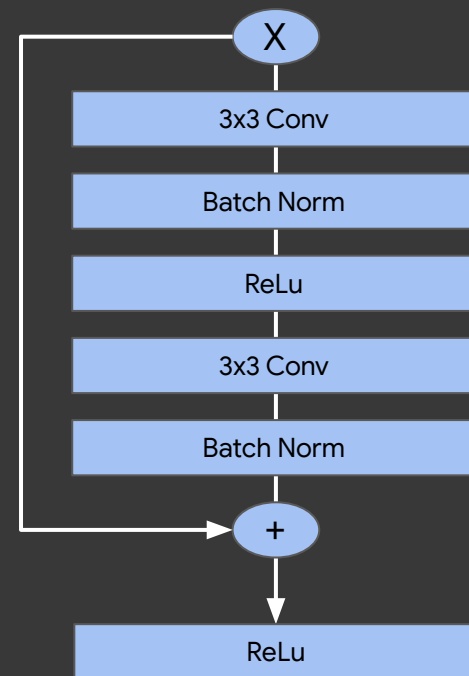
        self.act = tf.keras.layers.Activation('relu')
        self.add = tf.keras.layers.Add()

    def call(self, input_tensor):
        x = self.conv1(input_tensor)
        x = self.bn1(x)
        x = self.act(x)

        x = self.conv2(x)
        x = self.bn2(x)
        x = self.act(x)

        x = self.add([x, input_tensor])
        x = self.act(x)
        return x

```



```
class IdentityBlock(tf.keras.Model):
    def __init__(self, filters, kernel_size):
        super(IdentityBlock, self).__init__(name='')
```

```
self.conv1 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
self.bn1 = tf.keras.layers.BatchNormalization()
```

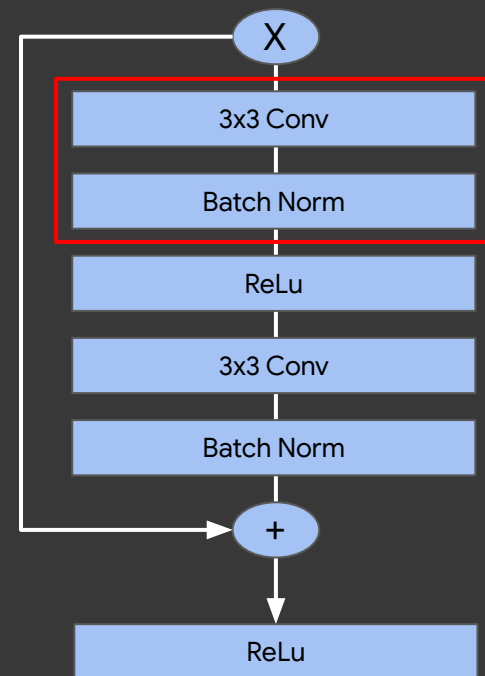
```
self.conv2 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
self.bn2 = tf.keras.layers.BatchNormalization()
```

```
self.act = tf.keras.layers.Activation('relu')
self.add = tf.keras.layers.Add()
```

```
def call(self, input_tensor):
    x = self.conv1(input_tensor)
    x = self.bn1(x)
    x = self.act(x)

    x = self.conv2(x)
    x = self.bn2(x)
    x = self.act(x)

    x = self.add([x, input_tensor])
    x = self.act(x)
    return x
```



```
class IdentityBlock(tf.keras.Model):
    def __init__(self, filters, kernel_size):
        super(IdentityBlock, self).__init__(name='')

        self.conv1 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
        self.bn1 = tf.keras.layers.BatchNormalization()
```

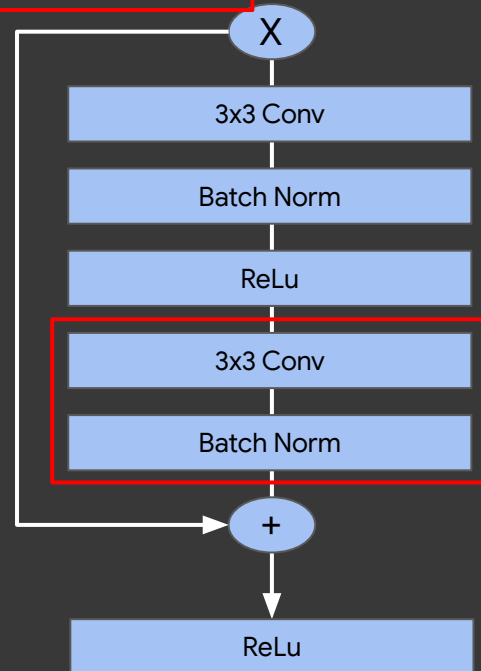
```
self.conv2 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
self.bn2 = tf.keras.layers.BatchNormalization()
```

```
self.act = tf.keras.layers.Activation('relu')
self.add = tf.keras.layers.Add()
```

```
def call(self, input_tensor):
    x = self.conv1(input_tensor)
    x = self.bn1(x)
    x = self.act(x)

    x = self.conv2(x)
    x = self.bn2(x)
    x = self.act(x)

    x = self.add([x, input_tensor])
    x = self.act(x)
    return x
```



```
class IdentityBlock(tf.keras.Model):
    def __init__(self, filters, kernel_size):
        super(IdentityBlock, self).__init__(name='')

        self.conv1 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
        self.bn1 = tf.keras.layers.BatchNormalization()

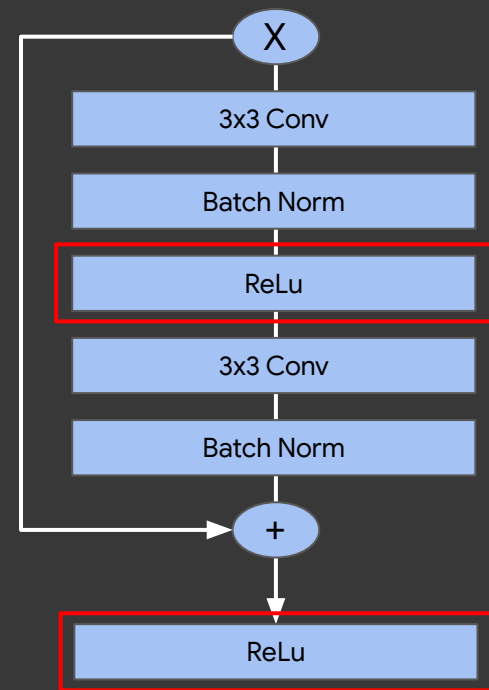
        self.conv2 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
        self.bn2 = tf.keras.layers.BatchNormalization()

        self.act = tf.keras.layers.Activation('relu')
        self.add = tf.keras.layers.Add()

    def call(self, input_tensor):
        x = self.conv1(input_tensor)
        x = self.bn1(x)
        x = self.act(x)

        x = self.conv2(x)
        x = self.bn2(x)
        x = self.act(x)

        x = self.add([x, input_tensor])
        x = self.act(x)
        return x
```



```

class IdentityBlock(tf.keras.Model):
    def __init__(self, filters, kernel_size):
        super(IdentityBlock, self).__init__(name='')

        self.conv1 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
        self.bn1 = tf.keras.layers.BatchNormalization()

        self.conv2 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
        self.bn2 = tf.keras.layers.BatchNormalization()

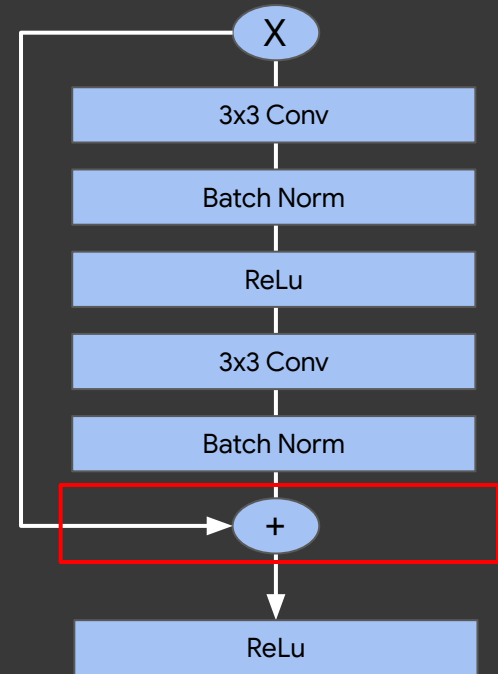
        self.act = tf.keras.layers.Activation('relu')
        self.add = tf.keras.layers.Add()

    def call(self, input_tensor):
        x = self.conv1(input_tensor)
        x = self.bn1(x)
        x = self.act(x)

        x = self.conv2(x)
        x = self.bn2(x)
        x = self.act(x)

        x = self.add([x, input_tensor])
        x = self.act(x)
        return x

```



```

class IdentityBlock(tf.keras.Model):
    def __init__(self, filters, kernel_size):
        super(IdentityBlock, self).__init__(name='')

        self.conv1 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
        self.bn1 = tf.keras.layers.BatchNormalization()

        self.conv2 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
        self.bn2 = tf.keras.layers.BatchNormalization()

        self.act = tf.keras.layers.Activation('relu')
        self.add = tf.keras.layers.Add()

```

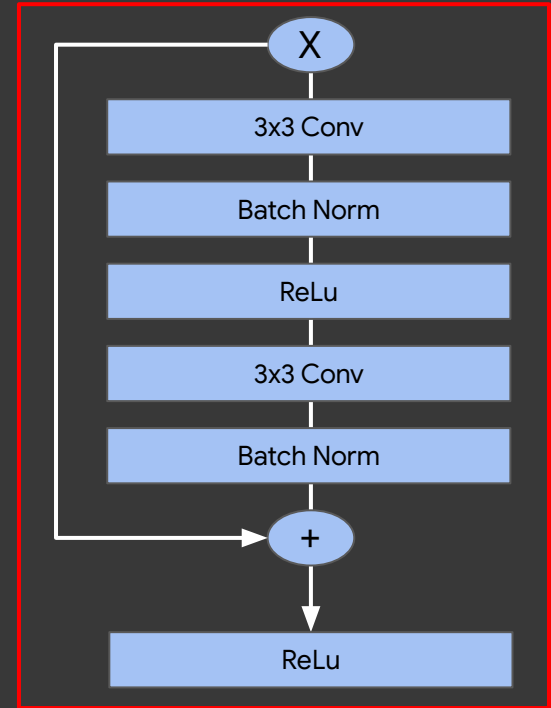
```

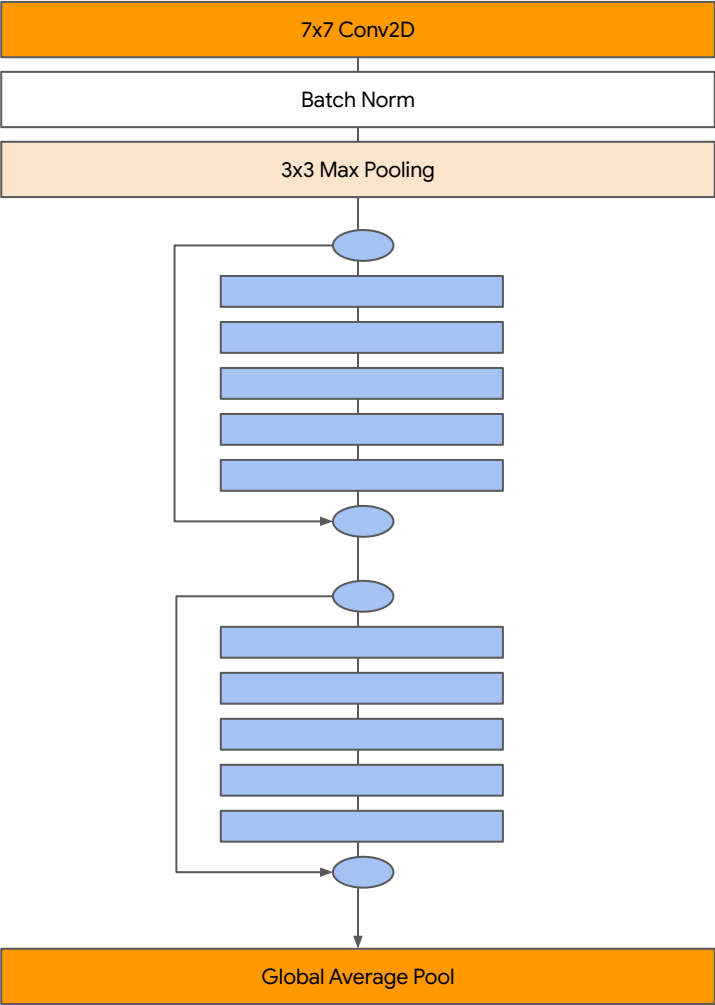
def call(self, input_tensor):
    x = self.conv1(input_tensor)
    x = self.bn1(x)
    x = self.act(x)

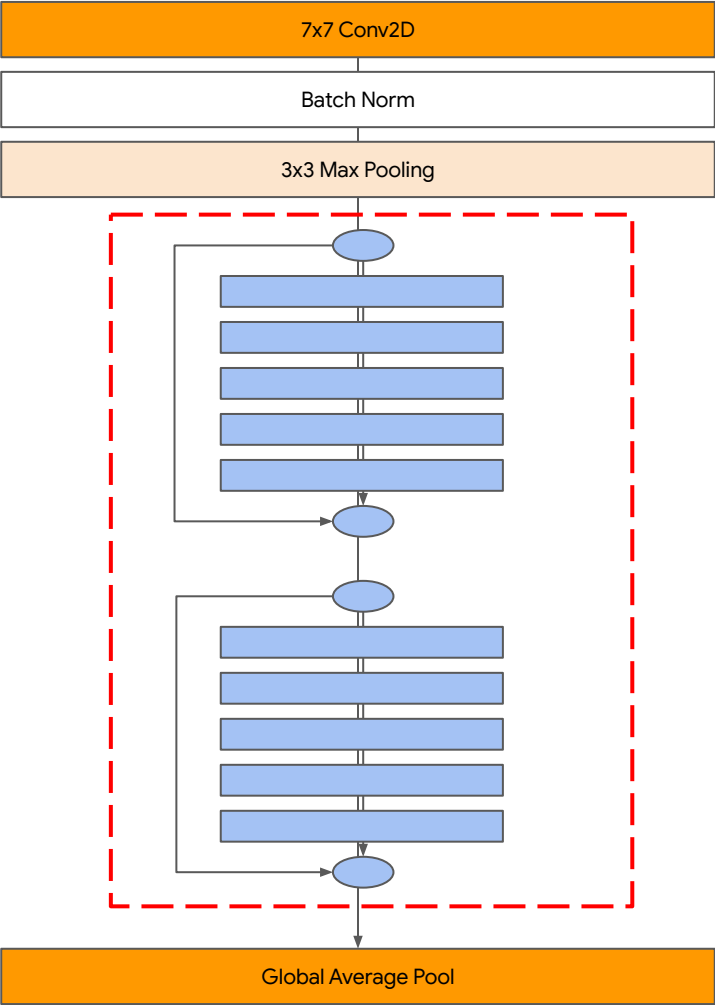
    x = self.conv2(x)
    x = self.bn2(x)
    x = self.act(x)

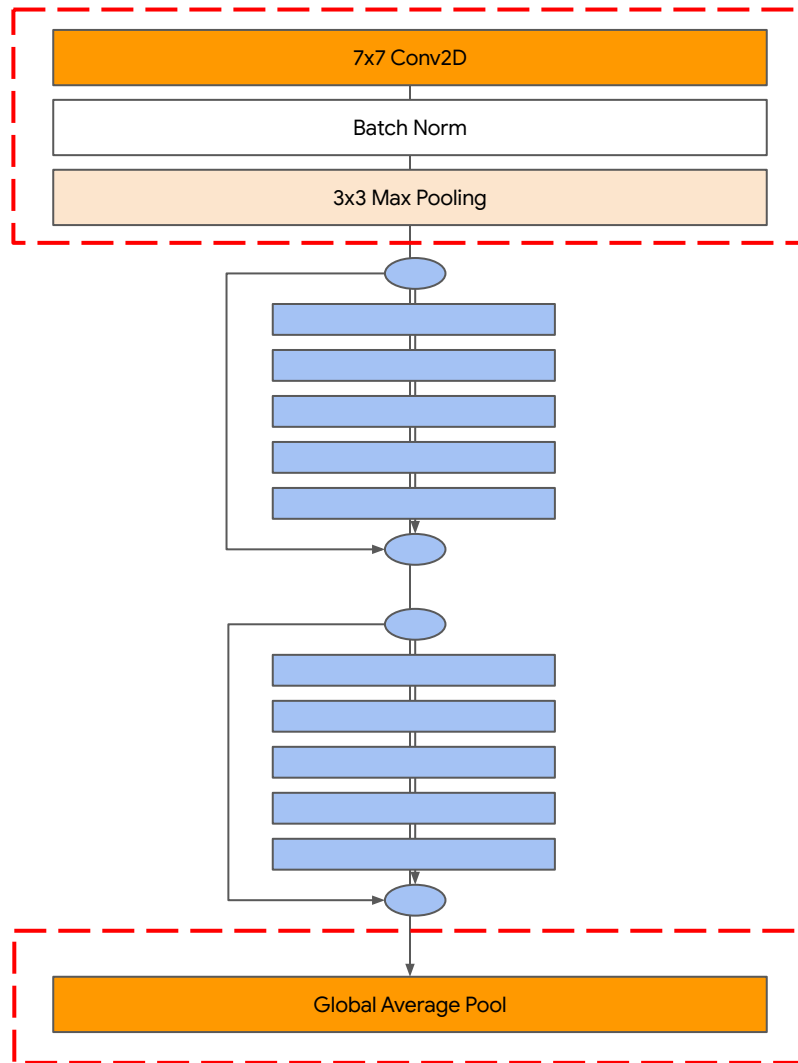
    x = self.add([x, input_tensor])
    x = self.act(x)
    return x

```

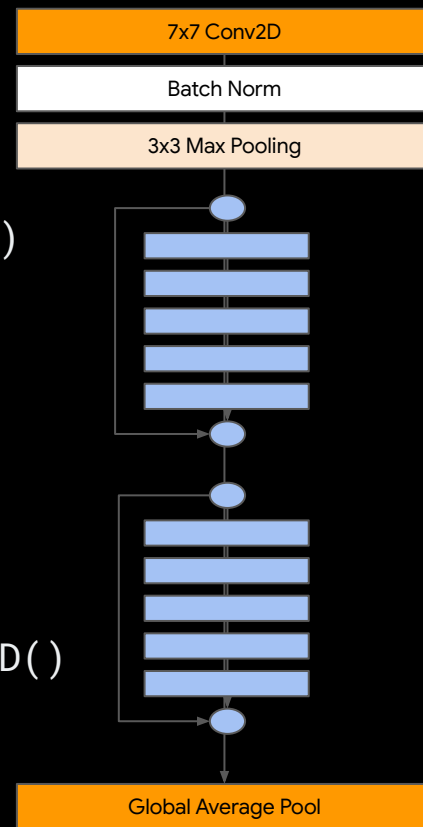








```
class ResNet(tf.keras.Model):  
    def __init__(self, num_classes):  
        super(ResNet, self).__init__()  
        self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')  
        self.bn = tf.keras.layers.BatchNormalization()  
        self.act = tf.keras.layers.Activation('relu')  
        self.max_pool = tf.keras.layers.MaxPool2D((3, 3))  
        self.id1a = IdentityBlock(64, 3)  
        self.id1b = IdentityBlock(64, 3)  
        self.global_pool = tf.keras.layers.GlobalAveragePooling2D()  
        self.classifier = tf.keras.layers.Dense(num_classes,  
                                                  activation='softmax')
```



```
class ResNet(tf.keras.Model):
```

```
    def __init__(self, num_classes):
```

```
        super(ResNet, self).__init__()
```

```
        self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')
```

```
        self.bn = tf.keras.layers.BatchNormalization()
```

```
        self.act = tf.keras.layers.Activation('relu')
```

```
        self.max_pool = tf.keras.layers.MaxPool2D((3, 3))
```

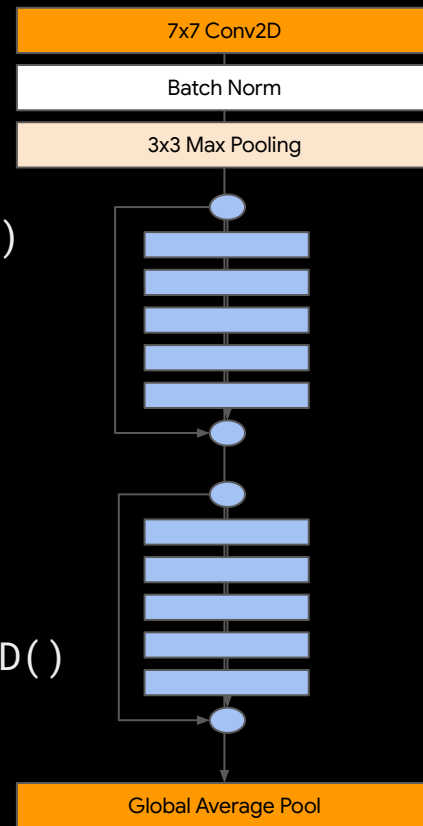
```
        self.id1a = IdentityBlock(64, 3)
```

```
        self.id1b = IdentityBlock(64, 3)
```

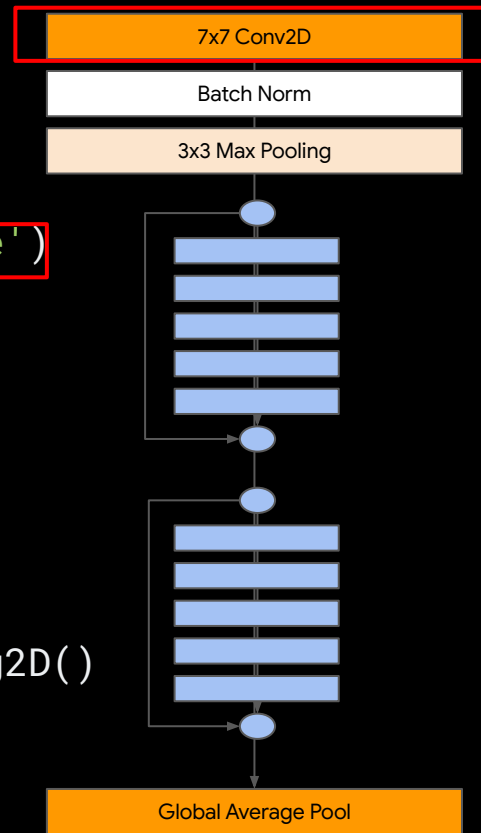
```
        self.global_pool = tf.keras.layers.GlobalAveragePooling2D()
```

```
        self.classifier = tf.keras.layers.Dense(num_classes,
```

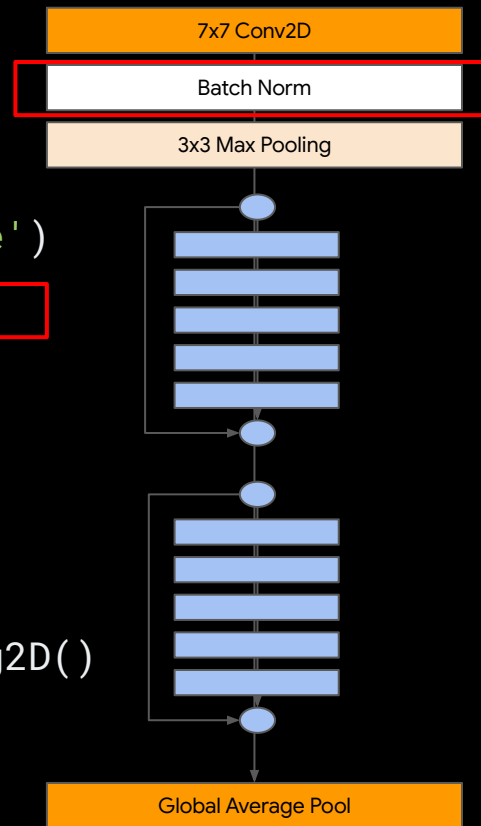
```
            activation='softmax')
```



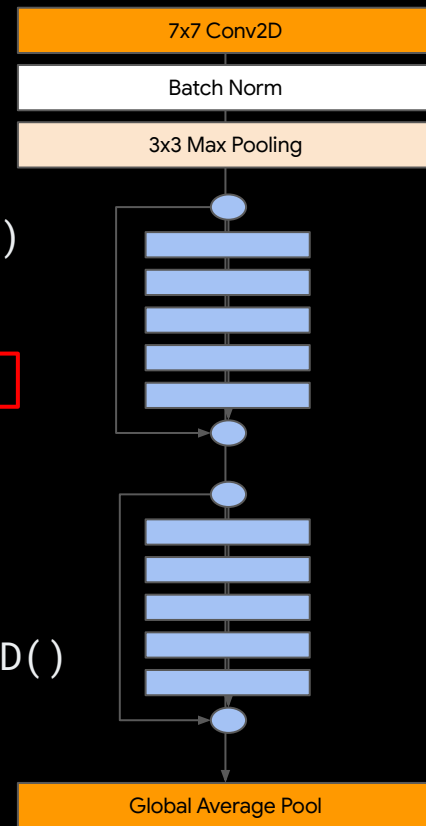
```
class ResNet(tf.keras.Model):  
    def __init__(self, num_classes):  
        super(ResNet, self).__init__()  
        self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')  
        self.bn = tf.keras.layers.BatchNormalization()  
        self.act = tf.keras.layers.Activation('relu')  
        self.max_pool = tf.keras.layers.MaxPool2D((3, 3))  
        self.id1a = IdentityBlock(64, 3)  
        self.id1b = IdentityBlock(64, 3)  
        self.global_pool = tf.keras.layers.GlobalAveragePooling2D()  
        self.classifier = tf.keras.layers.Dense(num_classes,  
                                                  activation='softmax')
```



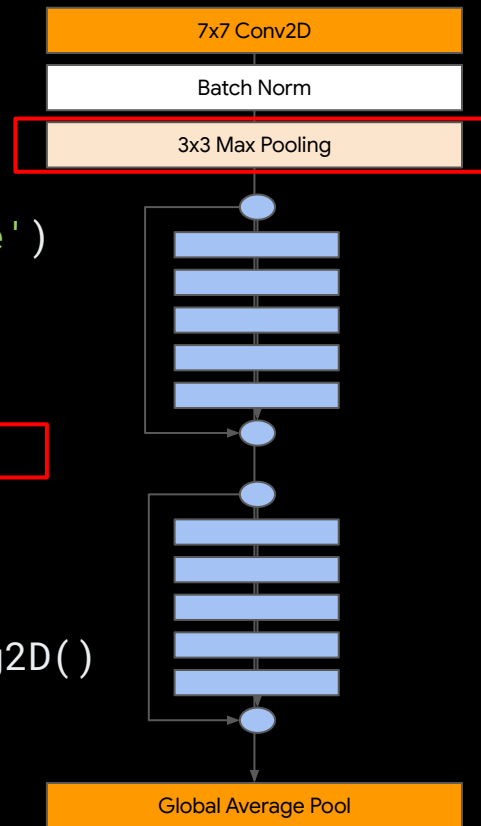
```
class ResNet(tf.keras.Model):  
    def __init__(self, num_classes):  
        super(ResNet, self).__init__()  
        self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')  
        self.bn = tf.keras.layers.BatchNormalization()  
        self.act = tf.keras.layers.Activation('relu')  
        self.max_pool = tf.keras.layers.MaxPool2D((3, 3))  
        self.id1a = IdentityBlock(64, 3)  
        self.id1b = IdentityBlock(64, 3)  
        self.global_pool = tf.keras.layers.GlobalAveragePooling2D()  
        self.classifier = tf.keras.layers.Dense(num_classes,  
                                                  activation='softmax')
```



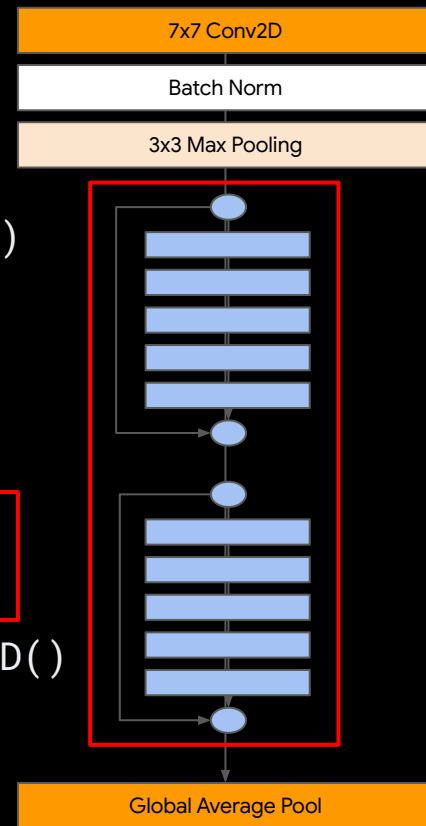
```
class ResNet(tf.keras.Model):  
    def __init__(self, num_classes):  
        super(ResNet, self).__init__()  
        self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')  
        self.bn = tf.keras.layers.BatchNormalization()  
        self.act = tf.keras.layers.Activation('relu')  
        self.max_pool = tf.keras.layers.MaxPool2D((3, 3))  
        self.id1a = IdentityBlock(64, 3)  
        self.id1b = IdentityBlock(64, 3)  
        self.global_pool = tf.keras.layers.GlobalAveragePooling2D()  
        self.classifier = tf.keras.layers.Dense(num_classes,  
                                                  activation='softmax')
```



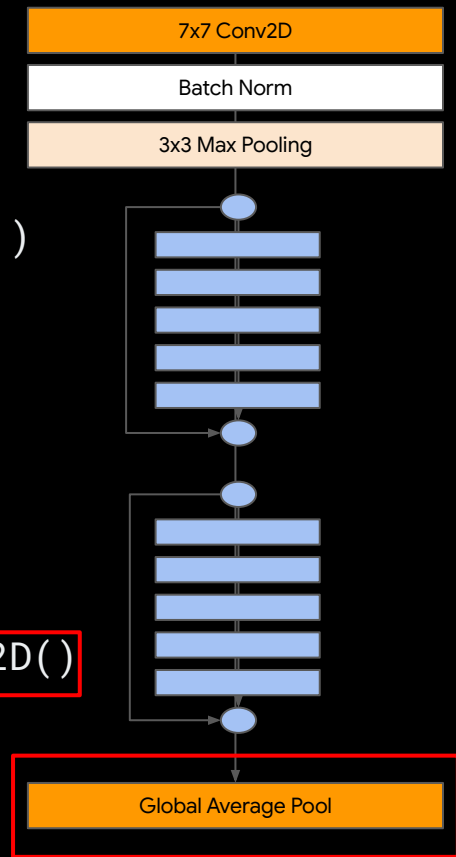
```
class ResNet(tf.keras.Model):  
    def __init__(self, num_classes):  
        super(ResNet, self).__init__()  
        self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')  
        self.bn = tf.keras.layers.BatchNormalization()  
        self.act = tf.keras.layers.Activation('relu')  
        self.max_pool = tf.keras.layers.MaxPool2D((3, 3))  
        self.id1a = IdentityBlock(64, 3)  
        self.id1b = IdentityBlock(64, 3)  
        self.global_pool = tf.keras.layers.GlobalAveragePooling2D()  
        self.classifier = tf.keras.layers.Dense(num_classes,  
                                                  activation='softmax')
```



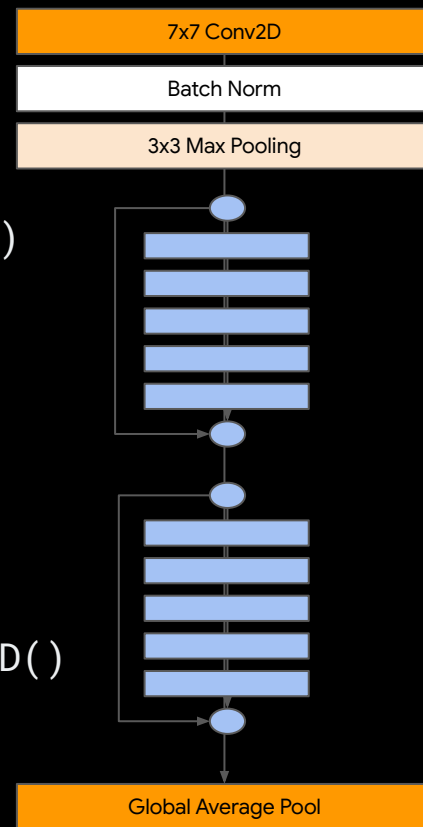
```
class ResNet(tf.keras.Model):  
    def __init__(self, num_classes):  
        super(ResNet, self).__init__()  
        self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')  
        self.bn = tf.keras.layers.BatchNormalization()  
        self.act = tf.keras.layers.Activation('relu')  
        self.max_pool = tf.keras.layers.MaxPool2D((3, 3))  
        self.id1a = IdentityBlock(64, 3)  
        self.id1b = IdentityBlock(64, 3)  
        self.global_pool = tf.keras.layers.GlobalAveragePooling2D()  
        self.classifier = tf.keras.layers.Dense(num_classes,  
                                                    activation='softmax')
```



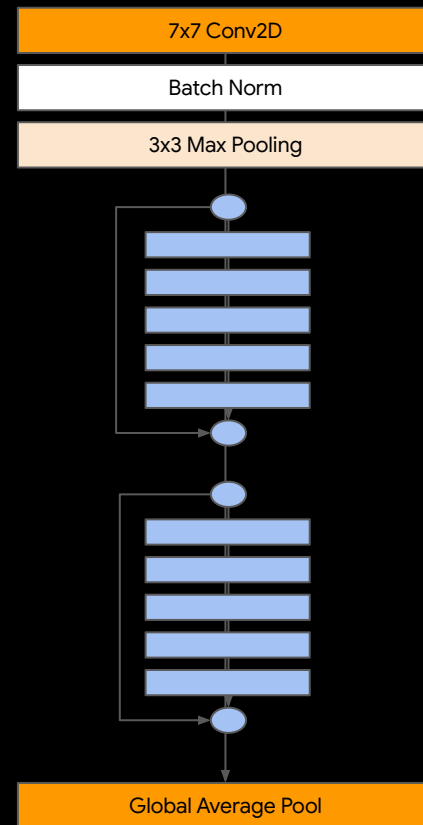

```
class ResNet(tf.keras.Model):  
    def __init__(self, num_classes):  
        super(ResNet, self).__init__()  
        self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')  
        self.bn = tf.keras.layers.BatchNormalization()  
        self.act = tf.keras.layers.Activation('relu')  
        self.max_pool = tf.keras.layers.MaxPool2D((3, 3))  
        self.id1a = IdentityBlock(64, 3)  
        self.id1b = IdentityBlock(64, 3)  
        self.global_pool = tf.keras.layers.GlobalAveragePooling2D()  
        self.classifier = tf.keras.layers.Dense(num_classes,  
                                                    activation='softmax')
```



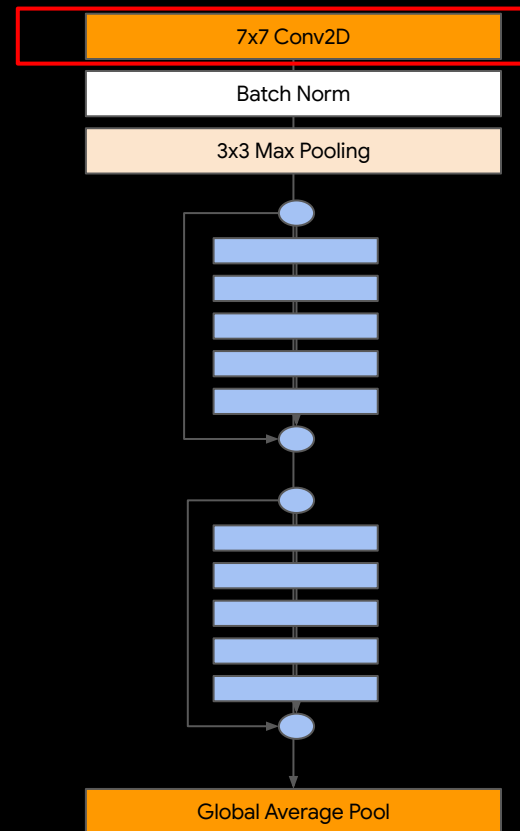
```
class ResNet(tf.keras.Model):  
    def __init__(self, num_classes):  
        super(ResNet, self).__init__()  
        self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')  
        self.bn = tf.keras.layers.BatchNormalization()  
        self.act = tf.keras.layers.Activation('relu')  
        self.max_pool = tf.keras.layers.MaxPool2D((3, 3))  
        self.id1a = IdentityBlock(64, 3)  
        self.id1b = IdentityBlock(64, 3)  
        self.global_pool = tf.keras.layers.GlobalAveragePooling2D()  
        self.classifier = tf.keras.layers.Dense(num_classes,  
                                                  activation='softmax')
```



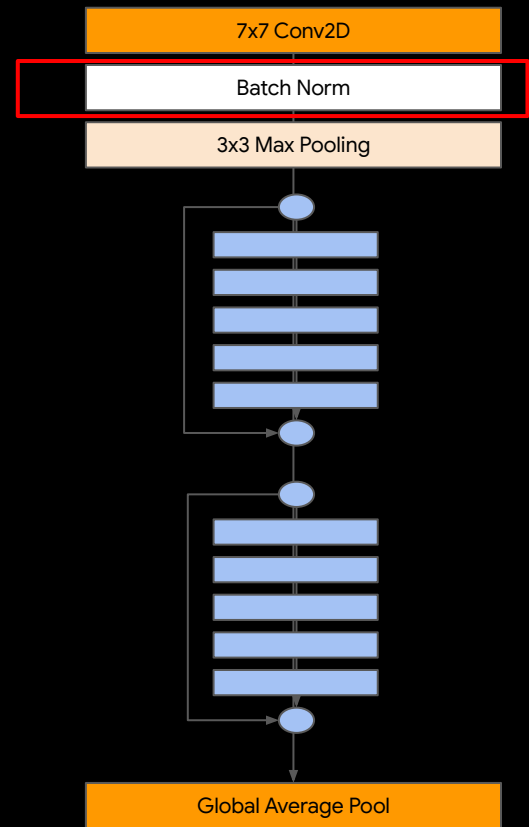
```
def call(self, inputs):  
    x = self.conv(inputs)  
    x = self.bn(x)  
    x = self.act(x)  
    x = self.max_pool(x)  
  
    x = self.id1a(x)  
    x = self.id1b(x)  
  
    x = self.global_pool(x)  
    return self.classifier(x)
```



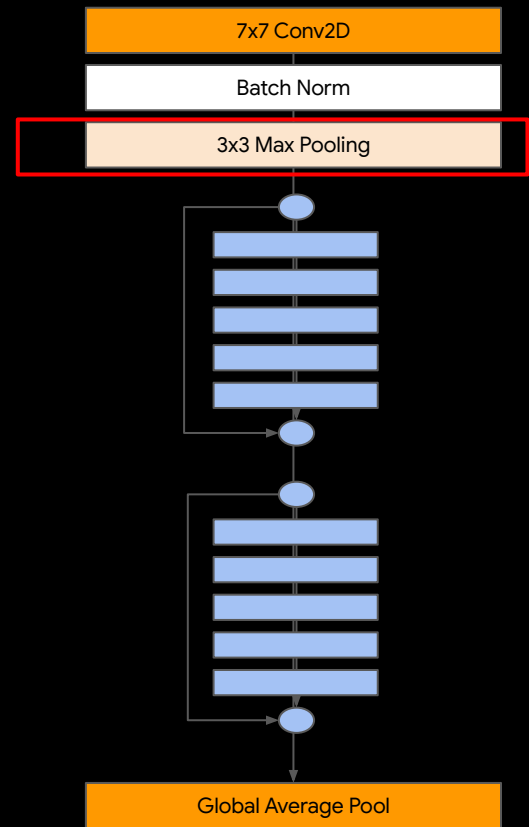
```
def call(self, inputs):  
    x = self.conv(inputs)  
    x = self.bn(x)  
    x = self.act(x)  
    x = self.max_pool(x)  
  
    x = self.id1a(x)  
    x = self.id1b(x)  
  
    x = self.global_pool(x)  
    return self.classifier(x)
```



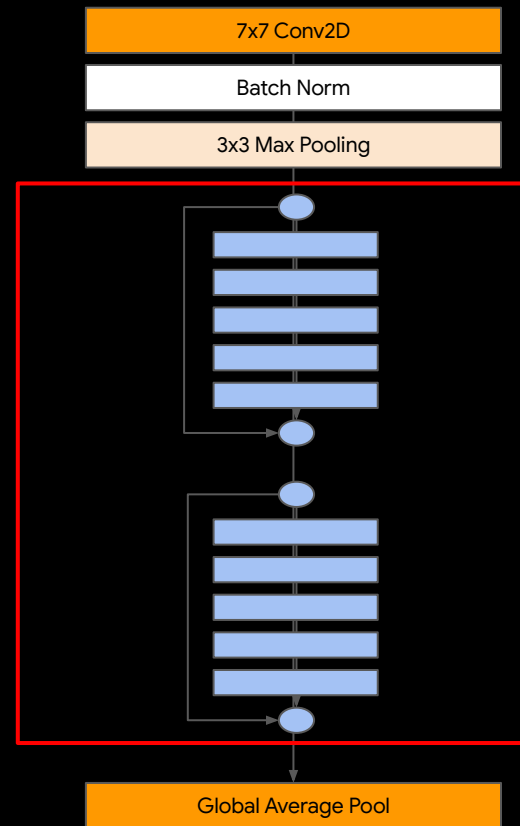
```
def call(self, inputs):  
    x = self.conv(inputs)  
    x = self.bn(x)  
    x = self.act(x)  
    x = self.max_pool(x)  
  
    x = self.id1a(x)  
    x = self.id1b(x)  
  
    x = self.global_pool(x)  
    return self.classifier(x)
```



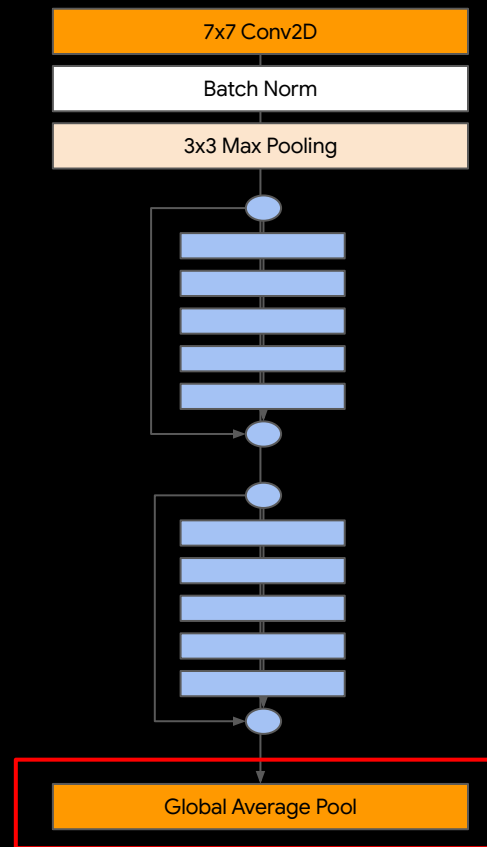
```
def call(self, inputs):  
    x = self.conv(inputs)  
    x = self.bn(x)  
    x = self.act(x)  
    x = self.max_pool(x)  
  
    x = self.id1a(x)  
    x = self.id1b(x)  
  
    x = self.global_pool(x)  
    return self.classifier(x)
```



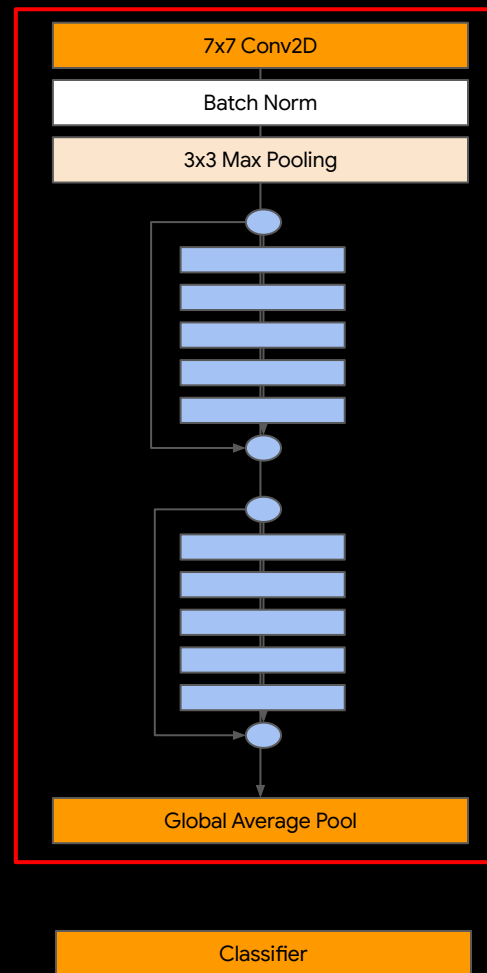
```
def call(self, inputs):  
    x = self.conv(inputs)  
    x = self.bn(x)  
    x = self.act(x)  
    x = self.max_pool(x)  
  
    x = self.id1a(x)  
    x = self.id1b(x)  
  
    x = self.global_pool(x)  
    return self.classifier(x)
```



```
def call(self, inputs):  
    x = self.conv(inputs)  
    x = self.bn(x)  
    x = self.act(x)  
    x = self.max_pool(x)  
  
    x = self.id1a(x)  
    x = self.id1b(x)  
  
    x = self.global_pool(x)  
    return self.classifier(x)
```




```
def call(self, inputs):  
    x = self.conv(inputs)  
    x = self.bn(x)  
    x = self.act(x)  
    x = self.max_pool(x)  
  
    x = self.id1a(x)  
    x = self.id1b(x)  
  
    x = self.global_pool(x)  
    return self.classifier(x)
```



```
resnet = ResNet(10)
resnet.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
dataset = tfds.load('mnist', split=tfds.Split.TRAIN)
dataset = dataset.map(preprocess).batch(32)
resnet.fit(dataset, epochs=1)
```

```
resnet = ResNet(10)
resnet.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
dataset = tfds.load('mnist', split=tfds.Split.TRAIN)
dataset = dataset.map(preprocess).batch(32)
resnet.fit(dataset, epochs=1)
```

```
resnet = ResNet(10)
resnet.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
dataset = tfds.load('mnist', split=tfds.Split.TRAIN)
dataset = dataset.map(preprocess).batch(32)
resnet.fit(dataset, epochs=1)
```

```
resnet = ResNet(10)
resnet.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
dataset = tfds.load('mnist', split=tfds.Split.TRAIN)
dataset = dataset.map(preprocess).batch(32)
resnet.fit(dataset, epochs=1)
```