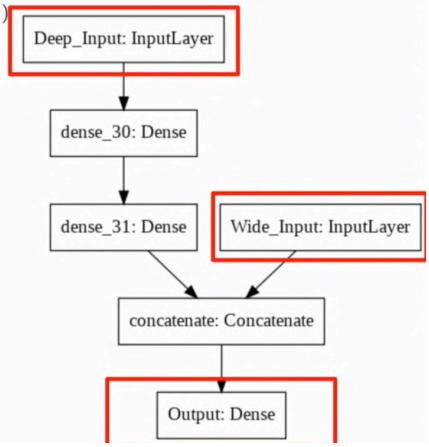
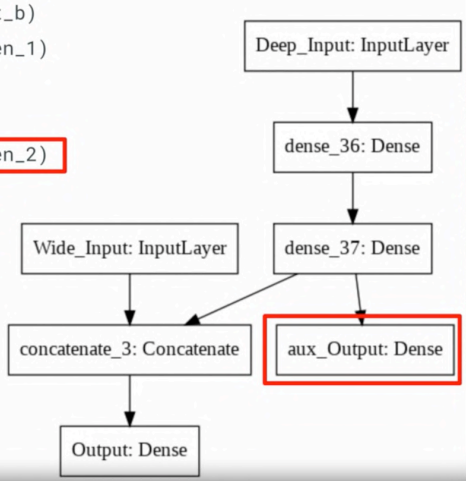
```
input_a = Input(shape=[1], name="Wide_Input")
input_b = Input(shape=[1], name="Deep_Input")
hidden_1 = Dense(30, activation="relu")(input_b)
hidden_2 = Dense(30, activation="relu")(hidden_1)
concat = concatenate([input_a, hidden_2])
output = Dense(1, name="Output")(concat)
model = Model(inputs=[input_a,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




```
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
```

Previously you saw how to implement a complex architecture using the functional API.

In this section, you'll see how this architecture can be encapsulated into a class for tidy your code and easier reuse. For cleaner codes when encapsulating the entire model, particularly useful if you want to orchestrate multiple models in a solution, you can create a class of your own.

When this class extends the base Keras model, you can then do everything that you would do with a model such as training and running inference et cetera.

When you extend the model class like this, you need to implement at least the following two methods.

```
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
```

The first init, initializes the class and should also be used to initialize the base class that this one extends. In this case that's the model class and then it creates the instances of the internal variables or states that this class will use. If it looks similar to what you did for custom layers last week, you'd be right, the pattern is pretty much identical.

```
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
```

The model had hidden layers and outputs that we're dense layers. We can create class variables that represent them using the init function.

```
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
```

The other function you'll need is the call function, and that gets executed when the class is constructed, and then here you can define your model outputs, which will be returned by the

```
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
```

The outputs are generated based on the inputs through the model architecture, so you effectively encapsulate the entire architecture here. You pass it the inputs, and it passes their data through the network architecture to get the outputs that it can then return back.

```
model = WideAndDeepModel()
```

Now, once you've defined this class, it's easy to create a model using an instance of it like this and this keeps your main code so much cleaner. But it also has lots more benefits because the creation of the layers is separated from the usage, you can do lots of interesting stuff in the call method.

For example, you could have loops defining multiple layers, you could have if then statements or other operations. You aren't limited to the static declaration of the model that you'd have with the functional or sequential APIs.

It would also allow you to define subnetworks that might be used in larger models, and you'll explore that next.

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

If you're using the sequential a functional APIs, there are still some limitations that you have to consider when you look at complex or exotic model architectures.

For example, networks like directed acyclic graphs, which are made up of directed layers that don't loop or cycle are pretty well-suited for sequential or functional APIs.

Examples of this are MobileNet and Inception. In these cases, the data flows from the inputs to the outputs, and sometimes it's in multiple branches as we've seen already, but the direction is always the same, it never loops back during training or inference.

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

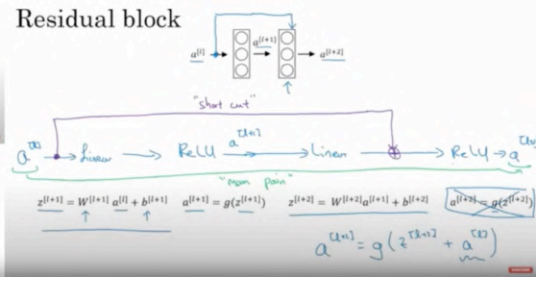
- Extends how you've been building models
- Continue to use functional and sequential code
- Modular architecture
- Try out experiments quickly
- Control flow in the network

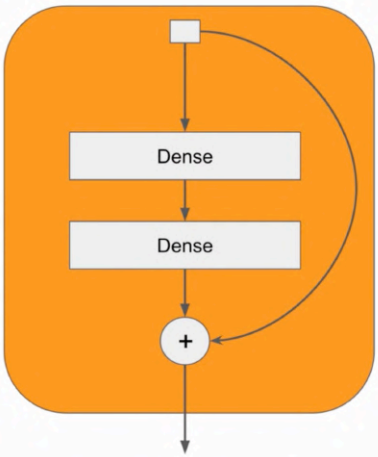
Benefits of subclassing models

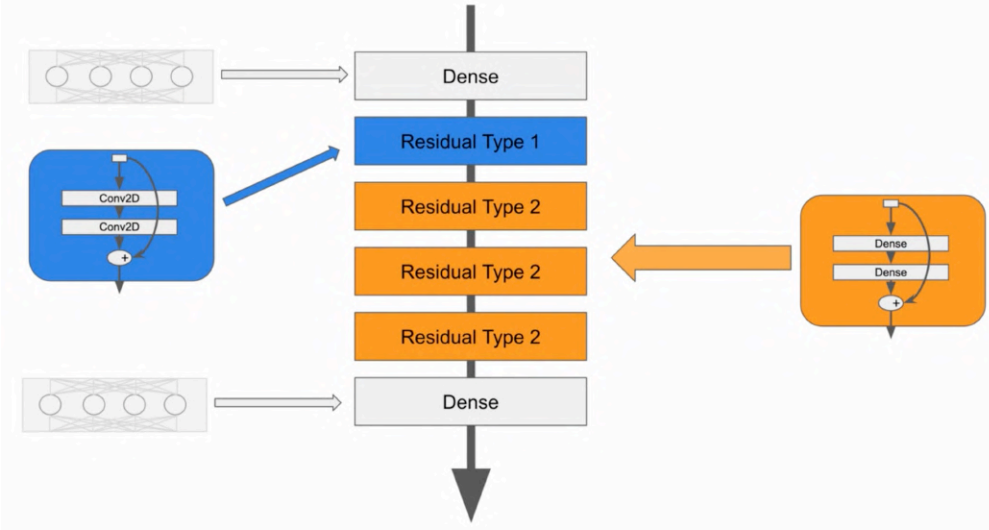
- Extends how you've been building models
- Continue to use functional and sequential code
- Modular architecture
- Try out experiments quickly
- Control flow in the network

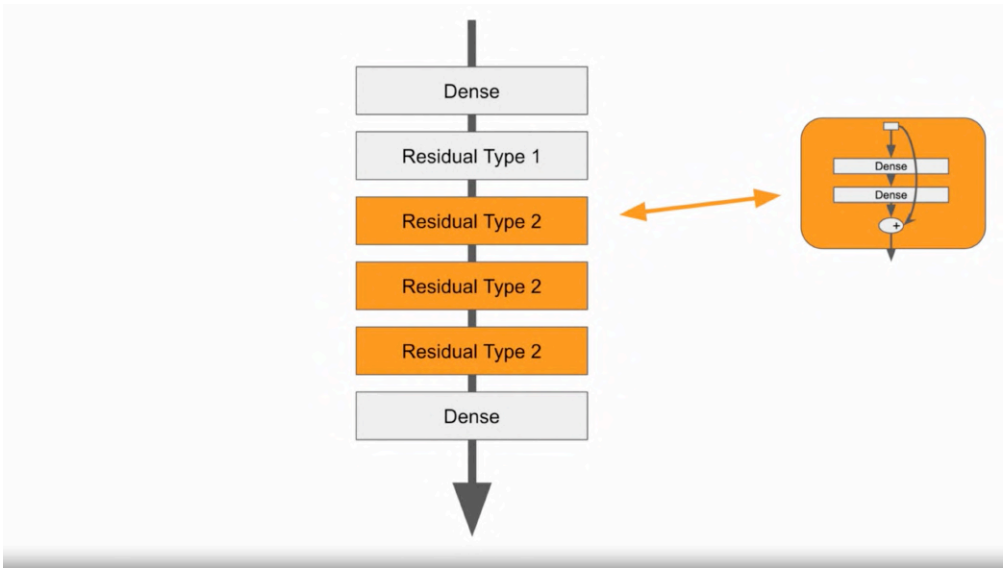
Residual Networks (ResNets)

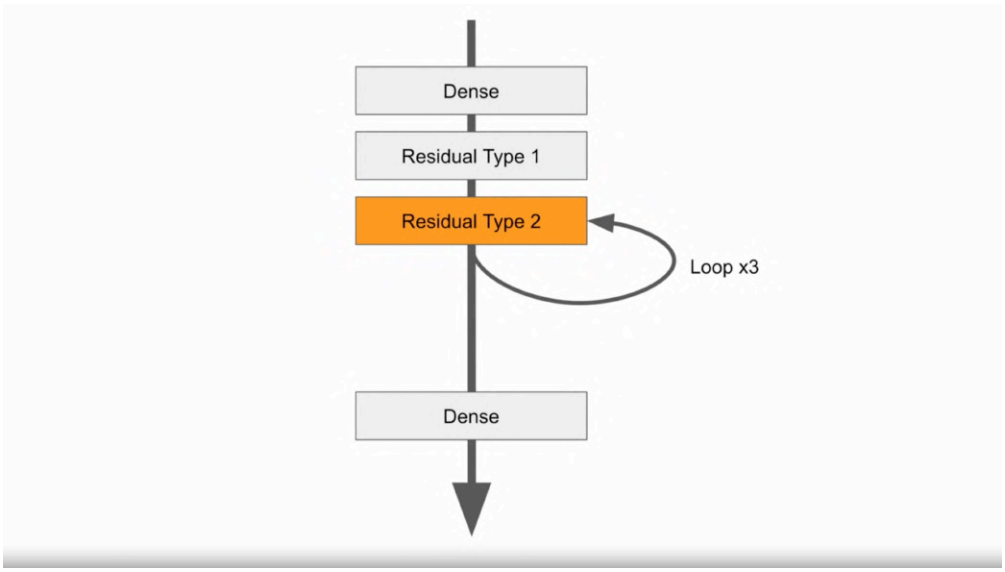
https://www.coursera.org/lecture/convolutional-neural-networks/resnets-HAhz9












```
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
```

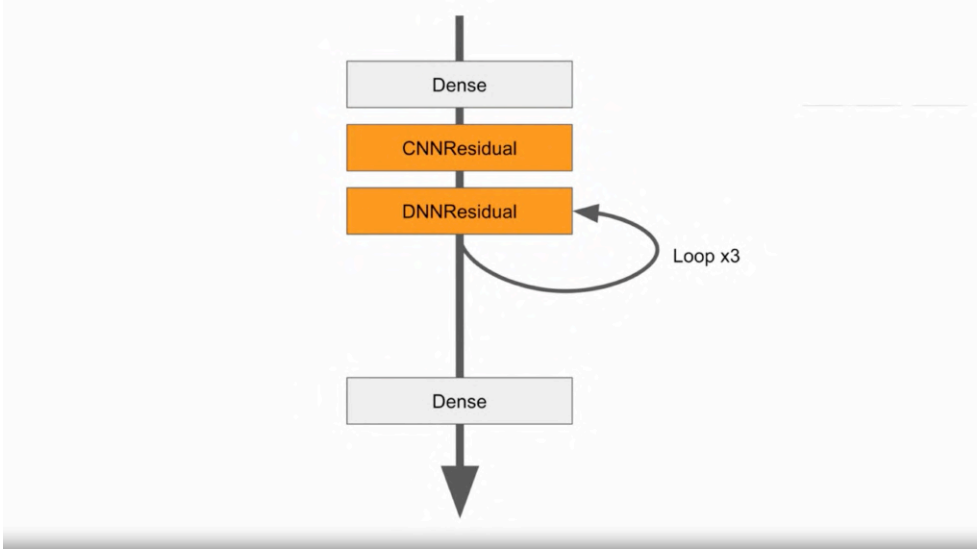
```
class CNNResidual(Layer):
  def __init__(self, layers, filters, **kwargs):
    super().__init__(**kwargs)
    self.hidden = [Conv2D(filters, (3, 3), activation="relu")
                   for _ in range(layers)]

  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
```

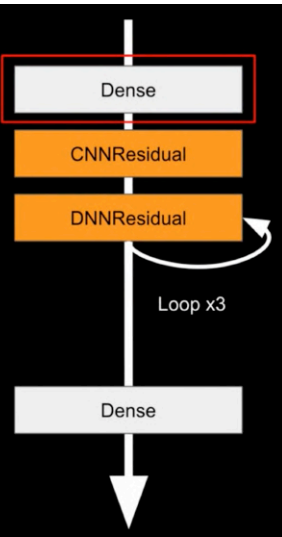
```
class CNNResidual(Layer):
  def __init__(self, layers, filters, **kwargs):
    super().__init__(**kwargs)
    self.hidden = [Conv2D(filters, (3, 3), activation="relu")
                   for _ in range(layers)]

  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
```

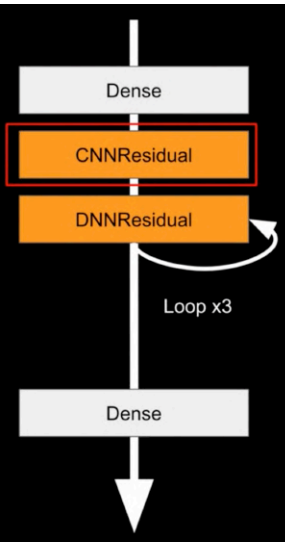
```
class DNNResidual(Layer):
  def __init__(self, layers, neurons, **kwargs):
    super().__init__(**kwargs)
    self.hidden = [Dense(neurons, activation="relu")
                   for _ in range(layers)]

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



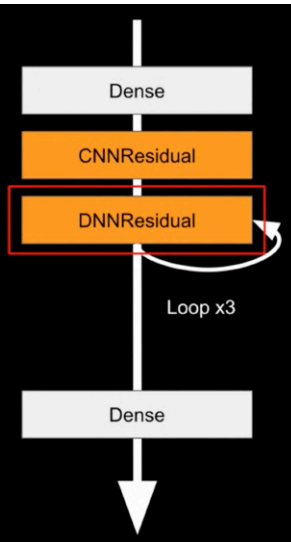

```
class MyResidual(Model):
  def __init__(self, **kwargs):
    self.hidden1 = Dense(30, activation="relu")
    self.block1 = CNNResidual(2, 32)
    self.block2 = DNNResidual(2, 64)
    self.out = Dense(1)

  def call(self, inputs):
    x = self.hidden1(inputs)
    x = self.block1(x)
    for _ in range(1, 3):
      x = self.block2(x)
    return self.out(x)
```



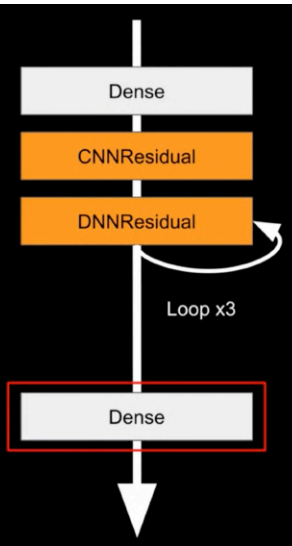

```
class MyResidual(Model):
  def __init__(self, **kwargs):
    self.hidden1 = Dense(30, activation="relu")
    self.block1 = CNNResidual(2, 32)
    self.block2 = DNNResidual(2, 64)
    self.out = Dense(1)

  def call(self, inputs):
    x = self.hidden1(inputs)
    x = self.block1(x)
    for _ in range(1, 3):
      x = self.block2(x)
    return self.out(x)
```



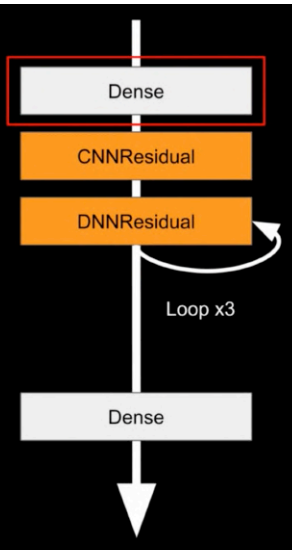
```
class MyResidual(Model):
  def __init__(self, **kwargs):
    self.hidden1 = Dense(30, activation="relu")
    self.block1 = CNNResidual(2, 32)
    self.block2 = DNNResidual(2, 64)
    self.out = Dense(1)

  def call(self, inputs):
    x = self.hidden1(inputs)
    x = self.block1(x)
    for _ in range(1, 3):
      x = self.block2(x)
    return self.out(x)
```



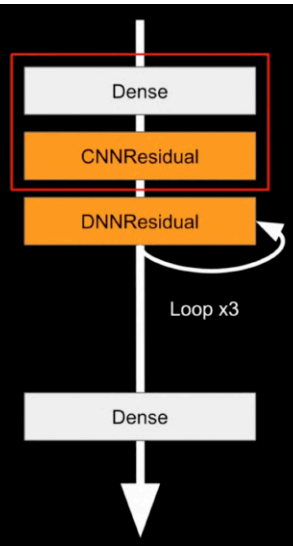

```
class MyResidual(Model):
  def __init__(self, **kwargs):
    self.hidden1 = Dense(30, activation="relu")
    self.block1 = CNNResidual(2, 32)
    self.block2 = DNNResidual(2, 64)
    self.out = Dense(1)

  def call(self, inputs):
    x = self.hidden1(inputs)
    x = self.block1(x)
    for _ in range(1, 3):
      x = self.block2(x)
    return self.out(x)
```



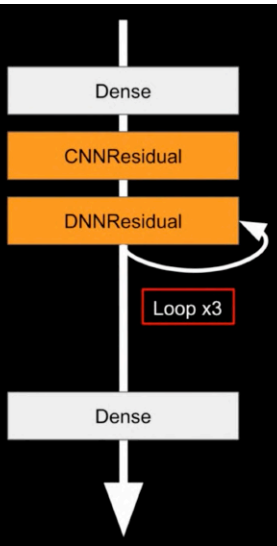

```
class MyResidual(Model):
  def __init__(self, **kwargs):
    self.hidden1 = Dense(30, activation="relu")
    self.block1 = CNNResidual(2, 32)
    self.block2 = DNNResidual(2, 64)
    self.out = Dense(1)

  def call(self, inputs):
    x = self.hidden1(inputs)
    x = self.block1(x)
    for _ in range(1, 3):
      x = self.block2(x)
    return self.out(x)
```



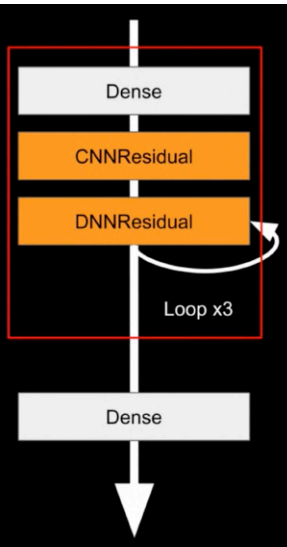
```python
class MyResidual(Model):
  def __init__(self, **kwargs):
    self.hidden1 = Dense(30, activation="relu")
    self.block1 = CNNResidual(2, 32)
    self.block2 = DNNResidual(2, 64)
    self.out = Dense(1)

  def call(self, inputs):
    x = self.hidden1(inputs)
    x = self.block1(x)
    for _ in range(1, 3):
      x = self.block2(x)
    return self.out(x)
```



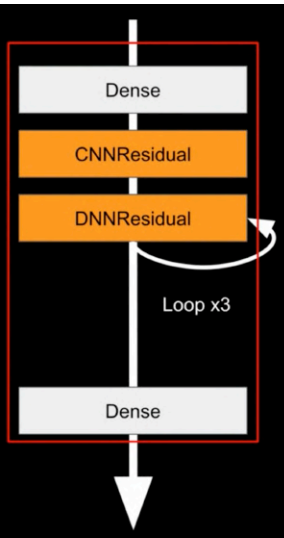

```python
class MyResidual(Model):
  def __init__(self, **kwargs):
    self.hidden1 = Dense(30, activation="relu")
    self.block1 = CNNResidual(2, 32)
    self.block2 = DNNResidual(2, 64)
    self.out = Dense(1)

  def call(self, inputs):
    x = self.hidden1(inputs)
    x = self.block1(x)
    for _ in range(1, 3):
      x = self.block2(x)
    return self.out(x)
```

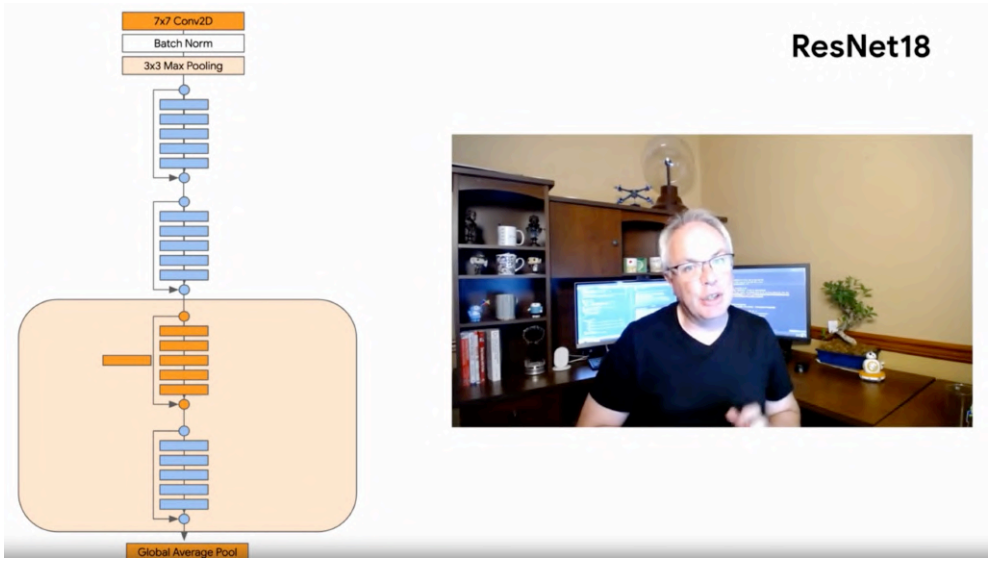


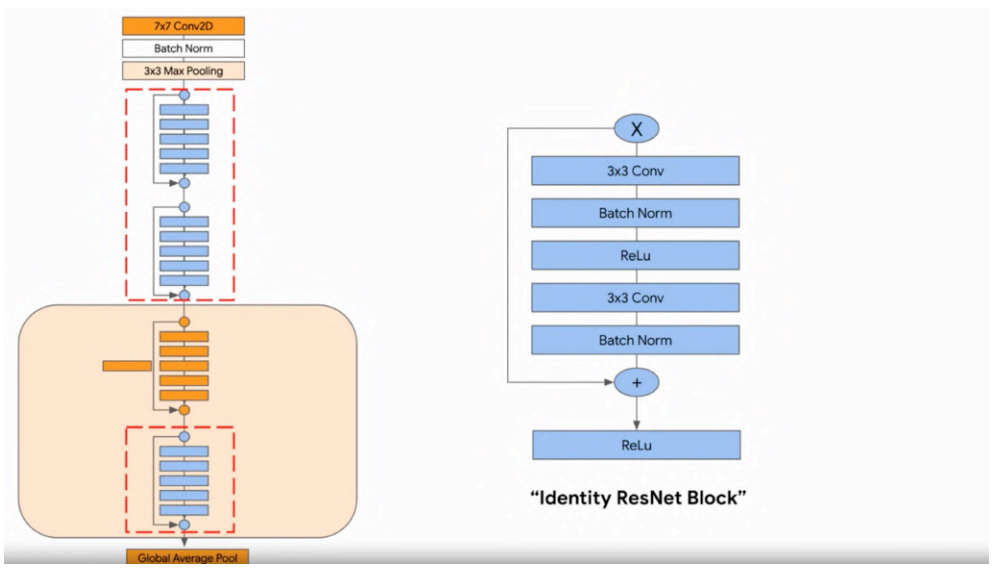

```python
class MyResidual(Model):
  def __init__(self, **kwargs):
    self.hidden1 = Dense(30, activation="relu")
    self.block1 = CNNResidual(2, 32)
    self.block2 = DNNResidual(2, 64)
    self.out = Dense(1)

  def call(self, inputs):
    x = self.hidden1(inputs)
    x = self.block1(x)
    for _ in range(1, 3):
      x = self.block2(x)
    return self.out(x)
```

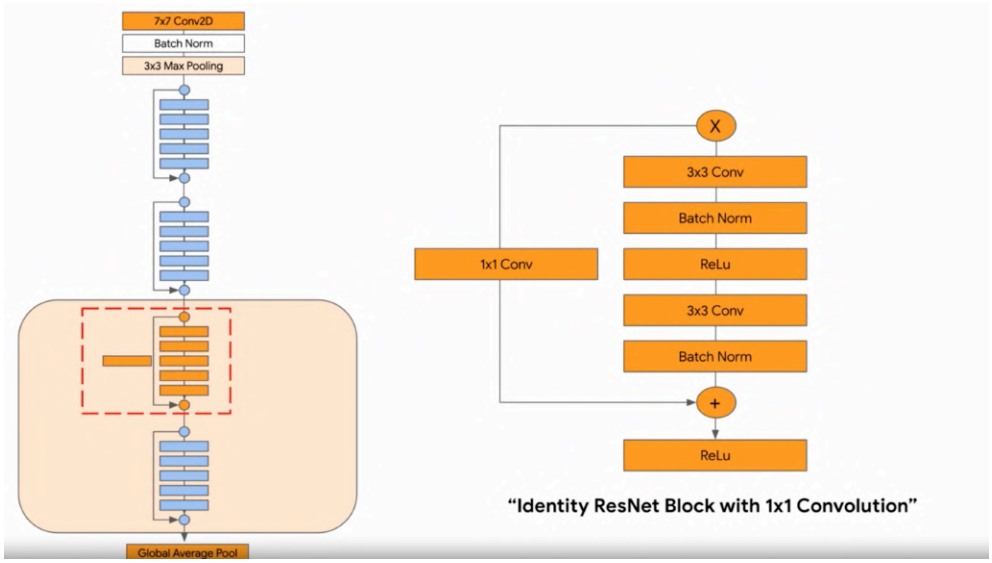


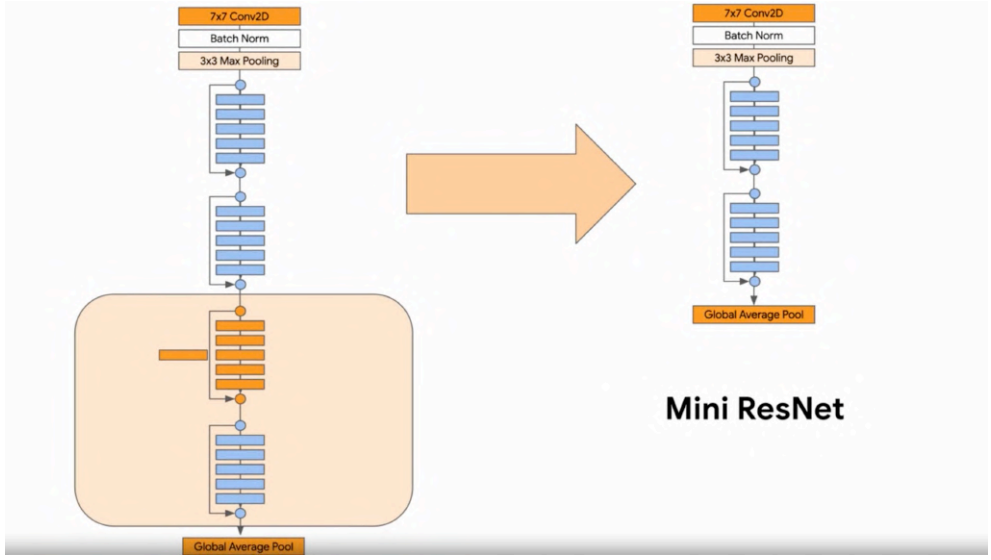
```python
class MyResidual(Model):
  def __init__(self, **kwargs):
    self.hidden1 = Dense(30, activation="relu")
    self.block1 = CNNResidual(2, 32)
    self.block2 = DNNResidual(2, 64)
    self.out = Dense(1)

  def call(self, inputs):
    x = self.hidden1(inputs)
    x = self.block1(x)
    for _ in range(1, 3):
      x = self.block2(x)
    return self.out(x)
```



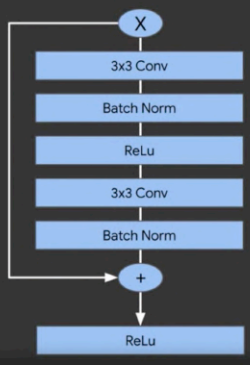




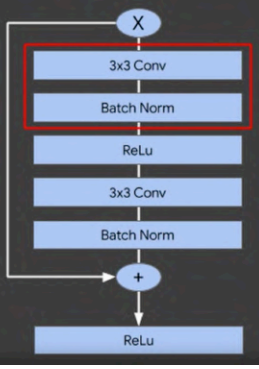




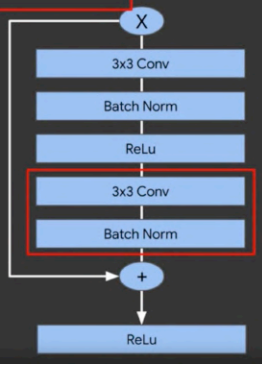

```python
class IdentityBlock(tf.keras.Model):
  def __init__(self, filters, kernel_size):
    super(IdentityBlock, self).__init__(name='')

    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
```



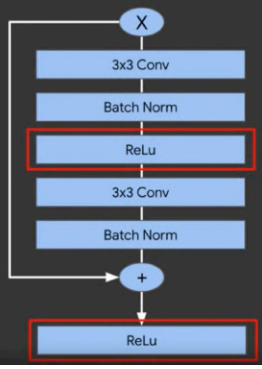
```python
class IdentityBlock(tf.keras.Model):
  def __init__(self, filters, kernel_size):
    super(IdentityBlock, self).__init__(name='')

    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
```



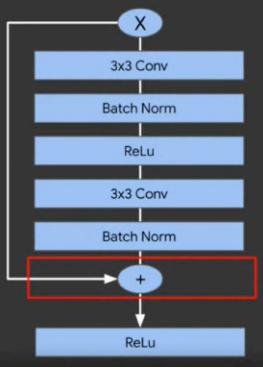

```python
class IdentityBlock(tf.keras.Model):
  def __init__(self, filters, kernel_size):
    super(IdentityBlock, self).__init__(name='')

    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
```



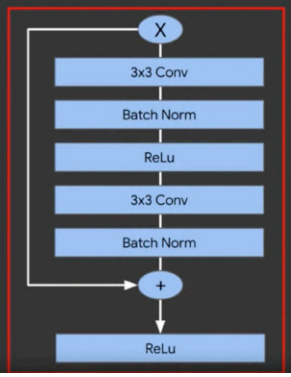

```python
class IdentityBlock(tf.keras.Model):
  def __init__(self, filters, kernel_size):
    super(IdentityBlock, self).__init__(name='')

    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
```

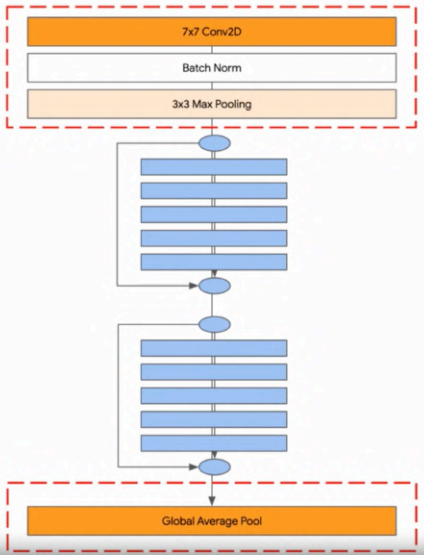


```python
class IdentityBlock(tf.keras.Model):
  def __init__(self, filters, kernel_size):
    super(IdentityBlock, self).__init__(name='')

    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
```

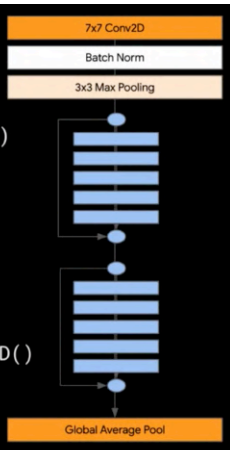



```python
class IdentityBlock(tf.keras.Model):
  def __init__(self, filters, kernel_size):
    super(IdentityBlock, self).__init__(name='')

    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
```

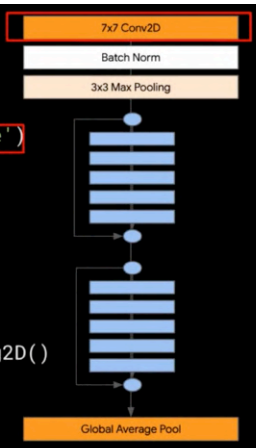




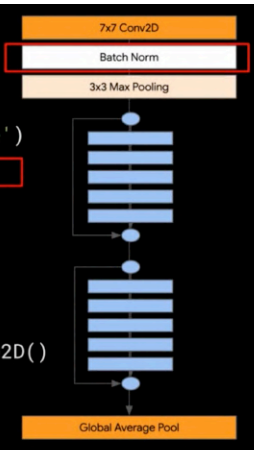
```
class ResNet(tf.keras.Model):
  def __init__(self, num_classes):
    super(ResNet, self).__init__()
    self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')
    self.bn = tf.keras.layers.BatchNormalization()
    self.act = tf.keras.layers.Activation('relu')
    self.max_pool = tf.keras.layers.MaxPool2D((3, 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')
```



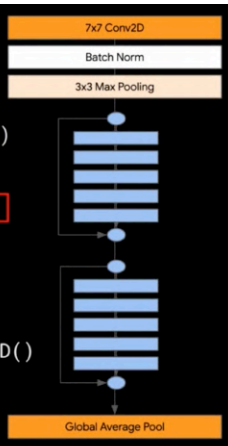

```
class ResNet(tf.keras.Model):
  def __init__(self, num_classes):
    super(ResNet, self).__init__()
    self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')
    self.bn = tf.keras.layers.BatchNormalization()
    self.act = tf.keras.layers.Activation('relu')
    self.max_pool = tf.keras.layers.MaxPool2D((3, 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')
```



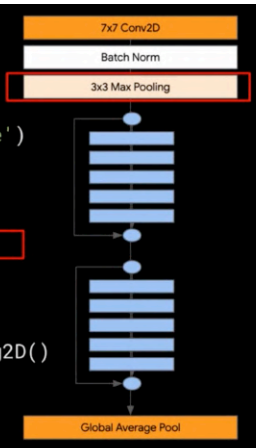

```
class ResNet(tf.keras.Model):
  def __init__(self, num_classes):
    super(ResNet, self).__init__()
    self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')
    self.bn = tf.keras.layers.BatchNormalization()
    self.act = tf.keras.layers.Activation('relu')
    self.max_pool = tf.keras.layers.MaxPool2D((3, 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')
```



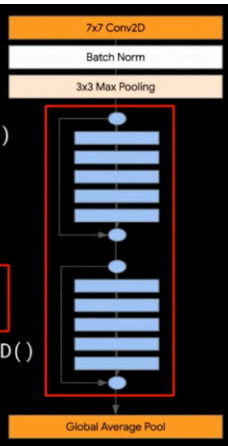
```
class ResNet(tf.keras.Model):
  def __init__(self, num_classes):
    super(ResNet, self).__init__()
    self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')
    self.bn = tf.keras.layers.BatchNormalization()
    self.act = tf.keras.layers.Activation('relu')
    self.max_pool = tf.keras.layers.MaxPool2D((3, 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')
```



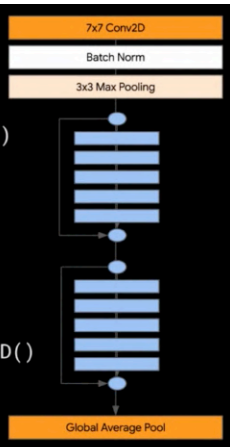

```
class ResNet(tf.keras.Model):
  def __init__(self, num_classes):
    super(ResNet, self).__init__()
    self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')
    self.bn = tf.keras.layers.BatchNormalization()
    self.act = tf.keras.layers.Activation('relu')
    self.max_pool = tf.keras.layers.MaxPool2D((3, 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')
```



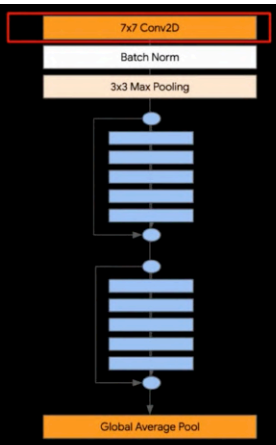

```
class ResNet(tf.keras.Model):
  def __init__(self, num_classes):
    super(ResNet, self).__init__()
    self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')
    self.bn = tf.keras.layers.BatchNormalization()
    self.act = tf.keras.layers.Activation('relu')
    self.max_pool = tf.keras.layers.MaxPool2D((3, 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')
```



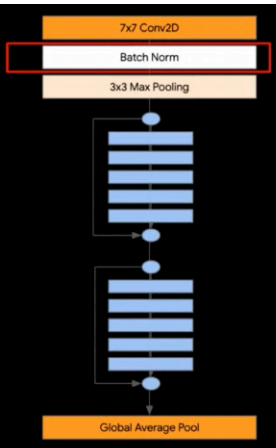
```python
class ResNet(tf.keras.Model):
  def __init__(self, num_classes):
    super(ResNet, self).__init__()
    self.conv = tf.keras.layers.Conv2D(64, 7, padding='same')
    self.bn = tf.keras.layers.BatchNormalization()
    self.act = tf.keras.layers.Activation('relu')
    self.max_pool = tf.keras.layers.MaxPool2D((3, 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')
```



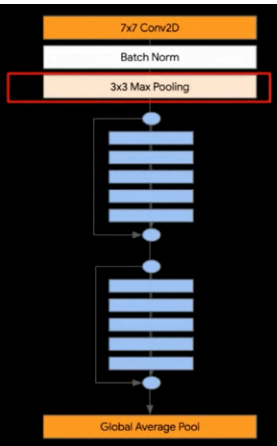

```python
def call(self, inputs):
  x = self.conv(inputs)
  x = self.bn(x)
  x = self.act(x)
  x = self.max_pool(x)

  x = self.id1a(x)
  x = self.id1b(x)

  x = self.global_pool(x)
  return self.classifier(x)
```



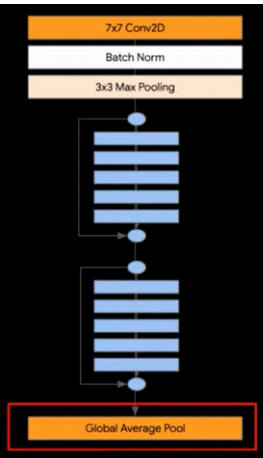

```python
def call(self, inputs):
  x = self.conv(inputs)
  x = self.bn(x)
  x = self.act(x)
  x = self.max_pool(x)

  x = self.id1a(x)
  x = self.id1b(x)

  x = self.global_pool(x)
  return self.classifier(x)
```



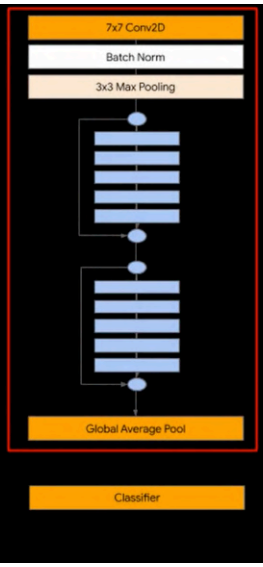
```
def call(self, inputs):
  x = self.conv(inputs)
  x = self.bn(x)
  x = self.act(x)
  x = self.max_pool(x)

  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)
```