

## **Models and Layers**

Sequential Models and Adding layers to the model using the `add()` method. Configuring layer parameters such as activation function, input shape, and output shape.

Sequential models in TensorFlow/Keras provide a simple way to build a neural network by stacking layers in a linear fashion. Each layer has one input tensor and one output tensor, making it straightforward to add and configure layers using the `add()` method.

### **Adding Layers to a Sequential Model**

The `add()` method is used to add layers to a sequential model one by one. Here's how you can add different types of layers and configure their parameters:

1. **Dense Layer:** A fully connected layer where each neuron is connected to every neuron in the previous layer.
2. **Activation Function:** Specifies the activation function to use for the layer.
3. **Input Shape:** Defines the shape of the input data that the layer expects.
4. **Output Shape:** Determined automatically based on the number of units and activation function.

### **Example Code**

Here's an example of creating a sequential model, adding layers to it, and configuring their parameters:

```
import tensorflow as tf
```

```
# Step 1: Create a Sequential model
```

```
model = tf.keras.Sequential()
```

```
# Step 2: Add layers to the model
```

```
# Add a Dense layer with 64 units, ReLU activation, and an input shape of (784,)
```

```
model.add(tf.keras.layers.Dense(64, activation='relu', input_shape=(784,)))
```

```
# Add another Dense layer with 32 units and ReLU activation
```

```
model.add(tf.keras.layers.Dense(32, activation='relu'))
```

```
# Add a Dropout layer with a dropout rate of 0.5
```

```
model.add(tf.keras.layers.Dropout(0.5))
```

# Add the output layer with 10 units and softmax activation

```
model.add(tf.keras.layers.Dense(10, activation='softmax'))
```

# Step 3: Compile the model

# Specify the optimizer, loss function, and metrics for training

```
model.compile(optimizer='adam',  
              loss='sparse_categorical_crossentropy',  
              metrics=['accuracy'])
```

# Display the model architecture

```
model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim`  
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)  
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 64)	50,240
dense_1 (Dense)	(None, 32)	2,080
dropout (Dropout)	(None, 32)	0
dense_2 (Dense)	(None, 10)	330

Total params: 52,650 (205.66 KB)  
Trainable params: 52,650 (205.66 KB)  
Non-trainable params: 0 (0.00 B)

Functional API Models and Creating complex model architectures with multiple inputs and outputs.  
Utilizing shared layers and layer reuse in model architectures

The Functional API in TensorFlow/Keras is a way to build complex neural network architectures that require more flexibility than the Sequential API. It allows you to define models with multiple inputs and outputs, shared layers, and non-linear topology. This is especially useful for tasks where the model's architecture cannot be strictly linear.

### Key Concepts

1. Inputs: Define the input layers of the model using `tf.keras.Input`. Each input can have a different shape.

2. Layers: Create layers and connect them to the inputs or other layers. This forms a directed acyclic graph of layers.
3. Models: Define the model by specifying its inputs and outputs using `tf.keras.Model`.

#### Example Code with Explanation

Let's create a model with two inputs: one for processing text data and another for processing image data. The model will have shared layers and produce two outputs.

```
import tensorflow as tf
```

```
from tensorflow.keras import layers
```

```
# Step 1: Define the Input Layers
```

```
input_text = tf.keras.Input(shape=(100,), name='text_input') # Input layer for text data
```

```
input_image = tf.keras.Input(shape=(64, 64, 3), name='image_input') # Input layer for image data
```

```
# Step 2: Define the Text Branch
```

```
# Embed the text input into a dense representation
```

```
text_features = layers.Embedding(input_dim=10000, output_dim=64)(input_text)
```

```
# Use an LSTM layer to process the embedded text
```

```
text_features = layers.LSTM(64)(text_features)
```

```
# Step 3: Define the Image Branch
```

```
# Apply a series of convolutional and pooling layers to process the image input
```

```
image_features = layers.Conv2D(32, (3, 3), activation='relu')(input_image)
```

```
image_features = layers.MaxPooling2D((2, 2))(image_features)
```

```
image_features = layers.Conv2D(64, (3, 3), activation='relu')(image_features)
```

```
image_features = layers.MaxPooling2D((2, 2))(image_features)
```

```
# Flatten the output from the convolutional layers
```

```
image_features = layers.Flatten()(image_features)
```

```
# Step 4: Concatenate the Features
```

```
# Combine the features from both branches
```

```
combined_features = layers.concatenate([text_features, image_features])
```

# Step 5: Add Shared Dense Layer

# Add a shared dense layer to process the combined features

```
shared_dense = layers.Dense(128, activation='relu')
```

```
combined_features = shared_dense(combined_features)
```

# Step 6: Define the Outputs

# Output A: Binary classification

```
output_a = layers.Dense(1, activation='sigmoid', name='output_a')(combined_features)
```

# Output B: Multi-class classification

```
output_b = layers.Dense(10, activation='softmax', name='output_b')(combined_features)
```

# Step 7: Create the Model

# Define the model with the specified inputs and outputs

```
model = tf.keras.Model(inputs=[input_text, input_image], outputs=[output_a, output_b])
```

# Step 8: Summarize the Model

```
model.summary()
```

Model: "functional\_4"

Layer (type)	Output Shape	Param #	Connected to
image_input (InputLayer)	(None, 64, 64, 3)	0	-
conv2d (Conv2D)	(None, 62, 62, 32)	896	image_input[0][0]
max_pooling2d (MaxPooling2D)	(None, 31, 31, 32)	0	conv2d[0][0]
text_input (InputLayer)	(None, 100)	0	-
conv2d_1 (Conv2D)	(None, 29, 29, 64)	18,496	max_pooling2d[0][0]
embedding (Embedding)	(None, 100, 64)	640,000	text_input[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 14, 14, 64)	0	conv2d_1[0][0]
lstm (LSTM)	(None, 64)	33,024	embedding[0][0]
flatten (Flatten)	(None, 12544)	0	max_pooling2d_1[0][0]
concatenate (Concatenate)	(None, 12608)	0	lstm[0][0], flatten[0][0]
dense_3 (Dense)	(None, 128)	1,613,952	concatenate[0][0]
output_a (Dense)	(None, 1)	129	dense_3[0][0]
output_b (Dense)	(None, 10)	1,290	dense_3[0][0]

Total params: 2,307,787 (8.80 MB)  
 Trainable params: 2,307,787 (8.80 MB)  
 Non-trainable params: 0 (0.00 B)

## Dense Layers

Dense layers, also known as fully connected layers, are a fundamental building block in neural networks. They are responsible for transforming the input into an output by applying a linear transformation followed by a non-linear activation function.

### Key Concepts

1. **Weights and Biases:** Each dense layer contains weights and biases that are learned during training.
2. **Activation Function:** Non-linear functions applied to the output of the linear transformation to introduce non-linearity into the model.

```
import tensorflow as tf
```

```
from tensorflow.keras import layers
```

```
# Step 1: Define the Input Layer
```

```
input_layer = tf.keras.Input(shape=(32,), name='input_layer') # Input layer with 32 features
```

```
# Step 2: Add Dense Layers
```

```
# First dense layer with 64 units and ReLU activation
```

```

dense_1 = layers.Dense(64, activation='relu', name='dense_1')(input_layer)

# Second dense layer with 64 units and ReLU activation
dense_2 = layers.Dense(64, activation='relu', name='dense_2')(dense_1)

# Third dense layer with 32 units and ReLU activation
dense_3 = layers.Dense(32, activation='relu', name='dense_3')(dense_2)

# Step 3: Output Layer

# Output layer with 10 units and softmax activation for classification
output_layer = layers.Dense(10, activation='softmax', name='output_layer')(dense_3)

# Step 4: Create the Model
model = tf.keras.Model(inputs=input_layer, outputs=output_layer)

# Step 5: Summarize the Model
model.summary()

```

Model: "functional\_5"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 32)	0
dense_1 (Dense)	(None, 64)	2,112
dense_2 (Dense)	(None, 64)	4,160
dense_3 (Dense)	(None, 32)	2,080
output_layer (Dense)	(None, 10)	330

Total params: 8,682 (33.91 KB)  
 Trainable params: 8,682 (33.91 KB)  
 Non-trainable params: 0 (0.00 B)

## Convolutional Layers

Convolutional layers are a core component of convolutional neural networks (CNNs), which are widely used for processing image data and other grid-like data such as time-series data. TensorFlow/Keras provides several types of convolutional layers: Conv1D, Conv2D, and Conv3D.

### Conv1D

Conv1D is used for sequence data, such as time series or text data. It performs a 1D convolution along the time axis.

```
import tensorflow as tf
```

```
from tensorflow.keras import layers
```

```
# Define the input layer for 1D convolution
```

```
input_1d = tf.keras.Input(shape=(100, 1), name='input_1d') # 100 timesteps with 1 feature each
```

```
# Add a Conv1D layer
```

```
conv1d_layer = layers.Conv1D(filters=32, kernel_size=3, activation='relu',  
name='conv1d_layer')(input_1d)
```

```
# Add a MaxPooling1D layer
```

```
maxpool1d_layer = layers.MaxPooling1D(pool_size=2, name='maxpool1d_layer')(conv1d_layer)
```

```
# Flatten the output
```

```
flatten_layer = layers.Flatten()(maxpool1d_layer)
```

```
# Add a Dense layer for classification
```

```
dense_layer = layers.Dense(10, activation='softmax', name='dense_layer')(flatten_layer)
```

```
# Create the model
```

```
model_1d = tf.keras.Model(inputs=input_1d, outputs=dense_layer)
```

```
model_1d.summary()
```

Model: "functional\_6"

Layer (type)	Output Shape	Param #
input_1d (InputLayer)	(None, 100, 1)	0
conv1d_layer (Conv1D)	(None, 98, 32)	128
maxpool1d_layer (MaxPooling1D)	(None, 49, 32)	0
flatten_1 (Flatten)	(None, 1568)	0
dense_layer (Dense)	(None, 10)	15,690

Total params: 15,818 (61.79 KB)  
Trainable params: 15,818 (61.79 KB)  
Non-trainable params: 0 (0.00 B)

## Conv2D

Conv2D is used for image data, performing a 2D convolution on image height and width.

```
import tensorflow as tf

from tensorflow.keras import layers

# Define the input layer for 2D convolution
input_2d = tf.keras.Input(shape=(64, 64, 3), name='input_2d') # 64x64 RGB image

# Add a Conv2D layer
conv2d_layer = layers.Conv2D(filters=32, kernel_size=(3, 3), activation='relu',
name='conv2d_layer')(input_2d)

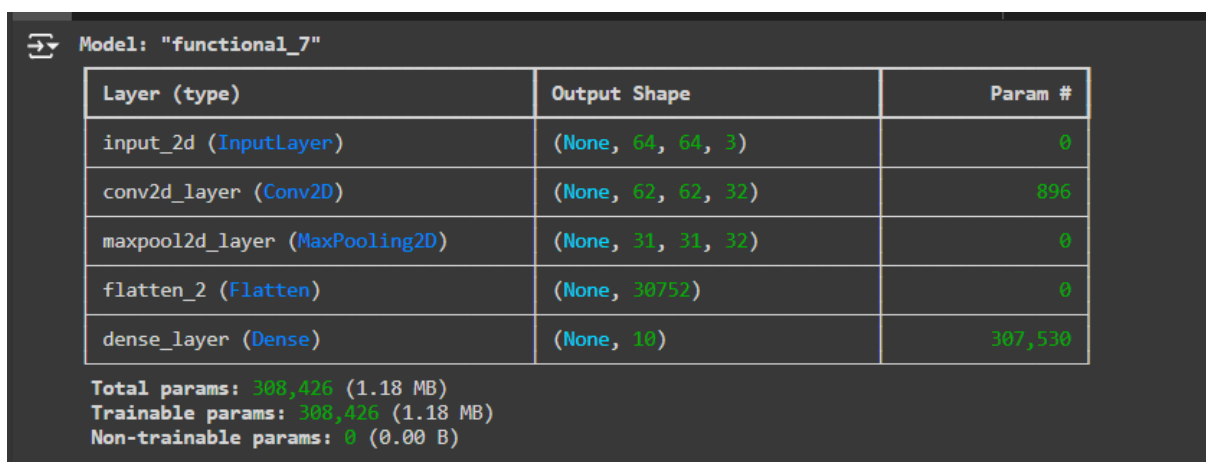
# Add a MaxPooling2D layer
maxpool2d_layer = layers.MaxPooling2D(pool_size=(2, 2), name='maxpool2d_layer')(conv2d_layer)

# Flatten the output
flatten_layer = layers.Flatten()(maxpool2d_layer)

# Add a Dense layer for classification
dense_layer = layers.Dense(10, activation='softmax', name='dense_layer')(flatten_layer)

# Create the model
model_2d = tf.keras.Model(inputs=input_2d, outputs=dense_layer)

model_2d.summary()
```



The screenshot shows a Jupyter Notebook interface with a dark theme. At the top, there is a tab labeled 'Model: "functional\_7"'. Below the tab is a table summarizing the model's layers. The table has three columns: 'Layer (type)', 'Output Shape', and 'Param #'. The layers listed are: 'input\_2d (InputLayer)' with output shape '(None, 64, 64, 3)' and 0 parameters; 'conv2d\_layer (Conv2D)' with output shape '(None, 62, 62, 32)' and 896 parameters; 'maxpool2d\_layer (MaxPooling2D)' with output shape '(None, 31, 31, 32)' and 0 parameters; 'flatten\_2 (Flatten)' with output shape '(None, 30752)' and 0 parameters; and 'dense\_layer (Dense)' with output shape '(None, 10)' and 307,530 parameters. Below the table, the total number of parameters is 308,426 (1.18 MB), with 308,426 trainable parameters (1.18 MB) and 0 non-trainable parameters (0.00 B).

Layer (type)	Output Shape	Param #
input_2d (InputLayer)	(None, 64, 64, 3)	0
conv2d_layer (Conv2D)	(None, 62, 62, 32)	896
maxpool2d_layer (MaxPooling2D)	(None, 31, 31, 32)	0
flatten_2 (Flatten)	(None, 30752)	0
dense_layer (Dense)	(None, 10)	307,530

Total params: 308,426 (1.18 MB)  
Trainable params: 308,426 (1.18 MB)  
Non-trainable params: 0 (0.00 B)

## Conv3D



Conv3D is used for volumetric data, such as video frames or 3D medical images.

```
import tensorflow as tf
```

```
from tensorflow.keras import layers
```

```
# Define the input layer for 3D convolution
```

```
input_3d = tf.keras.Input(shape=(64, 64, 64, 1), name='input_3d') # 64x64x64 volume with 1 channel
```

```
# Add a Conv3D layer
```

```
conv3d_layer = layers.Conv3D(filters=32, kernel_size=(3, 3, 3), activation='relu', name='conv3d_layer')(input_3d)
```

```
# Add a MaxPooling3D layer
```

```
maxpool3d_layer = layers.MaxPooling3D(pool_size=(2, 2, 2), name='maxpool3d_layer')(conv3d_layer)
```

```
# Flatten the output
```

```
flatten_layer = layers.Flatten()(maxpool3d_layer)
```

```
# Add a Dense layer for classification
```

```
dense_layer = layers.Dense(10, activation='softmax', name='dense_layer')(flatten_layer)
```

```
# Create the model
```

```
model_3d = tf.keras.Model(inputs=input_3d, outputs=dense_layer)
```

```
model_3d.summary()
```

Model: "functional_8"		
Layer (type)	Output Shape	Param #
input_3d (InputLayer)	(None, 64, 64, 64, 1)	0
conv3d_layer (Conv3D)	(None, 62, 62, 62, 32)	896
maxpool3d_layer (MaxPooling3D)	(None, 31, 31, 31, 32)	0
flatten_3 (Flatten)	(None, 953312)	0
dense_layer (Dense)	(None, 10)	9,533,130
Total params: 9,534,026 (36.37 MB)		
Trainable params: 9,534,026 (36.37 MB)		
Non-trainable params: 0 (0.00 B)		

## Recurrent Layers

Recurrent layers are used for sequence modeling, capturing temporal dependencies in sequential data. Common types are SimpleRNN, LSTM, and GRU.

### SimpleRNN

A basic recurrent neural network layer.

```
import tensorflow as tf
```

```
from tensorflow.keras import layers
```

```
# Define the input layer for RNN
```

```
input_rnn = tf.keras.Input(shape=(100, 1), name='input_rnn') # 100 timesteps with 1 feature each
```

```
# Add a SimpleRNN layer
```

```
simplernn_layer = layers.SimpleRNN(64, activation='relu', name='simplernn_layer')(input_rnn)
```

```
# Add a Dense layer for classification
```

```
dense_layer = layers.Dense(10, activation='softmax', name='dense_layer')(simplernn_layer)
```

```
# Create the model
```

```
model_rnn = tf.keras.Model(inputs=input_rnn, outputs=dense_layer)
```

```
model_rnn.summary()
```

Model: "functional\_9"

Layer (type)	Output Shape	Param #
input_rnn (InputLayer)	(None, 100, 1)	0
simplernn_layer (SimpleRNN)	(None, 64)	4,224
dense_layer (Dense)	(None, 10)	650

Total params: 4,874 (19.04 KB)  
 Trainable params: 4,874 (19.04 KB)  
 Non-trainable params: 0 (0.00 B)

## LSTM

Long Short-Term Memory network, which is capable of learning long-term dependencies.

```
import tensorflow as tf
```

```
from tensorflow.keras import layers
```

```
# Define the input layer for LSTM
```

```
input_lstm = tf.keras.Input(shape=(100, 1), name='input_lstm') # 100 timesteps with 1 feature each
```

```
# Add an LSTM layer
```

```
lstm_layer = layers.LSTM(64, name='lstm_layer')(input_lstm)
```

```
# Add a Dense layer for classification
```

```
dense_layer = layers.Dense(10, activation='softmax', name='dense_layer')(lstm_layer)
```

```
# Create the model
```

```
model_lstm = tf.keras.Model(inputs=input_lstm, outputs=dense_layer)
```

```
model_lstm.summary()
```

Model: "functional\_10"

Layer (type)	Output Shape	Param #
input_lstm (InputLayer)	(None, 100, 1)	0
lstm_layer (LSTM)	(None, 64)	16,896
dense_layer (Dense)	(None, 10)	650

Total params: 17,546 (68.54 KB)  
 Trainable params: 17,546 (68.54 KB)  
 Non-trainable params: 0 (0.00 B)

## GRU

Gated Recurrent Unit, a simpler and faster variant of LSTM.

```
import tensorflow as tf
```

```
from tensorflow.keras import layers
```

```
# Define the input layer for GRU
```

```
input_gru = tf.keras.Input(shape=(100, 1), name='input_gru') # 100 timesteps with 1 feature each
```

```
# Add a GRU layer
```

```
gru_layer = layers.GRU(64, name='gru_layer')(input_gru)
```

```
# Add a Dense layer for classification
```

```
dense_layer = layers.Dense(10, activation='softmax', name='dense_layer')(gru_layer)
```

```
# Create the model
```

```
model_gru = tf.keras.Model(inputs=input_gru, outputs=dense_layer)
```

```
model_gru.summary()
```

Model: "functional_11"		
Layer (type)	Output Shape	Param #
input_gru (InputLayer)	(None, 100, 1)	0
gru_layer (GRU)	(None, 64)	12,864
dense_layer (Dense)	(None, 10)	650

Total params: 13,514 (52.79 KB)  
 Trainable params: 13,514 (52.79 KB)  
 Non-trainable params: 0 (0.00 B)

Custom Layer Creation: Building custom layers in Keras using the subclassing API. Defining the forward pass and backward pass methods. Incorporating custom layers into models.

Creating custom layers in Keras using the subclassing API allows you to define unique operations that aren't covered by the built-in layers. This is done by subclassing `tf.keras.layers.Layer` and defining the necessary methods, such as `build`, `call`, and optionally, `compute_output_shape`.

Key Methods for Custom Layers

1. `__init__`: Initialization method where you can define the layer's parameters.
2. `build`: Method where you define the weights of the layer. It is called once the shape of the inputs is known.
3. `call`: Method that defines the forward pass logic. This is where the computation happens.
4. `compute_output_shape`: Method that computes the output shape of the layer (optional).

```
import tensorflow as tf
```

```
from tensorflow.keras import layers, models, initializers, regularizers
```

```
# Custom Dense Layer
```

```
class MyDenseLayer(layers.Layer):
```

```
    def __init__(self, units=32, activation=None, **kwargs):
```

```
        super(MyDenseLayer, self).__init__(**kwargs)
```

```
        self.units = units
```

```
        self.activation = tf.keras.activations.get(activation)
```

```
    def build(self, input_shape):
```

```
        # Add weight and bias parameters
```

```
        self.w = self.add_weight(
```

```

        shape=(input_shape[-1], self.units),
        initializer=initializers.GlorotUniform(),
        trainable=True,
        name='kernel'
    )
    self.b = self.add_weight(
        shape=(self.units,),
        initializer=initializers.Zeros(),
        trainable=True,
        name='bias'
    )

```

```

def call(self, inputs):
    # Forward pass: output = inputs @ W + b
    output = tf.matmul(inputs, self.w) + self.b
    if self.activation:
        output = self.activation(output)
    return output

```

```

def compute_output_shape(self, input_shape):
    return (input_shape[0], self.units)

```

# Testing the Custom Layer

# Define input

```
inputs = tf.keras.Input(shape=(64,))
```

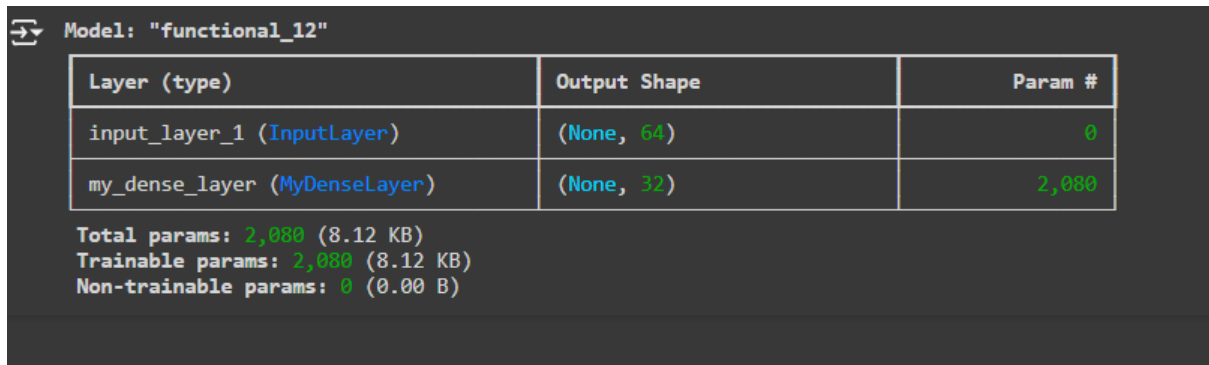
# Add custom dense layer

```
outputs = MyDenseLayer(units=32, activation='relu')(inputs)
```

# Create the model

```
model = models.Model(inputs=inputs, outputs=outputs)
```

```
model.summary()
```



The screenshot shows a Jupyter Notebook interface with a dark theme. At the top, it says 'Model: "functional\_12"'. Below this is a table with three columns: 'Layer (type)', 'Output Shape', and 'Param #'. The table contains two rows: 'input\_layer\_1 (InputLayer)' with output shape '(None, 64)' and 0 parameters, and 'my\_dense\_layer (MyDenseLayer)' with output shape '(None, 32)' and 2,080 parameters. Below the table, it lists 'Total params: 2,080 (8.12 KB)', 'Trainable params: 2,080 (8.12 KB)', and 'Non-trainable params: 0 (0.00 B)'.

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 64)	0
my_dense_layer (MyDenseLayer)	(None, 32)	2,080

Total params: 2,080 (8.12 KB)  
Trainable params: 2,080 (8.12 KB)  
Non-trainable params: 0 (0.00 B)

Pooling Layers using MaxPooling2D, GlobalMaxPooling1D, and

AveragePooling3D for downsampling and feature extraction.

Pooling Layers

Pooling layers are used to downsample feature maps, reducing their dimensionality and computational complexity while retaining important features. Common types of pooling layers include MaxPooling, AveragePooling, and GlobalPooling.

MaxPooling2D

MaxPooling2D performs max pooling operation on spatial data (e.g., images), which takes the maximum value from a patch of feature maps.

```
import tensorflow as tf
```

```
from tensorflow.keras import layers, models
```

```
# Define the input layer for 2D convolution
```

```
input_2d = tf.keras.Input(shape=(64, 64, 3), name='input_2d') # 64x64 RGB image
```

```
# Add a Conv2D layer
```

```
conv2d_layer = layers.Conv2D(filters=32, kernel_size=(3, 3), activation='relu',  
name='conv2d_layer')(input_2d)
```

```
# Add a MaxPooling2D layer
```

```
maxpool2d_layer = layers.MaxPooling2D(pool_size=(2, 2), name='maxpool2d_layer')(conv2d_layer)
```

```
# Flatten the output
```

```
flatten_layer = layers.Flatten()(maxpool2d_layer)
```

```
# Add a Dense layer for classification
```

```
dense_layer = layers.Dense(10, activation='softmax', name='dense_layer')(flatten_layer)
```

```
# Create the model
```

```
model_maxpool2d = models.Model(inputs=input_2d, outputs=dense_layer)
```

```
model_maxpool2d.summary()
```

input_2d (InputLayer)	(None, 64, 64, 3)	0
conv2d_layer (Conv2D)	(None, 62, 62, 32)	896
maxpool2d_layer (MaxPooling2D)	(None, 31, 31, 32)	0
flatten_4 (Flatten)	(None, 30752)	0
dense_layer (Dense)	(None, 10)	307,530
Total params: 308,426 (1.18 MB)		
Trainable params: 308,426 (1.18 MB)		
Non-trainable params: 0 (0.00 B)		

GlobalMaxPooling1D

GlobalMaxPooling1D performs global max pooling operation on 1D data, which takes the maximum value over the time dimension.

```
import tensorflow as tf
```

```
from tensorflow.keras import layers, models
```

```
# Define the input layer for 1D convolution
```

```
input_1d = tf.keras.Input(shape=(100, 64), name='input_1d') # 100 timesteps with 64 features each
```

```
# Add a Conv1D layer
```

```
conv1d_layer = layers.Conv1D(filters=32, kernel_size=3, activation='relu',  
name='conv1d_layer')(input_1d)
```

```
# Add a GlobalMaxPooling1D layer
```

```
globalmaxpool1d_layer =  
layers.GlobalMaxPooling1D(name='globalmaxpool1d_layer')(conv1d_layer)
```



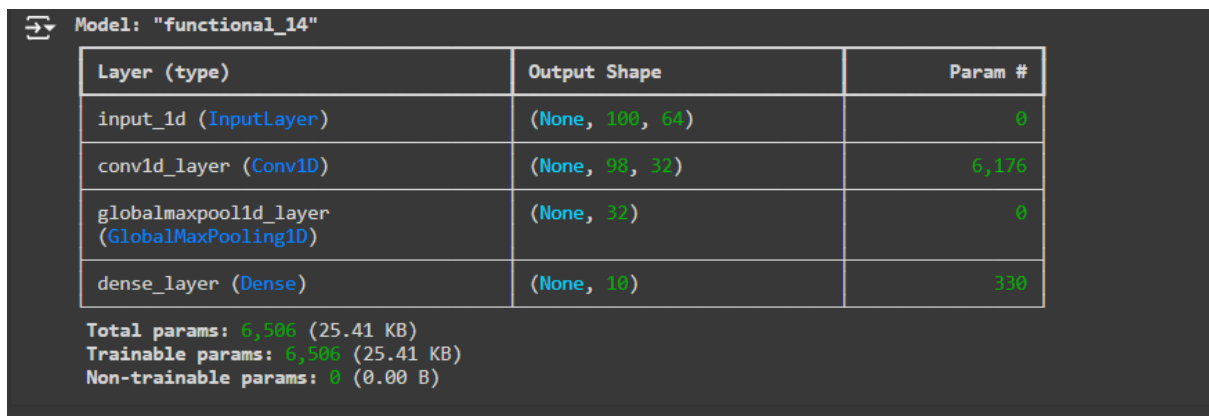
```
# Add a Dense layer for classification
```

```
dense_layer = layers.Dense(10, activation='softmax', name='dense_layer')(globalmaxpool1d_layer)
```

```
# Create the model
```

```
model_globalmaxpool1d = models.Model(inputs=input_1d, outputs=dense_layer)
```

```
model_globalmaxpool1d.summary()
```



The screenshot shows the summary of a Keras model named "functional\_14". It contains a table with 3 columns: Layer (type), Output Shape, and Param #. The layers are: input\_1d (InputLayer) with output shape (None, 100, 64) and 0 parameters; conv1d\_layer (Conv1D) with output shape (None, 98, 32) and 6,176 parameters; globalmaxpool1d\_layer (GlobalMaxPooling1D) with output shape (None, 32) and 0 parameters; and dense\_layer (Dense) with output shape (None, 10) and 330 parameters. Below the table, it states: Total params: 6,506 (25.41 KB), Trainable params: 6,506 (25.41 KB), and Non-trainable params: 0 (0.00 B).

Layer (type)	Output Shape	Param #
input_1d (InputLayer)	(None, 100, 64)	0
conv1d_layer (Conv1D)	(None, 98, 32)	6,176
globalmaxpool1d_layer (GlobalMaxPooling1D)	(None, 32)	0
dense_layer (Dense)	(None, 10)	330

Total params: 6,506 (25.41 KB)  
Trainable params: 6,506 (25.41 KB)  
Non-trainable params: 0 (0.00 B)

## AveragePooling3D

AveragePooling3D performs average pooling operation on volumetric data (e.g., 3D images or video frames), which takes the average value from a patch of feature maps.

```
import tensorflow as tf
```

```
from tensorflow.keras import layers, models
```

```
# Define the input layer for 3D convolution
```

```
input_3d = tf.keras.Input(shape=(64, 64, 64, 1), name='input_3d') # 64x64x64 volume with 1 channel
```

```
# Add a Conv3D layer
```

```
conv3d_layer = layers.Conv3D(filters=32, kernel_size=(3, 3, 3), activation='relu', name='conv3d_layer')(input_3d)
```

```
# Add an AveragePooling3D layer
```

```
averagepool3d_layer = layers.AveragePooling3D(pool_size=(2, 2, 2), name='averagepool3d_layer')(conv3d_layer)
```

```
# Flatten the output
```

```
flatten_layer = layers.Flatten()(averagepool3d_layer)
```

```
# Add a Dense layer for classification
```

```
dense_layer = layers.Dense(10, activation='softmax', name='dense_layer')(flatten_layer)
```

```
# Create the model
```

```
model_averagepool3d = models.Model(inputs=input_3d, outputs=dense_layer)
```

```
model_averagepool3d.summary()
```

Model: "functional\_15"

Layer (type)	Output Shape	Param #
input_3d (InputLayer)	(None, 64, 64, 64, 1)	0
conv3d_layer (Conv3D)	(None, 62, 62, 62, 32)	896
averagepool3d_layer (AveragePooling3D)	(None, 31, 31, 31, 32)	0
flatten_5 (Flatten)	(None, 953312)	0
dense_layer (Dense)	(None, 10)	9,533,130

Total params: 9,534,026 (36.37 MB)  
Trainable params: 9,534,026 (36.37 MB)  
Non-trainable params: 0 (0.00 B)