There are a number of ways that you can customize layer behavior in tensorflow. The first and the easiest is if you only need basic functionality as to not create a custom layer at all, but to use a lambda layer. This is a layer type that can be used to execute arbitrary code.

The purpose of the lambda layer, like I said, is to execute an arbitrary function within a sequential or a functional API model. It's best-suited for something quick and simple or if you want to experiment.

```
tf.keras.layers.Lambda(lambda x: tf.abs(x))
```

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

```
if(x>0):
    return x
else:
    return 0
```

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128),
  tf.keras.layers.Lambda(lambda x: tf.abs(x)),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

Some commonly used layers

Convolutional
Conv1D/Conv2D/Conv3D
SeparableConv2D
DepthwiseConv2D

Recurrent
LSTM
GRU

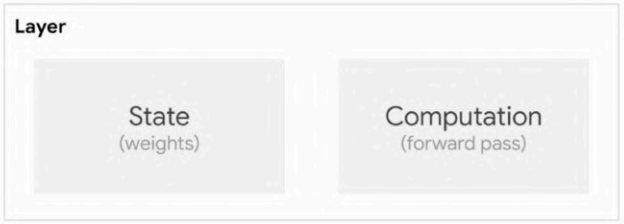
Pooling
MaxPooling2D
AveragePooling2D
GlobalAveragePooling2D

Merge
Add
Subtract
Multiply

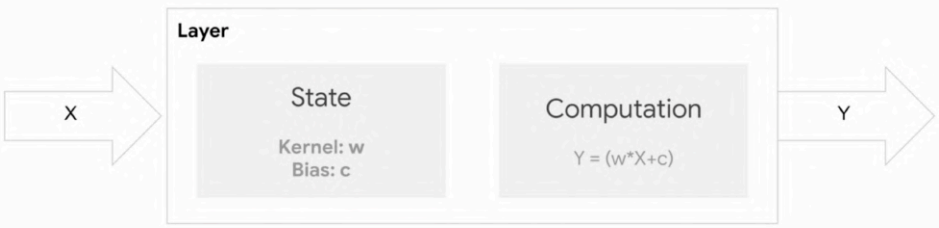
Activations (Advanced)
LeakyReLU
PReLU
ELU

Core
Activation
Lambda
Input
Dense
Dropout
BatchNormalization

What is a Layer?



Simple Dense Layer



```
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to outputs
      return tf.matmul(inputs, self.w) + self.b
```

Here's the complete code for this layer type, and I'm going to call this SimpleDense.

When creating a layer, you inherit from Keras's layer class by specifying it in parentheses after your class name like this.

```
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to outputs
      return tf.matmul(inputs, self.w) + self.b
```

Then your class will need at least these three methods.

The first of them, init, will initialize the class that accepts the parameters and it sets up the internal variables.

```
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to outputs
      return tf.matmul(inputs, self.w) + self.b
```

The second, build, will run when your instance is created. You'll use this to specify your local input states and any other housekeeping that's needed for that creation.

```
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs): # Defines the computation from inputs to outputs
      return tf.matmul(inputs, self.w) + self.b
```

The third, call, performs the computation and it's called during training to get the output from this cell.

```
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to outputs
      return tf.matmul(inputs, self.w) + self.b
```

Let's go back to init. The first thing that it needs to do is pass any initialization back to the base class. Remember that this is inheriting from the layer class, so some initialization needs to be performed there too, and that's done using the super keyword.

Then a local class variable called units will be set up to the parameter value of units that was passed in, will default to 32 units in this case, so if nothing is specified, this layer will have 32 units init.

```
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to outputs
      return tf.matmul(inputs, self.w) + self.b
```

Within the build, you'll initialize the states. In this case, we're calling them w and b. Remember that when we create the layer, we're not creating a single neuron, but a number of neurons specified by the units variable.

Every neuron will need to be initialized, and TensorFlow supports a number of built-in functions to initialize there values. One of these is the random normal initializer, which as its name suggests, initializes them randomly using a normal distribution.

```
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to outputs
      return tf.matmul(inputs, self.w) + self.b
```

```python
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to outputs
      return tf.matmul(inputs, self.w) + self.b
```

self.w will hold the states of the w's and they'll be in a tensor by creating them as a tf.Variable. This will be initialized using the w_init for its values, it's given the name kernels so that we can trace it later.

```python
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to outputs
      return tf.matmul(inputs, self.w) + self.b
```

Note that it's set up to be trainable, so when you're doing a model.fit, the value of w can be modified by TensorFlow.

```python
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to outputs
      return tf.matmul(inputs, self.w) + self.b
```

The bias is initialized differently using a tf.zeros_initializer function, which as the name suggests, will set it to zero.

self.b will then be a tensor of the number of units in the layer, and they'll all be initialized as zeros. As you can see, that'll also be trainable.

```
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to outputs
      return tf.matmul(inputs, self.w) + self.b
```

Call will do the computation, and as our self.w and self.b are tensors, we could do a matmul operation on them to multiply the inputs by w and then add b before returning it. The inputs here are our typical x-value, so y will be wx plus b.

```
my_dense = SimpleDense(units=1)
x = tf.ones((1, 1))
y = my_dense(x)
print(my_dense.variables)
```

Now let's look at it in action. Here's some simple code. In this case, I'll create a dense layer called SimpleDense and I'll initialize it with just one neuron. I'm going to initialize an x as a tensor with tf.ones((1, 1)), which returns a tensor of the shapes specified filled in with ones, so this will give me a one-by-one tensor, which contains the value one. I'm going to say y equals my_dense(x), so that a dense layer will be initialized. It has a single unit, so it will get the value of x.

```
my_dense = SimpleDense(units=1)
x = tf.ones((1, 1))
y = my_dense(x)
print(my_dense.variables)

  [<tf.Variable 'simple_dense_7/kernel:0' shape=(1, 1)
  dtype=float32, numpy=array([[0.03688493]], dtype=float32)>,

  <tf.Variable 'simple_dense_7/bias:0' shape=(1,)
  dtype=float32, numpy=array([0.], dtype=float32)>]
```

After this, we can look at the variables inside my dense, and we can see how they've been initialized by looking at the tensors.

The kernel or w received a random normal distribution and ended up with 0.036. The bias, as we saw before, is initialized with zeros, so it contains zero.

```python
import numpy as np

xs = np.array([-1.0,  0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model = tf.keras.Sequential([SimpleDense(units=1)])
model.compile(optimizer='sgd', loss='mean_squared_error')
model.fit(xs, ys, epochs=500,verbose=0)
print(model.predict([10.0]))
```

Expected Answer: 19 (y=2x-1)

Actual Answer: 0.36

(W = 0.036, B=0 => Y=.036 * 10 + 0 = 0.36)

```python
import numpy as np

xs = np.array([-1.0,  0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model = tf.keras.Sequential([SimpleDense(units=1)])
model.compile(optimizer='sgd', loss='mean_squared_error')
model.fit(xs, ys, epochs=500,verbose=0)
print(model.predict([10.0]))
```

```
[[18.981468]]
```

```
[<tf.Variable
'sequential_15/simple_dense_19/kernel:0' shape=(1, 1)
dtype=float32, numpy=array([[1.9972587]],
dtype=float32)>, <tf.Variable
'sequential_15/simple_dense_19/bias:0' shape=(1,)
dtype=float32, numpy=array([-0.991501],
dtype=float32)>]
```

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])
```

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  SimpleDense(128),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])
```

If you wanted to use simple dense, you could declare it like this. This model without the ReLu, won't perform as well as the previous architecture because of the positive impact using ReLu has as an activation function.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  SimpleDense(128),
  tf.keras.layers.Lambda(my_relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])
```

But you could use a lambda function with the ReLu you implemented earlier and then your model will perform quite well.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])
```

Your custom layer was missing the ability to do an activation on them. There was a workaround that we did using a lambda layer, but it's simpler and cleaner to specify an activation function on a layer. Let's take a look at how to expand our dense class to be able to do that.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  SimpleDense(128),
  tf.keras.layers.Lambda(my_relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])
```

When using our simple dense layer, we didn't have the facility to specify ReLU, but a workaround was to implement our own ReLU function and activate it using a lambda layer. This worked pretty well, but it's a bit hacky.

```
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to c
      return tf.matmul(inputs, self.w) + self.b
```

```python
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to c
      return tf.matmul(inputs, self.w) + self.b
```

```python
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to c
      return tf.matmul(inputs, self.w) + self.b
```

```python
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(name="kernel",
        initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
        trainable=True)

    b_init = tf.zeros_initializer()
    self.b = tf.Variable(name="bias",
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to c
      return tf.matmul(inputs, self.w) + self.b
```

```
class SimpleDense(Layer):

  def __init__(self, units=32, activation=None):
      super(SimpleDense, self).__init__()
      self.units = units
      self.activation = tf.keras.activations.get(activation)


  def call(self, inputs):
      return self.activation(tf.matmul(inputs, self.w) + self.b)
```

To update the layer implementation for activations, we don't need to change the build. I'll omit that for clarity, we only need to edit the init and call functions. In the init function, we have to specify that we'll accept an activation function.

The activation function can either be a string containing the name of the function or an instance of an activation object. We can default it to none so that if we don't receive the parameter, we won't use any activation function at all.

```
class SimpleDense(Layer):

  def __init__(self, units=32, activation=None):
      super(SimpleDense, self).__init__()
      self.units = units
      self.activation = tf.keras.activations.get(activation)


  def call(self, inputs):
      return self.activation(tf.matmul(inputs, self.w) + self.b)
```

Then we can set our self.activation variable to be the value of tf.Keras.activations.get(), with this activation name. This will set self.activation to be an instance of the named activation function. For example, if we pass ReLU as the activation, Keras will give us a ReLU function as self dot activation.

Remember, you can pass either a string naming the activation function or an object instance of one. If you pass something invalid, your code will fail at this line.

```
class SimpleDense(Layer):

  def __init__(self, units=32, activation=None):
      super(SimpleDense, self).__init__()
      self.units = units
      self.activation = tf.keras.activations.get(activation)


  def call(self, inputs):
      return self.activation(tf.matmul(inputs, self.w) + self.b)
```

Then in call, as you might be familiar with, we calculate the return value on the layer to be the inputs times w plus b. But we now need to activate that. For example, with ReLU, if the value of that is less than zero, we just return zero. Activating is as simple as calling the activation function with the results of the calculation like this and then returning that.