Published in Towards Data Science

Innat
Nov 8, 2020 · 21 min read · Listen

Model Sub-Classing and Custom Training Loop from Scratch in TensorFlow 2

A gentle introduction to the various components of model subclassing and custom training loop in TF.Keras.

Image by Nima Sarram on Unsplash

In this article, we will try to understand the **Model Sub-Classing API** and **Custom Training Loop from Scratch** in **TensorFlow 2**. It may not be a beginner or advanced introduction but aim to get rough intuition of what they are all about. The post is divided into three parts:

1. **Comparable Modelling Strategies in TensorFlow 2**
2. **Build an Inception Network with Model Sub-Classing API**
3. **End-to-End Training with Custom Training Loop from Scratch**

So, at first, we will see how many ways to define model using TensorFlow 2 and how they differ from each other. Next, we will see how feasible it is to build a complex neural architecture using the model subclassing API which is introduced in TF 2. And then we will implement a custom training loop and train these subclassing models end-to-end from scratch. We will also use Tensorboard in our custom training loop to track the model performance for each batch. We will also see how to save and load the model after training. In the end, we will measure the model performance via the confusion matrix and classification report, etc.

Comparable Modelling Strategies in TensorFlow 2

In *TF.Keras* there are basically three-way we can define a neural network, namely

- **Sequential API**
- **Functional API**
- **Model Subclassing API**

Among them, **Sequential API** is the easiest way to implement but comes with certain limitations. For example, with this API, we can't create a model that shares feature information with another layer except to its subsequent layer. In addition, multiple input and output are not possible to implement either. At this point, **Functional API** does solve these issues greatly. A model like Inception or ResNet is feasible to implement in Functional API. But often deep learning researcher wants to have more control over every nuance of the network and on the training pipelines and that's exactly what **Model Subclassing API** serves. Model Sub-Classing is a fully customizable way to implement the feed-forward mechanism for the custom-designed deep neural network in an object-oriented fashion.

Let's create a very basic neural network using these three APIs. It will be the same neural architecture and will see what are the implementation differences. This of course will not demonstrate the full potential, especially for **Functional** and **Model Sub-Classing API**. The architecture will be as follows:

```
Input - > Conv - > MaxPool - > BN - > Conv -> BN - > Droput - > GAP -
> Dense
```

Simple enough. As mentioned, let's create the neural nets with Sequential, Functional, and Model Sub-Classing respectively.

Sequential API

```
# declare input shape
seq_model = tf.keras.Sequential()
seq_model.add(tf.keras.Input(shape=imput_dim))

# Block 1
seq_model.add(tf.keras.layers.Conv2D(32, 3, strides=2, activation="relu"))
seq_model.add(tf.keras.layers.MaxPooling2D(3))
seq_model.add(tf.keras.layers.BatchNormalization())

# Block 2
seq_model.add(tf.keras.layers.Conv2D(64, 3, activation="relu"))
seq_model.add(tf.keras.layers.BatchNormalization())
seq_model.add(tf.keras.layers.Dropout(0.3))

# Now that we apply global max pooling.
seq_model.add(tf.keras.layers.GlobalMaxPooling2D())

# Finally, we add a classification layer.
seq_model.add(tf.keras.layers.Dense(output_dim))
```

Functional API

```
# declare input shape
input = tf.keras.Input(shape=(imput_dim))

# Block 1
x = tf.keras.layers.Conv2D(32, 3, strides=2, activation="relu")(input)
x = tf.keras.layers.MaxPooling2D(3)(x)
x = tf.keras.layers.BatchNormalization()(x)

# Block 2
x = tf.keras.layers.Conv2D(64, 3, activation="relu")(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.Dropout(0.3)(x)

# Now that we apply global max pooling.
gap = tf.keras.layers.GlobalMaxPooling2D()(x)

# Finally, we add a classification layer.
output = tf.keras.layers.Dense(output_dim)(gap)

# bind all
func_model = tf.keras.Model(input, output)
```

Model Sub-Classing API

```
class ModelSubClassing(tf.keras.Model):
    def __init__(self, num_classes):
        super(ModelSubClassing, self).__init__()
        # define all layers in init
        # Layer of Block 1
        self.conv1 = tf.keras.layers.Conv2D(
                        32, 3, strides=2, activation="relu"
                    )
        self.max1  = tf.keras.layers.MaxPooling2D(3)
        self.bn1   = tf.keras.layers.BatchNormalization()

        # Layer of Block 2
        self.conv2 = tf.keras.layers.Conv2D(64, 3, activation="relu")
        self.bn2   = tf.keras.layers.BatchNormalization()
        self.drop  = tf.keras.layers.Dropout(0.3)

        # GAP, followed by Classifier
        self.gap   = tf.keras.layers.GlobalAveragePooling2D()
        self.dense = tf.keras.layers.Dense(num_classes)


    def call(self, input_tensor, training=False):
        # forward pass: block 1
        x = self.conv1(input_tensor)
        x = self.max1(x)
        x = self.bn1(x)
```

```
        # forward pass: block 2
        x = self.conv2(x)
        x = self.bn2(x)

        # droput followed by gap and classifier
        x = self.drop(x)
        x = self.gap(x)
        return self.dense(x)
```

In **Model Sub-Classing** there are two most important functions *__init__* and *call*. Basically, we will define all the trainable *tf.keras* layers or custom implemented layers inside the *__init__* method and call those layers based on our network design inside the *call* method which is used to perform a forward propagation. (It's quite the same as the *forward* method that is used to build the model in PyTorch anyway.)

Let's run these models on the MNIST data set. We will load from *tf.keras.datasets*. However, the input image is **28** by **28** and in grayscale shape. We will repeat the axis three times so that we can feasibly experiment with the pretrained weight later on if necessary.

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# x_train.shape, y_train.shape: (60000, 28, 28) (60000,)
# x_test.shape,  y_test.shape : (10000, 28, 28) (10000,)

# train set / data
x_train = np.expand_dims(x_train, axis=-1)
x_train = np.repeat(x_train, 3, axis=-1)
x_train = x_train.astype('float32') / 255
# train set / target
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)


# validation set / data
x_test = np.expand_dims(x_test, axis=-1)
x_test = np.repeat(x_test, 3, axis=-1)
x_test = x_test.astype('float32') / 255
# validation set / target
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)


# ------------------------------------------------------------------
# compile
print('Sequential API')
seq_model.compile(
          loss      = tf.keras.losses.CategoricalCrossentropy(),
          metrics   = tf.keras.metrics.CategoricalAccuracy(),
          optimizer = tf.keras.optimizers.Adam())
# fit
seq_model.fit(x_train, y_train, batch_size=128, epochs=1)


# compile
print('\nFunctional API')
func_model.compile(
          loss      = tf.keras.losses.CategoricalCrossentropy(),
          metrics   = tf.keras.metrics.CategoricalAccuracy(),
          optimizer = tf.keras.optimizers.Adam())
# fit
func_model.fit(x_train, y_train, batch_size=128, epochs=1)


# compile
print('\nModel Sub-Classing API')
sub_classing_model = ModelSubClassing(10)
sub_classing_model.compile(
          loss      = tf.keras.losses.CategoricalCrossentropy(),
          metrics   = tf.keras.metrics.CategoricalAccuracy(),
          optimizer = tf.keras.optimizers.Adam())
# fit
sub_classing_model.fit(x_train, y_train, batch_size=128, epochs=1);
```

Output

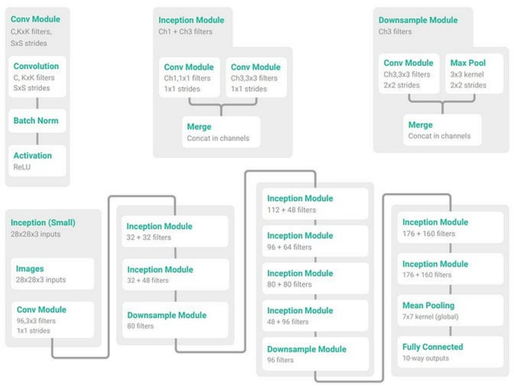
```
Sequential API
469/469 [==============================] - 2s 3ms/step - loss: 7.5747
- categorical_accuracy: 0.2516

Functional API
469/469 [==============================] - 2s 3ms/step - loss: 8.1335
- categorical_accuracy: 0.2368

Model Sub-Classing API
469/469 [==============================] - 2s 3ms/step - loss: 5.2695
- categorical_accuracy: 0.1731
```

Build an Inception Network with Model Sub-Classing API

The core data structures in TF.Keras is **layers** and **model** classes. A layer encapsulates both *state* (weight) and the transformation from inputs to outputs, i.e. the *call* method that is used to define the forward pass. However, these layers are also recursively composable. It means if we assign a *tf.keras.layers.Layer* instances as an attribute of another *tf.keras.layers.Layer*, the outer layer will start tracking the weights matrix of the inner layer. So, each layer will track the weights of its sublayers, both trainable and non-trainable. Such functionality is required when we need to build such a layer of a higher level of abstraction.

In this part, we will be building a small Inception model by subclassing the layers and model classes. Please see the diagram below. It's a small **Inception** network, src. If we give a close look we'll see that it mainly consists of three special modules, namely:

1. *Conv Module*
2. *Inception Module*
3. *Downsample Module*



A small Inception network, src.

Conv Module

From the diagram we can see, that it consists of one convolutional network, one batch normalization, and one relu activation. Also, it produces *C* times feature maps with *K x K* filters and *S x S* strides. Now, it would be very inefficient if we simply go with the sequential modeling approach because we will be re-using this module many times in the complete network. So, defining a functional block would be efficient and simple enough. But this time, we will prefer *layer subclassing* which is more *pythonic* and more efficient. To do that, we will create a class object that will inherit the *tf.keras.layers.Layer* classes.

Search Medium Write Sign up Sign In

```
    def __init__(
        self, kernel_num, kernel_size, strides, padding='same'
    ):
        super(ConvModule, self).__init__()
        # conv layer
        self.conv = tf.keras.layers.Conv2D(
                    kernel_num,
                    kernel_size=kernel_size,
                    strides=strides,
                    padding=padding
                )
        # batch norm layer
         self.bn   = tf.keras.layers.BatchNormalization()


    def call(self, input_tensor, training=False):
        x = self.conv(input_tensor)
        x = self.bn(x, training=training)
        x = tf.nn.relu(x)
        return x
```

Now, We can also initiate the object of this class and see the following properties.

Innat
38 Followers
AI Software Engineer @ AI Samurai Japan Ltd.
Follow
More from Medium
Molly Ru... in Towards Data Scie...
How ChatGPT Works: The Models Behind The Bot
Ahmed Be... in Towards Data Sc...
12 Python Decorators To Take Your Code To The Next Level
Arjun Sarkar
Implementation of all Loss

```
cm = ConvModule(96, (3,3), (1,1))
y = cm(tf.ones(shape=(2,32,32,3))) # first call to the `cm` will
create weights

print("weights:", len(cm.weights))
print("trainable weights:", len(cm.trainable_weights))


# output
weights: 6
trainable weights: 4
```

Inception Module

Next comes the Inception module. According to the above graph, it consists of two ***convolutional modules*** that then merge together. Now as we know to merge, here we need to ensure that the output feature maps dimension (***height*** and ***width***) needs to be the same.

```
class InceptionModule(tf.keras.layers.Layer):
    def __init__(self, kernel_size1x1, kernel_size3x3):
        super(InceptionModule, self).__init__()
        # two conv modules: they will take same input tensor
        self.conv1 = ConvModule(
                        kernel_size1x1, kernel_size=(1,1), strides=(1,1)
                      )
        self.conv2 = ConvModule(
                        kernel_size3x3, kernel_size=(3,3), strides=(1,1)
                      )
        self.cat   = tf.keras.layers.Concatenate()


    def call(self, input_tensor, training=False):
        x_1x1 = self.conv1(input_tensor)
        x_3x3 = self.conv2(input_tensor)
        x = self.cat([x_1x1, x_3x3])
        return x
```

Here you may notice that we are now hard-coded the exact ***kernel size*** and ***strides*** number for both convolutional layers according to the network (diagram). And also in ***ConvModule,*** we have already set ***padding*** to the '*same*', so that the dimension of the feature maps will be the same for both (***self.conv1*** and ***self.conv2***); which is required in order to concatenate them to the end.

Again, in this module, two variable performs as the placeholder, ***kernel_size1x1***, and ***kernel_size3x3***. This is for the purpose of course. Because we will need different numbers of feature maps for the different stages of the entire model. If we look into the diagram of the model, we will see that ***InceptionModule*** takes a different number of filters at different stages in the model.

Downsample Module

Lastly the downsampling module. The main intuition for downsampling is that we hope to get more relevant feature information that highly represents the inputs to the model. As it tends to remove the unwanted feature so that model can focus on the most relevant. There are many ways we can reduce the dimension of the feature maps (or inputs). For example: using ***strides*** **2** or using the conventional ***pooling*** operation. There are many types of pooling operations, namely: **MaxPooling, AveragePooling, and GlobalAveragePooling**.

From the diagram, we can see that the downsampling module contains one convolutional layer and one max-pooling layer which later merge together. Now, if we look closely at the diagram (top-right), we will see that the convolutional layer takes a **3** x **3** size filter with strides **2** x **2**. And the pooling layer (here *MaxPooling*) takes pooling size **3** x **3** with strides **2** x **2**. Fair enough, however, we also need to ensure that the dimension coming from each of them should be the same in order to merge at the end. Now, if we remember when we design the ***ConvModule,*** we purposely set the value of the ***padding*** argument to `*same*`. But in this case, we need to set it to `*valid*`.

```
class DownsampleModule(tf.keras.layers.Layer):
    def __init__(self, kernel_size):
        super(DownsampleModule, self).__init__()
        # conv layer
        self.conv3 = ConvModule(
                  kernel_size,
                  kernel_size=(3,3),
                  strides=(2,2),
                  padding="valid"
                )

        # pooling layer
        self.pool  = tf.keras.layers.MaxPooling2D(
                  pool_size=(3, 3),
                  strides=(2,2)
                )
        self.cat   = tf.keras.layers.Concatenate()


    def call(self, input_tensor, training=False):
        # forward pass
        conv_x = self.conv3(input_tensor, training=training)
        pool_x = self.pool(input_tensor)

        # merged
        return self.cat([conv_x, pool_x])
```

Model Class: Layers Encompassing

In general, we use the ***Layer*** class to define the inner computation blocks and will use the ***Model*** class to define the outer model, practically the object that we will train. In our case, in an ***Inception*** model, we define three computational blocks: *Conv Module, Inception Module,* and *Downsample Module*. These are created by subclassing the ***Layer*** class. And so next, we will use the ***Model*** class to encompass these computational blocks in order to create the entire ***Inception*** network. Typically the ***Model*** class has the same API as ***Layer*** but with some extra functionality.

Same as the ***Layer*** class, we will *initialize the computational block* inside the *init* method of the ***Model*** class as follows:

```
# the first conv module
self.conv_block = ConvModule(96, (3,3), (1,1))

# 2 inception module and 1 downsample module
self.inception_block1  = InceptionModule(32, 32)
self.inception_block2  = InceptionModule(32, 48)
self.downsample_block1 = DownsampleModule(80)

# 4 inception module and 1 downsample module
self.inception_block3  = InceptionModule(112, 48)
self.inception_block4  = InceptionModule(96, 64)
self.inception_block5  = InceptionModule(80, 80)
self.inception_block6  = InceptionModule(48, 96)
self.downsample_block2 = DownsampleModule(96)

# 2 inception module
self.inception_block7 = InceptionModule(176, 160)
self.inception_block8 = InceptionModule(176, 160)

# average pooling
self.avg_pool = tf.keras.layers.AveragePooling2D((7,7))
```

The amount of **filter number** for each computational block is set according to the design of the model (also visualized down below in the diagram). After initialing all the blocks, we will connect them according to the design (diagram). Here is the full **Inception** network using the **Model** subclass:

503 | 1

```
class MiniInception(tf.keras.Model):
    def __init__(self, num_classes=10):
        super(MiniInception, self).__init__()

        # the first conv module
        self.conv_block = ConvModule(96, (3,3), (1,1))

        # 2 inception module and 1 downsample module
        self.inception_block1  = InceptionModule(32, 32)
        self.inception_block2  = InceptionModule(32, 48)
        self.downsample_block1 = DownsampleModule(80)

        # 4 inception module and 1 downsample module
        self.inception_block3  = InceptionModule(112, 48)
        self.inception_block4  = InceptionModule(96, 64)
        self.inception_block5  = InceptionModule(80, 80)
        self.inception_block6  = InceptionModule(48, 96)
        self.downsample_block2 = DownsampleModule(96)

        # 2 inception module
        self.inception_block7 = InceptionModule(176, 160)
        self.inception_block8 = InceptionModule(176, 160)

        # average pooling
        self.avg_pool = tf.keras.layers.AveragePooling2D((7,7))

        # model tail
        self.flat      = tf.keras.layers.Flatten()
        self.classfier = tf.keras.layers.Dense(
```

Functions (Deep Learning) in NumPy, TensorFlow, and...

Somnath ... in JavaScript in Plai...

Coding Won't Exist In 5 Years. This Is Why

Help Status Writers Blog Careers Privacy Terms About
Text to speech

```
                                num_classes, activation='softmax'
                            )

    def call(self, input_tensor, training=False, **kwargs):
        # forward pass
        x = self.conv_block(input_tensor)
        x = self.inception_block1(x)
        x = self.inception_block2(x)
        x = self.downsample_block1(x)

        x = self.inception_block3(x)
        x = self.inception_block4(x)
        x = self.inception_block5(x)
        x = self.inception_block6(x)
        x = self.downsample_block2(x)

        x = self.inception_block7(x)
        x = self.inception_block8(x)
        x = self.avg_pool(x)

        x = self.flat(x)
        return self.classfier(x)


    def build_graph(self, raw_shape):
        x = tf.keras.layers.Input(shape=raw_shape)
        return Model(inputs=[x], outputs=self.call(x))
```

As you may notice, apart from the *__init__* and **call** method additionally we define a custom method ***build_graph***. We're using this as a helper function to plot the model summary information conveniently. Please, check out this discussion for more details. Anyway, let's check out the model's summary.

```
raw_input = (32, 32, 3)

# init model object
cm = MiniInception()

# The first call to the `cm` will create the weights
y = cm(tf.ones(shape=(0,*raw_input)))

# print summary
cm.build_graph(raw_input).summary()

# _________________________________________________________________
__

Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         [(None, 32, 32, 3)]       0
_________________________________________________________________
conv_module_329 (ConvModule) (None, 32, 32, 96)        3072
_________________________________________________________________
inception_module_136 (Incept (None, 32, 32, 64)        31040
_________________________________________________________________
inception_module_137 (Incept (None, 32, 32, 80)        30096
_________________________________________________________________
downsample_module_34 (Downsa (None, 15, 15, 160)       58000
_________________________________________________________________
inception_module_138 (Incept (None, 15, 15, 160)       87840
_________________________________________________________________
inception_module_139 (Incept (None, 15, 15, 160)       108320
_________________________________________________________________
inception_module_140 (Incept (None, 15, 15, 160)       128800
_________________________________________________________________
inception_module_141 (Incept (None, 15, 15, 144)       146640
_________________________________________________________________
downsample_module_35 (Downsa (None, 7, 7, 240)         124896
_________________________________________________________________
inception_module_142 (Incept (None, 7, 7, 336)         389520
_________________________________________________________________
inception_module_143 (Incept (None, 7, 7, 336)         544656
_________________________________________________________________
average_pooling2d_17 (Averag (None, 1, 1, 336)         0
_________________________________________________________________
flatten_13 (Flatten)         (None, 336)               0
_________________________________________________________________
dense_17 (Dense)             (None, 10)                3370
=================================================================
Total params: 1,656,250
Trainable params: 1,652,826
Non-trainable params: 3,424
```

Now, it is complete to build the entire **Inception** model via model subclassing.

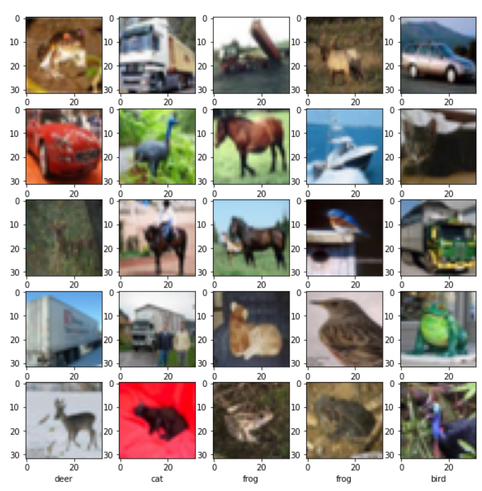
End-to-End Training with Custom Training Loop from Scratch

Now we have built a complex network, it's time to make it busy to learn something. We can now easily train the model simply just by using the ***compile*** and ***fit***. But here we will look at a custom training loop from scratch. This functionality is newly introduced in ***TensorFlow 2***. Please note, that this functionality is a little bit complex comparatively and more fit for the deep learning researcher.

Data Set

For the demonstration purpose, we will be using the CIFAR-10 data set. Let's prepare it first.

```
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
print(x_train.shape, y_train.shape) # (50000, 32, 32, 3) (50000, 1)
print(x_test.shape, y_test.shape)   # (10000, 32, 32, 3) (10000, 1)

# train set / data
x_train = x_train.astype('float32') / 255

# validation set / data
x_test = x_test.astype('float32') / 255

# target / class name
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']


plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.grid(False)
    plt.imshow(x_train[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[y_train[i][0]])
```



samples from cifar-10

Here we will convert the *class vector (**y_train, y_test**)* to the *multi-class matrix*. And also we will use ***tf.data*** API for better and more efficient input pipelines.

```
# train set / target
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
# validation set / target
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)


# Prepare the training dataset.
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
train_dataset = train_dataset.shuffle(buffer_size=1024).batch(batch_size)


# Prepare the validation dataset.
val_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
val_dataset = val_dataset.batch(batch_size)
```

Let's quickly check the data shape after *label conversion* and *input slicing*:

```
for i, (x, y) in enumerate(train_dataset):
    print(x.shape, y.shape)

    if i == 2:
        break


for i, (x, y) in enumerate(val_dataset):
    print(x.shape, y.shape)

    if i == 2:
       break
# output


(64, 32, 32, 3) (64, 10)
(64, 32, 32, 3) (64, 10)
```

```
(64, 32, 32, 3) (64, 10)


(64, 32, 32, 3) (64, 10)
(64, 32, 32, 3) (64, 10)
(64, 32, 32, 3) (64, 10)
```

so far so good. We have an input shape of **32** ***x*** **32** ***x*** **3** and a total of **10** classes to classify. However, it's not ideal to make a *test set* as a *validation set* but for demonstration purposes, we are not considering the ***train_test_split*** approach. Now, let's see what the custom training pipelines consist of in Tensorflow 2.

Training Mechanism

In *TF.Keras,* we have convenient training and evaluating loops, ***fit,*** and ***evaluate***. But we also can have leveraged the low-level control over the training and evaluation process. In that case, we need to write our own training and evaluation loops from scratch. Here are the recipes:

1. We open a **for loop** that will iterate over the number of epochs.
2. For each epoch, we open another **for loop** that will iterate over the datasets, in batches (***x***, ***y***).
3. For each batch, we open ***GradientTape()*** scope.
4. Inside this scope, we call the model, the **forward pass**, and compute the **loss**.
5. Outside this scope, we retrieve the **gradients of the weights** of the model with regard to the **loss**.
6. Next, we use the optimizer to **update the weights** of the model based on the gradients.

TensorFlow provides the ***tf.GradientTape()*** API for automatic differentiation, that is, computing the gradient of computation with respect to some inputs.

Below is a short demonstration of its operation process. Here we have some input (***x***) and trainable param (***w, b***). Inside the ***tf.GradientTape()*** scope, output (***y***, which basically would be the model output), and loss are measured. And outside the scope, we retrieve the gradients of the weight parameter with respect to the loss.

```
# x:input, w,b: trainable param - x*w + b
w = tf.Variable(tf.random.normal((3, 2)), name='w')
b = tf.Variable(tf.zeros(2, dtype=tf.float32), name='b')
x = [[1., 2., 3.]]


# Open a GradientTape to record the operations run
# during the forward pass, which enables auto-differentiation.
with tf.GradientTape(persistent=True) as tape:
    y = x @ w + b # output from the forward pass (for the actual model)

    # Compute the loss value for this minibatch.
    loss = tf.reduce_mean(y**2)


# Calculate gradients with respect to every trainable variable
grad = tape.gradient(loss, [w, b])
grad

# output
[
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
 array([[ -5.2607636,   1.4286567],
        [-10.521527 ,   2.8573134],
        [-15.782291 ,   4.28597  ]], dtype=float32)>,

<tf.Tensor: shape=(2,), dtype=float32,
numpy=array([-5.2607636,  1.4286567], dtype=float32)>
]
```

Now, let's implement the custom training recipes accordingly.

```
for epoch in range(epochs): # <----- start for loop, step 1

  # <-------- start for loop, step 2
  # Iterate over the batches of the dataset.
  for step, (x_batch_train, y_batch_train) in enumerate(train_dataset):

    # <-------- start gradient tape scope, step 3
    # Open a GradientTape to record the operations run
    # during the forward pass, which enables auto-differentiation.
    with tf.GradientTape() as tape:

       # Run the forward pass of the layer.
       # The operations that the layer applies
       # to its inputs are going to be recorded
       # on the GradientTape.
       logits = model(x_batch_train, training=True) <- step 4

       # Compute the loss value for this minibatch.
       loss_value = loss_fn(y_batch_train, logits)  <- step 4


    # compute the gradient of weights w.r.t. loss  <-------- step 5
    # Use the gradient tape to automatically retrieve
    # the gradients of the trainable variables with respect to the loss.
    grads = tape.gradient(loss_value, model.trainable_weights)

    # update the weight based on gradient  <---------- step 6
    # Run one step of gradient descent by updating
    # the value of the variables to minimize the loss.
    optimizer.apply_gradients(zip(grads, model.trainable_weights))
```

Great. However, we are still not talking about how we gonna add metrics to monitor this custom training loop. Obviously, we can use built-in metrics or even custom metrics in the training loop also. To add metrics in the training loop is fairly simple, here is the flow:

1. Call ***metric.update_state()*** after each batch
2. Call ***metric.result()*** when we need to display the current value of the metric
3. Call ***metric.reset_states()*** when we need to clear the state of the metric, typically we do this at the very end of an epoch.

Here is another thing to consider. The default runtime in TensorFlow 2.0 is eager execution. The above training loops are executing eagerly. But if we want graph compilation we can compile any function into a static graph with ***@tf.function*** decorator. This also speeds up the training step much faster. Here is the setup for the training and evaluation function with ***@tf.function*** decorator.

```
@tf.function
def train_step(x, y):
   '''
   input: x, y <- typically batches
   return: loss value
   '''

    # start the scope of gradient
    with tf.GradientTape() as tape:
        logits = model(x, training=True) # forward pass
        train_loss_value = loss_fn(y, logits) # compute loss

    # compute gradient
    grads = tape.gradient(train_loss_value, model.trainable_weights)

    # update weights
    optimizer.apply_gradients(zip(grads, model.trainable_weights))

    # update metrics
    train_acc_metric.update_state(y, logits)

    return train_loss_value


@tf.function
def test_step(x, y):
   '''
   input: x, y <- typically batches
   return: loss value
   '''
    # forward pass, no backprop, inference mode
    val_logits = model(x, training=False)

    # Compute the loss value
    val_loss_value = loss_fn(y, val_logits)

    # Update val metrics
    val_acc_metric.update_state(y, val_logits)

    return val_loss_value
```

Here we're seeing the usage of ***metrics.update_state()***. These functions return to the training loop where we will set up displaying the log message, ***metric.result()***, and also reset the metrics, ***metric.reset_states()***.

Here is the last thing we like to set up, the **TensorBoard**. There are some great functionalities in it to utilize such as: displaying per batches samples + confusion matrix, hyper-parameter tuning, embedding projector, model

graph, etc. For now, we will only focus on logging the training metrics on it. Simple enough but we will integrate it into the custom training loop. So, we can't use ***tf.keras.callbacks.TensorBoard*** but need to use the *TensorFlow Summary API*. The ***tf.summary*** module provides API for writing summary data on TensorBoard. We want to write the logging state after each batch operation to get more details. Otherwise, we may prefer at the end of each epoch. Let's create a directory where the message of the ***event*** will be saved. In the working directory, create the ***log/train*** and ***log/test***. Below is the full training pipelines. We recommend reading the code thoroughly at first in order to get the overall training flow.

```
# Instantiate an optimizer to train the model.
optimizer = tf.keras.optimizers.Adam()

# Instantiate a loss function
loss_fn = tf.keras.losses.CategoricalCrossentropy()

# Prepare the metrics.
train_acc_metric = tf.keras.metrics.CategoricalAccuracy()
val_acc_metric   = tf.keras.metrics.CategoricalAccuracy()

# tensorboard writer
train_writer = tf.summary.create_file_writer('logs/train/')
test_writer  = tf.summary.create_file_writer('logs/test/')


@tf.function
def train_step(step, x, y):
   '''
   input: x, y <- typically batches
   input: step <- batch step
   return: loss value
   '''

    # start the scope of gradient
    with tf.GradientTape() as tape:
        logits = model(x, training=True) # forward pass
        train_loss_value = loss_fn(y, logits) # compute loss

    # compute gradient
    grads = tape.gradient(train_loss_value, model.trainable_weights)

    # update weights
    optimizer.apply_gradients(zip(grads, model.trainable_weights))

    # update metrics
    train_acc_metric.update_state(y, logits)

    # write training loss and accuracy to the tensorboard
    with train_writer.as_default():
        tf.summary.scalar('loss', train_loss_value, step=step)
        tf.summary.scalar(
            'accuracy', train_acc_metric.result(), step=step
        )
    return train_loss_value


@tf.function
def test_step(step, x, y):
   '''
   input: x, y <- typically batches
   input: step <- batch step
   return: loss value
   '''
    # forward pass, no backprop, inference mode
    val_logits = model(x, training=False)

    # Compute the loss value
    val_loss_value = loss_fn(y, val_logits)

    # Update val metrics
    val_acc_metric.update_state(y, val_logits)

    # write test loss and accuracy to the tensorboard
    with test_writer.as_default():
        tf.summary.scalar('val loss', val_loss_value, step=step)
        tf.summary.scalar(
            'val accuracy', val_acc_metric.result(), step=step
        )
    return val_loss_value


# custom training loop
for epoch in range(epochs):
    t = time.time()
    # batch training

    # Iterate over the batches of the train dataset.
    for train_batch_step, (x_batch_train, \
                           y_batch_train) in enumerate(train_dataset):
        train_batch_step = tf.convert_to_tensor(
                                train_batch_step, dtype=tf.int64
                           )
        train_loss_value = train_step(
                                train_batch_step,
                                x_batch_train, y_batch_train
                           )

    # evaluation on validation set
    # Run a validation loop at the end of each epoch.
    for test_batch_step, (x_batch_val, \
                          y_batch_val) in enumerate(val_dataset):
        test_batch_step = tf.convert_to_tensor(
                                test_batch_step, dtype=tf.int64
                           )
        val_loss_value = test_step(
                                test_batch_step, x_batch_val, y_batch_val
                           )


    template = '
        ETA: {} - epoch: {} loss: {}  acc: {} val loss: {} val acc: {}\n
    '
    print(template.format(
        round((time.time() - t)/60, 2), epoch + 1,
        train_loss_value, float(train_acc_metric.result()),
        val_loss_value, float(val_acc_metric.result())
    ))

    # Reset metrics at the end of each epoch
    train_acc_metric.reset_states()
    val_acc_metric.reset_states()
```

Voila! We run the code in our local system, with RTX 2070. By enabling the ***mixed-precision*** we're able to increase the *batch size* up to **256**. Here is the log output:

```
ETA: 0.78 - epoch: 1 loss: 0.7587890625  acc: 0.5794399976730347 val
loss: 3.173828125 val acc: 0.10159999877214432

ETA: 0.29 - epoch: 2 loss: 0.63232421875  acc: 0.7421200275421143 val
loss: 1.0126953125 val acc: 0.5756999850273132

ETA: 0.32 - epoch: 3 loss: 0.453369140625  acc: 0.8073400259017944
val loss: 0.7734375 val acc: 0.7243000268936157

ETA: 0.33 - epoch: 4 loss: 0.474365234375  acc: 0.8501200079917908
val loss: 0.64111328125 val acc: 0.7628999948501587

..
..


ETA: 0.35 - epoch: 17 loss: 0.0443115234375  acc: 0.9857199788093567
val loss: 1.8603515625 val acc: 0.7465000152587891

ETA: 0.68 - epoch: 18 loss: 0.01328277587890625  acc:
0.9839400053024292 val loss: 0.65380859375 val acc:
0.7875999808311462

ETA: 0.53 - epoch: 19 loss: 0.035552978515625  acc:
0.9851599931716919 val loss: 1.0849609375 val acc: 0.7432000041007996

ETA: 0.4 - epoch: 20 loss: 0.04217529296875  acc: 0.9877399802207947
val loss: 3.078125 val acc: 0.7224000096321106
```

Overfitting! But that's ok for now. For that, we just need some care to consider such as **Image Augmentation**, **Learning Rate Schedule**, etc. In the working directory, run the following command to the live tensorboard. In the below command, ***logs*** are the folder name that we created manually to save the event logs.

```
tensorboard --logdir logs
```

Save and Load

There are various ways to save *TensorFlow* models depending on the API we're using. Model ***saving*** and ***re-loading*** in model subclassing are not the same as in *Sequential* or *Functional API*. It needs some special attention. Currently, there are two formats to store the model: *SaveModel* and HDF5. From the official doc:

> *The key difference between HDF5 and SavedModel is that HDF5 uses object configs to save the model architecture, while SavedModel saves the execution graph. Thus, SavedModels are able to save custom objects like subclassed models and custom layers without requiring the orginal code.*

So, it looks like *SavedModels* are able to save our custom subclassed models. But what if we want *HDF5* format for our custom subclassed models? According to the doc. we can do that either but we need some extra stuff. We must define the ***get_config*** method in our object. And also need to pass the object to the ***custom_object*** argument when loading the model. This argument must be a dictionary mapping: *tf.keras.models.load_model(path, custom_objects={'CustomLayer': CustomLayer}).* However, it seems like we can't use HDF5 for now as we don't use the ***get_config*** method in our customed object. However, it's actually a good practice to define this function in the

custom object. This will allow us to easily update the computation later if needed.

But for now, let's now save the model and reload it again with the *SavedModel* format.

```
model.save('net', save_format='tf')
```

After that, it will create a new folder named *net* in the working directory. It will contain ***assets***, ***saved_model.pb***, and ***variables***. The model architecture and training configuration, including the optimizer, losses, and metrics are stored in *saved_model.pb*. The weights are saved in the *variables* directory.

When saving the model and its layers, the *SavedModel* format stores the class name, **call** function, losses, and weights (and the config, if implemented). The **call** function defines the computation graph of the model/layer. In the absence of the model/layer config, the **call** function is used to create a model that exists like the original model which can be *trained*, *evaluated*, and used for *inference*. Later to re-load the saved model, we will do:

```
new_model = tf.keras.models.load_model("net", compile=False)
```

Set ***compile=False*** is optional, I do this to avoid warning logs. Also as we are doing custom loop training, we don't need any compilation.

So far, we have talked about saving the entire model (*computation graph* and *parameters*). But what, if we want to save the **trained weight** only and reload the weight when need it. Yeap, we can do that too. Simply just,

```
model.save_weights('net.h5')
```

It will save the weight of our model. Now, when it comes to re-load it again, here is one thing to keep in mind. We need to call the ***build*** method before we try to load the weight. It mainly initializes the layers in a subclassed model, so that the computation graph can build. For example, if we try as follows:

```
new_model = MiniInception()
new_model.load_weights('net.h5')

---------------------------------------
ValueError: Unable to load weights saved in HDF5 format
into a subclassed Model which has not created its variables yet.
Call the Model first, then load the weights.
```

To solve that we can do as follows:

```
new_model = MiniInception()
new_model.build((None, *x_train.shape[1:])) # or .build((x_train.shape))
new_model.load_weights('net.h5')
```

It will load successfully. Here is an awesome article regarding saving and serializing models in TF.Keras by François Chollet, must-read.

Evaluation and Prediction

Though not necessary, let's end up by measuring the model performance. **CIFAR-10** class label maps are as follows: **0**:airplane, **1**:automobile, **2**:bird, **3**:cat, **4**:deer, **5**:dog, **6**:frog, **7**:horse, **8**:ship, **9**:truck. Let's find the **classification report** first.

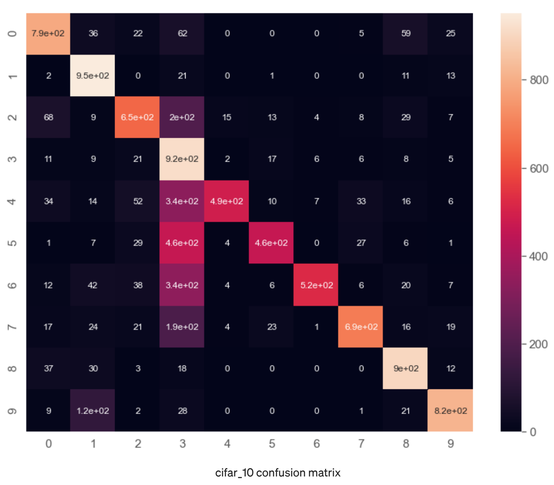
```
Y_pred = model.predict(x_test, verbose=1)
y_pred = np.argmax(Y_pred, axis=1)

target_names = [
      'airplane', 'automobile', 'bird', 'cat', 'deer',
      'dog', 'frog', 'horse', 'ship', 'truck'
]


classification_report(
      np.argmax(y_test, axis=1),
      y_pred,target_names=target_names
)
#-------------------------------------------------------------------

          precision    recall  f1-score   support

    airplane       0.81      0.79      0.80      1000
  automobile       0.76      0.95      0.85      1000
        bird       0.78      0.65      0.71      1000
         cat       0.36      0.92      0.51      1000
        deer       0.94      0.49      0.65      1000
         dog       0.87      0.46      0.60      1000
        frog       0.97      0.52      0.68      1000
       horse       0.89      0.69      0.78      1000
        ship       0.83      0.90      0.86      1000
       truck       0.90      0.81      0.85      1000

    accuracy                           0.72     10000
   macro avg       0.81      0.72      0.73     10000
weighted avg       0.81      0.72      0.73     10000
```

Next, multi-class **ROC AUC** score:

```
def multiclass_roc_auc_score(
      y_test, y_pred, average="macro"
):
    lb = LabelBinarizer()
    lb.fit(y_test)
    y_test = lb.transform(y_test)
    y_pred = lb.transform(y_pred)
    return roc_auc_score(y_test, y_pred, average=average)


multiclass_roc_auc_score(y_test,y_pred)
# ------------------------------------------------
#output: 0.8436111111111112
```

Confusion Matrix:

```
Y_pred = model.predict(x_test, verbose=2)
y_pred = np.argmax(Y_pred, axis=1)

cm = confusion_matrix(np.argmax(y_test, axis=1), y_pred)
cm = pd.DataFrame(cm, range(10),range(10))
plt.figure(figsize = (10,10))

sns.heatmap(cm, annot=True, annot_kws={"size": 12}) # font size
plt.show()
```



cifar_10 confusion matrix

Prediction / Inference on new sample:

```
target_names = [
      'airplane', 'automobile', 'bird', 'cat', 'deer',
      'dog', 'frog', 'horse', 'ship', 'truck'
]

# Give the link of the image here to test
test_image1 = image.load_img(image_file, target_size=(32,32))
test_image  = image.img_to_array(test_image1)
test_image  = np.expand_dims(test_image, axis =0)
prediction  = model.predict(test_image)[0]
label_index = np.argmax(prediction)
target_names[label_index]
```

EndNote

This ends here. Thank you so much for reading the article, hope you guys enjoy it. The article is a bit long, so here is a quick summary.

We first compare **TF.Keras** modeling APIs. Next, we use the ***Model Sub-Classing*** API to build a small **Inception** network step by step. Then we look at the process of newly introduced custom loop training in TensorFlow 2 with **GradientTape**. We've also trained the subclassed **Inception** model end to end. And lastly, we discuss custom model saving and reloading followed by measuring the performance of these trained models.

Interesting Reading

1. AutoGrad in PyTorch and GradientTape in TensorFlow 2
2. Multi-Input Multi-Output Modeling in TensorFlow 2 [Keras]
3. Selecting Loss and Metrics
4. Reload Best Checkpoint in Keras
5. Keras Dense Layer vs PyTorch Linear Layer
6. Keras Modeling API vs Sequential API
7. Keras Model.Predict Slower than NumPy!
8. Implement GradCAM in Keras
9. Keras Attention Layer for LSTM / GRU
10. TensorFlow Data API vs Numpy Array
11. Neural Network & Binary Classification Guidance
12. Quantization Aware Training with tf.GradientTape
13. Purpose of self.add_loss Function in Keras

Deep Learning TensorFlow Machine Learning Keras Research

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

By signing up, you will create a Medium account if you don't already have one. Review our Privacy Policy for more information about our privacy practices.

Get this newsletter