



# Chapter 12

## Custom Models and Training with TensorFlow




GitHub Page: [LINK](#)

Google Drive: [LINK](#)

- This chapter will introduce TensorFlow's lower level API
- This will help when require more control to write custom hyperparameters

## A Quick Tour of TensorFlow

- Powered by a lot of google services
- Open source since 2015 and the most popular deep learning library
- Summary of what TensorFlow offers:
  - Similar to NumPy but with GPU support
  - Supports distributed computing
  - Includes a kind of just-in-time (JIT). Allows compiler to optimize computation for speed and memory usage
  - Can be used other formats (using java on android)
  - implements autodiff and with lots of optimizers like RMSProp and Nadam
- offers lots of features on top of core TensorFlow, tf.keras is the most important, loading preprocessing operators like tf.data, [tf.io](#). And tf.image for image processing
- Lowest level operation is done with C++ code
- kernels dedicate specific device types (CPU, GPU, TPU)

 TensorFlow architecture on page 377

- APIs for other languages is also available
- Runs on windows, linux, macos, ios and android (using tensorflow lite)
- Also runs on browsers with TensorFlow.js
- TensorBoard for visualization

## Using TensorFlow like NumPy

- TensorFlow revolves around tensors, which flow from operation to operation

## Tensors and Operations

- can create a tensor with `tf.constant()`
- For operations
  - Basic math operations
    - `tf.add()`, `tf.multiply()`, `tf.square()`, `tf.exp()`, `tf.sqrt()`
  - Different operation names from numpy
    - `tf.reduce_mean()`, `tf.reduce_sum()`, `tf.reduce_max()`, `tf.math.log()`

```
import tensorflow as tf

tf.constant([[1., 2., 3.], [4., 5., 6.]])

tf.constant(42)

"""tf.tensor has dtype and shape"""

t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
print(t.shape)
print(t.dtype)

"""Indexing works like numpy"""

t[:, 1:]

"""all sorts of tensor operations are available"""

t + 10
```

```
tf.square(t)

t @ tf.transpose(t)
```

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/5bd4d95d-ca80-4168-bc51-a5227dc71410/tensors\\_and\\_operations.py](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/5bd4d95d-ca80-4168-bc51-a5227dc71410/tensors_and_operations.py)

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/dd0d2e04-bcdc-4890-beba-09be452d61d9/tensors\\_and\\_operations.ipynb](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/dd0d2e04-bcdc-4890-beba-09be452d61d9/tensors_and_operations.ipynb)

## Tensors and Numpy

- Can apply TensorFlow operation to Numpy and vice versa

```
import numpy as np
import tensorflow as tf
a = tf.constant([2., 4., 5.])
t = tf.constant([[1., 2., 3.], [4., 5., 6.]])

a.numpy()

tf.square(a)

np.square(t)
```

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/3766816c-83d6-48ed-b2a3-1bd8a66a06d3/tensors\\_and\\_numpy.py](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/3766816c-83d6-48ed-b2a3-1bd8a66a06d3/tensors_and_numpy.py)

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/829e56d2-1668-47fe-81ef-6b5e0f3cae36/Tensors\\_and\\_NumPy.ipynb](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/829e56d2-1668-47fe-81ef-6b5e0f3cae36/Tensors_and_NumPy.ipynb)

## Type Conversion

- type conversions can significantly hurt performance, therefore there's no automatic type conversion in TensorFlow

```
import tensorflow as tf

"""They have to be the same type"""

tf.constant(2.0) + tf.constant(40)

"""This Works: """
t2 = tf.constant(40., dtype=tf.float64)
tf.constant(2.0) + tf.cast(t2, tf.float32)
```

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/27d8b00b-51ee-49e2-91ad-63eea7369e71/type\\_conversion.py](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/27d8b00b-51ee-49e2-91ad-63eea7369e71/type_conversion.py)

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/5c0d2379-4816-45da-912c-fae51d3d55e5/type\\_conversion.ipynb](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/5c0d2379-4816-45da-912c-fae51d3d55e5/type_conversion.ipynb)

## Variables

- `tf.Tensor` are immutable
- If values needs to be tuned then `tf.variable` is what is required

```
import tensorflow as tf

v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
print(v)

v.assign(2*v)
v[0, 1].assign(42)
v[:, 2].assign([0., 1.])
v.scatter_nd_update(indices=[[0,0], [1,2]], updates=[100., 200.])
print(v)
```

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/33ca5837-6328-4057-8054-3ac765489448/variables.py
```

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/51d8d410-41d5-47fc-bf77-7128e39606fd/variables.ipynb
```

## Other data Structures

- TensorFlow also supports several other data structures
- Sparse tensors (`tf.SparseTensor`)
  - represents tensors containing mostly zeros. `tf.sparse` package contains operations for sparse tensors
- Tensor arrays (`tf.TensorArray`)
  - List of tensors
  - Constant at default but could be modified to variable
  - all must have same shape and data type
- Tagged tensors (`tf.RaggedTensor`)
  - represent static list of tensors
  - every tensor has the same shape and data type
  - contains operations for ragged tensors
- String tensors (`tf.string`)
  - Represents bytes of string not Unicode strings
- Sets
  - Are represented as regular sparse tensors
  - each set is represented by a vector in the tensor's last axis
- Queues
  - Store tensors across multiple steps
  - TF offers many types of queues:

- Simple First In
- First out (FIFO) queues
- queues that can prioritize some items (PriorityQueue)
- shuffle their items (RandomShuffleQueue)
- batch items of different shapes by padding (PaddingFIFOQueue)
- All in tf.queue package

## Customizing Models and Training Algorithms

### Custom loss Function

- Suppose you want to train a regression model and the training set is a bit noisy
- You could start by trying to clean up the data
- It's still not enough to remove the noise
- MSE might penalize large errors too much and cause imprecise models
- MAE training takes too long
- Huber loss is a good option and it's in the Keras API

```
def huber_fn(y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) < 1
    squared_loss = tf.square(error) / 2
    linear_loss = tf.abs(error) - 0.5
    return tf.where(is_small_error, squared_loss, linear_loss)

model.compile(loss=huber_fn, optimizer='nadam')
model.fit(train_data, train_label, epochs=10)
```

### Saving and Loading Models That Contains Custom Components

- Saving a model containing custom loss function works fine

- When loading it, it's required to provide a dictionary that maps the function name to the actual function

```
model = keras.model.load_model("my_model_with_a_custom_loss.h5",
    custom_objects={"huber_fn":huber_fn})
```

- Current implementation has any error between -1 and 1
- If requiring a different threshold, one solution is to create a function that creates a configured loss function

```
def create(threshold=1.0):
    def huber_fn(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < threshold
        squared_loss = tf.square(error) / 2
        linear_loss = threshold * tf.abs(error) - threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    return huber_fn

model.compile(loss=create_humber(2.0, optimizer='nadam')
```

- Threshold will not be saved when saving the model

```
model = keras.models.load_model("my_model_with_a_custom_loss_threshold_2.h5",
    custom_objects={"huber_fn", create_huber(2.0)})
```

- Can solve this by creating a subclass of the `keras.losses.Loss` class then implementing its `get_config()` method

```
class HuberLoss(keras.losses.Loss):
    def __init__(self, threshold=1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)
    def call(self, y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < threshold
        squared_loss = tf.square(error) / 2
        linear_loss = threshold * tf.abs(error) - threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold", self.threshold}
```

- constructor accepts `**kwargs` and passes them to the parent constructor

- `call()` method takes the labels and prediction and computes all the instance losses, and returns them
- The `get_config()` method returns a dictionary mapping each hyperparameter name to its value
- `get_config()` returns a dictionary mapping of each hyperparameter. It first calls parent class's `get_config` method then adds the new hyperparameter to this dictionary

```
model.compile(loss=HuberLoss(2.0), optimizer="nadam")
```

- Loading the model

```
model = keras.models.load_model("my_model_with_a_custom_loss_class.h5",
    custom_objects={"HuberLoss": HuberLoss})
```

## Custom Activation Functions, Initializers, Regularizers, and Constraints

- Most Keras functionality can be customized in some way

```
def my_softplus(z):
    return tf.math.log(tf.exp(z) + 1.0)
def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)
def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))
def my_positive_weights(weights):
    return tf.where(weights < 0., tf.zeros_like(weights), weights)

layer = keras.layers.Dense(30, activation=my_softplus,
    kernel_initializer=my_glorot_initializer,
    kernel_regularizer=my_l1_regularizer,
    kernel_constraint=my_positive_weights)
```

- If function has hyperparameters that needs to be saved then it's better to subclass the appropriate class

```
class MyL1Regularizer(keras.regularizers.Regularizer):
    def __init__(self, factor):
        self.factor = factor
```



```
def __call__(self, weights):
    return tf.reduce_sum(tf.abs(self.factor * weights))
def get_config(self):
    return {"factor" : self.factor}
```

- must implement call() method for losses, layers, activation function and models
- `__call__` methods for regularizers, initializers, and constraints

## Custom Metrics

- Losses and metrics are conceptually not the same thing
  - losses are used by gradient descent to train model and they must be differentiable
  - metrics are used to evaluate a model, they have to be easily interpretable and non-differentiable
- Defining custom metrics is exactly the same as defining custom loss function

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_hunber(2.0)])
```

- Using the code for metrics above would cause a issue since the it takes the average of accuracy from multiple batches
- A way to solve this is by using `keras.metrics.Precision` class

```
precision = keras.metrics.Precision()
```

- This solves the issue that was encountered with `create_hunber`
- If it's required to create a custom class that is similar to `precision`:

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/20d87d1b-4c79-4c64-a220-8053f4d84784/custom\\_metrics.ipynb](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/20d87d1b-4c79-4c64-a220-8053f4d84784/custom_metrics.ipynb)

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/a0c96597-b907-4df9-b0e3-12df784bacfb/custom\\_metrics.py](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/a0c96597-b907-4df9-b0e3-12df784bacfb/custom_metrics.py)

```
class HuberMetric(keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs)
        self.threshold = threshold
        self.huber_fn = create_huber(threshold)
        self.total = self.add_weight("total", initializer="zeros")
        self.count = self.add_weight("count", initializer="zeros")
    def update_state(self, y_true, y_pred, sample_weight=None):
        metric = self.huber_fn(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(metric))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))
    def result(self):
        return self.total / self.count
    def get_config = super().get_config()
    return {"base_config": super().get_config(), "threshold": self.threshold}
```

- Constructor uses `add_weight()` method create variables needed to keep track of metric's state
- The sum of all losses are total and the numbers of instances are count
- `update_state()` is called when it's used as an instance of the class as a function
- `Result()` computes and then return a final result
- If the class used as an instance then `update_state()` is called and then `result()` is called
- `get_config()` makes sure that threshold gets saved
- `reset_states()` resets all variables to 0

## Custom Layers

- You may want to build an architecture that contains an exotic layer which TensorFlow doesn't provide
- Or maybe you want to build a repetitive architecture
- Some layers have no weights
  - `keras.layers.Flatten()`

- `keras.layers.ReLU`
- The simplest layer without weights is `keras.layers.Lambda`

```
exponential_layer = keras.layers.Lambda(lambda x: tf.exp(x))
```

- This custom layer can be used in any API
  - Sequential API
  - Functional API
  - Subclassing API
- To create a layer with weights require a subclass of `keras.layers.Layer`
- The code below is the simplified version of Dense layer

```
class Custom_Dense(keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = keras.activation.get(activation)
    def build(self, batch_input_shape):
        self.kernel = self.add_weight(name="kernel", shape=[batch_input_shape[-1],
            self.units],
            self.bias = self.add_weight(name="bias", shape=[self.units], initializer="zeros")
        super().build(batch_input_shape)
    def call(self, X):
        return self.activation(X @ self.kernel + self.bias)
    def compute_output_shape(self, batch_input_shape):
        return tf.TensorShape(batch_input_shape.as_list()[:-1] + [self.units])
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "units" : self.units,
            "activation": keras.activations.serialize(self.activation)}
```


- The constructor takes all the hyperparameter arguments and it passes the kwargs to its parent class
- Then saves hyperparameter and apply the activation function
- `build()` creates layer's variables by using `add_weight()`
- `call()` performs desired operation in this case it computes the matrix and it outputs a layer
- `compute_output_shape()` returns the shape of the output
- `get_config()` saves the configurations

- To create a multi-input layer, the call() method should be a tuple containing all the inputs
- compute\_output\_shape() method should be a tuple containing each input's batch shape
- call method should return a list of outputs
- If the layer needs to have different behavior during training and testing, then it's required to add training argument to call
- The layer below adds Gaussian noise during training

```
class GaussianNoise(keras.layers.Layer):
    def __init__(self, stddev, **kwargs):
        super().__init__(**kwargs)
        self.stddev = stddev
    def call(self, X, training=None):
        if training:
            noise = tf.random.normal(tf.shape(X), stddev=self.stddev)
            return X + noise
        else:
            return X
    def compute_output_shape(self, batch_input_shape):
        return batch_input_shape
```

## Custom Models

- to create a model, just use a subclass of keras.Model and implement call()

 Figure 12-3 on page 395

- Inputs first goes though a dense layer
- then though residual block composed of 2 dense layers
- Then though the same residue block 3 more times
- Then it goes through the second residual block
- the final result goes though a dense layer output
- This model does not make sense but to illustrate the model
- To create a ResidualBlock

```
class ResidualBlock(keras.layers.Layer):
    def __init__(**kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(n_neurons, activation="elu",
            kernel_initializer="he_normal") for _ in range(n_layer)]
    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        return inputs + Z
```

- This layer is special since it contains other layers
- Automatically detects that the hidden attribute contains trackable objects
- Using Subclassing API to define the model:

```
class ResidualRegressor(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.Dense(30, activation="elu",
            kernel_initializer="he_normal")
        self.block1 = ResidualBlock(2, 30)
        self.block2 = ResidualBlock(2, 30)
        self.out = keras.layers.Dense(output_dim)
    def call(self, inputs):
        Z = self.hidden1(inputs)
        for _ in range(1+3):
            Z = self.block1(Z)
            Z = self.block2(Z)
        return self.out(Z)
```

- Create layers using constructor and use them in call()
- model can be used like any other model
- This model is a subclass of keras.Model therefore it can compile(), fit(), evaluate and predict()

## Losses and Metrics Based on Model Internals

- The custom losses and metrics defined earlier are based on labels and the predictions
- There will be times when you want to define losses based on other parts of your model
- To define custom loss based on model internals:

- compute it based on any part of the model you want
- pass in the result to the `add_loss()` method
- An example of building an regression MLP model composed of a stack of five hidden layers

```
class ReconstructionRegressor(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(30, activation="selu",
            kernel_initializer="lecun_normal")]
        self.out = keras.layers.Dense(output_dim)
    def build(self, batch_input_shape):
        n_inputs = batch_input_shape[-1]
        self.reconstruct = keras.layers.Dense(n_inputs)
        super().build(batch_input_shape)
    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        reconstruction = self.reconstruct(Z)
        recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
        self.add_loss(0.05 * recon_loss)
        return self.out(Z)
```

- Constructor creates the DNN with five dense hidden layers and one output layer
- build method creates a dense layer which is used to reconstruct the inputs in the model
- Call method processes computes the reconstruction loss and adds it to the model's list of losses to `add_loss()`
- call method passes the output of the hidden layers to the output layer and return the output
- It is very rare to need to customize models and algorithms but it might be needed

## Computing Gradients Using Autodiff

- When gradient is computing infinity, it will output NaN

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/496e1bc8-1e09-4d8d-a613-d37a0b5e441b/autodiff.ipynb
```

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/9ff31b79-8028-4bed-b20b-70d8f34a876f/autodiff.py
```

## Custom Training Loops

- In some rare cases `fit()` may not be flexible enough

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/0a8d0508-178c-4566-85e2-0f6d0272b872/custom_training_loop.ipynb
```

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/748341de-2cb1-4b29-854b-1fa610c8e78e/custom_training_loop.py
```

- Two loops one for epochs and another for batches inside the epochs
- then sample a random batch from the training set
- In `tf.GradientTape()` block
  - Predict for one batch
  - compute loss
  - compute mean of error using `tf.reduce_mean()`
- compute the gradient of the loss to each trainable variable using tape
- then update the mean loss and the metrics and status bar
- At the end displays another status bar to make it look complete
- To apply weight constrain:

```
for variable in model.variable:
    if variable.constraint is not None:
        variable.assign(variable.constraint(variable))
```

## TensorFlow Functions and Graphs

- Tensorflow and python function could be easily switched around
  - create TensorFlow function by either:

```
tf_cube = tf.function(cube)
```

- Using existing function and converting it to TensorFlow function
- To create a new TensorFlow function just add the following code in front of the function:

```
@tf.function
```

- To switch from TensorFlow function to Python function:

```
tf_cube.python_function(2)
```

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/9594c123-2a95-4293-a8a7-32844bacac0e/funtions\\_and\\_graphs.ipynb](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/9594c123-2a95-4293-a8a7-32844bacac0e/funtions_and_graphs.ipynb)

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b5fdb14-c85f-4214-874c-3af2fc030120/funtions\\_and\\_graphs.py](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b5fdb14-c85f-4214-874c-3af2fc030120/funtions_and_graphs.py)

## AutoGraph and Tracing

- TensorFlow generate graphs by analyzing Python function's source code to capture control flow statement



- The step is called AutoGraph
- After analyzing the function's code AutoGraph outputs an upgraded version of that function



Diagram Example on Page 408

- TensorFlow calls the output function "upgraded"
- Instead of passing the argument, it passes a symbolic tensor (tensor without any actual value, only name and data type)
- If the function is running graph mode, it means that each TensorFlow operation will add a node in the graph to represent itself and its output tensors
- In graph mode TF operations do not perform any computations

## TF Function Rules

- Creating TF functions should be as easy as using `@tf.function` but there are some rules:
  - Any external libraries will not be a part of the graph (Including NumPy and Standard library)
  - Can call other functions but all should follow same rules
  - Can only create a TF variable upon the first call, usually preferred to create variables outside of TF function
  - Python source code need to be assessable to TensorFlow
  - for loops will only work with tensor or a dataset
    - Do not do this:

```
for i in range(x):
    print(i)
```

- Do this insteadL

```
for i in tf.range(x):
    print(i)
```

- should prefer a vectorized implementation when possible rather than loop for optimal performance

## Exercises

1. TensorFlow is a opensource numerical computation library for large datasets especially machine learning. The core of TensorFlow is similar to NumPy but with GPU support, it also supports distributed computing, it's also a project of google therefore it's cross-platform including android, windows, Mac OS, IOS, Linux. Other deep learning libraries include PyTorch, Scikit-learn, theano, Dlib, ML .NET, Open CV and more.
2. TensorFlow has lots of more features built in like Keras, and other activation functions like RMSProp, Elu and more, all of it is specifically designed for machine learning while NumPy is for specifically for numerical computation, Another reason is that NumPy and TensorFlow has different names for functions and some work in different ways
3. They Both are the same except `tf.range(10)` is in 32 bit and `tf.constant(np.arange(10))` is in 64 bit
4. Other data structure other then regular tensors are: Sparse tensor, tensor array, tagged tensor, string tensor, sets and queues
5. You should use regular custom loss when it's only for regular python operations but if you would prefer hypermeter tuning using `__init__()` and `call()`, implementing `keras.losses.Loss` would be a better option
6. You should only use a normal metrics when you are dealing with very simple models or data that's not in batches, if you are dealing with anything more complicated then that then it's better to use `keras.metrics.Metric`. Another reason to use Keras metrics function is because it could save and modify hyperparameters, if that's something you would need then this a better way to go
7. You should distinguish the internal components from the model itself and decided which one to create
8. You could create a custom loop when `fit()` does not do what you require it to do or when you simply feel confident that the custom loop

will do precisely what you want to do

9. They must be converted to TF functions and follow the TF function rules
10. Rules for TF functions:
  - Cannot include any external libraries
  - Can call other functions but have to follow the same sets of rules
  - Can only create TF variable at first call
  - Python source code must be accessible
  - For loops must use `tf.range()`
11. Dynamic Keras models are useful in debugging as it doesn't compile any custom TF components therefore you can use python debugger. You could do that by setting `dynamic=True` when creating it. By using dynamic slows down training and prevents TF to use graph feature
12. Code:

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/
42240b47-1092-4a1e-9b7d-b4296aa51183/q12_custom_layer.ipynb
```

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/
ee8c9adb-8d04-4286-af01-ab585a444946/q12_custom_layer.py
```

13. Code:

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/
831c093b-7465-4748-b386-243c48eb9bf2/q_13_custom_training_loop.py
```

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/  
54417864-7a50-44dc-930f-461e1c1a5b7c/q_13_custom_training_  
loop.ipynb
```