

Introduction to Keras and TensorFlow

What's TensorFlow?

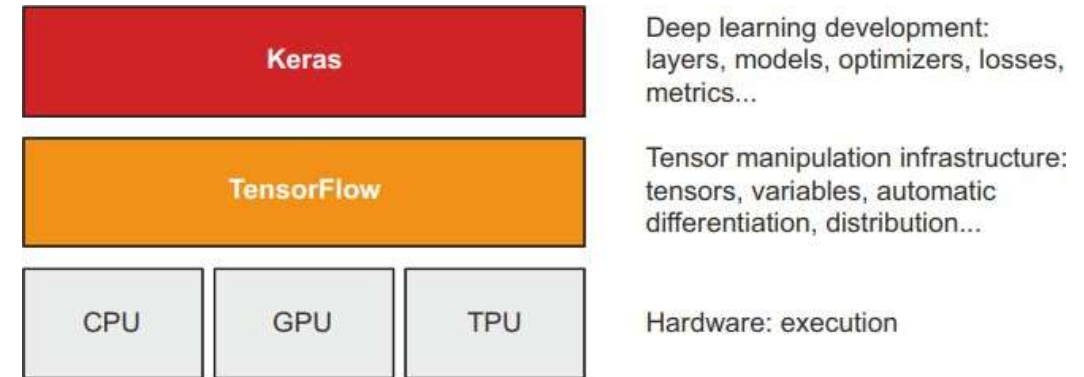
- TensorFlow is a Python-based, free, open source machine learning platform, developed primarily by Google.
- It can automatically compute the gradient of any differentiable expression, making it highly suitable for DL.
- It can run not only on CPUs, but also on GPUs and TPUs, highly parallel hardware accelerators.
- Computation defined in TensorFlow can be easily distributed across many machines.

What's TensorFlow?

- TensorFlow programs can be exported to other runtimes, such as C++, JavaScript (for browser-based applications) or TensorFlow Lite (for applications on mobile or embedded devices), etc. This makes TF applications easy to deploy in practical settings.
- TensorFlow scales fairly well
 - Oak Ridge National Lab have used it to train a 1.1 exaFLOPS extreme weather forecasting model on the 27,000 GPUs of the IBM Summit supercomputer

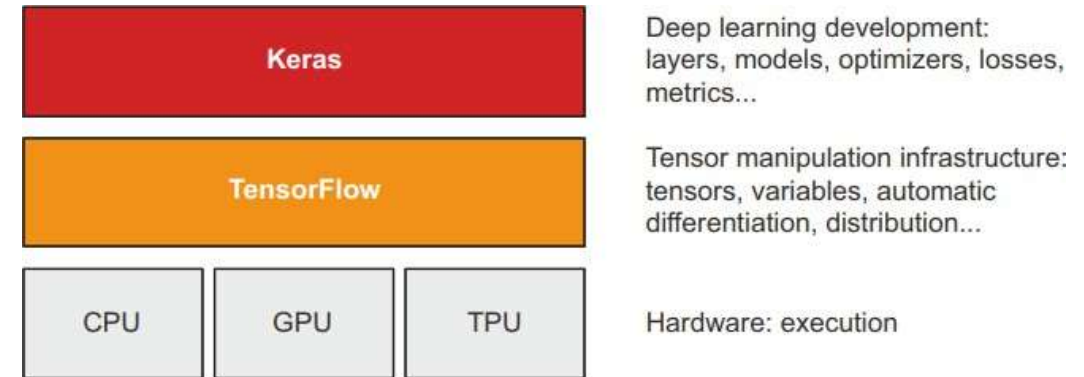
What's Keras?

- Keras is a **deep learning API** for Python, built on top of TensorFlow, to define and train any kind of deep learning model.
- Through TensorFlow, Keras can run on top of different types of **hardware**—**CPU, GPU, TPU** and can be seamlessly scaled to thousands of machines.



What's Keras?

- Keras prioritizes the developer experience- an API **for human beings**, not machines.
- Keras is easy to learn for beginners and highly productive for experts.
- Keras has a large and diverse **user base**- Google, Netflix, Uber, CERN, NASA, Yelp, Instacart, Square and used in **Waymo self-driving cars** and **YouTube recommendations**.

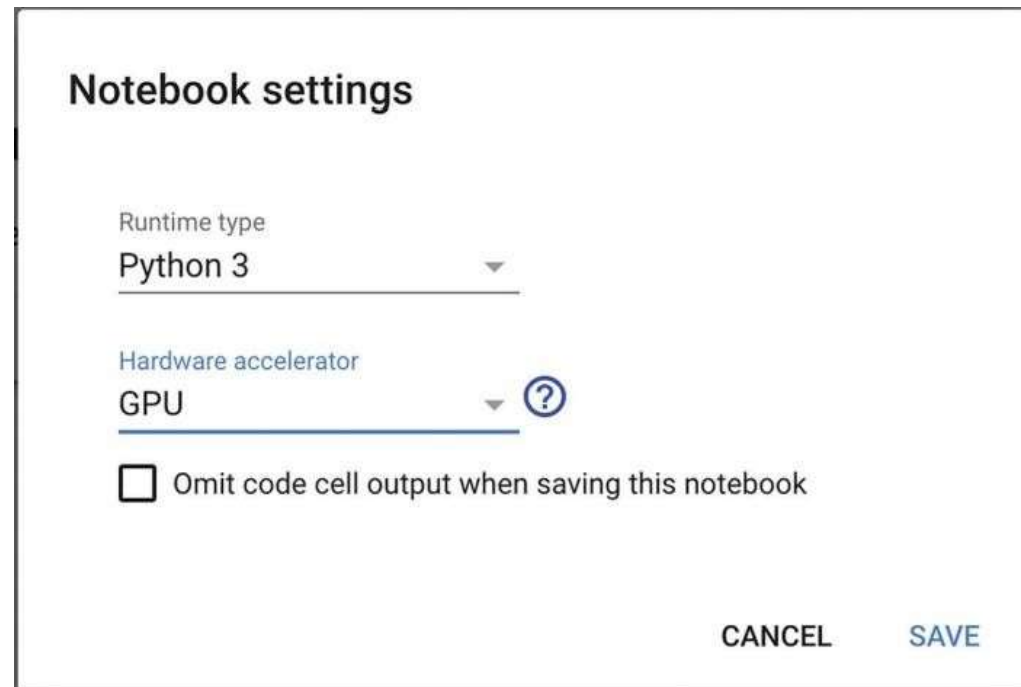


Setting up a deep learning workspace

- Highly recommended to **run deep learning code** on a modern **NVIDIA GPU** rather than computer's CPU
- To do deep learning on a GPU, you have three options:
 - Buy and install a **physical NVIDIA GPU** on your workstation.
 - Use GPU instances on **Google Cloud** or **AWS EC2**.
 - Use the free GPU runtime from **Colaboratory**.
- **Jupyter notebooks**: The preferred way to run deep learning experiments
- **Colaboratory**: a free Jupyter notebook service in the cloud, execute Keras scripts right away.
 - Gives access to a free (but limited) GPU runtime and even a TPU runtime.

Using The GPU Runtime With Colab

- select Runtime > Change Runtime Type in the menu and select GPU for the Hardware Accelerator.



The screenshot shows the 'Notebook settings' dialog box. It has two dropdown menus: 'Runtime type' set to 'Python 3' and 'Hardware accelerator' set to 'GPU'. A blue question mark icon is next to the 'GPU' selection. Below these is a checkbox labeled 'Omit code cell output when saving this notebook' which is currently unchecked. At the bottom right are 'CANCEL' and 'SAVE' buttons.

Notebook settings

Runtime type
Python 3

Hardware accelerator
GPU

☐ Omit code cell output when saving this notebook

CANCEL SAVE

First steps with TensorFlow

- First, low-level tensor manipulation. This requires **TensorFlow APIs**:
 - **Tensors**
 - **Tensor operations** such as **addition, relu, matmul**
 - **Backpropagation**, a way to compute the gradient of math expressions
- Second, high-level deep learning concepts. This requires **Keras APIs**.
 - **Layers**, which are combined into a model
 - **A loss function**, which defines the feedback signal used for learning
 - **An optimizer**, which determines how learning proceeds
 - **Metrics** to evaluate model performance, such as accuracy
 - **A training** loop that performs mini-batch stochastic gradient descent

Constant tensors

Listing 3.1 All-ones or all-zeros tensors

```
>>> import tensorflow as tf
>>> x = tf.ones(shape=(2, 1))
>>> print(x)
tf.Tensor(
[[1.]
 [1.]], shape=(2, 1), dtype=float32)
>>> x = tf.zeros(shape=(2, 1))
>>> print(x)
tf.Tensor(
[[0.]
 [0.]], shape=(2, 1), dtype=float32)
```

← Equivalent to `np.ones(shape=(2, 1))`

← Equivalent to `np.zeros(shape=(2, 1))`

Unlike NumPy arrays, TensorFlow tensors aren't assignable.

Listing 3.3 NumPy arrays are assignable

```
import numpy as np
x = np.ones(shape=(2, 2))
x[0, 0] = 0.
```

Listing 3.4 TensorFlow tensors are not assignable

```
x = tf.ones(shape=(2, 2))
x[0, 0] = 0.
```

← This will fail, as a tensor isn't assignable.

Creating a TensorFlow variable

- To create a variable, you need to provide some initial value, such as a random tensor.

Listing 3.5 Creating a TensorFlow variable

```
>>> v = tf.Variable(initial_value=tf.random.normal(shape=(3, 1)))
>>> print(v)
array([[ -0.75133973],
       [-0.4872893 ],
       [ 1.6626885 ]], dtype=float32)>
```

Listing 3.6 Assigning a value to a TensorFlow variable

```
>>> v.assign(tf.ones((3, 1)))
array([[1.],
       [1.],
       [1.]], dtype=float32)>
```

Listing 3.8 Using assign_add()

```
>>> v.assign_add(tf.ones((3, 1)))
array([[2.],
       [2.],
       [2.]], dtype=float32)>
```

Tensor operations: Doing math in TensorFlow

Listing 3.9 A few basic math operations

```
a = tf.ones((2, 2))
b = tf.square(a)
c = tf.sqrt(a)
d = b + c
e = tf.matmul(a, b)
e *= d
```

Multiply two tensors
(element-wise).

Take the square.

Take the square root.

Add two tensors (element-wise).

Take the product of two tensors
(as discussed in chapter 2).

Like NumPy

NumPy Can't Do

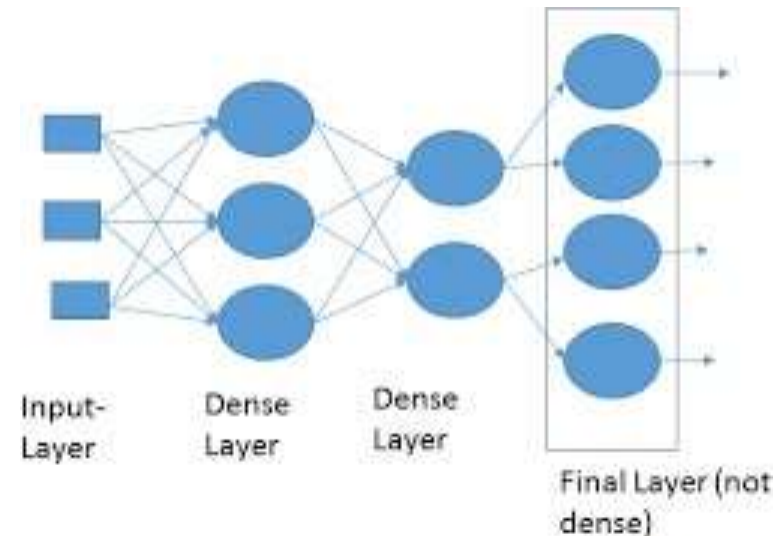
Listing 3.10 Using the GradientTape

```
input_var = tf.Variable(initial_value=3.)
with tf.GradientTape() as tape:
    result = tf.square(input_var)
gradient = tape.gradient(result, input_var)
```

Anatomy of a neural network: Understanding core Keras APIs

Layers: The building blocks of deep learning

- A **layer** is a data processing module that takes as input one or more tensors and that outputs one or more tensors.
- In Keras, a Layer is an object that encapsulates some state (weights) and some computation (a forward pass).



Listing 3.22 A Dense layer implemented as a Layer subclass

```
from tensorflow import keras  
  
class SimpleDense(keras.layers.Layer):
```

All Keras layers inherit
from the base Layer class.



The “compile” step: Configuring the learning process

- Once the model architecture is defined, we have to choose three more things
 - **Loss function** — a measure of success that will be minimized during training.
 - **Optimizer**—Determines how the network will be updated based on the loss function.
 - **Metrics**—The measures of success you want to monitor during training and validation, such as classification accuracy.
- Use the built-in **compile()** and **fit()** methods to start training model
 - The **compile()** method configures the training process.
 - The **fit()** method implements the training loop itself

The “compile” step...

```
model = keras.Sequential([keras.layers.Dense(1)])  
model.compile(optimizer="rmsprop",  
              loss="mean_squared_error",  
              metrics=["accuracy"])
```

Define a linear classifier.

Specify the optimizer by name: RMSprop (it's case-insensitive).

Specify the loss by name: mean squared error.

Specify a list of metrics: in this case, only accuracy.

The “compile” step: Keras Options

- **Optimizers:**

- SGD (with or without momentum)
- RMSprop
- Adam
- Adagrad

- **Losses:**

- CategoricalCrossentropy
- SparseCategoricalCrossentropy
- BinaryCrossentropy
- MeanSquaredError
- KLDivergence
- CosineSimilarity

The “compile” step: Keras Options

- **Metrics:**
 - CategoricalAccuracy
 - SparseCategoricalAccuracy
 - BinaryAccuracy
 - AUC
 - Precision
 - Recall

Understanding the fit() method

Listing 3.23 Calling fit() with NumPy data

```
history = model.fit(  
    inputs,  
    targets,  
    epochs=5,  
    batch_size=128  
)
```

The input examples,
as a NumPy array

The corresponding
training targets, as
a NumPy array

The training loop will
iterate over the data in
batches of 128 examples.

The training loop
will iterate over the
data 5 times.

```
>>> history.history  
{  
  "binary_accuracy": [0.855, 0.9565, 0.9555, 0.95, 0.951],  
  "loss": [0.6573270302042366,  
           0.07434618508815766,  
           0.07687718723714351,  
           0.07412414988875389,  
           0.07617757616937161]}
```

Monitoring loss and metrics on validation data

```
model.fit(  
    training_inputs,      | Training data, used to update  
    training_targets,    | the weights of the model  
    epochs=5,  
    batch_size=16,  
    validation_data=(val_inputs, val_targets) ← Validation data, used only  
    )                   | to monitor the validation  
                       | loss and metrics
```

Inference: Using a model after training

```
predictions = model(new_inputs)
```

← Takes a NumPy array or TensorFlow tensor and returns a TensorFlow tensor

```
predictions = model.predict(new_inputs, batch_size=128)
```

← Takes a NumPy array or a Dataset and returns a NumPy array

```
>>> predictions = model.predict(val_inputs, batch_size=128)
```

```
>>> print(predictions[:10])
```

```
[[0.3590725 ]  
 [0.82706255]  
 [0.74428225]  
 [0.682058  ]  
 [0.7312616 ]  
 [0.6059811 ]  
 [0.78046083]  
 [0.025846  ]  
 [0.16594526]  
 [0.72068727]]
```

Results of `predict()` on some validation data with the linear model we trained earlier. we get scalar scores that correspond to the model's prediction for each input sample.

Digit Recognition Example: Model configuration

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>
11490434/11490434 [=====] - 2s 0us/step

```
from tensorflow import keras
from tensorflow.keras import layers
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

- Here, our model consists of a sequence of two Dense layers. The second (and last) layer is a 10-way *softmax classification* layer, which means it will return an array of 10 probability scores (summing to 1).
- Each score will be the probability that the current digit image belongs to one of our 10 digit classes.

Digit Recognition Example: ...Data preprocessing

```
model.compile(optimizer="rmsprop",  
loss="sparse_categorical_crossentropy",  
metrics=["accuracy"])
```

```
train_images = train_images.reshape((60000, 28 * 28))  
train_images = train_images.astype("float32") / 255  
test_images = test_images.reshape((10000, 28 * 28))  
test_images = test_images.astype("float32") / 255
```

- Before training, the data is reshaped into the shape the model expects and scaled it so that all values are in the [0, 1] interval.

Digit Recognition Example: Training the model

```
model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

Epoch 1/5

469/469 [=====] - 4s 3ms/step - loss: 0.2559 - accuracy: 0.9253

Epoch 2/5

469/469 [=====] - 1s 3ms/step - loss: 0.1030 - accuracy: 0.9697

Epoch 3/5

469/469 [=====] - 1s 3ms/step - loss: 0.0683 - accuracy: 0.9797

Epoch 4/5

469/469 [=====] - 1s 3ms/step - loss: 0.0488 - accuracy: 0.9852

Epoch 5/5

469/469 [=====] - 2s 4ms/step - loss: 0.0371 - accuracy: 0.9893

<keras.callbacks.History at 0x7f0750517040>

Training
accuracy

Digit Recognition Example: Model testing

```
test_digits = test_images[0:10]
predictions = model.predict(test_digits)
predictions[0]
```

```
1/1 [=====] - 0s 60ms/step
array([3.8975578e-09, 4.2572078e-11, 5.4741922e-06, 7.7760007e-05,
       6.4008993e-12, 5.3976187e-08, 3.1037809e-15, 9.9991536e-01,
       2.4116693e-07, 1.1648450e-06], dtype=float32)
```

```
test_labels[0]
```

```
7
```

average accuracy over
the entire test set

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"test_acc: {test_acc}")
```

```
313/313 [=====] - 1s 2ms/step - loss: 0.0670 - accuracy: 0.9797
test_acc: 0.9797000288963318
```

- Test-set accuracy turns out to be 97.8%—that's a bit **lower than** the training set accuracy (98.9%).
- This gap is an example of **overfitting**: the fact that machine learning models tend to perform worse on new data than on their training data.

Summary

- TensorFlow is an industry-strength numerical computing framework that can run on CPU, GPU, or TPU. It can **automatically compute the gradient** of any differentiable expression.
- Key TensorFlow objects include tensors, variables, tensor operations, and the gradient tape.
- The central class of Keras is the **Layer**. A *layer* encapsulates some weights and some computation. Layers are assembled into *models*.

Summary

- Before you start training a model, you need to pick an *optimizer, a loss, and some metrics*, which you specify via the *model.compile()* method.
- To train a model, you can use the *fit()* method, which runs mini-batch gradient descent for you. You can also use it to monitor your loss and metrics on *validation data*.
- Once your model is trained, you use the *model.predict()* method to generate predictions on new inputs.