CH3. Introduction to Keras and TensorFlow

Il-Youp Kwak

Chung-Ang University

Introduction to Keras and TensorFlow

What’s Tensorflow

What’s Keras

Keras and Tensorflow: A brief history

First steps with Tensorflow

Anatomy of a neural network: Understanding Keras APIs

What's TensorFlow?

TensorFlow is a Python-based, free, open source machine learning platform, developed by Google.

- Compute the gradient of any differentiable expression
- Run not only on CPUs, but also on GPUs and TPUs, parallel hardware accelerators.
- Computation defined in TensorFlow can be easily distributed
- TensorFlow programs can be exported to other runtimes (C++, Java)

TensorFlow is much more than a single library. It's really a platform, home to a vast ecosystem of components,

What’s Keras?

Keras is a deep learning API for Python, built on top of TensorFlow

Provides a convenient way to define and train deep learning model.

Developed for research with the aim of enabling fast experimentation

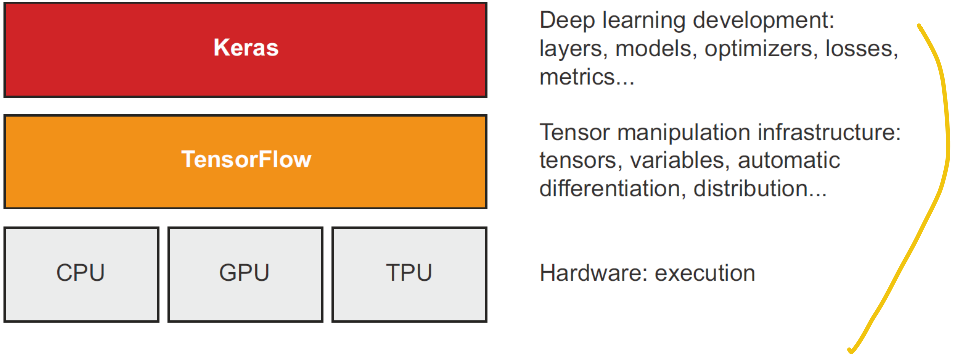

Figure 3.1 Keras and TensorFlow: TensorFlow is a low-level tensor computing platform, and Keras is a high-level deep learning API

Keras

- **Allows the same code to run seamlessly on CPU or GPU**
- **Has a user-friendly API that makes it easy to quickly prototype deep-learning models.**
- **Has built-in support for convolutional networks, recurrent networks, and any combination of both.**

MIT license, it can be freely used in commercial projects.

Keras and TensorFlow: A brief history

Keras predates TensorFlow by eight months. Keras ~~It~~ was released in March 2015, and TensorFlow was released in November 2015.

Keras was originally built on top of Theano, another tensor-manipulation library that provided automatic differentiation and GPU support

In late 2015, after the release of TensorFlow, Keras was refactored to a multibackend architecture: Use Keras with either Theano or TensorFlow

In 2017, two new additional backend options were added to Keras: CNTK and MXNet

In 2018, the TensorFlow leadership picked Keras as TensorFlow's official high-level API. As a result, the Keras API is front and center in TensorFlow 2.0, released in September 2019

First steps with TensorFlow

Training a neural network revolves around the following concepts:

1. low-level tensor manipulation: tensors, tensor operations, backpropagation
2. high-level deep learning concepts: layers, loss function, metrics, training loop

Constant tensors and variables

To do anything in TensorFlow, we're going to need some 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)) (for `tf.ones(shape=(2, 1))`)

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

$\begin{pmatrix}1\\1\end{pmatrix}$ $\begin{pmatrix}0\\0\end{pmatrix}$

Listing 3.2 Random tensors

```
>>> x = tf.random.normal(shape=(3, 1), mean=0., stddev=1.)
>>> print(x)
tf.Tensor(
[[-0.14208166]
 [-0.95319825]
 [ 1.1096532 ]], shape=(3, 1), dtype=float32)
>>> x = tf.random.uniform(shape=(3, 1), minval=0., maxval=1.)
>>> print(x)
tf.Tensor(
```

Tensor of random values drawn from a normal distribution with mean 0 and standard deviation 1. Equivalent to np.random.normal(size=(3, 1), loc=0., scale=1.).

Tensor of random values drawn from a uniform distribution between 0 and 1. Equivalent to np.random.uniform(size=(3, 1), low=0., high=1.).

A significant difference between NumPy arrays and TensorFlow tensors is that 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.
```

$$\begin{pmatrix}1&1\\1&1\end{pmatrix} \to \begin{pmatrix}0&1\\1&1\end{pmatrix}$$

Try to do the same thing in TensorFlow, and you will get an error: "EagerTensor object does not support item assignment."

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.

$$\begin{pmatrix}1&1\\1&1\end{pmatrix} \not\to \begin{pmatrix}0&1\\1&1\end{pmatrix}$$

To train a model, we'll need to update its state. That's where variables come in. tf.Variable is the class meant to manage modifiable state in TensorFlow

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)>
```

State of a variable can be modified via its assign method

Listing 3.6 Assigning a value to a TensorFlow variable

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

Listing 3.7 Assigning a value to a subset of a TensorFlow variable

```
>>> v[0, 0].assign(3.)
array([[3.],
       [1.],
       [1.]], dtype=float32)>
```

assign_add() and assign_sub() are efficient equivalents of += and -=

Listing 3.8 Using `assign_add()`

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

$v = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$

$v \mathrel{+}= \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$

$v = v + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$

$= \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix}$

$a \mathrel{+}= 1$

$a = a + 1$

$a \mathrel{-}= 1$

$a = a - 1$

Tensor operations: Doing math in TensorFlow

Just like NumPy, TensorFlow offers a large collection of tensor operations to express mathematical formulas.

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
```

- Take the square.
- Take the square root.
- Add two tensors (element-wise).
- Take the product of two tensors (as discussed in chapter 2).
- Multiply two tensors (element-wise).

$a = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$

$b = \begin{pmatrix} 1^2 & 1^2 \\ 1^2 & 1^2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$

$c = \begin{pmatrix} \sqrt{1} & \sqrt{1} \\ \sqrt{1} & \sqrt{1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$

Importantly, each of the preceding operations gets executed on the fly: at any point, you can print what the current result is, just like in NumPy. We call this eager execution.

A second look at the GradientTape API

TensorFlow seems to look a lot like NumPy. But here's something NumPy can't do:

$y = x^2$

$\frac{dy}{dx} = 2x$

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)
```

Retrieve the gradients of the loss of a model with respect to its weights: gradients = tape.gradient(loss, weights)

Only trainable variables are tracked by default. With a constant tensor, you'd have to manually mark it as being tracked by calling tape.watch() on it.

Listing 3.11 Using `GradientTape` with constant tensor inputs

```
input_const = tf.constant(3.)
with tf.GradientTape() as tape:
    tape.watch(input_const)
    result = tf.square(input_const)
gradient = tape.gradient(result, input_const)
```

An end-to-end example: A linear classifier in pure TensorFlow

Synthetic data: two classes of points in a 2D plane.

Listing 3.13 Generating two classes of random points in a 2D plane

```
num_samples_per_class = 1000
negative_samples = np.random.multivariate_normal(
    mean=[0, 3],
    cov=[[1, 0.5],[0.5, 1]],
    size=num_samples_per_class)
positive_samples = np.random.multivariate_normal(
    mean=[3, 0],
    cov=[[1, 0.5],[0.5, 1]],
    size=num_samples_per_class)
```

Generate the first class of points: 1000 random 2D points. cov=[[1, 0.5],[0.5, 1]] corresponds to an oval-like point cloud oriented from bottom left to top right.

Generate the other class of points with a different mean and the same covariance matrix.

Listing 3.14 Stacking the two classes into an array with shape (2000, 2)

```
inputs = np.vstack((negative_samples, positive_samples)).astype(np.float32)
```

Listing 3.15 Generating the corresponding targets (0 and 1)

```
targets = np.vstack((np.zeros((num_samples_per_class, 1), dtype="float32"),
                     np.ones((num_samples_per_class, 1), dtype="float32")))
```

$$(\overbrace{0,0,0,\ldots,0}^{1000},\ \underbrace{1,1,1,\ldots,1}_{1000})$$

Listing 3.16 Plotting the two point classes (see figure 3.6)

```
import matplotlib.pyplot as plt
plt.scatter(inputs[:, 0], inputs[:, 1], c=targets[:, 0])
plt.show()
```

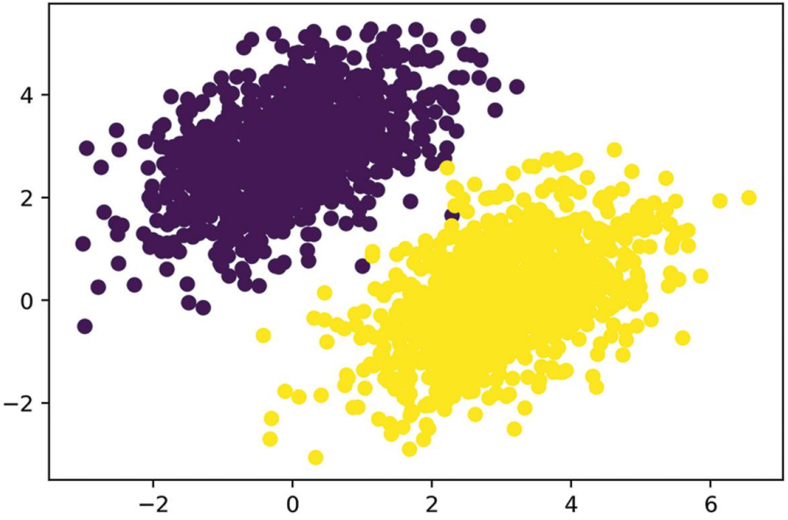

Figure 3.6 Our synthetic data: two classes of random points in the 2D plane

Create a linear classifier

Listing 3.17 Creating the linear classifier variables

The inputs will be 2D points.

The output predictions will be a single score per sample (close to 0 if the sample is predicted to be in class 0, and close to 1 if the sample is predicted to be in class 1).

$$= \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} [x, y] + b$$

2 1

```
input_dim = 2
output_dim = 1
W = tf.Variable(initial_value=tf.random.uniform(shape=(input_dim, output_dim)))
b = tf.Variable(initial_value=tf.zeros(shape=(output_dim,)))
```

1

Listing 3.18 The forward pass function

```
def model(inputs):
    return tf.matmul(inputs, W) + b
```

prediction = w1 * x + w2 * y + b.

Define our loss function

Listing 3.19 The mean squared error loss function MSE

per_sample_losses will be a tensor with the same shape as targets and predictions, containing per-sample loss scores.

```
def square_loss(targets, predictions):
    per_sample_losses = tf.square(targets - predictions)
    return tf.reduce_mean(per_sample_losses)
```

We need to average these per-sample loss scores into a single scalar loss value: this is what reduce_mean does.

Training step using gradient decent

Listing 3.20 The training step function

```
learning_rate = 0.1

def training_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = square_loss(predictions, targets)
    grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [W, b])
    W.assign_sub(grad_loss_wrt_W * learning_rate)
    b.assign_sub(grad_loss_wrt_b * learning_rate)
    return loss
```

Retrieve the gradient of the loss with regard to weights.

Forward pass, inside a gradient tape scope

Update the weights.

Training step using gradient decent

Listing 3.20 The training step function

```
learning_rate = 0.1

def training_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = square_loss(predictions, targets)
    grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [W, b])
    W.assign_sub(grad_loss_wrt_W * learning_rate)
    b.assign_sub(grad_loss_wrt_b * learning_rate)
    return loss
```

Retrieve the gradient of the loss with regard to weights.

Forward pass, inside a gradient tape scope

Update the weights.

40 steps of training (40 epochs)

Listing 3.21 The batch training loop

```
for step in range(40):
    loss = training_step(inputs, targets)
    print(f"Loss at step {step}: {loss:.4f}")
```

prediction = w1 * x + w2 * y + b. Thus, class 0 is defined as w1 * x + w2 * y + b < 0.5, and class 1 is defined as w1 * x + w2 * y + b > 0.5

Rule w1 * x + w2 * y + b = 0.5 becomes y = - w1 / w2 * x + (0.5 - b) / w2

```
x = np.linspace(-1, 4, 100)
y = - W[0] /  W[1] * x + (0.5 - b) / W[1]
plt.plot(x, y, "-r")
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
```

Generate 100 regularly spaced numbers between –1 and 4, which we will use to plot our line.

This is our line's equation.

Plot our line ("-r" means "plot it as a red line").

Plot our model's predictions on the same plot.

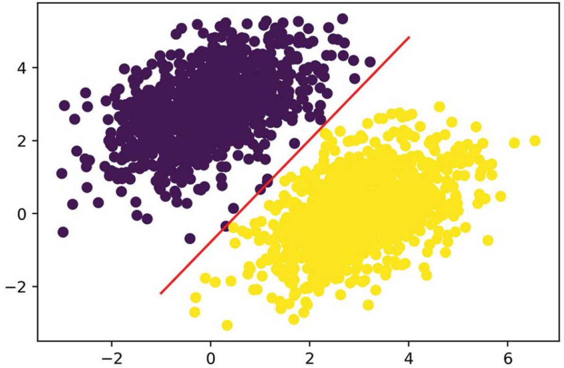

Figure 3.8 Our model, visualized as a line

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 tensors and that outputs tensors

Ex) densely connected layer, called fully connected or dense layer (Dense class in Keras)

Base Layer class in Keras

Layer is object that encapsulates some state (weights) and computation (forward pass)

weights are typically defined in a build() (although they could also be created in the constructor, __init__()), and the computation is defined in the call() method.

Listing 3.22 A Dense layer implemented as a Layer subclass

```
from tensorflow import keras

class SimpleDense(keras.layers.Layer):

    def __init__(self, units, activation=None):
        super().__init__()
        self.units = units
        self.activation = activation

    def build(self, input_shape):
        input_dim = input_shape[-1]
        self.W = self.add_weight(shape=(input_dim, self.units),
                                 initializer="random_normal")
        self.b = self.add_weight(shape=(self.units,),
                                 initializer="zeros")

    def call(self, inputs):
        y = tf.matmul(inputs, self.W) + self.b
        if self.activation is not None:
            y = self.activation(y)
        return y
```

All Keras layers inherit from the base Layer class.

Weight creation takes place in the build() method.

We define the forward pass computation in the call() method.

add_weight() is a shortcut method for creating weights. It is also possible to create standalone variables and assign them as layer attributes, like self.W = tf.Variable(tf.random.uniform(w_shape)).

X × W + b

1×a a×b 1×b

a

xW+b

The __call__() method of the base layer looks like this:

```
def __call__(self, inputs):
    if not self.built:
        self.build(inputs.shape)
        self.built = True
    return self.call(inputs)
```

Base Layer class in Keras

Once instantiated, a layer like this can be used just like a function, taking as input a TensorFlow tensor:

```
>>> my_dense = SimpleDense(units=32, activation=tf.nn.relu)
>>> input_tensor = tf.ones(shape=(2, 784))
>>> output_tensor = my_dense(input_tensor)
>>> print(output_tensor.shape)
(2, 32))
```

Instantiate our layer, defined previously.

Create some test inputs.

Call the layer on the inputs, just like a function.

Keras에서는 쉽게 표현

```
from tensorflow.keras import layers
layer = layers.Dense(32, activation="relu")
```

A dense layer with 32 output units

Automatic shape inference

layers didn't receive any information about the shape of their inputs—instead, they automatically inferred their input shape as being the shape of the first inputs they see.

```
from tensorflow.keras import models
from tensorflow.keras import layers
model = models.Sequential([
    layers.Dense(32, activation="relu"),
    layers.Dense(32)
])
```

Compare the Dense layer we implemented in chapter 2

```
model = NaiveSequential([
    NaiveDense(input_size=784, output_size=32, activation="relu"),
    NaiveDense(input_size=32, output_size=64, activation="relu"),
    NaiveDense(input_size=64, output_size=32, activation="relu"),
    NaiveDense(input_size=32, output_size=10, activation="softmax")
])
```

=

더 간단

```
model = keras.Sequential([
    SimpleDense(32, activation="relu"),
    SimpleDense(64, activation="relu"),
    SimpleDense(32, activation="relu"),
    SimpleDense(10, activation="softmax")
])
```

From layers to models

A deep learning model is a graph of layers. So far we've learned Sequential model

There are much broader variety of network topologies: Two-branch networks, Multihead networks, Residual connections, etc.

There are generally two ways of building such models in Keras: you could directly subclass the Model class, or you could use the Functional API, which lets you do more with less code. (Will be covered from chapter 7)

Picking the right network architecture is more an art than a science, and although there are some best practices and principles you can rely on.

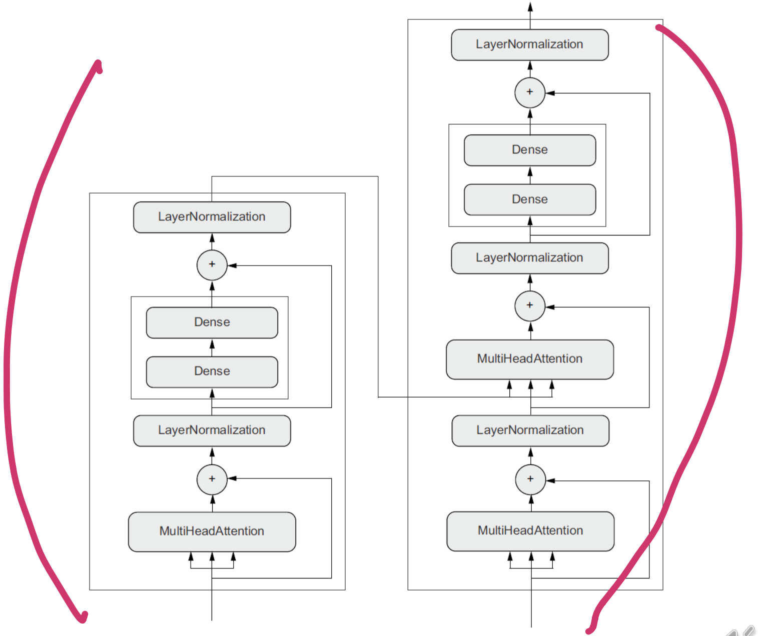

Figure 3.9 The Transformer architecture (covered in chapter 11). There's a lot going on here. Throughout the next few chapters, you'll climb your way up to understanding it.

Picking a loss function

Choosing the right loss function for the right problem is extremely important

There are simple guidelines you can follow to choose:

1. **Binary crossentropy for a two-class classification**
2. **Categorical crossentropy for a many-class classification problem**
3. **Mean squared error for regression problem**

Understanding the fit() method

key arguments:

- **The data (inputs and targets) to train on: NumPy arrays or a TensorFlow Dataset object.**
- **The number of epochs to train for: how many times the training loop iterate over the data passed.**
- **The batch size to use within each epoch of mini-batch gradient descent: the number of training examples considered to compute the gradients for one weight update step.**

Listing 3.23 Calling `fit()` with NumPy data

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

- `inputs`: The input examples, as a NumPy array
- `targets`: The corresponding training targets, as a NumPy array
- `epochs=5`: The training loop will iterate over the data 5 times.
- `batch_size=128`: The training loop will iterate over the data in batches of 128 examples.

```
>>> 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

To keep an eye on how the model does on new data, it's standard practice to reserve a subset of the training data as validation data: you won't be training the model on this data, but you will use it to compute a loss value and metrics value

Listing 3.24 Using the `validation_data` argument

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=0.1),
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])

indices_permutation = np.random.permutation(len(inputs))
shuffled_inputs = inputs[indices_permutation]
shuffled_targets = targets[indices_permutation]

num_validation_samples = int(0.3 * len(inputs))
val_inputs = shuffled_inputs[:num_validation_samples]
val_targets = shuffled_targets[:num_validation_samples]
training_inputs = shuffled_inputs[num_validation_samples:]
training_targets = shuffled_targets[num_validation_samples:]
model.fit(
    training_inputs,
    training_targets,
    epochs=5,
    batch_size=16,
    validation_data=(val_inputs, val_targets)
)
```

model 생성

To avoid having samples from only one class in the validation data, shuffle the inputs and targets using a random indices permutation.

Reserve 30% of the training inputs and targets for validation (we'll exclude these samples from training and reserve them to compute the validation loss and metrics).

7:3

Training data, used to update the weights of the model

Validation data, used only to monitor the validation loss and metrics

```
loss_and_metrics = model.evaluate(val_inputs, val_targets, batch_size=128)
```

You can compute the validation loss and metrics after the training is complete

Inference: Using a model after training

Once you've trained your model, you're going to want to use it to make predictions on new data

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

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

However, this will process all inputs in new_inputs at once, which may not be feasible if you're looking at a lot of data

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

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

128
128

Thank you!