02_how_to_use_tensorflow

September 29, 2021

1 How to use Tensorflow 2

Keras was designed as a high-level or meta API to accelerate the iterative workflow when designing and training deep neural networks with computational backends like TensorFlow, Theano, or CNTK. It has been integrated into TensorFlow in 2017 and is set to become the principal TensorFlow interface with the 2.0 release. You can also combine code from both libraries to leverage Keras' high-level abstractions as well as customized TensorFlow graph operations.

Please follow the installations instructions in `Installation Guide.md` in the root folder.

1.1 Imports & Settings

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
%matplotlib inline

from pathlib import Path
from copy import deepcopy
import numpy as np
import pandas as pd

import sklearn
from sklearn.datasets import make_circles # To generate the dataset

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras import optimizers
from tensorflow.keras.callbacks import TensorBoard

import matplotlib
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from mpl_toolkits.mplot3d import Axes3D  # 3D plots

import seaborn as sns
```

[3]:
```
# plotting style
sns.set_style('white')
# for reproducibility
np.random.seed(seed=42)
```

[4]:
```
gpu_devices = tf.config.experimental.list_physical_devices('GPU')
if gpu_devices:
    print('Using GPU')
    tf.config.experimental.set_memory_growth(gpu_devices[0], True)
else:
    print('Using CPU')
```

```
Using CPU
```

[5]:
```
results_path = Path('results')
if not results_path.exists():
    results_path.mkdir()
```

1.2 Input Data

1.2.1 Generate random data

The target `y` represents two classes generated by two circular distribution that are not linearly separable because class 0 surrounds class 1.

[6]:
```
# dataset params
N = 50000
factor = 0.1
noise = 0.1
```

[7]:
```
# generate data
X, y = make_circles(
    n_samples=N,
    shuffle=True,
    factor=factor,
    noise=noise)
```

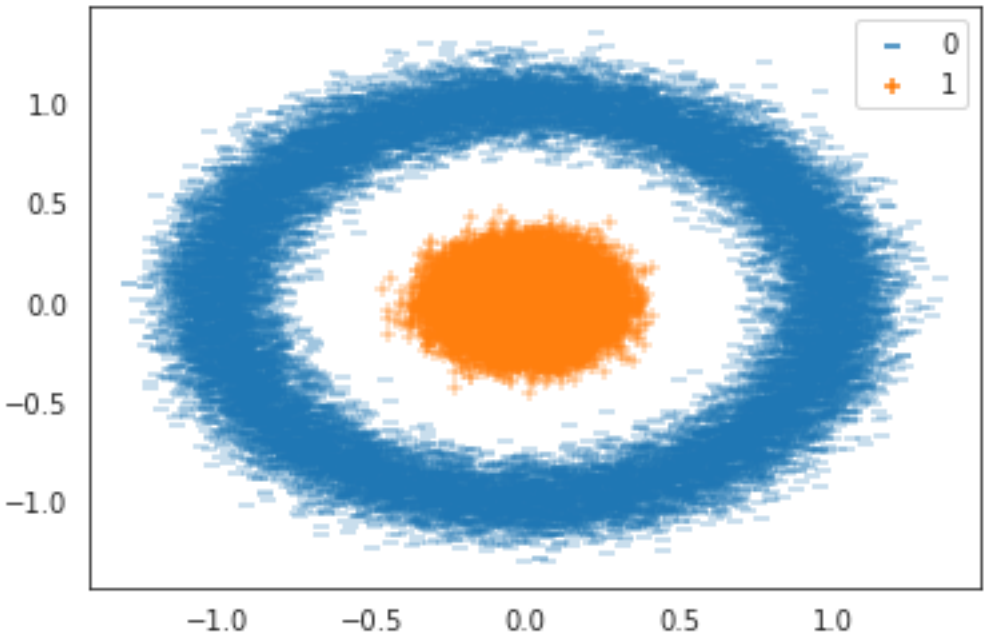
[8]:
```
# define outcome matrix
Y = np.zeros((N, 2))
for c in [0, 1]:
    Y[y == c, c] = 1
```

[9]:
```
f'Shape of: X: {X.shape} | Y: {Y.shape} | y: {y.shape}'
```

[9]:
```
'Shape of: X: (50000, 2) | Y: (50000, 2) | y: (50000,)'
```

1.2.2 Visualize Data

[10]:
```
sns.scatterplot(x=X[:, 0],
                y=X[:, 1],
                hue=y,
               style=y,
               markers=['_', '+']);
```



1.3 Build Keras Model

Keras supports both a slightly simpler Sequential and more flexible Functional API. We will introduce the former at this point and use the Functional API in more complex examples in the following chapters.

To create a model, we just need to instantiate a Sequential object and provide a list with the sequence of standard layers and their configurations, including the number of units, type of activation function, or name.

1.3.1 Define Architecture

[11]:
```
model = Sequential([
    Dense(units=3, input_shape=(2,), name='hidden'),
    Activation('sigmoid', name='logistic'),
    Dense(2, name='output'),
    Activation('softmax', name='softmax'),
])
```

The first hidden layer needs information about the number of features in the matrix it receives from the input layer via the input_shape argument. In our simple case, these are just two. Keras infers the number of rows it needs to process during training, through the batch_size argument that we will pass to the fit method below.

Keras infers the sizes of the inputs received by other layers from the previous layer's units argument.

Keras provides numerous standard building blocks, including recurrent and convolutional layers, various options for regularization, a range of loss functions and optimizers, and also preprocessing, visualization and logging (see documentation on GitHub for reference). It is also extensible.

The model's summary method produces a concise description of the network architecture, including a list of the layer types and shapes, and the number of parameters:

[12]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
hidden (Dense)               (None, 3)                 9
_________________________________________________________________
logistic (Activation)        (None, 3)                 0
_________________________________________________________________
output (Dense)               (None, 2)                 8
_________________________________________________________________
softmax (Activation)         (None, 2)                 0
=================================================================
Total params: 17
Trainable params: 17
Non-trainable params: 0
_________________________________________________________________
```

1.4 Compile Model

Next, we compile the Sequential model to configure the learning process. To this end, we define the optimizer, the loss function, and one or several performance metrics to monitor during training:

[13]:
```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

1.5 Tensorboard Callback

Keras uses callbacks to enable certain functionality during training, such as logging information for interactive display in TensorBoard (see next section):

[14]:
```
tb_callback = TensorBoard(log_dir=results_path / 'tensorboard',
                          histogram_freq=1,
                          write_graph=True,
```

```
                                write_images=True)
```

1.6 Train Model

To train the model, we call its fit method and pass several parameters in addition to the training data:

[15]:
```
training=model.fit(X,
          Y,
          epochs=50,
          validation_split=.2,
          batch_size=128,
          verbose=1,
          callbacks=[tb_callback])
```

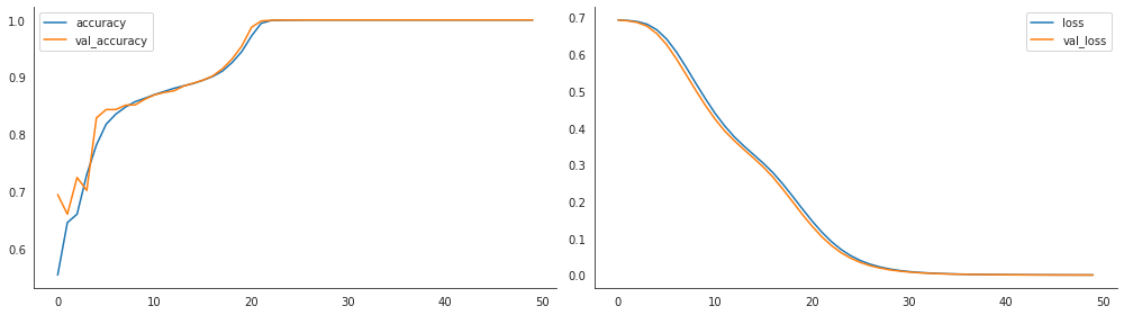
```
Epoch 1/50
  1/313 […] - ETA: 0s - loss: 0.7017 - accuracy:
0.3125WARNING:tensorflow:From
/home/stefan/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-
packages/tensorflow/python/ops/summary_ops_v2.py:1277: stop (from
tensorflow.python.eager.profiler) is deprecated and will be removed after
2020-07-01.
Instructions for updating:
use `tf.profiler.experimental.stop` instead.
WARNING:tensorflow:Callbacks method `on_train_batch_end` is slow compared to the
batch time (batch time: 0.0010s vs `on_train_batch_end` time: 0.0070s). Check
your callbacks.
313/313 [==============================] - 0s 1ms/step - loss: 0.6932 -
accuracy: 0.5534 - val_loss: 0.6929 - val_accuracy: 0.6943
Epoch 2/50
313/313 [==============================] - 0s 739us/step - loss: 0.6917 -
accuracy: 0.6453 - val_loss: 0.6908 - val_accuracy: 0.6600
Epoch 3/50
313/313 [==============================] - 0s 707us/step - loss: 0.6888 -
accuracy: 0.6600 - val_loss: 0.6863 - val_accuracy: 0.7242
Epoch 4/50
313/313 [==============================] - 0s 729us/step - loss: 0.6815 -
accuracy: 0.7295 - val_loss: 0.6760 - val_accuracy: 0.7016
Epoch 5/50
313/313 [==============================] - 0s 746us/step - loss: 0.6663 -
accuracy: 0.7814 - val_loss: 0.6559 - val_accuracy: 0.8286
Epoch 6/50
313/313 [==============================] - 0s 739us/step - loss: 0.6414 -
accuracy: 0.8174 - val_loss: 0.6264 - val_accuracy: 0.8431
Epoch 7/50
313/313 [==============================] - 0s 787us/step - loss: 0.6068 -
accuracy: 0.8352 - val_loss: 0.5882 - val_accuracy: 0.8432
Epoch 8/50
```

```
313/313 [==============================] - 0s 704us/step - loss: 0.5656 -
accuracy: 0.8475 - val_loss: 0.5454 - val_accuracy: 0.8508
Epoch 9/50
313/313 [==============================] - 0s 689us/step - loss: 0.5219 -
accuracy: 0.8569 - val_loss: 0.5027 - val_accuracy: 0.8514
Epoch 10/50
313/313 [==============================] - 0s 748us/step - loss: 0.4796 -
accuracy: 0.8629 - val_loss: 0.4613 - val_accuracy: 0.8613
Epoch 11/50
313/313 [==============================] - 0s 737us/step - loss: 0.4398 -
accuracy: 0.8694 - val_loss: 0.4241 - val_accuracy: 0.8689
Epoch 12/50
313/313 [==============================] - 0s 838us/step - loss: 0.4053 -
accuracy: 0.8749 - val_loss: 0.3917 - val_accuracy: 0.8731
Epoch 13/50
313/313 [==============================] - 0s 692us/step - loss: 0.3754 -
accuracy: 0.8803 - val_loss: 0.3648 - val_accuracy: 0.8759
Epoch 14/50
313/313 [==============================] - 0s 780us/step - loss: 0.3503 -
accuracy: 0.8846 - val_loss: 0.3408 - val_accuracy: 0.8844
Epoch 15/50
313/313 [==============================] - 0s 688us/step - loss: 0.3271 -
accuracy: 0.8888 - val_loss: 0.3176 - val_accuracy: 0.8893
Epoch 16/50
313/313 [==============================] - 0s 723us/step - loss: 0.3037 -
accuracy: 0.8945 - val_loss: 0.2934 - val_accuracy: 0.8946
Epoch 17/50
313/313 [==============================] - 0s 741us/step - loss: 0.2778 -
accuracy: 0.9010 - val_loss: 0.2656 - val_accuracy: 0.9020
Epoch 18/50
313/313 [==============================] - 0s 734us/step - loss: 0.2475 -
accuracy: 0.9105 - val_loss: 0.2334 - val_accuracy: 0.9145
Epoch 19/50
313/313 [==============================] - 0s 722us/step - loss: 0.2146 -
accuracy: 0.9255 - val_loss: 0.1997 - val_accuracy: 0.9318
Epoch 20/50
313/313 [==============================] - 0s 788us/step - loss: 0.1806 -
accuracy: 0.9448 - val_loss: 0.1655 - val_accuracy: 0.9545
Epoch 21/50
313/313 [==============================] - 0s 770us/step - loss: 0.1478 -
accuracy: 0.9721 - val_loss: 0.1337 - val_accuracy: 0.9872
Epoch 22/50
313/313 [==============================] - 0s 807us/step - loss: 0.1176 -
accuracy: 0.9942 - val_loss: 0.1048 - val_accuracy: 0.9982
Epoch 23/50
313/313 [==============================] - 0s 820us/step - loss: 0.0915 -
accuracy: 0.9990 - val_loss: 0.0808 - val_accuracy: 0.9994
Epoch 24/50
```

```
313/313 [==============================] - 0s 904us/step - loss: 0.0701 -
accuracy: 0.9995 - val_loss: 0.0613 - val_accuracy: 0.9998
Epoch 25/50
313/313 [==============================] - 0s 930us/step - loss: 0.0530 -
accuracy: 0.9998 - val_loss: 0.0463 - val_accuracy: 0.9999
Epoch 26/50
313/313 [==============================] - 0s 920us/step - loss: 0.0399 -
accuracy: 0.9999 - val_loss: 0.0347 - val_accuracy: 0.9999
Epoch 27/50
313/313 [==============================] - 0s 794us/step - loss: 0.0299 -
accuracy: 0.9999 - val_loss: 0.0260 - val_accuracy: 1.0000
Epoch 28/50
313/313 [==============================] - 0s 731us/step - loss: 0.0225 -
accuracy: 0.9999 - val_loss: 0.0196 - val_accuracy: 1.0000
Epoch 29/50
313/313 [==============================] - 0s 762us/step - loss: 0.0170 -
accuracy: 1.0000 - val_loss: 0.0148 - val_accuracy: 1.0000
Epoch 30/50
313/313 [==============================] - 0s 732us/step - loss: 0.0129 -
accuracy: 1.0000 - val_loss: 0.0113 - val_accuracy: 1.0000
Epoch 31/50
313/313 [==============================] - 0s 749us/step - loss: 0.0099 -
accuracy: 1.0000 - val_loss: 0.0087 - val_accuracy: 1.0000
Epoch 32/50
313/313 [==============================] - 0s 800us/step - loss: 0.0077 -
accuracy: 1.0000 - val_loss: 0.0068 - val_accuracy: 1.0000
Epoch 33/50
313/313 [==============================] - 0s 1ms/step - loss: 0.0060 -
accuracy: 1.0000 - val_loss: 0.0054 - val_accuracy: 1.0000
Epoch 34/50
313/313 [==============================] - 0s 894us/step - loss: 0.0048 -
accuracy: 1.0000 - val_loss: 0.0043 - val_accuracy: 1.0000
Epoch 35/50
313/313 [==============================] - 0s 626us/step - loss: 0.0039 -
accuracy: 0.9999 - val_loss: 0.0035 - val_accuracy: 1.0000
Epoch 36/50
313/313 [==============================] - 0s 712us/step - loss: 0.0032 -
accuracy: 1.0000 - val_loss: 0.0029 - val_accuracy: 1.0000
Epoch 37/50
313/313 [==============================] - 0s 643us/step - loss: 0.0026 -
accuracy: 0.9999 - val_loss: 0.0024 - val_accuracy: 1.0000
Epoch 38/50
313/313 [==============================] - 0s 689us/step - loss: 0.0022 -
accuracy: 1.0000 - val_loss: 0.0020 - val_accuracy: 1.0000
Epoch 39/50
313/313 [==============================] - 0s 852us/step - loss: 0.0019 -
accuracy: 0.9999 - val_loss: 0.0017 - val_accuracy: 1.0000
Epoch 40/50
```

```
313/313 [==============================] - 0s 682us/step - loss: 0.0016 -
accuracy: 0.9999 - val_loss: 0.0015 - val_accuracy: 1.0000
Epoch 41/50
313/313 [==============================] - 0s 687us/step - loss: 0.0014 -
accuracy: 0.9999 - val_loss: 0.0013 - val_accuracy: 1.0000
Epoch 42/50
313/313 [==============================] - 0s 656us/step - loss: 0.0013 -
accuracy: 0.9999 - val_loss: 0.0012 - val_accuracy: 1.0000
Epoch 43/50
313/313 [==============================] - 0s 979us/step - loss: 0.0011 -
accuracy: 0.9999 - val_loss: 0.0011 - val_accuracy: 1.0000
Epoch 44/50
313/313 [==============================] - 0s 900us/step - loss: 0.0010 -
accuracy: 0.9999 - val_loss: 9.5180e-04 - val_accuracy: 1.0000
Epoch 45/50
313/313 [==============================] - 0s 757us/step - loss: 9.3114e-04 -
accuracy: 0.9999 - val_loss: 8.5783e-04 - val_accuracy: 1.0000
Epoch 46/50
313/313 [==============================] - 0s 777us/step - loss: 8.5770e-04 -
accuracy: 0.9999 - val_loss: 7.9056e-04 - val_accuracy: 1.0000
Epoch 47/50
313/313 [==============================] - 0s 803us/step - loss: 7.8891e-04 -
accuracy: 0.9999 - val_loss: 7.2263e-04 - val_accuracy: 1.0000
Epoch 48/50
313/313 [==============================] - 0s 765us/step - loss: 7.4050e-04 -
accuracy: 0.9999 - val_loss: 6.7238e-04 - val_accuracy: 1.0000
Epoch 49/50
313/313 [==============================] - 0s 743us/step - loss: 6.9102e-04 -
accuracy: 0.9999 - val_loss: 6.3254e-04 - val_accuracy: 1.0000
Epoch 50/50
313/313 [==============================] - 0s 703us/step - loss: 6.4889e-04 -
accuracy: 0.9999 - val_loss: 5.9569e-04 - val_accuracy: 1.0000
```

[16]:
```
fig, axes = plt.subplots(ncols=2, figsize=(14,4))
pd.DataFrame(training.history)[['accuracy', 'val_accuracy']].plot(ax=axes[0])
pd.DataFrame(training.history)[['loss', 'val_loss']].plot(ax=axes[1])
sns.despine()
fig.tight_layout();
```



1.7 Get Weights

[17]:
```
hidden = model.get_layer('hidden').get_weights()
```

[18]:
```
[t.shape for t in hidden]
```

[18]:
```
[(2, 3), (3,)]
```

1.8 Plot Decision Boundary

The visualization of the decision boundary resembles the result from the manual network implementation. The training with Keras runs a multiple faster, though.

[19]:
```
n_vals = 200
x1 = np.linspace(-1.5, 1.5, num=n_vals)
x2 = np.linspace(-1.5, 1.5, num=n_vals)
xx, yy = np.meshgrid(x1, x2)  # create the grid
```

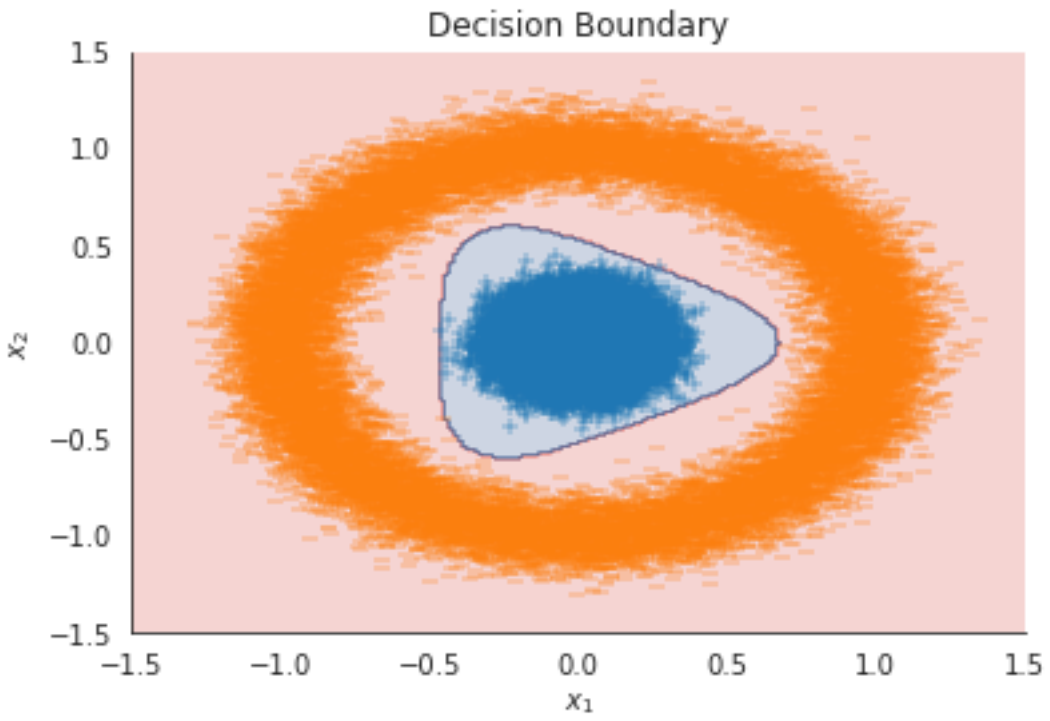
[20]:
```
X_ = np.array([xx.ravel(), yy.ravel()]).T
```

[21]:
```
y_hat = np.argmax(model.predict(X_), axis=1)
```

[22]:
```
# Create a color map to show the classification colors of each grid point
cmap = ListedColormap([sns.xkcd_rgb["pale red"],
                       sns.xkcd_rgb["denim blue"]])

# Plot the classification plane with decision boundary and input samples
plt.contourf(xx, yy, y_hat.reshape(n_vals, -1), cmap=cmap, alpha=.25)

# Plot both classes on the x1, x2 plane
data = pd.DataFrame(X, columns=['$x_1$', '$x_2$']).assign(Class=pd.Series(y).
 ↪map({0:'negative', 1:'positive'}))
sns.scatterplot(x='$x_1$', y='$x_2$', hue='Class', data=data, style=y,␣
 ↪markers=['_', '+'], legend=False)
sns.despine()
plt.title('Decision Boundary');
```



[23]:
```
%load_ext tensorboard
```

[24]:
```
%tensorboard --logdir results/tensorboard/
```

```
<IPython.core.display.HTML object>
```

[ ]:
```

```