TensorFlow 2 for Deep Learning

Hanzholah Shobri, Tria Rahmat M

12/2/24

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1

Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

Part I

Types of Machine Learning Problems

2 TensorFlow 2 for Regression Problems

TensorFlow 2 (TF2) provides a whole set of tools to implement machine learning to a wide range of problems, from preparing the data to deploying a model. In this post, I demonstrate how to use the package to build a regression model predicting continuous numerical values based on input data.

The objective of this demo is to show you the main elements of working with a TF2 model to tackle regression problems.

Important libraries are loaded, namely `tensorflow`, `numpy` to manipulate data, `pandas` to deal with table, and `seaborn` to create visualisation. As part of the setup, I also clean the `TF2` environment with `clear_session` function and set seed for random number generation using `set_random_seed` function.

For the visualisation, I attempted to use the `seaborn.objects` interface. The reason for this is that I am familiar with the `ggplot2` package in `R` when conducting data analysis, and I found that there is some similarity in both approach of creating a plot. For those who aren't familiar with `ggplot2` package, it employs the concept of layered grammar of graphics allowing you to describe any plots in a more structured way which result in more convenient and consistent code. You can see the documentations for the `seaborn.objects` interface here and the `ggplot2` package here.

```
import tensorflow as tf
import numpy as np
import pandas as pd
import seaborn.objects as so
```

```
tf.keras.backend.clear_session()
tf.keras.utils.set_random_seed(123)
```

2.1 Generate Random Data

For the sample problem, I generated a 1000 observations of random number with two independent variables x_0 and x_1. The target for prediction is calculated using simple formula below.

$$y = f(x) = 0.2 \times x_0 + 2.8 \times x_1 + \epsilon$$

All variables x_0 and x_1 as well as the error term ϵ follow a normal distribution $N(\mu, \sigma^2)$ with $\mu = 0$ and $\sigma^2 = 1$.

```
X = np.random.normal(size = (1000, 2))
y = 0.2 * X[:, 0] + 2.8 * X[:, 1] + np.random.normal(size = (1000,))
```

For evaluation of model, I split the data, 80% for training and 20% for testing.

```
X_train = X[:800]
y_train = y[:800]
X_test = X[800:]
y_test = y[800:]

X_train.shape, y_train.shape, X_test.shape, y_test.shape,
```

```
((800, 2), (800,), (200, 2), (200,))
```

2.2 A Very Simple TF2 Model

2.2.1 Defining the model

Defining a TF2 model can be accomplished easily by calling the `Sequential` method, followed by adding any types and number of layers. As the problem is very straightforward, tackling this should be relatively easy. For this reason, the model I defined here is very simple with only one `Dense` layer with one perceptron unit. In addition, we need to define the `input_shape` so that the model can determine the number of parameters it requires to predict all inputs. The `summary` method allows you to see the architecture of the model.

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(1, input_shape = (2, )))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 1)                 3
```

```
=================================================================
Total params: 3
Trainable params: 3
Non-trainable params: 0
_________________________________________________________________
```

Before training the model, you are required to choose the optimizer, loss function, and metrics, and `compile` those into the model. Here, I decided to use Stochastic Gradient Descent algorithm for optimizing the model (i.e., updating neural network parameters), Mean Squared Error (MSE) for determining how far the prediciton of the current model with actual values, and Mean Absolute Error (MAE) as a metric to evaluate the model.

```
model.compile(optimizer = "sgd", loss = "mse", metrics = ["mae"])
```

2.2.2 Training the model

Training can be done using `fit` method. The process is done after 100 `epochs` or cycles of the model updating its parameters based on input data. The `verbose` parameter that equals 0 means that the training will not print any information.

```
history = model.fit(X_train, y_train, epochs = 100, verbose = 0)
```

The `fit` method returns `History` object, which provides you the performance of the model during training. The `history.history` contains all loss and metric scores for all training epochs, and you can extract this information to evalate your model. I performed some manipulation basically to have a certain format of data (the long version, you might want to refer to tidy data by Hadley Wickham).

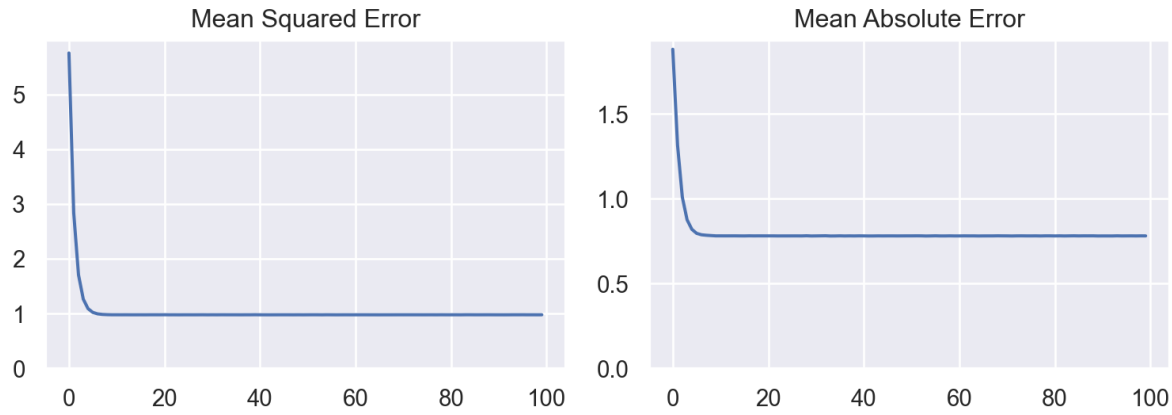
```
data = pd.DataFrame(history.history)
data = data.reset_index()
data = data.rename(columns = {
    "index":"epoch",
    "loss": "Mean Squared Error",
    "mae": "Mean Absolute Error"})
data = pd.melt(
    data,
    id_vars = "epoch",
    value_vars = ["Mean Squared Error", "Mean Absolute Error"],
    var_name = "metric",
    value_name = "value"
```

```
)

data.sort_values(by = "epoch").head()
```

	epoch	metric	value
0	0	Mean Squared Error	5.752564
100	0	Mean Absolute Error	1.881032
1	1	Mean Squared Error	2.836762
101	1	Mean Absolute Error	1.315763
2	2	Mean Squared Error	1.707698

We can then visualise how the model perform throughout the training.

```
(
    so.Plot(data, x = "epoch", y = "value")
    .facet("metric")
    .add(so.Line())
    .share(y = False)
    .limit(y = (0, None))
    .layout(size = (8, 3))
    .label(x = "", y = "")
)
```



2.2.3 Evaluating the model

Finally, we can check the model's performance on the test dataset. The `evaluate` method allows the users to see how the model perform when predicting unseen data. The model seems

to do good in predicting the actual output based on the MSE and MAE.

```
mse, mae = model.evaluate(X_test, y_test, verbose = 0)

print(f"Mean Squared Error : {mse:.2f}")
print(f"Mean Absolute Error: {mae:.2f}")
```

```
Mean Squared Error : 0.90
Mean Absolute Error: 0.77
```

2.3 Dealing with Non-linearity

It is well known that deep learning models are good for high dimensional and complex data. To illustrate the capability of a model in dealing with that type of data, I slightly modified the problem by squaring `x_1`, giving a non-linear property to the data. The final formula is presented below.

$$y = f(x) = 0.2 \times x_0 + 2.8 \times x_1^2 + \epsilon$$

All variables x_0 and x_1 as well as the error term ϵ follow a normal distribution $N(\mu, \sigma^2)$ with $\mu = 0$ and $\sigma^2 = 1$.

```
X = np.random.normal(size = (1000, 2))
y = 0.2 * X[:, 0] + 2.8 * X[:, 1] ** 2 + np.random.normal(size = (1000,))

X_train = X[:800]
y_train = y[:800]
X_test = X[800:]
y_test = y[800:]

X_train.shape, y_train.shape, X_test.shape, y_test.shape,
```

```
((800, 2), (800,), (200, 2), (200,))
```

Using the same approach as above might not give you the best result as you can see in the graphs below. Both MSE and MAE can be significantly higher compared to the values from the previous problem.

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(1, input_shape = (2, )))
model.compile(optimizer = "sgd", loss = "mse", metrics = ["mae"])
history = model.fit(X_train, y_train, epochs = 100, verbose = 0)

data = pd.DataFrame(history.history)
data = data.reset_index()
data = data.rename(columns = {"index":"epoch",
                              "loss": "Mean Squared Error",
                              "mae": "Mean Absolute Error"})
data = pd.melt(data,
               id_vars = "epoch",
               value_vars = ["Mean Squared Error", "Mean Absolute Error"],
               var_name = "metric",
               value_name = "value")

(
    so.Plot(data, x = "epoch", y = "value")
    .facet("metric")
    .add(so.Line())
    .share(y = False)
    .limit(y = (0, None))
    .layout(size = (8, 3))
    .label(x = "", y = "")
)
```

```
mse, mae = model.evaluate(X_test, y_test, verbose = 0)

print(f"Mean Squared Error : {mse:.2f}")
print(f"Mean Absolute Error: {mae:.2f}")
```

```
Mean Squared Error : 14.94
Mean Absolute Error: 2.75
```

2.4 Going Deeper by Using More Layers

As the name suggests, Deep Learning techniques leverages several intermediate representation of the data before finally decide what value to assign for any given input. This supports finding complex patterns that are usually inherent in real world data.

The previous model is modified simply by adding more `Dense` layers and increasing the number of the units. The `activation` function in a model is crucial for capturing non-linearity. The `relu` activation function is a function that gives either a positive value or zero which is suprisingly effective for balancing the trade-offs between finding non-linear pattern and efficient computation. As each subsequent layer can determine the number of parameters required through inferring the number of units from the previous layer, `input_shape` is only defined for the first layer.

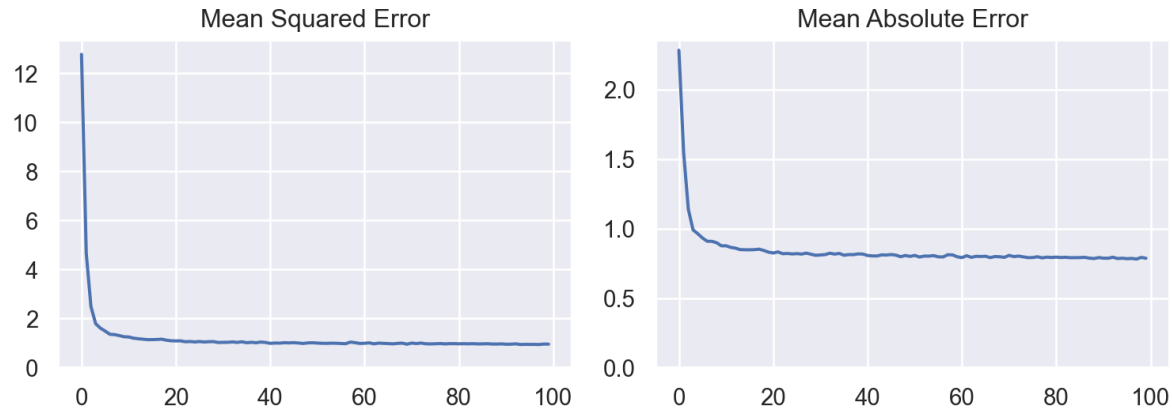
```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(32, activation = "relu", input_shape = (2, )))
model.add(tf.keras.layers.Dense(32, activation = "relu"))
model.add(tf.keras.layers.Dense(1))
model.compile(optimizer = "sgd", loss = "mse", metrics = ["mae"])

model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_2 (Dense)             (None, 32)                96

 dense_3 (Dense)             (None, 32)                1056

 dense_4 (Dense)             (None, 1)                 33
```

```
=================================================================
Total params: 1,185
Trainable params: 1,185
Non-trainable params: 0
_________________________________________________________________
```

As can be seen in the plots below and the values of MSE and MAE, the 'deeper' version of the model could better capture the inherent trend of the dataset leading to more superior model than the previous one.

```
history = model.fit(X_train, y_train, epochs = 100, verbose = 0)

data = pd.DataFrame(history.history)
data = data.reset_index()
data = data.rename(columns = {"index":"epoch",
                              "loss": "Mean Squared Error",
                              "mae": "Mean Absolute Error"})
data = pd.melt(data,
               id_vars = "epoch",
               value_vars = ["Mean Squared Error", "Mean Absolute Error"],
               var_name = "metric",
               value_name = "value")

(
    so.Plot(data, x = "epoch", y = "value")
    .facet("metric")
    .add(so.Line())
    .share(y = False)
    .limit(y = (0, None))
    .layout(size = (8, 3))
    .label(x = "", y = "")
)
```




```
mse, mae = model.evaluate(X_test, y_test, verbose = 0)

print(f"Mean Squared Error : {mse:.2f}")
print(f"Mean Absolute Error: {mae:.2f}")
```

```
Mean Squared Error : 1.04
Mean Absolute Error: 0.82
```

2.5 Conclusion

In this post, I demonstrate how to leverage a small subset of TensorFlow 2 capabilities to deal with artificial datasets. Even though here only includes problems with structured data with well defined problems and boundaries, Deep Learning model in essence allows anyone to do Machine Learning for highly unstructured data such as images and texts.

3 TensorFlow 2 for Classification Problems

Continuing previous post, this demo will show you how to leverage TensorFlow 2 (TF2) for dealing with classification problems. An additional technique to tune hyperparameter (which in this case is the number of epochs) is presented here. Similar to the previous demo, the data for illustration is randomly generated using `numpy` library.

The objective of this demo is to show you the main elements of working with a TF2 model to tackle classification problems.

The setup includes importing important libraries (`tensorflow`, `numpy`, and `matplotlib.pyplot`), freeing memory from old models/layers (if any) and setting the seed for random number generator.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

```
tf.keras.backend.clear_session()
tf.keras.utils.set_random_seed(123)
```

3.1 Generate Random Data

For this use case, I genereated two classes of data based on multivariate normal distribution with specified means and covariance matrices. Both values are determined arbitrarily.

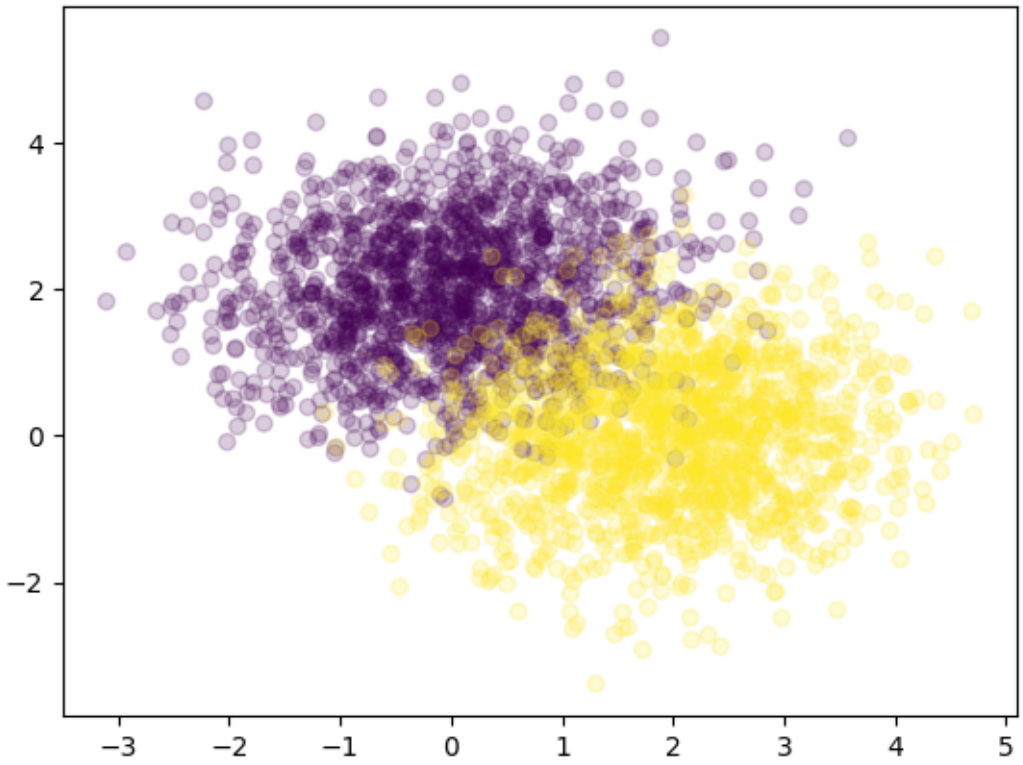
```
# random data generation
SAMPLE_SIZE = 1500

class_1 = np.random.multivariate_normal(
    mean = [0, 2],
    cov  = [[1, 0.1], [0.1, 1]],
    size = SAMPLE_SIZE
)
```

```
class_2 = np.random.multivariate_normal(
    mean = [2, 0],
    cov  = [[1, 0.1], [0.1, 1]],
    size = SAMPLE_SIZE
)

# append both classes
X = np.concatenate([class_1, class_2])
X = X.astype("float32")

y = np.concatenate([np.zeros((SAMPLE_SIZE, 1)), np.ones((SAMPLE_SIZE, 1))])
y = y.astype("int")

X.shape, y.shape
```

```
((3000, 2), (3000, 1))
```

As there are only two variables within the data, making sense of it is easier as we only requires a scatter plot to see how data is dispersed along x and y axes. As you can see from the figure below, there is an area where points from class 1 and class 2 overlap.

```
plt.scatter(X[:, 0], X[:, 1], c = y[:, 0], alpha = .2)
plt.show()
```



3.2 Slice the Data

To help slicing two python variables with the same length (`X` and `y`), I created a vector of data indices where the order is shuffled. This then server as a reference to determine which points belong to which datasets (training, validation, or testing).

I split the data into train and test datasets (80% and 20%), before splitting the train dataset further for hyperparameter tuning into partial train and validation (80% and 20%).

```
# define randomized indices for splitting
indices = np.arange(SAMPLE_SIZE * 2)
np.random.shuffle(indices)

# split data into `train` and `test datasets`
split_locaction = round(SAMPLE_SIZE * .8)

X_train = X[indices[:split_locaction]]
y_train = y[indices[:split_locaction]]
```

```
X_test = X[indices[split_locaction:]]
y_test = y[indices[split_locaction:]]

X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

```
((1200, 2), (1200, 1), (1800, 2), (1800, 1))
```

```
# split train data into `partial` and `validation` for hyperparameter tuning
split_locaction = round(len(X_train) * .8)

partial_X_train = X_train[:split_locaction]
partial_y_train = y_train[:split_locaction]

X_val = X_train[split_locaction:]
y_val = y_train[split_locaction:]

partial_X_train.shape, partial_y_train.shape, X_val.shape, y_val.shape
```

```
((960, 2), (960, 1), (240, 2), (240, 1))
```

3.3 Hyperparameter Optimization

Hyperparameter optimization or tuning can be applied to any parameters controlling the behaviours of the machine learning algorithm which are not learned during training. In doing so, we need to separate the test data and leverage two subsets of training data instead. Otherwise, there might be any leak of information from the 'unseen data' which might alter the result of the trained algorithm giving it the capability to perform better on the test dataset. This opposes the idea of ML model that should be able to do well given unknown input, which, in this case is represented as test dataset.

The hyperparameter to be tuned is the simple one, in this case number of epochs.The process includes training a network with simplifed architecture, then analyses the performance of the network throughout the training. The optimal number of epochs is decided based on how accuracy and loss values moves throughout time.

The actual workflow for creating the model, compiling its optimizer, loss function, and metrics, and fitting it to the data is similar to what you can see from the previous demo. The difference here is that I did not use `model.add` method to put a layer in the model. Instead, I gave a list of several `Dense` layers as an argument when instantiating a `Sequential` model. In addition,

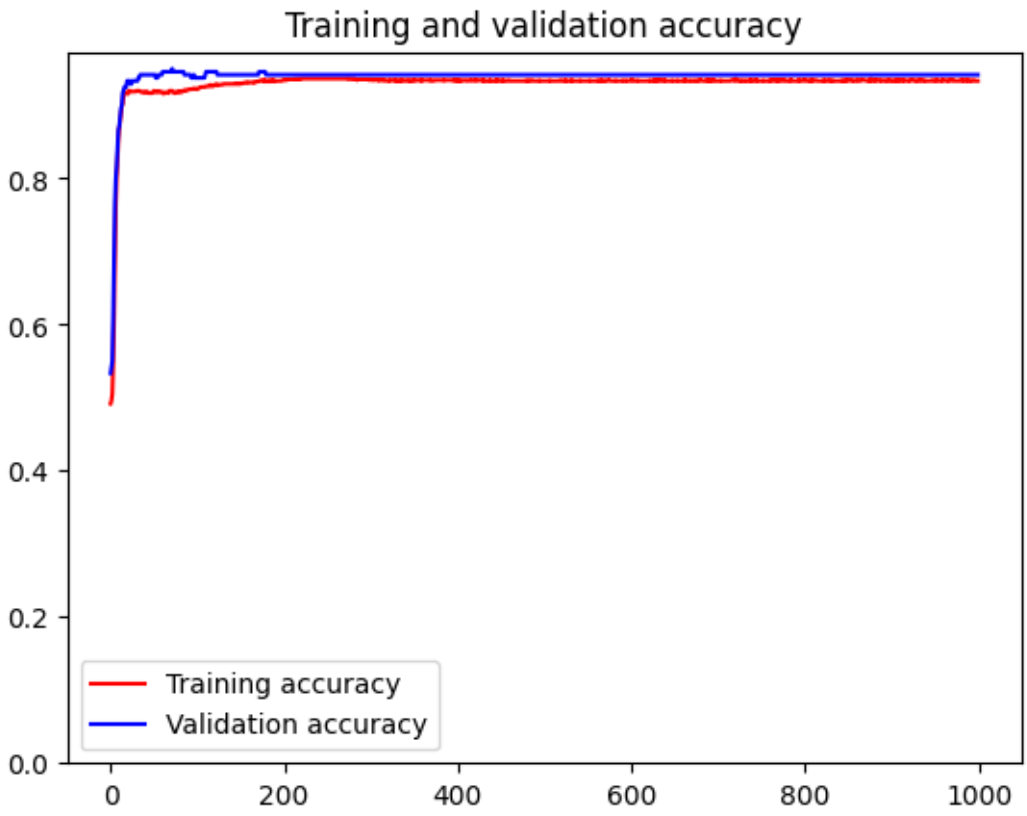
the number of units in each layer is a reduced one (we will increase it when training with full train data). I also set the learning rate for the `SGD` optimizer into 0.005.

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(8, input_shape = (2,), activation = "relu"),
    tf.keras.layers.Dense(8, activation = "relu"),
    tf.keras.layers.Dense(1, activation = "sigmoid")
])

model.compile(optimizer = tf.keras.optimizers.SGD(learning_rate = 0.005),
              loss = "binary_crossentropy",
              metrics = ["accuracy"])

history = model.fit(partial_X_train,
                    partial_y_train,
                    validation_data=(X_val, y_val),
                    epochs = 1000,
                    verbose = 0)
```

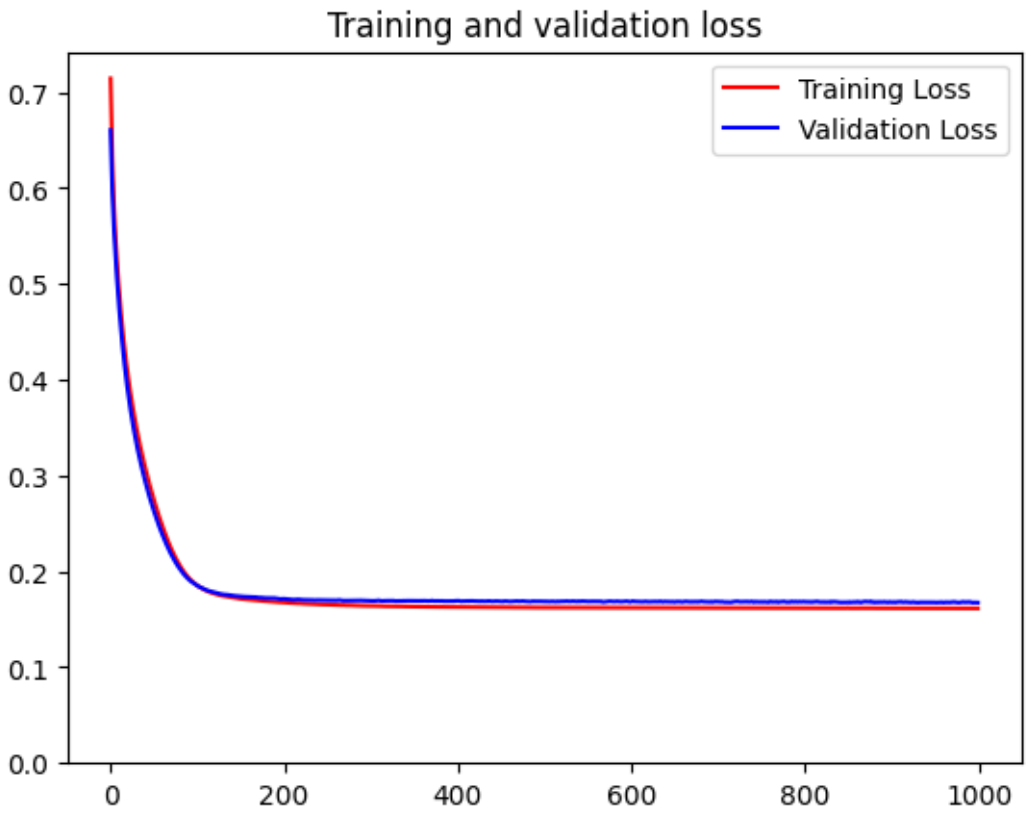
The model is fitted using the `partial_X_train` and `partial_y_train` with a set of validation data. By using validation data, we might observe how the performance of the model throughout training.

Below, we can see the values of training and validation accuracy and loss given a certain training epoch. Because of the values for the validation seems to resemble training values, it can be inferred that the model does not overfit. Overfitting may cause the training accuracy to be significantly higher than validation accuracy and training loss to be significantly lower than validation loss.

```
# Plot the training results
accuracy     = history.history['accuracy']
val_accuracy = history.history['val_accuracy']
epochs       = range(len(accuracy))

plt.plot(epochs, accuracy, 'r', label='Training accuracy')
plt.plot(epochs, val_accuracy, 'b', label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.ylim(ymin=0)
plt.legend()
plt.show()
```




```
# Plot the training results
loss     = history.history['loss']
val_loss = history.history['val_loss']
epochs   = range(len(accuracy))

plt.plot(epochs, loss, 'r', label='Training Loss')
plt.plot(epochs, val_loss, 'b', label='Validation Loss')
plt.title('Training and validation loss')
plt.ylim(ymin=0)
plt.legend()
plt.show()
```



3.4 Fitting with Full Training Data

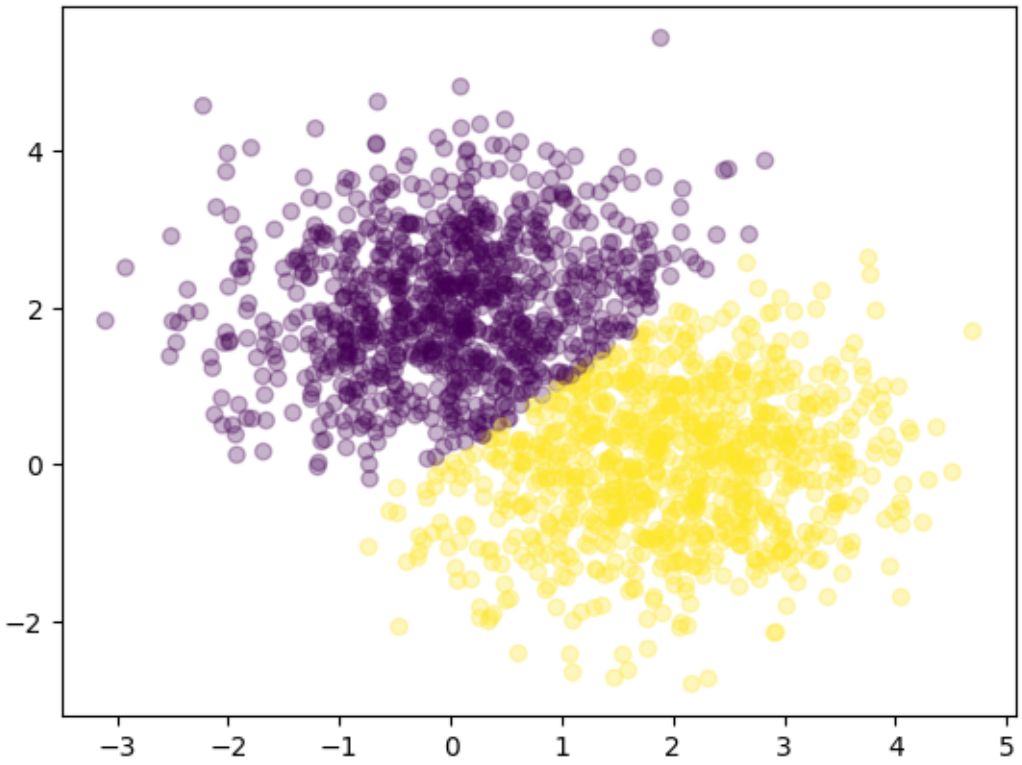
After observing how the simplified model performs, we were able to decide at which epoch we want to stop training our model. In this case, we selected 175 as the subsequent epochs does not give improvement to the model (the loss seemed to stop decreasing). We then could fit our model with full training data and increase the number of units for each `Dense` layer.

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(64, input_shape = (2,), activation = "relu"),
    tf.keras.layers.Dense(64, activation = "relu"),
    tf.keras.layers.Dense(1, activation = "sigmoid")
])

model.compile(optimizer = tf.keras.optimizers.SGD(learning_rate = 0.005),
              loss = "binary_crossentropy",
              metrics = ["accuracy"])
```

```
history = model.fit(X_train, y_train, epochs = 175, verbose = 0)
```

Next, we see how the model classifies each data point from the graph below.

```
y_pred = model.predict(X_test, verbose = 0)

plt.scatter(X_test[:, 0], X_test[:, 1], c = y_pred[:, 0] > .5, alpha = .3)
plt.show()
```



We could also `evaluate` the performance on the test dataset. The model can reach more than 80% accuracy.

```
loss, accuracy = model.evaluate(X_test, y_test, verbose = 0)

print(f"Loss     : {loss:.3f}")
print(f"Accuracy: {accuracy:.3f}")
```

```
Loss    : 0.176
Accuracy: 0.926
```

3.5 Conclusion

In this post, we continue our demonstration of TensorFlow 2 with classification problems. The model successfully achieve a decent accuracy score for this simple case. Additionally, we have touched the concept of hyperparameter tuning which is essential for doing machine learning.

Part II

Deep Learning Case Studies

4 MNIST

```
import numpy as np
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import models, layers, optimizers, backend
```

```
# load dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

# transform image data
train_images = train_images.reshape((60000, 28 * 28)) / 255
test_images = test_images.reshape((10000, 28 * 28)) / 255

train_images.shape, train_labels.shape, test_images.shape, test_labels.shape
```

```
def explore(train_images,
            train_labels,
            test_images,
            test_labels,
            label_count,
            neuron_count,
            learning_rate,
            momentum):

    # define ann architecture
    model = models.Sequential()
    model.add(layers.Dense(neuron_count, activation = "relu", input_shape = (28 * 28,)))
    model.add(layers.Dense(label_count, activation = "softmax"))

    # define optimizer, loss function, and metrics
    optimizer = optimizers.RMSprop(learning_rate = learning_rate, momentum = momentum)
```

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

    # train ann model
    history = model.fit(train_images, train_labels, epochs = 20, batch_size = 64, verbose

    # evaluate ann model
    test_loss, test_acc = model.evaluate(test_images, test_labels, verbose = 0)

    return history, test_loss, test_acc
```

```
# set hyperparameters
learning_rates = np.logspace(-1, -4, 5)
momentums = np.logspace(-1, -4, 5)
neuron_counts = 2 ** np.arange(7, 12)

hyparameters_list = []
for learning_rate in learning_rates:
    for momentum in momentums:
        for neuron_count in neuron_counts:
            hyparameters_list.append({
                "learning_rate": learning_rate,
                "momentum": momentum,
                "neuron_count": neuron_count
            })
```

```
output = []
for hyparameters in hyparameters_list:
    history, test_loss, test_acc = explore(
        train_images,
        train_labels,
        test_images,
        test_labels,
        label_count = 10,
        learning_rate = hyparameters["learning_rate"],
        momentum = hyparameters["momentum"],
        neuron_count = hyparameters["neuron_count"]
    )
```

```
    output.append({
        "history": history,
        "test_loss": test_loss,
        "test_acc": test_acc,
        "hyperparameters": hyparameters
    })
    backend.clear_session()

    print(f"A model is trained with hyperparameters of {hyparameters}")
```

5 IMDB

```
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.datasets import imdb
from tensorflow.keras import models, layers, optimizers, backend
```

```
# load dataset
num_words = 10000

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words = num_word

train_data.shape, train_labels.shape, test_data.shape, test_labels.shape
```

```
# preprocess
X_train = np.zeros(shape = (len(train_data), num_words), dtype = float)
X_test = np.zeros(shape = (len(test_data), num_words), dtype = float)

for i, seq in enumerate(train_data):
    X_train[i, seq] = 1.

for i, seq in enumerate(test_data):
    X_test[i, seq] = 1.

y_train = train_labels.astype(float)
y_test = test_labels.astype(float)

print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
```

```
partial_X_train = X_train[:12500]
partial_y_train = y_train[:12500]
X_val = X_train[12500:]
y_val = y_train[12500:]
```

```
def explore(X_train,
            y_train,
            X_val,
            y_val,
            n_units,
            n_layers,
            activation,
            learning_rate,
            momentum):

    # define ann architecture
    model = models.Sequential()
    for i in range(n_layers):
        model.add(layers.Dense(n_units, activation = activation))
    model.add(layers.Dense(1, activation = "sigmoid"))

    # define optimizer, loss function, and metrics
    optimizer = optimizers.RMSprop(learning_rate = learning_rate, momentum = momentum)


    # train ann model
    model.build(input_shape = (10000,))
    model.compile(optimizer = optimizer, loss = "binary_crossentropy", metrics = ["accurac
    model.fit(X_train, y_train, epochs = 20, batch_size = 64, verbose = 0)

    # evaluate ann model
    val_loss, val_acc = model.evaluate(X_val, y_val, verbose = 0)

    return val_loss, val_acc
```

```
# set hyperparameters
learning_rate_list  = np.logspace(-2, -4, 5)
momentum_list       = np.linspace(0.1, 0.9, 5)
n_unit_list         = [32, 64]
n_hidden_layer_list = [1, 3]
activation_list     = ["relu", "tanh"]

param_list = []
for learning_rate in learning_rate_list:
    for momentum in momentum_list:
        for n_units in n_unit_list:
```

```
            for n_layers in n_hidden_layer_list:
                for activation in activation_list:
                    param_list.append({
                        "learning_rate": learning_rate,
                        "momentum": momentum,
                        "n_units": n_units,
                        "n_layers": n_layers,
                        "activation": activation
                    })
```

```
results = []
for params in param_list:
    val_loss, val_acc = explore(
        partial_X_train,
        partial_y_train,
        X_val,
        y_val,
        n_units = params["n_units"],
        n_hidden_layer = params["n_hidden_layer"],
        activation = params["activation"],
        learning_rate = params["learning_rate"],
        momentum = params["momentum"],
    )

    results.append({"val_loss": val_loss,
                    "val_acc": val_acc,
                    "params": params})

    backend.clear_session()
```

```
# get optimal parameters
val_accuracies = [result["val_acc"] for result in results]
opt_params     = results[np.argmax(val_accuracies)]["params"]

opt_params
```

```
# define ann architecture
model = models.Sequential()
for i in range(opt_params["n_layers"]):
    model.add(layers.Dense(opt_params["n_units"], activation = opt_params["activation"]))
```

```
model.add(layers.Dense(1, activation = "sigmoid"))

# define optimizer, loss function, and metrics
optimizer = optimizers.RMSprop(learning_rate = opt_params["learning_rate"],
                               momentum = opt_params["momentum"])

# train ann model
model.build(input_shape = (10000,))
model.compile(optimizer = optimizer, loss = "binary_crossentropy", metrics = ["accuracy"])

history = model.fit(X_train, y_train, epochs = 20, batch_size = 64, verbose = 0)
```

```
loss = history['loss']

epochs = range(1, len(loss) + 1)

blue_dots = 'bo'
solid_blue_line = 'b'

plt.plot(epochs, loss, solid_blue_line, label = 'Training loss')
plt.title('Training loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

```
accuracy = history['accuracy']

epochs = range(1, len(accuracy) + 1)

blue_dots = 'bo'
solid_blue_line = 'b'

plt.plot(epochs, accuracy, solid_blue_line, label = 'Training accuracy')
plt.title('Training accuracy')
plt.xlabel('Epochs')
plt.ylabel('accuracy')
plt.legend()
```

```
plt.show()
```

```
model.evaluate(X_test, y_test)
```

6 Reuters

```
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.datasets import reuters
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import models, layers, optimizers, backend
```

```
# load dataset
num_words = 10000

(train_data, train_labels,), (test_data, test_labels) = reuters.load_data(num_words = num_

train_data.shape, train_labels.shape, test_data.shape, test_labels.shape
```

```
seq_len = 300 # the avg is 145.54

X_train = [seq[:seq_len] for seq in train_data]
X_train = [np.append([0] * (seq_len - len(seq)), seq) for seq in X_train]
X_train = np.array(X_train).astype(int)

y_train = to_categorical(train_labels)

X_test = [seq[:seq_len] for seq in test_data]
X_test = [np.append([0] * (seq_len - len(seq)), seq) for seq in X_test]
X_test = np.array(X_test).astype(int)

y_test = to_categorical(test_labels)

X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

```
partial_X_train = X_train[:4500]
partial_y_train = y_train[:4500]
X_val = X_train[4500:]
```

```
y_val = y_train[4500:]
```

```
def explore(X_train,
            y_train,
            X_val,
            y_val,
            embedding_dim,
            learning_rate,
            momentum):

    # define ann architecture
    model = models.Sequential()
    model.add(layers.Embedding(num_words, embedding_dim, input_length = seq_len))
    model.add(layers.Dense(64, activation = "relu"))
    model.add(layers.Dense(46, activation = "sigmoid"))

    # define optimizer, loss function, and metrics
    optimizer = optimizers.RMSprop(learning_rate = learning_rate, momentum = momentum)

    # train ann model
    model.compile(optimizer = optimizer, loss = "categorical_crossentropy", metrics = ["ac
    model.fit(X_train, y_train, epochs = 20, batch_size = 64, verbose = 0)

    # evaluate ann model
    val_loss, val_acc = model.evaluate(X_val, y_val, verbose = 0)

    return val_loss, val_acc
```

```
# set hyperparameters
learning_rate_list  = np.logspace(-2, -4, 5)
momentum_list       = np.linspace(0.1, 0.9, 5)
embedding_dim_list  = 2 ** np.arange(3, 7)

param_list = []
for learning_rate in learning_rate_list:
    for momentum in momentum_list:
        for embedding_dim in embedding_dim_list:
            param_list.append({
                "learning_rate": learning_rate,
                "momentum": momentum,
```

```
                "embedding_dim": embedding_dim
            })

results = []
for params in param_list:
    val_loss, val_acc = explore(
        partial_X_train,
        partial_y_train,
        X_val,
        y_val,
        embedding_dim = params["embedding_dim"],
        learning_rate = params["learning_rate"],
        momentum = params["momentum"],
    )

    results.append({"val_loss": val_loss,
                    "val_acc": val_acc,
                    "params": params})

    backend.clear_session()
```

```
# get optimal parameters
val_accuracies = [result["val_acc"] for result in results]
opt_params     = results[np.argmax(val_accuracies)]["params"]

opt_params
```

```
# define ann architecture
model = models.Sequential()
for i in range(opt_params["n_layers"]):
    model.add(layers.Dense(opt_params["n_units"], activation = opt_params["activation"]))
model.add(layers.Dense(1, activation = "sigmoid"))

# define optimizer, loss function, and metrics
optimizer = optimizers.RMSprop(learning_rate = opt_params["learning_rate"],
                               momentum = opt_params["momentum"])

# train ann model
model.build(input_shape = (10000,))
model.compile(optimizer = optimizer, loss = "binary_crossentropy", metrics = ["accuracy"])
```

```
history = model.fit(X_train, y_train, epochs = 20, batch_size = 64, verbose = 0)
```

```
loss = history['loss']

epochs = range(1, len(loss) + 1)

blue_dots = 'bo'
solid_blue_line = 'b'

plt.plot(epochs, loss, solid_blue_line, label = 'Training loss')
plt.title('Training loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

```
accuracy = history['accuracy']

epochs = range(1, len(accuracy) + 1)

blue_dots = 'bo'
solid_blue_line = 'b'

plt.plot(epochs, accuracy, solid_blue_line, label = 'Training accuracy')
plt.title('Training accuracy')
plt.xlabel('Epochs')
plt.ylabel('accuracy')
plt.legend()

plt.show()
```

```
model.evaluate(X_test, y_test)
```

7 Boston Housing

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.base import clone
from tensorflow.keras.datasets import boston_housing
from tensorflow.keras import models, layers, backend
```

```
# load dataset
(X_train, y_train), (X_test, y_test) = boston_housing.load_data()

X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

```
# rescale and shift data based on training set
transform_mean = np.mean(X_train)
transform_std  = np.std(X_train, ddof = 1)

X_train -= transform_mean
X_train /= transform_std

X_test -= transform_mean
X_test /= transform_std
```

```
model_nn = models.Sequential()
model_nn.add(layers.Dense(128, activation = "relu", input_shape = (13,)))
model_nn.add(layers.Dense(128, activation = "relu"))
model_nn.add(layers.Dense(128, activation = "relu"))
model_nn.add(layers.Dense(1))
model_nn.compile(optimizer = "adam", loss = "mse", metrics = ["mae", "mse"])

initial_weight_nn = model_nn.get_weights()
```

```
model_nn_reg = models.Sequential()
model_nn_reg.add(layers.Dense(64, activation = "relu", input_shape = (13,)))
model_nn_reg.add(layers.Dropout(0.3))
model_nn_reg.add(layers.Dense(64, activation = "relu", kernel_regularizer='l1_l2'))
model_nn_reg.add(layers.Dropout(0.3))
model_nn_reg.add(layers.Dense(64, activation = "relu", kernel_regularizer='l1_l2'))
model_nn_reg.add(layers.Dropout(0.3))
model_nn_reg.add(layers.Dense(1))

model_nn_reg.compile(optimizer = "adam", loss = "mse", metrics = ["mae", "mse"])
initial_weight_nn_reg = model_nn_reg.get_weights()
```

```
lm_base = LinearRegression()
```

```
indices = np.arange(len(X_train))
np.random.seed(123)
np.random.shuffle(indices)

k_fold = 5
sample_size = np.ceil(len(X_train) / k_fold).astype(int)
```

```
mse_nn, mse_nn_reg, mse_lm = [], [], []
mae_nn, mae_nn_reg, mae_lm = [], [], []

for i in range(k_fold):
    # configure model with exact parameters
    model_lm = clone(lm_base)
    model_nn.set_weights(initial_weight_nn)
    model_nn_reg.set_weights(initial_weight_nn_reg)

    # split into partial_train and validation
    id_start, id_end = i * sample_size, (i+1) * sample_size

    mask_train = np.concatenate((indices[:id_start], indices[id_end:]))
    mask_val   = indices[id_start:id_end]

    X_val = X_train[mask_val]
    y_val = y_train[mask_val]
    partial_X_train = X_train[mask_train]
    partial_y_train = y_train[mask_train]
```

```
    # fit and predict
    model_lm.fit(partial_X_train, partial_y_train)
    model_nn.fit(partial_X_train, partial_y_train, epochs = 500, verbose = 0)
    model_nn_reg.fit(partial_X_train, partial_y_train, epochs = 500, verbose = 0)

    y_pred_lm = model_lm.predict(X_val)
    y_pred_nn = model_nn.predict(X_val, verbose = 0)
    y_pred_nn_reg = model_nn_reg.predict(X_val, verbose = 0)

    # save results
    mse_nn.append(mean_squared_error(y_val, y_pred_nn))
    mse_nn_reg.append(mean_squared_error(y_val, y_pred_nn_reg))
    mse_lm.append(mean_squared_error(y_val, y_pred_lm))

    mae_nn.append(mean_absolute_error(y_val, y_pred_nn))
    mae_nn_reg.append(mean_absolute_error(y_val, y_pred_nn_reg))
    mae_lm.append(mean_absolute_error(y_val, y_pred_lm))
```

```
print(f"Avg MSE of Neral Network : {np.mean(mse_nn):.2f}")
print(f"Avg MSE of NN Regulaized : {np.mean(mse_nn_reg):.2f}")
print(f"Avg MSE of Linear Model  : {np.mean(mse_lm):.2f}")
```

```
print(f"Avg MAE of Neral Network : {np.mean(mae_nn):.2f}")
print(f"Avg MAE of NN Regulaized : {np.mean(mae_nn_reg):.2f}")
print(f"Avg MAE of Linear Model  : {np.mean(mae_lm):.2f}")
```

Part III

Preparing Data for Deep Learning

8 Handling Text Data

9 Handling Image Data

10 Handling Time Series Data

Summary

In summary, this book has no content whatsoever.

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.