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.

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

1 TensorFlow for Regression Problems

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

$$y = 2x_0 + 5.2x_1 + \epsilon$$

```
X = np.random.normal(size = (1000, 2))
y = 2 * X[:, 0] + 5.2 * X[:, 1] + np.random.normal(size = (1000,))

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

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(1, input_shape = (2, )))
model.compile(optimizer = "sgd", loss = "mse", metrics = ["mse", "mae"])
model.fit(X_train, y_train, epochs = 300, verbose = 0)
model.evaluate(X_test, y_test)
```

```
7/7 [==============================] - 0s 2ms/step - loss: 1.1128 - mse: 1.1128 - mae: 0.846
```

```
[1.1127723455429077, 1.1127723455429077, 0.8464701771736145]
```

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(64, input_shape = (2, ), activation = "relu"))
model.add(tf.keras.layers.Dense(64, activation = "relu"))
model.add(tf.keras.layers.Dense(1))
model.compile(optimizer = "sgd", loss = "mse", metrics = ["mse", "mae"])
model.fit(X_train, y_train, epochs = 300, verbose = 0)
model.evaluate(X_test, y_test)
```

```
7/7 [==============================] - 0s 3ms/step - loss: 1.1905 - mse: 1.1905 - mae: 0.871
```

```
[1.1904633045196533, 1.1904633045196533, 0.871860146522522]
```

$$y = 2x_0 + 5.2(x_1)^2 + \epsilon$$

```
X = np.random.normal(size = (1000, 2))
y = 2 * X[:, 0] + 5.2 * X[:, 1] ** 2 + np.random.normal(size = (1000,))

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

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(1, input_shape = (2, )))
model.compile(optimizer = "sgd", loss = "mse", metrics = ["mse", "mae"])
model.fit(X_train, y_train, epochs = 300, verbose = 0)
model.evaluate(X_test, y_test)
```

```
7/7 [==============================] - 0s 2ms/step - loss: 44.0046 - mse: 44.0046 - mae: 4.78
```

```
[44.004600524902344, 44.004600524902344, 4.780708312988281]
```

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(64, input_shape = (2, ), activation = "relu"))
model.add(tf.keras.layers.Dense(64, activation = "relu"))
model.add(tf.keras.layers.Dense(1))
model.compile(optimizer = "sgd", loss = "mse", metrics = ["mse", "mae"])
model.fit(X_train, y_train, epochs = 300, verbose = 0)
model.evaluate(X_test, y_test)
```

```
7/7 [==============================] - 0s 4ms/step - loss: 1.1275 - mse: 1.1275 - mae: 0.820
```

```
[1.1274707317352295, 1.1274707317352295, 0.8208972215652466]
```

2 TensorFlow for Classification Problems

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

```
SAMPLE_NUMBER = 1000

negatives = np.random.multivariate_normal(
    mean = [0, 2],
    cov  = [[1, 0.5], [0.5, 1]],
    size = SAMPLE_NUMBER
)

positives = np.random.multivariate_normal(
    mean = [2.5, 0],
    cov  = [[1, 0.5], [0.5, 1]],
    size = SAMPLE_NUMBER
)
```

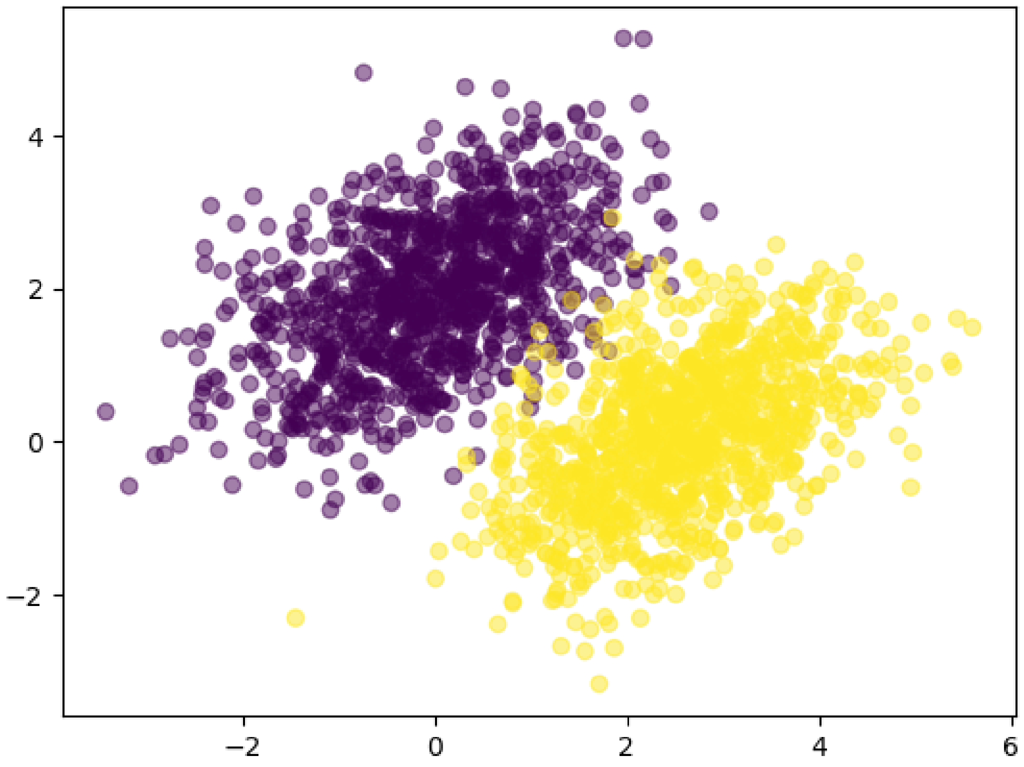
```
Xs = np.vstack((negatives, positives)).astype(np.float32)
ys = np.vstack((np.zeros((SAMPLE_NUMBER, 1), dtype = np.float32),
                np.ones(( SAMPLE_NUMBER, 1), dtype = np.float32)))

Xs.shape, ys.shape
```

```
((2000, 2), (2000, 1))
```

```
plt.scatter(Xs[:, 0], Xs[:, 1], c = ys[:, 0], alpha = .5)
```

```
<matplotlib.collections.PathCollection at 0x24962914340>
```



```
# build the linear classifier
input_dim = 2
output_dim = 1
learning_rate = 0.1
```

```
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(64, activation = "relu"),
    tf.keras.layers.Dense(1, activation = "sigmoid")
])
model.compile(optimizer = "adam", loss = "binary_crossentropy", metrics = ["acc"])
```

```
model.fit(Xs, ys, epochs = 500, verbose = 0)
```

```
<keras.callbacks.History at 0x24963c8fa30>
```

```
y_pred = model.predict(Xs)
```

```
63/63 [==============================] - 0s 1ms/step
```

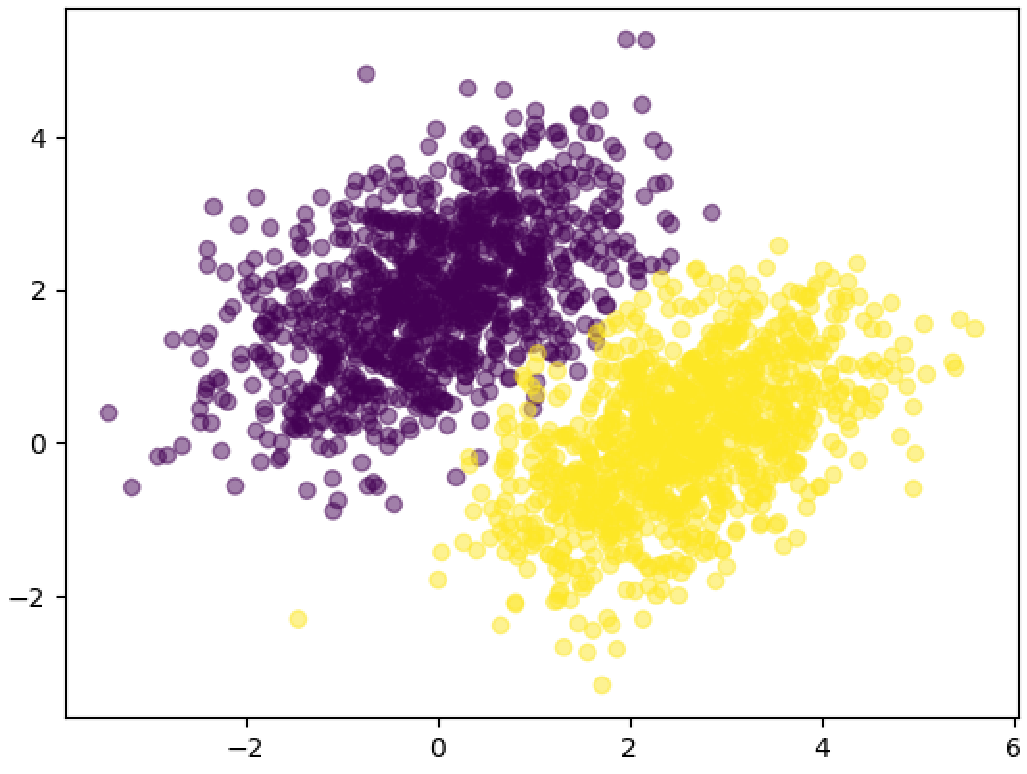
```
model.evaluate(Xs, ys)
```

```
63/63 [==============================] - 0s 2ms/step - loss: 0.0146 - acc: 0.9945
```

```
[0.014589206315577703, 0.9944999814033508]
```

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

```
<matplotlib.collections.PathCollection at 0x248e66194c0>
```



Part II

Deep Learning Case Studies

3 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}")
```

4 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"]))
```

```python
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)
```

```python
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()
```

```python
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)
```

5 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)
```

6 Boston Housing

```python
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
```

```python
# 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
```

```python
# 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
```

```python
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

7 Handling Text Data

8 Handling Image Data

9 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.