Lab Internals - Deep Learning

Highlighted Code to be filled in

1. **Training Deep Neural Networks**

```
import pandas as pd
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from sklearn.model_selection import train_test_split
from pathlib import Path
from time import strftime
import matplotlib.pyplot as plt

# Loading CIFAR-10 dataset
cifar10 = cifar10.load_data()
(X_train_full, y_train_full), (X_test, y_test) = cifar10
print("Shape of the training dataset:", X_train_full.shape)
print("Shape of the test dataset:", X_test.shape)

# Class names for CIFAR-10
class_names = ["airplanes", "cars", "birds", "cats", "deer", "dogs", "frogs",
               "horses", "ships", "trucks"]

# Splitting dataset into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(
   X_train_full, y_train_full, test_size=5000, random_state=42, stratify=y_train_full)

# Setting random seed for reproducibility
tf.random.set_seed(42)

# Defining the model
model = tf.keras.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=[32, 32, 3]))
for _ in range(20):
   model.add(tf.keras.layers.Dense(100, activation="swish", kernel_initializer="he_normal"))

# Adding the output layer
model.add(tf.keras.layers.Dense(10, activation="softmax"))

# Compiling the model
optimizer = tf.keras.optimizers.Nadam(learning_rate=5e-5)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
metrics=["accuracy"])
```

```
# Setting up log directory for TensorBoard
logdir = "./logs"
def get_run_logdir(logdir="logs"):
    """
    Returns directory path to store all logs into. For convenience, log files for each training-run
    gets stored in a folder named as timestamp. By default, it considers the root log folder name
    as "logs". For otherwise, that needs to be passed in the parameter.
    """
    return Path(logdir) / strftime("%Y_%m_%d_%H_%M_%S")

# Gets the log path for current training run
run_logdir = get_run_logdir()

# Callbacks for model training
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    "./model_weights/my_cifar10_model.keras", save_best_only=True)
early_stopping_callback = tf.keras.callbacks.EarlyStopping(patience=20,
restore_best_weights=True)
tensorboard_callback = tf.keras.callbacks.TensorBoard(run_logdir)

# Preparing list of callbacks to be passed into training process
callbacks = [early_stopping_callback, model_checkpoint_callback, tensorboard_callback]

# Loading TensorBoard extension
%load_ext tensorboard
%tensorboard --logdir logdir

# Training the model
model.fit(X_train, y_train, epochs=100, validation_data=(X_val, y_val), callbacks=callbacks)

# Evaluating the model
model.evaluate(X_val, y_val)

# Clearing the session for the next model
tf.keras.backend.clear_session()

# Setting random seed again for reproducibility
tf.random.set_seed(42)
```

```
# Defining the model with Batch Normalization
model = tf.keras.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=[32, 32, 3]))
for _ in range(20):
    model.add(tf.keras.layers.Dense(100, kernel_initializer="he_normal"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Activation("swish"))

# Adding the output layer
model.add(tf.keras.layers.Dense(10, activation="softmax"))

# Compiling the model with a different learning rate
optimizer = tf.keras.optimizers.Nadam(learning_rate=5e-4)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
metrics=["accuracy"])

# Setting up log directory for the new model
run_logdir = get_run_logdir()

# Callbacks for the new model
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    "./model_weights/my_cifar10_bn_model.keras", save_best_only=True)
early_stopping_callback = tf.keras.callbacks.EarlyStopping(patience=20,
restore_best_weights=True)
tensorboard_callback = tf.keras.callbacks.TensorBoard(run_logdir)
callbacks = [early_stopping_callback, model_checkpoint_callback, tensorboard_callback]

# Training the new model
model.fit(X_train, y_train, epochs=100, validation_data=(X_val, y_val), callbacks=callbacks)

# Evaluating the new model
model.evaluate(X_val, y_val)
```

2. Training Deep Neural Networks with Pretrained Layers

```
# Imports required packages
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist
from sklearn.model_selection import train_test_split

# Loads fashion MNIST dataset
fashion = fashion_mnist.load_data()

# Considering dataset is organized in tuple, items are referenced as follows
(X_train_full, y_train_full), (X_test, y_test) = fashion

# Checks the shape of the datasets
print("Train dataset shape:", X_train_full.shape, "\nTest dataset shape:", X_test.shape)

# Each training and test example is assigned to one of the following labels
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal",
               "Shirt", "Sneaker", "Bag", "Ankle boot"]

# Normalizes the data between 0 and 1 for effective neural network model training
X_train_full, X_test = X_train_full / 255., X_test / 255.

# Splits train dataset further to separate 5000 instances to be used as validation set
X_train, X_val, y_train, y_val = train_test_split(
    X_train_full, y_train_full, test_size=5000, random_state=42, stratify=y_train_full
)

# Stores class IDs of the target items
# "Pullover" and "T-shirt/top" are considered as positive and negative class, respectively
pos_class_id = class_names.index("Pullover")
neg_class_id = class_names.index("T-shirt/top")

def split_dataset(X, y):
    """
    Splits the dataset having all items into a pair of tuples - one for dataset for 8-class
classification task
    and another for dataset for the remaining 2-class classification task.
    """
    y_for_B = (y == pos_class_id) | (y == neg_class_id)
    y_A = y[~y_for_B]
    y_B = (y[y_for_B] == pos_class_id).astype(np.float32)
    old_class_ids = list(set(range(10)) - set([neg_class_id, pos_class_id]))
```

```python
    for old_class_id, new_class_id in zip(old_class_ids, range(8)):
        y_A[y_A == old_class_id] = new_class_id  # Reorder class IDs for A
    return (X[~y_for_B], y_A), (X[y_for_B], y_B)

# Splits train, validation, and test data into respective datasets for classification tasks A and B
(X_train_A, y_train_A), (X_train_B, y_train_B) = split_dataset(X_train, y_train)
(X_val_A, y_val_A), (X_val_B, y_val_B) = split_dataset(X_val, y_val)
(X_test_A, y_test_A), (X_test_B, y_test_B) = split_dataset(X_test, y_test)

# Considers only 200 instances for training for classification task B
X_train_B = X_train_B[:200]
y_train_B = y_train_B[:200]

# Creates a dense neural network for classification task A
tf.random.set_seed(42)
model_A = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(100, activation="relu", kernel_initializer="he_normal"),
    tf.keras.layers.Dense(100, activation="relu", kernel_initializer="he_normal"),
    tf.keras.layers.Dense(100, activation="relu", kernel_initializer="he_normal"),
    tf.keras.layers.Dense(8, activation="softmax")
])

model_A.compile(loss="sparse_categorical_crossentropy",
                optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
                metrics=["accuracy"])

# Fits the model
history = model_A.fit(X_train_A, y_train_A, epochs=20, validation_data=(X_val_A, y_val_A))

# Saves the model to be used later for transfer learning
model_A.save("./models/my_fashion_mnist_model_A.keras")

# Creates a dense neural network for classification task B
tf.random.set_seed(42)
model_B = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(100, activation="relu", kernel_initializer="he_normal"),
    tf.keras.layers.Dense(100, activation="relu", kernel_initializer="he_normal"),
    tf.keras.layers.Dense(100, activation="relu", kernel_initializer="he_normal"),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
```

```
model_B.compile(loss="binary_crossentropy",
            optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
            metrics=["accuracy"])

# Fits the model
history = model_B.fit(X_train_B, y_train_B, epochs=20, validation_data=(X_val_B, y_val_B))

# Evaluates the model on test dataset
model_B.evaluate(X_test_B, y_test_B)

# Loads the model trained for classification task A
model_A = tf.keras.models.load_model("./models/my_fashion_mnist_model_A.keras")

# Creates a new model copying all layers except for the output layer from model A
model_B_on_A = tf.keras.Sequential(model_A.layers[:-1])

# Adds a one-node output layer to the new model
model_B_on_A.add(tf.keras.layers.Dense(1, activation="sigmoid"))

tf.random.set_seed(42)

# Clones the network architecture of model A into a new model
model_A_clone = tf.keras.models.clone_model(model_A)

# Copies learned weights of model A into the new cloned model
model_A_clone.set_weights(model_A.get_weights())

# Creates model_B_on_A again
model_B_on_A = tf.keras.Sequential(model_A_clone.layers[:-1])
model_B_on_A.add(tf.keras.layers.Dense(1, activation="sigmoid"))

# Freezes all hidden layers before training
for layer in model_B_on_A.layers[:-1]:
   layer.trainable = False

optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer, metrics=["accuracy"])

# Fits the model over a shorter iteration to train only the output layer
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4, validation_data=(X_val_B,
y_val_B))
```

```
# Allows hidden layers trainable before further training
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer, metrics=["accuracy"])

# Fits the full model
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16, validation_data=(X_val_B,
y_val_B))

# Evaluates the model B against the test dataset
model_B_on_A.evaluate(X_test_B, y_test_B)
```

3. Convolutional Neural Networks

```
# --------------------------- Imports Required Packages ---------------------------

import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist

# --------------------------- Loading MNIST Dataset ---------------------------

# NOTE: Downloading for the first time may take few minutes to complete
mnist = tf.keras.datasets.mnist.load_data()

# Considering dataset is organized in tuple, items are referenced as follows
(X_train_full, y_train_full), (X_test, y_test) = mnist

# Checks the shape of the datasets
print("Full training set shape:", X_train_full.shape)
print("Test set shape:", X_test.shape)
# Full training set shape: (60000, 28, 28)
# Test set shape: (10000, 28, 28)

# --------------------------- Normalizing and Splitting Data ---------------------------

# Normalizes the data between 0 and 1 for effective neural network model training
X_train_full = X_train_full / 255.
X_test = X_test / 255.

# Splits train dataset further to separate 5000 instances to be used as validation set
X_train, X_val = X_train_full[:-5000], X_train_full[-5000:]
y_train, y_val = y_train_full[:-5000], y_train_full[-5000:]

# To match the input shape of the CNN model, a channel dimension gets added to each dataset
X_train = X_train[..., np.newaxis]
X_val = X_val[..., np.newaxis]
X_test = X_test[..., np.newaxis]

# Checks for the updated shape
X_train.shape
# (55000, 28, 28, 1)
```

```
# -------------------------- Creating CNN Model --------------------------

# Creates CNN model by having convoluted, pooling, dropout and dense layers in the specified
order for this experiment.
# Each convoluted layer is further initialized with specific kernel size, padding, activation and
initialization.
tf.random.set_seed(42)

model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="relu",
kernel_initializer="he_normal"),
    tf.keras.layers.Conv2D(64, kernel_size=3, padding="same", activation="relu",
kernel_initializer="he_normal"),
    tf.keras.layers.MaxPool2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Dense(128, activation="relu", kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation="softmax")
])

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

# -------------------------- Training the Model --------------------------

# Fits the model.
model.fit(X_train, y_train, epochs=10, validation_data=(X_val, y_val))

# -------------------------- Saving the Model --------------------------

# Saves the trained model for later reference
# NOTE: Make sure the folder "models" exists under the current working directory
model.save("./models/my_mnist_cnn_model.keras")

# -------------------------- Evaluating the Model --------------------------

# Evaluates the model on test dataset
model.evaluate(X_test, y_test)
```

4. Convolutional Neural Networks using Pretrained Model

```
# -------------------------- Imports Required Packages --------------------------

import numpy as np
import tensorflow as tf
import tensorflow_datasets as tfds    # THIS MIGHT NEED TO BE INSTALLED SEPARATELY
import matplotlib.pyplot as plt

# -------------------------- Loading TF-Flowers Dataset --------------------------

tf_flowers, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)

# Extract useful information about the dataset
dataset_size = info.splits["train"].num_examples    # Number of instances
class_names = info.features["label"].names          # Name of the flowers
n_classes = info.features["label"].num_classes      # Count of types of flowers

# Prints the size of the dataset
print(dataset_size)
# 3670

# Prints the types of flower the dataset has images of
print(class_names)
# ['dandelion', 'daisy', 'tulips', 'sunflowers', 'roses']

# Prints the count of flower types
print(n_classes)

# -------------------------- Splitting Dataset --------------------------

# As this dataset does not provide a separate test dataset, training set gets further split
# into test [first 10%], validation [next 15%], and train [remaining 75%] dataset.
test_set_raw, val_set_raw, train_set_raw = tfds.load(
    "tf_flowers",
    split=["train[:10%]", "train[10%:25%]", "train[25%:]"],
    as_supervised=True
)

# -------------------------- Visualizing Sample Images --------------------------

# Plots any 9 flowers to check how they look
plt.figure(figsize=(12, 10))
index = 0
```

for image, label in val_set_raw.take(9):
    index += 1
    plt.subplot(3, 3, index)
    plt.imshow(image)
    plt.title(f"Class: {class_names[label]}")
    plt.axis("off")
plt.show()

-------------------------- Preprocessing Data --------------------------

tf.random.set_seed(42)
batch_size = 32

# Configures a preprocessing layer for image resizing
preprocess = tf.keras.Sequential([
    tf.keras.layers.Resizing(height=224, width=224, crop_to_aspect_ratio=True), # To resize each image
    tf.keras.layers.Lambda(tf.keras.applications.xception.preprocess_input)   # Preprocess for Xception model
])

# Image instances in different datasets get passed through the preprocessing layer to get resized.
train_set = train_set_raw.map(lambda X, y: (preprocess(X), y))
train_set = train_set.shuffle(1000, seed=42).batch(batch_size).prefetch(1) # Shuffle only train dataset for effective batch processing
val_set = val_set_raw.map(lambda X, y: (preprocess(X), y)).batch(batch_size)
test_set = test_set_raw.map(lambda X, y: (preprocess(X), y)).batch(batch_size)

-------------------------- Defining the Model --------------------------

# NOTE THAT DOWNLOADING XCEPTION WEIGHTS (PRETRAINED ON IMAGENET) MAY TAKE SEVERAL MINUTES TO COMPLETE
tf.random.set_seed(42)

# Loads Xception pretrained model without top layers [i.e., global average pooling and dense output layer]
base_model = tf.keras.applications.xception.Xception(weights="imagenet", include_top=False)

# Adds a global average pooling layer at the output of the base model
avg = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)

# Adds a dense output layer
output = tf.keras.layers.Dense(n_classes, activation="softmax")(avg)

```
# Assembles the model with layers created in the above steps
model = tf.keras.Model(inputs=base_model.input, outputs=output)

# -------------------------- Freezing Base Model Layers --------------------------

for layer in base_model.layers:
    layer.trainable = False

optimizer = tf.keras.optimizers.SGD(learning_rate=0.1, momentum=0.9)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
metrics=["accuracy"])

# -------------------------- Training the Model --------------------------

# NOTE THAT FOLLOWING MODEL TRAINING MAY TAKE SEVERAL MINUTES TO
COMPLETE IF RUN ON CPU
history = model.fit(train_set, validation_data=val_set, epochs=3)

# -------------------------- Saving the Model --------------------------

# [OPTIONAL], Additionally, the above model with non-trainable Xception weights can be saved
for later reference.
# NOTE: Make sure the folder "models" exists under the current working directory
model.save("./models/my_tf-flowers_model_on_non-trainable_xception_weights.keras")

# -------------------------- Listing Base Model Layers --------------------------

# Lists base model's layers
for indices in zip(range(33), range(33, 66), range(66, 99), range(99, 132)):
    for idx in indices:
        print(f"{idx:3}: {base_model.layers[idx].name:22}", end="")
    print()

# -------------------------- Fine-Tuning the Model --------------------------

for layer in base_model.layers[56:]:
    layer.trainable = True

optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
metrics=["accuracy"])

history = model.fit(train_set, validation_data=val_set, epochs=10)
```

```
# --------------------------- Evaluating the Model ---------------------------

# After the model training, the validation accuracy was observed to be 90.56%.
# Now, model's performance on test dataset is evaluated below.
model.evaluate(test_set)
```

5. Sequence Processing using Univariate TimeSeries

```
# -------------------------- Imports Required Packages --------------------------

import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt

# -------------------------- Loading and Preprocessing Data --------------------------

# Load dataset
ridership = pd.read_csv("./../Data/CTA-Ridership-Daily_Boarding_Totals_20240829.csv",
parse_dates=["service_date"])
display(ridership)

# Setting date column as index
ridership = ridership.sort_values("service_date").set_index("service_date")
display(ridership)

# Drops the calculated column "total_rides" as this is just element-wise addition from columns
"bus" and "rail_boardings".
ridership = ridership.drop("total_rides", axis=1)

# Removes duplicate observations, if any
ridership = ridership.drop_duplicates()
ridership.shape
# (8521, 3)

# -------------------------- Visualizing Data --------------------------

# Looks at the first few months of 2019
ridership["2019-03":"2019-05"].plot(grid=True, marker=".", figsize=(8, 3.5))
plt.show()

# Creates a 7-day differencing time-series
diff_7 = ridership[["bus", "rail_boardings"]].diff(7)["2019-03":"2019-05"]

# Prepares a figure with two plots - one for the overlayed time-series and another for the
differencing time-series
fig, axs = plt.subplots(2, 1, sharex=True, figsize=(8, 5))

# Plots the original time-series
ridership.plot(ax=axs[0], legend=False, marker=".")
ridership.shift(7).plot(ax=axs[0], grid=True, legend=False, linestyle=":")
```

```
# Plots the differencing time-series
diff_7.plot(ax=axs[1], grid=True, marker=".")

# [OPTIONAL] Sets y-axis limit of the first plot
axs[0].set_ylim([170_000, 900_000])

plt.show()

# --------------------------- Measuring Forecasting Performance ---------------------------

# Measures the performance of naive forecasting over metric Mean Absolute Error (MAE)
diff_7.abs().mean()
# bus              43915.608696
# rail_boardings    42143.271739

# The same performance is also expressed in Mean Absolute Percentage Error (MAPE)
(diff_7 / ridership[["bus", "rail_boardings"]]["2019-03":"2019-05"]).abs().mean()

# --------------------------- Preparing Train, Validation, and Test Sets ---------------------------

# Splits the time-series into training, validation, and testing periods
rail_train = ridership["rail_boardings"]["2016-01":"2018-12"] / 1e6  # 3 years
rail_val = ridership["rail_boardings"]["2019-01":"2019-05"] / 1e6    # 5 months
rail_test = ridership["rail_boardings"]["2019-06":] / 1e6            # remaining period from 2019-06

# Prepares TensorFlow specific datasets
seq_length = 56    # represents sequence of past 8 weeks (56 days) of ridership data

tf.random.set_seed(42)

rail_train_ds = tf.keras.utils.timeseries_dataset_from_array(
   rail_train.to_numpy(),
   targets=rail_train[seq_length:],
   sequence_length=seq_length,
   batch_size=32,
   shuffle=True,
   seed=42
)

rail_val_ds = tf.keras.utils.timeseries_dataset_from_array(
   rail_val.to_numpy(),
   targets=rail_val[seq_length:],
   sequence_length=seq_length,
```

```
    batch_size=32,
    shuffle=False
)

# -------------------------- Building and Training Linear Model --------------------------

tf.random.set_seed(42)

# Creates a linear model
model = tf.keras.Sequential([tf.keras.layers.Dense(1, input_shape=[seq_length])])

# Early stopping callback
early_stopping_callback = tf.keras.callbacks.EarlyStopping(
    monitor="val_mae", patience=50, restore_best_weights=True
)

# Compile model
model.compile(
    loss=tf.keras.losses.Huber(),
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.02, momentum=0.9),
    metrics=["mae"]
)

# Train the model
history = model.fit(rail_train_ds, validation_data=rail_val_ds, epochs=500,
callbacks=[early_stopping_callback])

# Evaluate the model
val_loss, val_mae = model.evaluate(rail_val_ds)
print("Validation MAE of the Linear Model:", val_mae * 1e6)

# -------------------------- Building and Training Simple RNN --------------------------

tf.keras.backend.clear_session()
tf.random.set_seed(42)

univar_simple_rnn = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 1]),
    tf.keras.layers.Dense(1)
])

early_stopping_callback = tf.keras.callbacks.EarlyStopping(
    monitor="val_mae", patience=50, restore_best_weights=True
)
```

```
univar_simple_rnn.compile(
    loss="huber",
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.05, momentum=0.9),
    metrics=["mae"]
)

history = univar_simple_rnn.fit(
    rail_train_ds, validation_data=rail_val_ds, epochs=500, callbacks=[early_stopping_callback]
)

val_loss, val_mae = univar_simple_rnn.evaluate(rail_val_ds)
print("Validation MAE of the Simple RNN:", val_mae * 1e6)

# -------------------------- Building and Training Deep RNN --------------------------

tf.keras.backend.clear_session()
tf.random.set_seed(42)

univar_deep_rnn = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 1]),
    tf.keras.layers.SimpleRNN(32, return_sequences=True),
    tf.keras.layers.SimpleRNN(32),
    tf.keras.layers.Dense(1)
])

early_stopping_callback = tf.keras.callbacks.EarlyStopping(
    monitor="val_mae", patience=50, restore_best_weights=True
)

univar_deep_rnn.compile(
    loss="huber",
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9),
    metrics=["mae"]
)

history = univar_deep_rnn.fit(
    rail_train_ds, validation_data=rail_val_ds, epochs=500, callbacks=[early_stopping_callback]
)

val_loss, val_mae = univar_deep_rnn.evaluate(rail_val_ds)
print("Validation MAE of the Deep RNN:", val_mae * 1e6)
```

6. **Sequence Processing using Multivariate TimeSeries**

```
# Prepares dataset with multiple features as input for modeling
# Ridership values are once again scaled down by a factor of one million,
# to ensure the values are near the 0–1 range

ridership_multivar = ridership[["bus", "rail_boardings"]] / 1e6
ridership_multivar["next_day_type"] = ridership["day_type"].shift(-1)  # we know tomorrow's type

ridership_multivar = pd.get_dummies(ridership_multivar)  # one-hot encode the day type

# Changes datatypes of day type columns from bool to float to create TensorFlow dataset
ridership_multivar["next_day_type_A"] = ridership_multivar["next_day_type_A"].astype(float)
ridership_multivar["next_day_type_U"] = ridership_multivar["next_day_type_U"].astype(float)
ridership_multivar["next_day_type_W"] = ridership_multivar["next_day_type_W"].astype(float)

# Shows the encoded multivariate dataset
display(ridership_multivar)
print(ridership.shape)

# Splits the time-series into three periods, for training, validation, and testing
multivar_train = ridership_multivar["2016-01":"2018-12"]     # 3 years
multivar_val = ridership_multivar["2019-01":"2019-05"]       # 5 months
multivar_test = ridership_multivar["2019-06":]               # remaining period from 2019-06

# Prepares TensorFlow specific datasets
tf.random.set_seed(42)

multivar_train_ds = tf.keras.utils.timeseries_dataset_from_array(
    multivar_train.to_numpy(),                                # use all 5 columns as input
    targets=multivar_train["rail_boardings"][seq_length:],  # forecast only the rail series
    sequence_length=seq_length,
    batch_size=32,
    shuffle=True,
    seed=42
)

multivar_val_ds = tf.keras.utils.timeseries_dataset_from_array(
    multivar_val.to_numpy(),
    targets=multivar_val["rail_boardings"][seq_length:],
    sequence_length=seq_length,
    batch_size=32
)
```

```
# Forecasting Using a Simple RNN
# Resets all the keras states
tf.keras.backend.clear_session()

tf.random.set_seed(42)

# Creates an RNN with 32 recurrent neurons followed by a dense output layer with one output
neuron
# The same model was used before for univariate forecasting, but it is now being used for
multivariate forecasting
multivar_simple_rnn = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),   # Now, model accepts 5 inputs
    tf.keras.layers.Dense(1)
])

# Sets callback to stop training when model does not improve after a certain number of training
iterations
early_stopping_callback = tf.keras.callbacks.EarlyStopping(
    monitor="val_mae", patience=50, restore_best_weights=True
)

# Sets the model optimizer and compiles it with specific loss function and metric
multivar_simple_rnn.compile(
    loss="huber",
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.05, momentum=0.9),
    metrics=["mae"]
)

# Starts model training process over specified training, validation data and callbacks
history = multivar_simple_rnn.fit(
    multivar_train_ds, validation_data=multivar_val_ds, epochs=500,
callbacks=[early_stopping_callback]
)

# After training, model gets evaluated against validation data
val_loss, val_mae = multivar_simple_rnn.evaluate(multivar_val_ds)
print("Validation MAE of the Multivariate Simple RNN:", val_mae * 1e6)
```