# Tuning Neural Networks - 1

*One should look for what is and not what he thinks should be. (Albert Einstein)*

# Tuning Neural Networks: Topic introduction

In this part of the course, we will cover the following concepts:

- Visualize model performance using Neptune
- Performance tuning and keras Tuner
- Accelerating NN training

# Module completion checklist

| Objective | Complete |
| --- | --- |
| Visualize model performance using Neptune | |
| Introduce performance tuning and keras tuner | |

# Loading packages

- Let's load the packages we will be using:

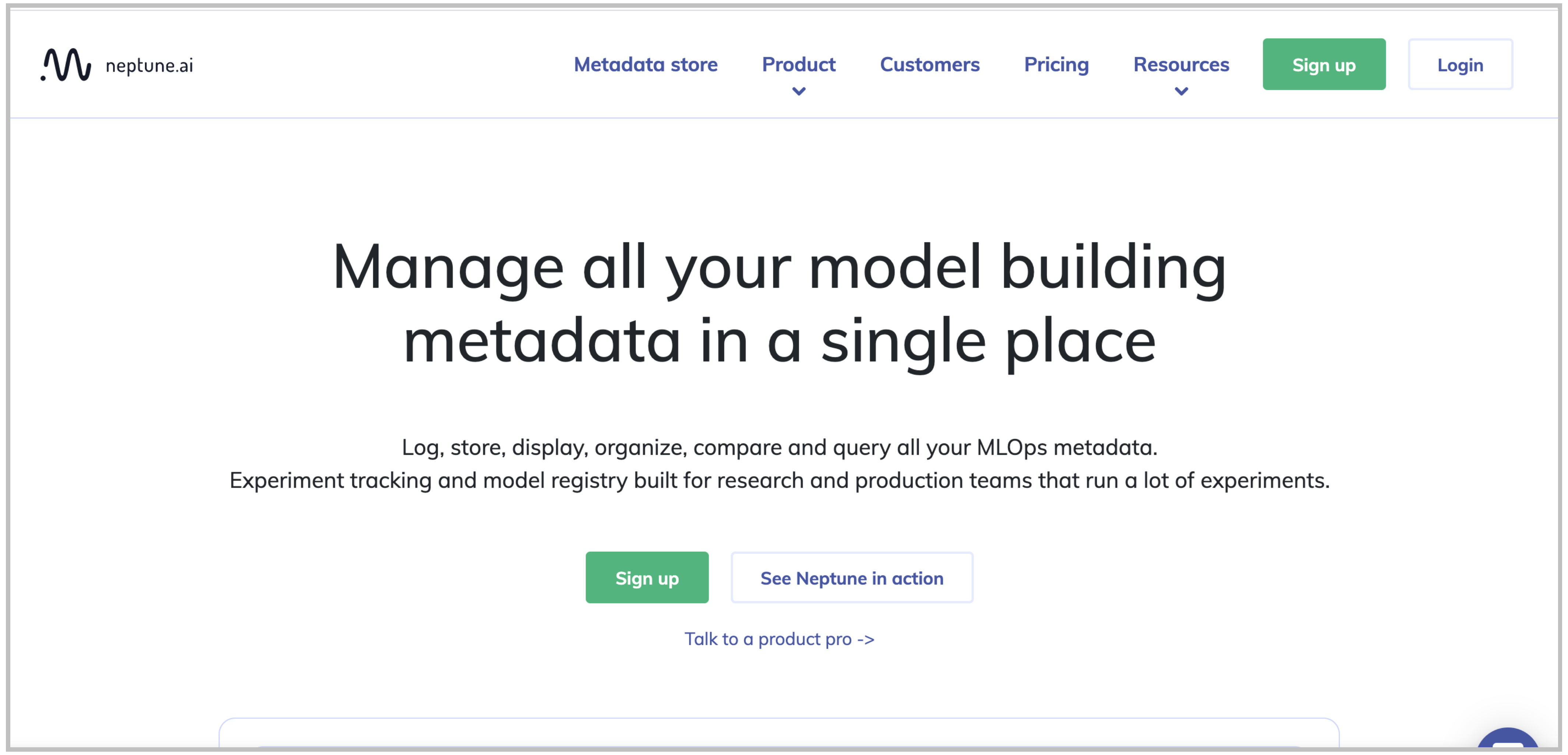
```
# Helper packages.
import os
import pickle
import pandas as pd
import numpy as np
# Scikit-learn packages.
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn import preprocessing

# TensorFlow and supporting packages.
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from tensorflow.keras.models import load_model
```

```
/opt/conda/envs/python-r-course-test/bin/python:1: DeprecationWarning: `import kerastuner`
is deprecated, please use `import keras_tuner`.
```

```
from tensorflow.keras.layers import Dropout
import neptune.new as neptune
from neptune.new.integrations.tensorflow_keras import NeptuneCallback
```

# Introducing Neptune

- To track our model's metadata, we will use **Neptune.ai**
- This web-based experiment tracking tool allows us to organize and compare performance metrics such as loss and accuracy
- We'll use Neptune today to visualize the performance metrics for various models
- Refer to this *link* to know more about Neptune and how it works



# Using Neptune

- Setting up an account on Neptune is fairly easy! Please refer to the instruction manual to set up your Neptune account
- Once your account is setup, make a note of the **user name** and **API token**
- You'll need these details to initialize Neptune and track experiments later in this module

- In this session, we will build neural network models using **TensorFlow,** evaluate them, and then visualize the results using **Neptune**

# Setting up project parameters for Neptune

- The first step is to **initialize** the Neptune client using the `init` function
  - Set `project_qualified_name` parameter to `USER_NAME/PROJECT_NAME`, in order to track the project you have set up in your Neptune account
  - In the code notebook, we have configured the project name as `sandbox`
  - Set your `API_TOKEN` token to the one you noted down from your Neptune account

```
# Initialize your Neptune client.
run = neptune.init(project='USER_NAME/PROJECT_NAME', #<- track your project
                   api_token = 'API_TOKEN')          #<- set your API token
```

- The second step is to **integrate Neptune with Tensorflow** by adding Neptune to the callback of the model with the `NeptuneCallback` function

```
from neptune.new.integrations.tensorflow_keras import NeptuneCallback
```

# Directory settings

- In this section we will build a neural network model using TensorFlow, evaluate it, and then visualize our results using **Neptune**
- Before that, we'll encode the directory structure into `variables` in order to maximize the efficiency of your workflow
- Let the `data_dir` be the variable corresponding to your `data` folder

```
from pathlib import Path
home_dir = Path(".").resolve()
main_dir = home_dir.parent.parent
print(main_dir)
```

```
data_dir = str(main_dir) + "/data"
print(data_dir)
```

# Load the data

- The `credit_card_data` dataset contains information about credit card **defaulters**
- Our goal is to predict if the customer will **default on a credit card payment or not**

```
credit_card = pd.read_csv(str(data_dir) + '/credit_card_data.csv')
print(credit_card.head())
```

```
   ID  LIMIT_BAL  SEX  ...  PAY_AMT5  PAY_AMT6  default_payment_next_month
0   1      20000    2  ...         0         0                           1
1   2     120000    2  ...         0      2000                           1
2   3      90000    2  ...      1000      5000                           0
3   4      50000    2  ...      1069      1000                           0
4   5      50000    1  ...       689       679                           0

[5 rows x 25 columns]
```

# Data prep: convenience function

- Lucky for you, Data Society wrote a time-saving function to perform all of the cleaning and split steps on the credit card dataset at once!

```
def data_prep(df):

    # Fill missing values with mean
    df = df.fillna(df.mean()['BILL_AMT1'])
    # Drop an unnecessary identifier column.
    df = df.drop('ID',axis = 1)

    # Convert 'sex' into dummy variables.
    sex = pd.get_dummies(df['SEX'], prefix = 'sex', drop_first = True)
    # Convert 'education' into dummy variables.
    education = pd.get_dummies(df['EDUCATION'], prefix = 'education', drop_first = True)
    # Convert 'marriage' into dummy variables.
    marriage = pd.get_dummies(df['MARRIAGE'], prefix = 'marriage', drop_first = True)
    # Drop `sex`, `education`, `marriage` from the data.
    df.drop(['SEX', 'EDUCATION', 'MARRIAGE'], axis = 1, inplace = True)

    # Concatenate `sex`, `education`, `marriage` dummies to our dataset.
    df = pd.concat([df, sex, education, marriage], axis=1)
```

```
    # Separate predictors from data.
    X = df.drop(['default_payment_next_month'], axis=1)
    # Separate target from data.
    y = df['default_payment_next_month']
```

# Data prep: convenience function - cont’d

```
# Set the seed to 1.
np.random.seed(1)

# Split data into train, test, and validation set, use a 70 - 15 - 15 split.
# First split data into train-test with 70% for train and 30% for test.
X_train, X_test, y_train, y_test = train_test_split(X.values,
                                                    y,
                                                    test_size = .3,
                                                    random_state = 1)

# Then split the test data into two halves: test and validation.
X_test, X_val, y_test, y_val = train_test_split(X_test,
                                                y_test,
                                                test_size = .5,
                                                random_state = 1)

print("Train shape:", X_train.shape, "Test shape:", X_test.shape, "Val shape:", X_val.shape)
```

```
# Transforms features by scaling each feature to a given range.
# The default is the range between 0 and 1.
min_max_scaler = preprocessing.MinMaxScaler()
X_train_scaled = min_max_scaler.fit_transform(X_train)
X_test_scaled = min_max_scaler.transform(X_test)
X_val_scaled = min_max_scaler.transform(X_val)

return X_train_scaled, X_test_scaled, X_val_scaled, y_train, y_test, y_val
```

# Data prep

```
X_train_scaled, X_test_scaled, X_val_scaled, y_train, y_test, y_val = data_prep(credit_card)
```

```
Train shape: (21000, 30) Test shape: (4500, 30) Val shape: (4500, 30)
```

# Define metrics

- Let's wrap all metrics of interest into one list `METRICS`

```
METRICS = [
      keras.metrics.TruePositives(name='tp'),
      keras.metrics.FalsePositives(name='fp'),
      keras.metrics.TrueNegatives(name='tn'),
      keras.metrics.FalseNegatives(name='fn'),
      keras.metrics.BinaryAccuracy(name='accuracy'),
      keras.metrics.Precision(name='precision'),
      keras.metrics.Recall(name='recall'),
      keras.metrics.AUC(name='auc')
]
```

# Define model function

- Let’s now create a convenience function to define and compile a simple neural network model with an input layer, two hidden layers, and an output layer

```
def create_model(lr = .75):
  # Let's set the seed so that we can reproduce the results.
  tf.random.set_seed(1)
  opt = Adam(learning_rate = lr) # <- set optimizer

  model = Sequential([
          Dense(32, activation='relu',
                input_dim=X_train_scaled.shape[1]),#<- set input and 1st hidden layer
          Dense(32, activation='relu'),           #<- set 2nd hidden layer
          Dense(1, activation='sigmoid')          #<- set output layer

  ])

  model.compile(optimizer = opt,               #<- set optimizer
                loss='binary_crossentropy',   #<- set loss function to binary_crossentropy
                metrics= METRICS)             #<- set performance metric
  return model
```

# Launching Neptune: create callback

- We first create the callback so that the model will be logged by Neptune and we can visualize the metrics
- To create the callback and configure the runs in neptune, we use the function `NeptuneCallback`

```
neptune_cbk = NeptuneCallback(run=run)
callbacks = [neptune_cbk]

# Create and compile the model.
model = create_model()
```

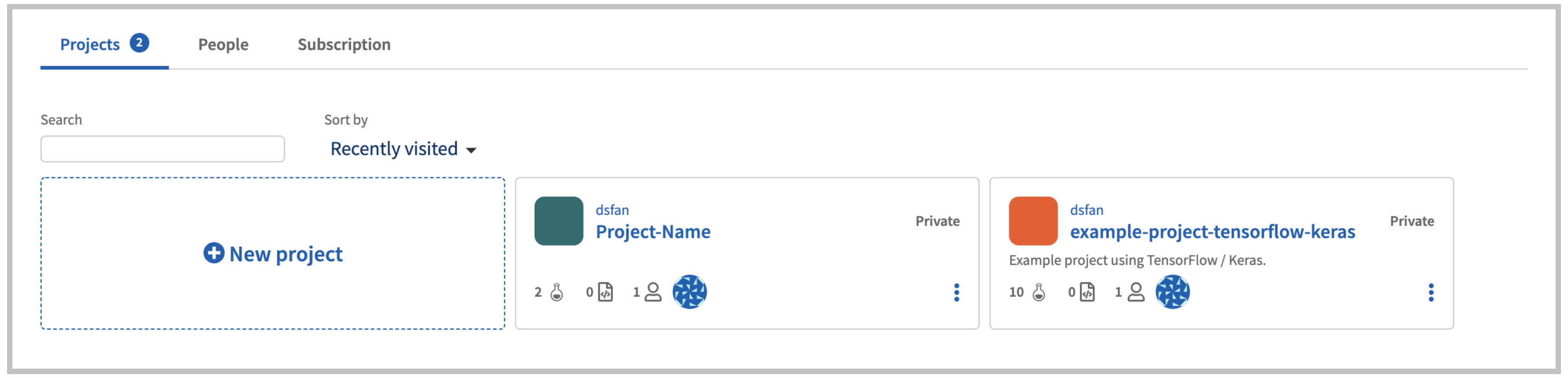
# Fitting the model with Neptune callback

- The final step is to fit the model
- We will pass our list `callbacks` as a `callbacks` argument in the `fit()` function

```
# Fit the model with NeptuneCallback
model_default = model.fit(X_train_scaled, y_train,
                          validation_data = (X_val_scaled, y_val),
                          epochs = 25,
                          verbose = 0,          #<- silence the epoch output in console (optional)
                          callbacks = callbacks)#<- add callbacks
```

# View the model on neptune.ai

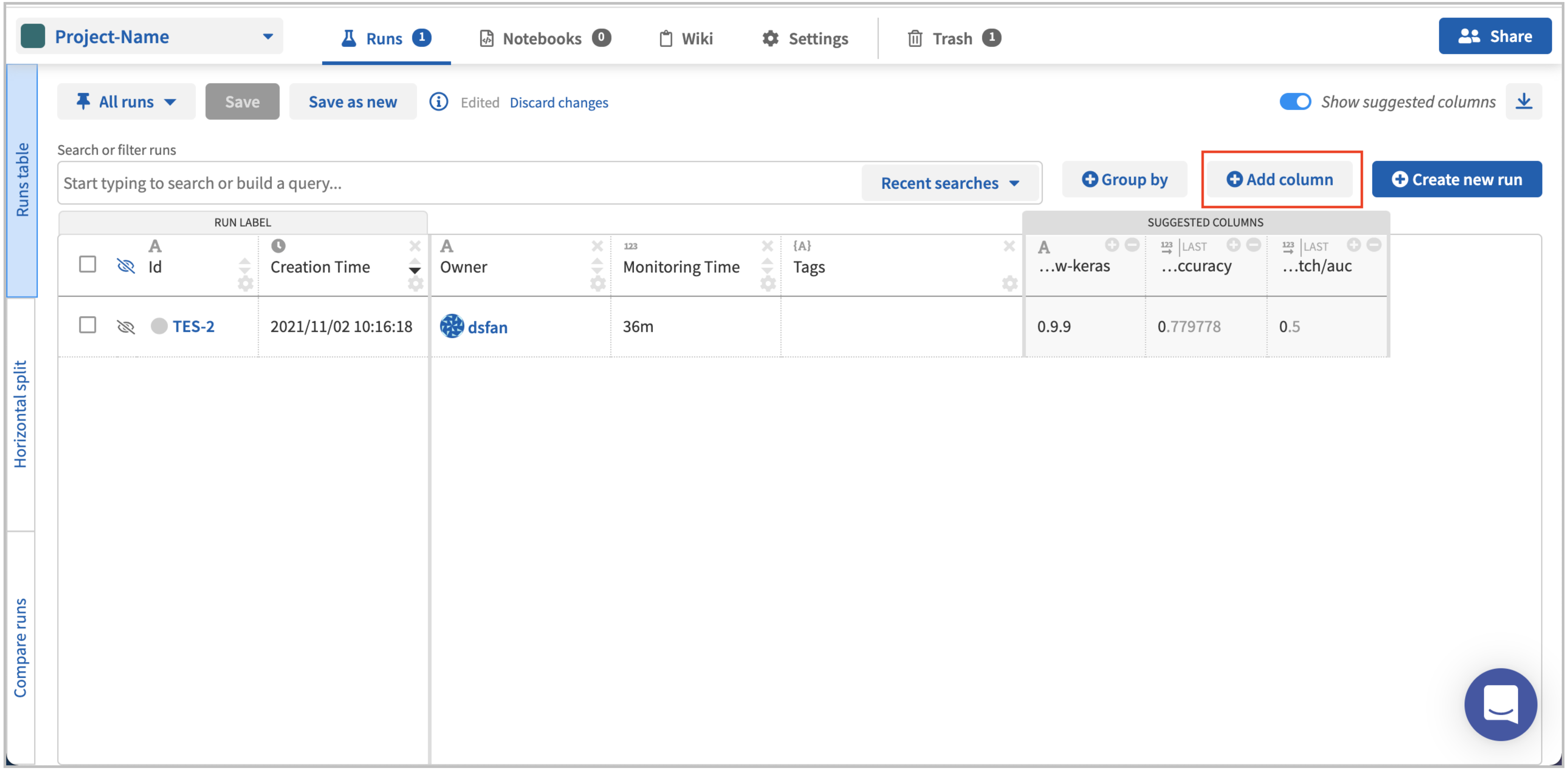
- To view the model that we just ran, launch the Neptune in browser as shown below and click on the project we created earlier - `sandbox`



# Neptune - Runs table

- On the `Runs table` page, you will see all the models that you've added with the `NeptuneCallback` function
- It is a summary page that lists the Id, Creation Time, Owner, Monitoring Time of the model
- You can also add the metrics, CPU usage, and/or other system information about the model to this page by clicking on the `Add column` button

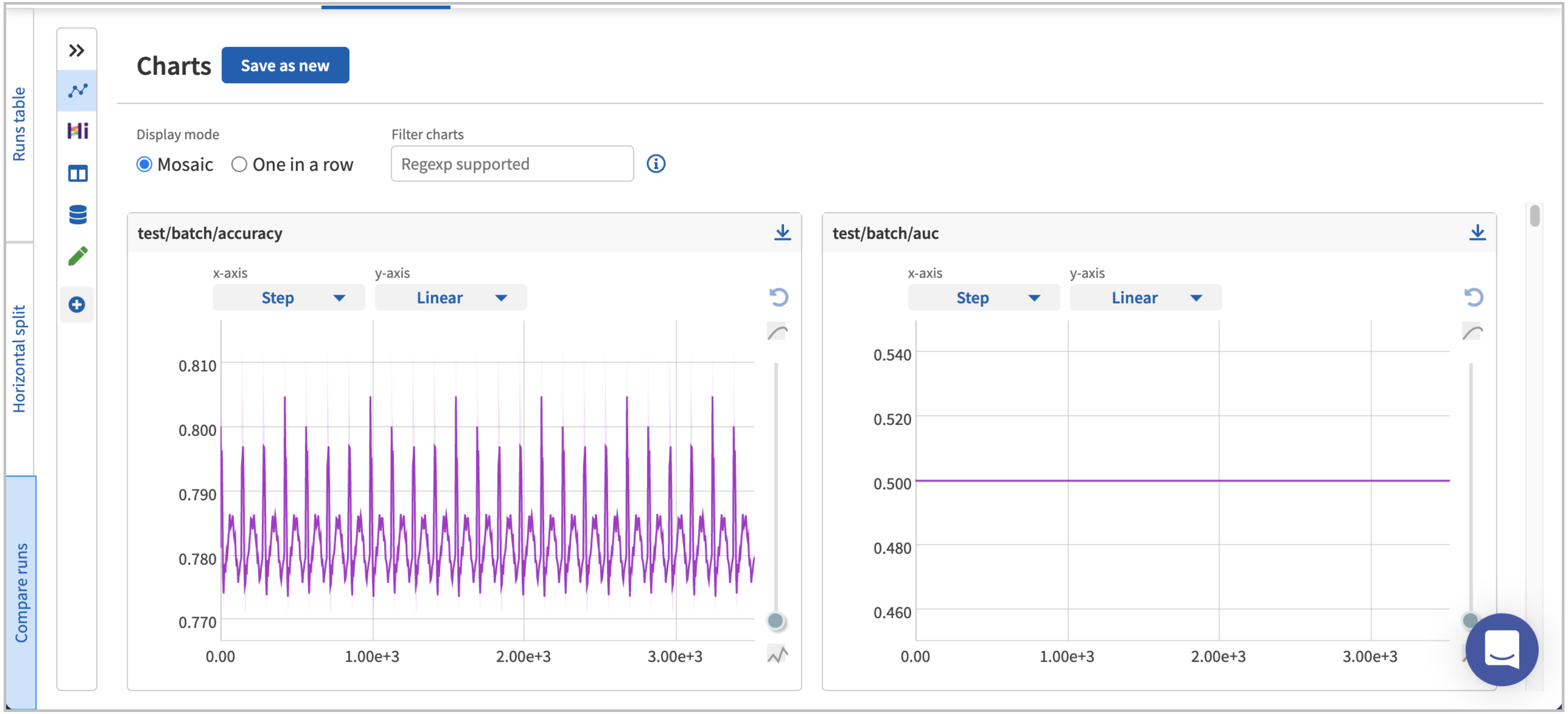
# Neptune - Runs table (cont'd)



# Neptune - Compare runs

- On the Compare runs page, you will see multiple charts detailing the metrics that you specify (accuracy, auc, etc.) by train/test and batch/epoch
- You can customize this dashboard to view the most important metrics and compare the performance of multiple models side-by-side!

# Neptune - Compare runs (cont'd)



# Evaluate the model

- We'll now predict on test data and evaluate the metrics

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

```
[0.5322661399841309, #<- loss
 0.0,                #<- tp
 0.0,                #<- fp
 3491.0,             #<- tn
 1009.0,             #<- fn
 0.7757777571678162, #<- accuracy
 0.0,                #<- precision
 0.0,                #<- recall
 0.5]                #<- auc
```

- The last step is to stop the neptune run after logging the meta data

```
run.stop()
```

```
Shutting down background jobs, please wait a moment...
Done!
```

# Module completion checklist

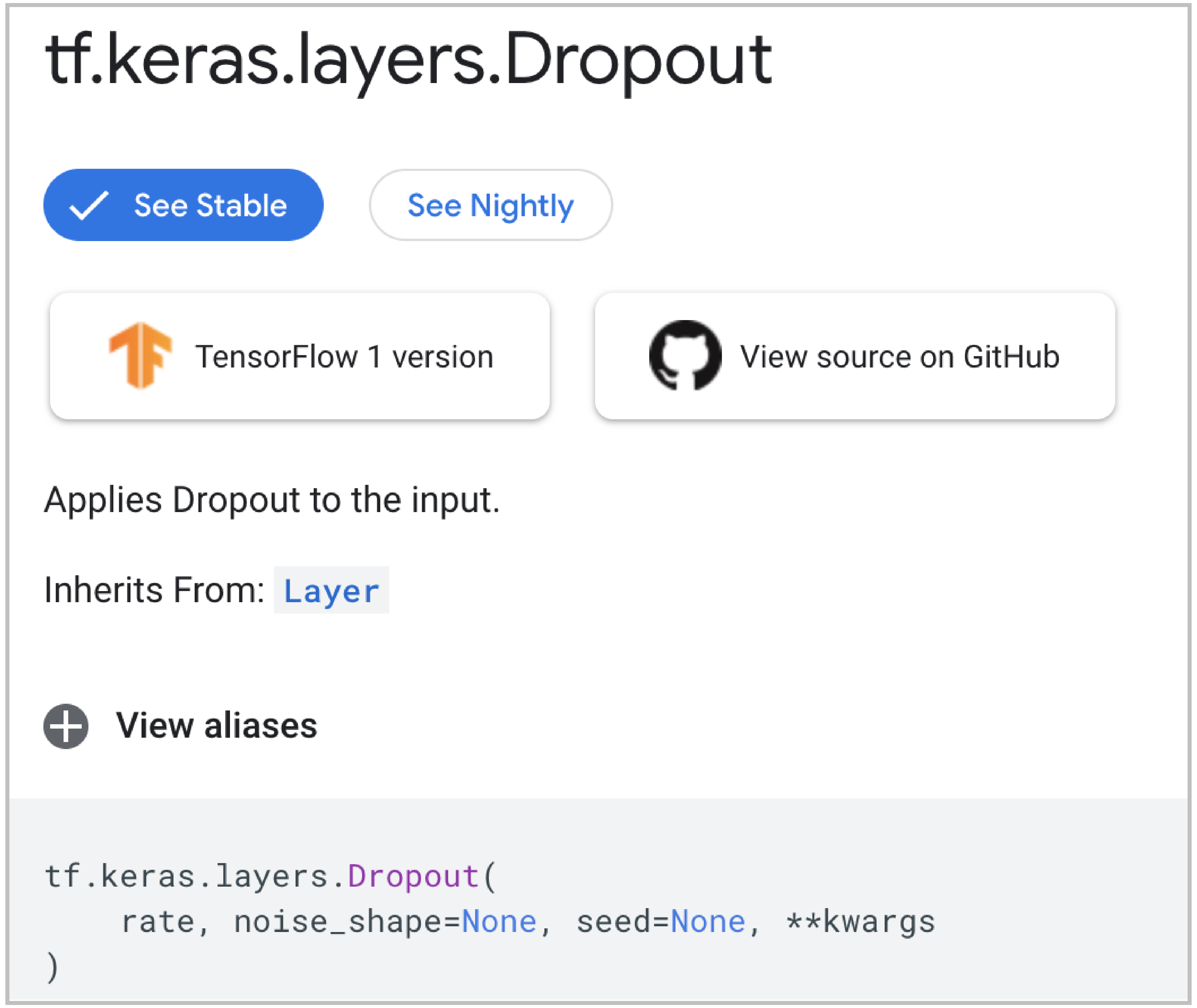
| Objective | Complete |
| --- | --- |
| Visualize model performance using Neptune | ✔ |
| Introduce performance tuning and keras tuner | |

# Performance tuning

- From the plots and the metrics, we can se that our model is not doing well
  - it performs better on training data, achieving accuracy of over `77%`, but it severely oscillates, which may be a sign of a high learning rate
  - the accuracy for validation data is constant and the loss oscillates a bit across epochs
  - the accuracy is lower for validation data
- We can tune the model using a Keras package called `kerastuner` that has various tuning modules for building optimized models

# Preventing overfitting using dropout

- **Dropout** randomly drops neurons (along with their connections) from the neural network during training
- It prevents neurons in layers from co-adapting too much
- It takes 2 arguments:
  - `rate`: a proportion of neurons to be dropped (~`0.1-0.4`)
  - `seed`: “locks” the random number generator for reproduceable our results




- For more information, visit the *documentation page*

# Tuning using Keras Tuner

- Keras Tuner comes with algorithms like Random Search, Bayesian Optimization, and Hyperband for tuning the hyperparameters
- We'll tune hyperparameters like:
  - Optimizer
  - Activation function
  - Learning rate
  - Number of neurons
  - Rate of the dropout layer
- Then, we'll create a model with the optimized parameters

# Types of hyperparameters

- In Keras Tuner, each hyperparameter can be tuned based on its type, such as:
  - `Float`
  - `Int`
  - `Boolean`
  - `Choice`
- The activation function, learning rate, and optimizer are the `Choice` type
- The number of hidden neurons and rate in the dropout layer are numerical hyperparameters of type `Int` and `Float`, respectively
- Each hyperparameter has options which can be set while we tune it
- The `Float` and `Int` type parameters have a:
  - minimum value
  - maximum value
  - default value
  - step value (the minimal step between two hyperparameter values)
- The `Choice` type parameter requires a set of possible values

# Knowledge check

Link: ***https://forms.gle/YPJNgyCz1fcnxt6n9***

# Module completion checklist

| Objective | Complete |
|---|---|
| Visualize model performance using Neptune | ✔ |
| Introduce performance tuning and keras tuner | ✔ |

# Congratulations on completing this module!

You are now ready to try Tasks 1-9 in the Exercise for this topic