DATA SOCIETY:

Tuning Neural Networks - 2

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

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

Module completion checklist

Objective	Complete
Tuning the model with Keras Tuner	
Accelerating NN training	

Define the tuner

- As discussed earlier, we add a hidden layer and set the number of neurons using `Int` type and set the minimum and maximum values respectively
- We add a couple of options to choose from using the `Choice` type of parameter for the activation function
- We then define the drop out layer in the similar way as the hidden layer

```
def tune_model(hp):

    # Number of 1st hidden layer neurons.
    units = hp.Int('units',          #<- Number of neurons
                   min_value = 8,   #<- min value
                   max_value = 64,  #<- max value
                   step = 8)

    # Tuning activation function for 1st hidden layer.
    activation = hp.Choice('activation', #<- Activation function
                          [              #<- Types of activation functions
                              'relu',
                              'tanh',
                              'sigmoid'
                          ])

    dropout_1 = hp.Float('dropout_1',
                         min_value = 0.0,
                         max_value = 0.5
```

Define the tuner (cont'd)

- Similar to the activation function, we define the optimizer and the learning rate using the `Choice` parameter
- Upon defining all the layers, we add them to the model and then compile it

```
# Tuning optimizer.
optimizer = hp.Choice('optimizer', ['adam', 'sgd', 'rmsprop'])

# Tuning learning rate
lr = hp.Choice('learning_rate', [1e-2, 1e-3, 1e-4])

model = keras.Sequential()

model.add(keras.layers.Dense(units = units,
                             activation = activation,
                             input_dim = X_train_scaled.shape[1]))

# Tuning number of neurons in 2nd hidden layer
model.add(keras.layers.Dense(units = units,
                             activation = activation))

# Tuning rate hyperparameter of dropout layer
model.add(Dropout(rate = dropout_1))
```

Define the tuner (cont'd)

- We will use the **Random Search tuner** which randomly samples the hyperparameter combinations based on our input and tests them to find the optimal combination
- Our `objective` is to achieve **maximum validation accuracy** using the optimal parameters
- `max_trials` represents the number of hyperparameter combinations that will be tested by the Random search tuner
- `executions_per_trial` represents the number of models that should be built and fit for each trial

```
MAX_TRIALS = 10
EXECUTIONS_PER_TRIAL = 5
tuner = RandomSearch(
    tune_model,
    objective = 'val_accuracy',
    max_trials = MAX_TRIALS,
    executions_per_trial = EXECUTIONS_PER_TRIAL,
    directory = 'final_tuned_model',
    project_name = 'final_tuned_model',
    seed = 1
)
```

View search space summary

- Once the tuner is setup, the search space can be checked using `search_space_summary()`

```
tuner.search_space_summary()
```

```
Search space summary
|-Default search space size: 5
units (Int)
|-default: None
|-max_value: 64
|-min_value: 8
|-sampling: None
|-step: 8
activation (Choice)
|-default: relu
|-ordered: False
|-values: ['relu', 'tanh', 'sigmoid']
dropout_1 (Float)
|-default: 0.25
|-max_value: 0.5
|-min_value: 0.0
|-sampling: None
|-step: 0.05
optimizer (Choice)
|-default: adam
|-ordered: False
|-values: ['adam', 'sgd', 'rmsprop']
learning_rate (Choice)
|-default: 0.01
|-ordered: True
|-values: [0.01, 0.001, 0.0001]
```

Fit the model

- We finally fit the model using the `search` function
- It takes the mandatory training data to fit the model and the validation data can be provided as an optional input
- The `epochs` parameter is used to define the number of training epochs for each hyperparameter combination

```
tuner.search(x=X_train_scaled,
             y=y_train,
             verbose=0,
             epochs=25,
             validation_data=(X_val_scaled, y_val))
```

View the optimal parameters

- The `get_best_trials` function is used to get the dictionary of optimal hyperparameters

```
optimal_params = tuner.oracle.get_best_trials(num_trials=1)[0].hyperparameters.values
optimal_params
```

```
{'units': 32,
 'activation': 'tanh',
 'dropout_1': 0.15,
 'optimizer': 'adam',
 'learning_rate': 0.01}
```

Define and compile optimized model

```
def create_optimized_model(units, activation, dropout_1, optimizer, learning_rate, dropout_seed =
1):

    # Set up model.
    model = Sequential()

    model.add(Dense(units,
                    input_dim = X_train_scaled.shape[1],
                    activation = activation))
    model.add(Dense(units,
                    activation = activation))

    if dropout_1 is not None:
        model.add(Dropout(rate = dropout_1, seed = dropout_seed))
        model.add(Dense(1, activation = 'sigmoid'))
```

```
    # Compile model.
    model.compile(loss = 'binary_crossentropy', optimizer = optimizer, metrics = METRICS)
    return model
```

Setup Neptune run for optimized model

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

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

```
# Create and compile the optimized model.
tb_model = create_optimized_model(**optimal_params)
```

Fit the optimized model

```
tb_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
```

Evaluate optimized model on test data

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

```
[0.44843247532844543, #<- loss
 371.0,               #<- tp
 209.0,               #<- fp
 3282.0,              #<- tn
 638.0,               #<- fn
 0.8117777705192566,  #<- accuracy
 0.6396551728248596,  #<- precision
 0.367690771818161,   #<- recall
 0.7587265968322754]  #<- auc
```

- We can see that validation accuracy of the optimized model seems to be higher than the baseline model

```
run.stop()
```

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

Module completion checklist

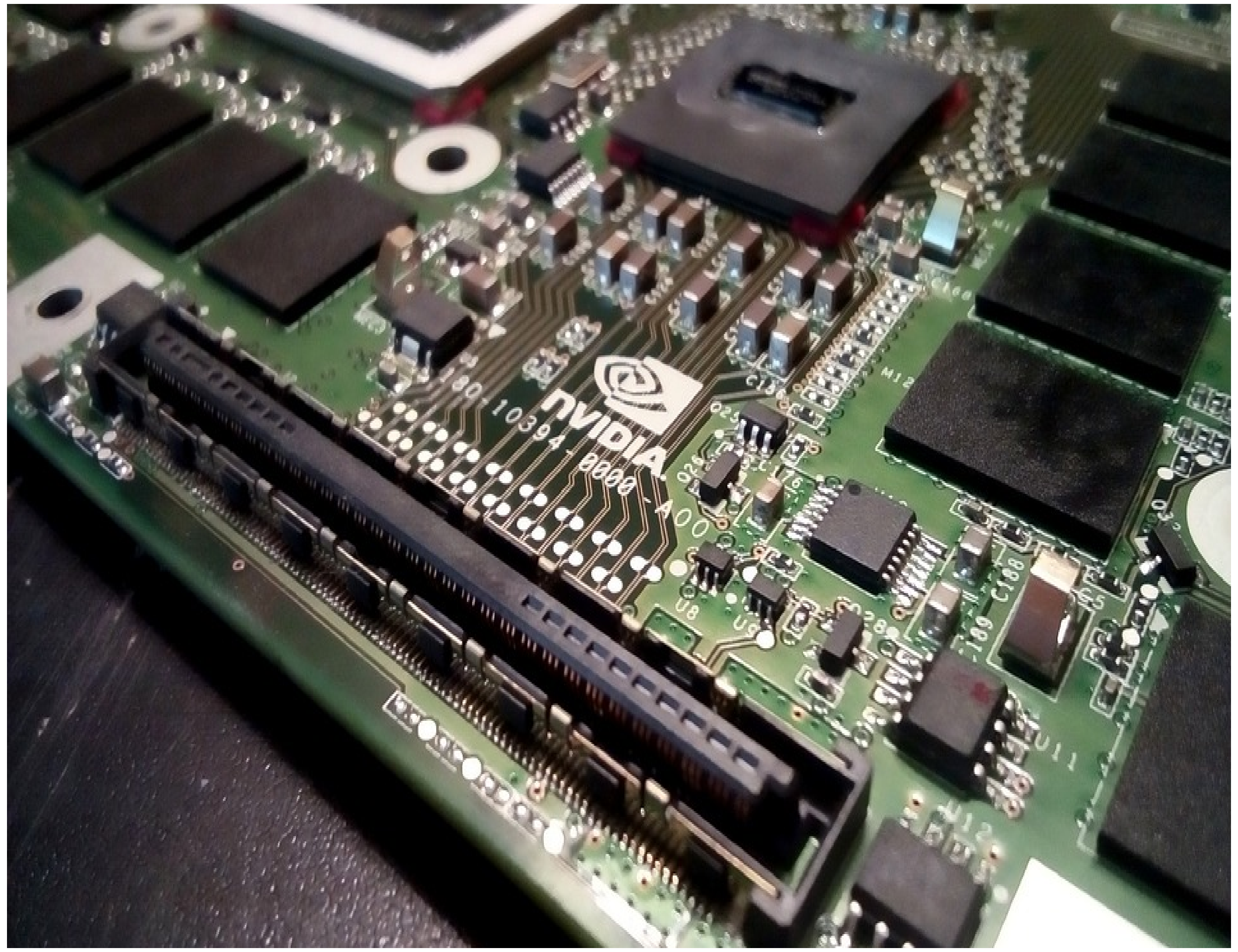
Objective	Complete
Tuning the model with Keras Tuner	✔
Accelerating NN training	

Processing speed

- Recent advances in deep learning have made the use of large, deep neural networks with tens of millions of parameters suitable for a number of applications that require real-time processing
- The sheer size of these networks can represent a challenging computational burden, even for modern **central processing units (CPUs)**.
- Let's look at the types of processors that are available that can help us speed up computation: the TPU and the GPU

Graphics processing unit

- A **graphics processing unit (GPU)** is an electronic circuit **specially designed to process graphics** such as images and video
- Using a GPU can help speed up computation
- It breaks complex problems into thousands of smaller tasks and executes them simultaneously
- It's more suitable than a CPU for processing large blocks of data and thousands of tasks in parallel
- GPUs are used in video game consoles, mobile phones, embedded systems, etc.



CPU vs GPU? Which one to use when?

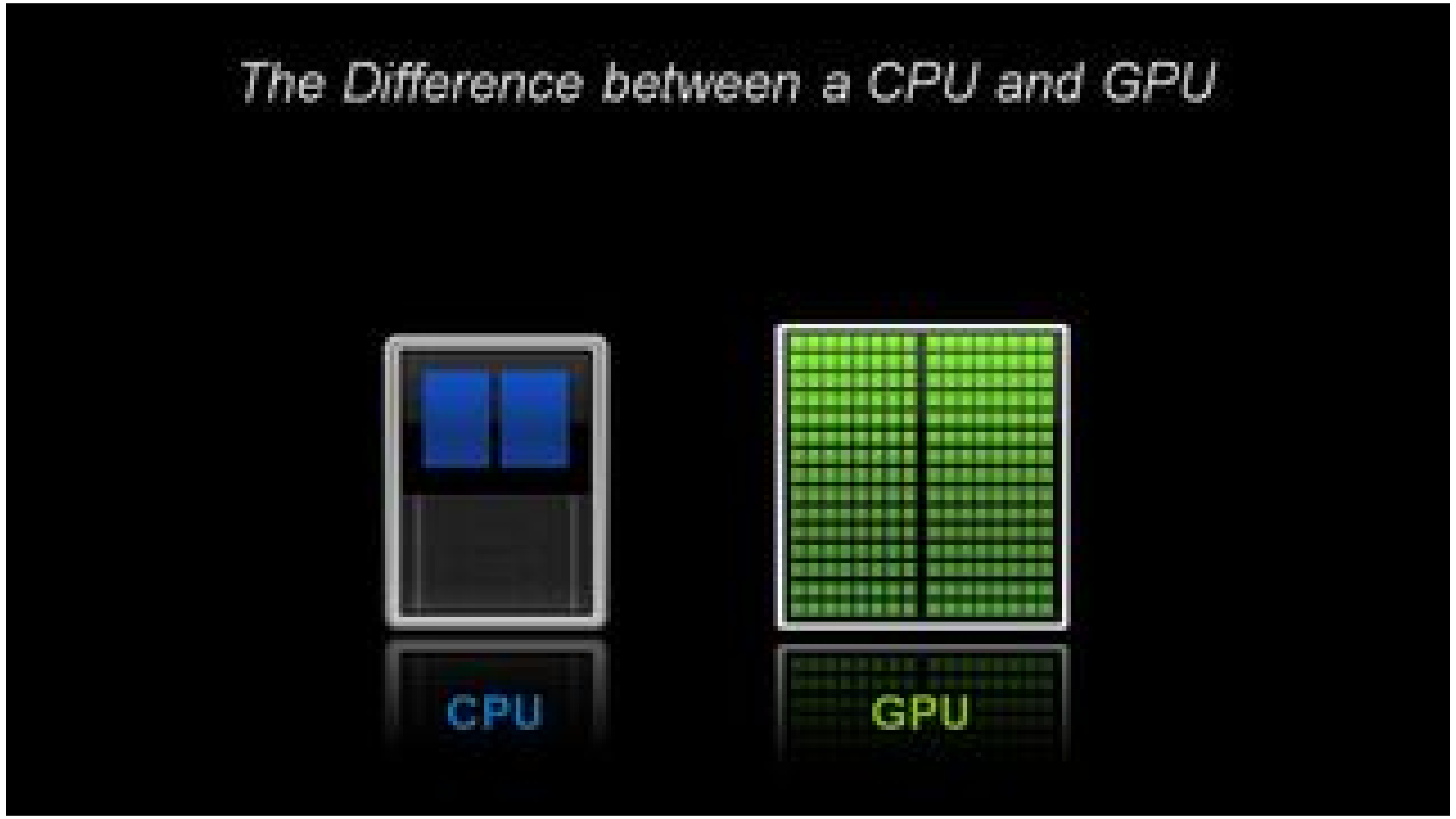
~~CPU (Central Processing Unit): ~~

- Ideal for general-purpose applications
- Composed of just a few cores
- Can handle a few software threads at a time
- Can run small models with small batch sizes

~~GPU (Graphics Processing Unit): ~~

- More suited for specialized applications
- Composed of hundreds of cores
- Can handle thousands of threads simultaneously
- Can run medium to large models with larger batch sizes

CPU vs GPU? Which one to use when? (cont'd)



Source : Nvidia blog

Tensor processing unit

- A tensor processing unit (TPU) is Google's custom-developed **application-specific integrated circuit (ASIC)**
- It's specifically designed for Google's TensorFlow framework
- Using a TPU can accelerate the training of complex neural networks as it's very fast at performing vector and matrix computations
- Models that train for weeks or months converge in hours on TPUs
- TPU is available in Google Colab when run on a remote host

Cloud TPU

- The Cloud TPU family of products make the benefits of TPUs available via cloud computing resources
- With Cloud TPUs you can:
  - accelerate machine learning applications
  - scale your applications quickly
  - cost-effectively manage your machine learning workloads
  - start with well-optimized, open source reference models

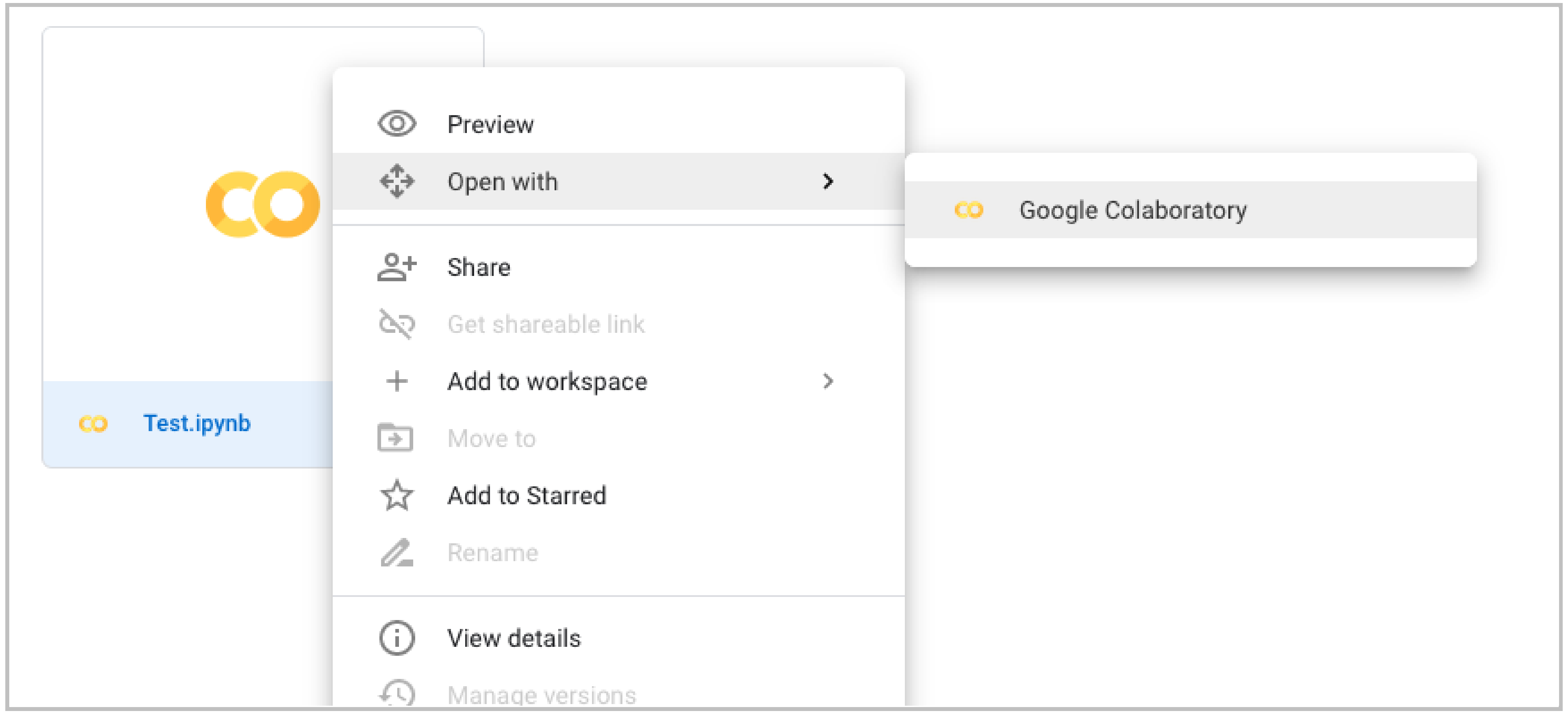
- Check the Google Cloud website, which is available at ***this link***
- The site will explain how to set up a Google Cloud environment and configure a custom TPU machine. The link also provides steps to train a model on Cloud TPU

Comparing CPU, GPU, & TPU

- Several cloud infrastructures provide the flexibility to select the hardware of your choice before runtime
- If you have access to Google Colab, you can try out all three–a CPU, a GPU, and a TPU
  - Open your Google account by going to **http://drive.google.com** in your web browser
  - Click on the app menu icon (the dots in the upper-right corner next to the logo) and scroll down to the bottom and click on `More from G Suite Marketplace`
  - Search for **Colaboratory** and install

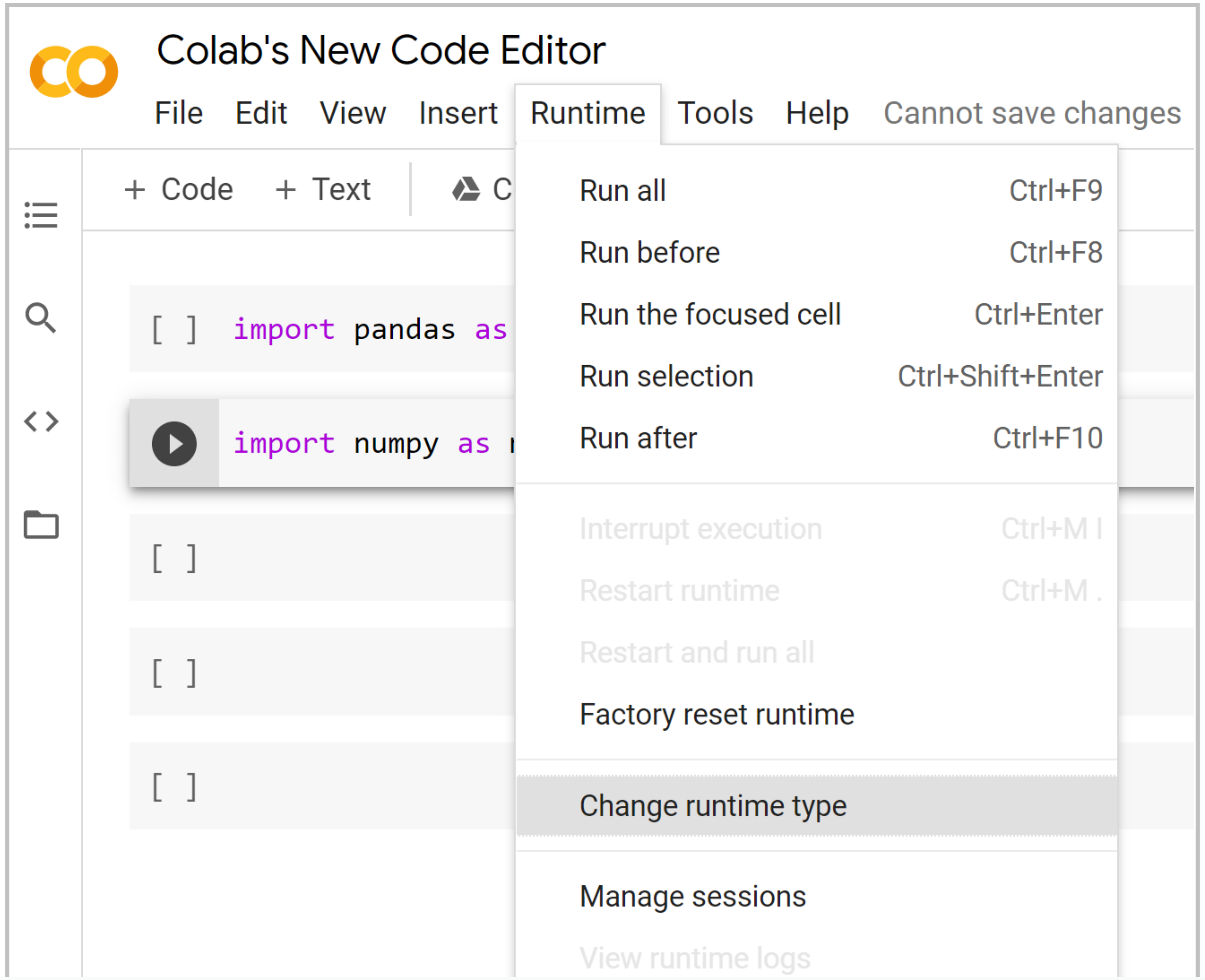
Comparing CPU, GPU, & TPU

- Upload a Jupyter notebook (`*.ipynb`) to open with Colab

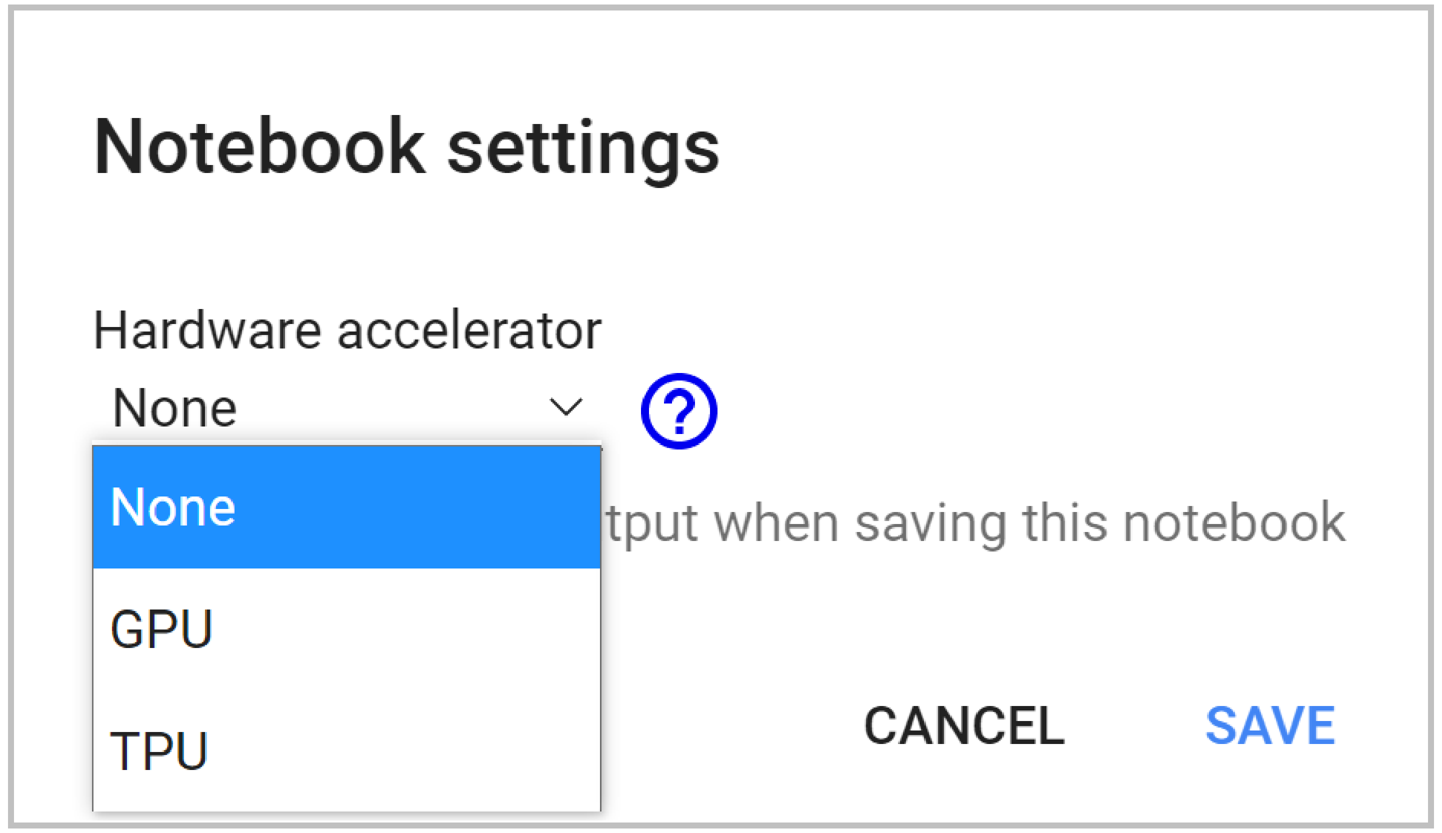


Comparing CPU, GPU, & TPU

- Open Jupyter notebook within Colab
- Go to `Runtime` and click on `Change runtime type`



- Select the hardware accelerator you want to run your code in and compare the results



Distributed training in TensorFlow

- We can also speed up processing by distributing a large dataset or large model over multiple machines
- TensorFlow comes with a module named `tf.distribute.Strategy` that can distribute training across multiple GPUs, machines, or TPUs
- Using this API, you can distribute your existing models and training code with minimal code changes

Strategies

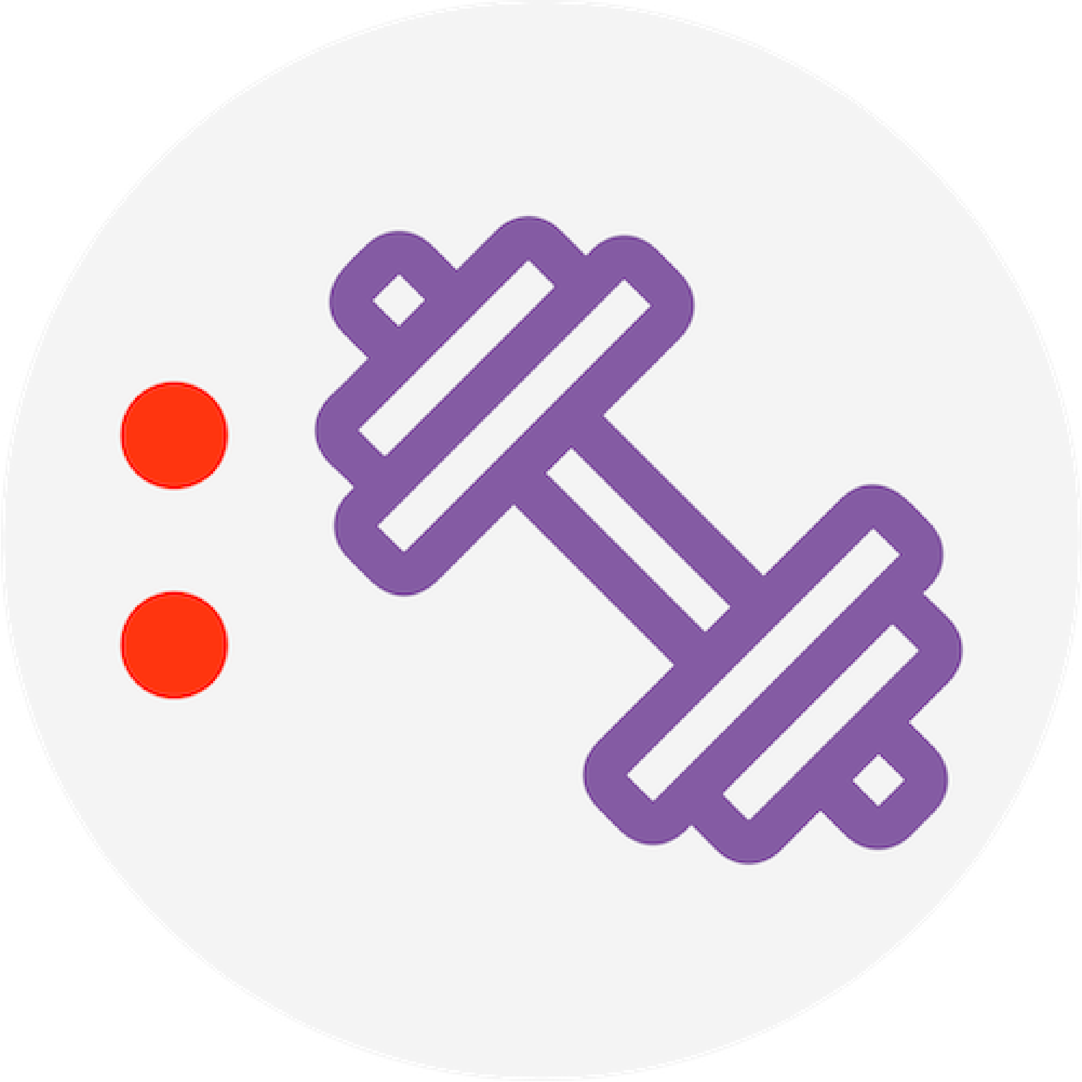
- `tf.distribute.Strategy` lets you choose from various types of strategies for different situations
- The **hardware platform** strategy, for example, enables you to scale training onto:
  - multiple GPUs on one machine
  - multiple machines in a network (a.k.a. cluster)
  - cloud TPUs

- You can read more about distributed training with TensorFlow using ***this link***

Knowledge check

Link: **https://forms.gle/AXYwutd8vSGx7NRe8**

Exercise



Module completion checklist

Objective	Complete
Tuning the model with Keras Tuner	✔
Accelerating NN training	✔

Tuning Neural Networks: Topic summary

In this part of the course, we have covered:

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

Congratulations on completing this module!