.This course focuses on **implementing tools** and t**echniques t**o effectively manage your **modeling resources** and **best serve batch** and **real-time inference requests**.
In this course, you'll implement **effective search strategies for the best mode**l that will scale for various serving needs while **constraining model complexity and hardware requirements**.
You'll **optimize and manage this compute storage and IO resources** your model needs in production environments during its life cycle.
In this journey, you will continue to use the **TFX library** and rely on tools like **AutoML** for finding the **best suitable model and TensorFlow model analysis to address model fairness, explainability issues, and mitigate bottlenecks.** Along the way, you'll learn about more specialized **scenarios** like **dimensionality reduction** and **pruning** to manage your model resources wisely
 exploring **pipelining** and **parallelism** and high-performance **ingestion** to make the most of your computational resources.
A Production ML system must **run nonstop**, **at the minimum cost** while producing the maximum performance.

Hyperparameter Tuning

Keras Tuner.
similarities between hyperparameter tuning and neural architecture search.
 Neural architecture search, or NAS: automating the design of neural networks -> find the optimal architecture.

Types of parameters in ML Models:

Trainable parameters:
- learned by the algorithm during training. (weights bias of a neural netwok)

Hyperparamters:
- set before launching the learning process
- not updated in each training ste (learning rate or the number of units in a dense layer)
-

Hyperparameter
- architecture options
- activation functions
- weight initialization strategy
- optimization hyperparameters (learning rate, stop condition),

Tuning methods:
- Random search
- Hyperband
- Bayesian optimization
- sklearn

Keras Autotuner

Get Started | autotuner.py 2, U

autotuner.py > ...

```python
import tensorflow  as tf
from tensorflow import keras
mnist = tf.keras.datasets.mnist

(x_train,y_train),(x_test,y_test) = mnist.load_data()
x_train, x_test = x_train /255.0, x_test /255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation = 'softmax')
])

model.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])
model.fit(x_train,y_train, epochs=5)
model.evaluate(x_test,y_test)
```

?

```
pip install -q -U keras-tuner
```

Is this architecture optimal?

- Do the model need more or less hidden units to perform well?
- How does model size affect the convergence speed?
- Is there any trade off between convergence speed, model size and accuracy?
- Search automation is the natural path to take
- Keras tuner built in search functionality.

Other search strategen:

The parameters you choose will vary based on your strategy. But the important one to note is the **objective**.

In this case, our objective is val_accuracy, so we want to maximize on the validation accuracy. You can find details on the rest at the Keras site. Search can take a while to complete and use a lot of compute resources.
But you can configure a an **early stopping callback** that stops the search when the conditions are met. So for example, here I'm monitoring the validation loss and the patience is set to **five**, which means that it doesn't change significantly, or if it doesn't change significantly in five epics, then stop searching on this iteration. And you set the **call back as a search parameter.**
The rest of the parameter specify **how to search,** such as the **data** and **label**, the number of **epics to train** for, and the validation split. So for example how much data you will use to validate against the training set.
As it searches, you'll see the results of each trial.

In the example: searching on one parameter, th**e units in the dense hidden layer**. As you can see as it updates, you can keep track of the best value so far. Which at this point that we have in the slide is 48, and in this case actually it ended up that way too when it completed. So I can go back and try my architecture again and use the results of the Keras tuner to set the number of units manually, this time with 48 neurons in the layer, which is the number that we got from from Keras tuner.

And when it's retrained, you'll see the results. It's only been five epics and the value isn't quite as good as it was before hand, but our epics are three times quicker than they were. So I can maybe train for longer to get better results knowing that at the very least I've optimized part of my architecture.

```
from gc import callbacks
from pickletools import optimize
from tracemalloc import stop

from yaml import DirectiveToken
import tensorflow  as tf
from tensorflow import keras
import keras_tuner as kt

mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test) = mnist.load_data()
x_train, x_test = x_train /255.0, x_test /255.0

def model_builder(hp):
   model = keras.Sequential()
   model.add(keras.layers.Flatten(input_shape=(28,
28)))
```

```
   # hp_units=hp.Int('units', min_value=16,
max_value=512, step=16)
   # model.add(keras.layers.Dense(units=hp_units,
activation='relu'))
   model.add(keras.layers.Dense(hp.Choice('units',
[16,16,512]), activation='relu'))
   model.add(tf.keras.layers.Dropout(0.2))
   model.add(keras.layers.Dense(10))

   model.compile(optimizer='adam',
           loss='sparse_categorical_crossentropy',
           metrics=['accuracy'])
   return model

tuner = kt.Hyperband(model_builder,
                   objective = 'val_accuracy',
                   max_epochs=10,
                   factor=3,
                   directory='my_dir',
                   project_name = 'intro_to_kt')
# kt.RandomSearch

stop_early =
tf.keras.callbacks.EarlyStopping(monitor="val_loss",

patience=5)
# 5 steps not significantly changed

tuner.search(x_train,
           y_train,
           epochs=50,
           validation_split=0.2,
           callbacks=[stop_early])
```

```
best_model = tuner.get_best_models()[0]
```

week1 lab keras_tuner

Developing machine learning models is usually an iterative process. You start with an initial design then reconfigure until you get a model that can be trained efficiently in terms of time and compute resources. As you may already know, these settings that you adjust are called *hyperparameters*. These are the variables that govern the training process and the topology of an ML model. These remain constant over the training process and directly impact the performance of your ML program.

The process of finding the optimal set of hyperparameters is called *hyperparameter tuning* or *hypertuning*, and it is an essential part of a machine learning pipeline. Without it, you might end up with a model that has unnecessary parameters and take too long to train.

Hyperparameters are of two types:

1. *Model hyperparameters* which influence model selection such as the number and width of hidden layers
2. *Algorithm hyperparameters* which influence the speed and quality of the learning algorithm such as the learning rate for Stochastic Gradient Descent (SGD) and the number of nearest neighbors for a k Nearest Neighbors (KNN) classifier.

For more complex models, the number of hyperparameters can increase dramatically and tuning them manually can be quite challenging.

Ungraded Lab: Intro to Keras Tuner

Developing machine learning models is usually an iterative process. You start with an initial design then reconfigure until you get a model that can be trained efficiently in terms of time and compute resources. As you may already know, these settings that you adjust are called *hyperparameters*. These are the variables that govern the training process and the topology of an ML model. These remain constant over the training process and directly impact the performance of your ML program.

The process of finding the optimal set of hyperparameters is called *hyperparameter tuning* or *hypertuning*, and it is an essential part of a machine learning pipeline. Without it, you might end up with a model that has unnecessary parameters and take too long to train.

Hyperparameters are of two types:

1. *Model hyperparameters* which influence model selection such as the number and width of hidden layers
2. *Algorithm hyperparameters* which influence the speed and quality of the learning algorithm such as the learning rate for Stochastic Gradient Descent (SGD) and the number of nearest neighbors for a k Nearest Neighbors (KNN) classifier.

For more complex models, the number of hyperparameters can increase dramatically and tuning them manually can be quite challenging.

In this lab, you will practice hyperparameter tuning with [Keras Tuner](), a package from the Keras team that automates this process. For comparison, you will first train a baseline model with pre-selected hyperparameters, then redo the process with tuned hyperparameters. Some of the examples and discussions here are taken from the [official tutorial provided by Tensorflow]() but we've expounded on a few key parts for clarity.

Let's begin!

Note: The notebooks in this course are shared with read-only access. To be able to save your work, kindly select File > Save a Copy in Drive from the Colab menu and run the notebook from there. You will need a Gmail account to save a copy.

Download and prepare the dataset

Let us first load the [Fashion MNIST dataset]() into your workspace. You will use this to train a machine learning model that classifies images of clothing.

```
# Import keras
```

```
from tensorflow import keras
```

```
# Download the dataset and split into train and test sets
(img_train, label_train), (img_test, label_test) = keras.datasets.fashion_mnist
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
32768/29515 [=================================] - 0s 0us/step
40960/29515 [=========================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
26427392/26421880 [==============================] - 0s 0us/step
26435584/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
16384/5148 [=================================================================
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
4423680/4422102 [==============================] - 0s 0us/step
4431872/4422102 [==============================] - 0s 0us/step
```

For preprocessing, you will normalize the pixel values to make the training converge faster.

```
# Normalize pixel values between 0 and 1
img_train = img_train.astype('float32') / 255.0
img_test = img_test.astype('float32') / 255.0
```

Baseline Performance

As mentioned, you will first have a baseline performance using arbitrarily handpicked parameters so you can compare the results later. In the interest of time and resource limits provided by Colab, you will just build a shallow dense neural network (DNN) as shown below. This is to demonstrate the concepts without involving huge datasets and long tuning and training times. As you'll see later, even small models can take some time to tune. You can extend the concepts here when you get to build more complex models in your own projects.

```
# Build the baseline model using the Sequential API
b_model = keras.Sequential()
b_model.add(keras.layers.Flatten(input_shape=(28, 28)))
b_model.add(keras.layers.Dense(units=512, activation='relu', name='dense_1')) #
b_model.add(keras.layers.Dropout(0.2))
b_model.add(keras.layers.Dense(10, activation='softmax'))

# Print model summary
b_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense_1 (Dense)             (None, 512)               401920
```

```
 dropout (Dropout)           (None, 512)               0

 dense (Dense)               (None, 10)                5130

=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
```

As shown, we hardcoded all the hyperparameters when declaring the layers. These include the number of hidden units, activation, and dropout. You will see how you can automatically tune some of these a bit later.

Let's then setup the loss, metrics, and the optimizer. The learning rate is also a hyperparameter you can tune automatically but for now, let's set it at `0.001`.

```
# Setup the training parameters
b_model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
            loss=keras.losses.SparseCategoricalCrossentropy(),
            metrics=['accuracy'])
```

With all settings set, you can start training the model. We've set the number of epochs to 10 but feel free to increase it if you have more time to go through the notebook.

```
# Number of training epochs.
NUM_EPOCHS = 10

# Train the model
b_model.fit(img_train, label_train, epochs=NUM_EPOCHS, validation_split=0.2)
```

```
Epoch 1/10
1500/1500 [==============================] - 7s 3ms/step - loss: 0.5159 - acc
Epoch 2/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3945 - acc
Epoch 3/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3587 - acc
Epoch 4/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3319 - acc
Epoch 5/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3183 - acc
Epoch 6/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3055 - acc
Epoch 7/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2911 - acc
Epoch 8/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2805 - acc
Epoch 9/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2753 - acc
Epoch 10/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2636 - acc
<keras.callbacks.History at 0x7fb8e002bf50>
```

Finally, you want to see how this baseline model performs against the test set.

```
# Evaluate model on the test set
b_eval_dict = b_model.evaluate(img_test, label_test, return_dict=True)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.3503 - accur
```

Let's define a helper function for displaying the results so it's easier to compare later.

```
# Define helper function
def print_results(model, model_name, eval_dict):
  '''
  Prints the values of the hyparameters to tune, and the results of model evalu

  Args:
    model (Model) - Keras model to evaluate
    model_name (string) - arbitrary string to be used in identifying the model
    eval_dict (dict) -  results of model.evaluate
  '''
  print(f'\n{model_name}:')

  print(f'number of units in 1st Dense layer: {model.get_layer("dense_1").units
  print(f'learning rate for the optimizer: {model.optimizer.lr.numpy()}')

  for key,value in eval_dict.items():
    print(f'{key}: {value}')

# Print results for baseline model
print_results(b_model, 'BASELINE MODEL', b_eval_dict)
```

```
BASELINE MODEL:
number of units in 1st Dense layer: 512
learning rate for the optimizer: 0.0010000000474974513
loss: 0.35033318400382996
accuracy: 0.878000020980835
```

That's it for getting the results for a single set of hyperparameters. As you can see, this process can be tedious if you want to try different sets of parameters. For example, will your model improve if you use `learning_rate=0.00001` and `units=128`? What if `0.001` paired with `256`? The process will be even more difficult if you decide to also tune the dropout and try out other activation functions as well. Keras Tuner solves this problem by having an API to automatically search for the optimal set. You will just need to set it up once then wait for the results. You will see how this is done in the next sections.

Keras Tuner

To perform hypertuning with Keras Tuner, you will need to:

- Define the model
- Select which hyperparameters to tune
- Define its search space
- Define the search strategy

Install and import packages

You will start by installing and importing the required packages.

```
# Install Keras Tuner
!pip install -q -U keras-tuner
```

```
     |████████████████████████████████| 135 kB 4.7 MB/s
```

```
# Import required packages
import tensorflow as tf
import kerastuner as kt
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:3: DeprecationWa
  This is separate from the ipykernel package so we can avoid doing imports u
```

Define the model

The model you set up for hypertuning is called a *hypermodel*. When you build this model, you define the hyperparameter search space in addition to the model architecture.

You can define a hypermodel through two approaches:

- By using a model builder function
- By [subclassing the `HyperModel` class] of the Keras Tuner API

In this lab, you will take the first approach: you will use a model builder function to define the image classification model. This function returns a compiled model and uses hyperparameters you define inline to hypertune the model.

The function below basically builds the same model you used earlier. The difference is there are two hyperparameters that are setup for tuning:

- the number of hidden units of the first Dense layer
- the learning rate of the Adam optimizer

You will see that this is done with a HyperParameters object which configures the hyperparameter you'd like to tune. For this exercise, you will:

- use its `Int()` method to define the search space for the Dense units. This allows you to set a minimum and maximum value, as well as the step size when incrementing between these values.
- use its `Choice()` method for the learning rate. This allows you to define discrete values to include in the search space when hypertuning.

You can view all available methods and its sample usage in the official documentation

```
def model_builder(hp):
  '''
  Builds the model and sets up the hyperparameters to tune.

  Args:
    hp - Keras tuner object

  Returns:
    model with hyperparameters to tune
  '''

  # Initialize the Sequential API and start stacking the layers
  model = keras.Sequential()
  model.add(keras.layers.Flatten(input_shape=(28, 28)))

  # Tune the number of units in the first Dense layer
  # Choose an optimal value between 32-512
  hp_units = hp.Int('units', min_value=32, max_value=512, step=32)
  model.add(keras.layers.Dense(units=hp_units, activation='relu', name='dense_1

  # Add next layers
  model.add(keras.layers.Dropout(0.2))
  model.add(keras.layers.Dense(10, activation='softmax'))

  # Tune the learning rate for the optimizer
  # Choose an optimal value from 0.01, 0.001, or 0.0001
  hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])

  model.compile(optimizer=keras.optimizers.Adam(learning_rate=hp_learning_rate)
                loss=keras.losses.SparseCategoricalCrossentropy(),
                metrics=['accuracy'])

  return model
```

Instantiate the Tuner and perform hypertuning

Now that you have the model builder, you can then define how the tuner can find the optimal set of hyperparameters, also called the search strategy. Keras Tuner has [four tuners](#) available with built-in strategies - `RandomSearch`, `Hyperband`, `BayesianOptimization`, and `Sklearn`.

In this tutorial, you will use the Hyperband tuner. Hyperband is an algorithm specifically developed for hyperparameter optimization. It uses adaptive resource allocation and early-stopping to quickly converge on a high-performing model. This is done using a sports championship style bracket wherein the algorithm trains a large number of models for a few epochs and carries forward only the top-performing half of models to the next round. You can read about the intuition behind the algorithm in section 3 of this paper.

Hyperband determines the number of models to train in a bracket by computing 1 + $\log_{\texttt{factor}}(\texttt{max\_epochs})$ and rounding it up to the nearest integer. You will see these parameters (i.e. `factor` and `max_epochs` passed into the initializer below). In addition, you will also need to define the following to instantiate the Hyperband tuner:

- the hypermodel (built by your model builder function)
- the `objective` to optimize (e.g. validation accuracy)
- a `directory` to save logs and checkpoints for every trial (model configuration) run during the hyperparameter search. If you re-run the hyperparameter search, the Keras Tuner uses the existing state from these logs to resume the search. To disable this behavior, pass an additional `overwrite=True` argument while instantiating the tuner.
- the `project_name` to differentiate with other runs. This will be used as a subdirectory name under the `directory`.

You can refer to the documentation for other arguments you can pass in.

```
# Instantiate the tuner
tuner = kt.Hyperband(model_builder,
                     objective='val_accuracy',
                     max_epochs=10,
                     factor=3,
                     directory='kt_dir',
                     project_name='kt_hyperband')
```

Let's see a summary of the hyperparameters that you will tune:

```
# Display hypertuning settings
tuner.search_space_summary()
```

```
Search space summary
Default search space size: 2
units (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 512, 'step'
learning_rate (Choice)
{'default': 0.01, 'conditions': [], 'values': [0.01, 0.001, 0.0001], 'orderec
```

You can pass in a callback to stop training early when a metric is not improving. Below, we define an EarlyStopping callback to monitor the validation loss and stop training if it's not improving after 5 epochs.

```
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
```

You will now run the hyperparameter search. The arguments for the search method are the same as those used for `tf.keras.model.fit` in addition to the callback above. This will take around 10 minutes to run.

```
# Perform hypertuning
tuner.search(img_train, label_train, epochs=NUM_EPOCHS, validation_split=0.2, c
```

```
Trial 30 Complete [00h 00m 46s]
val_accuracy: 0.8854166865348816

Best val_accuracy So Far: 0.8872500061988831
Total elapsed time: 00h 12m 03s
INFO:tensorflow:Oracle triggered exit
INFO:tensorflow:Oracle triggered exit
```

You can get the top performing model with the get_best_hyperparameters() method.

```
# Get the optimal hyperparameters from the results
best_hps=tuner.get_best_hyperparameters()[0]

print(f"""
The hyperparameter search is complete. The optimal number of units in the first
layer is {best_hps.get('units')} and the optimal learning rate for the optimize
is {best_hps.get('learning_rate')}.
""")

best_model = tuner.get_best_models()[0]
```

```
The hyperparameter search is complete. The optimal number of units in the fir
layer is 192 and the optimal learning rate for the optimizer
is 0.001.
```

Build and train the model

Now that you have the best set of hyperparameters, you can rebuild the hypermodel with these values and retrain it.

```
# Build the model with the optimal hyperparameters
h_model = tuner.hypermodel.build(best_hps)
h_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 784)               0
```

```
 dense_1_1 (Dense)          (None, 192)               150720

 dropout_1 (Dropout)        (None, 192)               0

 dense_1 (Dense)            (None, 10)                1930

=================================================================
Total params: 152,650
Trainable params: 152,650
Non-trainable params: 0
_________________________________________________________________
```

```
# Train the hypertuned model
h_model.fit(img_train, label_train, epochs=NUM_EPOCHS, validation_split=0.2)
```

```
Epoch 1/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.5370 - acc
Epoch 2/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.4007 - acc
Epoch 3/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3673 - acc
Epoch 4/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3420 - acc
Epoch 5/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3281 - acc
Epoch 6/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3146 - acc
Epoch 7/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3026 - acc
Epoch 8/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2920 - acc
Epoch 9/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2830 - acc
Epoch 10/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2732 - acc
<keras.callbacks.History at 0x7fb84d6808d0>
```

You will then get its performance against the test set.

```
# Evaluate the hypertuned model against the test set
h_eval_dict = h_model.evaluate(img_test, label_test, return_dict=True)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.3471 - accur
```

We can compare the results we got with the baseline model we used at the start of the notebook. Results may vary but you will usually get a model that has less units in the dense layer, while having comparable loss and accuracy. This indicates that you reduced the model size and saved compute resources while still having more or less the same accuracy.

```
# Print results of the baseline and hypertuned model
print_results(b_model, 'BASELINE MODEL', b_eval_dict)
print_results(h_model, 'HYPERTUNED MODEL', h_eval_dict)
```

```
3 print_results(h_model, 'HYPERTUNED MODEL', h_eval_dict)
```

```
BASELINE MODEL:
number of units in 1st Dense layer: 512
learning rate for the optimizer: 0.0010000000474974513
loss: 0.35033318400382996
accuracy: 0.878000020980835

HYPERTUNED MODEL:
number of units in 1st Dense layer: 10
learning rate for the optimizer: 0.0010000000474974513
loss: 0.34706664085388184
accuracy: 0.8773000240325928
```

Bonus Challenges (optional)

If you want to keep practicing with Keras Tuner in this notebook, you can do a factory reset (`Runtime > Factory reset runtime`) and take on any of the following:

- hypertune the dropout layer with `hp.Float()` or `hp.Choice()`
- hypertune the activation function of the 1st dense layer with `hp.Choice()`
- determine the optimal number of Dense layers you can add to improve the model. You can use the code here as reference.
- explore pre-defined `HyperModel` classes - HyperXception and HyperResNet for computer vision applications.

Wrap Up

In this tutorial, you used Keras Tuner to conveniently tune hyperparameters. You defined which ones to tune, the search space, and search strategy to arrive at the optimal set of hyperparameters. These concepts will again be discussed in the next sections but in the context of AutoML, a package that automates the entire machine learning pipeline. On to the next!

0 s Abgeschlossen um 09:11