Hyperparameter Tuning with KerasTuner and TensorFlow

Understand best practices to optimize your model's architecture and hyperparameters with KerasTuner and TensorFlow

Image by Zoltan Tasi on upslash

Building machine learning models is an iterative process that involves optimizing the model's performance and compute resources. The settings that you adjust during each iteration are called *hyperparameters*. They govern the training process and are held constant during training.

The process of searching for optimal hyperparameters is called *hyperparameter tuning* or *hypertuning*, and is essential in any machine learning project. Hypertuning helps boost performance and reduces model complexity by removing unnecessary parameters (e.g., number of units in a dense layer).

There are two types of hyperparameters:

1. *Model hyperparameters* that influence model architecture (e.g., number and width of hidden layers in a DNN)
2. *Algorithm hyperparameters* that influence the speed and quality of training (e.g., learning rate and activation function).

The number of hyperparameter combinations, even in a shallow DNN, can grow insanely large causing a manual search for an optimal set simply not feasible nor scalable.

This post will introduce you to Keras Tuner, a library made to automate the hyperparameter search. We'll build and compare the results of three deep learning models trained on the Fashion MNIST dataset:

- Baseline model with pre-selected hyperparameters
- Optimized hyperparameters with Hyperband algorithm
- Tuned ResNet architecture with Bayesian Optimization

You can view the jupyter notebook here.

Imports and Preprocessing

Let us first import the required modules and print their versions in case you want to reproduce the notebook. We are using TensorFlow version 2.5.0 and KerasTuner version 1.0.1.

```
import tensorflow as tf
import kerastuner as kt

from tensorflow import keras

print(f"TensorFlow Version: {tf.__version__}")
print(f"KerasTuner Version: {kt.__version__}")
```

```
>>> TensorFlow Version: 2.5.0
>>> KerasTuner Version: 1.0.1
```

We'll begin by loading in the Fashion MNIST dataset. The goal is to train a machine learning model to classify different images of clothing.

Since the images are greyscale, which means each pixel value represents a number between 1 and 255, we can divide each pixel by 255 to normalize the values between 0 and 1. This will make training converge faster.

Baseline Model

As mentioned, we will first train a shallow dense neural network (DNN) with preselected hyperparameters giving us a baseline performance. We'll see later how simple models, like this shallow DNN, can take some time to tune.

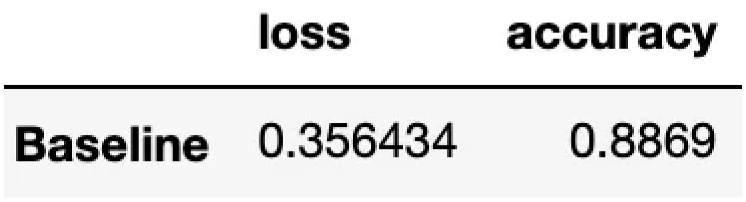
Notice how we hardcode all of the hyperparameters in the code above. These include the number of hidden layers (in our case there is 1 hidden layer), the number of units in our hidden layer (512), its activation function (ReLu), and the dropout percentage (0.2). We'll tune all of these hyperparameters in the next section.

Let’s setup the optimizer, loss, and metrics. One more hyperparameter we’ll tune later on is the learning rate, but for now we’ll set it equal to 0.001.

With our model’s settings defined, we are ready to train! We’ll set the number of epochs to 20 and use early stopping to interrupt training if there is no performance gained after 5 epochs.

Finally, we’ll want to see how our model performs against the test set so we’ll define a helper function to easily display the results.

Baseline 0.356434 0.8869



	loss	accuracy
Baseline	0.356434	0.8869

Figure 1. Baseline evaluation results | Image by author

There's the results for a single set of hyperparameters. Imagine trying out different learning rates, dropout percentages, number of hidden layers, and number of neurons in each hidden layer. As you can see, manual hypertuning is simply not feasible nor scalable. In the next section you'll see how Keras Tuner solves these problems simply by automating the process and searching the hyperparameter space in an efficient way.

Keras Tuner

Keras Tuner is a simple, distributable hyperparameter optimization framework that automates the painful process of manually searching for optimal hyperparameters. Keras Tuner comes with Random Search, Hyperband, and Bayesian Optimization built-in search algorithms, and is designed to fit many use cases including:

- Distributed tuning
- Custom training loops (e.g., GANs, reinforcement learning, etc.)
- Adding hyperparameters outside of the model building function (preprocessing, data augmentation, test time augmentation, etc.)

These processes are outside the scope of this write-up, but feel free to read more in the official documentation.

There are four steps to hypertune our shallow DNN using Keras Tuner:

1. Define the model
2. Specify which hyperparameters to tune
3. Define the search space
4. Define the search algorithm

Define the Model

The model we set up for hypertuning is called a *hypermodel*. We define the hyperparameter search space when we build our hypermodel.

There are two ways to build a hypermodel:

1. By using a model builder function
2. Using a HyperModel subclass of the Keras Tuner API

We will be using the first approach to define our DNN in the model building function. You'll Notice how the hyperparameters are defined inline. Our model building function uses the defined hyperparameters to return a compiled model.

Specify Which Hyperparameters to Tune

The model we'll be building is very similar to the shallow DNN we trained earlier, except we'll be tuning four of the model's hyperparameters:

- The number of hidden layers
- The number of units in each hidden layer
- The dropout percentage after each hidden layer
- The learning rate of the Adam optimizer

Define the Search Space

This is done by passing a HyperParameters object as a parameter to the model building function that configures the hyperparameters you'd like to tune. In our function we will use:

- hp.Int() to define the search space for the number of hidden layers and units in each hidden layer. This allows you to define minimum and maximum values, as well as a step size to increment by.
- hp.Float() to define the search space for the dropout percentage. This is similar to hp.Int() except it takes floating values.
- hp.Choice() to define the search space of the learning rate. This allows you to define discrete values.

For more information on all the available methods and their usage visit the official documentation.

Define the Search Algorithm

After building our model builder function, we can instantiate the tuner and specify a search strategy. For our use case we will use the Hyperband algorithm. Hyperband is a novel bandit-based approach made specifically for hyperparameter optimization. The research paper was published in 2018 and details a process that quickly converges on a high-performing model through adaptive resource-allocation and early-stopping.

The idea is simple, Hyperband uses a sports championship style bracket and begins by randomly selecting a large number of models with random hyperparameter permutations from the search space. Each model is trained for a few epochs and only the top-performing half of models moves on to the next round.

To instantiate our tuner, we will need to define the following hyperparameters:

- Our hypermodel (built by our model builder function)
- The objective (the direction (min or max) will be automatically inferred for built-in metrics — for custom metrics we can use kerastuner.Objective)
- Factor and max_epochs are used to calculate the amount of models in each bracket by taking the log base factor of max_epochs + 1. This number is rounded up to the nearest integer.
- Hyperband iterations is used to control the resource budget you're willing to allocate to hypertuning. Hyperband iterations is the number of times you iterate over the entire search algorithm.
- Directory saves logs and checkpoints for each trial run during the hyperparameter search allowing us to pick up the search where we last left off. You can disable this behavior by setting an additional hyperparameter 'overwrite' = True.
- Project_name is used to differentiate with other runs and is a subdirectory under directory.

Please refer to the official documentation for a list of all available arguments.

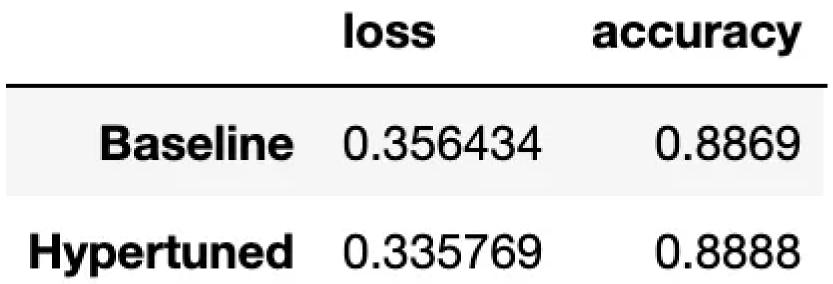
We can see the search space summary with:

We can set callbacks like early stopping to stop training early when metrics aren't improving.

Let's start the search.

After the search is finished, we can get the best hyperparameters and retrain the model.

And then we'll evaluate our hypertuned model on the test set!



	loss	accuracy
Baseline	0.356434	0.8869
Hypertuned	0.335769	0.8888

Figure 2. Hypertuned model evaluation | Image by author

Through hypertuning we found an architecture with 100,000 less parameters than our baseline model that performs slightly better on the test set. If we performed more iterations of the hyperband algorithm, we'd most likely find an even better performing architecture.

HyperResnet

In addition to defining our own hypermodels, Keras Tuner provides two predefined tunable models, HyperXception and HyperResnet. These models search over the following architectures and hyperparameters:

- The version of the model
- Depth of convolutional layers
- Pooling
- Learning rate
- Optimization algorithm

Let’s see how we can use HyperXception or HyperResnet with our tuner.

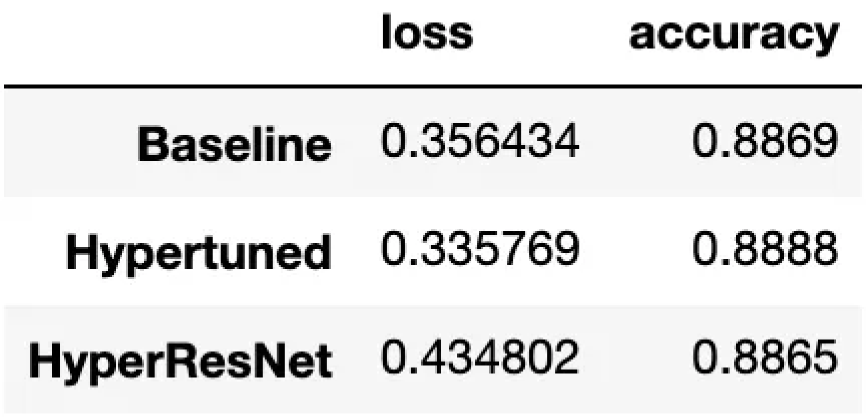
We specify the input shape and number of classes in our HyperResnet model. This time we’ll use Bayesian Optimization as our search algorithm, which searches the hyperparameter space by focusing on more promising regions. Also, we use a different project_name so our tuner can differentiate between the run from earlier.

Next, we have to preprocess our data to match HyperResnet’s requirements. HyperResnet expects features to be the same shape as a convolutional layer and one-hot encoded labels.

With the following small block of code, we can begin the search.

Same as above we can get the best hyperparameters and retrain the model.

Finally, we’ll evaluate the model on the test set.



	loss	accuracy
Baseline	0.356434	0.8869
Hypertuned	0.335769	0.8888
HyperResNet	0.434802	0.8865

Figure 3. HyperResNet model evaluation | Image by author

In a production setting, we'd have to consider more than just test set accuracy. The best deployment option is our Hypertuned DNN because it performs slightly better on the test set and has the least amount of parameters.

Wrap-Up

Hypertuning is an essential part of a machine learning pipeline. In this post, we trained a baseline model showing why manual searching for optimal hyperparameters is hard. We explored Keras Tuner in-depth and how it is used to automate the hyperparameter search. Finally, we hypertuned a predefined HyperResnet model.