Automated machine learning (**AutoML**) provides methods to find the optimal neural architecture and the best hyperparameter settings for a given neural network. We have already covered neural architecture search in detail while discussing the `RandWireNN` model in *Chapter 5*, *Hybrid Advanced Models*.

In this chapter, we will look more broadly at the AutoML tool for PyTorch—**Auto-PyTorch**—which performs both neural architecture search and hyperparameter search. We will also look at another AutoML tool called **Optuna** that performs hyperparameter search for a PyTorch model.

At the end of this chapter, non-experts will be able to design machine learning models with little domain experience, and experts will drastically speed up their model selection process.

This chapter is broken down into the following topics:

- Finding the best neural architectures with AutoML
- Using Optuna for hyperparameter search

Finding the best neural architectures with AutoML

One way to think of machine learning algorithms is that they automate the process of learning relationships between given inputs and outputs. In traditional software engineering, we would have to explicitly write/code these relationships in the form of functions that take in input and return output. In the machine learning world, machine learning models find such functions for us. Although we automate to a certain extent, there is still a lot to be done. Besides mining and cleaning data, here are a few routine tasks to be performed in order to get those functions:

Choosing a machine learning model (or a model family and then a model)

Deciding the model architecture (especially in the case of deep learning)

Choosing hyperparameters

Adjusting hyperparameters based on validation set performance

Trying different models (or model families)

These are the kinds of tasks that justify the requirement of a human machine learning expert. Most of these steps are manual and either take a lot of time or need a lot of expertise to discount the required time, and we have far fewer machine learning experts than needed to create and deploy machine learning models that are increasingly popular, valuable, and useful across both industries and academia.

This is where AutoML comes to the rescue. AutoML has become a discipline within the field of machine learning that aims to automate the previously listed steps and beyond.

In this section, we will take a look at Auto-PyTorch—an AutoML tool created to work with PyTorch. In the form of an exercise, we will find an optimal neural network along with the hyperparameters to perform handwritten digit classification—a task that we worked on in *Chapter 1*, *Overview of Deep Learning Using PyTorch*.

The difference from the first chapter will be that this time, we do not decide the architecture or the hyperparameters, and instead let Auto-PyTorch figure that out for us. We will first load the dataset, then define an Auto-PyTorch model search instance,

and finally run the model searching routine, which will provide us with a best-performing model.

Tool citation
Auto-PyTorch [16.1] *Auto-PyTorch Tabular: Multi-Fidelity MetaLearning for Efficient and Robust AutoDL*, *Lucas Zimmer*, *Marius Lindauer*, and *Frank Hutter [16.2]*

Using Auto-PyTorch for optimal MNIST model search

We will execute the model search in the form of a Jupyter Notebook. In the text, we only show the important parts of the code. The full code can be found in our github repository [16.3]

Loading the MNIST dataset

We will now discuss the code for loading the dataset step by step, as follows:

1. First, we import the relevant libraries, like this:

```
import torch
from autoPyTorch import AutoNetClassification
```

The last line is crucial, as we import the relevant Auto-PyTorch module here. This will help us set up and execute a model search session.

1. Next, we load the training and test datasets using Torch **application programming interfaces (APIs)**, as follows:

```
train_ds = datasets.MNIST(...)
test_ds = datasets.MNIST(...)
```

1. We then convert these dataset tensors into training and testing input (X) and output (y) arrays, like this:

```
X_train, X_test, y_train, y_test = train_ds.data.numpy().reshape(-1, 28*28),
test_ds.data.numpy().reshape(-1, 28*28) ,train_ds.targets.numpy(),
test_ds.targets.numpy()
```

Note that we are reshaping the images into flattened vectors of size 784. In the next section, we will be defining an Auto-PyTorch model searcher that expects a flattened feature vector as input, and hence we do the reshaping.

Auto-PyTorch currently (at the time of writing) only provides support for featurized and image data in the form of `AutoNetClassification` and `AutoNetImageClassification` respectively. While we are using featurized data in this exercise, we leave it as an exercise for the reader to use image data instead[16.4] .

Running a neural architecture search with Auto-PyTorch

Having loaded the dataset in the preceding section, we will now use Auto-PyTorch to define a model search instance and use it to perform the tasks of neural architecture search and hyperparameter search. We'll proceed as follows:

1. This is the most important step of the exercise, where we define an `autoPyTorch` model search instance, like this:

```
autoPyTorch = AutoNetClassification("tiny_cs",  # config preset
            log_level='info', max_runtime=2000, min_budget=100, max_budget=1500)
```

The configs here are derived from the examples provided in the Auto-PyTorch repository [16.5] . But generally, `tiny_cs` is used for faster searches with fewer hardware requirements.

The budget argument is all about setting constraints on resource consumption by the Auto-PyTorch process. As a default, the unit of a budget is time—that is, how much **central processing unit/graphics processing unit** (**CPU/GPU**) time we are comfortable spending on the model search.

1. After instantiating an Auto-PyTorch model search instance, we execute the search by trying to fit the instance on the training dataset, as follows:

```
autoPyTorch.fit(X_train, y_train, validation_split=0.1)
```

Internally, Auto-PyTorch will run several `trials` of different model architectures and hyperparameter settings based on methods mentioned in the original paper [16.2] .

The different `trials` will be benchmarked against the 10% validation dataset, and the best-performing `trial` will be returned as output. The command in the preceding code snippet should output the following:

```
{'optimized_hyperparameter_config': {'CreateDataLoader:batch_size': 125,
  'Imputation:strategy': 'median',
  'InitializationSelector:initialization_method': 'default',
  'InitializationSelector:initializer:initialize_bias': 'No',
  'LearningrateSchedulerSelector:lr_scheduler': 'cosine_annealing',
  'LossModuleSelector:loss_module': 'cross_entropy_weighted',
  'NetworkSelector:network': 'shapedresnet',
  'NormalizationStrategySelector:normalization_strategy': 'standardize',
  'OptimizerSelector:optimizer': 'sgd',
  'PreprocessorSelector:preprocessor': 'truncated_svd',
  'ResamplingStrategySelector:over_sampling_method': 'none',
  'ResamplingStrategySelector:target_size_strategy': 'none',
  'ResamplingStrategySelector:under_sampling_method': 'none',
  'TrainNode:batch_loss_computation_technique': 'standard',
  'LearningrateSchedulerSelector:cosine_annealing:T_max': 10,
  'LearningrateSchedulerSelector:cosine_annealing:eta_min': 2,
  'NetworkSelector:shapedresnet:activation': 'relu',
  'NetworkSelector:shapedresnet:blocks_per_group': 4,
  'NetworkSelector:shapedresnet:max_units': 13,
  'NetworkSelector:shapedresnet:num_groups': 2,
  'NetworkSelector:shapedresnet:resnet_shape': 'brick',
  'NetworkSelector:shapedresnet:use_dropout': 0,
  'NetworkSelector:shapedresnet:use_shake_drop': 0,
  'NetworkSelector:shapedresnet:use_shake_shake': 0,
  'OptimizerSelector:sgd:learning_rate': 0.0682914696764946 5,
  'OptimizerSelector:sgd:momentum': 0.9343847098348538,
  'OptimizerSelector:sgd:weight_decay': 0.0002425066735211845,
  'PreprocessorSelector:truncated_svd:target_dim': 100},
 'budget': 40.0,
 'loss': -96.45,
 'info': {'loss': 0.12337125303244502,
  'model_parameters': 176110.0,
  'train_accuracy': 96.28550185873605,
  'lr_scheduler_converged': 0.0,
  'lr': 0.0682914696764946 5,
  'val_accuracy': 96.45}}
```

Figure 16 .1 – Auto-PyTorch model accuracy

Figure 16 .1 basically shows the hyperparameter setting that Auto-PyTorch finds optimal for the given task—for example, the learning rate is `0.068`, momentum is `0.934`, and so on. The preceding screenshot also shows the training and validation set accuracy for the chosen optimal model configuration.

1. Having converged to an optimal trained model, we can now make predictions on our test set using that model, as follows:

```
y_pred = autoPyTorch.predict(X_test)print("Accuracy score", np.mean(y_pred.reshape(-1)
== y_test))
```

It should output something like this:

```
Accuracy score 0.964
```

Figure 16 .2 – Auto-PyTorch model accuracy

As we can see, we have obtained a model with a decent test-set performance of 96.4%. For context, a random choice on this task would lead to a performance rate of 10%. We have obtained this good performance without defining either the model architecture or the hyperparameters. Upon setting a higher budget, a more extensive search could lead to an even better performance.

Also, the performance will vary based on the hardware (machine) on which the search is being performed. Hardware with more compute power and memory can run more searches in the same time budget, and hence can lead to a better performance.

Visualizing the optimal AutoML model

In this section, we will look at the best-performing model that we have obtained by running the model search routine in the previous section. We'll proceed as follows:

1. Having already looked at the hyperparameters in the preceding section, let's look at the optimal model architecture that Auto-PyTorch has devised for us, as follows:

```
pytorch_model = autoPyTorch.get_pytorch_model()
print(pytorch_model)
```

It should output something like this:

```
pytorch_model = autoPyTorch.get_pytorch_model()
print(pytorch_model)
```

```
Sequential(
  (0): Linear(in_features=100, out_features=100, bias=True)
  (1): Sequential(
    (0): ResBlock(
      (layers): Sequential(
        (0): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): ReLU()
        (2): Linear(in_features=100, out_features=100, bias=True)
        (3): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (4): ReLU()
        (5): Linear(in_features=100, out_features=100, bias=True)
      )
    )
    (1): ResBlock(
      (layers): Sequential(
        (0): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): ReLU()
        (2): Linear(in_features=100, out_features=100, bias=True)
        (3): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (4): ReLU()
        (5): Linear(in_features=100, out_features=100, bias=True)
      )
    )
    (2): ResBlock(
      (layers): Sequential(
        (0): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): ReLU()
        (2): Linear(in_features=100, out_features=100, bias=True)
        (3): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (4): ReLU()
        (5): Linear(in_features=100, out_features=100, bias=True)
      )
    )
    (3): ResBlock(
                                                    .
                                                    .
                                                    .
    )
    (3): ResBlock(
      (layers): Sequential(
        (0): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): ReLU()
        (2): Linear(in_features=100, out_features=100, bias=True)
        (3): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (4): ReLU()
        (5): Linear(in_features=100, out_features=100, bias=True)
      )
    )
    (3): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): ReLU()
    (5): Linear(in_features=100, out_features=10, bias=True)
  )
```

Figure 16 .3 – Auto-PyTorch model architecture

The model consists of some structured residual blocks containing fully connected layers, batch normalization layers, and ReLU activations. At the end, we see a final fully connected layer with 10 outputs—one for each digit from 0 to 9.

1. We can also visualize the actual model graph using `torchviz`, as shown in the next code snippet:

Copy Explain

```
x = torch.randn(1, pytorch_model[0].in_features)
y = pytorch_model(x)
arch = make_dot(y.mean(), params=dict(pytorch_model.named_parameters()))
arch.format="pdf"
arch.filename = "convnet_arch"
arch.render(view=False)
```

This should save a `convnet_arch.pdf` file in the current working directory, which should look like this upon opening:

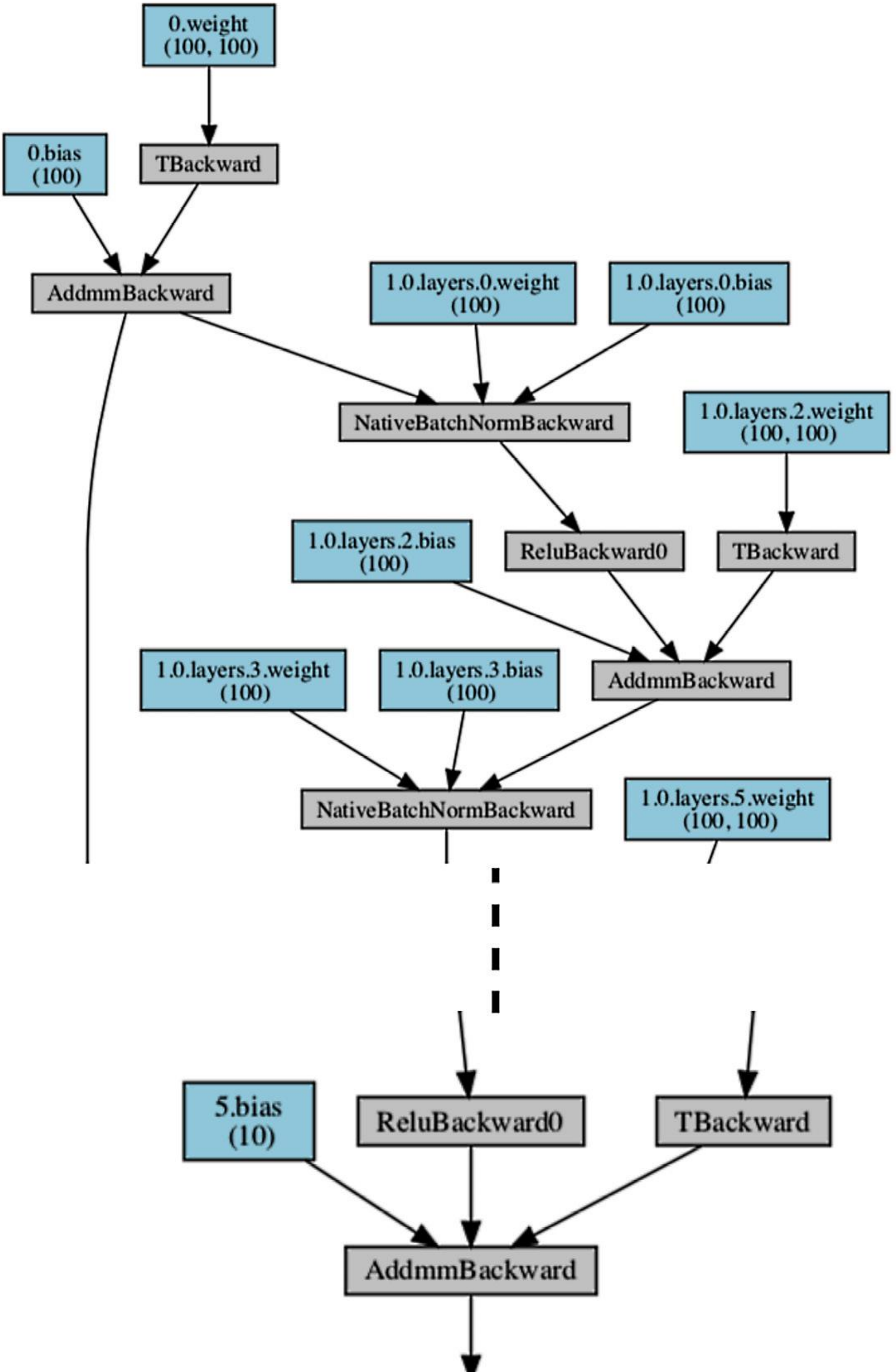
0.weight
(100, 100)
0.bias
(100)
TBackward
AddmmBackward
1.0.layers.0.weight
(100)
1.0.layers.0.bias
(100)
NativeBatchNormBackward
1.0.layers.2.weight
(100, 100)
1.0.layers.2.bias
(100)
ReluBackward0
TBackward
1.0.layers.3.weight
(100)
1.0.layers.3.bias
(100)
AddmmBackward
NativeBatchNormBackward
1.0.layers.5.weight
(100, 100)
5.bias
(10)
ReluBackward0
TBackward
AddmmBackward

Figure 16 .4 – Auto-PyTorch model diagram

1. To peek into how the model converged to this solution, we can look at the search space that was used during the model-finding process with the following code:

```
autoPyTorch.get_hyperparameter_search_space()
```

This should output the following:

```
Configuration space object:
  Hyperparameters:
    CreateDataLoader:batch_size, Type: Constant, Value: 125
    Imputation:strategy, Type: Categorical, Choices: {median}, Default: median
    InitializationSelector:initialization_method, Type: Categorical, Choices: {default}, Default: default
    InitializationSelector:initializer:initialize_bias, Type: Constant, Value: No
    LearningrateSchedulerSelector:cosine_annealing:T_max, Type: Constant, Value: 10
    LearningrateSchedulerSelector:cosine_annealing:eta_min, Type: Constant, Value: 2
    LearningrateSchedulerSelector:lr_scheduler, Type: Categorical, Choices: {cosine_annealing}, Default: cosine_annea
ling
    LossModuleSelector:loss_module, Type: Categorical, Choices: {cross_entropy_weighted}, Default: cross_entropy_weig
hted
    NetworkSelector:network, Type: Categorical, Choices: {shapedresnet}, Default: shapedresnet
    NetworkSelector:shapedresnet:activation, Type: Constant, Value: relu
    NetworkSelector:shapedresnet:blocks_per_group, Type: UniformInteger, Range: [1, 4], Default: 2
    NetworkSelector:shapedresnet:max_units, Type: UniformInteger, Range: [10, 1024], Default: 101, on log-scale
    NetworkSelector:shapedresnet:num_groups, Type: UniformInteger, Range: [1, 9], Default: 5
    NetworkSelector:shapedresnet:resnet_shape, Type: Constant, Value: brick
    NetworkSelector:shapedresnet:use_dropout, Type: Constant, Value: 0
    NetworkSelector:shapedresnet:use_shake_drop, Type: Constant, Value: 0
    NetworkSelector:shapedresnet:use_shake_shake, Type: Constant, Value: 0
    NormalizationStrategySelector:normalization_strategy, Type: Categorical, Choices: {standardize}, Default: standar
dize
    OptimizerSelector:optimizer, Type: Categorical, Choices: {sgd}, Default: sgd
    OptimizerSelector:sgd:learning_rate, Type: UniformFloat, Range: [0.0001, 0.1], Default: 0.0031622777, on log-scal
e
    OptimizerSelector:sgd:momentum, Type: UniformFloat, Range: [0.1, 0.999], Default: 0.5495
    OptimizerSelector:sgd:weight_decay, Type: UniformFloat, Range: [1e-05, 0.1], Default: 0.050005
    PreprocessorSelector:preprocessor, Type: Categorical, Choices: {truncated_svd}, Default: truncated_svd
    PreprocessorSelector:truncated_svd:target_dim, Type: Constant, Value: 100
    ResamplingStrategySelector:over_sampling_method, Type: Categorical, Choices: {none}, Default: none
    ResamplingStrategySelector:target_size_strategy, Type: Categorical, Choices: {none}, Default: none
    ResamplingStrategySelector:under_sampling_method, Type: Categorical, Choices: {none}, Default: none
    TrainNode:batch_loss_computation_technique, Type: Categorical, Choices: {standard}, Default: standard
  Conditions:
    LearningrateSchedulerSelector:cosine_annealing:T_max | LearningrateSchedulerSelector:lr_scheduler == 'cosine_anne
aling'
    LearningrateSchedulerSelector:cosine_annealing:eta_min | LearningrateSchedulerSelector:lr_scheduler == 'cosine_an
nealing'
    NetworkSelector:shapedresnet:activation | NetworkSelector:network == 'shapedresnet'
    NetworkSelector:shapedresnet:blocks_per_group | NetworkSelector:network == 'shapedresnet'
    NetworkSelector:shapedresnet:max_units | NetworkSelector:network == 'shapedresnet'
    NetworkSelector:shapedresnet:num_groups | NetworkSelector:network == 'shapedresnet'
    NetworkSelector:shapedresnet:resnet_shape | NetworkSelector:network == 'shapedresnet'
    NetworkSelector:shapedresnet:use_dropout | NetworkSelector:network == 'shapedresnet'
    NetworkSelector:shapedresnet:use_shake_drop | NetworkSelector:network == 'shapedresnet'
    NetworkSelector:shapedresnet:use_shake_shake | NetworkSelector:network == 'shapedresnet'
    OptimizerSelector:sgd:learning_rate | OptimizerSelector:optimizer == 'sgd'
    OptimizerSelector:sgd:momentum | OptimizerSelector:optimizer == 'sgd'
    OptimizerSelector:sgd:weight_decay | OptimizerSelector:optimizer == 'sgd'
    PreprocessorSelector:truncated_svd:target_dim | PreprocessorSelector:preprocessor == 'truncated_svd'
```

Figure 16 .5 – Auto-PyTorch model search space

It essentially lists the various ingredients required to build the model, with an allocated range per ingredient. For instance, the learning rate is allocated a range of **0.0001** to **0.1** and this space is sampled in a log scale—this is not linear but logarithmic sampling.

In *Figure 16 .1*, we have already seen the exact hyperparameter values that Auto-PyTorch samples from these ranges as optimal values for the given task. We can also alter these hyperparameter ranges manually, or even add more hyperparameters, using the `HyperparameterSearchSpaceUpdates` sub-module under the Auto-PyTorch module [16.6] .

This concludes our exploration of Auto-PyTorch—an AutoML tool for PyTorch. We successfully built an MNIST digit classification model using Auto-PyTorch, without specifying either the model architecture or the hyperparameters. This exercise will help you to get started with using this and other AutoML tools to build PyTorch models in an automated fashion. Some other similar tools are listed here - Hyperopt [16.7], Tune [16.8], Hypersearch [16.9], Skorcj [16.10], BoTorch [16.11] and Optuna [16.12]

While we cannot cover all of these tools in this chapter, in the next section we will discuss Optuna, which is a tool focused exclusively on finding an optimal set of hyperparameters and one that works well with PyTorch.

Using Optuna for hyperparameter search

Optuna is one of the hyperparameter search tools that supports PyTorch. You can read in detail about the search strategies used by the tool, such as **TPE (Tree-Structured Parzen Estimation)** and **CMA-ES (Covariance Matrix Adaptation Evolution Strategy)** in the *Optuna* paper [16.13] . Besides the advanced hyperparameter search methodologies, the tool provides a sleek API, which we will explore in a moment.

Tool citation
Optuna: A Next-Generation Hyperparameter Optimization Framework.
Takuya Akiba, *Shotaro Sano*, *Toshihiko Yanase*, *Takeru Ohta*, and *Masanori Koyama* (2019, in KDD).

In this section, we will once again build and train the `MNIST` model, this time using Optuna to figure out the optimal hyperparameter setting. We will discuss important parts of the code step by step, in the form of an exercise. The full code can be found in our github [16.14].

Defining the model architecture and loading dataset

First, we will define an Optuna-compliant model object. By Optuna-compliant, we mean adding APIs within the model definition code that are provided by Optuna to enable the parameterization of the model hyperparameters. To do this, we'll proceed as follows:

1. First, we import the necessary libraries, as follows:

```
import torch
import optuna
```

The `optuna` library will manage the hyperparameter search for us throughout the exercise.

1. Next, we define the model architecture. Because we want to be flexible with some of the hyperparameters—such as the number of layers and the number of units in each layer—we need to include some logic in the model definition code. So, first, we have declared that we need anywhere in between `1` to `4` convolutional layers and `1` to `2` fully connected layers thereafter, as illustrated in the following code snippet:

```
class ConvNet(nn.Module):
    def __init__(self, trial):
        super(ConvNet, self).__init__()
        num_conv_layers =  trial.suggest_int("num_conv_layers", 1, 4)
        num_fc_layers = trial.suggest_int("num_fc_layers", 1, 2)
```

1. We then successively append the convolutional layers, one by one. Each convolutional layer is instantly followed by a `ReLU` activation layer, and for each convolutional layer, we declare the depth of that layer to be between `16` and `64`.

The stride and padding are fixed to `3` and `True` respectively, and the whole convolutional block is then followed by a `MaxPool` layer, then a `Dropout` layer, with dropout probability ranging anywhere between `0.1` to `0.4` (another hyperparameter), as illustrated in the following code snippet:

```
        self.layers = []
        input_depth = 1 # grayscale image
        for i in range(num_conv_layers):
            output_depth = trial.suggest_int(f"conv_depth_{i}", 16, 64)
            self.layers.append(nn.Conv2d(input_depth, output_depth, 3, 1))
            self.layers.append(nn.ReLU())
            input_depth = output_depth
        self.layers.append(nn.MaxPool2d(2))
        p = trial.suggest_float(f"conv_dropout_{i}", 0.1, 0.4)
        self.layers.append(nn.Dropout(p))
        self.layers.append(nn.Flatten())
```

1. Next, we add a flattening layer so that fully connected layers can follow. We have to define a `_get_flatten_shape` function to derive the shape of the flattening layer output. We then successively add fully connected layers, where the number of units is declared to be between `16` and `64`. A `Dropout` layer follows each fully connected layer, again with the probability range of `0.1` to `0.4`.

Finally, we append a fixed fully connected layer that outputs `10` numbers (one for each class/digit), followed by a `LogSoftmax` layer. Having defined all the layers, we then instantiate our model object, as follows:

```
        input_feat = self._get_flatten_shape()
        for i in range(num_fc_layers):
            output_feat = trial.suggest_int(f"fc_output_feat_{i}", 16, 64)
            self.layers.append(nn.Linear(input_feat, output_feat))
            self.layers.append(nn.ReLU())
            p = trial.suggest_float(f"fc_dropout_{i}", 0.1, 0.4)
            self.layers.append(nn.Dropout(p))
            input_feat = output_feat
        self.layers.append(nn.Linear(input_feat, 10))
        self.layers.append(nn.LogSoftmax(dim=1))
        self.model = nn.Sequential(*self.layers)
    def _get_flatten_shape(self):
        conv_model = nn.Sequential(*self.layers)
        op_feat = conv_model(torch.rand(1, 1, 28, 28))
        n_size = op_feat.data.view(1, -1).size(1)
        return n_size
```

This model initialization function is conditioned on the `trial` object, which is facilitated by Optuna and which will decide the hyperparameter setting for our model. Finally, the `forward` method is quite straightforward, as can be seen in the following code snippet:

```
    def forward(self, x):
        return self.model(x)
```

Thus, we have defined our model object and we can now move on to loading the dataset.

1. The code for dataset loading is the same as in *Chapter 1, Overview of Deep Learning Using PyTorch* and is shown again in the following snippet:

```
train_dataloader = torch.utils.data.DataLoader(...)
test_dataloader = ...
```

In this section, we have successfully defined our parameterized model object as well as loaded the dataset. We will now define the model training and testing routines, along with the optimization schedule.

Defining the model training routine and optimization schedule

Model training itself involves hyperparameters such as optimizer, learning rate, and so on. In this part of the exercise, we will define the model training procedure while utilizing Optuna's parameterization capabilities. We'll proceed as follows:

1. First, we define the training routine. Once again, the code is the same as the training routine code we had for this model in the exercise found in *Chapter 1*, *Overview of Deep Learning Using PyTorch*, and is shown again here:

```
def train(model, device, train_dataloader, optim, epoch):
    for b_i, (X, y) in enumerate(train_dataloader):
        …
```

1. The model testing routine needs to be slightly augmented. To operate as per Optuna API requirements, the test routine needs to return a model performance metric—accuracy, in this case—so that Optuna can compare different hyperparameter settings based on this metric, as illustrated in the following code snippet:

```
def test(model, device, test_dataloader):
    with torch.no_grad():
        for X, y in test_dataloader:
            …
    accuracy = 100. * success/ len(test_dataloader.dataset)
    return accuracy
```

1. Previously, we would instantiate the model and the optimization function with the learning rate, and start the training loop outside of any function. But to follow the Optuna API requirements, we do all that under an `objective` function, which takes in the same `trial` object that was fed as an argument to the `__init__` method of our model object.

The `trial` object is needed here too because there are hyperparameters associated with deciding the learning rate value and choosing an optimizer, as illustrated in the following code snippet:

```
def objective(trial):
    model = ConvNet(trial)
    opt_name = trial.suggest_categorical("optimizer", ["Adam", "Adadelta", "RMSprop",
"SGD"])
    lr = trial.suggest_float("lr", 1e-1, 5e-1, log=True)
    optimizer = getattr(optim,opt_name)(model.parameters(), lr=lr)
    for epoch in range(1, 3):
        train(model, device, train_dataloader, optimizer, epoch)
        accuracy = test(model, device,test_dataloader)
        trial.report(accuracy, epoch)
        if trial.should_prune():
            raise optuna.exceptions.TrialPruned()
    return accuracy
```

For each epoch, we record the accuracy returned by the model testing routine. Additionally, at each epoch, we check if we will prune—that is, if we will skip—the current epoch. This is another feature offered by Optuna to speed up the hyperparameter search process so that we don't waste time on poor hyperparameter settings.

Running Optuna's hyperparameter search

In this final part of the exercise, we will instantiate what is called an **Optuna study** and, using the model definition and the training routine, we will execute Optuna's hyperparameter search process for the given model and the given dataset. We'll proceed as follows:

1. Having prepared all the necessary components in the preceding sections, we are ready to start the hyperparameter search process—something that is called a `study` in Optuna terminology. A `trial` is one hyperparameter-search iteration in a `study`. The code can be seen in the following snippet:

```
study = optuna.create_study(study_name="mastering_pytorch", direction="maximize")
study.optimize(objective, n_trials=10, timeout=2000)
```

The `direction` argument helps Optuna compare different hyperparameter settings. Because our metric is accuracy, we will need to `maximize` the metric. We allow a maximum of `2000` seconds for the `study` or a maximum of `10` different searches—whichever finishes first. The preceding command should output the following:

```
[I 2020-10-24 18:39:34,357] A new study created in memory with name: mastering_pytorch

epoch: 1 [0/60000 (0%)]  training loss: 2.314928
epoch: 1 [16000/60000 (27%)]     training loss: 2.339143
epoch: 1 [32000/60000 (53%)]     training loss: 2.554311
epoch: 1 [48000/60000 (80%)]     training loss: 2.392770

Test dataset: Overall Loss: 2.4598, Overall Accuracy: 974/10000 (10%)

epoch: 2 [0/60000 (0%)]  training loss: 2.352818
epoch: 2 [16000/60000 (27%)]     training loss: 2.425988
epoch: 2 [32000/60000 (53%)]     training loss: 2.432955
epoch: 2 [48000/60000 (80%)]     training loss: 2.497166

[I 2020-10-24 18:44:51,667] Trial 0 finished with value: 9.82 and parameters: {'num_conv_layers': 4, 'num_fc_layers':
2, 'conv_depth_0': 20, 'conv_depth_1': 18, 'conv_depth_2': 38, 'conv_depth_3': 27, 'conv_dropout_3': 0.18560304003563
005, 'fc_output_feat_0': 54, 'fc_dropout_0': 0.18233257074201586, 'fc_output_feat_1': 55, 'fc_dropout_1': 0.104182596
77735323, 'optimizer': 'RMSprop', 'lr': 0.49822431360836333}. Best is trial 0 with value: 9.82.

[I 2020-10-24 18:46:24,551] Trial 1 finished with value: 95.68 and parameters: {'num_conv_layers': 1, 'num_fc_layer
s': 2, 'conv_depth_0': 39, 'conv_dropout_0': 0.3950204757059781, 'fc_output_feat_0': 17, 'fc_dropout_0': 0.3760852329
345368, 'fc_output_feat_1': 40, 'fc_dropout_1': 0.29727560678671294, 'optimizer': 'Adadelta', 'lr': 0.254984294053231
25}. Best is trial 1 with value: 95.68.

[I 2020-10-24 18:51:37,575] Trial 2 finished with value: 98.77 and parameters: {'num_conv_layers': 3, 'num_fc_layer
s': 2, 'conv_depth_0': 27, 'conv_depth_1': 28, 'conv_depth_2': 46, 'conv_dropout_2': 0.3274565117338556, 'fc_output_f
eat_0': 57, 'fc_dropout_0': 0.12348496153785013, 'fc_output_feat_1': 54, 'fc_dropout_1': 0.36784682560478876, 'optimi
zer': 'Adadelta', 'lr': 0.4290610978292583}. Best is trial 2 with value: 98.77.

[I 2020-10-24 18:55:41,400] Trial 3 finished with value: 98.28 and parameters: {'num_conv_layers': 2, 'num_fc_layer
s': 1, 'conv_depth_0': 38, 'conv_depth_1': 40, 'conv_dropout_1': 0.3592746030824463, 'fc_output_feat_0': 20, 'fc_drop
out_0': 0.22476024022504099, 'optimizer': 'Adadelta', 'lr': 0.3167228174356792}. Best is trial 2 with value: 98.77.

[I 2020-10-24 18:59:54,755] Trial 4 finished with value: 10.28 and parameters: {'num_conv_layers': 2, 'num_fc_layer
s': 2, 'conv_depth_0': 26, 'conv_depth_1': 50, 'conv_dropout_1': 0.30220610162727457, 'fc_output_feat_0': 42, 'fc_dro
pout_0': 0.1561741472895425, 'fc_output_feat_1': 33, 'fc_dropout_1': 0.31642189637209367, 'optimizer': 'RMSprop', 'l
r': 0.45189990541514835}. Best is trial 2 with value: 98.77.

[I 2020-10-24 19:02:39,390] Trial 5 finished with value: 98.12 and parameters: {'num_conv_layers': 2, 'num_fc_layer
s': 1, 'conv_depth_0': 31, 'conv_depth_1': 22, 'conv_dropout_1': 0.3612944916702828, 'fc_output_feat_0': 25, 'fc_drop
out_0': 0.2839369529837842, 'optimizer': 'SGD', 'lr': 0.11490140528643872}. Best is trial 2 with value: 98.77.

[I 2020-10-24 19:06:33,825] Trial 6 finished with value: 98.29 and parameters: {'num_conv_layers': 2, 'num_fc_layer
s': 2, 'conv_depth_0': 24, 'conv_depth_1': 55, 'conv_dropout_1': 0.34239043023224586, 'fc_output_feat_0': 35, 'fc_dro
pout_0': 0.17065510224232447, 'fc_output_feat_1': 46, 'fc_dropout_1': 0.19804499857448277, 'optimizer': 'Adadelta',
'lr': 0.42138811722164293}. Best is trial 2 with value: 98.77.

[I 2020-10-24 19:09:33,855] Trial 7 pruned.

[I 2020-10-24 19:10:33,804] Trial 8 pruned.

[I 2020-10-24 19:15:36,906] Trial 9 pruned.
```

Figure 16 .6 – Optuna logs

As we can see, the third `trial` is the most optimal trial, producing a test set accuracy of 98.77%, and the last three `trials` are pruned. In the logs, we also see the hyperparameters for each non-pruned `trial`. For the most optimal `trial`, for example, there are three convolutional layers with 27, 28, and 46 feature maps respectively, and then there are two fully connected layers with 57 and 54 units/neurons respectively, and so on.

1. Each `trial` is given a completed or a pruned status. We can demarcate those with the following code:

```
pruned_trials = [t for t in study.trials if t.state ==
optuna.trial.TrialState.PRUNED]complete_trials = [t for t in study.trials if t.state
== optuna.trial.TrialState.COMPLETE]
```

1. And finally, we can specifically look at all the hyperparameters of the most successful `trial` with the following code:

```
print("results: ")
trial = study.best_trial
for key, value in trial.params.items():
    print("{}: {}".format(key, value))
```

You will see the following output:

```
results:
num_trials_conducted:  10
num_trials_pruned:  3
num_trials_completed:  7
results from best trial:
accuracy:  98.77
hyperparameters:
num_conv_layers: 3
num_fc_layers: 2
conv_depth_0: 27
conv_depth_1: 28
conv_depth_2: 46
conv_dropout_2: 0.3274565117338556
fc_output_feat_0: 57
fc_dropout_0: 0.12348496153785013
fc_output_feat_1: 54
fc_dropout_1: 0.36784682560478876
optimizer: Adadelta
lr: 0.4290610978292583
```

Figure 16 .7 – Optuna optimal hyperparameters

As we can see, the output shows us the total number of `trials` and the number of successful `trials` performed. It further shows us the model hyperparameters for the most successful `trial`, such as the number of layers, the number of neurons in layers, learning rate, optimization schedule, and so on.

This brings us to the end of the exercise. We have managed to use Optuna to define a range of hyperparameter values for different kinds of hyperparameters for a handwritten digit classification model. Using Optuna's hyperparameter search algorithm, we ran 10 different `trials` and managed to obtain the highest accuracy of 98.77% in one of those `trials`. The model (architecture and hyperparameters) from the most successful `trial` can be used for training with larger datasets, thereby serving in a production system.

Using the lessons from this section, you can use Optuna to find the optimal hyperparameters for any neural network model written in PyTorch. Optuna can also be used in a distributed fashion if the model is extremely large and/or there are way too many hyperparameters to tune [16.15] .

Lastly, Optuna supports not only PyTorch but other popular machine learning libraries too, such as `TensorFlow`, `Sklearn`, `MXNet`, and so on.

Summary

In this chapter, we discussed AutoML, which aims to provide methods for model selection and hyperparameter optimization. AutoML is useful for beginners who have little expertise on making decisions such as how many layers to put in a model, which optimizer to use, and so on. AutoML is also useful for experts to both speed up the model training process and discover superior model architectures for a given task that would be nearly impossible to figure manually.

In the next chapter, we will study another increasingly important and crucial aspect of machine learning, especially deep learning. We will closely look at how to interpret output produced by PyTorch models—a field popularly known as model interpretability or explainability.

Previous Chapter

Next Chapter