Hyperparameter Tuning

Introduction

Hyperparameter tuning is a central process in the development of high-performing deep learning models. Unlike model parameters, which are learned during training (such as the weights in a neural network), hyperparameters are set before training begins and influence the learning process itself. These include values like the learning rate, number of hidden layers, number of neurons per layer, activation functions, batch size, optimizer types, dropout rates, weight decay, and many others.

Hyperparameter tuning is a black-box optimization problem where we try to find the optimal configuration of hyperparameters that minimizes (or maximizes) an objective function, usually the validation loss or validation accuracy. Poorly chosen hyperparameters can lead to slow convergence, poor performance, or even training failure. Deep learning models are often involve a large number of parameters and extensive architectural flexibility. This high capacity requires careful calibration of hyperparameters to achieve generalization, that is, good performance on the training data as well as on unseen test data. Some hyperparameters, like learning rate, have a significant impact on model convergence and stability. Others, such as dropout rate or batch size, influence regularization and efficiency.

Types of Hyperparameters

Hyperparameters in deep leanring can be broadly classified as follows:

Architecture Hyperparameters

- Number of layers (depth)
- Number of neurons per layer (width)
- Type of activation function (ReLU, Tanh, LeakyReLU, etc.)
- Convolutional kernel size, stride, padding (for CNNs)
- Recurrent unit type (LSTM, GRU) and sequence length (for RNNs)

Training Hyperparameters

- Learning rate (often the most important)
- Batch size
- Optimizer (SGD, Adam, RMSProp, etc.)

- Number of training epochs
- Learning rate schedule (step decay etc.)

Regularization Hyperparameters

- Dropout rate
- Weight decay (L2 regularization)
- Early stopping patience
- Data augmentation parameters

Common Methods for Hyperparameter Tuning

Manual Search

This is the most intuitive approach: try different values based on experience, theory, or trial and error. While this can work well for experienced practitioners, it is time-consuming and non-scalable, especially in high-dimensional hyperparameter spaces.

Grid Search

Grid search exhaustively evaluates a manually specified subset of the hyperparameter space. For example, for learning rate $\in \{0.01, 0.001, 0.0001\}$ and batch size $\in \{32, 64, 128\}$, grid search tests all 9 combinations. Grid search-based approaches are simple and parallelizable, but they are inefficient in high-dimensional spaces and don't account for diminishing returns, since all combinations are treated equally.

Random Search

Instead of trying every combination, random search samples values randomly from predefined distributions. Random search has often been proven to perform better than grid search, especially when only a few hyperparameters are influential. It is easy to implement, more efficient, and allows for early stopping.

Gradient-Based Tuning

Some recent methods attempt to differentiate through the training process to obtain gradients with respect to hyperparameters. While this is more theoretical and harder to implement due to computational and memory constraints, it allows optimization with respect to validation loss. Examples include Hypergradient descent or DARTS.

Evolutionary Algorithms and Reinforcement Learning

These methods simulate evolution or learning agents that propose new hyperparameter configurations over time.

- Evolutionary algorithms encode the network architecture into a value vector (chromosome) and search for the optimal configuration using operators such as mutation, selection, crossover.
- RL agents propose better hyperparameter configurations over time as a result of having learned a suitable policy.

The Optuna Library

Optuna is an open-source, automatic hyperparameter optimization framework designed to handle model optimization in a scalable, flexible, and efficient way. In an Optuna implementation, we define:

- A search space (what parameters can vary and in what range)
- An objective function (what to optimize – error, validation loss etc.)
- An optimization strategy (random search, grid, search , TPE etc.)

Further details can be found in the official documentation.

A Step-by-Step Example

In the following example, we use Optuna to tune the hyperparameters of a classification neural network.

The model is defined as follows:

```
class NNet(nn.Module):
    def __init__(self, input_size, hidden_size, dropout_rate):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Dropout(dropout_rate),
```

```
            nn.Linear(hidden_size, 10)
            )
    def forward(self, x):
        return self.layers(x)

    def train_model(self, train_dataset, optimizer_name, learn_rate, batch_size):
       ...


    def eval_model(self, test_dataset, batch_size = 64):
       ...
        return error
```

Optuna formulates the hyperparameter search as an optimization problem. We define the objective of the optimization as “minimizing the error on the valudation data set”. We can define the objective as a simple function, or, using a more flexible approach, as a callable object:

```
class Objective:
    def __init__(self, train_dataset, validation_dataset):
        self.train_dataset = train_dataset
        self.validation_dataset = validation_dataset

    def __call__(self, trial):

        # sample hyperparameters
        hidden_size = trial.suggest_int('hidden_size', 64, 256)
        dropout_rate = trial.suggest_float('dropout_rate', 0.1, 0.5)
        learn_rate = trial.suggest_float('learn_rate', 1e-4, 1e-1, log = True)
        optimizer_name = trial.suggest_categorical('optimizer', ['Adam', 'SGD'])
        batch_size = trial.suggest_categorical('batch_size', [32, 64, 128])

        model = NNet(28*28, hidden_size, dropout_rate)
        model.train_model(self.train_dataset, optimizer_name, learn_rate,
batch_size)
        err = model.eval_model(self.validation_dataset, batch_size)

        return err
```

In the `Objective` class, we train the model on the training set and return the error on the validation set. The `__call__()` method allows an object to be called like a function (similar to overloading the “round brackets” operator in order to create a functor object).

The `__call__()` method has a single parameter in the form of a `Trial` object, which will be used by Optuna later.

In the `__call__()` method we do the following important tasks:

- We define which hyperparameters of the network should be searched for (hidden_size, dropout_rate etc.).
- For each of these hyperparameters, we sample a value from a domain unique to each parameter. This is done using `trial.suggest_*` methods. For example:
    - We generate a value for the `hidden_size` hyperparameter as an integer from the interval [64, 256]
    - We generate a value for the optimizer (i.e. which optimizer shall be used for training the neural network) as a categorical value from the set `['Adam', 'SGD']`
- We define a neural network model, and train it using the previously-generated values of the hyperparameters. Then, we measure the error on the validation set. Lower errors mean that the model trained using the current hyperparameter values is better.

In the `__main__` function, we define the optimization problem as an Optuna Study:

```
study = optuna.create_study(study_name = 'Network Optimizer')
```

Then, we start the actual optimization process, which involves finding the combination of hyperparameter values which minimize the specified objective:

```
study.optimize(Objective(train_dataset, validation_dataset), n_trials = 5)
```

The optimizer method takes as parameters the objective (what should be minimized) and the number of trials to carry out (how many combinations of hyperparameters should be tried). This results in the following steps:

- Within a trial, a combination of values is generated for the hyperparameters.
- A model is built and trained using the hyperparameter values.
- The objective is measured (error on the validation set).
- Optuna moves on to the next trial, where the previous three steps are repeated (with other hyperparameter values).

When the `n_trials` trials have completed, the Optuna Study is over. We present the results in the form of:

- The best value achieved during the study (the lowest validation error).
- The values of the hyperparameters which caused the error to minimize (the hyperparameters which result in the best model)

Tasks

For completing the tasks, the code samples that are provided with this documentation can be used as starting points.

1) For the fully-connected network, add the possibility to search for architectures with multiple layers (up to three) of various sizes.
2) Implement a comvolutional network for the same data set and tune its hyperparameters using Optuna. Training-related parameters (learning rate, optimizer) can be searched for as exemplified, and in addition, convolution network-specific hyperparameters should be searched, such as:
    - Number of convolutional layers
    - Number of filters per layer
    - Filter (kernel) size – we suggest sampling from a few fixed sizes ([3x3], [5x5] etc.)
    - Pooling type (“max”, “avg”)
    - Pooling kernel size (2x2, 3x3, ...)