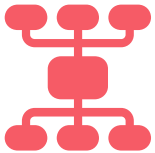
DeepLearning.AI

Welcome to Course 2

Hyperparameter optimization

In Course 1 you explored:



Pipelines

In Course 1 you explored:

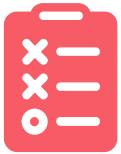
Pipelines

Build and train

In Course 1 you explored:

Pipelines

Build and train



Evaluate

In Course 1 you explored:

Pipelines

Build and train

Evaluate

Inspect

In this course you'll dive into:

Optimization

In this course you'll dive into:

Optimization

Images

In this course you'll dive into:

Optimization

Images

Text

In this course you'll dive into:

Optimization

Images

Text

Efficiency

In this course you'll dive into:

Optimization

Images

Text

Efficiency

Some concepts in this course will revisit topics covered in the previous course

In this course you'll dive into:

Optimization

Images

Text

Efficiency

Optimization: Finding the optimal value

Adjust
hyperparameters

Optimization: Finding the optimal value

Adjust hyperparameters

Improve a metric

Optimization: Finding the optimal value

Adjust hyperparameters

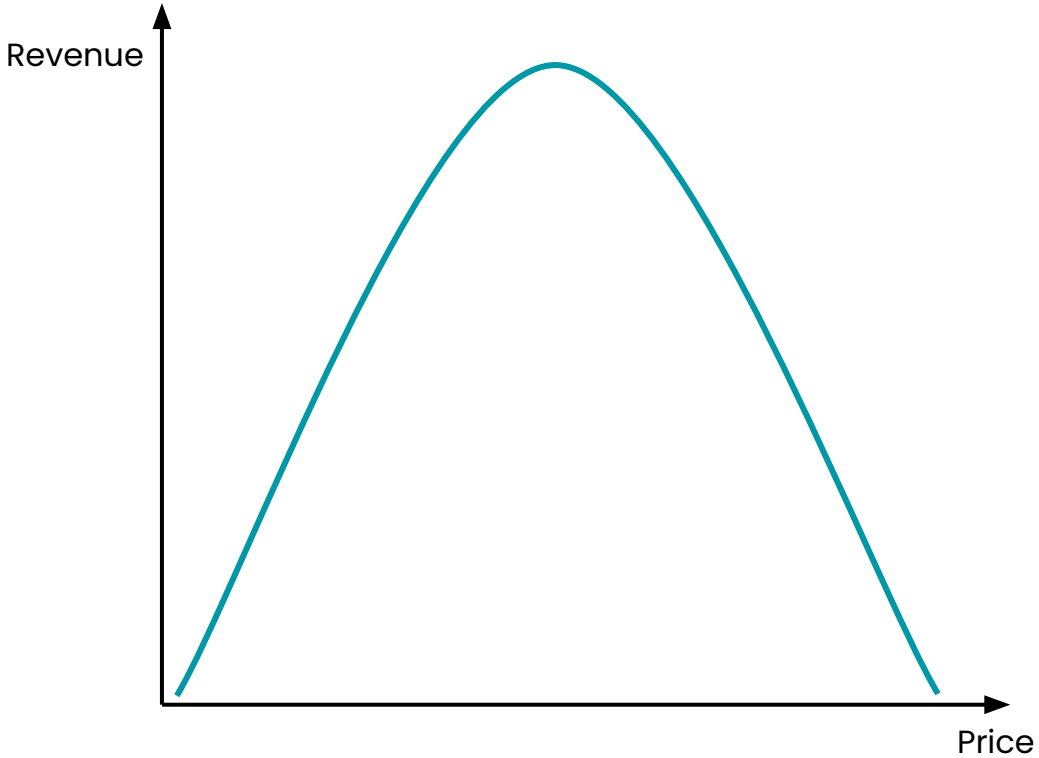
Improve a metric

Accuracy

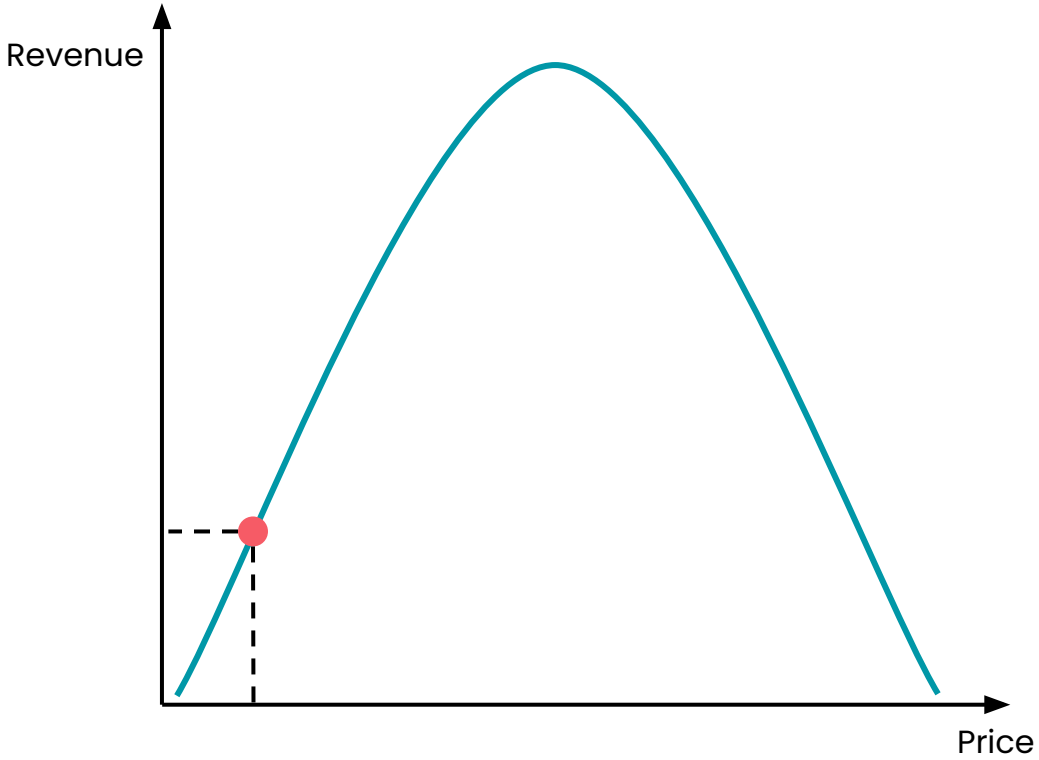
Speed

Efficiency

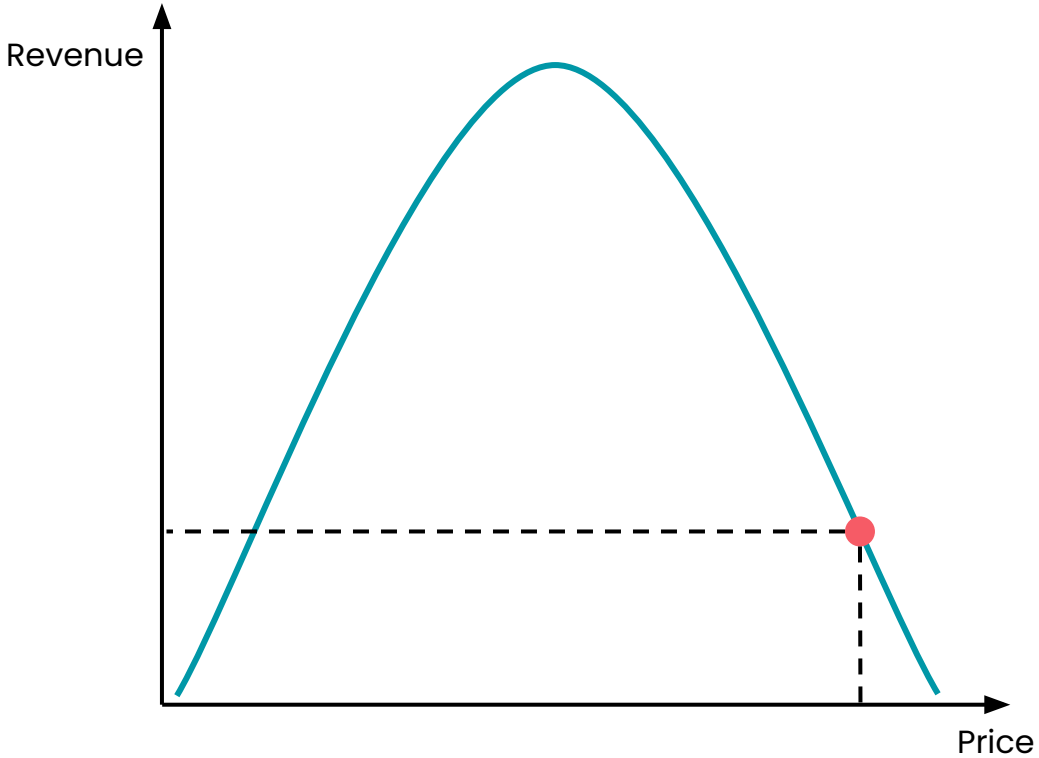
Finding the optimal price for your app



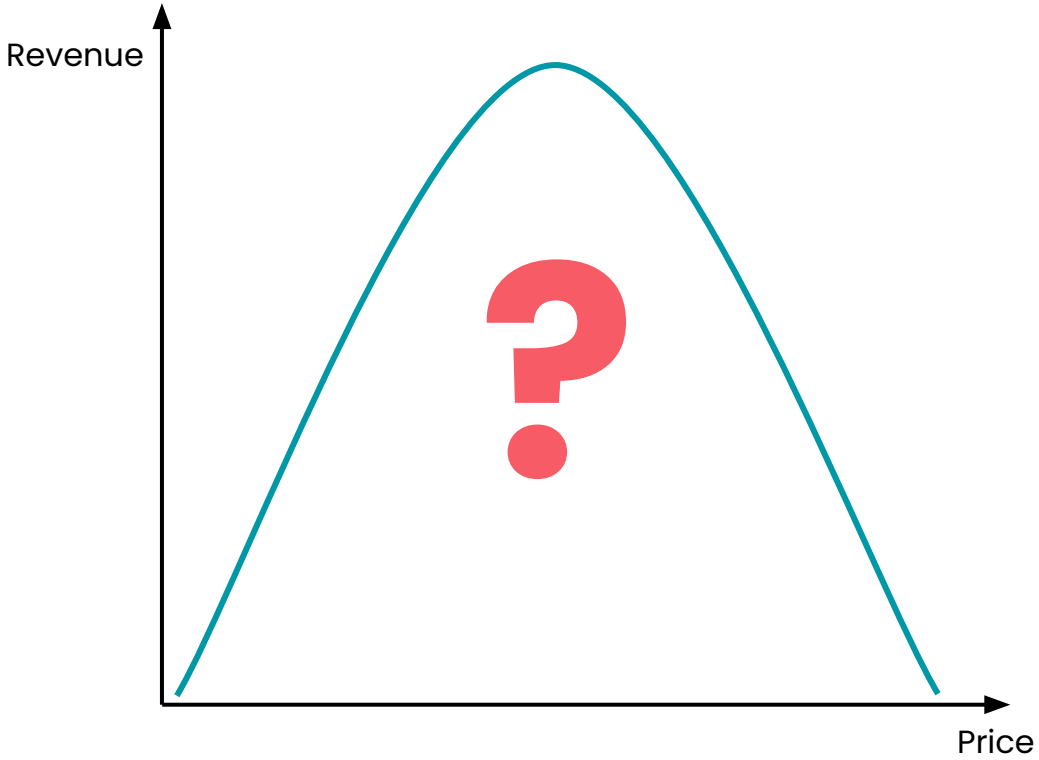
Finding the optimal price for your app



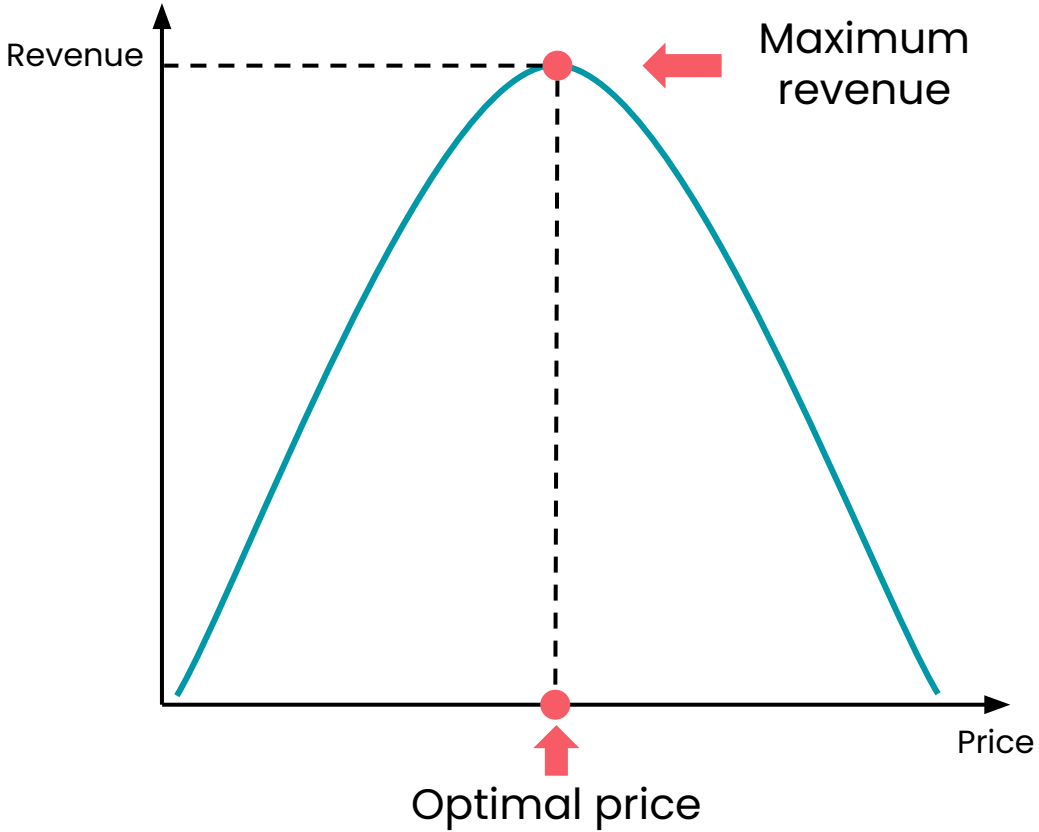
Finding the optimal price for your app



Finding the optimal price for your app



Finding the optimal price for your app



DeepLearning.AI

Evaluation Metrics

Hyperparameter optimization

Accuracy

Percentage of correct predictions made by your model

$$Accuracy = \frac{Correct\ predictions}{All\ predictions}$$

Accuracy

Percentage of correct predictions made by your model

$$Accuracy = \frac{Correct\ predictions}{All\ predictions}$$

Maximize

Accuracy

Percentage of correct predictions made by your model

$$Accuracy = \frac{Correct\ predictions}{All\ predictions}$$

Maximize

Can be unreliable with imbalanced datasets

Classification outcomes

Classification outcomes

Predicted

True positive

Actual

Classification outcomes

Predicted

Actual

True positive

False positive

Classification outcomes

Predicted

Actual

True positive

False positive

True negative

Classification outcomes

Predicted

Actual

True positive

False negative

False positive

True negative

Precision

Measures how often the model's positive predictions are correct

Precision

Measures how often the model's positive predictions are correct

$$Precision = \frac{True\ positives}{True\ positives + False\ positives}$$

Precision

Measures how often the model's positive predictions are correct

$$Precision = \frac{True\ positives}{True\ positives + False\ positives}$$

Maximize

Precision

Measures how often the model's positive predictions are correct

$$Precision = \frac{True\ positives}{True\ positives + False\ positives}$$

Maximize

Use when false positives are costly

Recall

Measures how often the model identifies true positives from all positives

Recall

Measures how often the model identifies true positives from all positives

$$Recall = \frac{True\ positives}{True\ positives + False\ negatives}$$

Recall

Measures how often the model identifies true positives from all positives

$$Recall = \frac{True\ positives}{True\ positives + False\ negatives}$$

Maximize

Recall

Measures how often the model identifies true positives from all positives

$$Recall = \frac{True\ positives}{True\ positives + False\ negatives}$$

Maximize

Use when false negatives are costly

F1 Score

Balances false positives and false negatives

$$F1\ Score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

F1 Score

Balances false positives and false negatives

$$F1\ Score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Maximize

F1 Score

Balances false positives and false negatives

$$F1\ Score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Maximize

Preferred over accuracy with imbalanced datasets

Torchmetrics has built-in classes for:

Accuracy

Precision

Recall

F1 Score

```
import torchmetrics

def evaluate_metrics(model, val_dataloader, device, num_classes=10):
    model.eval()
    # Initialize metrics
    accuracy_metric = torchmetrics.Accuracy(
        task="multiclass", num_classes=num_classes, average="macro"
    ).to(device)
    precision_metric = torchmetrics.Precision(
        task="multiclass", num_classes=num_classes, average="macro"
    ).to(device)
    recall_metric = torchmetrics.Recall(
        task="multiclass", num_classes=num_classes, average="macro"
    ).to(device)
    f1_metric = torchmetrics.F1Score(
        task="multiclass", num_classes=num_classes, average="macro"
    ).to(device)
```

```
import torchmetrics

def evaluate_metrics(model, val_dataloader, device, num_classes=10):
    model.eval()
    # Initialize metrics
    accuracy_metric = torchmetrics.Accuracy(
        task="multiclass", num_classes=num_classes, average="macro"
    ).to(device)
    precision_metric = torchmetrics.Precision(
        task="multiclass", num_classes=num_classes, average="macro"
    ).to(device)
    recall_metric = torchmetrics.Recall(
        task="multiclass", num_classes=num_classes, average="macro"
    ).to(device)
    f1_metric = torchmetrics.F1Score(
        task="multiclass", num_classes=num_classes, average="macro"
    ).to(device)
```

```
import torchmetrics

def evaluate_metrics(model, val_dataloader, device, num_classes=10):
    model.eval()
    # Initialize metrics
    accuracy_metric = torchmetrics.Accuracy(
        task="multiclass", num_classes=num_classes, average="macro"
    ).to(device)
    precision_metric = torchmetrics.Precision(
        task="multiclass", num_classes=num_classes, average="macro"
    ).to(device)
    recall_metric = torchmetrics.Recall(
        task="multiclass", num_classes=num_classes, average="macro"
    ).to(device)
    f1_metric = torchmetrics.F1Score(
        task="multiclass", num_classes=num_classes, average="macro"
    ).to(device)
```

```
import torchmetrics

def evaluate_metrics(model, val_dataloader, device, num_classes=10):
    model.eval()
    # Initialize metrics
    accuracy_metric = torchmetrics.Accuracy(
        task="multiclass", num_classes=num_classes, average="micro"
    ).to(device)
    precision_metric = torchmetrics.Precision(
        task="multiclass", num_classes=num_classes, average="micro"
    ).to(device)
    recall_metric = torchmetrics.Recall(
        task="multiclass", num_classes=num_classes, average="micro"
    ).to(device)
    f1_metric = torchmetrics.F1Score(
        task="multiclass", num_classes=num_classes, average="micro"
    ).to(device)
```

DeepLearning.AI

Introduction to Optimization

Hyperparameter optimization

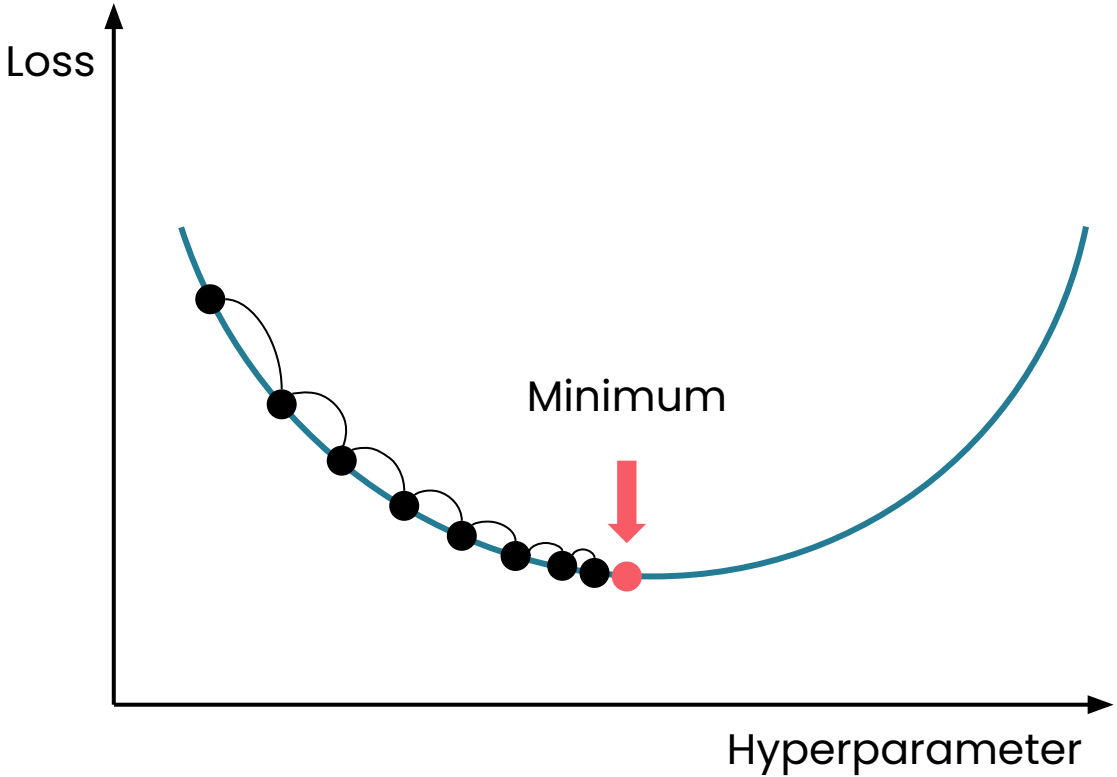
App performance

48%

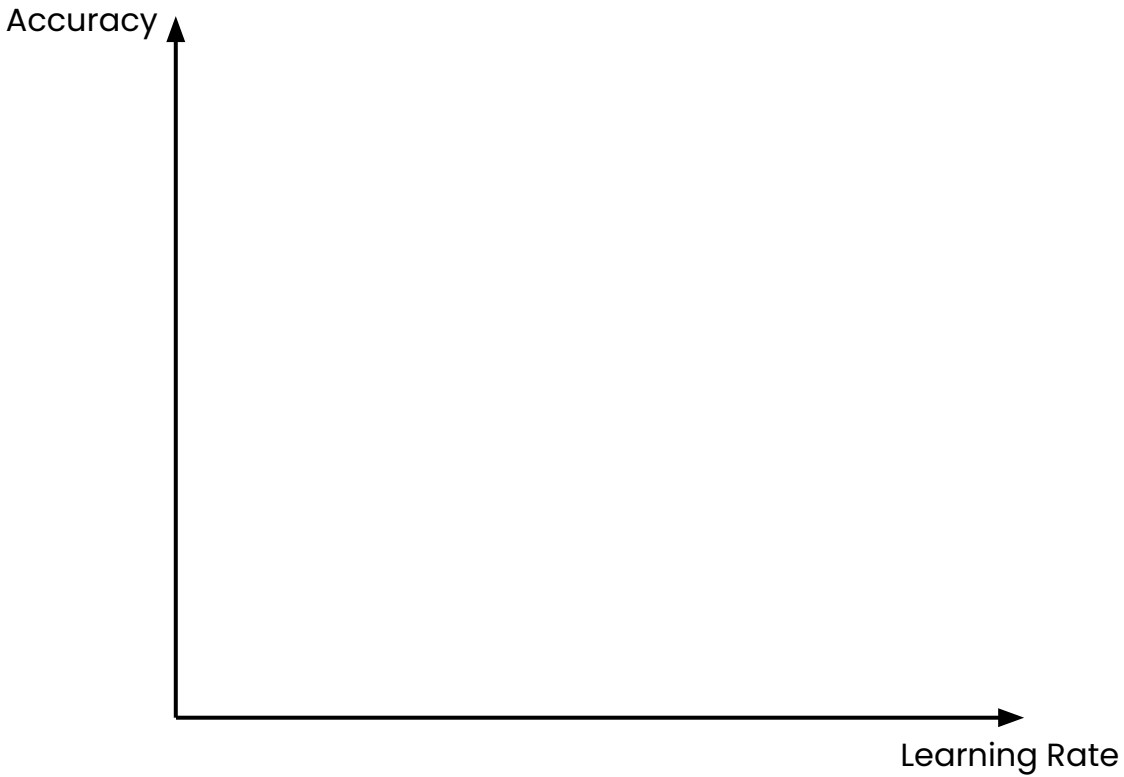
ACCURACY

DeepLearning.AI

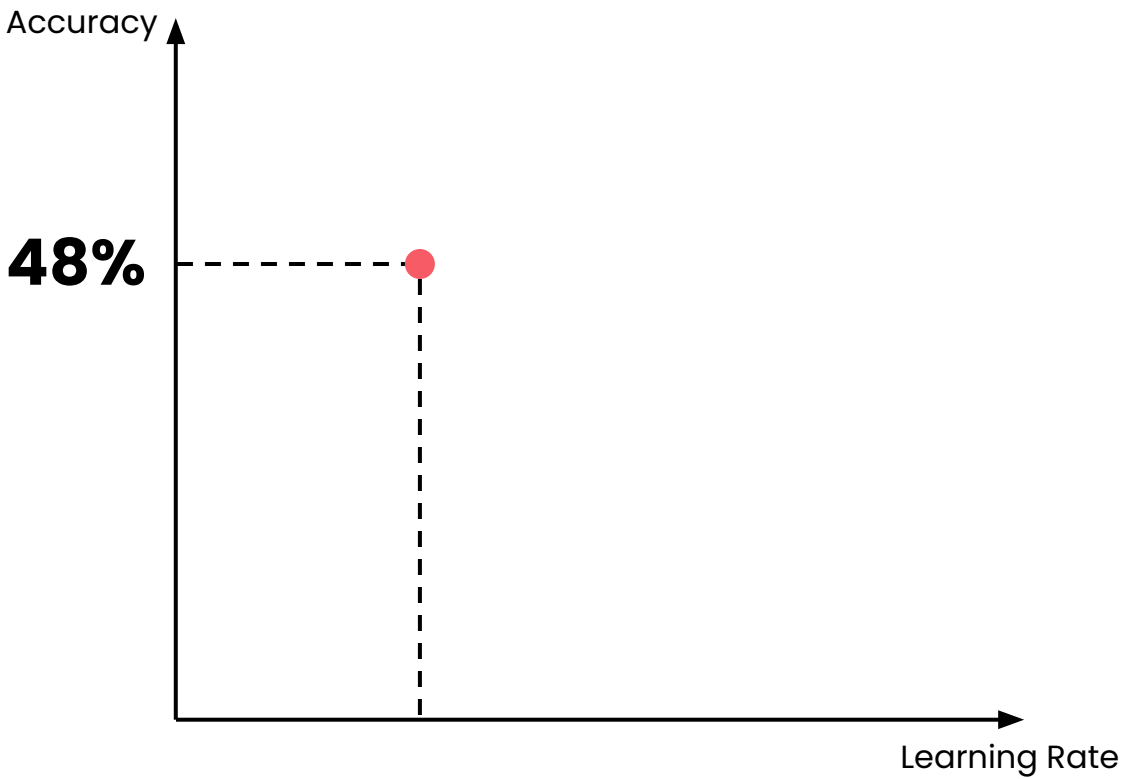
Learning Rate



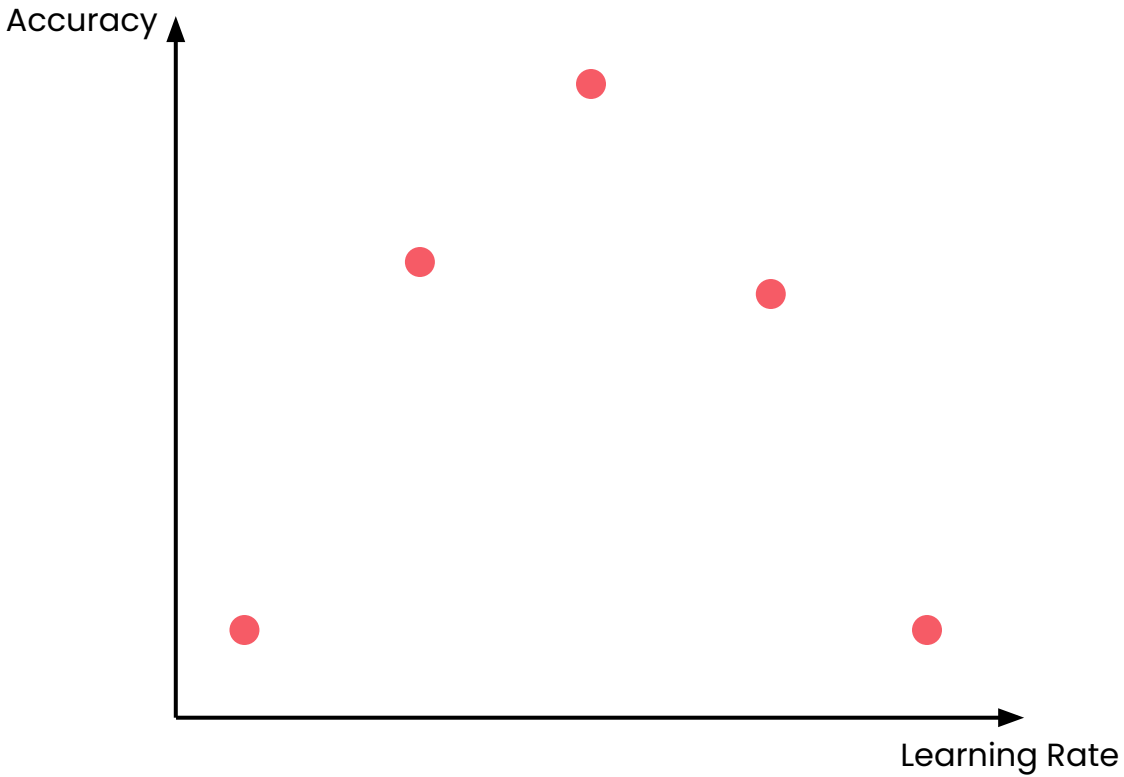
Effect of learning rate on accuracy



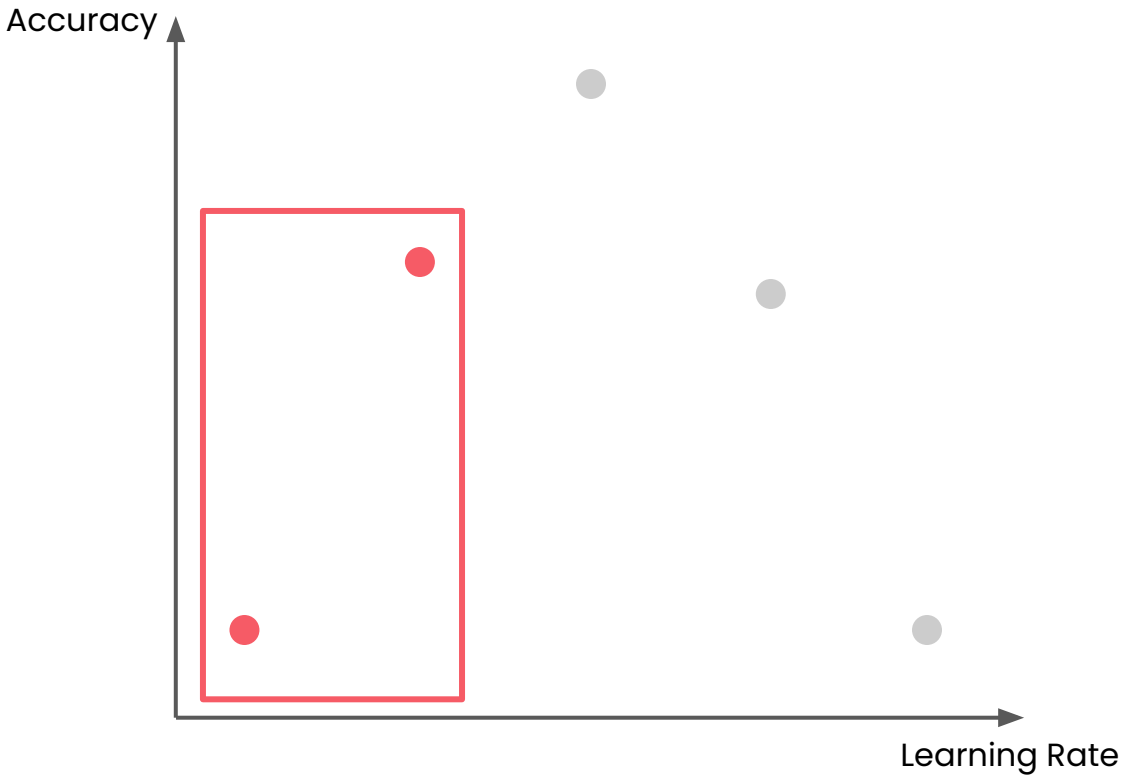
Effect of learning rate on accuracy



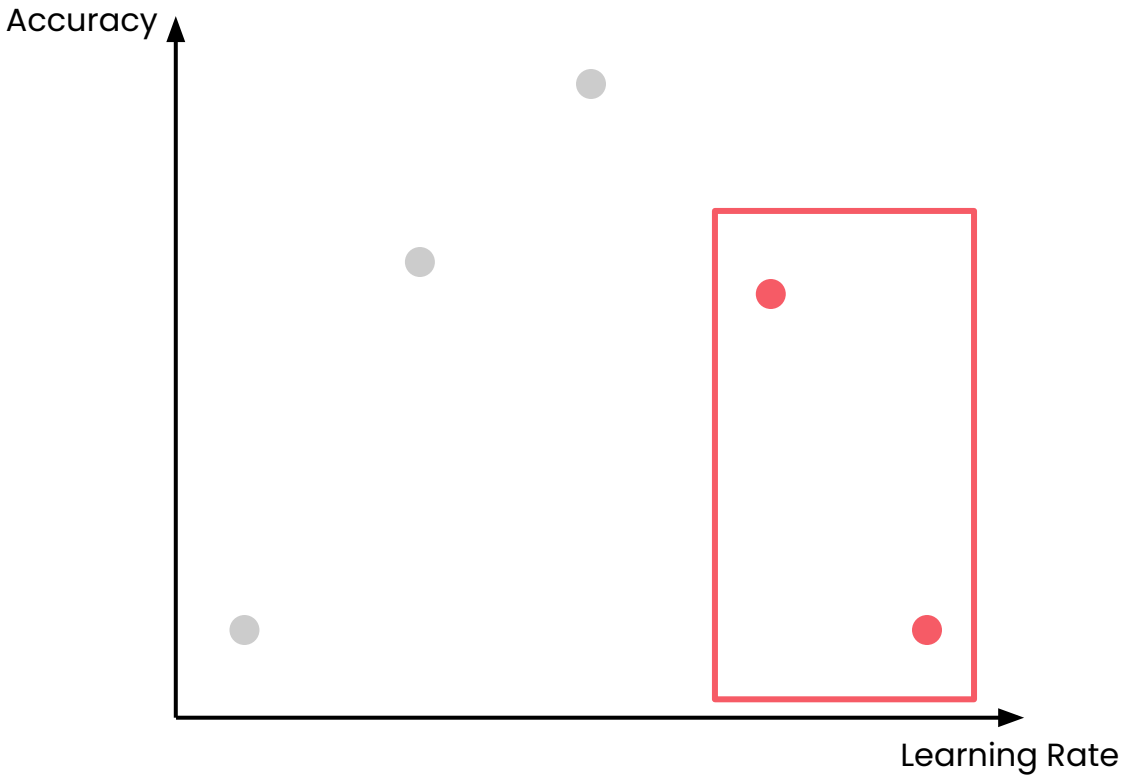
Effect of learning rate on accuracy



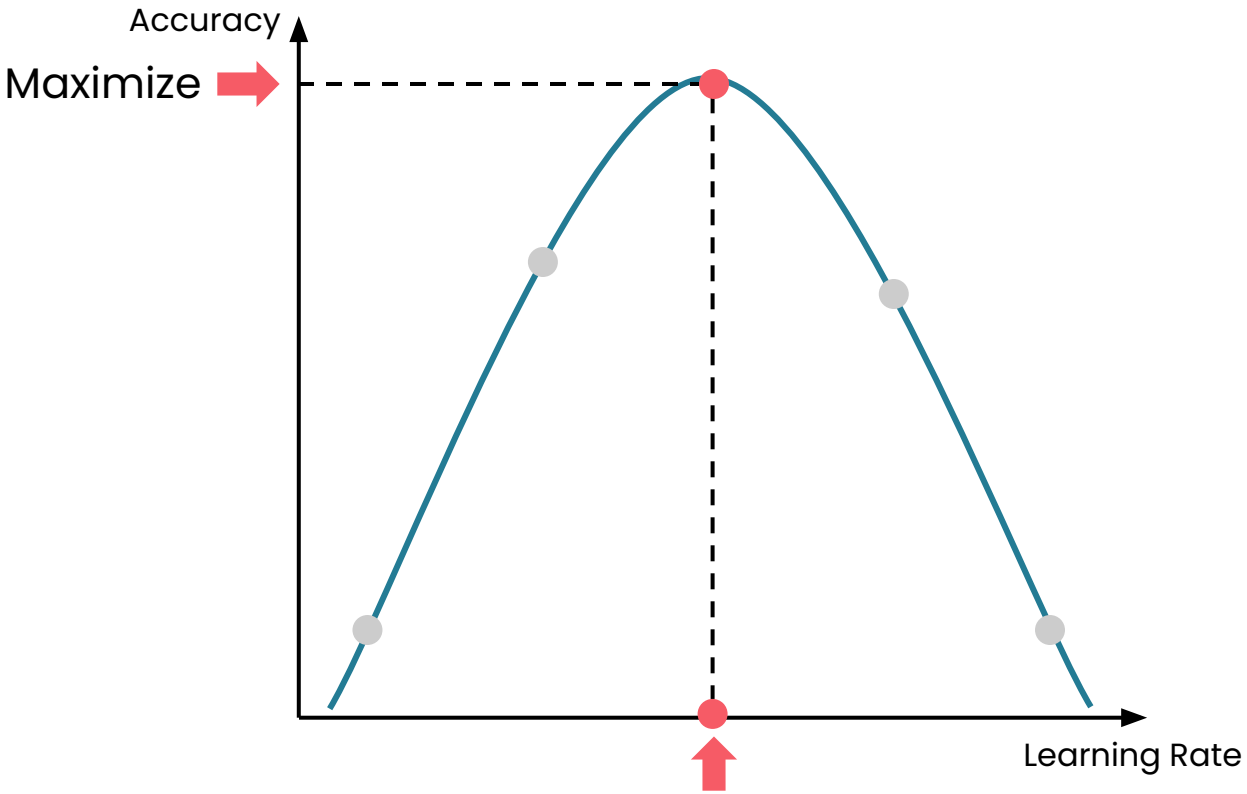
Effect of learning rate on accuracy



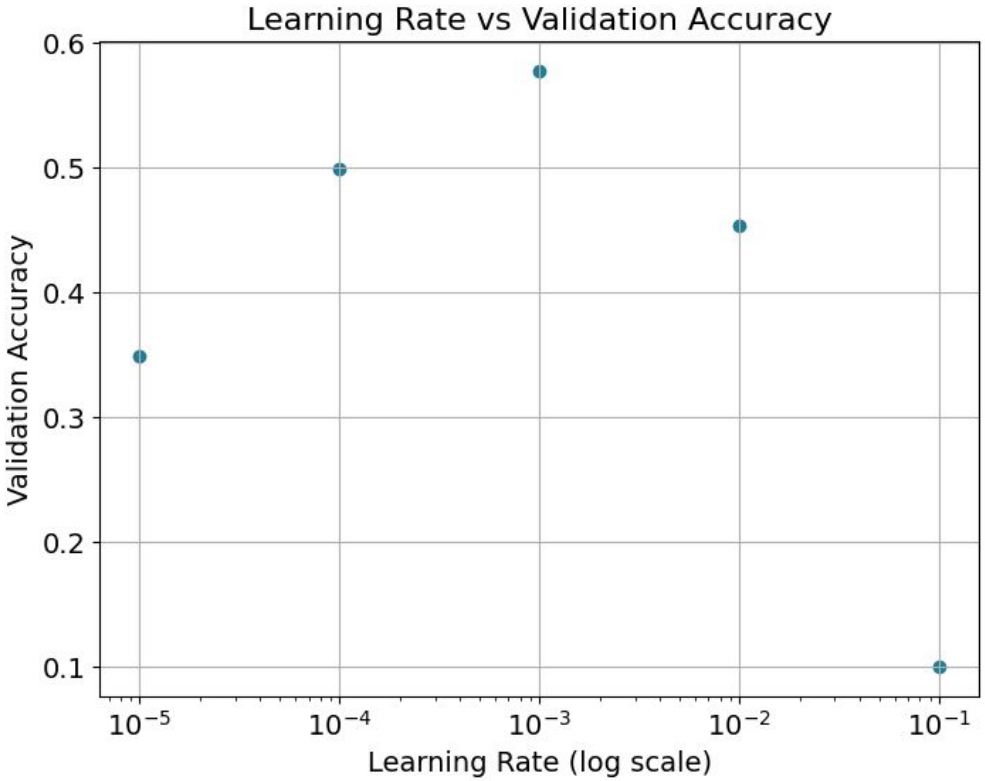
Effect of learning rate on accuracy



Effect of learning rate on accuracy



Effect of learning rate on accuracy



Optimization: fine-tuning hyperparameters

Maximize

- Accuracy
- Precision
- Recall
- F1 Score

Optimization: fine-tuning hyperparameters

Maximize

- Accuracy
- Precision
- Recall
- F1 Score

Minimize

- Training/inference time
- Mean squared error
- Log loss

How can you improve your metrics?

Externally

- More data
- Data quality
- Feature representation

How can you improve your metrics?

Externally

- More data
- Data quality
- Feature representation

Internally

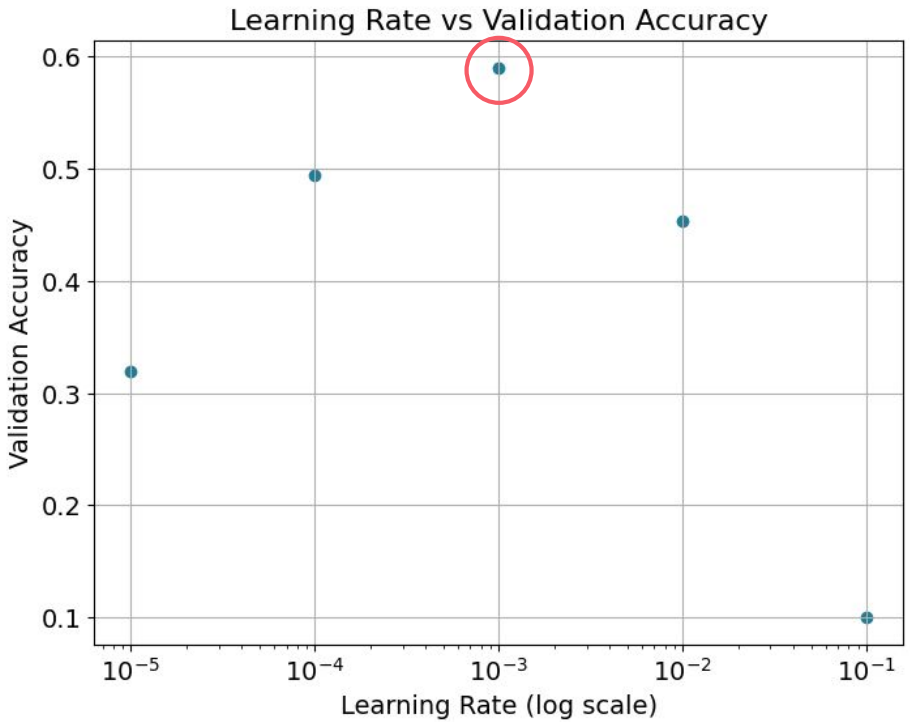
- Number of layers
- Dropout
- Batch size
- Learning rate

DeepLearning.AI

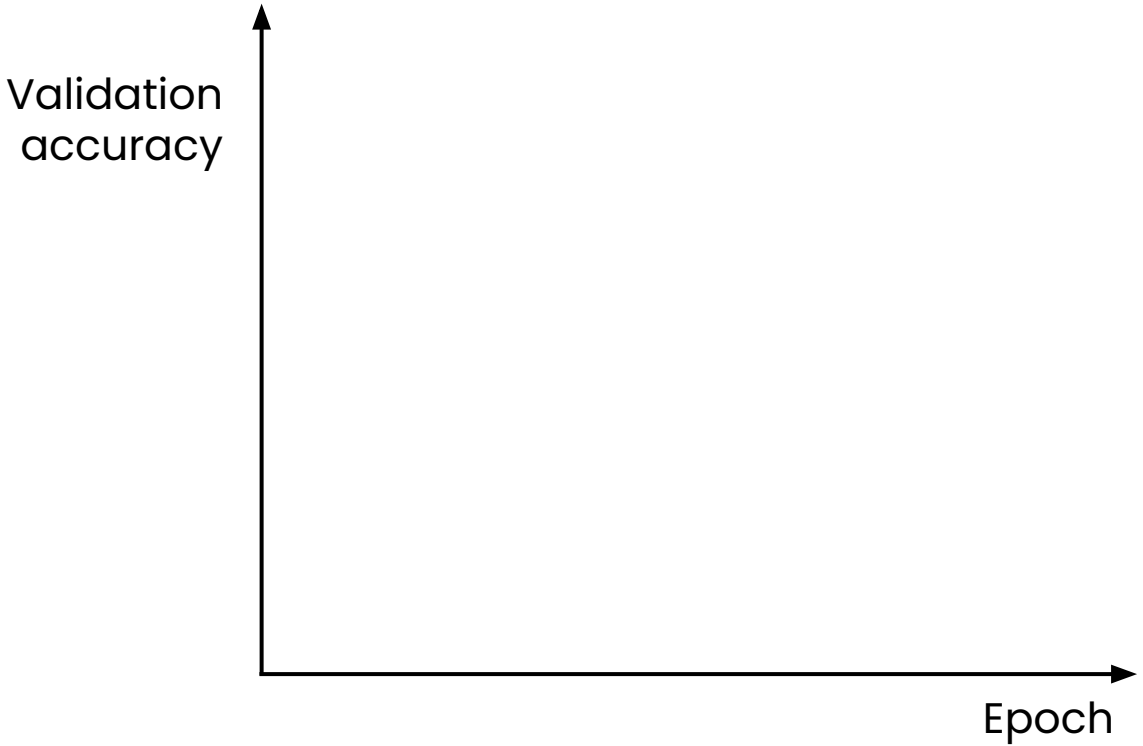
Learning Rate Schedulers

Hyperparameter optimization

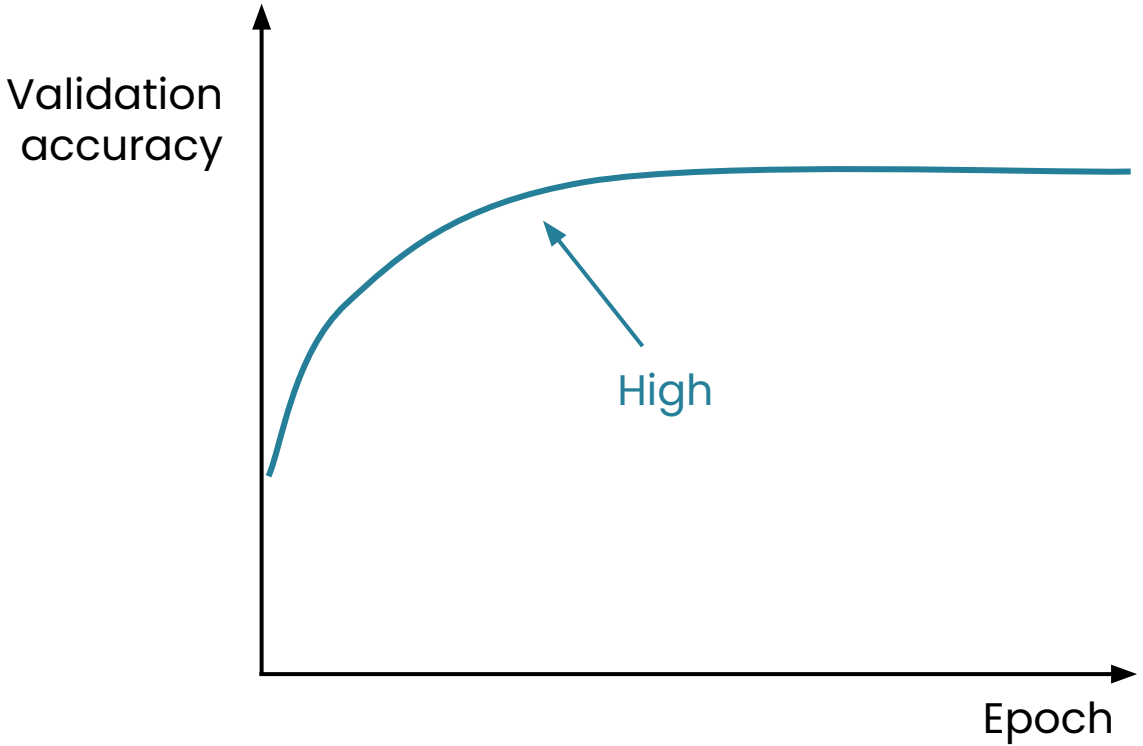
Effect of learning rate on accuracy



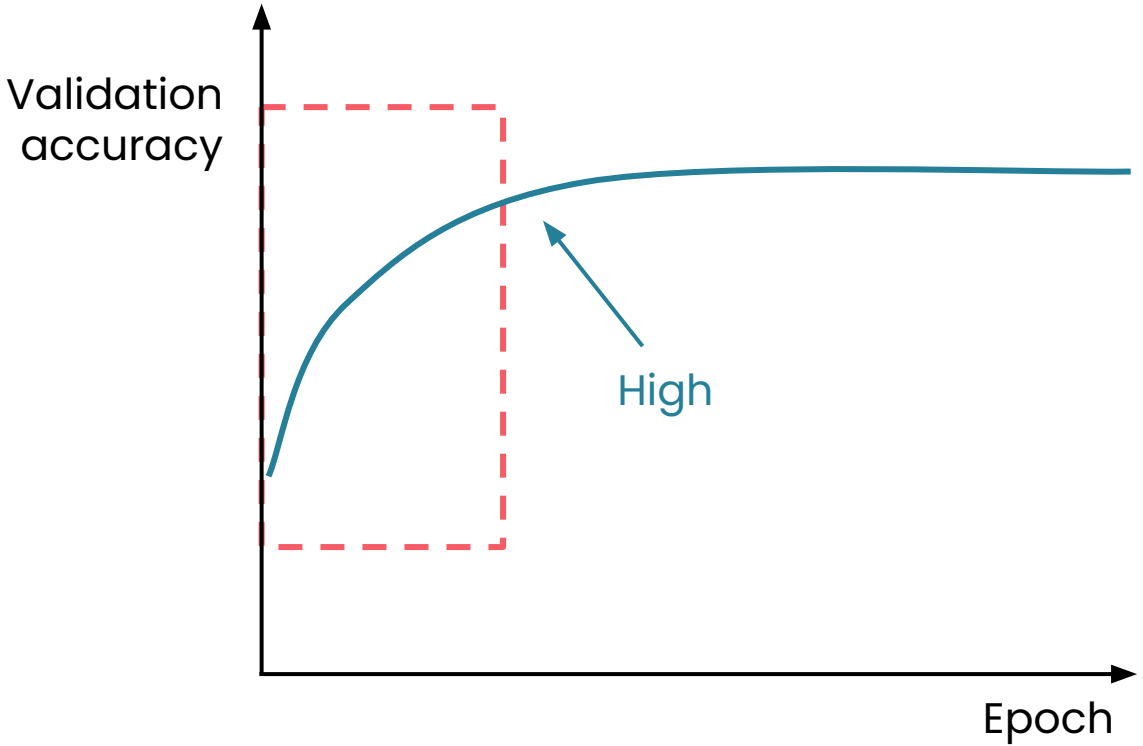
Accuracy per epoch



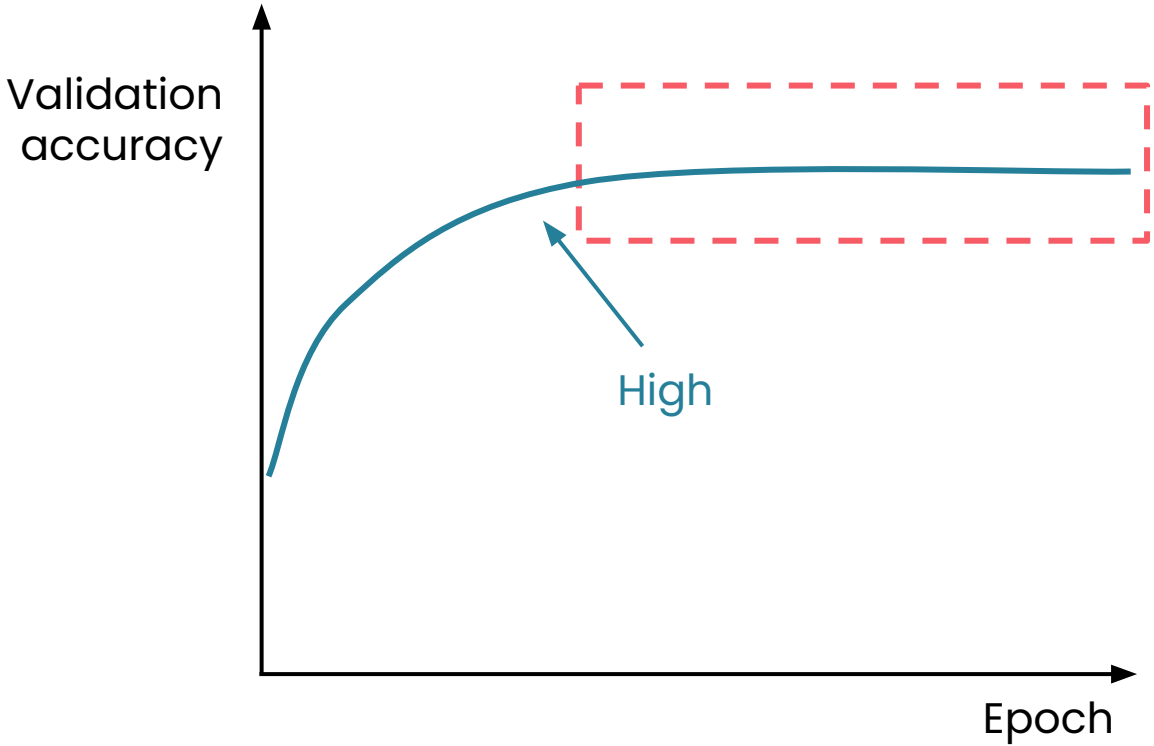
Accuracy per epoch



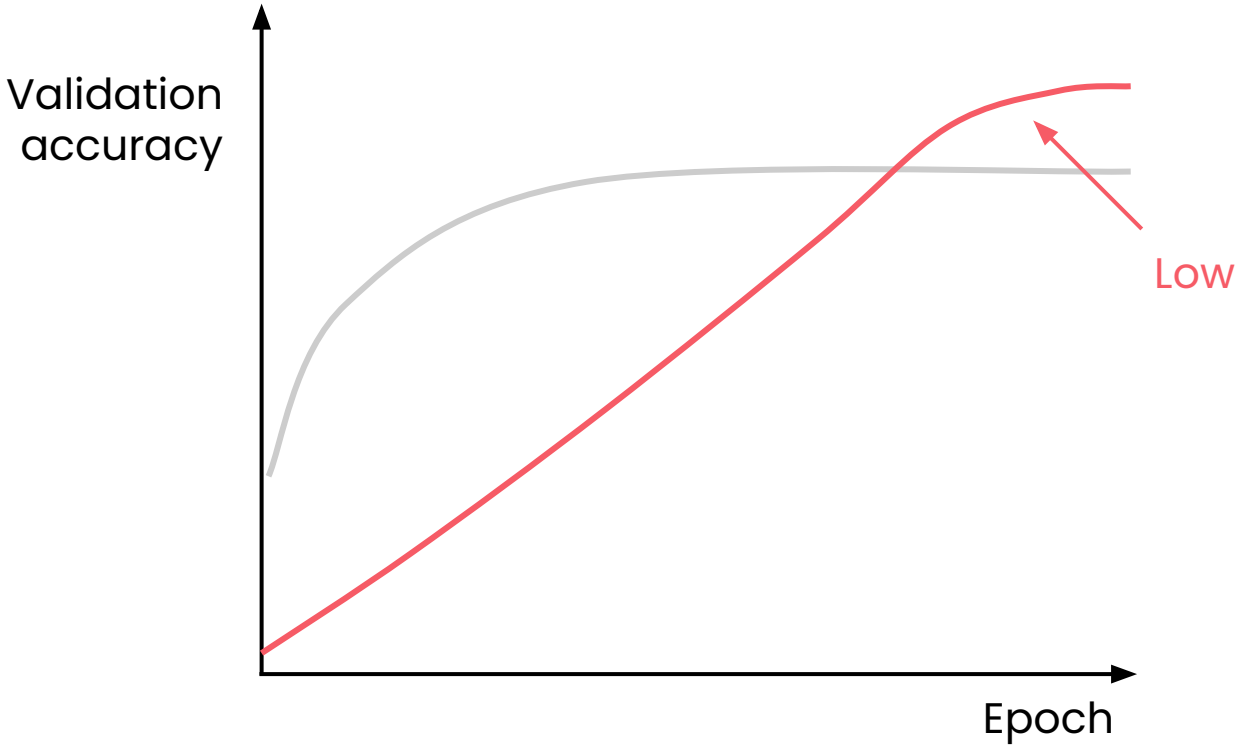
Accuracy per epoch



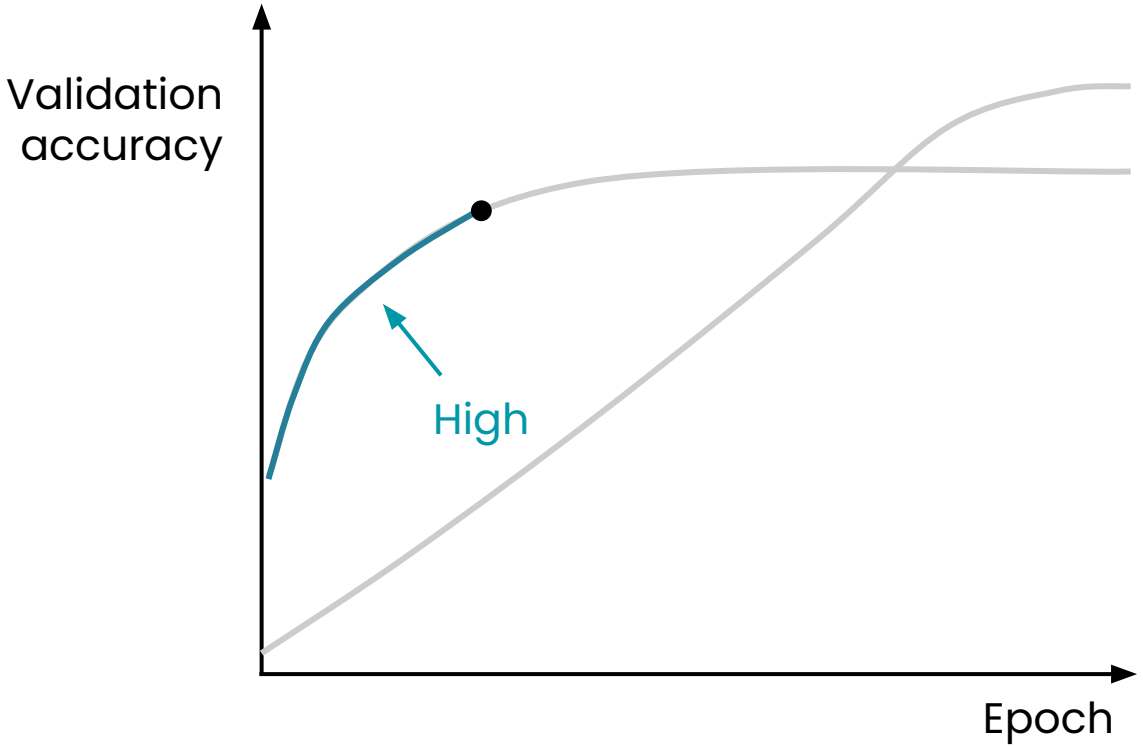
Accuracy per epoch



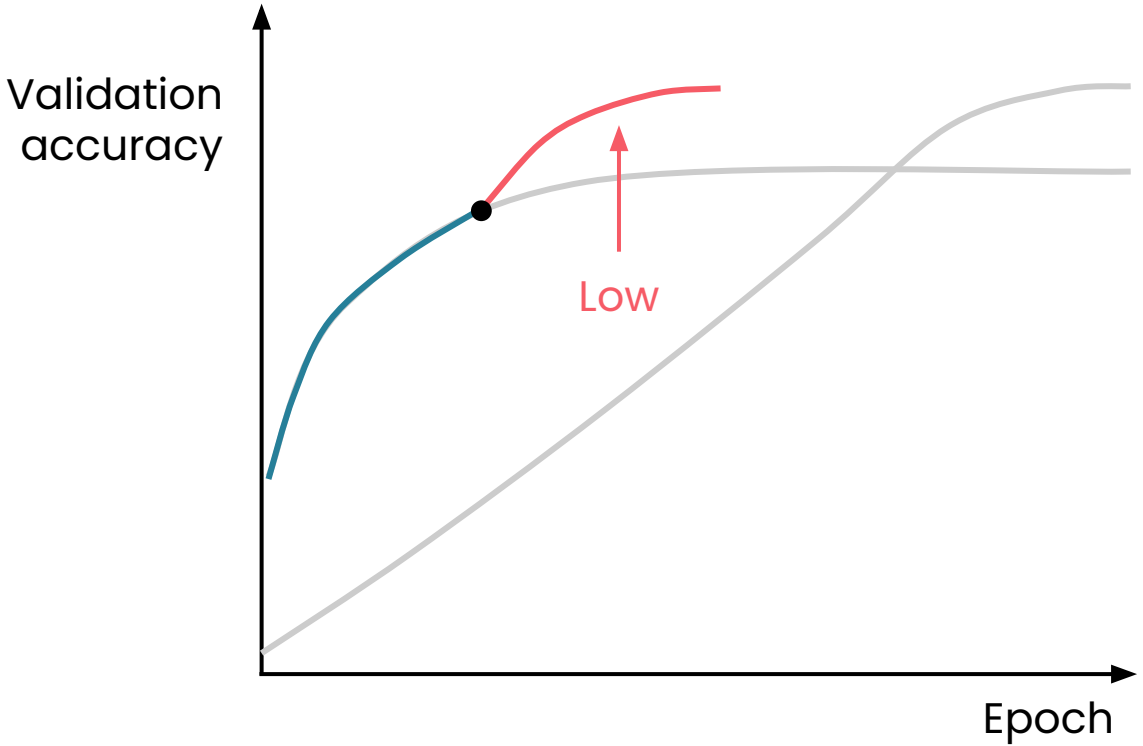
Accuracy per epoch



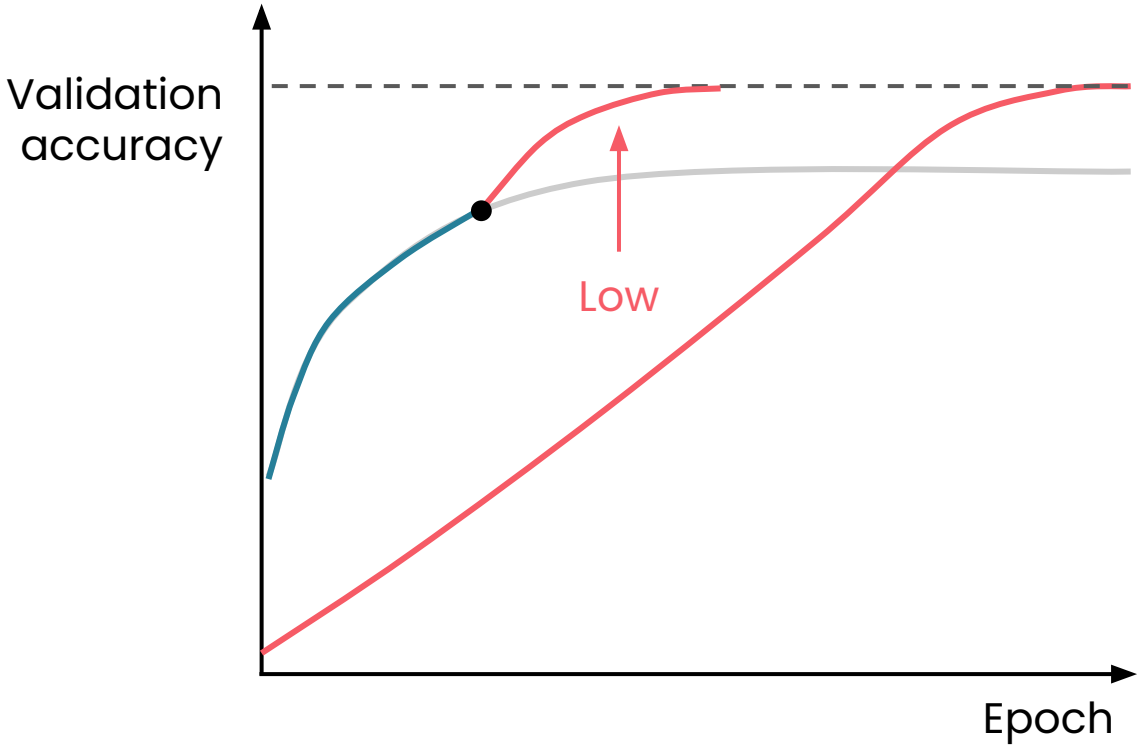
Accuracy per epoch



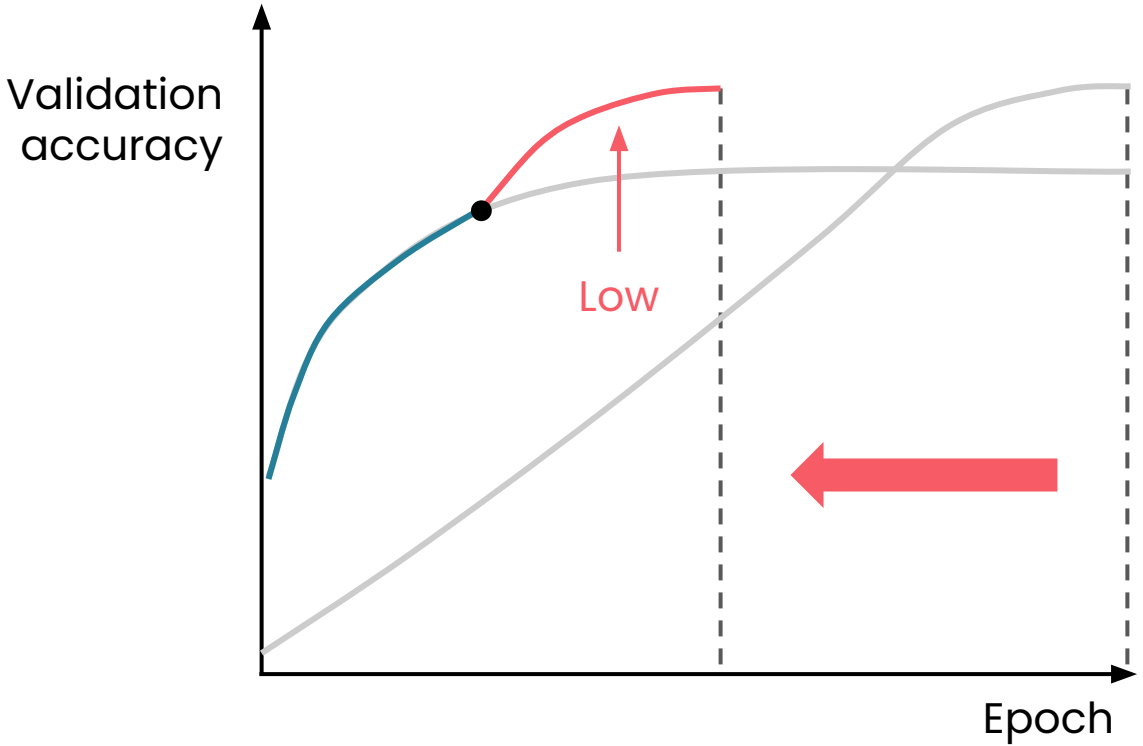
Accuracy per epoch



Accuracy per epoch



Accuracy per epoch



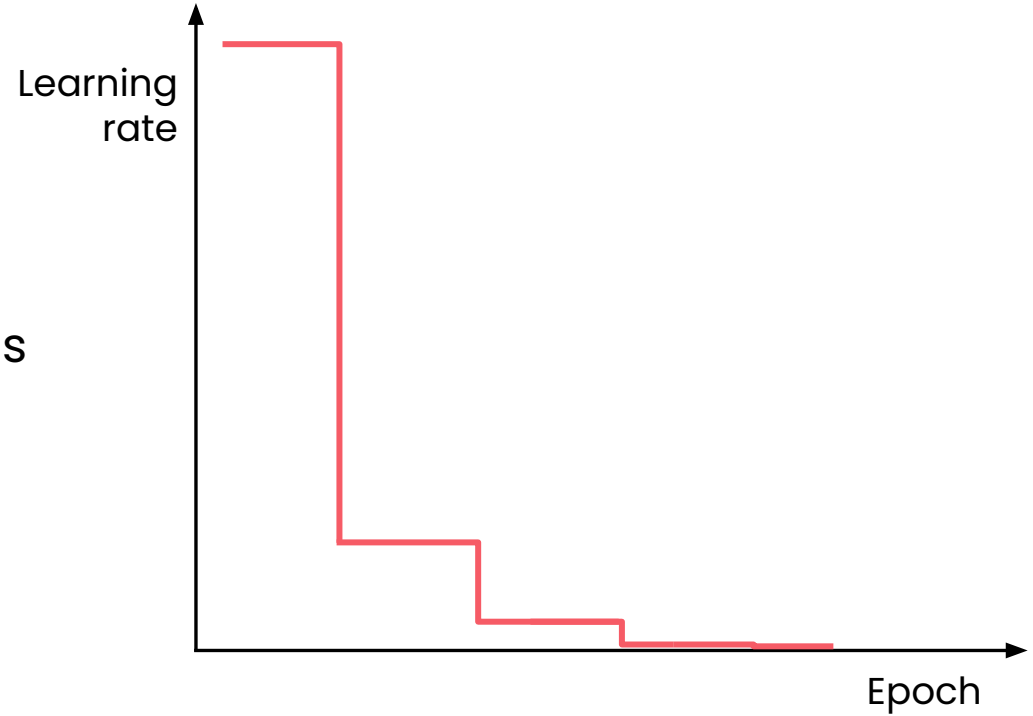
Learning rate schedulers

1. StepLR
2. ReduceLROnPlateau
3. CosineAnnealingLR

StepLR

Gamma: Factor of rate decay

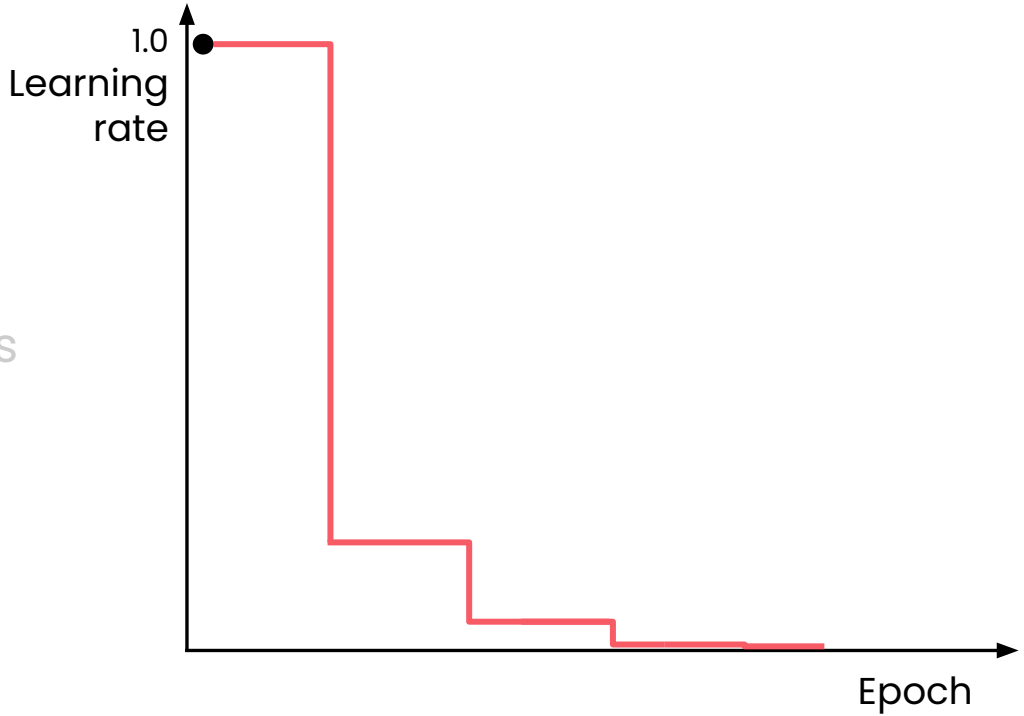
Step size: Number of epochs



StepLR

Gamma: Factor of rate decay

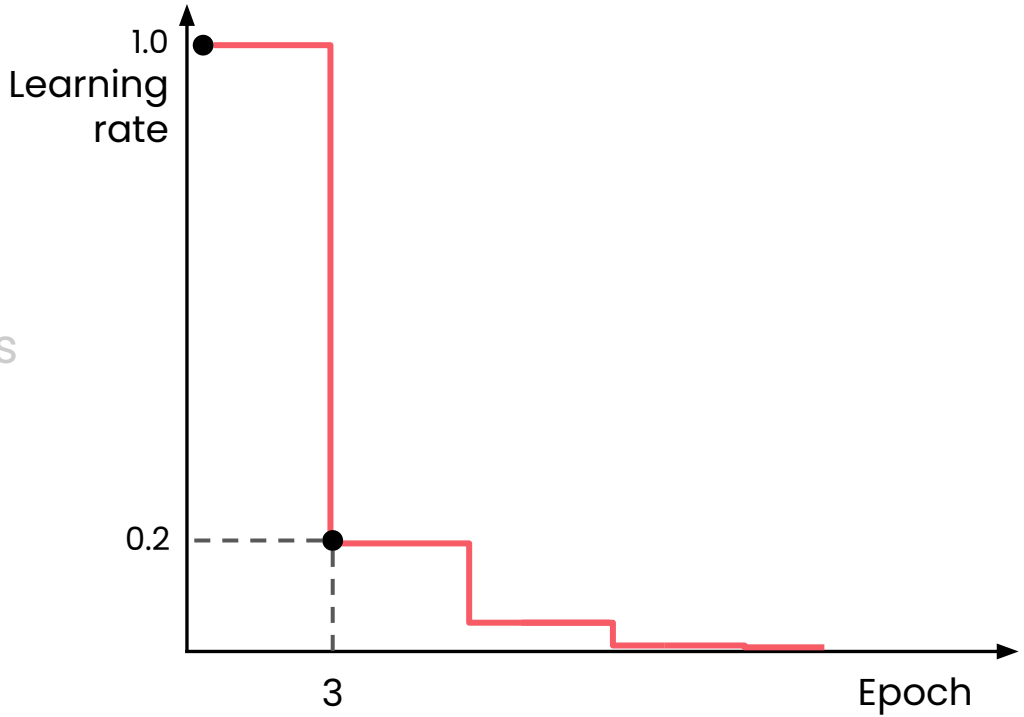
Step size: Number of epochs



StepLR

Gamma: Factor of rate decay

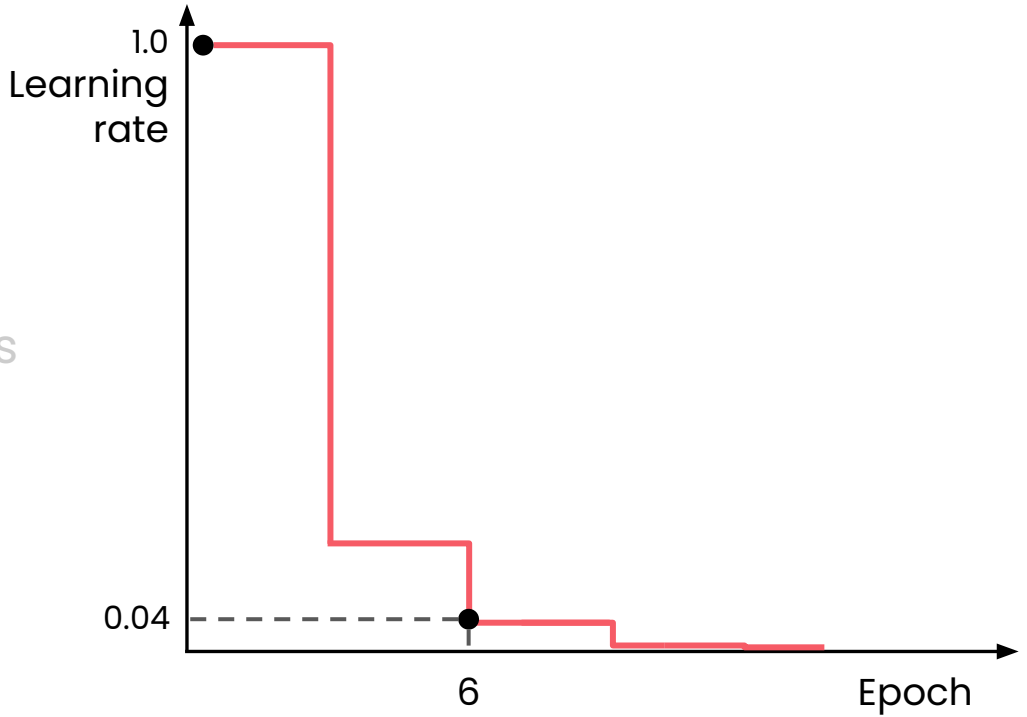
Step size: Number of epochs



StepLR

Gamma: Factor of rate decay

Step size: Number of epochs



StepLR

```
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.2)

...

scheduler.step()
```

StepLR

```
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.2)

...

scheduler.step()
```

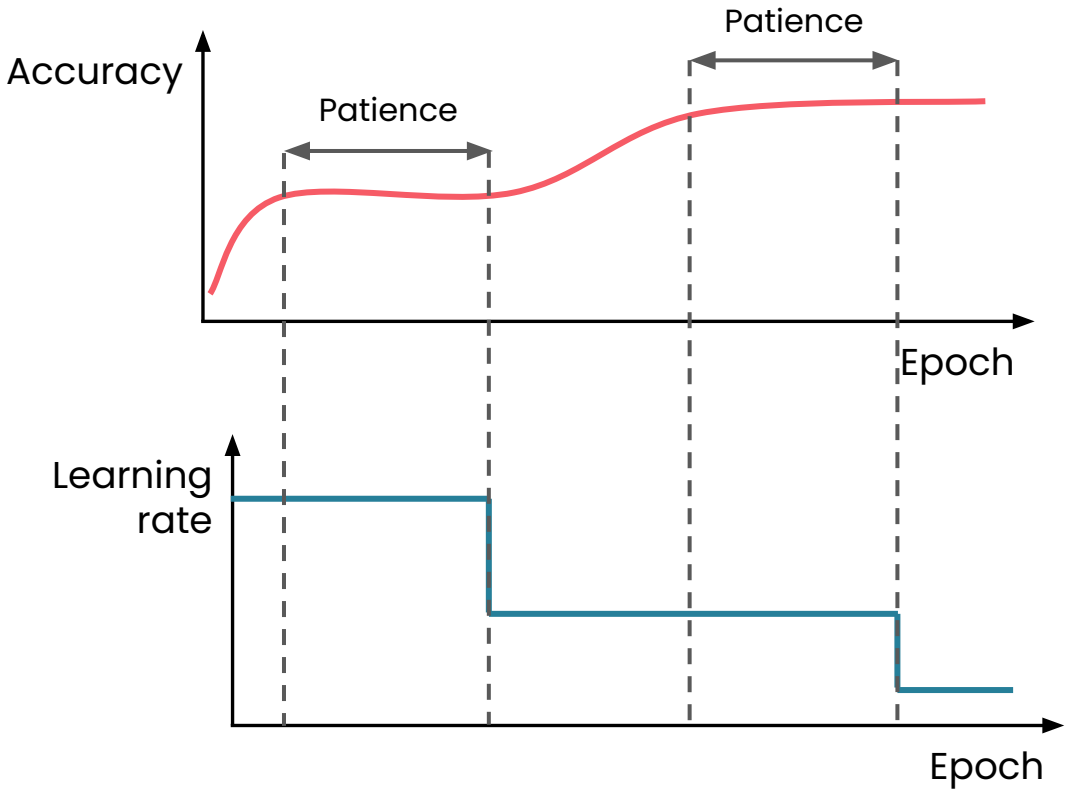
StepLR

```
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.2)

...

scheduler.step()
```

ReduceLROnPlateau

Factor: Factor of rate decay

Patience: Number of epochs without improvement



ReduceLROnPlateau

```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='max',
factor=0.2, patience=3)

...

scheduler.step(val_loss)
```

ReduceLROnPlateau

```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='max',
factor=0.2, patience=3)

...

scheduler.step(val_acc)
```

ReduceLROnPlateau

```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='max',
factor=0.2, patience=3)

...

scheduler.step(val_acc)
```

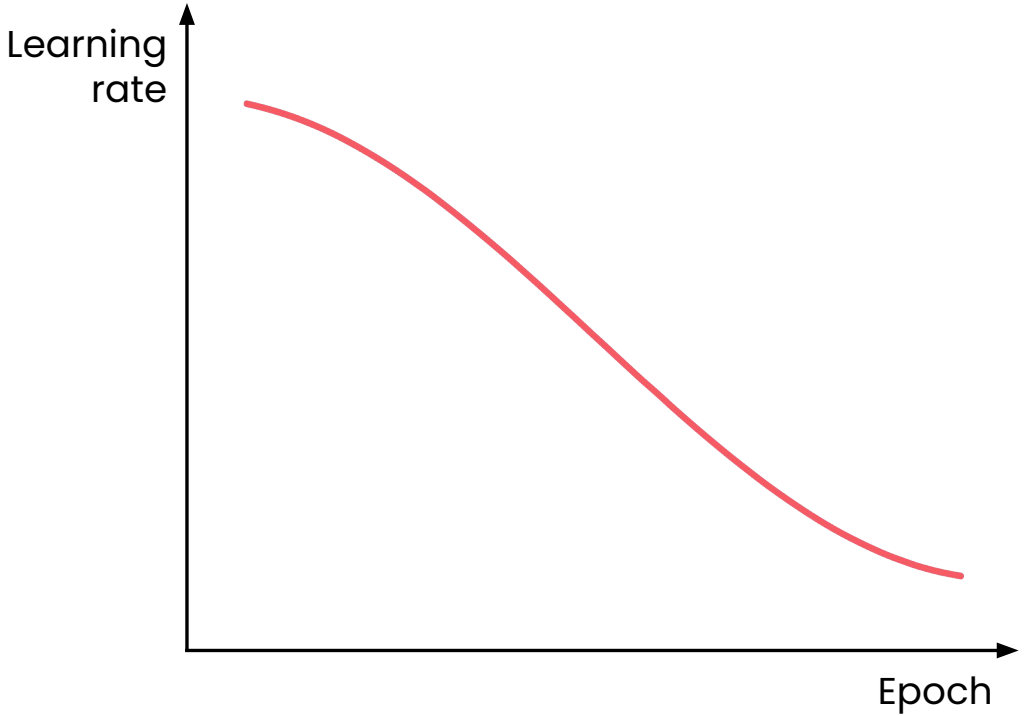
ReduceLROnPlateau

```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='max',
factor=0.2, patience=3)

...

scheduler.step(val_acc)
```

CosineAnnealingLR

T_max: Max number of iterations

eta_min: Min learning rate



CosineAnnealingLR

```
scheduler_cosine = optim.lr_scheduler.CosineAnnealingLR(optimizer,
T_max=n_epochs, eta_min=0.0002)

...

scheduler.step()
```

CosineAnnealingLR

```
scheduler_cosine = optim.lr_scheduler.CosineAnnealingLR(optimizer,
T_max=n_epochs, eta_min=0.0002)

...

scheduler.step()
```

CosineAnnealingLR

```
scheduler_cosine = optim.lr_scheduler.CosineAnnealingLR(optimizer,
T_max=n_epochs, eta_min=0.0002)

...

scheduler.step()
```

DeepLearning.AI

Tuning Hyperparameters

Hyperparameter optimization

Tunable hyperparameters

Architectural

- Number of layers
- Number of neurons per layer
- Activation function

Tunable hyperparameters

Architectural

- Number of layers
- Number of neurons per layer
- Activation function

Training

- Learning rate + schedulers
- Optimizer
- Batch size

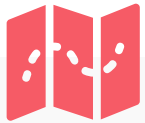
Tunable hyperparameters

Architectural

- Number of layers
- Number of neurons per layer
- Activation function

Training

- Learning rate + schedulers
- Optimizer
- Batch size



Regularization

- Weight decay
- Dropout
- Early stopping
- Batch normalization

Tunable hyperparameters

Architectural

- Number of layers
- Number of neurons per layer
- Activation function

Training

- Learning rate + schedulers
- Optimizer
- Batch size

Regularization

- Weight decay
- Dropout
- Early stopping
- Batch normalization

And more...!

Tunable hyperparameters

Architectural

- Number of layers
- Number of neurons per layer
- Activation function

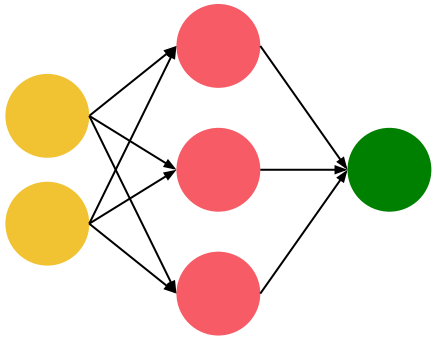
Training

- Learning rate + schedulers
- Optimizer
- Batch size

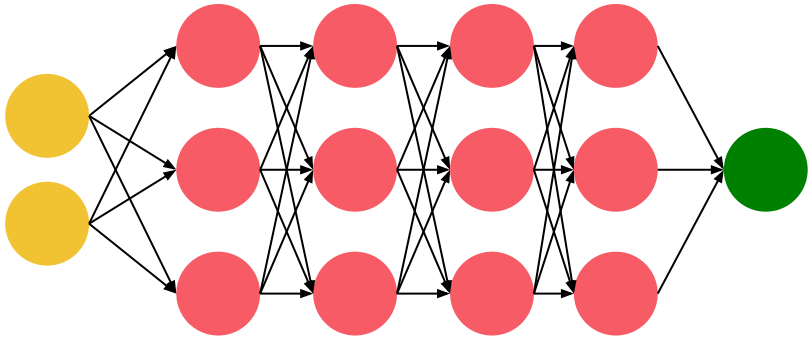
Regularization

- Weight decay
- Dropout
- Early stopping
- Batch normalization

Number of layers

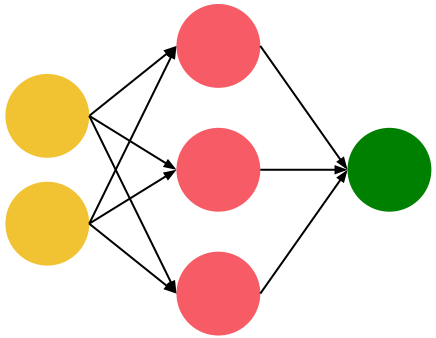


Shallow

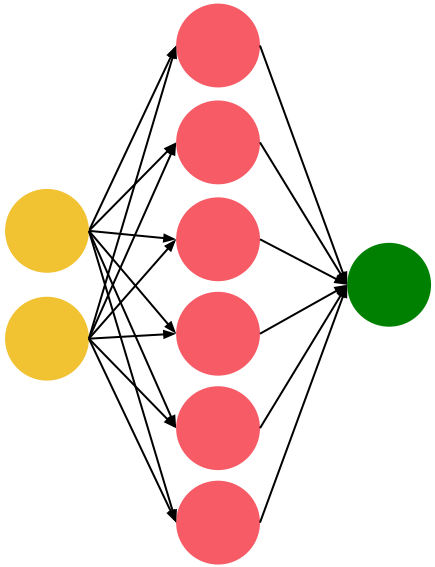


Deep

Number of neurons per layer



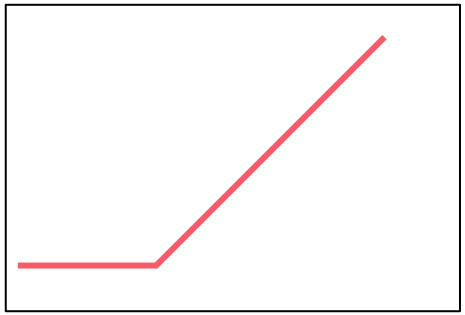
Few



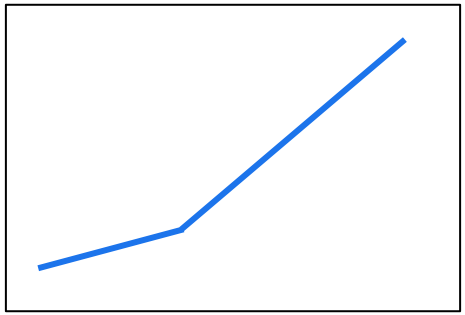
Many

Activation functions

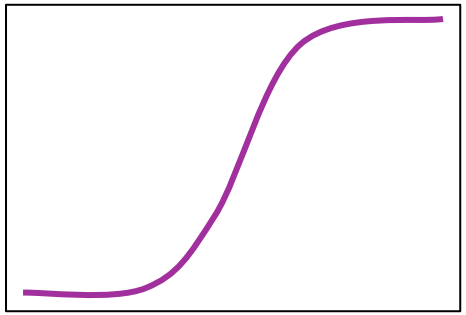
ReLU



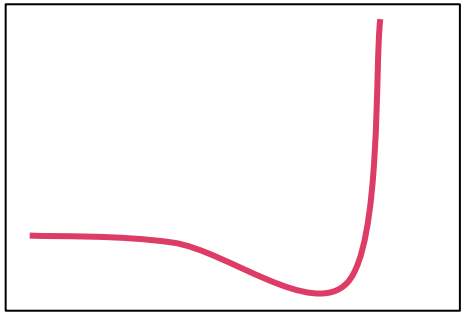
Leaky ReLU



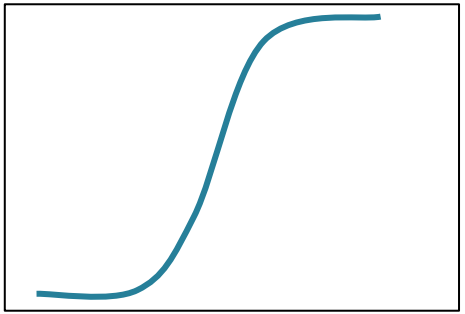
Tanh



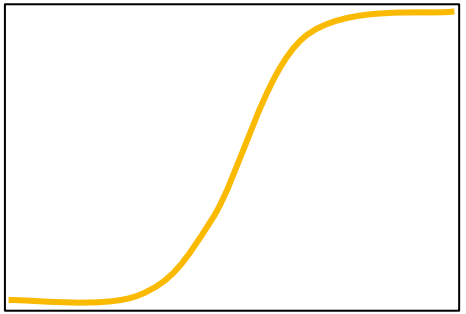
SiLU



Sigmoid

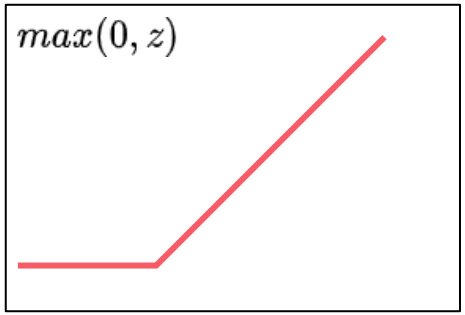


Softmax



Activation functions

ReLU

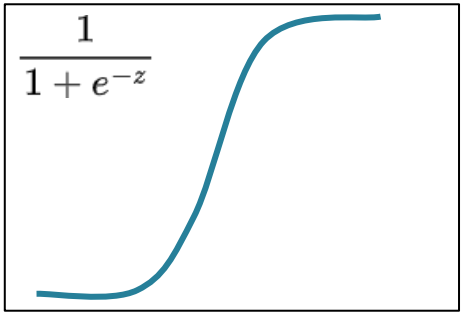


Leaky ReLU

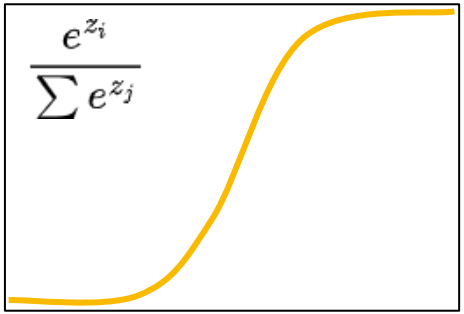
Tanh

SiLU

Sigmoid



Softmax



Tunable hyperparameters

Architectural

- Number of layers
- Number of neurons per layer
- Activation function

Training

- Learning rate + schedulers
- Optimizer
- Batch size

Regularization

- Weight decay
- Dropout
- Early stopping
- Batch normalization

Optimizer

Gradient Descent	RMSProp	Adam	Stochastic Gradient Descent (SGD)
Mini-Batch Gradient Descent	SGD with Momentum	Adaptive Gradient Descent	AdaDelta

Optimizer

Gradient Descent	RMSProp	Adam	Stochastic Gradient Descent (SGD)
Mini-Batch Gradient Descent	SGD with Momentum	Adaptive Gradient Descent	AdaDelta

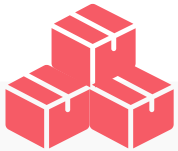
Optimizer

Gradient Descent	RMSProp	Adam	Stochastic Gradient Descent (SGD)
Mini-Batch Gradient Descent	SGD with Momentum	Adaptive Gradient Descent	AdaDelta

Optimizer

Gradient Descent	RMSProp	Adam	Stochastic Gradient Descent (SGD)
Mini-Batch Gradient Descent	SGD with Momentum	Adaptive Gradient Descent	AdaDelta

Batch size



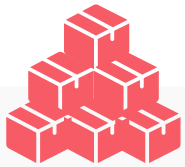
Small

- Longer to train
- Less memory
- Escape local minima

Batch size

Small

- Longer to train
- Less memory
- Escape local minima



Large

- Quicker training
- More memory
- Stuck in local minima

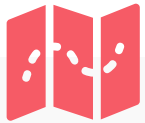
Tunable hyperparameters

Architectural

- Number of layers
- Number of neurons per layer
- Activation function

Training

- Learning rate + schedulers
- Optimizer
- Batch size



Regularization

- Weight decay
- Dropout
- Early stopping
- Batch normalization

Weight decay

Penalizing large weights by adding the squares of the weights to the loss function

$$Loss = Error(Y - \hat{Y}) + \lambda \sum_{1}^{n} w_i^2$$

Weight decay

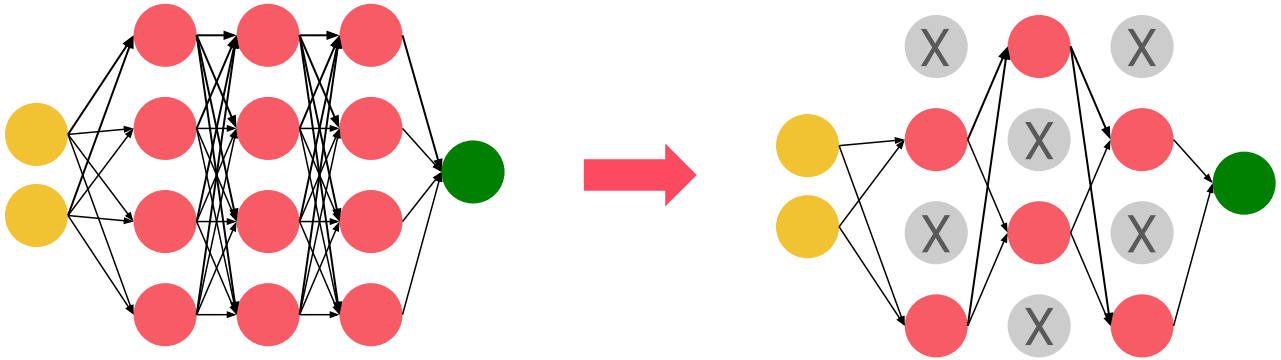
Penalizing large weights by adding the squares of the weights to the loss function

$$Loss = Error(Y - \hat{Y}) + \lambda \sum_{1}^{n} w_i^2$$

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)
```

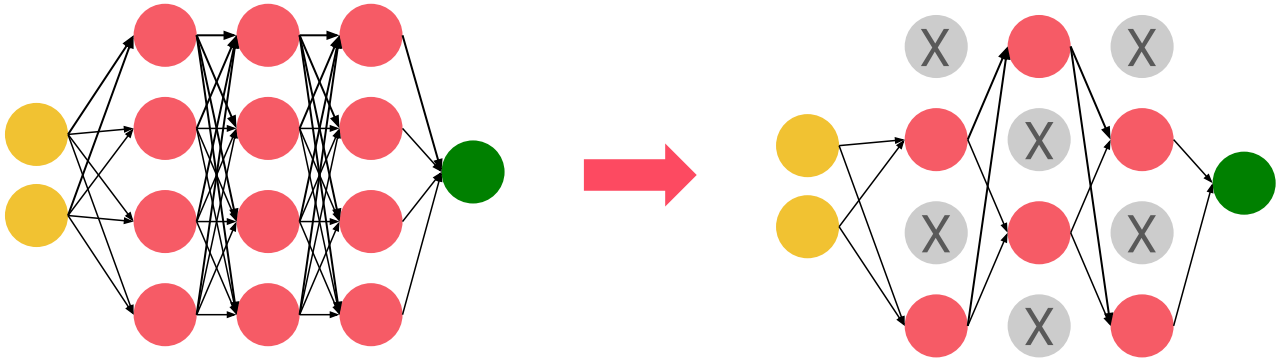
Dropout

Randomly disabling a fraction of neurons during training.



Dropout

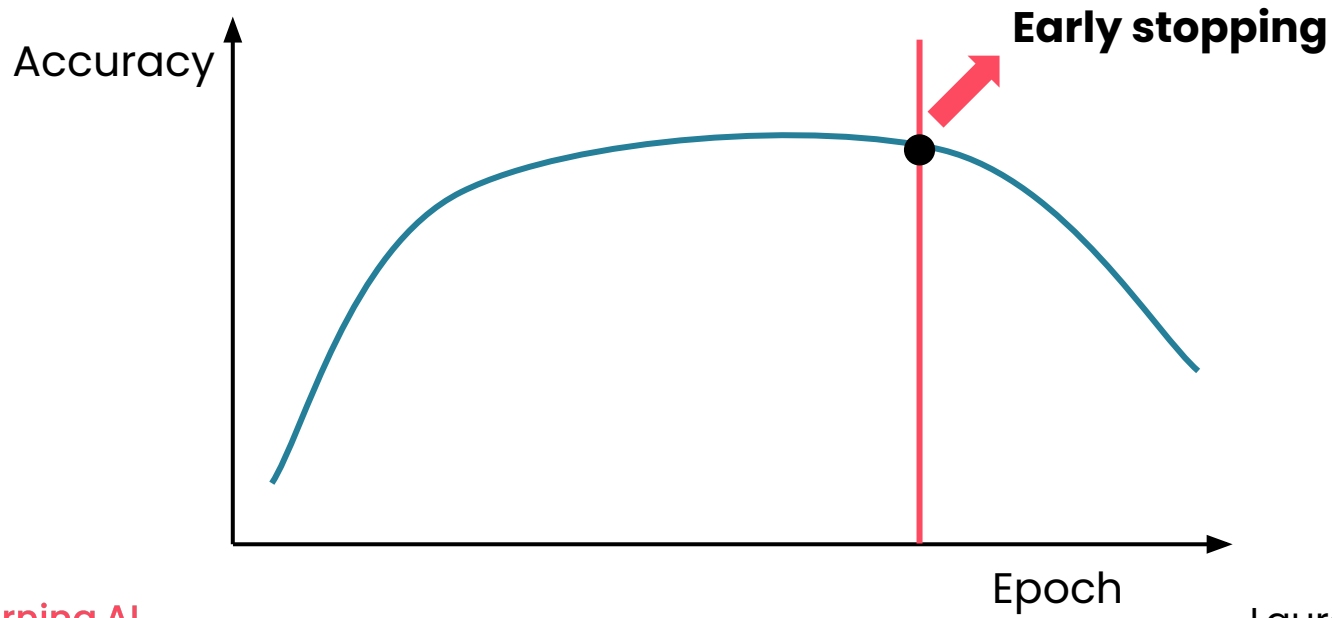
Randomly disabling a fraction of neurons during training.



```
nn.Dropout(p=0.5)  # 50% dropout rate
```

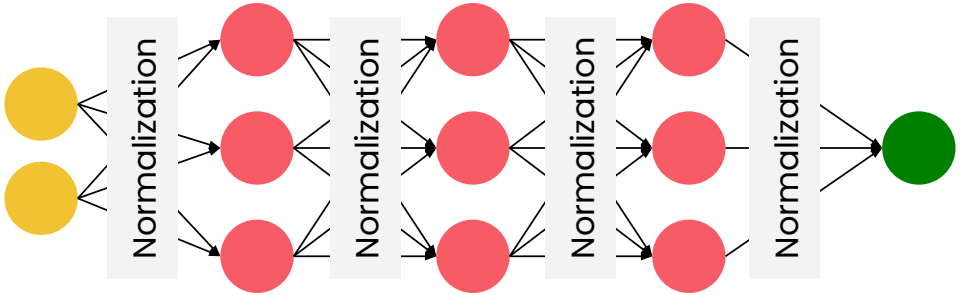
Early stopping

Number of epochs to wait if improvement stalls



Batch normalization

Normalizes the inputs to a layer




```
nn.BatchNorm2d(64)  # For a convolutional layer with 64 channels
```

Where to start?

Establish a baseline

Where to start?

Establish a baseline

Use defaults

Where to start?

Establish a baseline

Use defaults

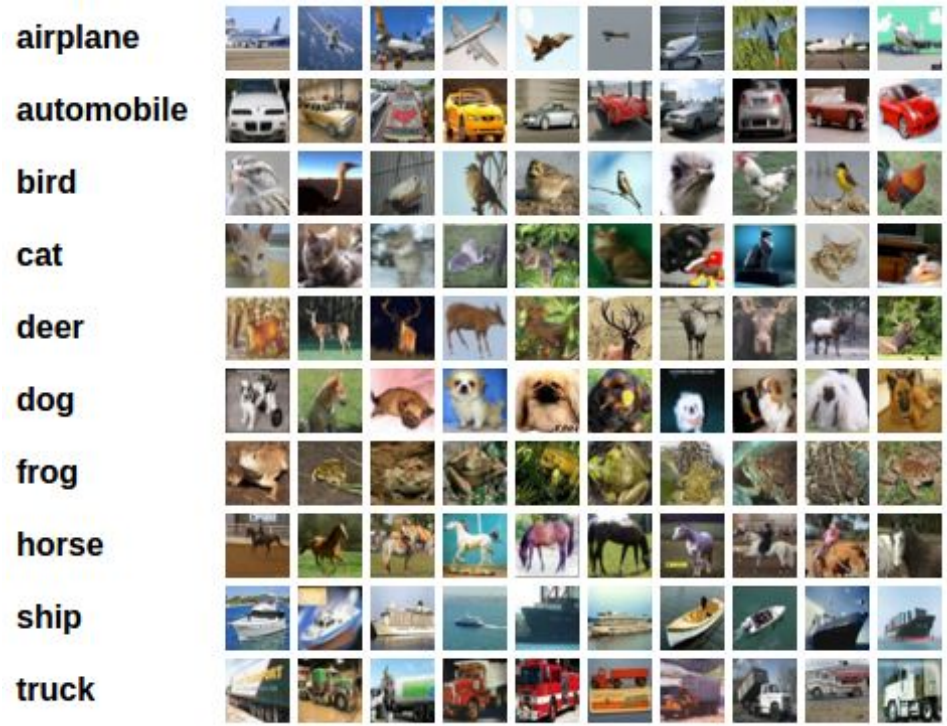
Reference points in literature

DeepLearning.AI

Flexible Architecture Design

Hyperparameter optimization

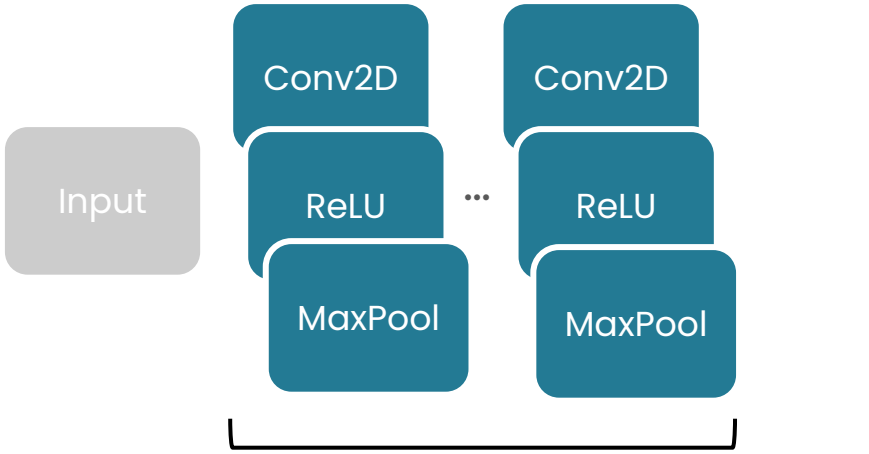
CIFAR-10 dataset



University of Toronto

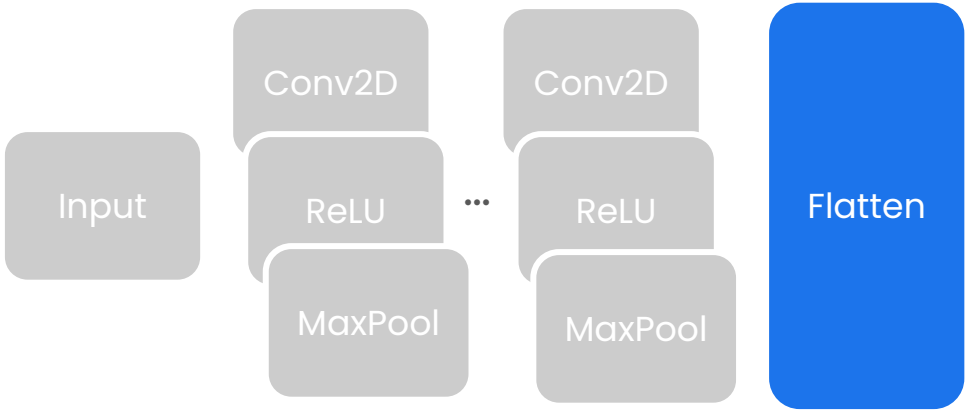
A flexible architecture

A flexible architecture

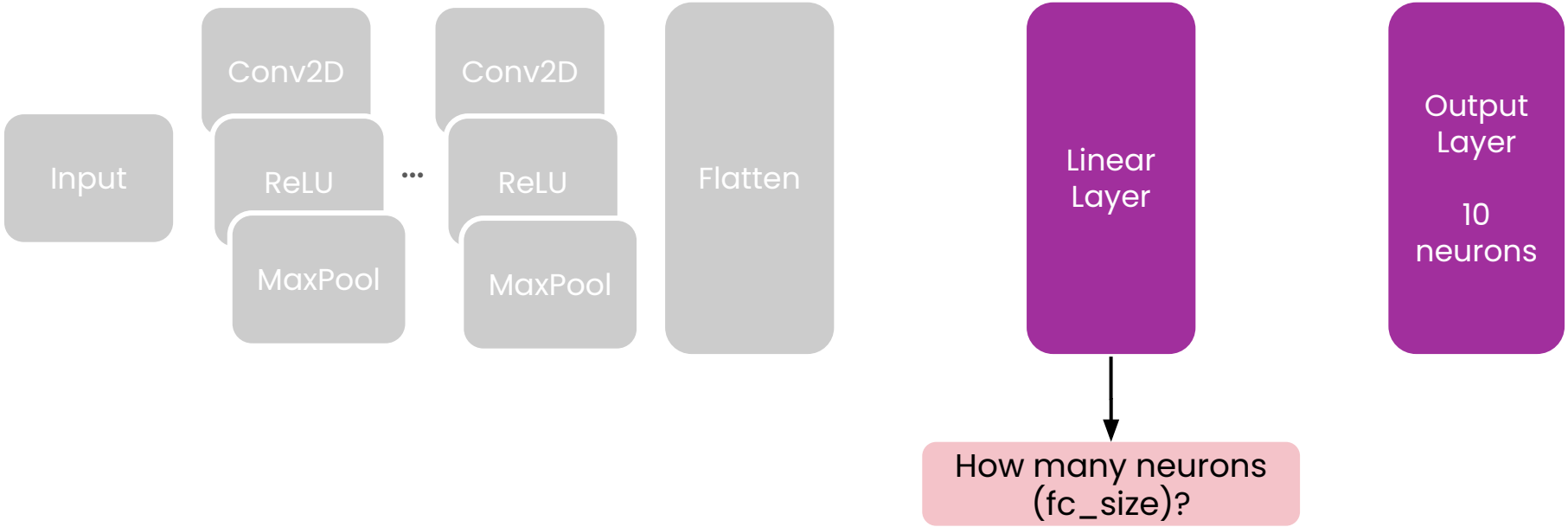


How many blocks (n_layers)?
Number of filters per block (n_filters)?
Kernel size (kernel_sizes)?

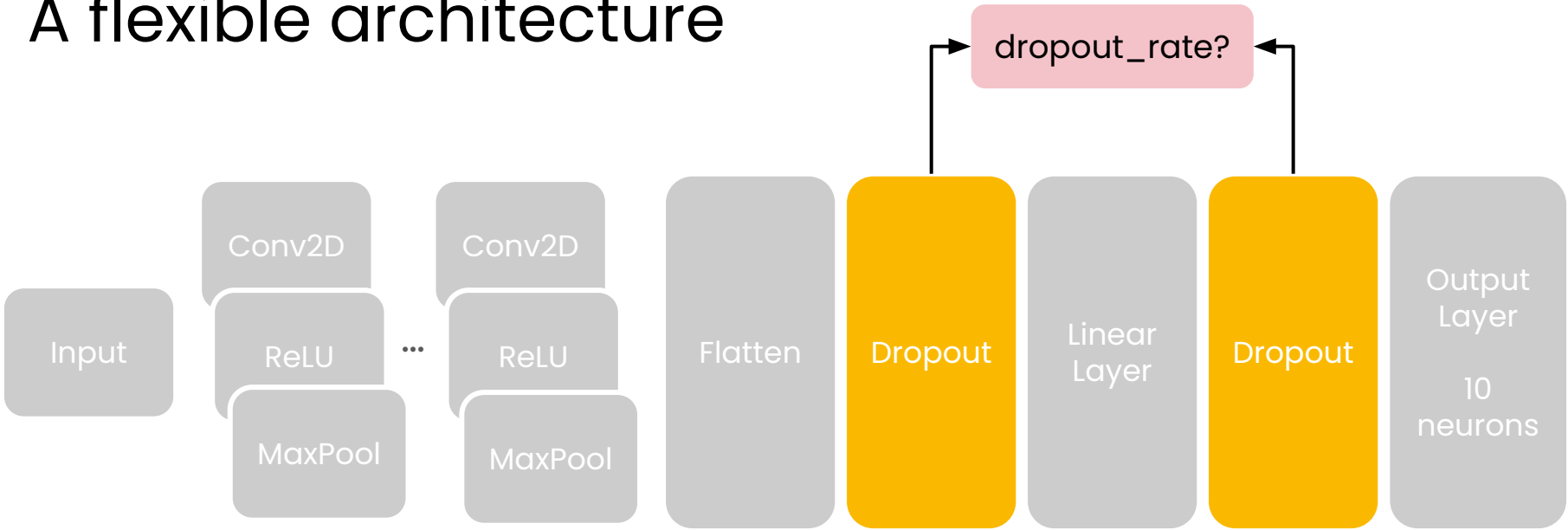
A flexible architecture



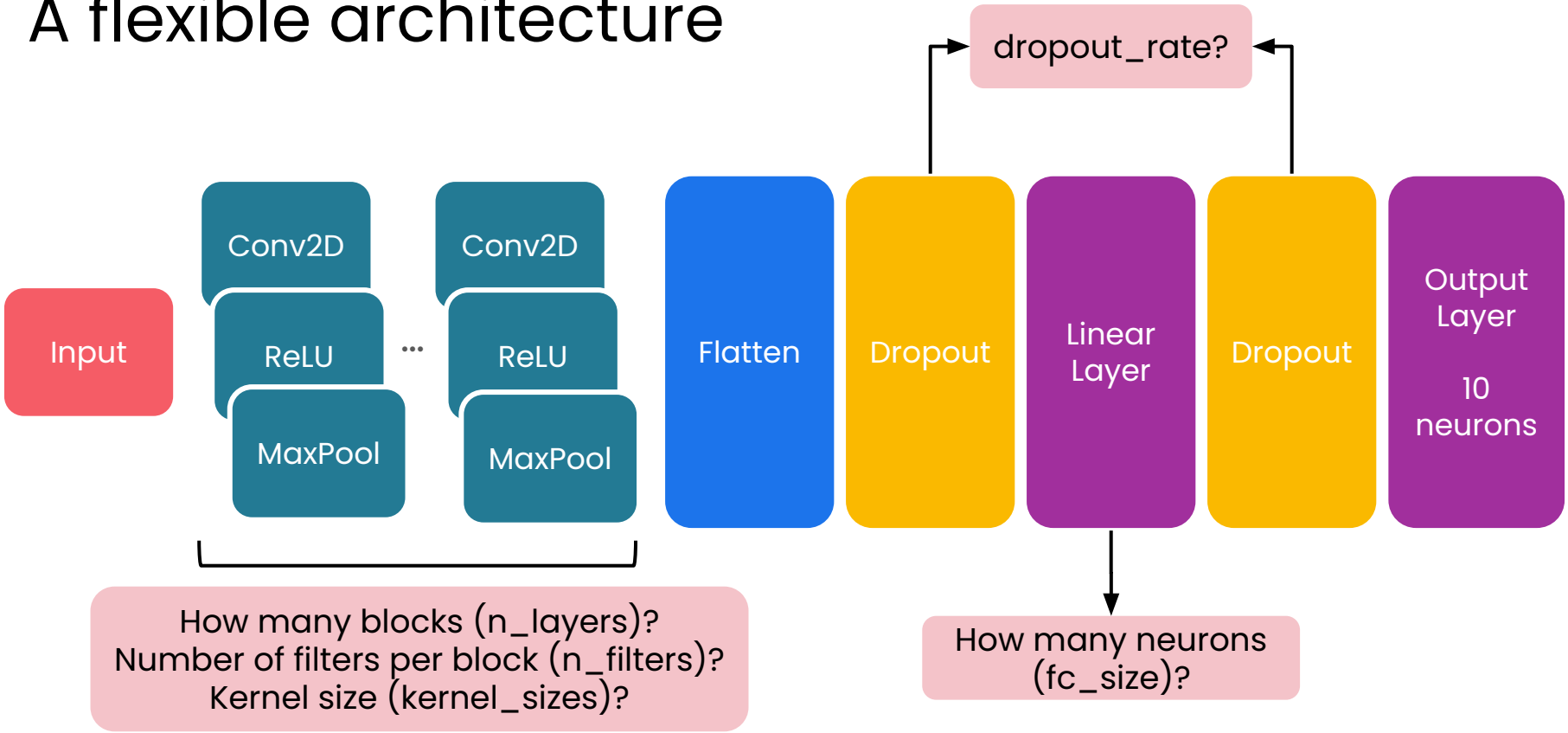
A flexible architecture



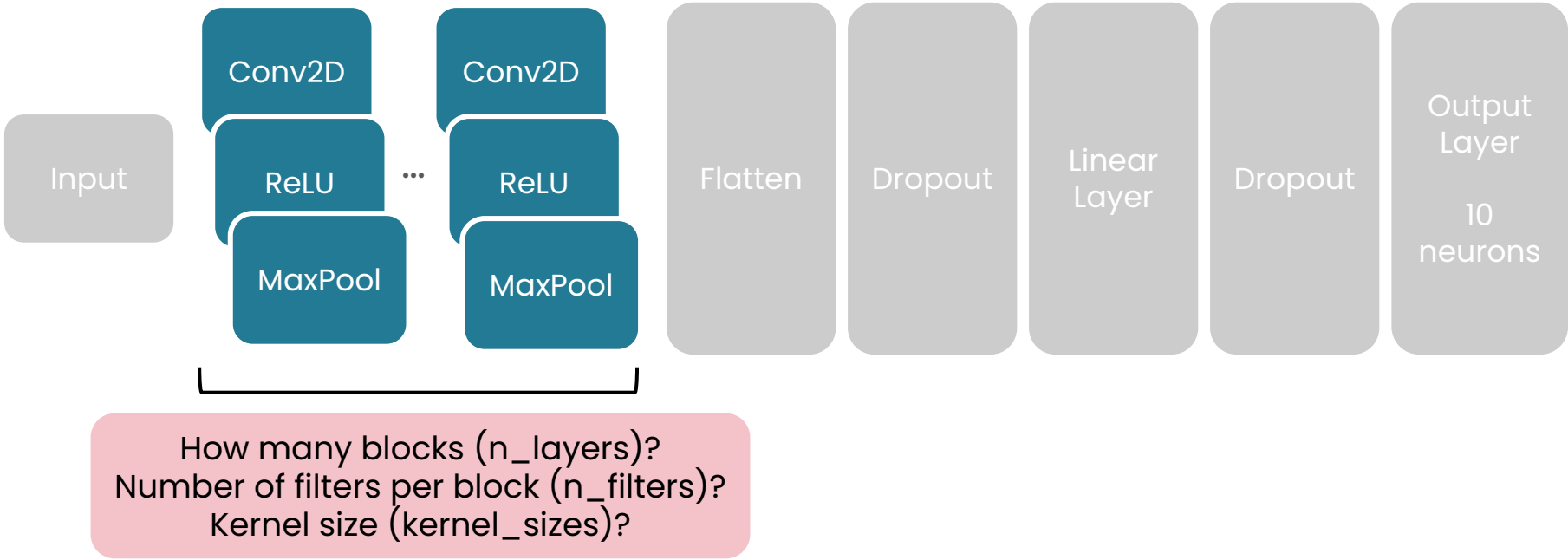
A flexible architecture



A flexible architecture



Convolutional Blocks



```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):
        super(FlexibleCNN, self).__init__()

        blocks = []
        in_channels = 3  # RGB input

        for i in range(n_layers):
            out_channels = n_filters[i]
            kernel_size = kernel_sizes[i]
            padding = (kernel_size - 1) // 2  # 'same' padding

            block = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size, padding=padding),
                nn.ReLU(),
                nn.MaxPool2d(kernel_size=2, stride=2)
            )

            blocks.append(block)
            in_channels = out_channels
```

```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):
        super(FlexibleCNN, self).__init__()

        blocks = []
        in_channels = 3  # RGB input

        for i in range(n_layers):
            out_channels = n_filters[i]
            kernel_size = kernel_sizes[i]
            padding = (kernel_size - 1) // 2  # 'same' padding

            block = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size, padding=padding),
                nn.ReLU(),
                nn.MaxPool2d(kernel_size=2, stride=2)
            )

            blocks.append(block)
            in_channels = out_channels
```

```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):
        super(FlexibleCNN, self).__init__()

        blocks = []
        in_channels = 3  # RGB input

        for i in range(n_layers):
            out_channels = n_filters[i]
            kernel_size = kernel_sizes[i]
            padding = (kernel_size - 1) // 2  # 'same' padding

            block = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size, padding=padding),
                nn.ReLU(),
                nn.MaxPool2d(kernel_size=2, stride=2)
            )

            blocks.append(block)
            in_channels = out_channels
```

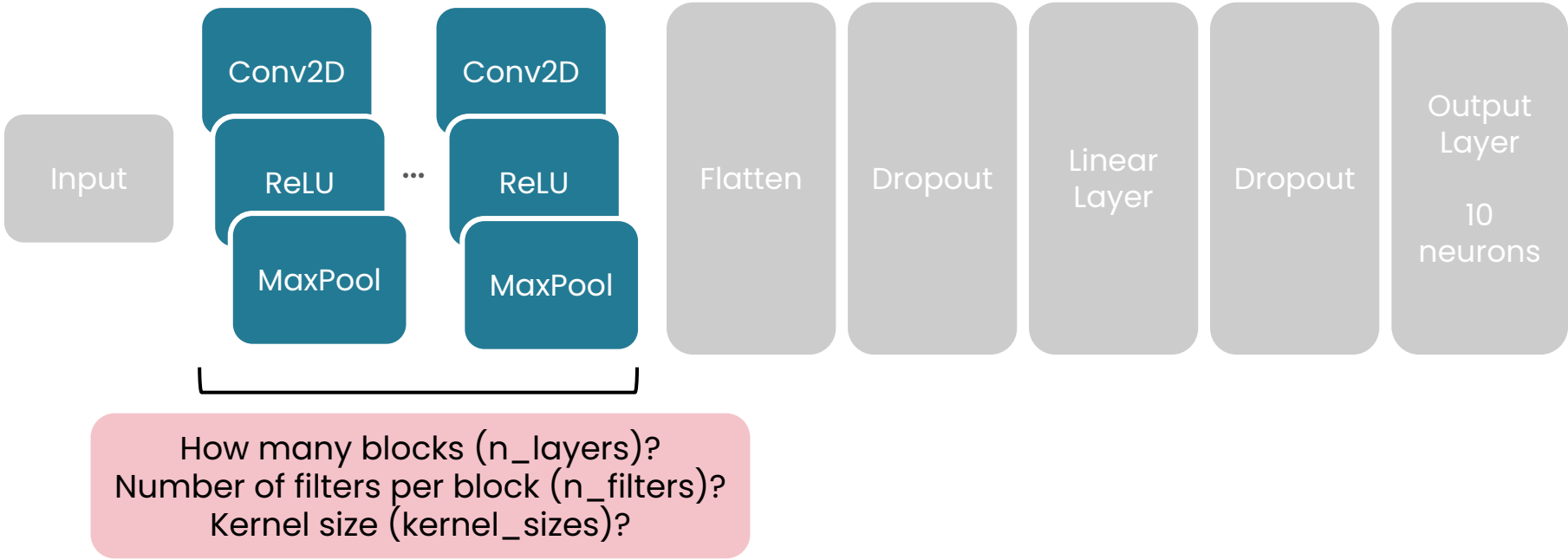
```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):
        super(FlexibleCNN, self).__init__()

        blocks = []
        in_channels = 3  # RGB input

        for i in range(n_layers):
            out_channels = n_filters[i]
            kernel_size = kernel_sizes[i]
            padding = (kernel_size - 1) // 2  # 'same' padding

            block = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size, padding=padding),
                nn.ReLU(),
                nn.MaxPool2d(kernel_size=2, stride=2)
            )

            blocks.append(block)
            in_channels = out_channels
```

```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):
        super(FlexibleCNN, self).__init__()

        blocks = []
        in_channels = 3  # RGB input

        for i in range(n_layers):
            out_channels = n_filters[i]
            kernel_size = kernel_sizes[i]
            padding = (kernel_size - 1) // 2  # 'same' padding

            block = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size, padding=padding),
                nn.ReLU(),
                nn.MaxPool2d(kernel_size=2, stride=2)
            )

            blocks.append(block)
            in_channels = out_channels
```

```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):
        super(FlexibleCNN, self).__init__()

        blocks = []
        in_channels = 3  # RGB input

        for i in range(n_layers):
            out_channels = n_filters[i]
            kernel_size = kernel_sizes[i]
            padding = (kernel_size - 1) // 2  # 'same' padding

            block = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size, padding=padding),
                nn.ReLU(),
                nn.MaxPool2d(kernel_size=2, stride=2)
            )

            blocks.append(block)
            in_channels = out_channels
```

```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):
        super(FlexibleCNN, self).__init__()

        blocks = []
        in_channels = 3  # RGB input

        for i in range(n_layers):
            out_channels = n_filters[i]
            kernel_size = kernel_sizes[i]
            padding = (kernel_size - 1) // 2  # 'same' padding

            block = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size, padding=padding),
                nn.ReLU(),
                nn.MaxPool2d(kernel_size=2, stride=2)
            )

            blocks.append(block)
            in_channels = out_channels
```

Convolutional Blocks



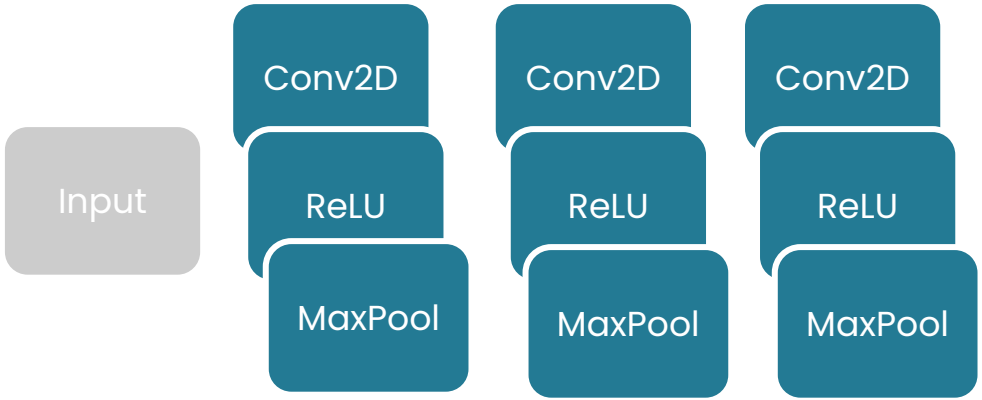
```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):
        super(FlexibleCNN, self).__init__()

        blocks = []
        in_channels = 3  # RGB input

        for i in range(n_layers):
            out_channels = n_filters[i]
            kernel_size = kernel_sizes[i]
            padding = (kernel_size - 1) // 2  # 'same' padding

            block = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size, padding=padding),
                nn.ReLU(),
                nn.MaxPool2d(kernel_size=2, stride=2)
            )

            blocks.append(block)
            in_channels = out_channels
```

```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):
        super(FlexibleCNN, self).__init__()

        blocks = []
        in_channels = 3  # RGB input

        for i in range(n_layers):
            out_channels = n_filters[i]
            kernel_size = kernel_sizes[i]
            padding = (kernel_size - 1) // 2  # 'same' padding

            block = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size, padding=padding),
                nn.ReLU(),
                nn.MaxPool2d(kernel_size=2, stride=2)
            )

            blocks.append(block)
            in_channels = out_channels
```

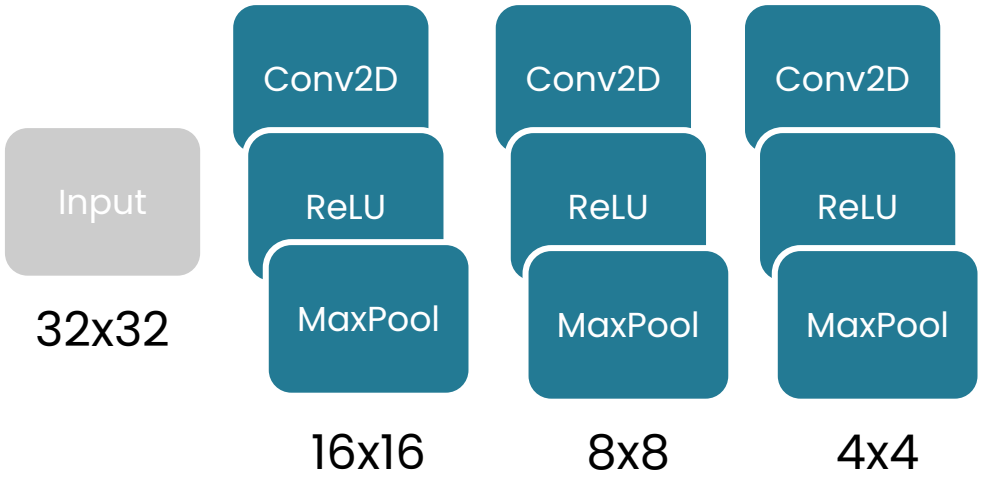
```
n_filters = [64, 32]
kernel_sizes = [3, 5]
```

```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):
        super(FlexibleCNN, self).__init__()

        blocks = []
        in_channels = 3  # RGB input

        for i in range(n_layers):
            out_channels = n_filters[i]
            kernel_size = kernel_sizes[i]
            padding = (kernel_size - 1) // 2  # 'same' padding

            block = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size, padding=padding),
                nn.ReLU(),
                nn.MaxPool2d(kernel_size=2, stride=2)
            )

            blocks.append(block)
            in_channels = out_channels
```

Spatial dimensions

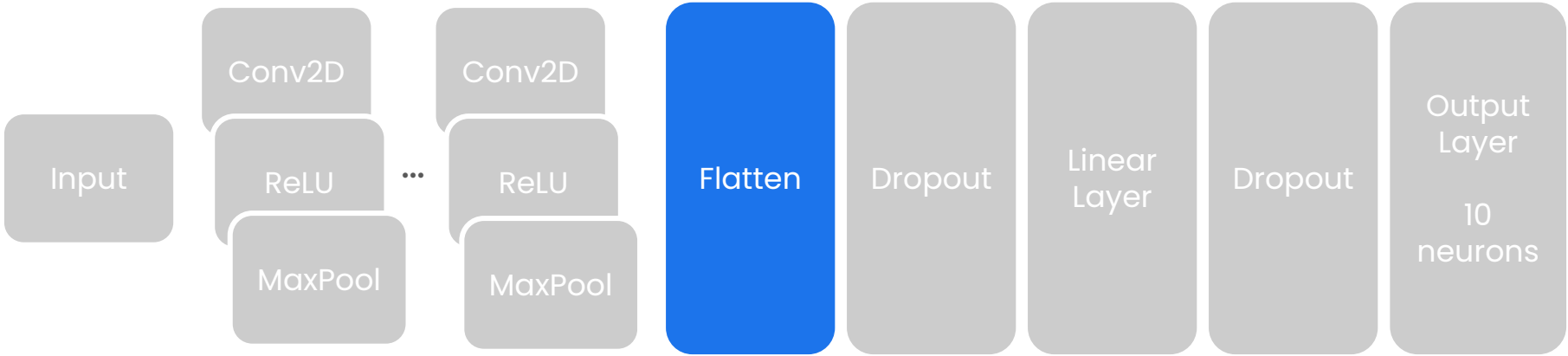


Spatial dimensions

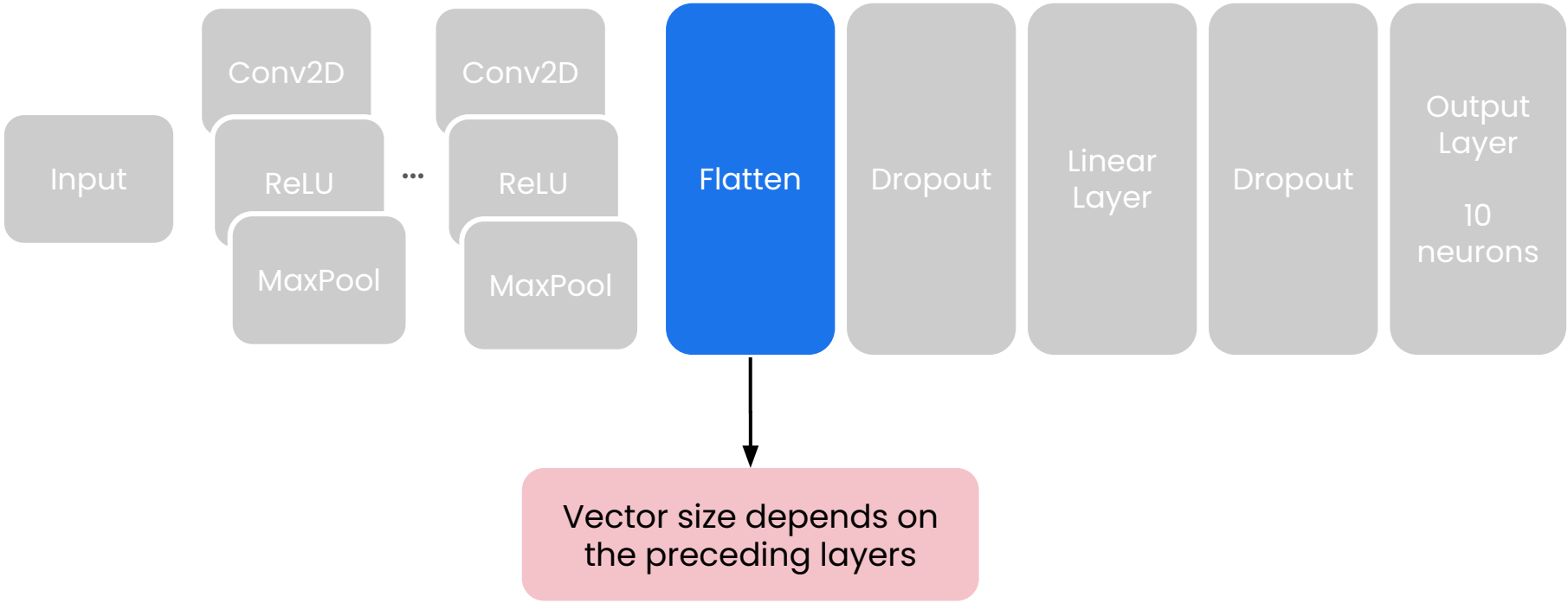



```
nn.MaxPool2d(kernel_size=2, stride=2)
```

Flatten layer



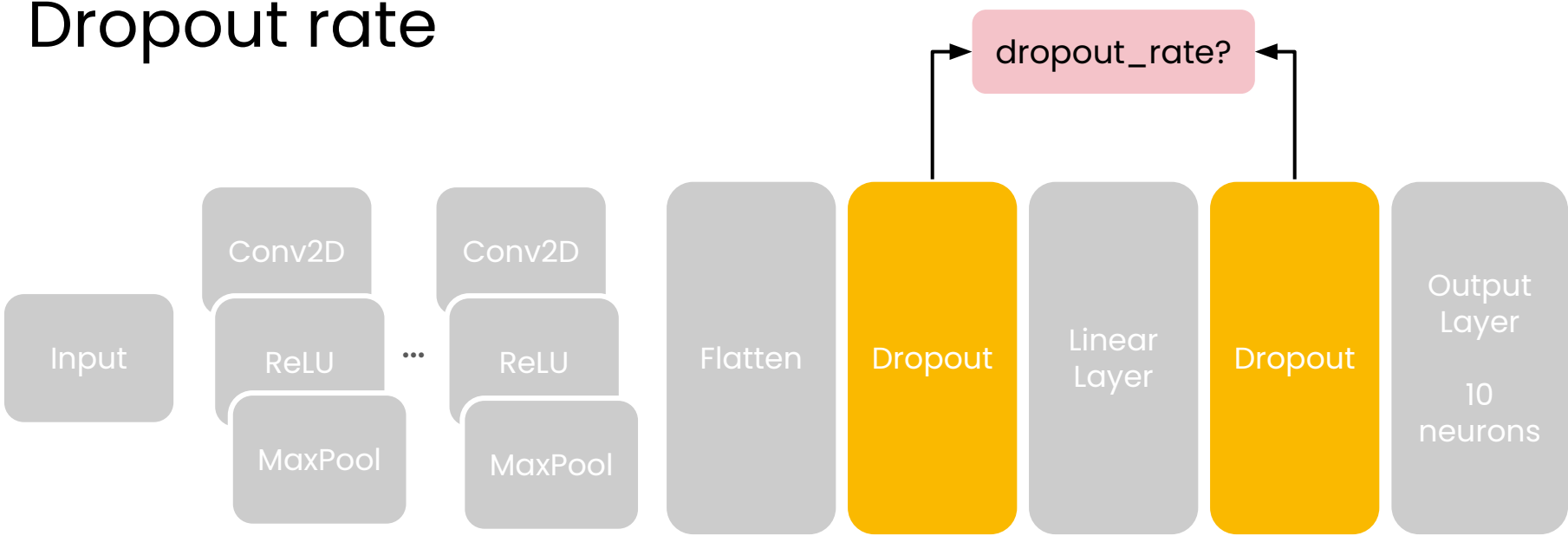
Flatten layer



```
def forward(self, x):
        # Extract the device from the input tensor
        device = x.device

        # Apply feature extractor
        x = self.features(x)

        # Flatten for the classifier
        flattened = torch.flatten(x, 1)
        flattened_size = flattened.size(1)

        # Dynamically initialize classifier on first forward pass
        if self.classifier is None:
            self._create_classifier(flattened_size, device)

        return self.classifier(flattened)
```

```
def forward(self, x):
        # Extract the device from the input tensor
        device = x.device

        # Apply feature extractor
        x = self.features(x)

        # Flatten for the classifier
        flattened = torch.flatten(x, 1)
        flattened_size = flattened.size(1)

        # Dynamically initialize classifier on first forward pass
        if self.classifier is None:
            self._create_classifier(flattened_size, device)

        return self.classifier(flattened)
```

Dropout rate



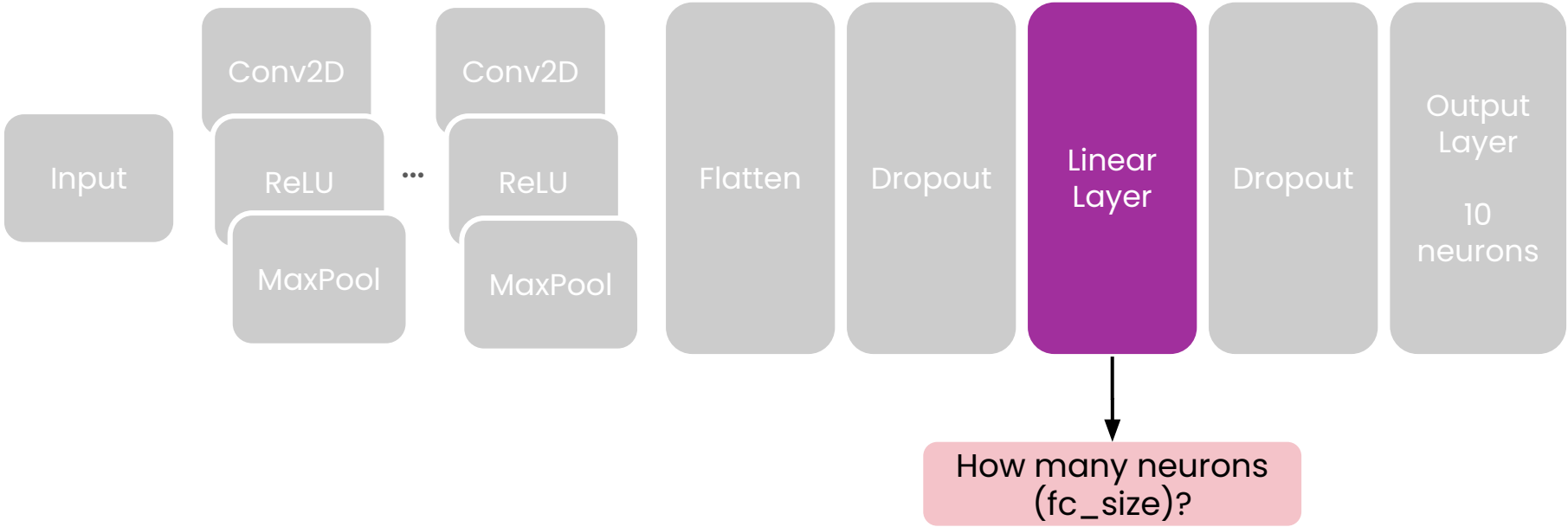
Parametrizing dropout

```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):

    ...

    def _create_classifier(self, flattened_size, device):
            # Dynamically create the classifier based on the flattened size

            self.classifier = nn.Sequential(
                nn.Dropout(self.dropout_rate),
                nn.Linear(flattened_size, self.fc_size),
                nn.ReLU(inplace=True),
                nn.Dropout(self.dropout_rate),
                nn.Linear(self.fc_size, 10)  # CIFAR-10: 10 classes
            ).to(device)
```

Parametrizing dropout

```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):

        ...

        def _create_classifier(self, flattened_size, device):
            # Dynamically create the classifier based on the flattened size

            self.classifier = nn.Sequential(
                nn.Dropout(self.dropout_rate),
                nn.Linear(flattened_size, self.fc_size),
                nn.ReLU(inplace=True),
                nn.Dropout(self.dropout_rate),
                nn.Linear(self.fc_size, 10)  # CIFAR-10: 10 classes
            ).to(device)
```

Number of neurons in FC layer



Number of neurons in FC layer

```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):

    ...

    def _create_classifier(self, flattened_size, device):
            # Dynamically create the classifier based on the flattened size

            self.classifier = nn.Sequential(
                nn.Dropout(self.dropout_rate),
                nn.Linear(flattened_size, self.fc_size),
                nn.ReLU(inplace=True),
                nn.Dropout(self.dropout_rate),
                nn.Linear(self.fc_size, 10)  # CIFAR-10: 10 classes
            ).to(device)
```

Number of neurons in FC layer

```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):

    ...

    def _create_classifier(self, flattened_size, device):
            # Dynamically create the classifier based on the flattened size

            self.classifier = nn.Sequential(
                nn.Dropout(self.dropout_rate),
                nn.Linear(flattened_size, self.fc_size),
                nn.ReLU(inplace=True),
                nn.Dropout(self.dropout_rate),
                nn.Linear(self.fc_size, 10)  # CIFAR-10: 10 classes
            ).to(device)
```

Number of neurons in FC layer

```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):

        ...

        def _create_classifier(self, flattened_size, device):
            # Dynamically create the classifier based on the flattened size

            self.classifier = nn.Sequential(
                nn.Dropout(self.dropout_rate),
                nn.Linear(flattened_size, self.fc_size),
                nn.ReLU(inplace=True),
                nn.Dropout(self.dropout_rate),
                nn.Linear(self.fc_size, 10)  # CIFAR-10: 10 classes
            ).to(device)
```

Number of neurons in FC layer

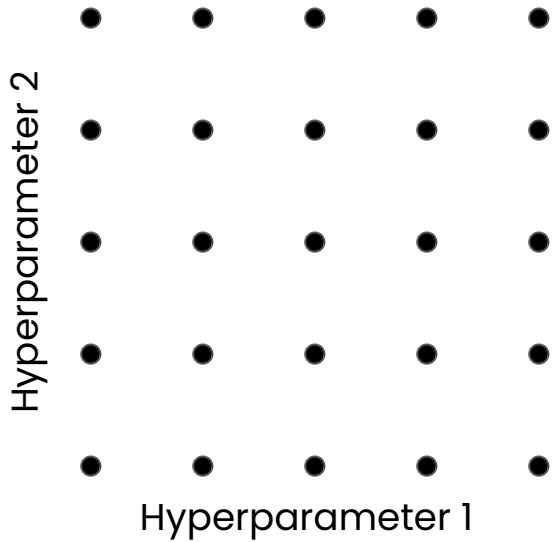
```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, kernel_sizes, dropout_rate, fc_size):

        ...

    def _create_classifier(self, flattened_size, device):
            # Dynamically create the classifier based on the flattened size

            self.classifier = nn.Sequential(
                nn.Dropout(self.dropout_rate),
                nn.Linear(flattened_size, self.fc_size),
                nn.ReLU(inplace=True),
                nn.Dropout(self.dropout_rate),
                nn.Linear(self.fc_size, 10)  # CIFAR-10: 10 classes
            ).to(device)
```

DeepLearning.AI

Hyperparameter Optimization with Optuna

Hyperparameter optimization

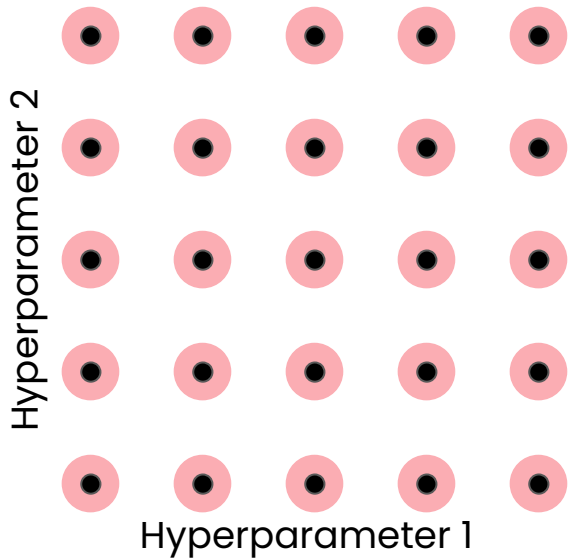
Grid search



Hyperparameter 1 → 5 values
Hyperparameter 2 → 5 values

5*5 = 25 combinations

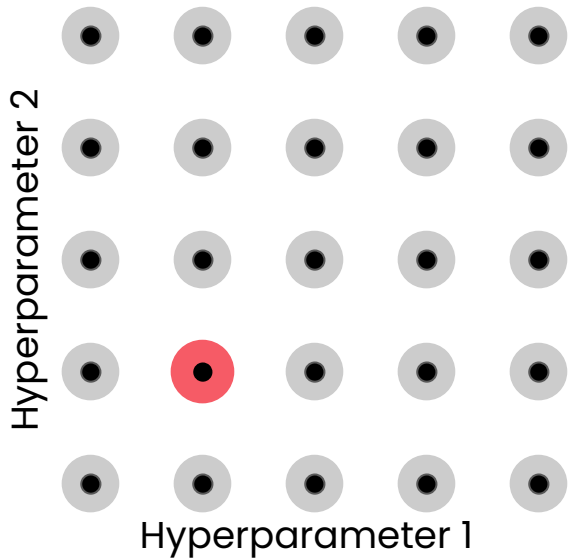
Grid search



Hyperparameter 1 → 5 values
Hyperparameter 2 → 5 values

5*5 = 25 combinations

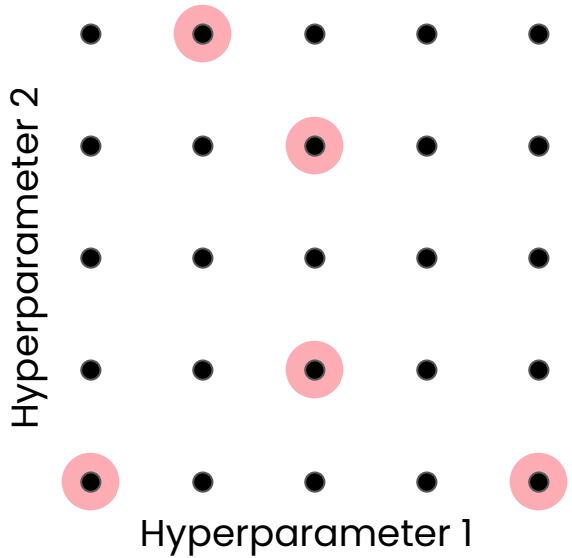
Grid search



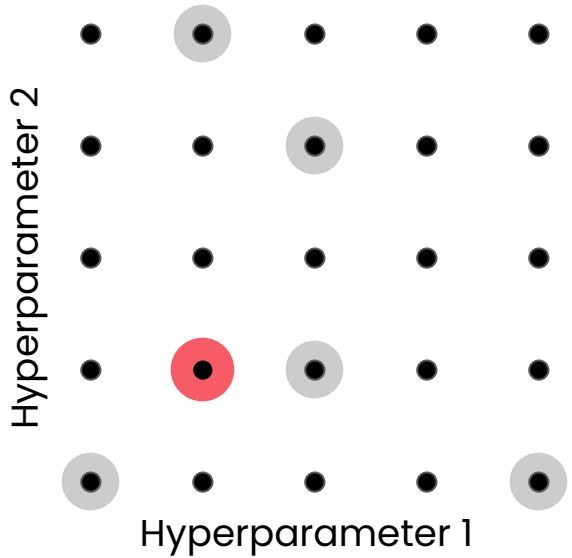
5 values of Hyperparameter 1
5 values of Hyperparameter 2

5*5 = 25 combinations

Random search

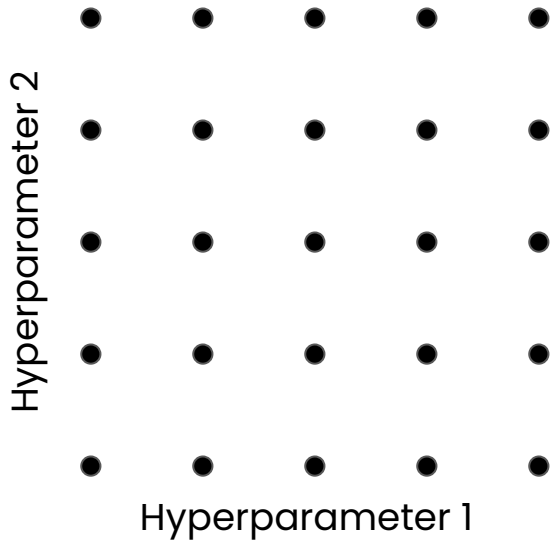


Random search

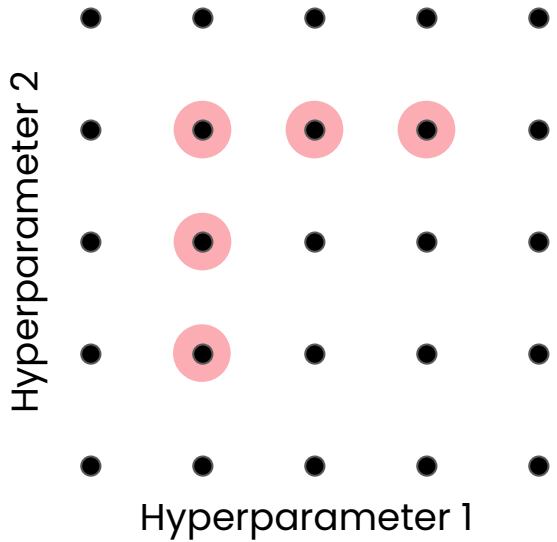


- Doesn't learn from previous trials
- Spends time on unpromising areas

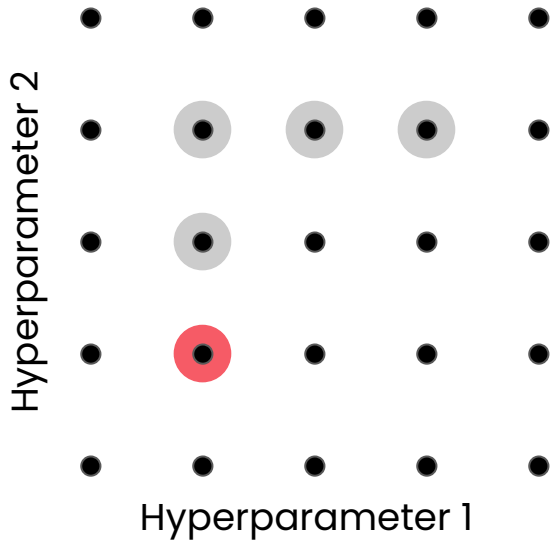
Optuna: Tree-Structured Parzen Estimator



Optuna: Tree-Structured Parzen Estimator



Optuna: Tree-Structured Parzen Estimator

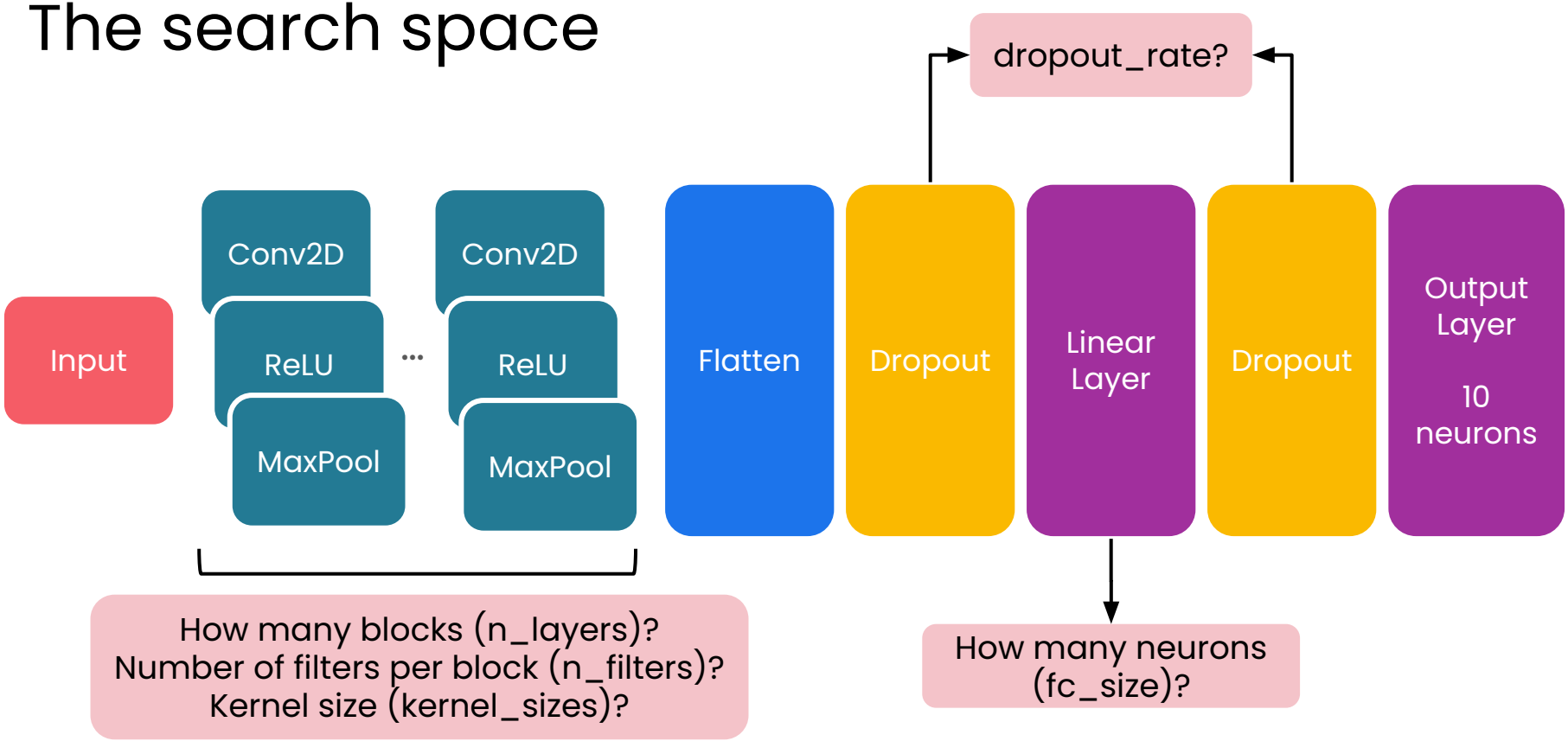


- Uses results from previous trials
- Focuses on regions with high potential
- Powerful for large search spaces

Optuna for hyperparameter tuning

1. Define the search space
2. Run a study
3. Identify the best configuration

The search space



Defining the search space

```
def objective(trial, device):
    # === Choose hyperparameters ===

    # Feature extractor parameters
    n_layers = trial.suggest_int("n_layers", 1, 3)
    n_filters = [trial.suggest_int(f"n_filters_{i}", 16, 128) for i in range(n_layers)]
    kernel_sizes = [trial.suggest_categorical(f"kernel_size_{i}", [3, 5]) for i in
range(n_layers)]

    # Classifier parameters
    dropout_rate = trial.suggest_float("dropout_rate", 0.1, 0.5)
    fc_size = trial.suggest_int("fc_size", 64, 256)
```

Defining the search space

```
def objective(trial, device):
    # === Choose hyperparameters ===

    # Feature extractor parameters
    n_layers = trial.suggest_int("n_layers", 1, 3)
    n_filters = [trial.suggest_int(f"n_filters_{i}", 16, 128) for i in range(n_layers)]
    kernel_sizes = [trial.suggest_categorical(f"kernel_size_{i}", [3, 5]) for i in
range(n_layers)]

    # Classifier parameters
    dropout_rate = trial.suggest_float("dropout_rate", 0.1, 0.5)
    fc_size = trial.suggest_int("fc_size", 64, 256)
```

Defining the search space

```
def objective(trial, device):
    # === Choose hyperparameters ===

    # Feature extractor parameters
    n_layers = trial.suggest_int("n_layers", 1, 3)
    n_filters = [trial.suggest_int(f"n_filters_{i}", 16, 128) for i in range(n_layers)]
    kernel_sizes = [trial.suggest_categorical(f"kernel_size_{i}", [3, 5]) for i in
range(n_layers)]

    # Classifier parameters
    dropout_rate = trial.suggest_float("dropout_rate", 0.1, 0.5)
    fc_size = trial.suggest_int("fc_size", 64, 256)
```

Defining the search space

```
def objective(trial, device):
    # === Choose hyperparameters ===

    # Feature extractor parameters
    n_layers = trial.suggest_int("n_layers", 1, 3)
    n_filters = [trial.suggest_int(f"n_filters_{i}", 16, 128) for i in range(n_layers)]
    kernel_sizes = [trial.suggest_categorical(f"kernel_size_{i}", [3, 5]) for i in
range(n_layers)]

    # Classifier parameters
    dropout_rate = trial.suggest_float("dropout_rate", 0.1, 0.5)
    fc_size = trial.suggest_int("fc_size", 64, 256)
```

Defining the search space

```
def objective(trial, device):
    # === Choose hyperparameters ===

    # Feature extractor parameters
    n_layers = trial.suggest_int("n_layers", 1, 3)
    n_filters = [trial.suggest_int(f"n_filters_{i}", 16, 128) for i in range(n_layers)]
    kernel_sizes = [trial.suggest_categorical(f"kernel_size_{i}", [3, 5]) for i in
range(n_layers)]

    # Classifier parameters
    dropout_rate = trial.suggest_float("dropout_rate", 0.1, 0.5)
    fc_size = trial.suggest_int("fc_size", 64, 256)
```

Defining the search space

```
def objective(trial, device):
    # === Choose hyperparameters ===

    # Feature extractor parameters
    n_layers = trial.suggest_int("n_layers", 1, 3)
    n_filters = [trial.suggest_int(f"n_filters_{i}", 16, 128) for i in range(n_layers)]
    kernel_sizes = [trial.suggest_categorical(f"kernel_size_{i}", [3, 5]) for i in
range(n_layers)]

    # Classifier parameters
    dropout_rate = trial.suggest_float("dropout_rate", 0.1, 0.5)
    fc_size = trial.suggest_int("fc_size", 64, 256)
```

Defining the search space

```
def objective(trial, device):
    # === Choose hyperparameters ===

    # Feature extractor parameters
    n_layers = trial.suggest_int("n_layers", 1, 3)
    n_filters = [trial.suggest_int(f"n_filters_{i}", 16, 128) for i in range(n_layers)]
    kernel_sizes = [trial.suggest_categorical(f"kernel_size_{i}", [3, 5]) for i in
range(n_layers)]

    # Classifier parameters
    dropout_rate = trial.suggest_float("dropout_rate", 0.1, 0.5)
    fc_size = trial.suggest_int("fc_size", 64, 256)
```

Fixed parameters

```
learning_rate = 0.001
batch_size = 128
n_epochs = 10
```

Fixed parameters

```
learning_rate = 0.001
batch_size = 128
n_epochs = 10
```

Optimizable learning rate:

```
learning_rate = trial.suggest_float("learning_rate", 1e-4, 1e-2, log=True)
```

Objective function

```
def objective(trial, device):

...

# === Train and evaluate the model ===

   # Training model
   n_epochs = 10 # This hyperparameter is fixed
   helper_utils.train_model(model=model, optimizer=optimizer, train_dataloader=train_loader,
n_epochs=n_epochs,  loss_fcn=loss_fcn, device=device)

   accuracy = helper_utils.evaluate_accuracy(model, val_loader, device)

   return accuracy
```

Objective function

```
def objective(trial, device):

...

# === Train and evaluate the model ===

    # Training model
    n_epochs = 10 # This hyperparameter is fixed
    helper_utils.train_model(model=model, optimizer=optimizer, train_dataloader=train_loader,
n_epochs=n_epochs,  loss_fcn=loss_fcn, device=device)

    accuracy = helper_utils.evaluate_accuracy(model, val_loader, device)

    return accuracy
```

Optuna study

```
# Create a study object and optimize the objective function
study = optuna.create_study(direction='maximize') # The goal is to maximize
accuracy

# Start the optimization process (it takes about 8 minutes for 20 trials)
n_trials = 20
study.optimize(lambda trial: objective(trial, device), n_trials=n_trials) # use
more trials in practice
```

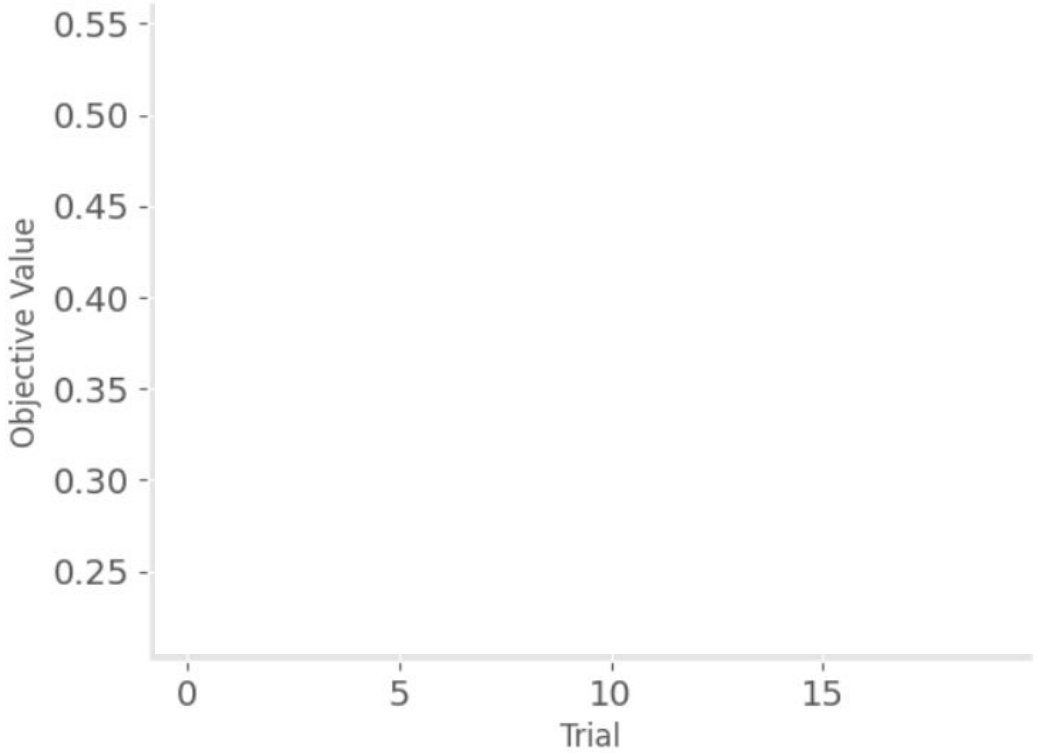
Optuna study

```
# Create a study object and optimize the objective function
study = optuna.create_study(direction='maximize') # The goal is to maximize
accuracy

# Start the optimization process (it takes about 8 minutes for 20 trials)
n_trials = 20
study.optimize(lambda trial: objective(trial, device), n_trials=n_trials) # use
more trials in practice
```

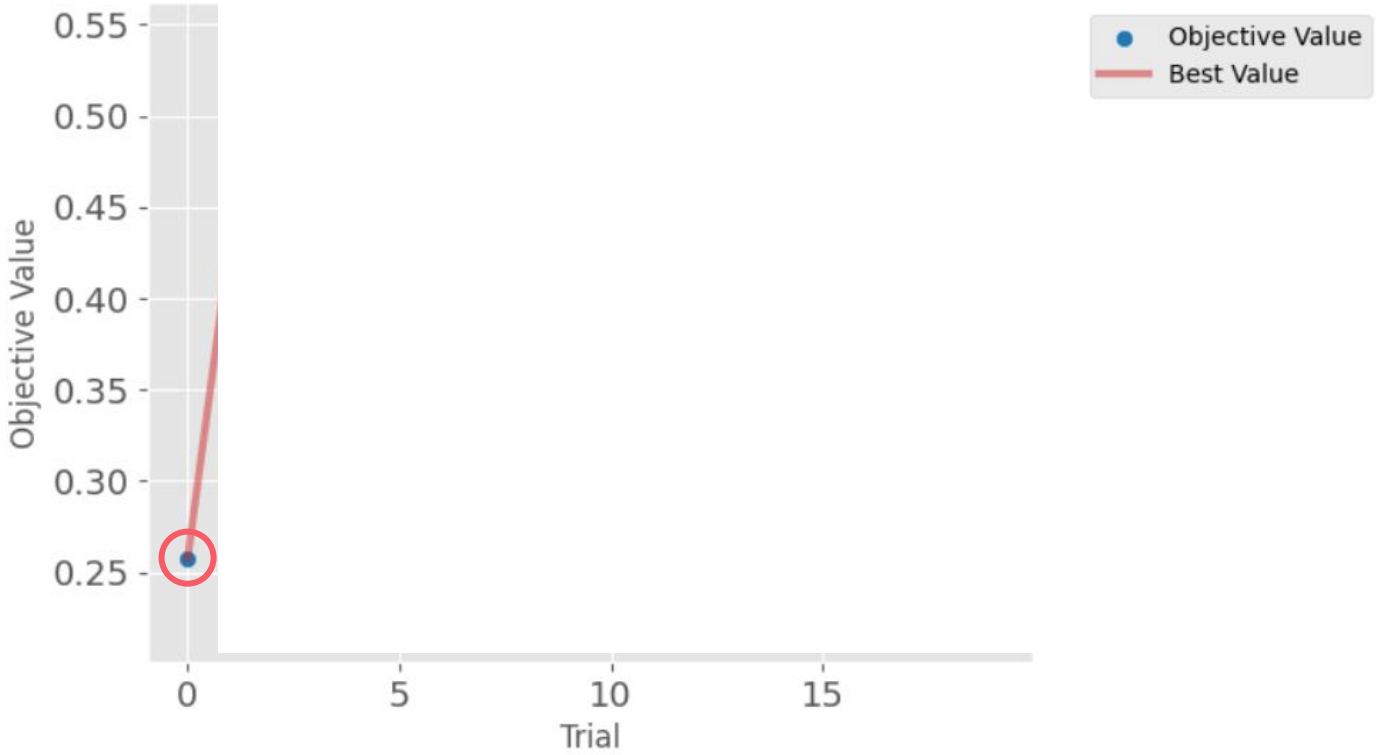
Optuna study

```
# Create a study object and optimize the objective function
study = optuna.create_study(direction='maximize') # The goal is to maximize
accuracy

# Start the optimization process (it takes about 10 minutes for 15 trials)
n_trials = 20
study.optimize(lambda trial: objective(trial, device), n_trials=n_trials) # use
more trials in practice
```

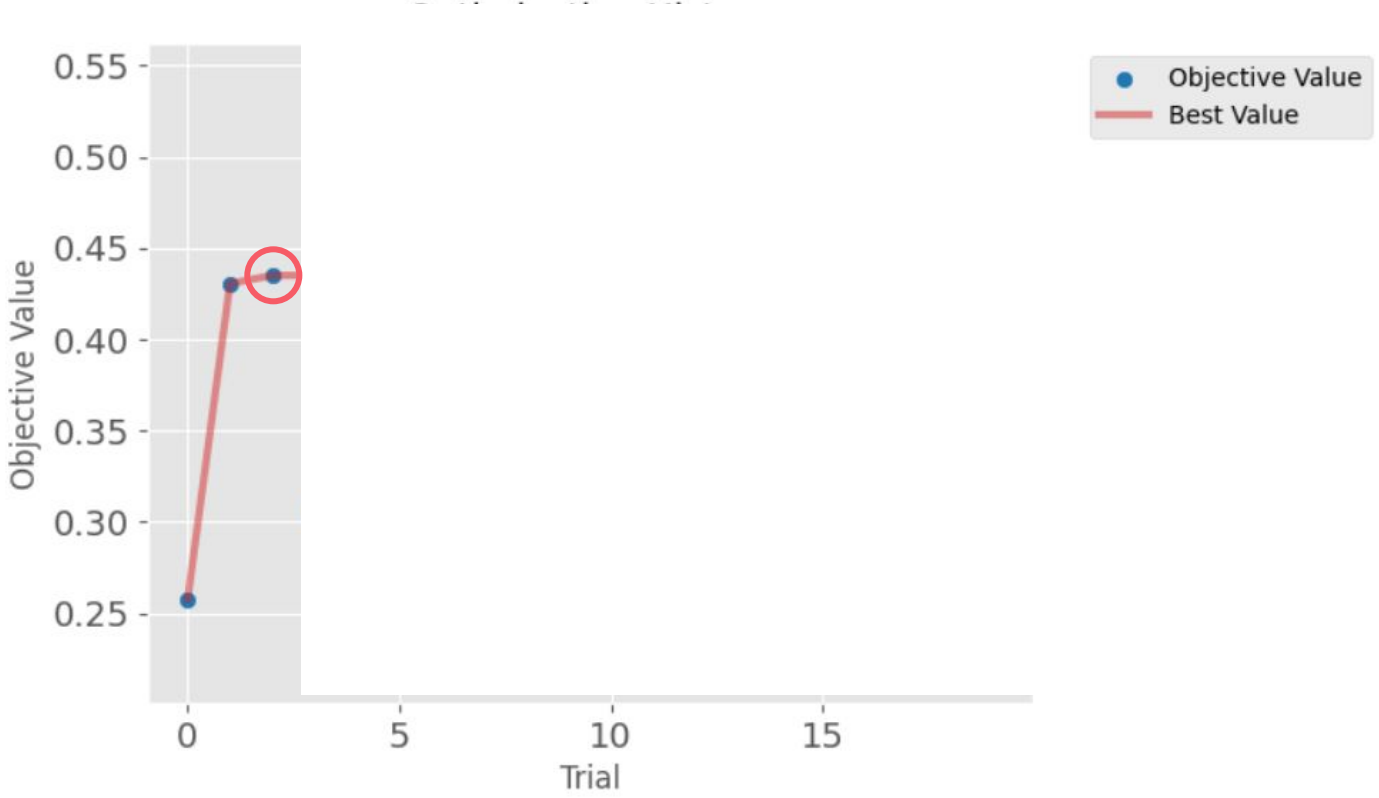
Optimization history



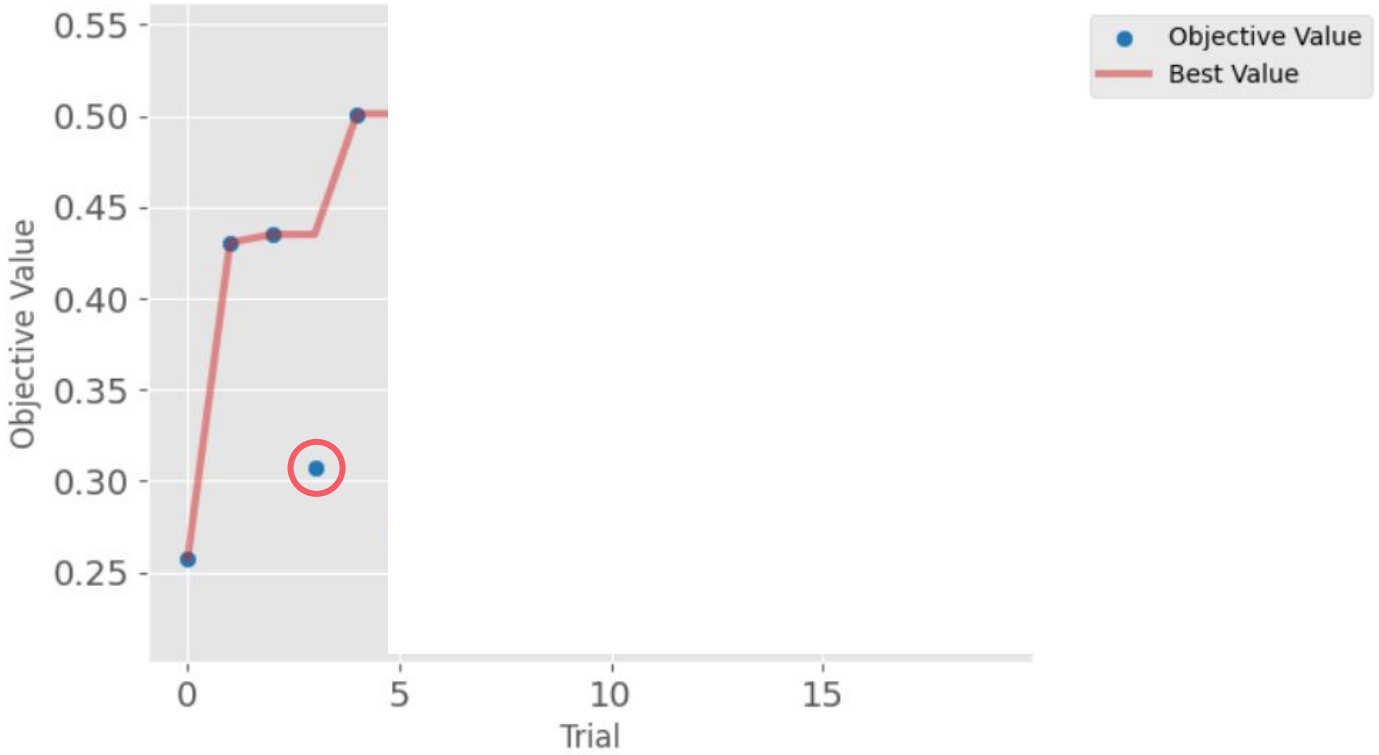
Optimization history



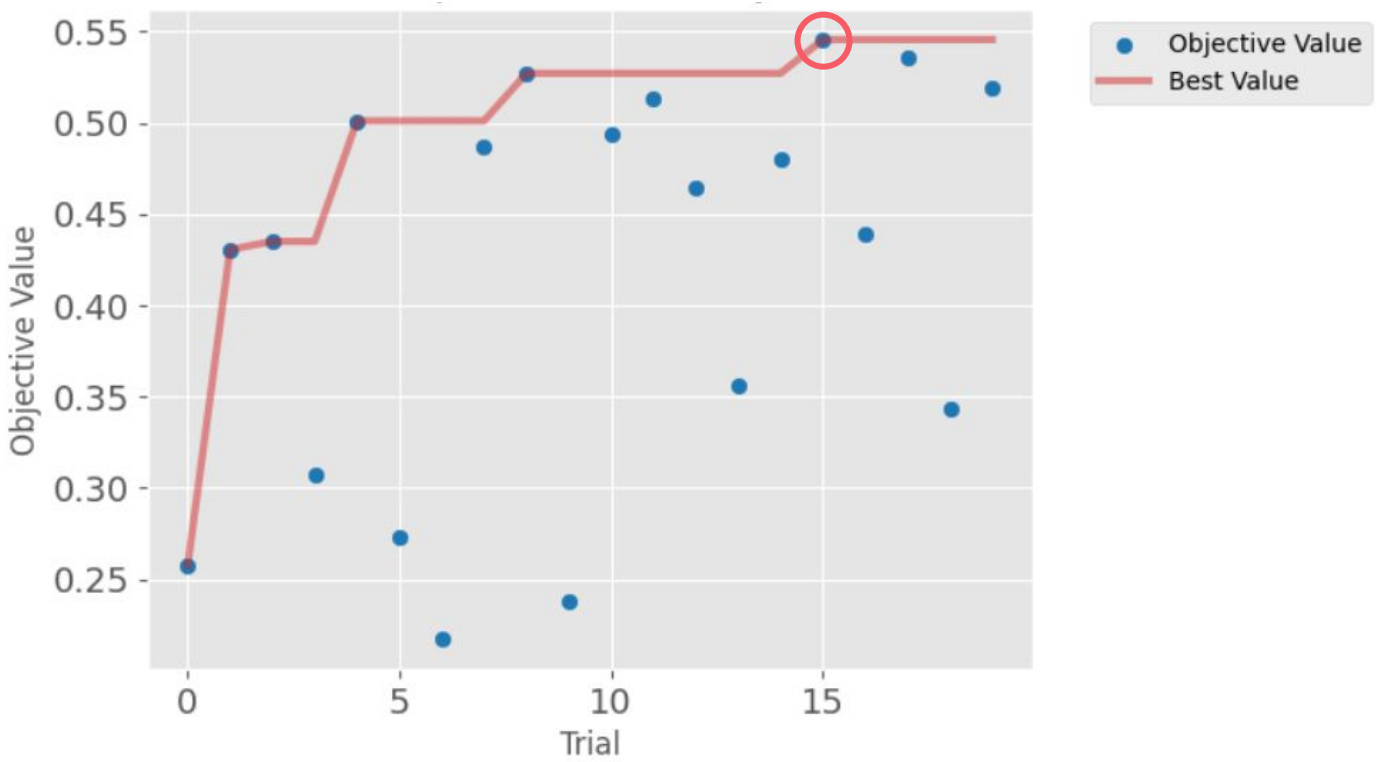
Optimization history



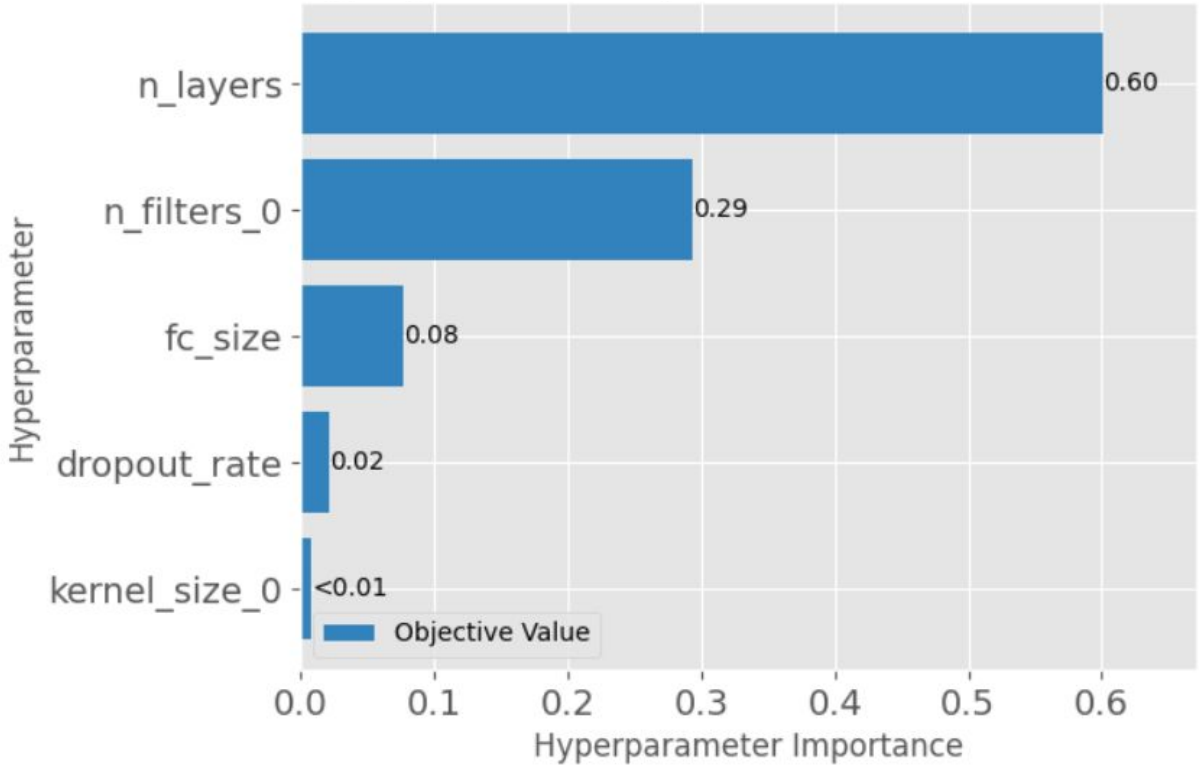
Optimization history



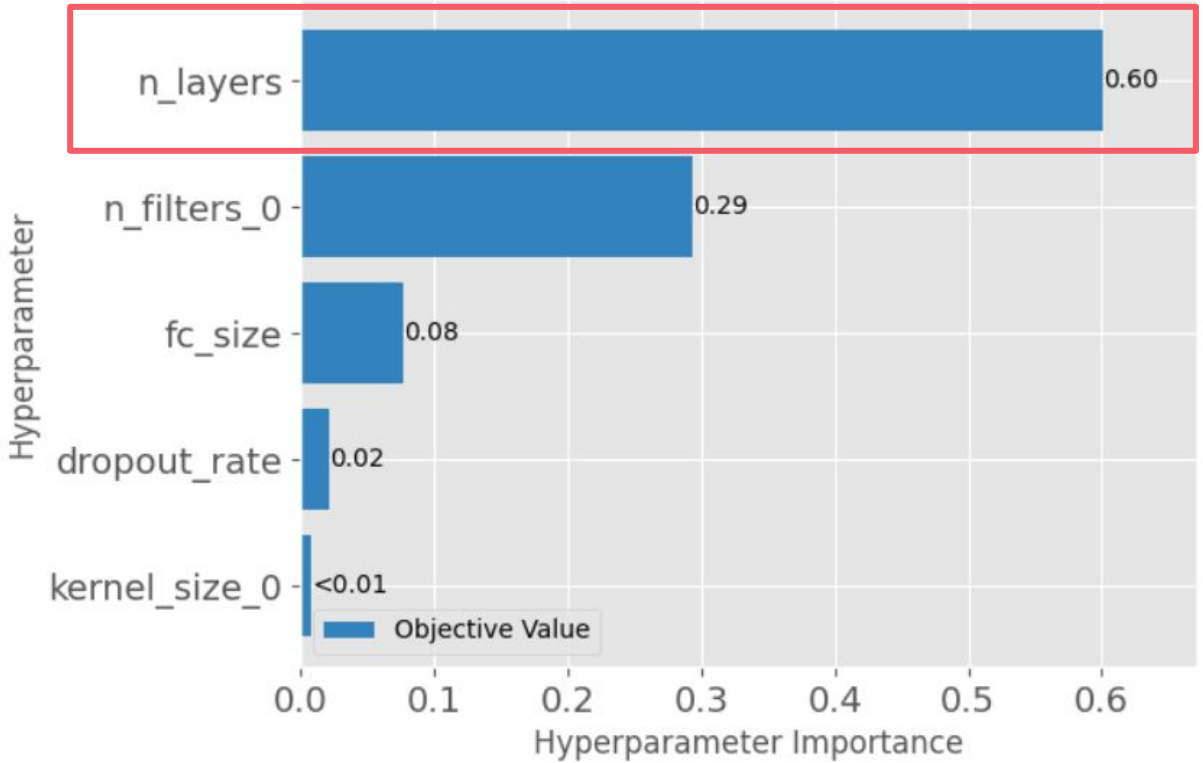
Optimization history



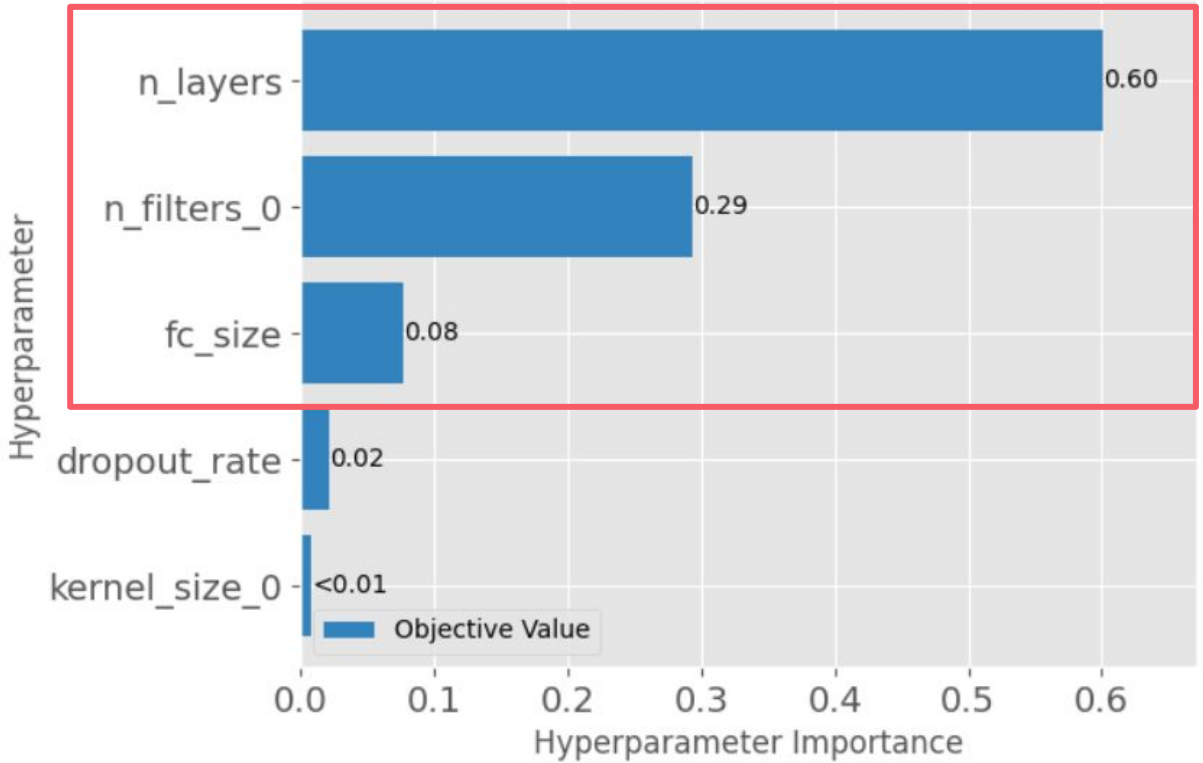
Hyperparameter importance



Hyperparameter importance



Hyperparameter importance



Locking hyperparameters

Study 1

n_layers = ?

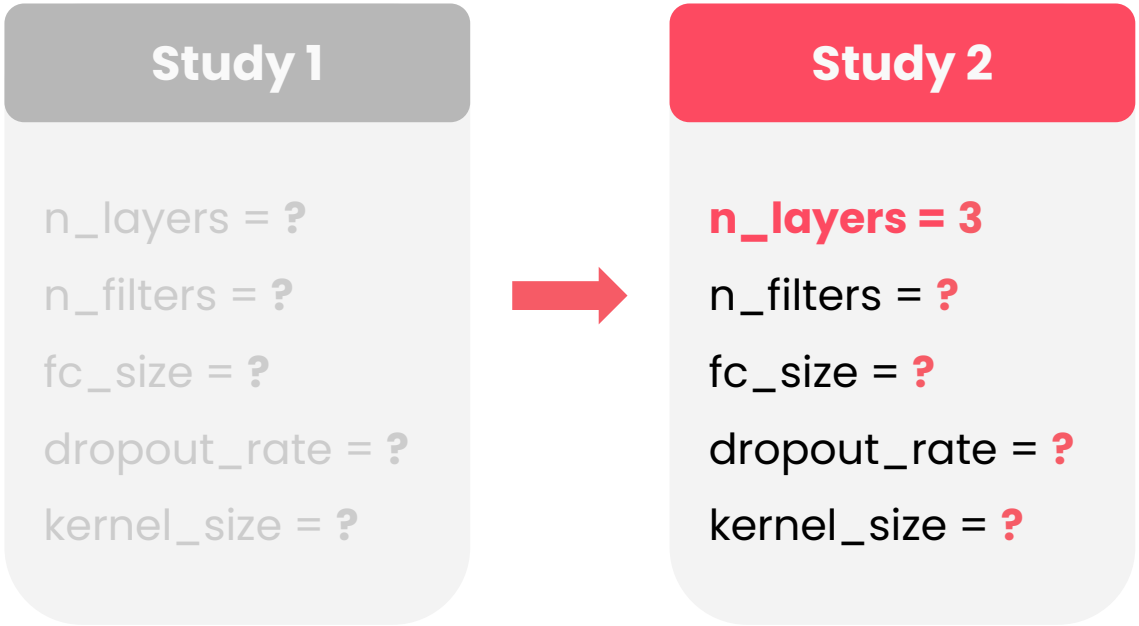
n_filters = ?

fc_size = ?

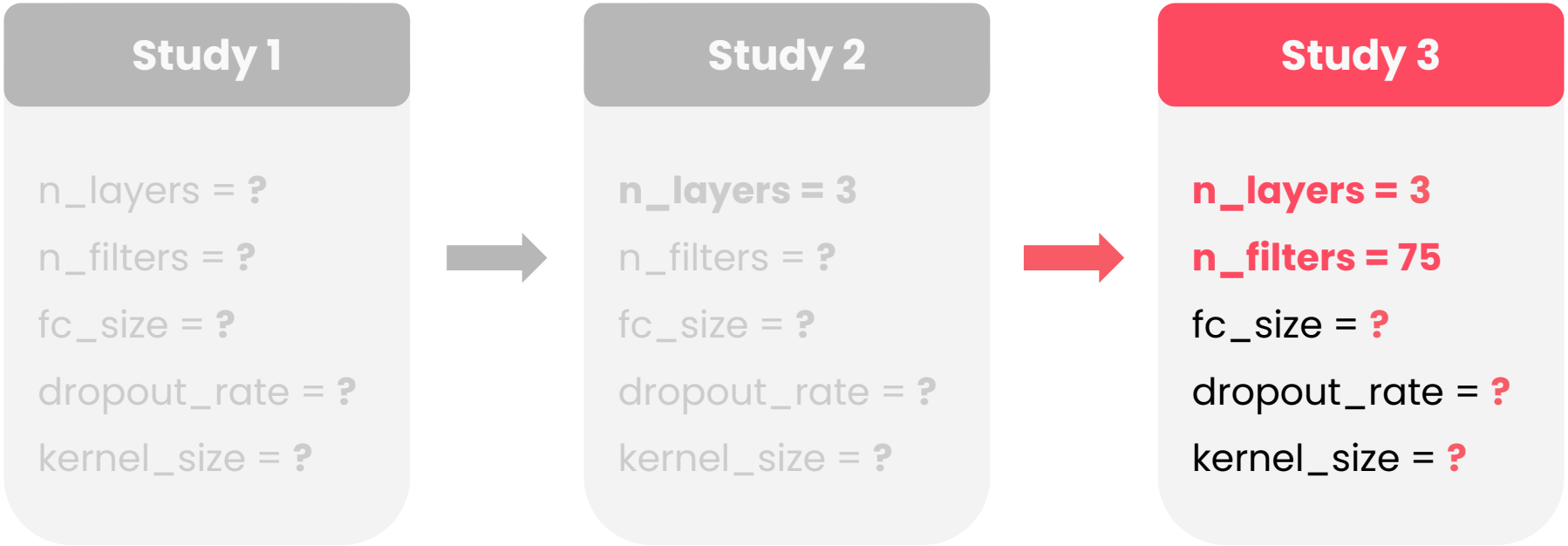
dropout_rate = ?

kernel_size = ?

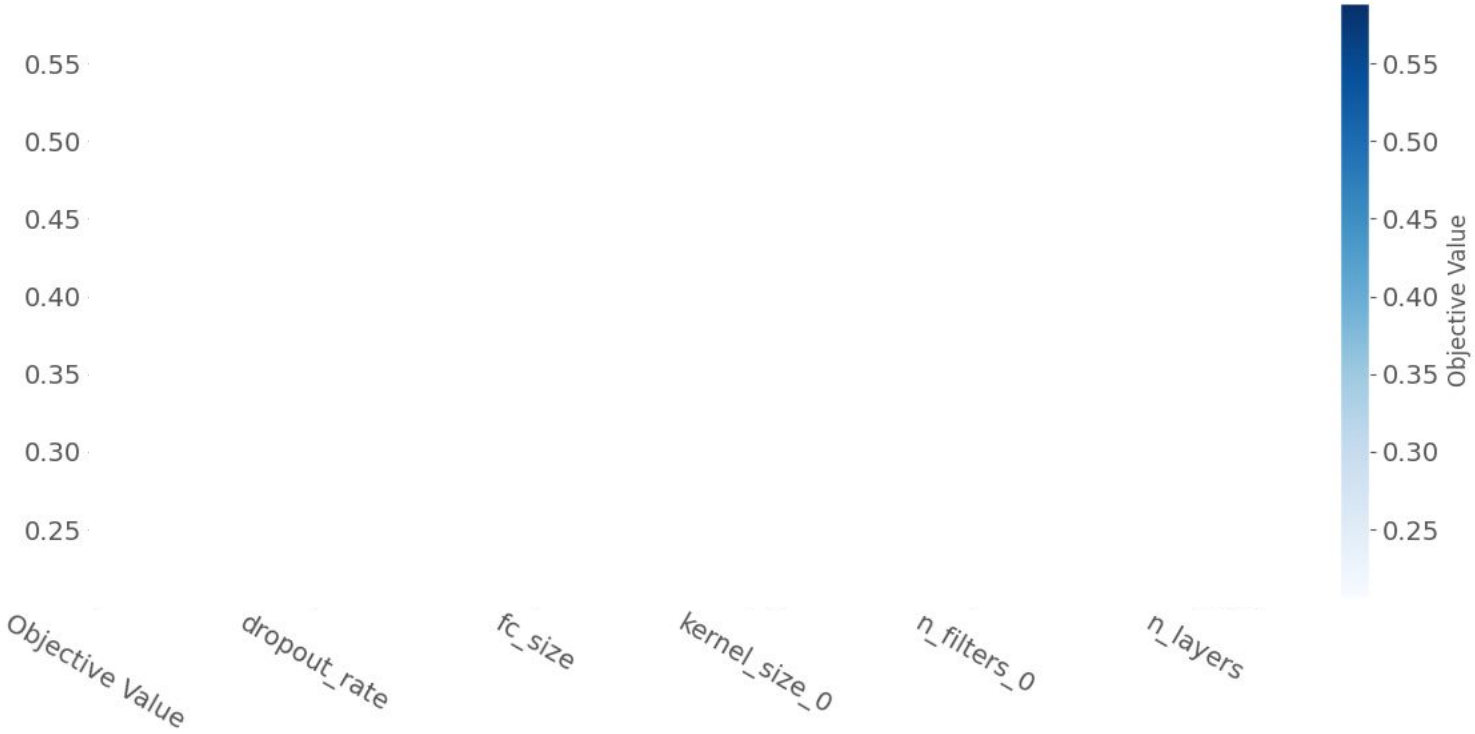
Locking hyperparameters



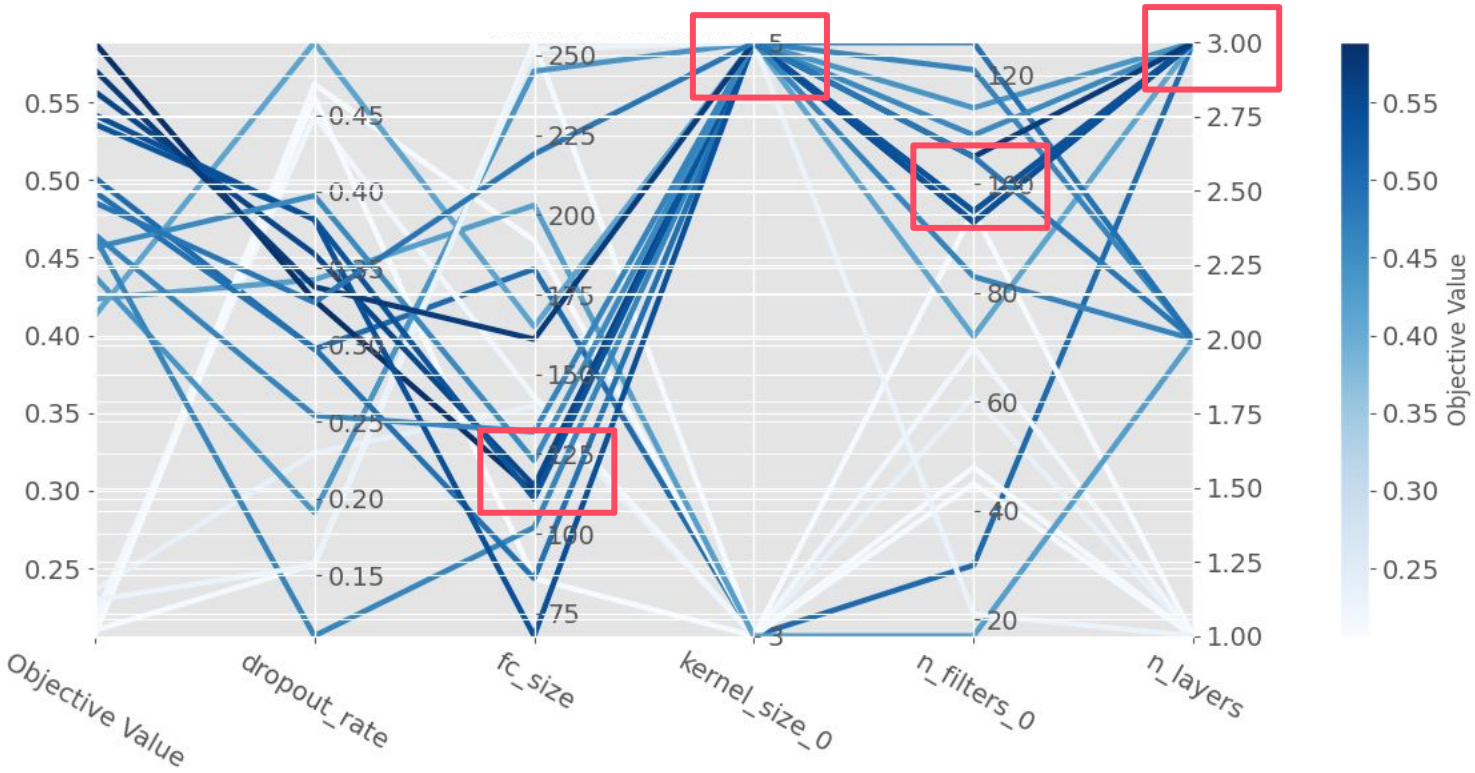
Locking hyperparameters



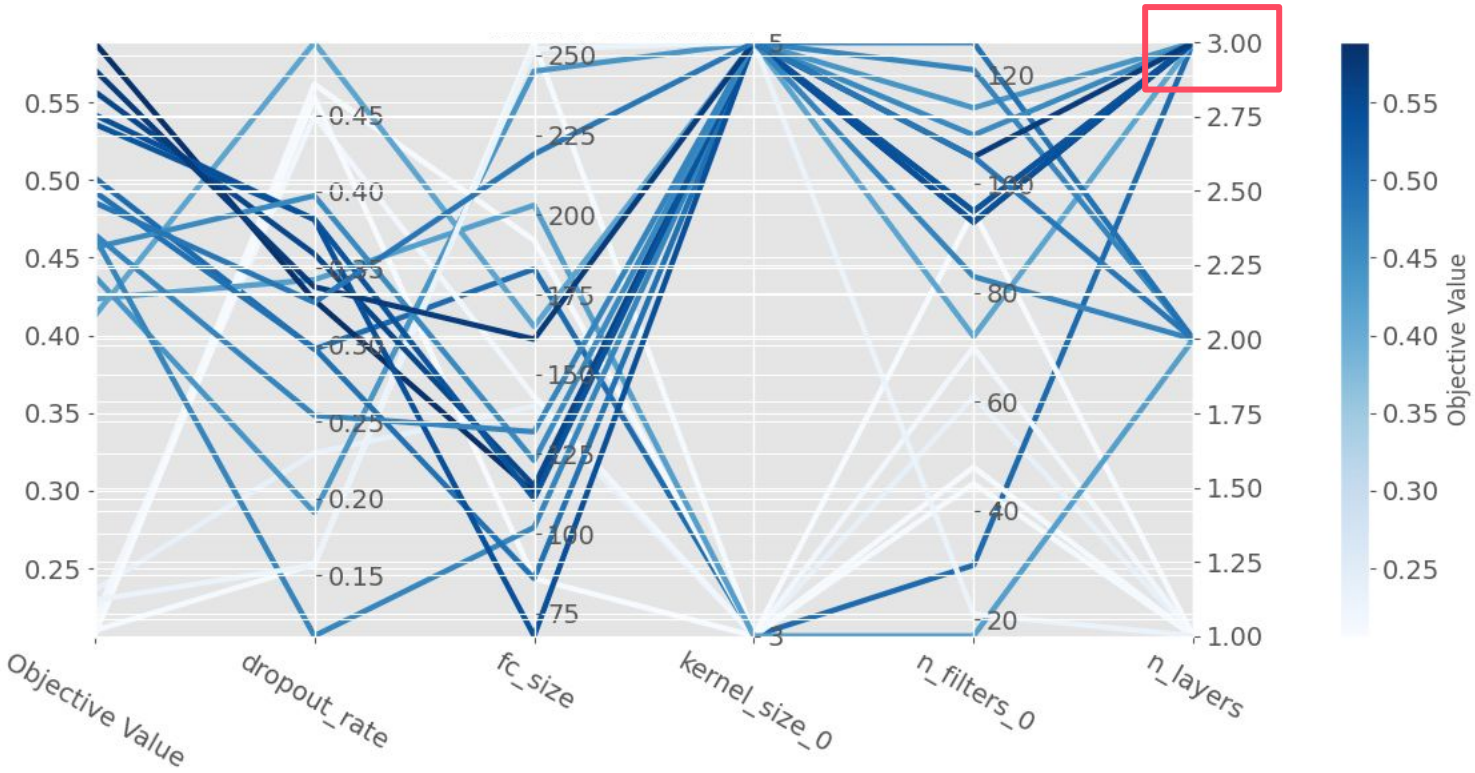
Parallel coordinate plot



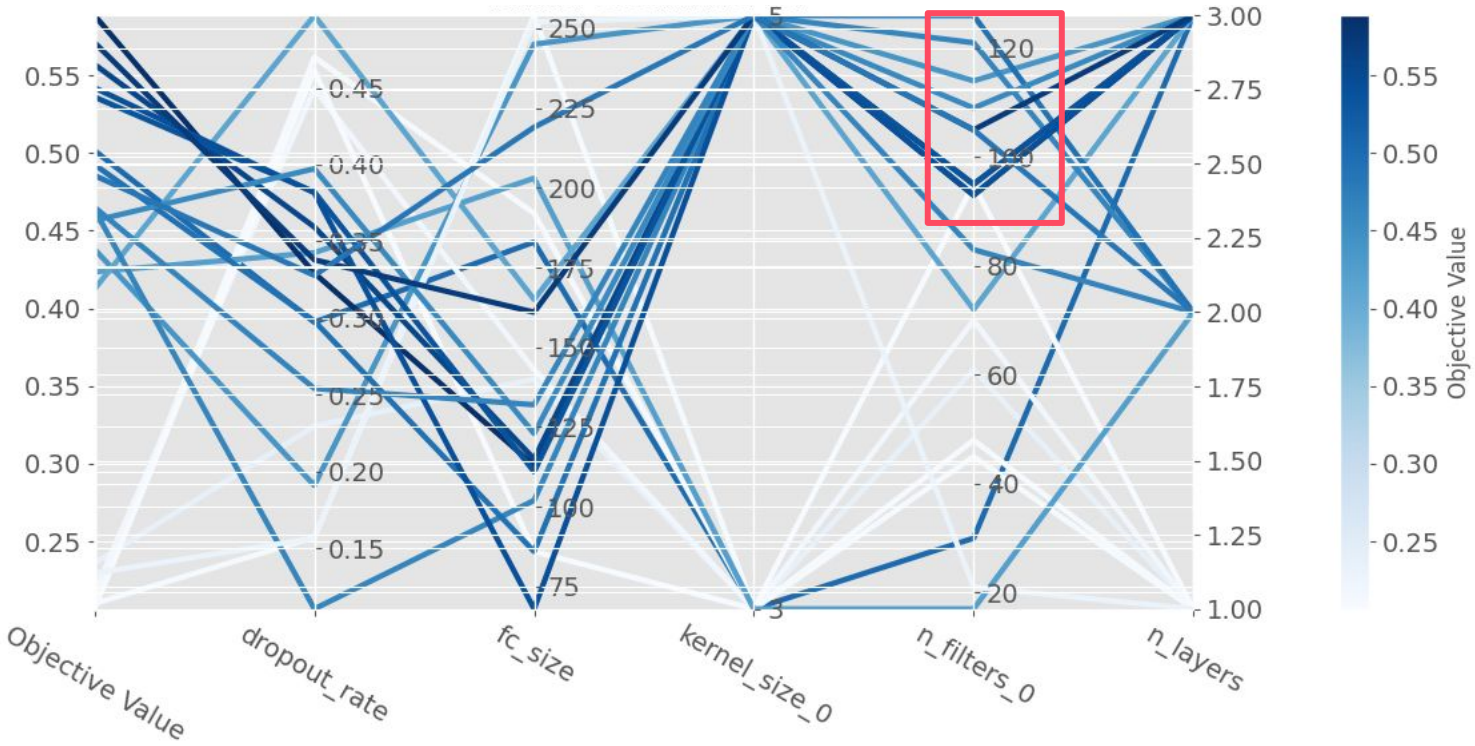
Parallel coordinate plot



Parallel coordinate plot



Parallel coordinate plot



```
# Extract the dataframe with the results
df = study.trials_dataframe()
df

# Extract and print the best trial
trial = study.best_trial
print("Best trial:")
print(f"  Value (Accuracy): {trial.value:.4f}")
print("  Hyperparameters:")
for key, value in trial.params.items():
    print(f"    {key}: {value}")

# Print the best hyperparameters
print("Best hyperparameters:")
print(study.best_params)
```

Best parameters

```
Best trial:
  Value (Accuracy): 0.5560
  Hyperparameters:
{'dropout_rate': 0.441682492142502,
 'fc_size': 216,
 'kernel_size_0': 3,
 'kernel_size_1': 3,
 'kernel_size_2': 5,
 'n_filters_0': 75,
 'n_filters_1': 124,
 'n_filters_2': 98,
 'n_layers': 3}
```

Best parameters

```
Best trial:
  Value (Accuracy): 0.5560
  Hyperparameters:
{'dropout_rate': 0.441682492142502,
 'fc_size': 216,
 'kernel_size_0': 3,
 'kernel_size_1': 3,
 'kernel_size_2': 5,
 'n_filters_0': 75,
 'n_filters_1': 124,
 'n_filters_2': 98,
 'n_layers': 3}
```

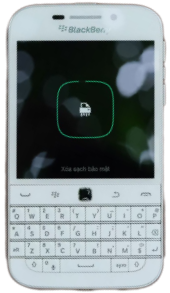
DeepLearning.AI

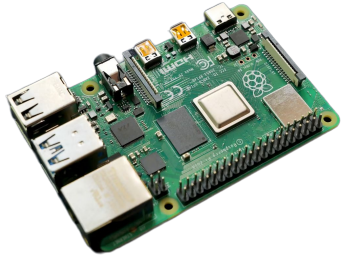
Optimizing Model Efficiency

Hyperparameter optimization

Deployment device

vs.





Additional deployment factors

Memory
footprint

Additional deployment factors

Memory footprint

Inference time

Additional deployment factors

Memory footprint

Inference time

Power consumption

Additional deployment factors

Memory footprint

Inference time

Power consumption

Latency and throughput requirements

Additional deployment factors

Memory footprint

Inference time

Power consumption

Latency and throughput requirements

Compare 2 models across 3 metrics

Models

- OptimizedCNN
- ResNet34

Metrics

- Accuracy
- Model size
- Inference time

```python
def get_model_size(model):

    # Model parameters
    param_size = 0
    for param in model.parameters():
        param_size += param.nelement() * param.element_size()

    # Model buffers, non-trainable parameters
    buffer_size = 0
    for buffer in model.buffers():
        buffer_size += buffer.nelement() * buffer.element_size()

    size_in_mb = (param_size + buffer_size) / 1024**2
    return size_in_mb
```

```
def get_model_size(model):

    # Model parameters
    param_size = 0
    for param in model.parameters():
        param_size += param.nelement() * param.element_size()

    # Model buffers, non-trainable parameters
    buffer_size = 0
    for buffer in model.buffers():
        buffer_size += buffer.nelement() * buffer.element_size()

    size_in_mb = (param_size + buffer_size) / 1024**2
    return size_in_mb
```

```
def get_model_size(model):

    # Model parameters
    param_size = 0
    for param in model.parameters():
        param_size += param.nelement() * param.element_size()

    # Model buffers, non-trainable parameters
    buffer_size = 0
    for buffer in model.buffers():
        buffer_size += buffer.nelement() * buffer.element_size()

    size_in_mb = (param_size + buffer_size) / 1024**2
    return size_in_mb
```

```
def get_model_size(model):

    # Model parameters
    param_size = 0
    for param in model.parameters():
        param_size += param.nelement() * param.element_size()

    # Model buffers, non-trainable parameters
    buffer_size = 0
    for buffer in model.buffers():
        buffer_size += buffer.nelement() * buffer.element_size()

    size_in_mb = (param_size + buffer_size) / 1024**2
    return size_in_mb
```

```
def get_model_size(model):

    # Model parameters
    param_size = 0
    for param in model.parameters():
        param_size += param.nelement() * param.element_size()

    # Model buffers, non-trainable parameters
    buffer_size = 0
    for buffer in model.buffers():
        buffer_size += buffer.nelement() * buffer.element_size()

    size_in_mb = (param_size + buffer_size) / 1024**2
    return size_in_mb
```

Measuring inference time

How long it takes to make a prediction, which is relevant for:

Real-time applications

Batch processing

```
def measure_inference_time(model, input_data, num_iterations=100):
    model.eval()  # Set to evaluation mode

    # Move input to the same device as the model
    device = next(model.parameters()).device
    input_data = input_data.to(device)

    # Warmup
    with torch.no_grad():
        for _ in range(10):
            _= model(input_data)

    ...
```

```
def measure_inference_time(model, input_data, num_iterations=100):
    model.eval()  # Set to evaluation mode

    # Move input to the same device as the model
    device = next(model.parameters()).device
    input_data = input_data.to(device)

    # Warmup
    with torch.no_grad():
        for _ in range(10):
            _= model(input_data)

    ...
```

```
def measure_inference_time(model, input_data, num_iterations=100):
    model.eval()  # Set to evaluation mode

    # Move input to the same device as the model
    device = next(model.parameters()).device
    input_data = input_data.to(device)

    # Warmup
    with torch.no_grad():
        for _ in range(10):
            _= model(input_data)

    ...
```

```
def measure_inference_time(model, input_data, num_iterations=100):
    model.eval()  # Set to evaluation mode

    # Move input to the same device as the model
    device = next(model.parameters()).device
    input_data = input_data.to(device)

    # Warmup
    with torch.no_grad():
        for _ in range(10):
            _= model(input_data)

    ...
```

```
def measure_inference_time(model, input_data, num_iterations=100):
    model.eval()  # Set to evaluation mode

    ...

    # Measure
    start_time = time.time()
    with torch.no_grad():
        for _ in range(num_iterations):
            _= model(input_data)
    end_time = time.time()

    avg_time = (end_time - start_time) / num_iterations
    return avg_time * 1000  # Convert to ms
```

```
def measure_inference_time(model, input_data, num_iterations=100):
    model.eval()  # Set to evaluation mode

    ...

    # Measure
    start_time = time.time()
    with torch.no_grad():
        for _ in range(num_iterations):
            _= model(input_data)
    end_time = time.time()

    avg_time = (end_time - start_time) / num_iterations
    return avg_time * 1000  # Convert to ms
```

```python
def evaluate_efficiency(model, test_loader, device):

    # Model size
    model_size = get_model_size(model)

    # Measure inference time
    data_iter = iter(test_loader)
    batch = next(data_iter)
    inputs = batch[0]  # Get just the inputs
    sample_input = inputs[:1].to(device)  # Take first example and maintain batch dimension
    inf_time = measure_inference_time(model, sample_input)

    # Evaluate accuracy
    test_accuracy = helper_utils.evaluate_accuracy(model, test_loader, device)

    return {
        "accuracy": test_accuracy,
        "model_size_mb": model_size,
        "inference_time_ms": inf_time,
    }
```

```
def evaluate_efficiency(model, test_loader, device):

    # Model size
    model_size = get_model_size(model)

    # Measure inference time
    data_iter = iter(test_loader)
    batch = next(data_iter)
    inputs = batch[0]  # Get just the inputs
    sample_input = inputs[:1].to(device)  # Take first example and maintain batch dimension
    inf_time = measure_inference_time(model, sample_input)

    # Evaluate accuracy
    test_accuracy = helper_utils.evaluate_accuracy(model, test_loader, device)

    return {
        "accuracy": test_accuracy,
        "model_size_mb": model_size,
        "inference_time_ms": inf_time,
    }
```

Evaluating model efficiency

```
results = evaluate_efficiency(model, test_loader, device)
print(results)
```

Evaluating model efficiency

```
results = evaluate_efficiency(model, test_loader, device)
print(results)
```

Output:

```
{
  'accuracy': 0.80,
  'model_size_mb': 9.2,
  'inference_time_ms': 0.51
}
```

Comparing models

```
models = {
    "optimized_cnn": model_CNN,
    "resnet34": model_resnet
}

results = {}
for name, model in models.items():
    results[name] = evaluate_efficiency(model, test_loader, device)

# Convert results to DataFrame for better visualization
results_df = pd.DataFrame(results).T
print(results_df)
```

Comparing models

```
models = {
    "optimized_cnn": model_CNN,
    "resnet34": model_resnet
}

results = {}
for name, model in models.items():
    results[name] = evaluate_efficiency(model, test_loader, device)

# Convert results to DataFrame for better visualization
results_df = pd.DataFrame(results).T
print(results_df)
```

Comparing models

```
models = {
    "optimized_cnn": model_CNN,
    "resnet34": model_resnet
}

results = {}
for name, model in models.items():
    results[name] = evaluate_efficiency(model, test_loader, device)

# Convert results to DataFrame for better visualization
results_df = pd.DataFrame(results).T
print(results_df)
```

Comparing models

```
models = {
    "optimized_cnn": model_CNN,
    "resnet34": model_resnet
}

results = {}
for name, model in models.items():
    results[name] = evaluate_efficiency(model, test_loader, device)

# Convert results to DataFrame for better visualization
results_df = pd.DataFrame(results).T
print(results_df)
```

Comparing models

model	accuracy	model_size_mb	inference_time_ms
Optimized_CNN	0.75	9.19	0.34
ResNet34	0.73	81.25	3.63

Plotting model efficiency metrics

Constraint-based selection

For environments with strict technical limits

Constraint-based selection

For environments with strict technical limits

Filter out models that exceed memory/speed limits

Constraint-based selection

For environments with strict technical limits

Filter out models that exceed memory/speed limits

Select the one with the highest accuracy

```python
def select_best_model_constraint_based(results, max_size_mb, max_inference_ms):
    results = results.to_dict(orient="index")

    # Filter models that meet both size and inference time constraints
    viable_models = {
        name: metrics for name, metrics in results.items()
        if metrics["model_size_mb"] <= max_size_mb and
           metrics["inference_time_ms"] <= max_inference_ms
    }

    if not viable_models:
        # If no models satisfy the constraints, inform the user
        print("No models meet all constraints. Consider relaxing constraints.")
        return None

    # Among viable models, select the one with the highest accuracy
    best_model = max(viable_models.items(), key=lambda x: x[1]["accuracy"])
    return best_model[0], viable_models  # Return best model name and all viable models
```

```
def select_best_model_constraint_based(results, max_size_mb, max_inference_ms):
    results = results.to_dict(orient="index")

    # Filter models that meet both size and inference time constraints
    viable_models = {
        name: metrics for name, metrics in results.items()
        if metrics["model_size_mb"] <= max_size_mb and
           metrics["inference_time_ms"] <= max_inference_ms
    }

    if not viable_models:
        # If no models satisfy the constraints, inform the user
        print("No models meet all constraints. Consider relaxing constraints.")
        return None

    # Among viable models, select the one with the highest accuracy
    best_model = max(viable_models.items(), key=lambda x: x[1]["accuracy"])
    return best_model[0], viable_models  # Return best model name and all viable models
```

```
def select_best_model_constraint_based(results, max_size_mb, max_inference_ms):
    results = results.to_dict(orient="index")

    # Filter models that meet both size and inference time constraints
    viable_models = {
        name: metrics for name, metrics in results.items()
        if metrics["model_size_mb"] <= max_size_mb and
           metrics["inference_time_ms"] <= max_inference_ms
    }

    if not viable_models:
        # If no models satisfy the constraints, inform the user
        print("No models meet all constraints. Consider relaxing constraints.")
        return None

    # Among viable models, select the one with the highest accuracy
    best_model = max(viable_models.items(), key=lambda x: x[1]["accuracy"])
    return best_model[0], viable_models  # Return best model name and all viable models
```

```
def select_best_model_constraint_based(results, max_size_mb, max_inference_ms):
    results = results.to_dict(orient="index")

    # Filter models that meet both size and inference time constraints
    viable_models = {
        name: metrics for name, metrics in results.items()
        if metrics["model_size_mb"] <= max_size_mb and
           metrics["inference_time_ms"] <= max_inference_ms
    }

    if not viable_models:
        # If no models satisfy the constraints, inform the user
        print("No models meet all constraints. Consider relaxing constraints.")
        return None

    # Among viable models, select the one with the highest accuracy
    best_model = max(viable_models.items(), key=lambda x: x[1]["accuracy"])
    return best_model[0], viable_models  # Return best model name and all viable models
```

Weighted scoring

Use it when your requirements are flexible

Weighted scoring

Use it when your requirements are flexible

Assign relative importance to each metric

Weighted scoring

Use it when your requirements are flexible

Assign relative importance to each metric

Calculate a weighted score that combines all metrics

Weighted scoring

```
def select_best_model_weighted(results, weights=None):
    results = results.to_dict(orient="index")

    if weights is None:
        # Default weights: prioritize accuracy more than efficiency
        weights = {"accuracy": 0.5, "model_size_mb": 0.2, "inference_time_ms": 0.3}

    metrics = list(weights.keys())  # List of metrics to consider
    normalized = {name: {} for name in results}  # Initialize normalized results
```

Weighted scoring

```
def select_best_model_weighted(results, weights=None):
    results = results.to_dict(orient="index")

    if weights is None:
        # Default weights: prioritize accuracy more than efficiency
        weights = {"accuracy": 0.5, "model_size_mb": 0.2, "inference_time_ms": 0.3}

    metrics = list(weights.keys())  # List of metrics to consider
    normalized = {name: {} for name in results}  # Initialize normalized results
```

$$Weighted\ score = (0.5 * accuracy) + (0.2 * model\_size\_mb) + (0.3 * inference\_time\_ms)$$

Normalize your metrics

Accuracy

Percentage

Model size

Megabytes

Inference time

Milliseconds

Normalize your metrics

Accuracy	Model size	Inference time
Percentage	Megabytes	Milliseconds
Maximize	Minimize	Minimize

```
# Normalize each metric across all models
    for metric in metrics:
        values = [res[metric] for res in results.values()]  # Get all values for this metric
        min_val, max_val = min(values), max(values)
        range_val = max_val - min_val if max_val != min_val else 1.0  # Avoid division by zero

        for name, res in results.items():
            value = res[metric]
            if metric == "accuracy":
                # For accuracy: higher is better -> normalize directly
                norm_value = (value - min_val) / range_val
            else:
                # For model size and inference time: lower is better -> inverse normalization
                norm_value = 1 - (value - min_val) / range_val
            normalized[name][metric] = norm_value
```

```
# Normalize each metric across all models
    for metric in metrics:
        values = [res[metric] for res in results.values()]  # Get all values for this metric
        min_val, max_val = min(values), max(values)
        range_val = max_val - min_val if max_val != min_val else 1.0  # Avoid division by zero

        for name, res in results.items():
            value = res[metric]
            if metric == "accuracy":
                # For accuracy: higher is better -> normalize directly
                norm_value = (value - min_val) / range_val
            else:
                # For model size and inference time: lower is better -> inverse normalization
                norm_value = 1 - (value - min_val) / range_val
            normalized[name][metric] = norm_value
```

```
# Normalize each metric across all models
    for metric in metrics:
        values = [res[metric] for res in results.values()]  # Get all values for this metric
        min_val, max_val = min(values), max(values)
        range_val = max_val - min_val if max_val != min_val else 1.0  # Avoid division by zero

        for name, res in results.items():
            value = res[metric]
            if metric == "accuracy":
                # For accuracy: higher is better -> normalize directly
                norm_value = (value - min_val) / range_val
            else:
                # For model size and inference time: lower is better -> inverse normalization
                norm_value = 1 - (value - min_val) / range_val
            normalized[name][metric] = norm_value
```

$$x' = \frac{x - min(x)}{max(x) - min(x)}$$

```
# Normalize each metric across all models
    for metric in metrics:
        values = [res[metric] for res in results.values()]  # Get all values for this metric
        min_val, max_val = min(values), max(values)
        range_val = max_val - min_val if max_val != min_val else 1.0  # Avoid division by zero

        for name, res in results.items():
            value = res[metric]
            if metric == "accuracy":
                # For accuracy: higher is better -> normalize directly
                norm_value = (value - min_val) / range_val
            else:
                # For model size and inference time: lower is better -> inverse normalization
                norm_value = 1 - (value - min_val) / range_val
            normalized[name][metric] = norm_value
```

$$x' = 1 - \frac{x - min(x)}{max(x) - min(x)}$$

```
# Compute weighted score for each model
    scores = {
        name: sum(weights[metric] * normalized[name][metric] for metric in metrics)
        for name in results
    }

    # Select model with highest weighted score
    best_model = max(scores.items(), key=lambda x: x[1])
    return best_model[0], scores  # Return best model name and all scores
```

```
# Compute weighted score for each model
    scores = {
        name: sum(weights[metric] * normalized[name][metric] for metric in metrics)
        for name in results
    }

    # Select model with highest weighted score
    best_model = max(scores.items(), key=lambda x: x[1])
    return best_model[0], scores  # Return best model name and all scores
```

Computing weighted scores

model	accuracy	model_size_mb	inference_time_ms
ResNet18	0.92	42	2.0
ResNet34	0.93	81	3.8
ResNet50	0.94	98	5.0
ResNet101	0.95	168	8.0
ResNet152	0.96	230	12.0

Computing weighted scores

Normalized

model	accuracy	model_size_mb	inference_time_ms
ResNet18	0.0	1.0	1.0
ResNet34	0.25	0.79	0.82
ResNet50	0.5	0.70	0.7
ResNet101	0.75	0.33	0.4
ResNet152	1.0	0.0	0.0

Computing weighted scores

Normalized

model	accuracy	model_size_mb	inference_time_ms
ResNet18	0.0	1.0	1.0
ResNet34	0.25	0.79	0.82
ResNet50	0.5	0.70	0.7
ResNet101	0.75	0.33	0.4
ResNet152	1.0	0.0	0.0

$$Weighted\ score = (0.5 * accuracy) + (0.2 * model\_size\_mb) + (0.3 * inference\_time\_ms)$$

Computing weighted scores

Normalized

model	accuracy	model_size_mb	inference_time_ms	weighted_score
ResNet18	0.0	1.0	1.0	0.50
ResNet34	0.25	0.79	0.82	
ResNet50	0.5	0.70	0.7	
ResNet101	0.75	0.33	0.4	
ResNet152	1.0	0.0	0.0	

$$Weighted\ score\ for\ ResNet18 = (0.5 * 0.0) + (0.2 * 1.0) + (0.3 * 1.0)$$

Computing weighted scores

Normalized

model	accuracy	model_size_mb	inference_time_ms	weighted_score
ResNet18	0.0	1.0	1.0	0.50
ResNet34	0.25	0.79	0.82	0.53
ResNet50	0.5	0.70	0.7	0.60
ResNet101	0.75	0.33	0.4	0.56
ResNet152	1.0	0.0	0.0	0.50

Computing weighted scores

Normalized

model	accuracy	model_size_mb	inference_time_ms	weighted_score
ResNet18	0.0	1.0	1.0	0.50
ResNet34	0.25	0.79	0.82	0.53
ResNet50	0.5	0.70	0.7	0.60
ResNet101	0.75	0.33	0.4	0.56
ResNet152	1.0	0.0	0.0	0.50

What you've learned in this module:

What you've learned in this module:

What optimization involves

What you've learned in this module:

What optimization involves

Using schedulers to optimize learning rate over time

What you've learned in this module:

What optimization involves

Using schedulers to optimize learning rate over time

Which hyperparameters can be tuned

What you've learned in this module:

What optimization involves

Using schedulers to optimize learning rate over time

Which hyperparameters can be tuned

How to design a flexible CNN architecture

What you've learned in this module:

What optimization involves

Using schedulers to optimize learning rate over time

Which hyperparameters can be tuned

How to design a flexible CNN architecture

Automating hyperparameter optimization with Optuna

What you've learned in this module:

What optimization involves

Using schedulers to optimize learning rate over time

Which hyperparameters can be tuned

How to design a flexible CNN architecture

Automating hyperparameter optimization with Optuna

Optimizing model efficiency for real-world constraints