

# Layer initialization and transfer learning

INTRODUCTION TO DEEP LEARNING WITH PYTORCH



**Jasmin Ludolf**

Senior Data Science Content Developer,  
DataCamp

# Layer initialization

```
import torch.nn as nn

layer = nn.Linear(64, 128)
print(layer.weight.min(), layer.weight.max())
```

```
(tensor(-0.1250, grad_fn=<MinBackward1>), tensor(0.1250, grad_fn=<MaxBackward1>))
```

- A layer weights are initialized to small values
- Keeping both the input data and layer weights small ensures stable outputs

# Layer initialization

```
import torch.nn as nn

layer = nn.Linear(64, 128)
nn.init.uniform_(layer.weight)

print(layer.weight.min(), layer.weight.max())
```

```
(tensor(0.0002, grad_fn=<MinBackward1>), tensor(1.0000, grad_fn=<MaxBackward1>))
```

# Transfer learning

- Reusing a model trained on a first task for a second similar task
  - Trained a model on US data scientist salaries
  - Use weights to train on European salaries

```
import torch

layer = nn.Linear(64, 128)
torch.save(layer, 'layer.pth')

new_layer = torch.load('layer.pth')
```

# Fine-tuning

- A type of transfer learning
  - Smaller learning rate
  - Train part of the network (we **freeze** some of them)
  - Rule of thumb: freeze early layers of network and fine-tune layers closer to output layer

```
import torch.nn as nn

model = nn.Sequential(nn.Linear(64, 128),
                      nn.Linear(128, 256))

for name, param in model.named_parameters():
    if name == '0.weight':
        param.requires_grad = False
```

# Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

# Evaluating model performance

INTRODUCTION TO DEEP LEARNING WITH PYTORCH



**Jasmin Ludolf**

Senior Data Science Content Developer,  
DataCamp

# Training, validation and testing

- A dataset is typically split into three subsets:

	Percent of data	Role
Training	80-90%	Adjusts model parameters
Validation	10-20%	Tunes hyperparameters
Test	5-10%	Evaluates final model performance

- Track **loss** and **accuracy** during training and validation



# Calculating training loss

## For each epoch:

- Sum the loss across all batches in the dataloader
- Compute the **mean training loss** at the end of the epoch

```
training_loss = 0.0
for inputs, labels in trainloader:
    # Run the forward pass
    outputs = model(inputs)
    # Compute the loss
    loss = criterion(outputs, labels)
    # Backpropagation
    loss.backward() # Compute gradients
    optimizer.step() # Update weights
    optimizer.zero_grad() # Reset gradients

    # Calculate and sum the loss
    training_loss += loss.item()
epoch_loss = training_loss / len(trainloader)
```

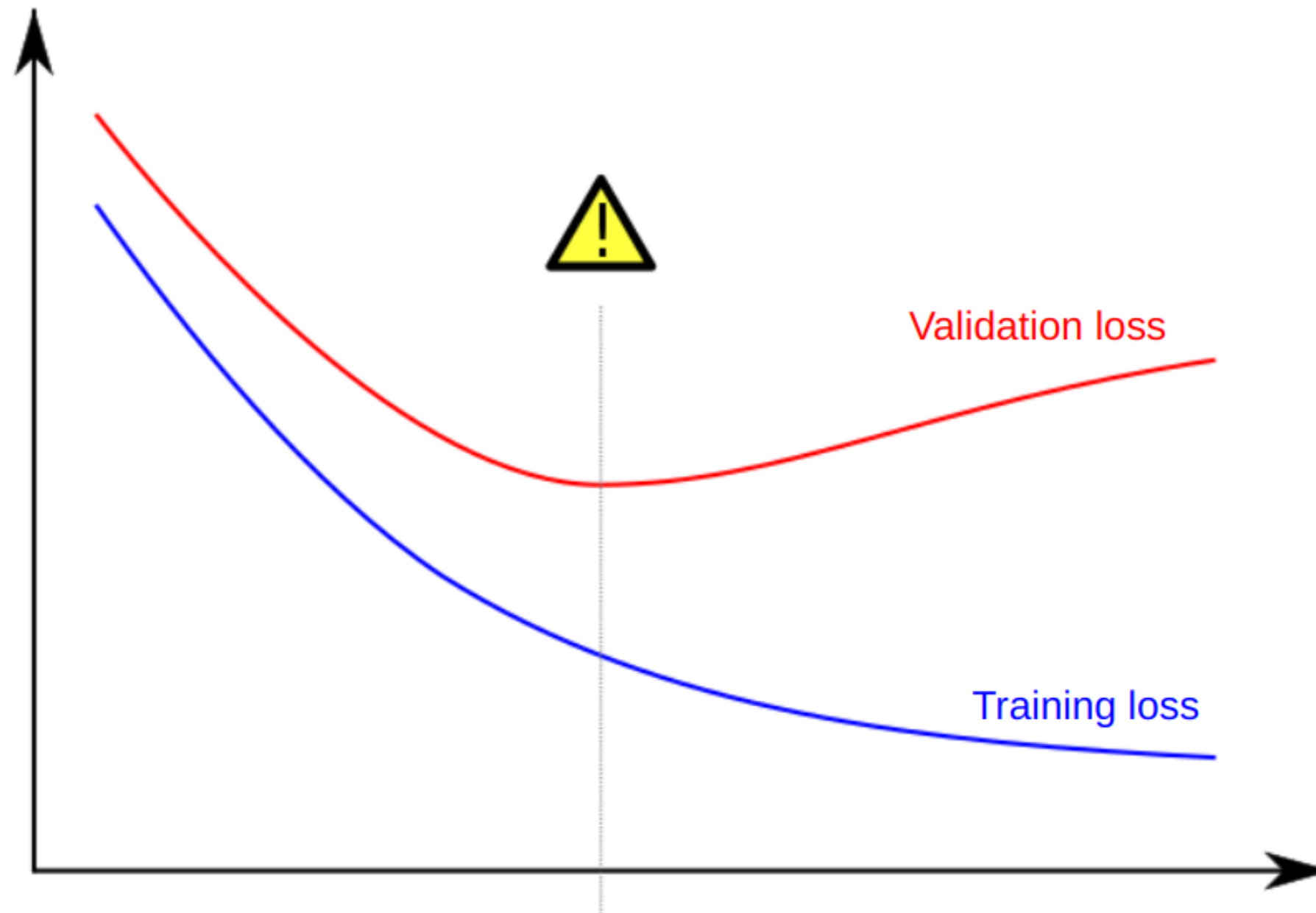
# Calculating validation loss

```
validation_loss = 0.0
model.eval() # Put model in evaluation mode

with torch.no_grad(): # Disable gradients for efficiency
    for inputs, labels in validationloader:
        # Run the forward pass
        outputs = model(inputs)
        # Calculate the loss
        loss = criterion(outputs, labels)
        validation_loss += loss.item()

epoch_loss = validation_loss / len(validationloader) # Compute mean loss
model.train() # Switch back to training mode
```

# Overfitting



# Calculating accuracy with torchmetrics

```
import torchmetrics

# Create accuracy metric
metric = torchmetrics.Accuracy(task="multiclass", num_classes=3)

for features, labels in dataloader:
    outputs = model(features) # Forward pass
    # Compute batch accuracy (keeping argmax for one-hot labels)
    metric.update(outputs, labels.argmax(dim=-1))

# Compute accuracy over the whole epoch
accuracy = metric.compute()

# Reset metric for the next epoch
metric.reset()
```

# Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

# Fighting overfitting

INTRODUCTION TO DEEP LEARNING WITH PYTORCH



**Jasmin Ludolf**

Senior Data Science Content Developer,  
DataCamp

# Reasons for overfitting

- **Overfitting:** the model does not generalize to unseen data
  - Model memorizes training data
  - Performs well on **training** data but poorly on **validation** data
- Possible causes:

Problem	Solutions
Dataset is not large enough	Get more data / use data augmentation
Model has too much capacity	Reduce model size / add dropout
Weights are too large	Weight decay

# Fighting overfitting

Strategies:

- Reducing model size or adding dropout layer
- Using weight decay to force parameters to remain small
- Obtaining new data or augmenting data



# "Regularization" using a dropout layer

- Randomly zeroes out elements of the input tensor **during training**

```
model = nn.Sequential(nn.Linear(8, 4),  
                      nn.ReLU(),  
                      nn.Dropout(p=0.5))  
  
features = torch.randn((1, 8))  
print(model(features))
```

```
tensor([[1.4655, 0.0000, 0.0000, 0.8456]], grad_fn=<MulBackward0>)
```

- Dropout is added **after** the activation function
- Behaves differently in training vs. evaluation - use `model.train()` for training and `model.eval()` to disable dropout during evaluation

# Regularization with weight decay

```
optimizer = optim.SGD(model.parameters(), lr=0.001, weight_decay=0.0001)
```

- Controlled by the **weight\_decay** parameter in the optimizer, typically set to a small value (e.g., 0.0001)
- **Weight decay** encourages smaller weights by adding a penalty during optimization
- Helps reduce overfitting, keeping weights smaller and improving generalization

# Data augmentation



# Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

# Improving model performance

INTRODUCTION TO DEEP LEARNING WITH PYTORCH



**Jasmin Ludolf**

Senior Data Science Content Developer,  
DataCamp

# Steps to maximize performance

- Can we solve the problem?
- Set a performance baseline
- Increase performance on the validation set
- Achieve the best possible performance

**Step 1:**  
Overfit the training set

**Step 2:**  
Reduce overfitting

**Step 3:**  
Fine-tune the  
hyperparameters

# Step 1: overfit the training set

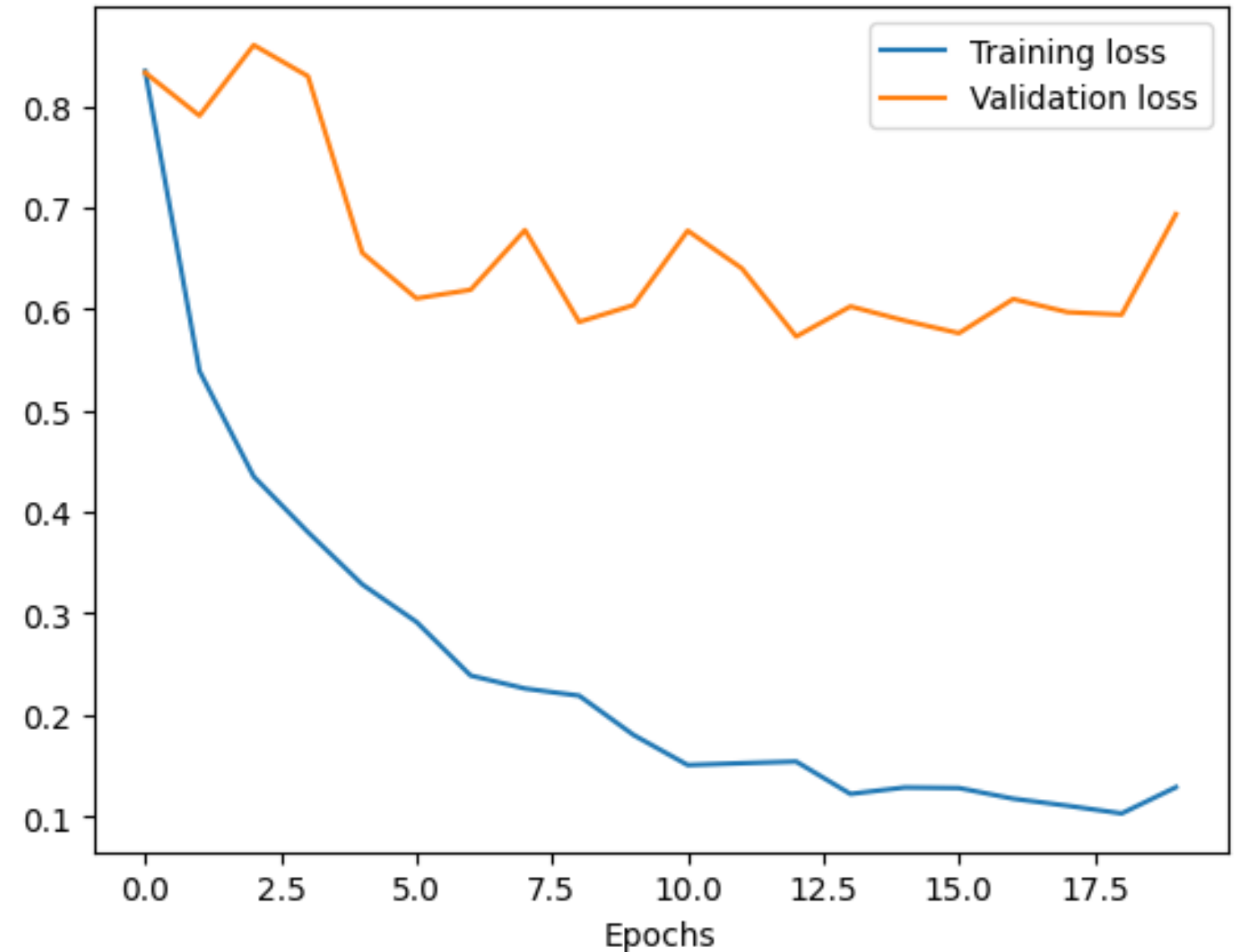
- Modify the training loop to overfit a single data point

```
features, labels = next(iter(data_loader))
for i in range(1000):
    outputs = model(features)
    loss = criterion(outputs, labels)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

- Should reach 1.0 accuracy and 0 loss
- Then scale up to the entire training set
  - Keep default hyperparameters

# Step 2: reduce overfitting

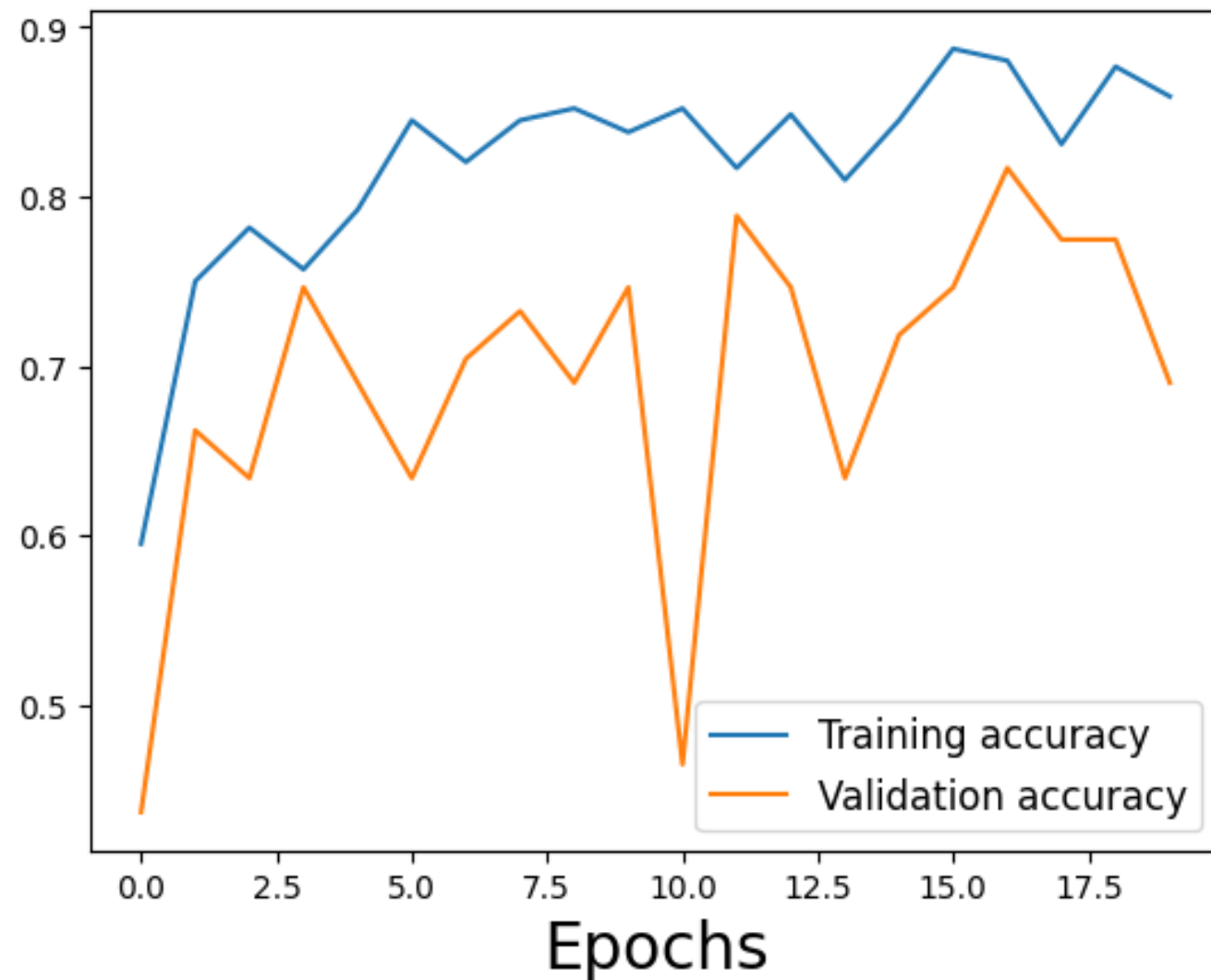
- **Goal:** maximize the validation accuracy
- Experiment with:
  - Dropout
  - Data augmentation
  - Weight decay
  - Reducing model capacity
- Keep track of each hyperparameter and validation accuracy



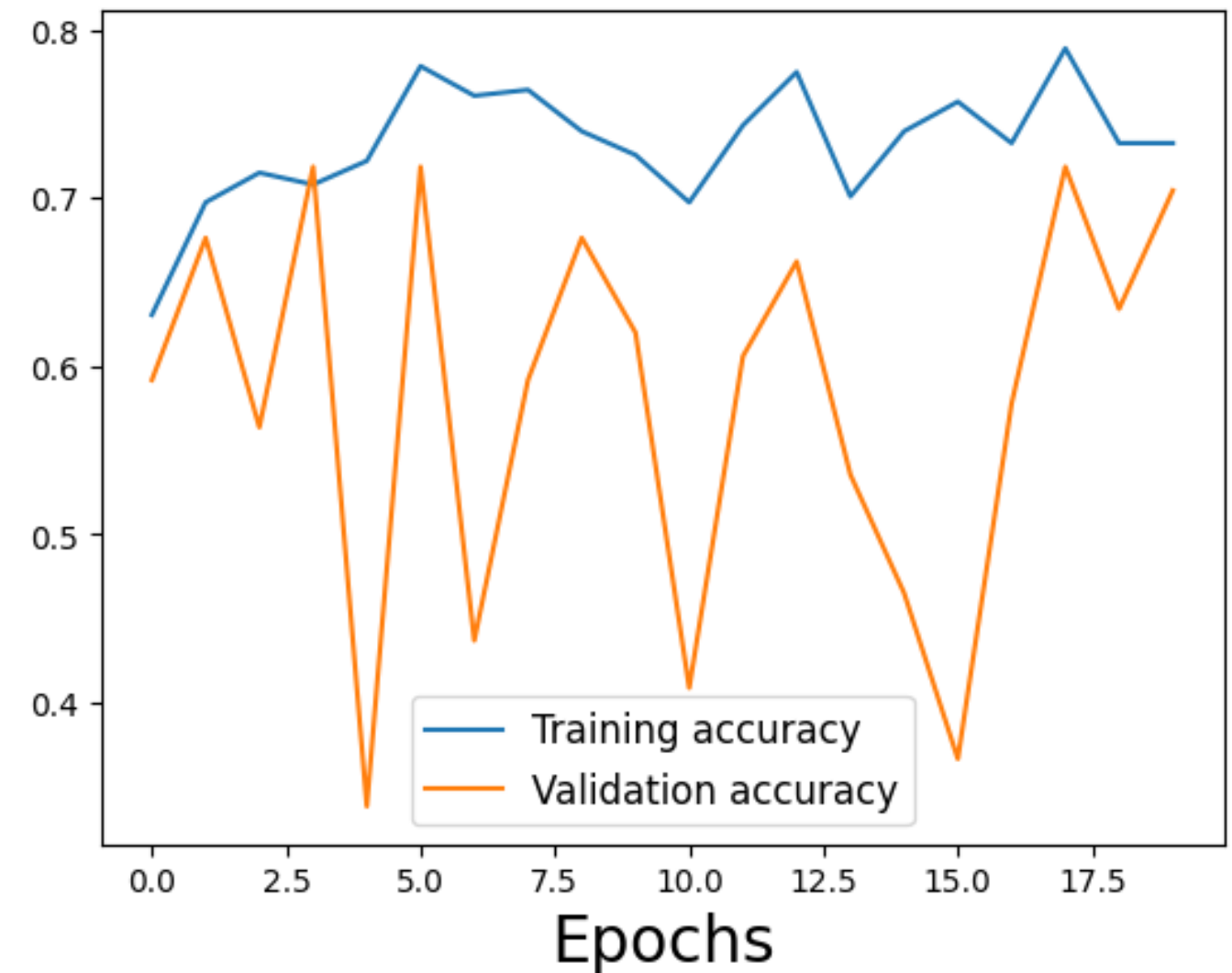


# Step 2: reduce overfitting

Original model overfitting training data



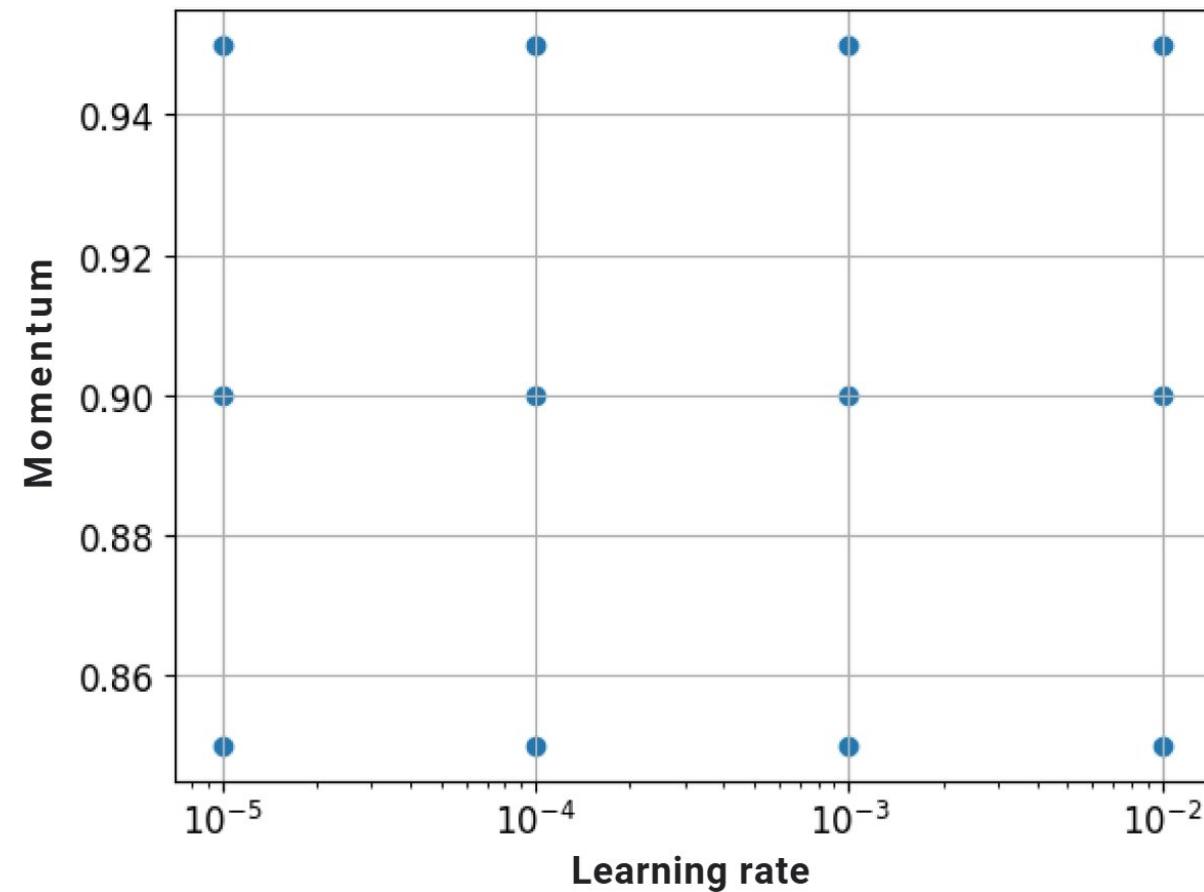
Updated model with too much regularization



# Step 3: fine-tune hyperparameters

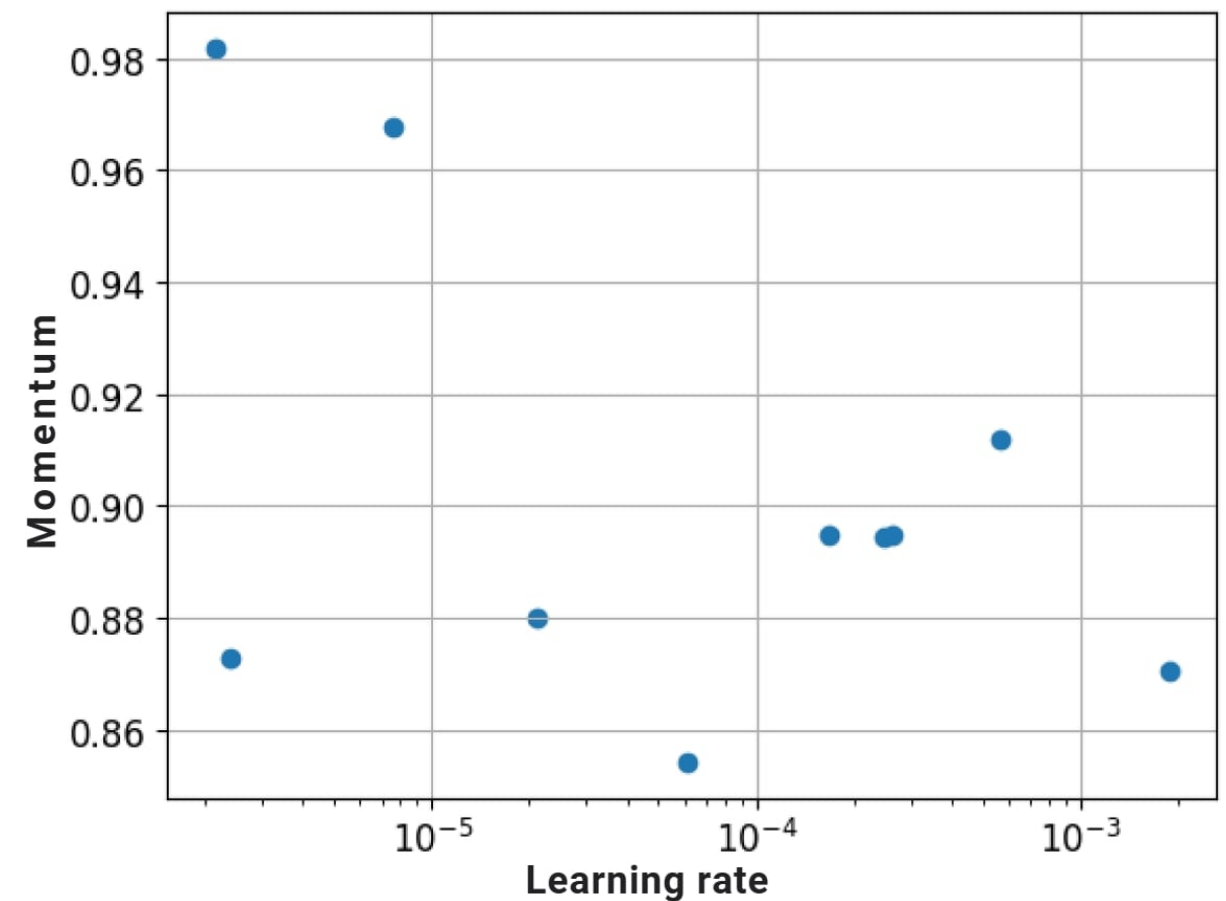
- Grid search

```
for factor in range(2, 6):  
    lr = 10 ** -factor
```



- Random search

```
factor = np.random.uniform(2, 6)  
lr = 10 ** -factor
```



# Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

# Wrap-up video

INTRODUCTION TO DEEP LEARNING WITH PYTORCH



**Jasmin Ludolf**

Senior Data Science Content Developer,  
DataCamp

# Summary

- **Chapter 1**
  - Discovered deep learning
  - Created small neural networks
  - Discovered linear layers
- **Chapter 2**
  - Used loss and activation functions
  - Calculated derivatives
  - Use backpropagation
- **Chapter 3**
  - Trained a neural network
  - Played with learning rate and momentum
  - And learned about their impact
- **Chapter 4**
  - Strategies to improve your model
  - Reduced overfitting
  - Evaluated model performance

# Next steps

- Course
  - [Intermediate Deep Learning with PyTorch](#)
  - [Explainable AI in Python](#)
- Learn
  - Probability and statistics
  - Linear algebra
  - Calculus
- Practice
  - Pick a dataset on Kaggle
  - [Detecting Cybersecurity Threats using Deep Learning](#)
  - Train a neural network
- Create
  - Use deep learning to make an app

# Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH