

A deeper dive into loading data

INTRODUCTION TO DEEP LEARNING WITH PYTORCH



Back to our animals dataset

```
import pandas as pd  
pd.read_csv('animals.csv')
```

| animal_name | hair | feathers | eggs | milk | predator | fins | legs | tail | type |
|-------------|------|----------|------|------|----------|------|------|------|------|
| skimmer | 0 | 1 | 1 | 0 | 1 | 0 | 2 | 1 | 2 |
| gull | 0 | 1 | 1 | 0 | 1 | 0 | 2 | 1 | 2 |
| seahorse | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 4 |
| tuatara | 0 | 0 | 1 | 0 | 1 | 0 | 4 | 1 | 3 |
| squirrel | 1 | 0 | 0 | 1 | 0 | 0 | 2 | 1 | 1 |

Type key: **mammal** (1), **bird** (2), **reptile** (3), **fish** (4), **amphibian** (5), **bug** (6), **invertebrate** (7).

Back to our animals dataset: defining features

```
import numpy as np  
  
# Define input features  
  
features = animals.iloc[:, 1:-1]  
  
X = features.to_numpy()  
  
print(X)
```

```
array([[0, 1, 1, 0, 1, 0, 2, 1],  
       [0, 1, 1, 0, 1, 0, 2, 1],  
       [0, 0, 1, 0, 0, 1, 0, 1],  
       [0, 0, 1, 0, 1, 0, 4, 1],  
       [1, 0, 0, 1, 0, 0, 2, 1]])
```

Back to our animals dataset: defining target values

```
# Define target features (ground truth)  
target = animals.iloc[:, -1]  
y = target.to_numpy()
```

```
array([2, 2, 4, 3, 1])
```

Recalling TensorDataset

```
import torch  
from torch.utils.data import TensorDataset
```

```
# Instantiate dataset class  
dataset = TensorDataset(torch.tensor(X).float(), torch.tensor(y).float())
```

```
# Access an individual sample  
sample = dataset[0]  
input_sample, label_sample = sample  
print('input sample:', input_sample)  
print('label_sample:', label_sample)
```

```
input sample: tensor([0., 1., 1., 0., 1., 0., 2., 1.])  
label_sample: tensor(2.)
```

Recalling DataLoader

```
from torch.utils.data import DataLoader  
  
batch_size = 2  
shuffle = True  
  
# Create a DataLoader  
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)
```

Recalling DataLoader

```
# Iterate over the dataloader
for batch_inputs, batch_labels in dataloader:
    print('batch inputs', batch_inputs)
    print('batch labels', batch_labels)
```

```
batch inputs: tensor([[0., 0., 1., 0., 0., 1., 0., 1.],
                      [0., 1., 1., 0., 1., 0., 2., 1.]])
batch labels: tensor([4., 2.])
batch inputs: tensor([[0., 1., 1., 0., 1., 0., 2., 1.],
                      [1., 0., 0., 1., 0., 0., 2., 1.]])
batch labels: tensor([2., 1.])
batch inputs: tensor([[0., 0., 1., 0., 1., 0., 4., 1.]])
batch labels: tensor([3.])
```

Evaluating model performance

INTRODUCTION TO DEEP LEARNING WITH PYTORCH



Training, validation and testing

- Raw dataset is usually split in three subsets:

| | Percent of data | Role |
|------------|-----------------|---|
| Training | 80-90% | Used to adjust the model's parameters |
| Validation | 10-20% | Used for hyperparameter tuning |
| Testing | 5-10% | Only used once to calculate final metrics |

Model evaluation metrics

- In this video, we'll focus on evaluating:
 - **Loss**
 - Training
 - Validation
 - **Accuracy**
 - Training
 - Validation
- In classification, **accuracy** measures how well model correctly predicts ground truth labels

Calculating training loss

- For each epoch:
 - we sum up the loss for each iteration of the training set dataloader
 - at the end of the epoch, we calculate the mean training loss

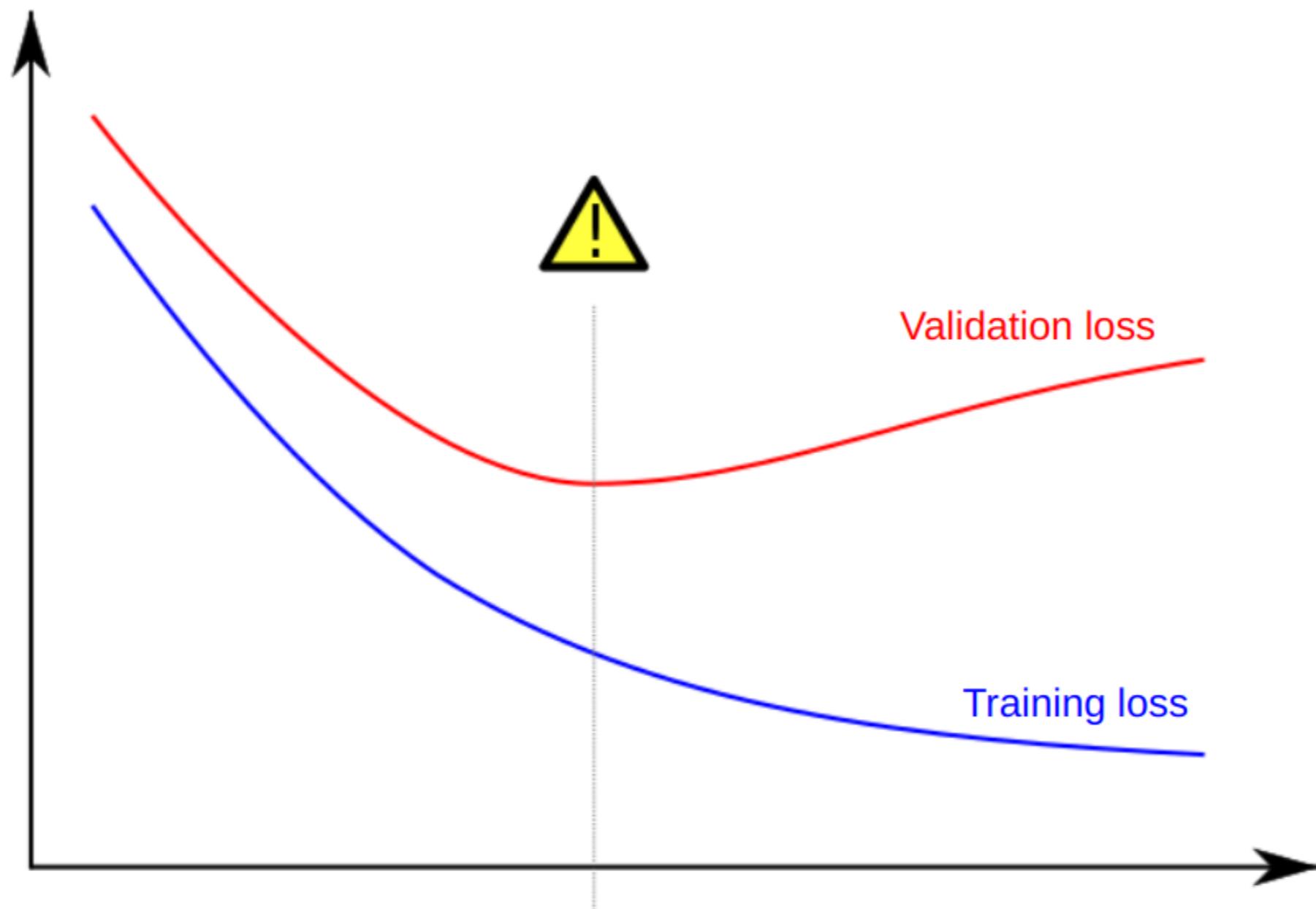
```
training_loss = 0.0
for i, data in enumerate(trainloader, 0):
    # Run the forward pass
    ...
    # Calculate the loss
    loss = criterion(outputs, labels)
    # Calculate the gradients
    ...
    # Calculate and sum the loss
    training_loss += loss.item()
epoch_loss = training_loss / len(trainloader)
```

Calculating validation loss

- After the training epoch, we iterate over the validation set and calculate the average validation loss

```
validation_loss = 0.0
model.eval() # Put model in evaluation mode
with torch.no_grad(): # Speed up the forward pass
    for i, data in enumerate(validationloader, 0):
        # Run the forward pass
        ...
        # Calculate the loss
        loss = criterion(outputs, labels)
        validation_loss += loss.item()
epoch_loss = validation_loss / len(validationloader)
model.train()
```

Overfitting



Calculating accuracy with torchmetrics

```
import torchmetrics

# Create accuracy metric using torch metrics
metric = torchmetrics.Accuracy(task="multiclass", num_classes=3)
for i, data in enumerate(dataloader, 0):
    features, labels = data
    outputs = model(features)
    # Calculate accuracy over the batch
    acc = metric(outputs, labels.argmax(dim=-1))
# Calculate accuracy over the whole epoch
acc = metric.compute()
print(f"Accuracy on all data: {acc}")
# Reset the metric for the next epoch (training or validation)
metric.reset()
```

Fighting overfitting

INTRODUCTION TO DEEP LEARNING WITH PYTORCH



Reasons for overfitting

- **Overfitting:** the model does not generalize to unseen data.
 - model memorizes training data
 - good performances on the training set / poor performances on the validation set
- 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))  
model(i)
```

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

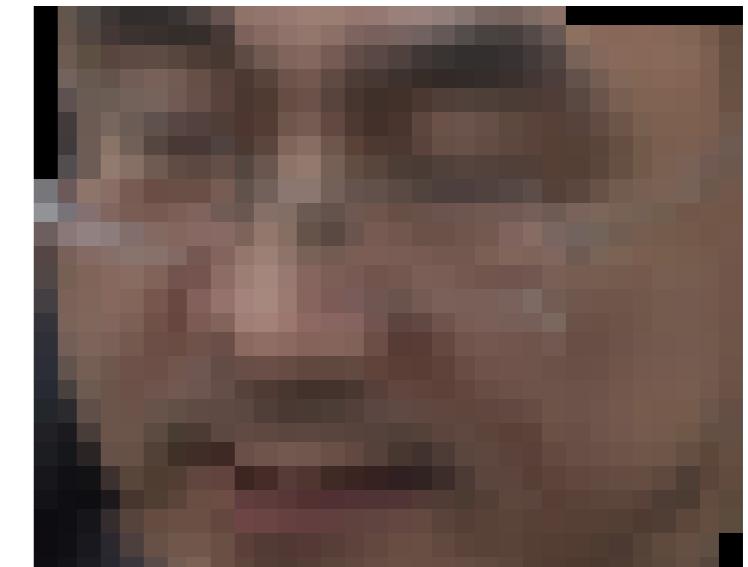
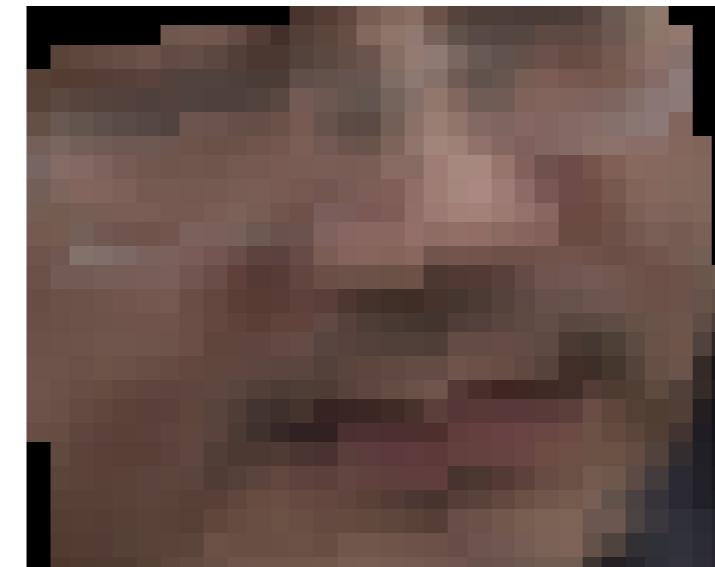
- Dropout is added **after** the activation function
- Behaves differently during training and evaluation; we must remember to switch modes using `model.train()` and `model.eval()`

Regularization with weight decay

```
optimizer = optim.SGD(model.parameters(), lr=1e-3, weight_decay=1e-4)
```

- Optimizer's `weight_decay` parameter takes values between zero and one
 - Typically small value, e.g. $1e-3$
- Weight decay adds penalty to loss function to discourage large weights and biases
- The higher the parameter, the less likely the model is to overfit

Data augmentation



Improving model performance

INTRODUCTION TO DEEP LEARNING WITH PYTORCH



Steps to maximize performance

- Overfit the training set
 - can we solve the problem?
 - sets a performance baseline
- Reduce overfitting
 - improve performances on the validation set
- Fine-tune hyperparameters

Step 1:

Overfit the training set

Step 2:

Reduce overfitting

Step 3:

Fine-tune the hyperparameters

Step 1: overfit the training set

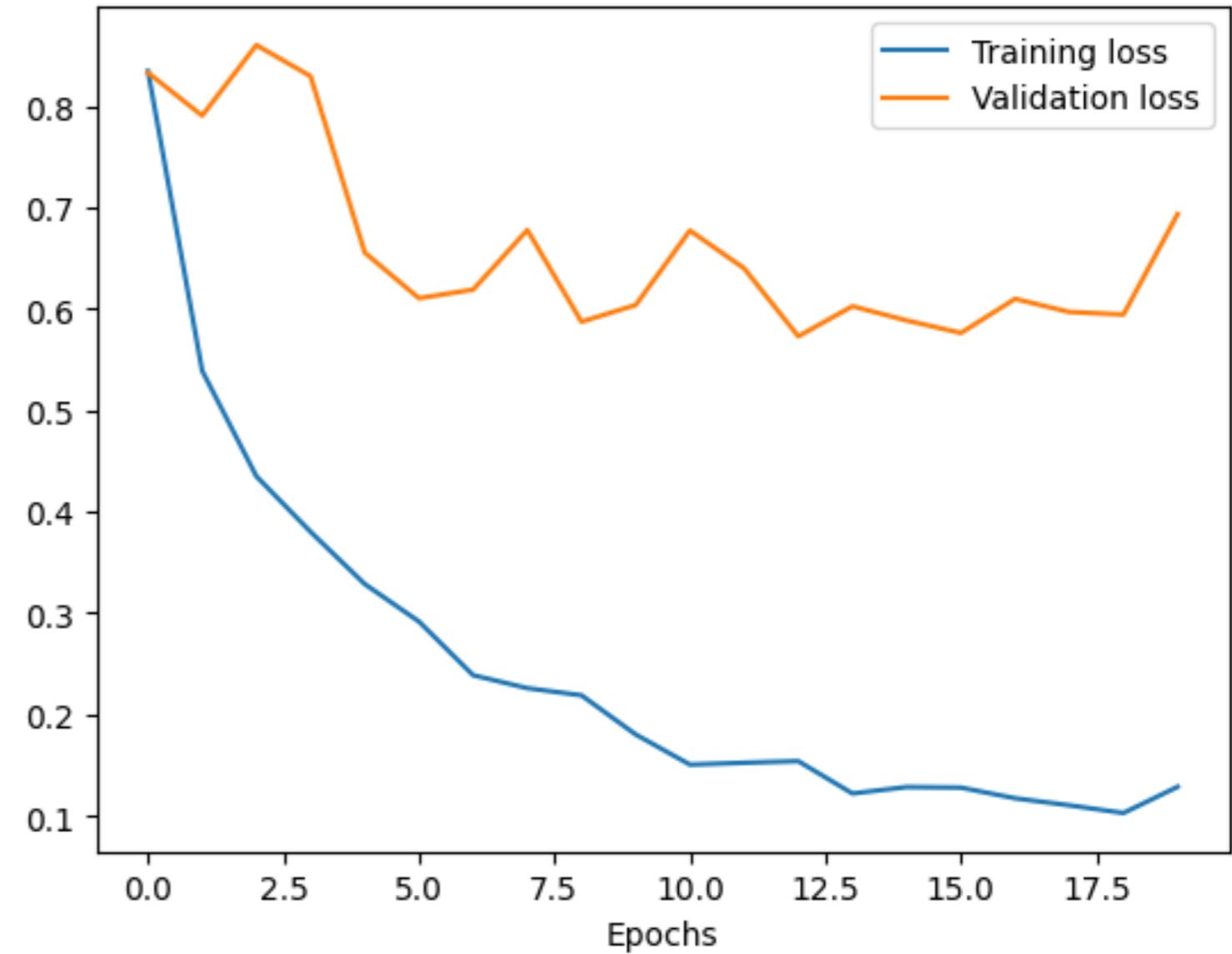
- Modify the training loop to overfit a single data point (batch size of 1)

```
features, labels = next(iter(trainloader))
for i in range(1e3):
    outputs = model(features)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
```

- should reach 1.0 accuracy and 0 loss
- helps finding bugs in the code
- **Goal:** minimize the training loss
 - create large enough model
 - use a default learning rate

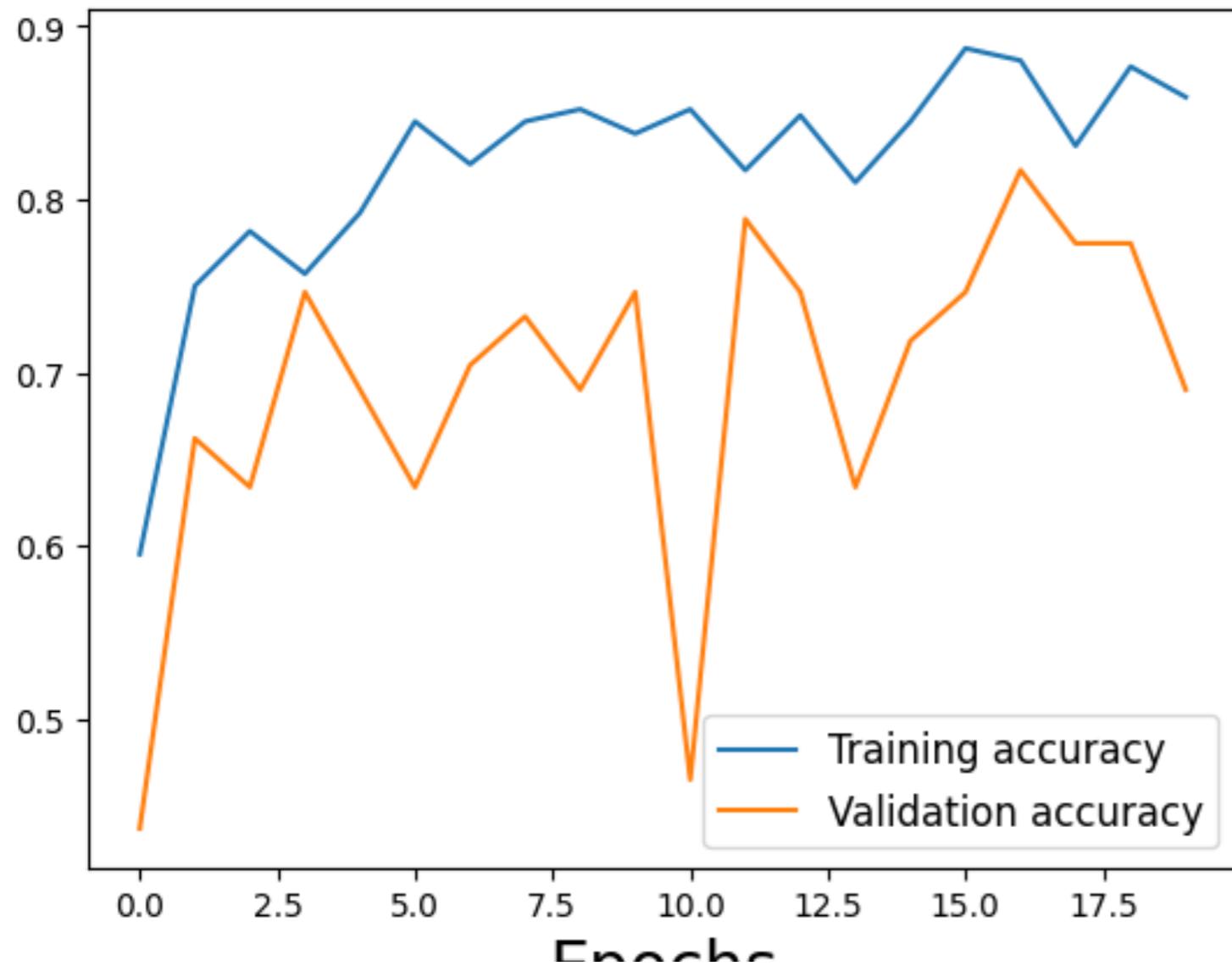
Step 2: reduce overfitting

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

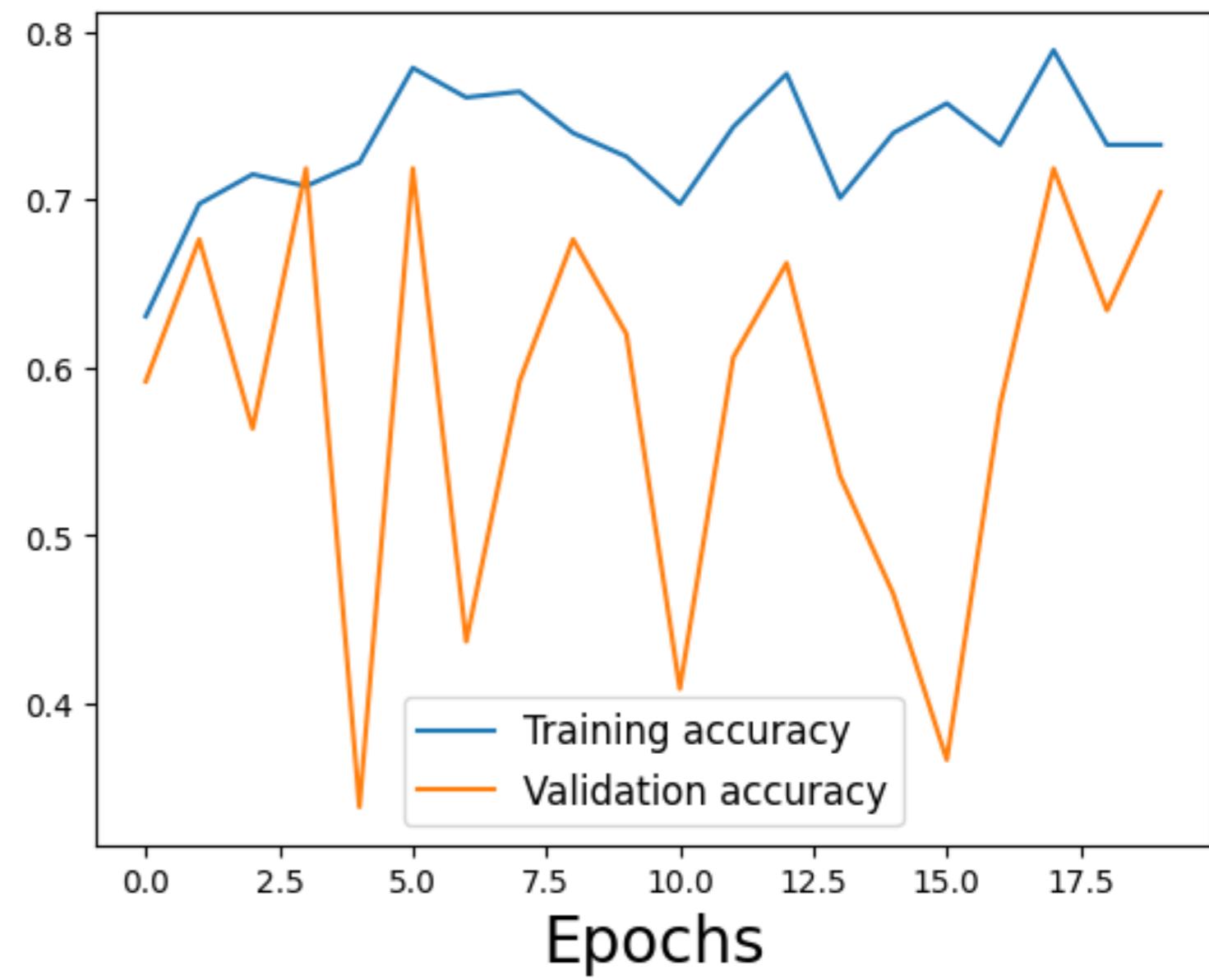


Step 2: reduce overfitting

Original model overfitting the training data



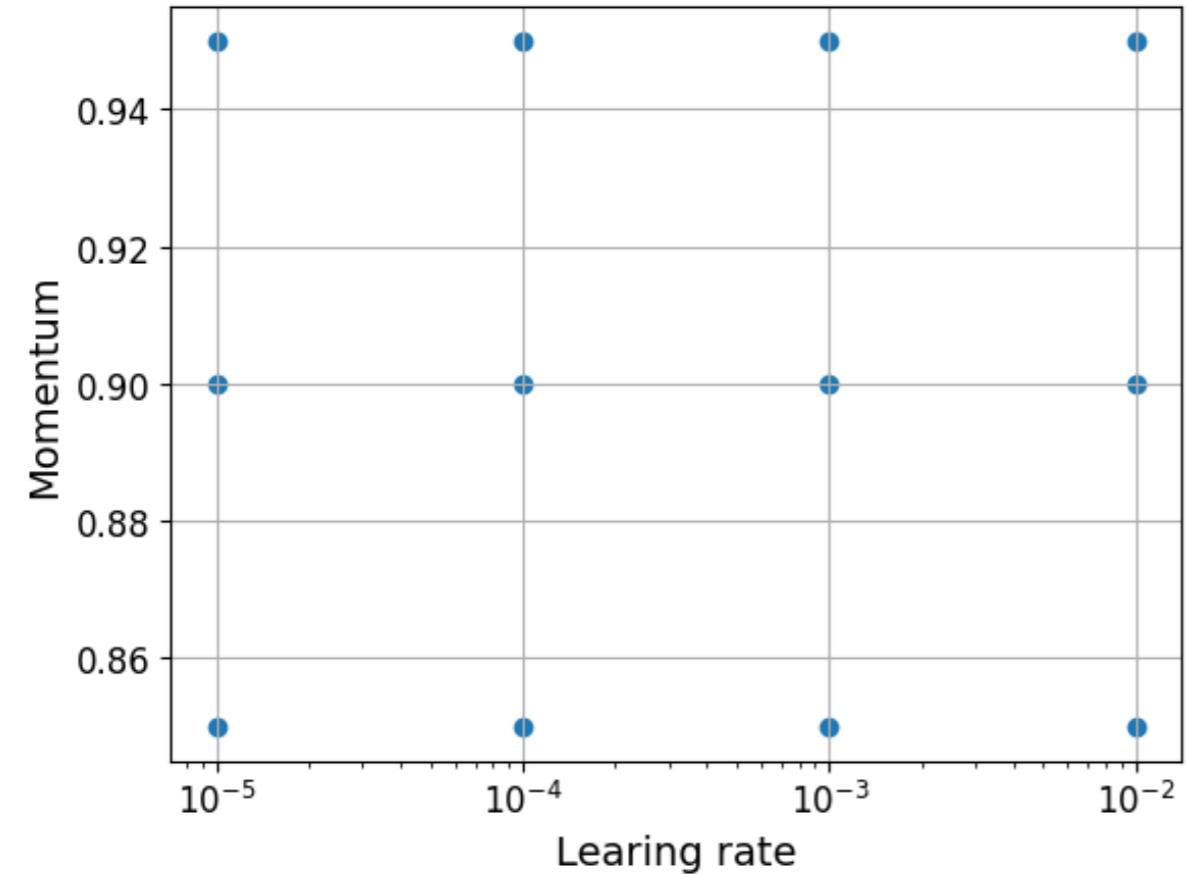
Model with too much regularization



Step 3: fine-tune hyperparameters

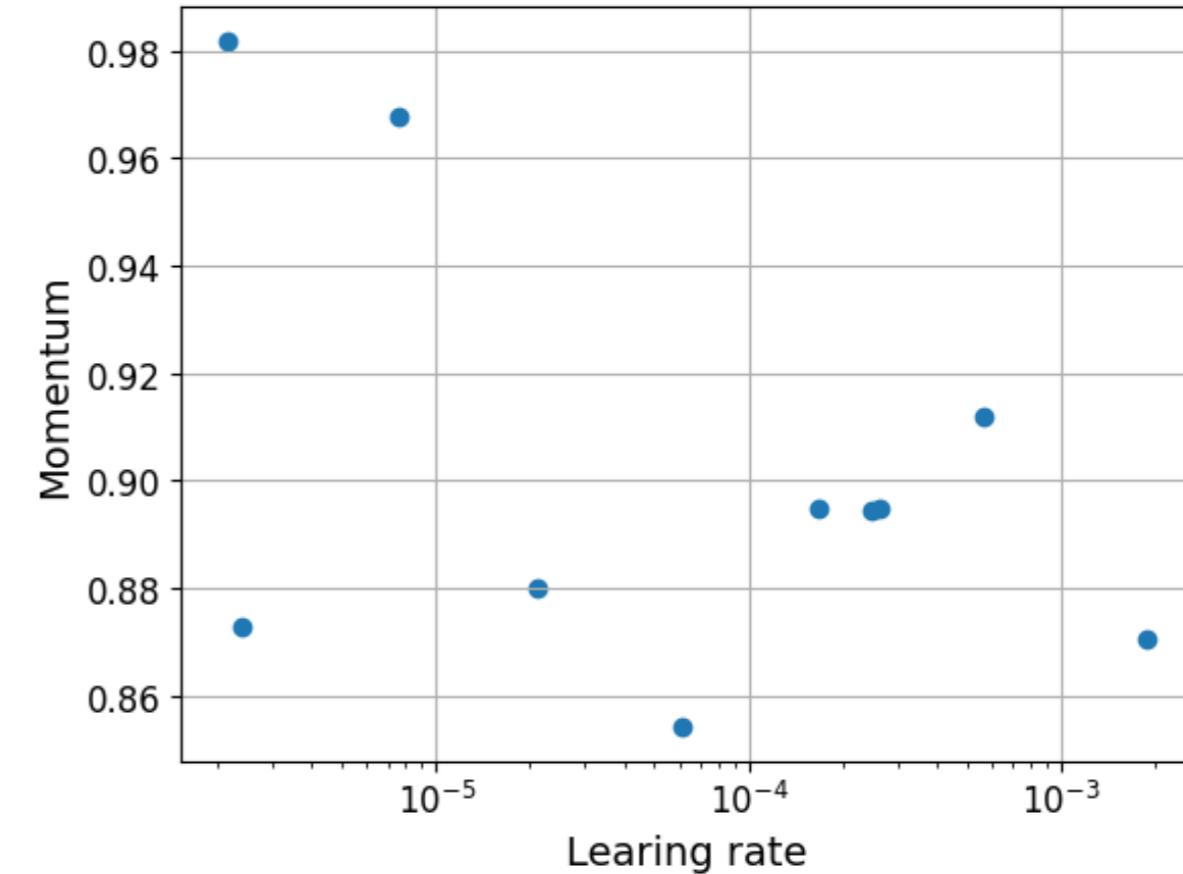
- Grid search

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



- Random search

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



Wrap-up video

INTRODUCTION TO DEEP LEARNING WITH PYTORCH



Summary

- **Chapter 1**
 - Discovered deep learning
 - Created small neural networks
 - Discovered linear layers and activation functions
- **Chapter 2**
 - Created and used loss functions
 - Calculated derivatives and use backpropagation
 - Trained a neural network
- **Chapter 3**
 - Manipulated the architecture of a neural network
 - Played with learning rate and momentum
 - Learned about transfer learning
- **Chapter 4**
 - Learned about dataloaders
 - Reduced overfitting
 - Evaluated model performance

Next steps

- Course
 - [Intermediate Deep Learning with PyTorch](#)
- Learn
 - Probability and statistics
 - Linear algebra
 - Calculus
- Practice
 - Pick a dataset on Kaggle
 - Check out DataCamp workspace
 - Train a neural network

