PyTorch and object-oriented programming

INTERMEDIATE DEEP LEARNING WITH PYTORCH

Michal Oleszak
Machine Learning Engineer

What we will learn

How to train robust deep learning models:

- Improving training with optimizers
- Mitigating vanishing and exploding gradients
- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks (RNNs)
- Multi-input and multi-output models

Prerequisites

The course assumes you are comfortable with the following topics:

- Neural networks training:
  - Forward pass
  - Loss calculation
  - Backward pass (backpropagation)
- Training models with PyTorch:
  - Datasets and DataLoaders
  - Model training loop
  - Model evaluation
- Prerequisite course: Introduction to Deep Learning with PyTorch

Object-Oriented Programming (OOP)

- We will use OOP to define:
  - PyTorch Datasets
  - PyTorch Models
- In OOP, we create objects with:
  - Abilities (methods)
  - Data (attributes)

Object-Oriented Programming (OOP)

```python
class BankAccount:
    def __init__(self, balance):
        self.balance = balance
```

- `__init__` is called when `BankAccount` object is created
- `balance` is the attribute of the `BankAccount` object

```python
account = BankAccount(100)
print(account.balance)
```

```
100
```

Object-Oriented Programming (OOP)

- Methods: Python functions to perform tasks
- `deposit` method increases balance

```
class BankAccount:
    def __init__(self, balance):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
```

```
account = BankAccount(100)
account.deposit(50)
print(account.balance)
```

```
150
```

Water potability dataset

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalomethanes	Turbidity	Potability
0	0.587349	0.577747	0.386298	0.568199	0.647347	0.292985	0.654522	0.795029	0.630115	0
1	0.643654	0.441300	0.314381	0.439304	0.514545	0.356685	0.377248	0.202914	0.520358	0
2	0.388934	0.470876	0.506122	0.524364	0.561537	0.142913	0.249922	0.401487	0.219973	0
3	0.725820	0.715942	0.506141	0.521683	0.751819	0.148683	0.467200	0.658678	0.242428	0
4	0.610517	0.532588	0.237701	0.270288	0.495155	0.494792	0.409721	0.469762	0.585049	0
...	...	...	...	...	...	...	...	...	...	...
2006	0.636224	0.580511	0.277748	0.418063	0.522486	0.342184	0.310364	0.402799	0.627156	1
2007	0.470143	0.548826	0.301347	0.538273	0.498565	0.231359	0.565061	0.175889	0.395061	1
2008	0.817826	0.087434	0.656389	0.670774	0.369089	0.431872	0.563265	0.285745	0.578674	1
2009	0.424187	0.464092	0.459656	0.541633	0.615572	0.388360	0.397780	0.449156	0.440004	1
2010	0.322425	0.492891	0.841409	0.492136	0.656047	0.588709	0.471422	0.503458	0.591867	1

PyTorch Dataset

```python
from torch.utils.data import Dataset

class WaterDataset(Dataset):
    def __init__(self, csv_path):
        super().__init__()
        df = pd.read_csv(csv_path)
        self.data = df.to_numpy()

    def __len__(self):
        return self.data.shape[0]

    def __getitem__(self, idx):
        features = self.data[idx, :-1]
        label = self.data[idx, -1]
        return features, label
```

- **init:** load data, store as numpy array
  - `super().__init__()` ensures `WaterDataset` behaves like torch `Dataset`
- **len:** return the size of the dataset
- **getitem:**
  - take one argument called `idx`
  - return features and label for a single sample at index `idx`

PyTorch DataLoader

```
dataset_train = WaterDataset(
    "water_train.csv"
)
```

```
from torch.utils.data import DataLoader

dataloader_train = DataLoader(
    dataset_train,
    batch_size=2,
    shuffle=True,
)
```

```
features, labels = next(iter(dataloader_train))
print(f"Features: {features},\nLabels: {labels}")
```

```
Features: tensor([
  [0.4899, 0.4180, 0.6299, 0.3496, 0.4575,
   0.3615, 0.3259, 0.5011, 0.7545],
  [0.7953, 0.6305, 0.4480, 0.6549, 0.7813,
   0.6566, 0.6340, 0.5493, 0.5789]
]),
Labels: tensor([1., 0.])
```

PyTorch Model

Sequential model definition:

```
net = nn.Sequential(
  nn.Linear(9, 16),
  nn.ReLU(),
  nn.Linear(16, 8),
  nn.ReLU(),
  nn.Linear(8, 1),
  nn.Sigmoid(),
)
```

Class-based model definition:

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(9, 16)
        self.fc2 = nn.Linear(16, 8)
        self.fc3 = nn.Linear(8, 1)

    def forward(self, x):
        x = nn.functional.relu(self.fc1(x))
        x = nn.functional.relu(self.fc2(x))
        x = nn.functional.sigmoid(self.fc3(x))
        return x

net = Net()
```

Let's practice!

INTERMEDIATE DEEP LEARNING WITH PYTORCH

Optimizers, training, and evaluation

INTERMEDIATE DEEP LEARNING WITH PYTORCH

Michal Oleszak
Machine Learning Engineer

Training loop

```
import torch.nn as nn
import torch.optim as optim

criterion = nn.BCELoss()
optimizer = optim.SGD(net.parameters(), lr=0.01)

for epoch in range(1000):
    for features, labels in dataloader_train:
        optimizer.zero_grad()
        outputs = net(features)
        loss = criterion(
          outputs, labels.view(-1, 1)
        )
        loss.backward()
        optimizer.step()
```

- Define loss function and optimizer
  - `BCELoss` for binary classification
  - `SGD` optimizer
- Iterate over epochs and training batches
- Clear gradients
- Forward pass: get model's outputs
- Compute loss
- Compute gradients
- Optimizer's step: update params

How an optimizer works

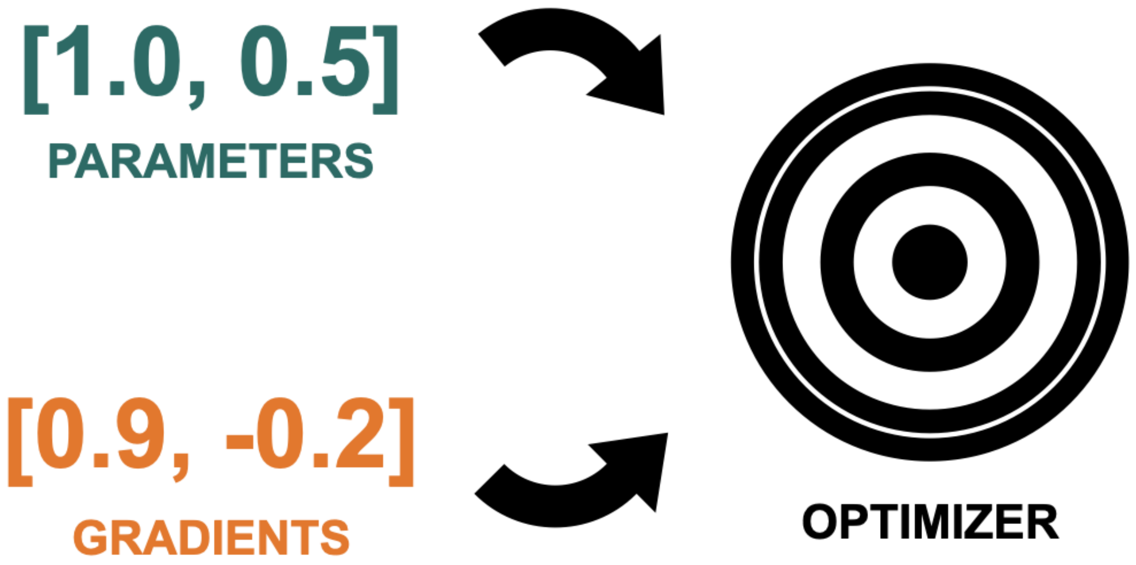
[1.0, 0.5]

PARAMETERS

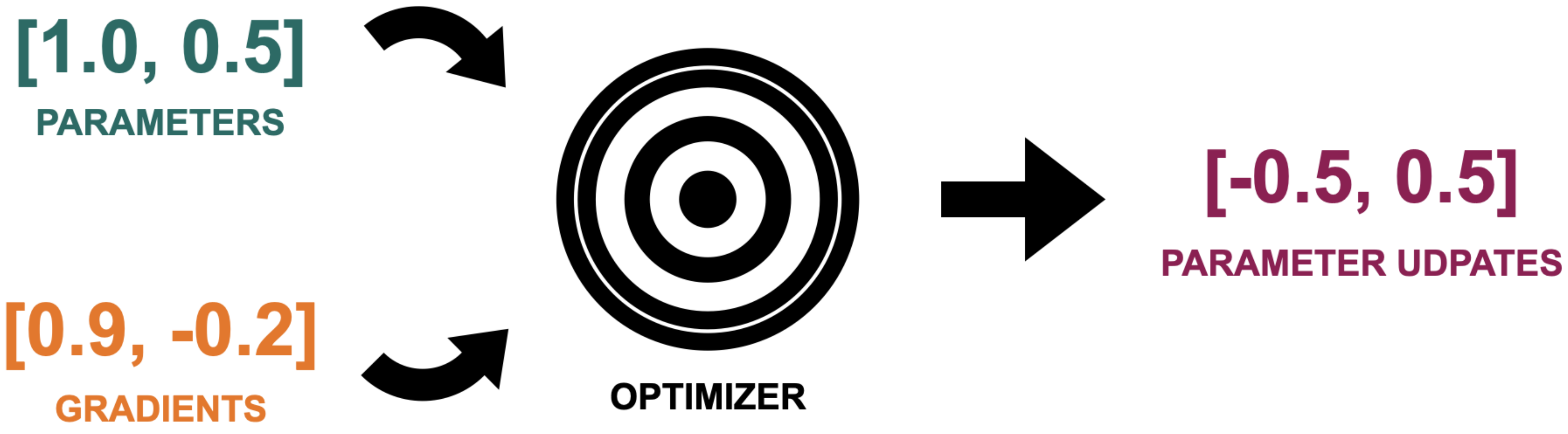
[0.9, -0.2]

GRADIENTS

How an optimizer works

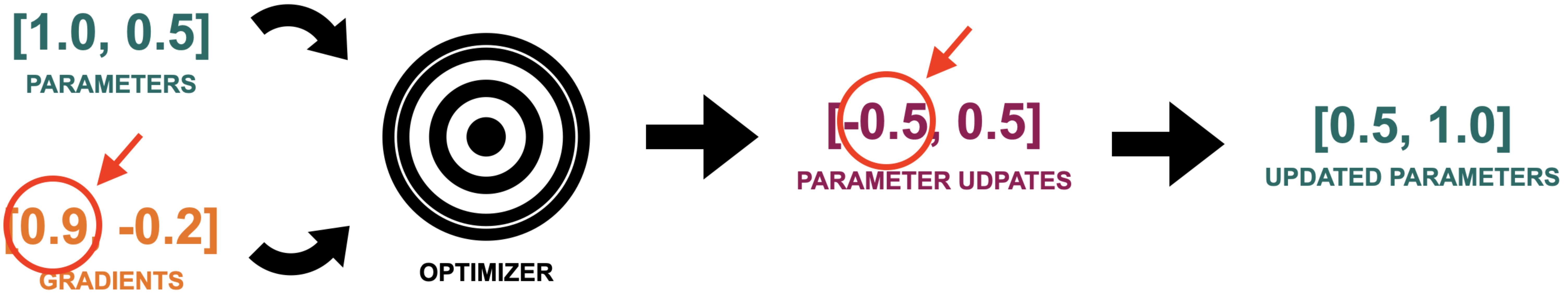


How an optimizer works



How an optimizer works

How an optimizer works



Stochastic Gradient Descent (SGD)

```
optimizer = optim.SGD(net.parameters(), lr=0.01)
```

- Update depends on learning rate
- Simple and efficient, for basic models
- Rarely used in practice

Adaptive Gradient (Adagrad)

```
optimizer = optim.Adagrad(net.parameters(), lr=0.01)
```

- Adapts learning rate for each parameter
- Good for sparse data
- May decrease the learning rate too fast

Root Mean Square Propagation (RMSprop)

```
optimizer = optim.RMSprop(net.parameters(), lr=0.01)
```

- Update for each parameter based on the size of its previous gradients

Adaptive Moment Estimation (Adam)

```
optimizer = optim.Adam(net.parameters(), lr=0.01)
```

- Arguably the most versatile and widely used
- RMSprop + gradient momentum
- Often used as the go-to optimizer

Model evaluation

```
from torchmetrics import Accuracy

acc = Accuracy(task="binary")

net.eval()
with torch.no_grad():
    for features, labels in dataloader_test:
        outputs = net(features)
        preds = (outputs >= 0.5).float()
        acc(preds, labels.view(-1, 1))

accuracy = acc.compute()
print(f"Accuracy: {accuracy}")
```

```
Accuracy: 0.6759443283081055
```

- Set up accuracy metric
- Put model in eval mode and iterate over test data batches with no gradients
- Pass data to model to get predicted probabilities
- Compute predicted labels
- Update accuracy metric

Let's practice!

INTERMEDIATE DEEP LEARNING WITH PYTORCH

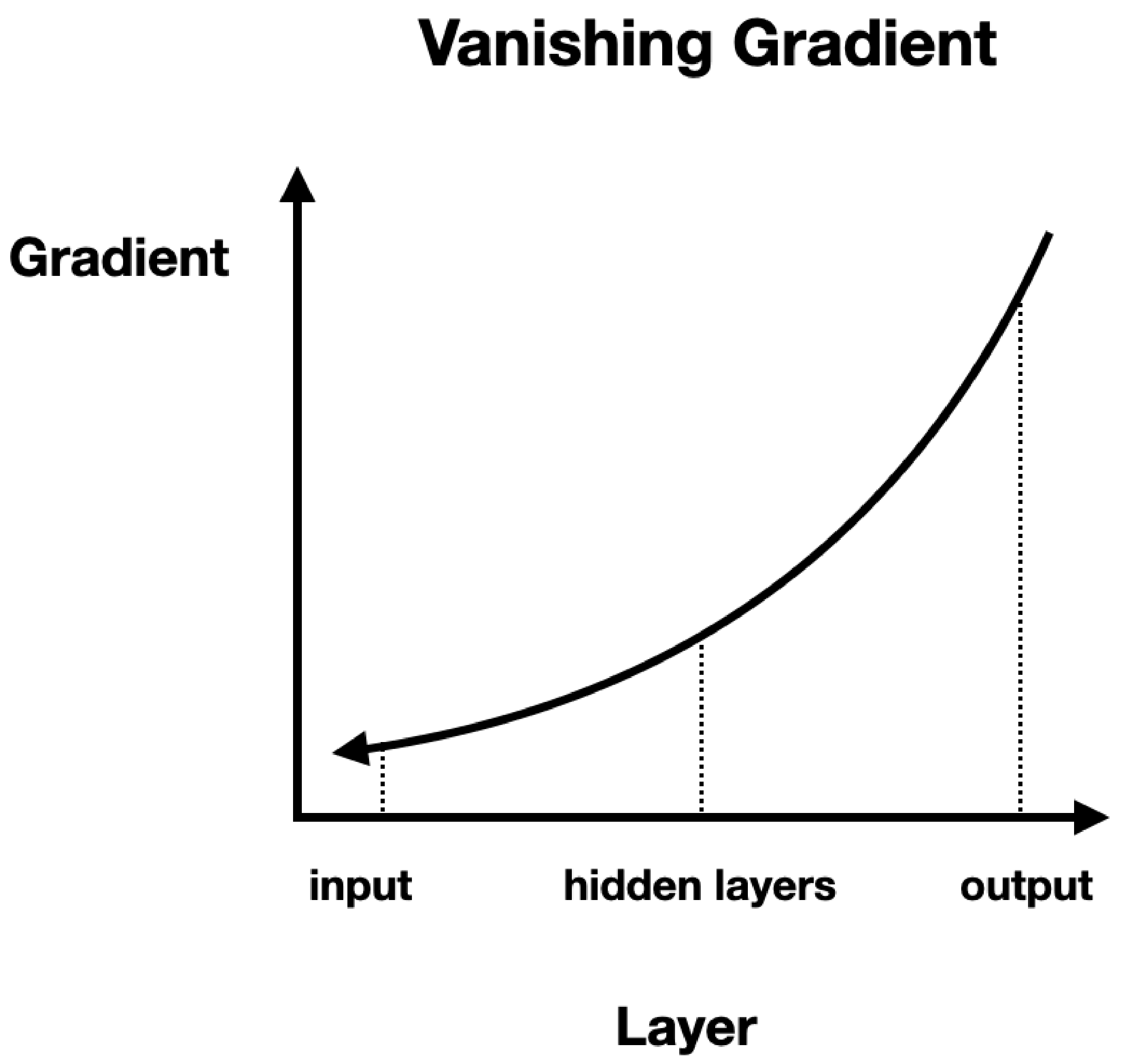
Vanishing and exploding gradients

INTERMEDIATE DEEP LEARNING WITH PYTORCH

Michal Oleszak
Machine Learning Engineer

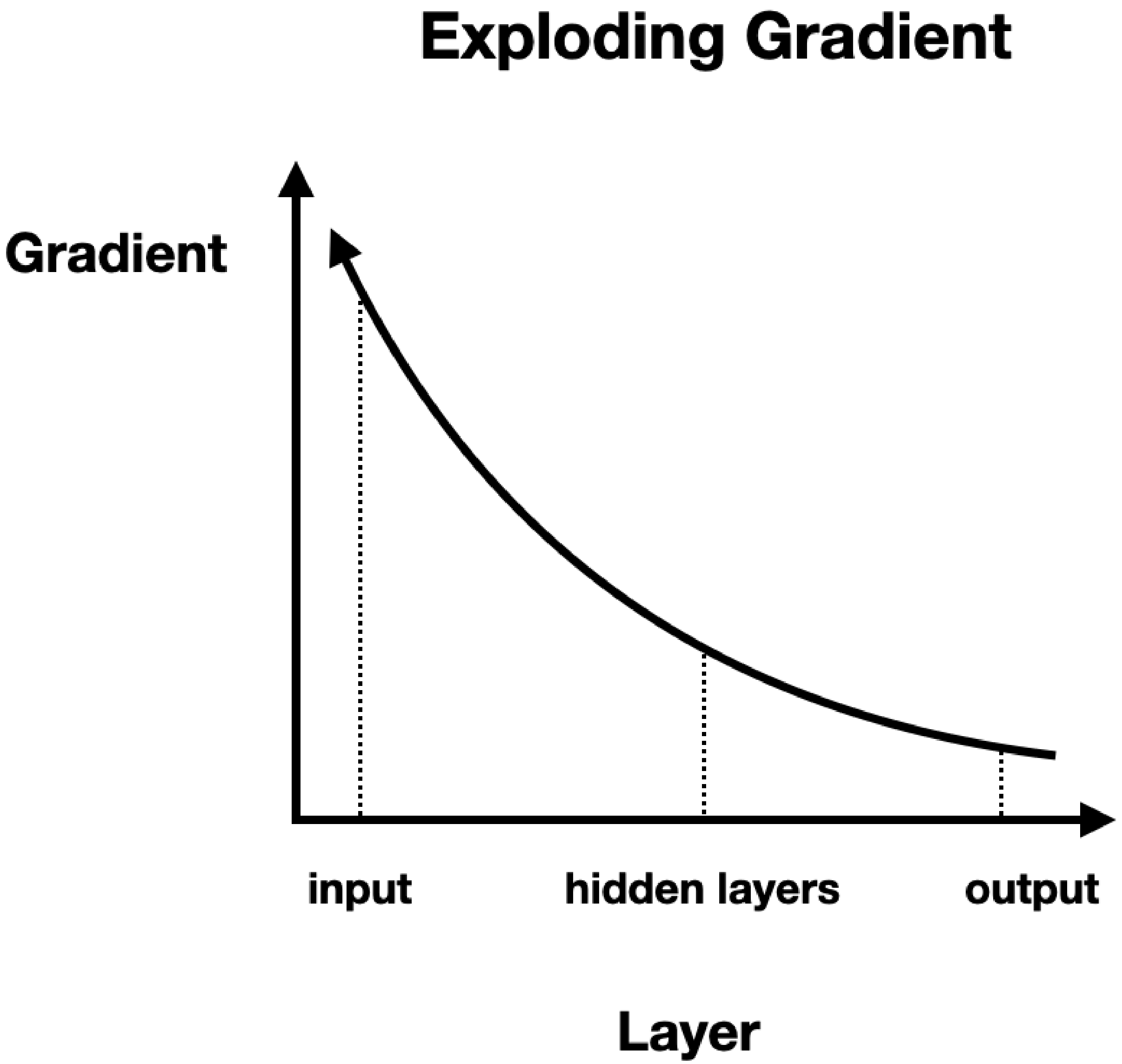
Vanishing gradients

- Gradients get smaller and smaller during backward pass
- Earlier layers get small parameter updates
- Model doesn't learn



Exploding gradients

- Gradients get bigger and bigger
- Parameter updates are too large
- Training diverges



Solution to unstable gradients

1. Proper weights initialization
2. Good activations
3. Batch normalization

Weights initialization

```
layer = nn.Linear(8, 1)
print(layer.weight)
```

```
Parameter containing:
tensor([[-0.0195,  0.0992,  0.0391,  0.0212,
         -0.3386, -0.1892, -0.3170,  0.2148]])
```

Weights initialization

Good initialization ensures:

- Variance of layer inputs = variance of layer outputs
- Variance of gradients the same before and after a layer

How to achieve this depends on the activation:

- For ReLU and similar, we can use He/Kaiming initialization

Weights initialization

```
import torch.nn.init as init

init.kaiming_uniform_(layer.weight)
print(layer.weight)
```

```
Parameter containing:
tensor([[-0.3063, -0.2410,  0.0588,  0.2664,
          0.0502, -0.0136,  0.2274,  0.0901]])
```

He / Kaiming initialization

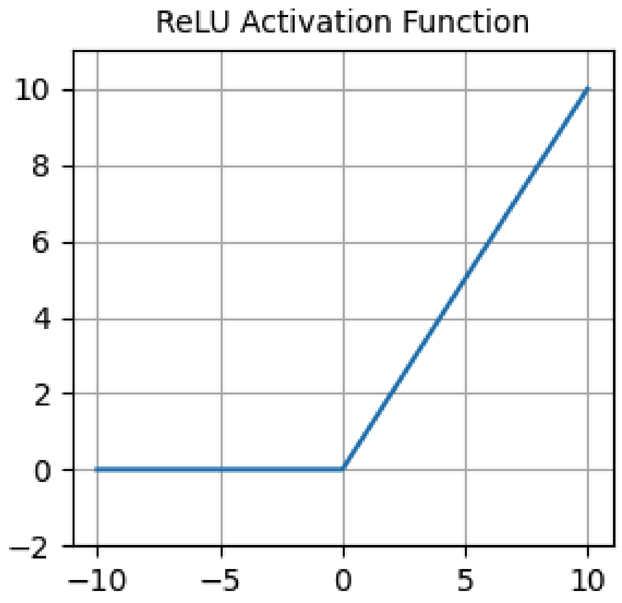
```
init.kaiming_uniform_(self.fc1.weight)
init.kaiming_uniform_(self.fc2.weight)
init.kaiming_uniform_(
  self.fc3.weight,
  nonlinearity="sigmoid",
)
```

He / Kaiming initialization

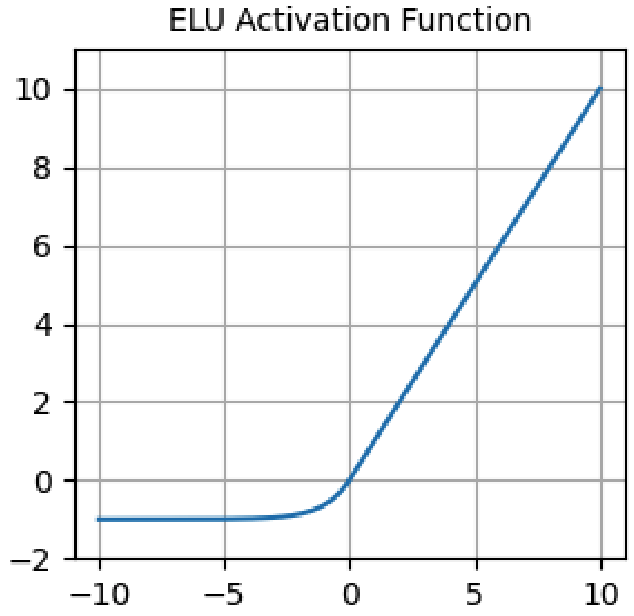
```python
import torch.nn as nn
import torch.nn.init as init

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(9, 16)
        self.fc2 = nn.Linear(16, 8)
        self.fc3 = nn.Linear(8, 1)

        init.kaiming_uniform_(self.fc1.weight)
        init.kaiming_uniform_(self.fc2.weight)
        init.kaiming_uniform_(
          self.fc3.weight,
          nonlinearity="sigmoid",
        )
```

```python
    def forward(self, x):
        x = nn.functional.relu(self.fc1(x))
        x = nn.functional.relu(self.fc2(x))
        x = nn.functional.sigmoid(self.fc3(x))
        return x
```

Activation functions



- Often used as the default activation
- `nn.functional.relu()`
- Zero for negative inputs - dying neurons



- `nn.functional.elu()`
- Non-zero gradients for negative values - helps against dying neurons
- Average output around zero - helps against vanishing gradients

Batch normalization

After a layer:

1. Normalize the layer's outputs by:
   - Subtracting the mean
   - Dividing by the standard deviation
2. Scale and shift normalized outputs using learned parameters

Model learns optimal inputs distribution for each layer:

- Faster loss decrease
- Helps against unstable gradients

Batch normalization

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(9, 16)
        self.bn1 = nn.BatchNorm1d(16)

        ...

    def forward(self, x):
        x = self.fc1(x)
        x = self.bn1(x)
        x = nn.functional.elu(x)
```

Let's practice!

INTERMEDIATE DEEP LEARNING WITH PYTORCH