Running a forward pass

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Maham Faisal Khan
Senior Data Science Content Developer

What is a forward pass?

- Input data is **passed forward** or **propagated** through a network
- Computations performed at each layer
- Outputs of each layer passed to each subsequent layer
- **Output** of final layer: "prediction"
- Used for both **training** and prediction

Some possible outputs:

- **Binary classification**
  - Single probability between 0 and 1
- **Multiclass classification**
  - Distribution of probabilities summing to 1
- **Regression** values
  - Continuous numerical predictions

Is there also a backward pass?

- **Backward pass**, or **backpropagation** is used to update weights and biases during training
- In the "training loop", we:
  1. **Propagate** data forward
  2. **Compare** outputs to true values (ground-truth)
  3. **Backpropagate** to update model weights and biases
  4. **Repeat** until weights and biases are tuned to produce useful outputs

Binary classification: forward pass

```
# Create input data of shape 5x6
input_data = torch.tensor(
        [[-0.4421,  1.5207,  2.0607, -0.3647,  0.4691,  0.0946],
        [-0.9155, -0.0475, -1.3645,  0.6336, -1.9520, -0.3398],
        [ 0.7406,  1.6763, -0.8511,  0.2432,  0.1123, -0.0633],
        [-1.6630, -0.0718, -0.1285,  0.5396, -0.0288, -0.8622],
        [-0.7413,  1.7920, -0.0883, -0.6685,  0.4745, -0.4245]])
```

```
# Create binary classification model
model = nn.Sequential(
  nn.Linear(6, 4), # First linear layer
  nn.Linear(4, 1), # Second linear layer
  nn.Sigmoid() # Sigmoid activation function
)

# Pass input data through model
output = model(input_data)
```

Binary classification: forward pass

```
print(output)
```

```
tensor([[0.5188], [0.3761], [0.5015], [0.3718], [0.4663]],
    grad_fn=<SigmoidBackward0>)
```

- **Outputs:**
  - five probabilities between zero and one
  - one value for each sample (row) in data
- **Classification:**
  - Class = 1 for first and third values: `0.5188` , `0.5015`
  - Class = 0 for second, fourth and fifth values: `0.3761` , `0.3718` , `0.4633`

Multi-class classification: forward pass

```
# Specify model has three classes
n_classes = 3

# Create multiclass classification model
model = nn.Sequential(
  nn.Linear(6, 4), # First linear layer
  nn.Linear(4, n_classes), # Second linear layer
  nn.Softmax(dim=-1) # Softmax activation
)

# Pass input data through model
output = model(input_data)
print(output.shape)
```

```
torch.Size([5, 3])
```

Multi-class classification: forward pass

```
print(output)
```

```
tensor([[0.4969, 0.3606, 0.1425],
        [0.5105, 0.3262, 0.1633],
        [0.3253, 0.3174, 0.3572],
        [0.5499, 0.3361, 0.1141],
        [0.4117, 0.3366, 0.2517]], grad_fn=<SoftmaxBackward0>)
```

- **Outputs:**
  - The output dimension is 5×3
  - Each row sums to one
  - Value with highest probability is assigned predicted label in each row
  - Row 1 = class 1 (mammal), row 2 = class 1 (mammal), row 3 = class 3 (reptile)

Regression: forward pass

```
# Create regression model
model = nn.Sequential(
  nn.Linear(6, 4), # First linear layer
  nn.Linear(4, 1) # Second linear layer
)

# Pass input data through model
output = model(input_data)

# Return output
print(output)
```

```
tensor([[0.3818],
        [0.0712],
        [0.3376],
        [0.0231],
        [0.0757]],
        grad_fn=<AddmmBackward0>)
```

Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Using loss functions to assess model predictions

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Maham Faisal Khan
Senior Data Science Content Developer

Why do we need a loss function?

Loss function:

- Gives feedback to model during training
- Takes in model prediction $\hat{y}$ and ground truth y
- Outputs a float

Why do we need a loss function?

hair	feathers	eggs	milk	airborne	aquatic	predator	toothed	backbone	breathes	venomous	fins	legs	tail	domestic	catsize	class
1	0	0	1	0	0	1	1	1	1	0	0	4	0	0	1	0

- Predicted class = 0 -> **correct** = low loss
- Predicted class = 1 -> **wrong** = high loss
- Predicted class = 2 -> **wrong** = high loss

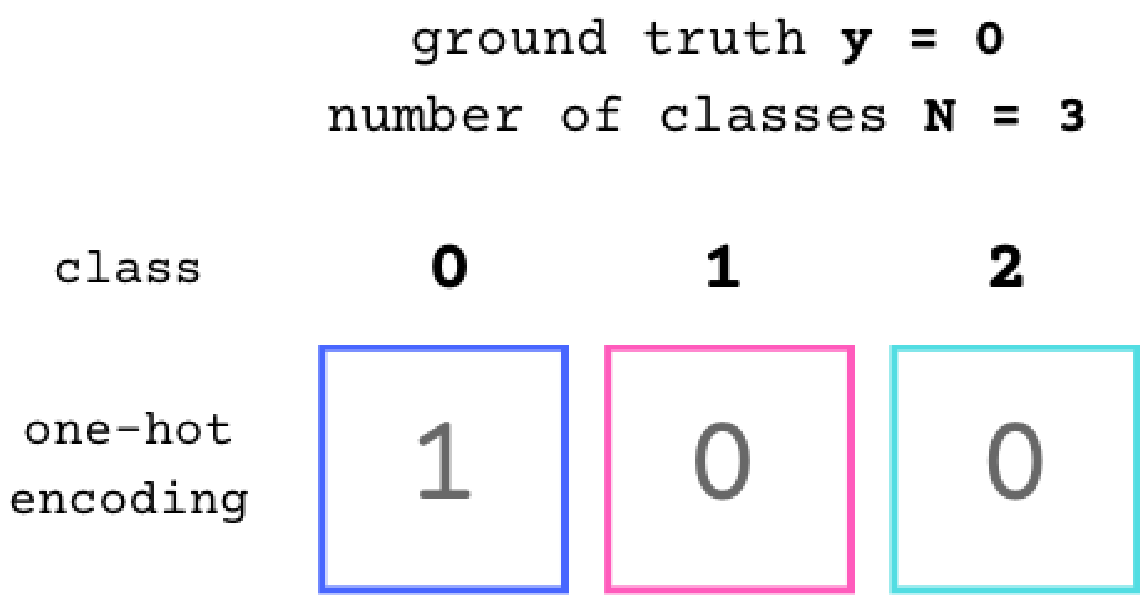
One-hot encoding concepts

- $loss = F(y, \hat{y})$
- y is a single **integer** (class label)
    - e.g. $y = 0$ when y is a mammal
- $\hat{y}$ is a **tensor** (output of softmax)
    - If N is the number of classes, e.g. N = 3
    - $\hat{y}$ is a tensor with N dimensions,
        - e.g. $\hat{y}$ = `[0.57492, 0.034961, 0.15669]`

How do we compare an integer with a tensor?

One-hot encoding concepts

Transforming true label to tensor of zeros and ones




```
one_hot_numpy = np.array([1, 0, 0])
```

Transforming labels with one-hot encoding

```
import torch.nn.functional as F

F.one_hot(torch.tensor(0), num_classes = 3)
```

```
tensor([1, 0, 0])
```

```
F.one_hot(torch.tensor(1), num_classes = 3)
```

```
tensor([0, 1, 0])
```

```
F.one_hot(torch.tensor(2), num_classes = 3)
```

```
tensor([0, 0, 1])
```

Cross entropy loss in PyTorch

```
from torch.nn import CrossEntropyLoss

scores = tensor([[-0.1211,  0.1059]])
one_hot_target = tensor([[1, 0]])

criterion = CrossEntropyLoss()
criterion(scores.double(), one_hot_target.double())
```

```
tensor(0.8131, dtype=torch.float64)
```

Bringing it all together

Loss function takes

- **scores**
  - model predictions **before** the final softmax function
- **one_hot_target**
  - one hot encoded ground truth label

and outputs

- **loss**
  - a single **float.**

Our training goal is to minimize loss.

Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Using derivatives to update model parameters

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Maham Faisal Khan
Senior Data Science Content Developer

Minimizing the loss

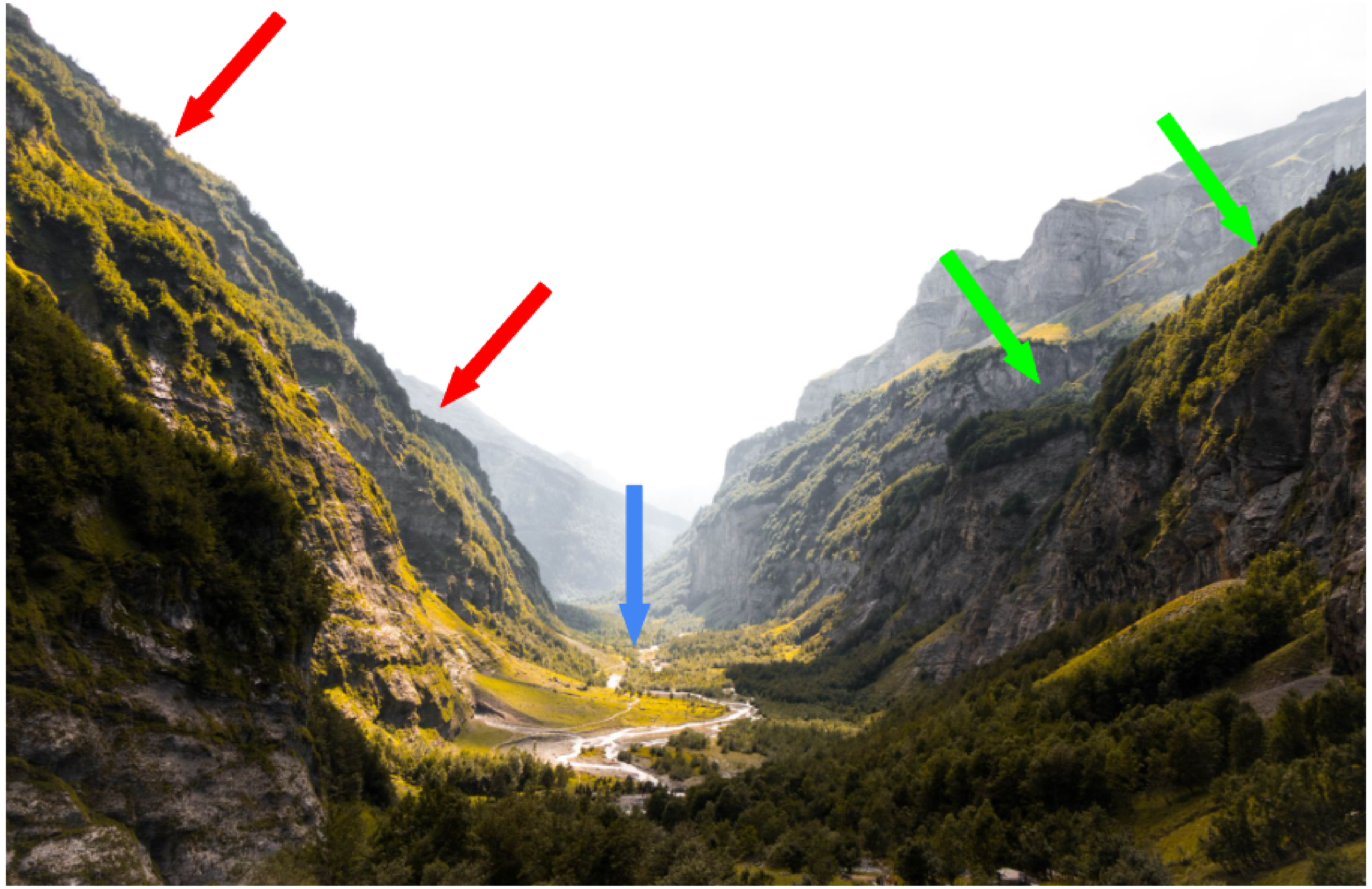
We need to minimize loss

- High loss: model prediction is wrong
- Low loss: model prediction is correct

An analogy for derivatives

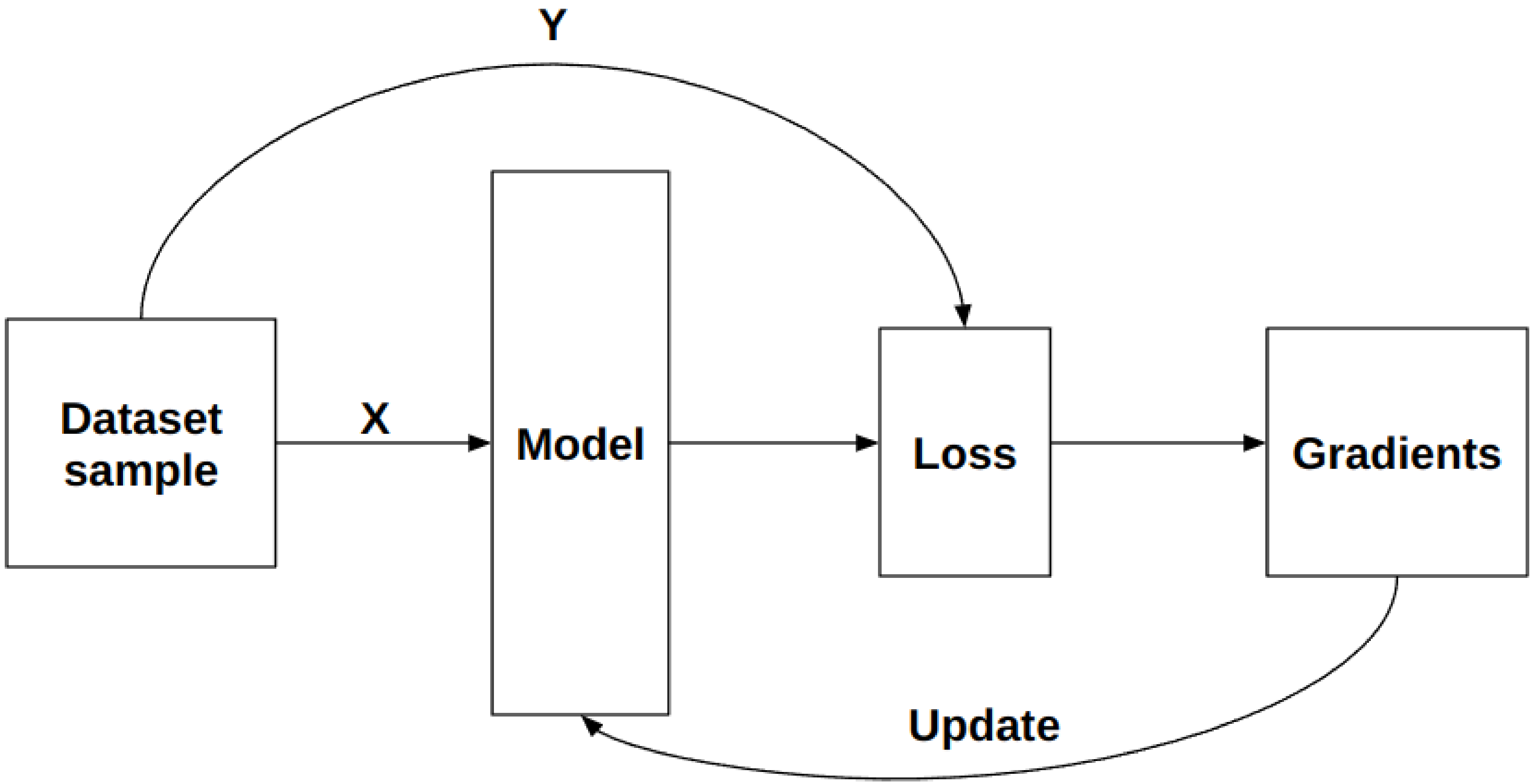
Hiking down a mountain to the valley floor:

- **steep slopes:**
  - a step makes us lose a lot of elevation = derivative is high (red arrows)
- **gentler slopes:**
  - a step makes us lose a little bit of elevation = derivative is low (green arrows)
- **valley floor:**
  - not losing elevation by taking a step = derivative is null (blue arrow)



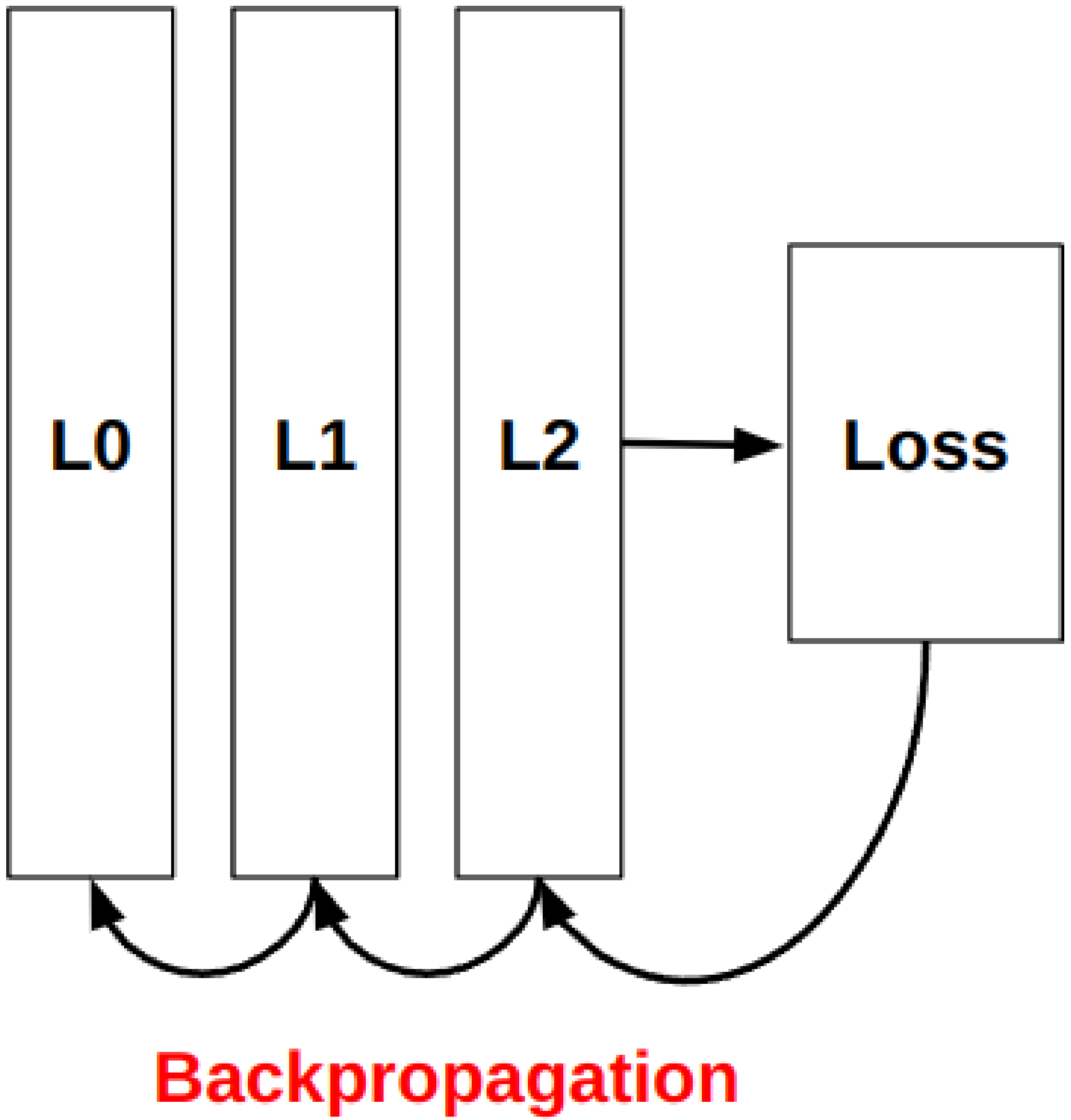
Connecting derivatives and model training

Model training: updating a model's parameters to **minimize** the loss.



Backpropagation concepts

- Consider a network made of three layers, $L0$, $L1$ and $L2$
  - we calculate local gradients for $L0$, $L1$ and $L2$ using **backpropagation**
  - we calculate loss gradients with respect to $L2$, then use $L2$ gradients to calculate $L1$ gradients, and so on



Backpropagation in PyTorch

```
# Create the model and run a forward pass
model = nn.Sequential(nn.Linear(16, 8),
                      nn.Linear(8, 4),
                      nn.Linear(4, 2))
prediction = model(sample)
```

```
# Calculate the loss and compute the gradients
criterion = CrossEntropyLoss()
loss = criterion(prediction, target)
loss.backward()
```

```
# Access each layer's gradients
model[0].weight.grad, model[0].bias.grad
model[1].weight.grad, model[1].bias.grad
model[2].weight.grad, model[2].bias.grad
```

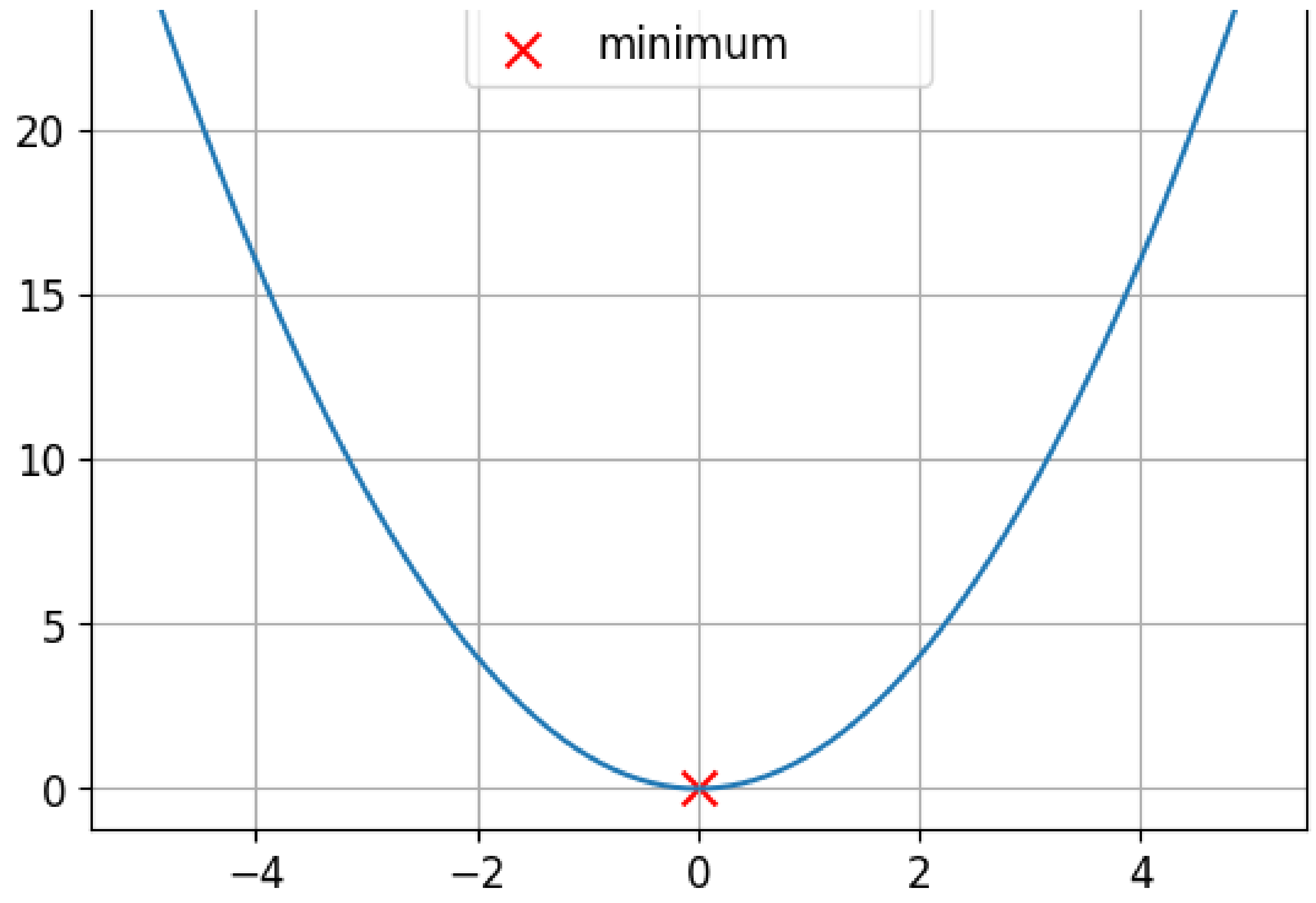
Updating model parameters

- Update the weights by subtracting local gradients scaled by the **learning rate**

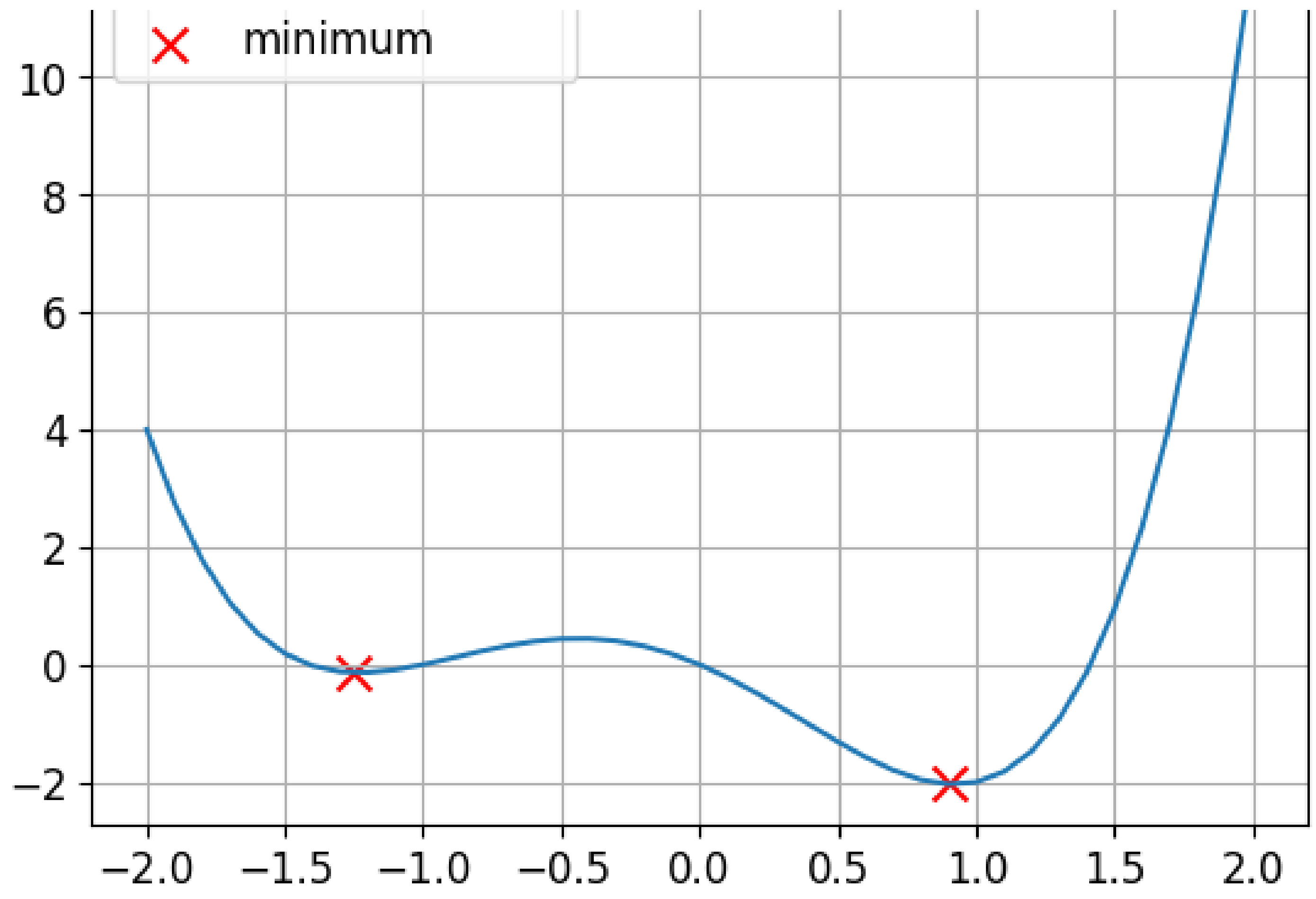
```
# Learning rate is typically small
lr = 0.001

# Update the weights
weight = model[0].weight
weight_grad = model[0].weight.grad
weight = weight - lr * weight_grad

# Update the biases
bias = model[0].bias
bias_grad = model[0].bias.grad
bias = bias - lr * bias_grad
```

Convex and non-convex functions

This is a **convex function.**



This is a **non-convex function.**



Gradient descent

- For non-convex functions, we will use an iterative process such as **gradient descent**
- In PyTorch, an **optimizer** takes care of weight updates
- The most common **optimizer** is stochastic gradient descent (SGD)

```
import torch.optim as optim


# Create the optimizer
optimizer = optim.SGD(model.parameters(), lr=0.001)
```

- Optimizer handles updating model parameters (or weights) after calculation of local gradients

```
optimizer.step()
```

Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Writing our first training loop

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Maham Faisal Khan
Senior Data Science Content Developer

Training a neural network

1. Create a model
2. Choose a loss function
3. Create a dataset
4. Define an optimizer
5. Run a training loop, where for each sample of the dataset, we repeat:
    - Calculating loss (forward pass)
    - Calculating local gradients
    - Updating model parameters

Introducing the Data Science Salary dataset

- This dataset contains salary data for data science-related jobs.
- The features are: `experience_level` , `employment_type` , `remote_ratio` and `company_size` . They were turned into categories.

```
experience_level  employment_type  remote_ratio  company_size  salary_in_usd
       0                 0              0.5            1            0.036
       1                 0              1.0            2            0.133
       2                 0              0.0            1            0.234
       1                 0              1.0            0            0.076
       2                 0              1.0            1            0.170
```

- The target is salary in US dollars; it is **not a category but a continuous quantity**
- For regression problems, we cannot use softmax or sigmoid as last activation function
- We need a different loss function than cross-entropy

Introducing the Mean Squared Error Loss

- The mean squared error loss (MSE loss) is the squared difference between the prediction and the ground truth.

```
def mean_squared_loss(prediction, target):
  return np.mean((prediction - target)**2)
```

- in PyTorch

```
criterion = nn.MSELoss()
# Prediction and target are float tensors
loss = criterion(prediction, target)
```

- This loss is used for regression problems (e.g., when trying to fit a linear regression model).

Before the training loop

```
# Create the dataset and the dataloader
dataset = TensorDataset(torch.tensor(features).float(), torch.tensor(target).float())
dataloader = DataLoader(dataset, batch_size=4, shuffle=True)

# Create the model
model = nn.Sequential(nn.Linear(4, 2),
                      nn.Linear(2, 1))

# Create the loss and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.001)
```

The training loop

```
# Loop through the dataset multiple times
for epoch in range(num_epochs):
    for data in dataloader:
        # Set the gradients to zero
        optimizer.zero_grad()
        # Get feature and target from the data loader
        feature, target = data
        # Run a forward pass
        pred = model(feature)
        # Compute loss and gradients
        loss = criterion(pred, target)
        loss.backward()
        # Update the parameters
        optimizer.step()
```

Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH