AI 기반 영상 데이터 분석 실습
- Day 4 -

Course overview

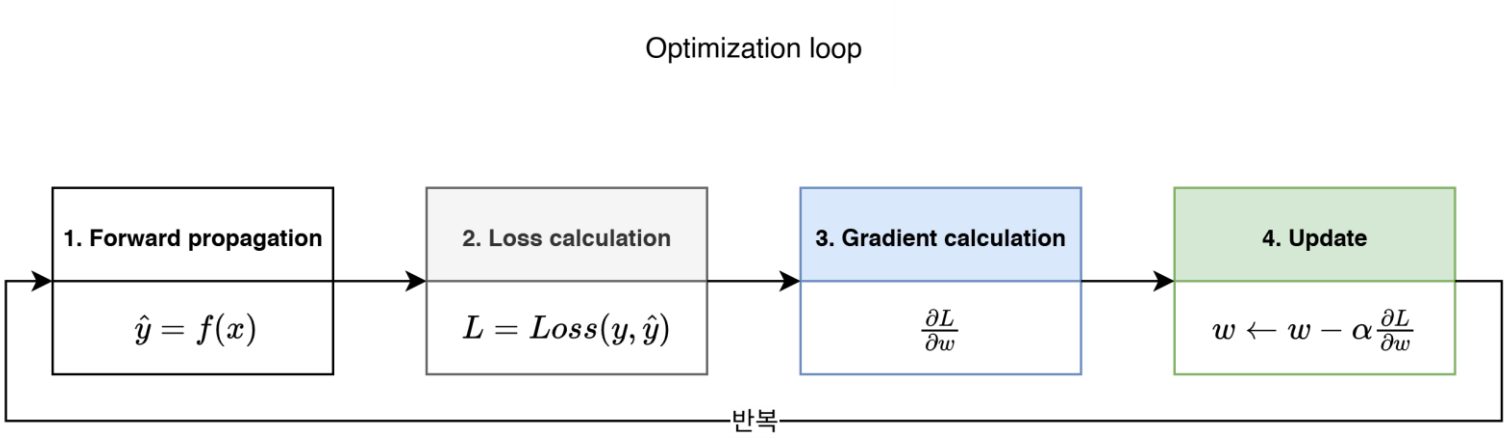
Course overview

Day	Morning session (이론)	Afternoon Lab session (실습)
Jan. 19 (Mon)	• 인공지능 개요 • Vision task 소개 • MNIST Classification review	• GitHub 활용법 • 주제 선정 • 데이터 수집
Jan. 20 (Tue)	• 영상처리 기초 • 이미지 전처리 기법	• 전처리, 증강, 시각화 • Dataset split • Data set class & Data loading
Jan. 21 (Wed)	• Fundamentals on CNN • Basic architectures	• CNN architecture 구현 (Resnet)
Jan. 22 (Thu)	• Weight update (Gradient descent, Optimizers) • Loss monitoring	• Torch model • 아키텍처 선택 및 구현 (Resnet + @) • 모델 학습 및 하이퍼파라미터 튜닝
Jan. 26 (Mon)	• Model evaluation • Grad-CAM	• 모델 평가 • Grad-CAM 실습
Jan. 27 (Tue)	• Github에 프로젝트 업로드 • 프로젝트 발표	

Optimization

Optimization

Optimization



Loss function

Loss function

$$f(\boldsymbol{x}_i, \boldsymbol{W})$$

- For learning, we need to adjust the weight matrix $\boldsymbol{W}$ so that the predicted class scores align with the ground truth labels in the training data.
- To achieve this, we must quantify how well the model's predictions match the true labels.
- Thus, we measure our "unhappiness" with the predictions using a **Loss function** (also known as a cost function or objective function).

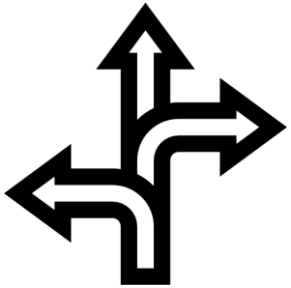
Loss function

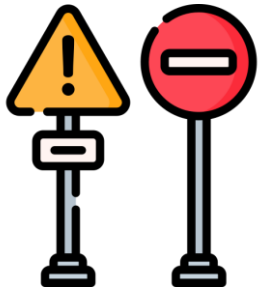
Loss function

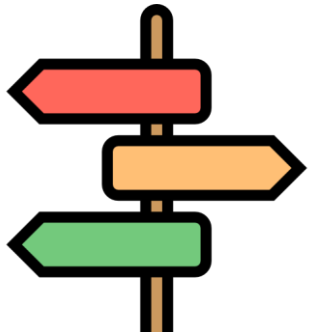
- Measure of loss incurred in taking any of the available decision or action
  - I.e., a mathematical function that measures the degree of difference between the actual and predicted outputs of a model
  - Also called as cost function – overall measure

Role of loss function

- Guide the optimization (learning) process by providing a measure of how well the model is performing







Loss function

Loss function

- In learning process, our goal is to **minimize the total loss** 
  - The loss will be high if we're doing a poor job of classifying the training data, and it will be low if we're doing well
  - Some authors consider a utility function → value to maximize
- It is our choice which loss functions to use based on the purpose
  → important role of AI engineers

Loss function

General classification loss

- Cross Entropy

  - Information: $I(x) = -\log P(x)$
  - Entropy: $H(x) = E_{x\sim P}[I(x)] = -E_{x\sim P}[\log P(x)] = -\sum P(x)\log P(x)$
  - KL-Divergence: $D_{KL}(P||Q) = E_{x\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = E_{x\sim P}[\log P(x) - \log Q(x)]$
    - A.k.a. Relative Entropy, Difference between two distribution
  - **Cross Entropy**:

$$H(P,Q) = H(P) + D_{KL}(P||Q)$$
$$= -E_{x\sim P}[\log Q(x)] = -\sum P(x)\log Q(x)$$

    - The average amount of information we are spending to estimate true distribution P using the distribution Q
      → higher cross-entropy = worse prediction

Loss function

Binary Cross Entropy (BCE)

- Common choices for <u>binary classification</u>
- Recap> Cross entropy: $-\sum P(x)\log Q(x)$
- **Binary cross entropy loss**

$$L_{BCE} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)]$$

  - For i^{th} data whose label is 1, $P(x) = y_i = 1, Q(x) = \hat{y}_i$
  - For i^{th} data whose label is 0, $P(x) = y_i = 0, Q(x) = 1 - \hat{y}_i$

Loss function

Categorical Cross Entropy

- Multi-class label – category (i.e., one-hot encoding)

Data No.	Category
1	2
2	3
3	1
4	2

One-hot encoded label		
0	1	0
0	0	1
1	0	0
0	1	0

Loss function

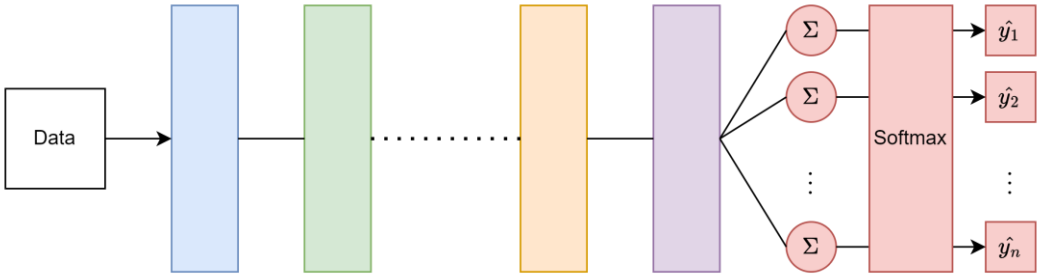
Categorical Cross Entropy

- Multi-class label – category (i.e., one-hot encoding)

Data No.	Category
1	2
2	3
3	1
4	2

One-hot encoded label		
0	1	0
0	0	1
1	0	0
0	1	0

- Multi-class classification model → predicting probabilities for each category



Loss function

Categorical Cross Entropy

- Common choices for multi-class classification: categorical cross entropy
  - Comparison One-hot encoded label vs Predicted probabilities vector
    - Example

Data no.	y			$\hat{y}$		
1	1	0	0	0.8	0.1	0.1
2	0	0	1	0.1	0.0	0.9
3	0	1	0	0.2	0.6	0.2

Loss function

Categorical Cross Entropy

- Common choices for multi-class classification: categorical cross entropy
  - Recap> Cross entropy: $-\sum P(x) \log Q(x)$

Data no.	y			$\hat{y}$		
1	1	0	0	0.8	0.1	0.1
2	0	0	1	0.1	0.0	0.9
3	0	1	0	0.2	0.6	0.2

Ground truth probability for each category = $P(x)$

Predicted probability for each category = $Q(x)$

Loss function

Categorical Cross Entropy

- Common choices for multi-class classification: categorical cross entropy
  - Recap> Cross entropy: $-\sum P(x)\log Q(x)$

Data no.	y			$\hat{y}$		
1	1	0	0	0.8	0.1	0.1
2	0	0	1	0.1	0.0	0.9
3	0	1	0	0.2	0.6	0.2

Cross entropy for data 1

$$CE = -\sum P(x)\log Q(x) = -(1\cdot\log 0.8 + 0\cdot\log 0.1 + 0\cdot\log 0.1)$$

$$= -\log 0.8$$

Loss function

Categorical Cross Entropy

- Common choices for multi-class classification: categorical cross entropy
  - Recap> Cross entropy: $-\sum P(x)\log Q(x)$

Data no.	y			$\hat{y}$		
1	1	~~0~~	~~0~~	0.8	~~0.1~~	~~0.1~~
2	0	0	1	0.1	0.0	0.9
3	0	1	0	0.2	0.6	0.2

Cross entropy for data 1

$$CE = -\sum P(x)\log Q(x) = -(1\cdot\log 0.8 + 0\cdot\log 0.1 + 0\cdot\log 0.1)$$

$$= -\log 0.8$$

We can recognize that we don't need to care about $\hat{y}$ for zeros

Loss function

Categorical Cross Entropy

- Common choices for multi-class classification: categorical cross entropy
  - Recap> Cross entropy: $-\sum P(x)\log Q(x)$

Data no.	y			$\hat{y}$		
1	1	0	0	0.8	0.1	0.1
2	0	0	1	0.1	0.0	0.9
3	0	1	0	0.2	0.6	0.2

Cross entropy loss for all data

$$L_{CE} = -\frac{1}{n}\sum\sum P(x)\log Q(x) = -\frac{1}{3}(\log 0.8 + \log 0.9 + \log 0.6)$$

→ Categorical cross entropy

Loss function

Categorical Cross Entropy

- In formulated form,

$$L_{CE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_{i,c}\log\left(\hat{y}_{i,c}\right)$$

  - n – number of samples, C – number of classes
  - $y_{i,c}$ – binary indicator (0 or 1) if class label c is the correct classification for sample i
  - $\hat{y}_{i,c}$ - predicted probability for the class c for sample i

Optimization

Optimization

- The goal of optimization is to find $\boldsymbol{W}$ that minimizes the loss function
  - Strategy #1: Random Search (Bad idea)
  - Strategy #2: Random Local Search
  -  Strategy #3: Following Gradient

Optimization

Strategy #1: A first very bad idea solution: Random search

- To simply try out many different random weights and keep track of what works best
- Pseudocode

Algorithm 1 Random Search for Minimizing Loss

```
bestloss ← ∞
for num ← 1 to 1000 do
    W ← randomly initialize W ~ N(0, 1)
    loss ← L(X_train, Y_train, W)
    if loss < bestloss then
        bestloss ← loss
        bestW ← W
    end if
    print "in attempt num, the loss was loss, best bestloss"
end for
```

Optimization

Strategy #1: A first very bad idea solution: Random search

- To simply try out many different random weights and keep track of what works best
- **Core idea: iterative refinement**
  - Our strategy will be to start with random weights and iteratively refine them over time to get lower loss

Optimization

Strategy #2: Random Local Search

- To apply random perturbation $\delta \boldsymbol{W}$ to the current weight $\boldsymbol{W}$, and if the loss at the perturbed $\boldsymbol{W} + \delta \boldsymbol{W}$ is lower than before, perform update.

- Pseudocode

Algorithm 2 Random Local Search

```
W ← randomly initialize 𝒩(0, 1)
bestloss ← ∞
for i ← 1 to 1000 do
    step_size ← 0.0001
    W_try ← W + 𝒩(0, step_size^2)
    loss ← L(X_train, Y_train, W_try)
    if loss < bestloss then
        W ← W_try
        bestloss ← loss
    end if
    print "iter i, loss is bestloss"
end for
```

- Much better and stable than Strategy #1

Optimization

Strategy #3: Following the Gradient

- In the previous section we tried to find a direction in the weight-space that would improve our weight vector by updating current weight
  - We **can compute the best direction** along which we should change our weight vector that is mathematically guaranteed to be the direction of the steepest descent
- This direction will be related to the **gradient** of the loss function

$$\nabla L = \left[\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_n}\right]$$

  - A gradient of a function is a vector of partial derivatives.

Optimization

Computing the gradient

- Two ways to compute gradient
  - **Numerical gradient**
    - Approximation (Slow) but easy
  - **Analytic gradient**
    - Using calculus

Optimization

Numerical gradient

- Computing the gradient numerically with finite differences

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- Setting h as small value allow us estimate derivatives

Algorithm 3 Numerical Gradient Computation

Input: differentiable function f, input vector $\mathbf{x}$
$h \leftarrow 10^{-5}$ ▷ Step size
$f_x \leftarrow f(\mathbf{x})$ ▷ Evaluate function at original point
$\nabla \leftarrow$ zero vector with same shape as $\mathbf{x}$
for each index i in $\mathbf{x}$ **do**
  $x_{\text{old}} \leftarrow \mathbf{x}[i]$
  $\mathbf{x}[i] \leftarrow x_{\text{old}} + h$ ▷ Perturb the i-th dimension
  $f_{x+h} \leftarrow f(\mathbf{x})$ ▷ Evaluate function at perturbed point
  $\nabla[i] \leftarrow \frac{f_{x+h} - f_x}{h}$ ▷ Approximate $\frac{\partial f}{\partial x_i}$
  $\mathbf{x}[i] \leftarrow x_{\text{old}}$ ▷ Restore original value
end for
return ∇

The updates to all components occur simultaneously

Optimization

Numerical gradient

- After computing the gradients of the loss function, we update the weights in the **opposite direction of the gradient**, since this is the direction that reduces the loss

$$w_{new} = w_{old} - \alpha \nabla L$$

  - The hyperparameter α controls how big each update step is—this is called the **learning rate**. We'll talk more about it later

Optimization

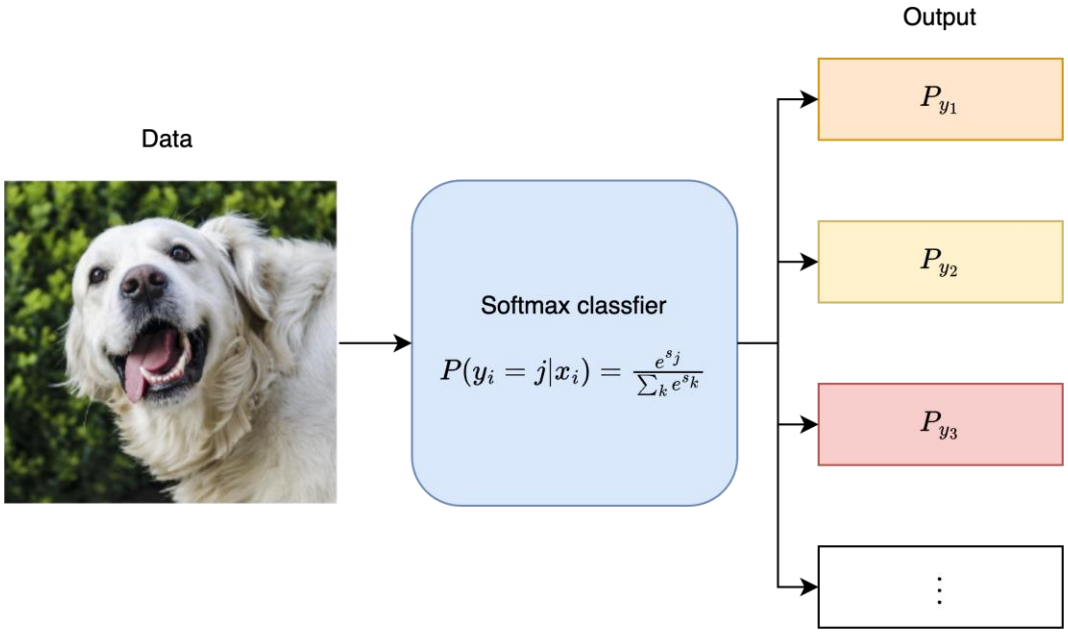
Analytic gradient

- Using calculus
  - Which allows us to derive a direct formula for the gradient (no approximations)
  - Very fast to compute

Softmax classifier

Example: Softmax classifier

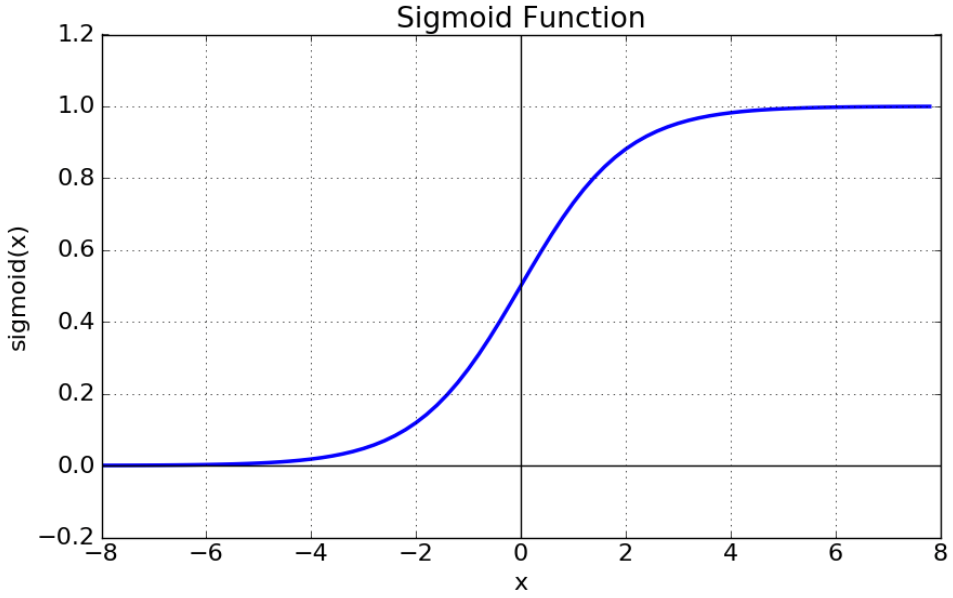
- Another popular choice for linear classifier
- Generalized version of logistic regression



Softmax classifier

Logistic regression

- Statistical method and machine learning algorithm used for binary classification problems
  - Note: Despite its name, logistic regression is used primarily for classification tasks rather than regression tasks
- Core of logistic regression: **Sigmoid function**



$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- $0 \leq \sigma(x) \leq 1$
- $\sigma(-x) = 1 - \sigma(x)$

- Map feature → 0~1
- Output of sigmoid can be treated as probability
- $x = 0 (\sigma(x) = 0.5)$ can be (fair) decision boundary

Softmax classifier

Logistic regression

- Model equation

$$\hat{y} = \sigma_{\boldsymbol{w}}(\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{w}\cdot\boldsymbol{x}}} = \frac{1}{1 + e^{-\sum w_j x_j}}$$

  - It contains linear mapping function $\boldsymbol{w} \cdot \boldsymbol{x}$

Softmax classifier

Softmax classifier

- Linear mapping function

$$f(\boldsymbol{x}_i; \boldsymbol{W}) = \boldsymbol{W}\boldsymbol{x}_i$$

  - Softmax classifier → Treat $f(\boldsymbol{x}_i; \boldsymbol{W})$ as unnormalized log probability
  - Utilize softmax function to normalize this output
    - Softmax function

$$\sigma(\boldsymbol{z})_i = \frac{e^{\boldsymbol{z}_i}}{\sum_{j=1}^{K} e^{\boldsymbol{z}_j}}$$

Optimization

Analytic gradient – Softmax classifier example

- Softmax classifier

$$Z = XW \rightarrow z_{i,c} = x_i^T w_c$$

$$\hat{y}_{i,c} = \frac{e^{z_{i,c}}}{\sum_{k=1}^{C} e^{z_{i,k}}}$$

- Cross-entropy loss

$$L_{CE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_{i,c}\, log\left(\hat{y}_{i,c}\right)$$

- Goal

$$\frac{\partial L}{\partial W} \leftarrow \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial W}$$

Optimization

Analytic gradient – Softmax classifier example

- For a fixed i,

$$L_i = -\sum_{c=1}^{C} y_{i,c}\, log(\hat{y}_{i,c})$$

- At first,

$$log(\hat{y}_{i,c}) = \log\left(\frac{e^{z_{i,c}}}{\sum_{k=1}^{C} e^{z_{i,k}}}\right) = z_{i,c} - \log\left(\sum_{k=1}^{C} e^{z_{i,k}}\right)$$

- So,

$$L_i = -\sum_{c=1}^{C} y_{i,c}\, log(\hat{y}_{i,c}) = -\sum_{c=1}^{C} y_{i,c} z_{i,c} + \sum_{c=1}^{C} y_{i,c} \log\left(\sum_{k=1}^{C} e^{z_{i,k}}\right)$$

Optimization

Analytic gradient – Softmax classifier example

- Because the second term does not depend on c, and $\sum_{c=1}^{C} y_{i,c} = 1$,

$$L_i = -\sum_{c=1}^{C} y_{i,c}\, log(\hat{y}_{i,c}) = -\sum_{c=1}^{C} y_{i,c} z_{i,c} + \sum_{c=1}^{C} y_{i,c} \log\left(\sum_{k=1}^{C} e^{z_{i,k}}\right)$$

$$= -\sum_{c=1}^{C} y_{i,c} z_{i,c} + \log\left(\sum_{k=1}^{C} e^{z_{i,k}}\right)$$

Optimization

Analytic gradient – Softmax classifier example

- Now, differentiate w.r.t. $z_{i,j}$

$$L_i = -\sum_{c=1}^{C} y_{i,c} z_{i,c} + \log\left(\sum_{k=1}^{C} e^{z_{i,k}}\right)$$

$$\rightarrow \frac{\partial L_i}{\partial z_{i,j}} = -y_{i,j} + \frac{e^{z_{i,j}}}{\sum_{k=1}^{C} e^{z_{i,k}}} = -y_{i,j} + \hat{y}_{i,j}$$

- For entire data,

$$\frac{\partial L}{\partial z_{i,j}} = \frac{1}{n}\left(\hat{y}_{i,j} - y_{i,j}\right)$$

- In matrix form,

$$\frac{\partial L}{\partial Z} = \frac{1}{n}\left(\hat{Y} - Y\right)$$

Optimization

Analytic gradient – Softmax classifier example

- Now, let's derive $\frac{\partial Z}{\partial W}$

$$Z = XW$$

$$\rightarrow \frac{\partial Z}{\partial W} = X$$

- Finally,

$$\frac{\partial L}{\partial Z} = \frac{1}{n}\left(\hat{Y} - Y\right), \qquad \frac{\partial Z}{\partial W} = X$$

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial W} = \frac{1}{n} X^T \left(\hat{Y} - Y\right)$$

Optimization

Analytic gradient – Softmax classifier example

- Now update,

$$W_{new} \leftarrow W_{old} - \alpha \nabla_W L$$

Gradient Descent

Gradient descent

- The procedure of repeatedly evaluating the gradient and the performing a parameter update (as we see slides before) is call **Gradient Descent**

Algorithm 4 Gradient Descent

Input: Objective function $L(W)$, learning rate α, number of iterations T
Initialize weights W
for $t = 1$ to T **do**
    Compute gradient: $\nabla L(W)$
    Update weights: $W \leftarrow W - \alpha \cdot \nabla L(W)$
end for
Output: Optimized weights W

- This simple loop is at the core of all Neural Network libraries.

Softmax Classifier

Python implementation

- Utilizing Softmax classifier

```
import torch
import torchvision
import numpy as np
import matplotlib.pyplot as plt

class SoftmaxClassifier():
    def __init__(self):
        self.w = np.random.normal(0,1,(10, 28*28+1)) # bias trixk

    def forward(self, x_batch):
        return x_batch @ self.w.T

    def predict(self, x_batch):
        return np.argmax(self.forward(x_batch), axis=1)

    def __call__(self, x_batch):
        return self.forward(x_batch)
```

```
# Load MNIST data
train_dataset = torchvision.datasets.MNIST('./', train=True,
download=True)
test_dataset = torchvision.datasets.MNIST('./', train=False,
download=True)

x_train = train_dataset.data.numpy()
y_train = train_dataset.targets.numpy()
y_train_onehot = np.zeros((len(y_train), 10))
y_train_onehot[np.arange(len(y_train)), y_train] = 1

x_test = test_dataset.data.numpy()
y_test = test_dataset.targets.numpy()
y_test_onehot = np.zeros((len(y_test), 10))
y_test_onehot[np.arange(len(y_test)), y_test] = 1

# Normalize image to be 0~1
x_train = x_train/255.0
x_test = x_test/255.0

# Flatten image
x_train = x_train.reshape(-1, 28*28)
x_test = x_test.reshape(-1, 28*28)
```

Softmax Classifier

Python implementation

- Utilizing Softmax classifier

```
# Build model
Clf = SoftmaxClassifier()
alpha = 0.0005 # learning rate
Epochs = 100
batch_size = 64

for epoch in range(Epochs):
    for i in range(0, len(x_train), batch_size):
        batch = x_train[i:i+batch_size]
        batch = np.concat([batch, np.ones(shape=(len(batch),1))], axis=1)

        batch_label = y_train_onehot[i:i+batch_size]
        pred = Clf(batch)

        Clf.w = Clf.w - alpha *  (pred - batch_label).T @ batch # calculate gradient and update

    # Test
    pred_test = Clf(np.concat([x_test, np.ones(shape=(len(x_test),1))], axis=1))
    pred_label_test = np.argmax(pred_test, axis=1)
    print(f"epoch: {epoch}, val_acc: {np.average(pred_label_test==y_test):.4}")
```

Softmax Classifier

Python implementation

- Utilizing Softmax classifier + Torch optimizer

```
import torch
import torch.nn as nn
import torchvision
from torch.utils.data import DataLoader, TensorDataset

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load MNIST data
train_dataset = torchvision.datasets.MNIST('./', train=True, download=True)
test_dataset = torchvision.datasets.MNIST('./', train=False, download=True)

x_train = train_dataset.data
y_train = train_dataset.targets

x_test = test_dataset.data
y_test = test_dataset.targets

# Normalize image to be 0~1
x_train = x_train/255.0
x_test = x_test/255.0

# Flatten image
x_train = x_train.reshape(-1, 28*28)
x_test = x_test.reshape(-1, 28*28)

# DataLoader
train_loader = DataLoader(TensorDataset(x_train, y_train), batch_size=128)#, shuffle=True)
test_loader  = DataLoader(TensorDataset(x_test, y_test), batch_size=128)#, shuffle=False)
```

Softmax Classifier

Python implementation

- Utilizing Softmax classifier + Torch optimizer

```
class SoftmaxLinear(nn.Module):
    def __init__(self, input_dim=28*28, num_classes=10):
        super().__init__()
        self.fc = nn.Linear(input_dim, num_classes)

    def forward(self, x):
        logits = self.fc(x)
        return logits

model = SoftmaxLinear().to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

for epoch in range(0, 100):
    model.train()
    total_loss = 0.0

    for x, y in train_loader:
        x, y = x.to(device), y.to(device)

        optimizer.zero_grad()
        logits = model(x)
        loss = criterion(logits, y)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()


    print(f"Epoch [{epoch+1}] Loss: {total_loss/len(train_loader):.4f}")
```

Backpropagation

Backpropagation

Backpropagation

- A way of computing gradients of expressions through recursive application of **chain rule**
  - **Essential for designing, training and debugging neural networks**
- Gradient of model

$$\nabla f(x, W, b)$$

  - In neural networks:
    - f: loss function (e.g., SVM's hinge loss)
    - W, b: weights and biases, x: training data
    - We treat:
      - Training data as fixed
      - Model parameters (W, b) as variables to optimize

Backpropagation

Interpretation of the gradient

- Let's consider simple function $f(x, y) = xy$
  - Partial derivatives

$$f(x, y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y, \quad \frac{\partial f}{\partial y} = y$$

    - Represent how rapidly a function changes with respect to a variable in an infinitesimally small neighborhood **around a specific point** (x, y).
    - $\frac{\partial}{\partial x}$ is operator (not a division)
    - The gradient ∇f is the vector of partial derivatives

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right] = [y, x]$$

Backpropagation

Interpretation of the gradient

- For other simple functions,
  - Adding function

$$f(x, y) = x + y \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1, \quad \frac{\partial f}{\partial y} = 1$$

    - Increasing either x, y would increase the output of f
  - Max function

$$f(x, y) = \max(x, y) \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1\ (x \geq y), \quad \frac{\partial f}{\partial y} = 1\ (y \geq x)$$

    - If $x \geq y$, the function is not sensitive to the setting of y
    - If $y \geq x$, the function is not sensitive to the setting of x

Backpropagation

Compound expressions with chain rule

- Now, let's start to consider more complicate expressions that involve multiple composed functions, for example,

$$f(x, y, z) = (x + y)z$$

  - We can break down this into two expressions

$$q = x + y$$

$$f = qz$$

  - We can also how to compute the derivates of both expressions,

$$\frac{\partial q}{\partial x} = 1, \qquad \frac{\partial q}{\partial y} = 1, \qquad \frac{\partial f}{\partial z} = q, \qquad \frac{\partial f}{\partial q} = z$$

Backpropagation

Compound expressions with chain rule

- Ultimately, we are interested in the gradient of f with respect to its input x, y, z

$$f(x, y, z) = (x + y)z$$

$$\frac{\partial q}{\partial x} = 1, \qquad \frac{\partial q}{\partial y} = 1, \qquad \frac{\partial f}{\partial z} = q, \qquad \frac{\partial f}{\partial q} = z$$

  - The **chain rule** tells us that the correct way to "chain" these gradient expressions together is through multiplication

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = z \times 1 = z$$
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = z \times 1 = z$$
$$\frac{\partial f}{\partial z} = q = x + y$$

Backpropagation

Compound expressions with chain rule

- As a result,

$$f(x, y, z) = (x + y)z$$

$$\nabla f = [z, z, x + y]$$

  - For point $(x, y, z) = (-2{,}5, -4)$

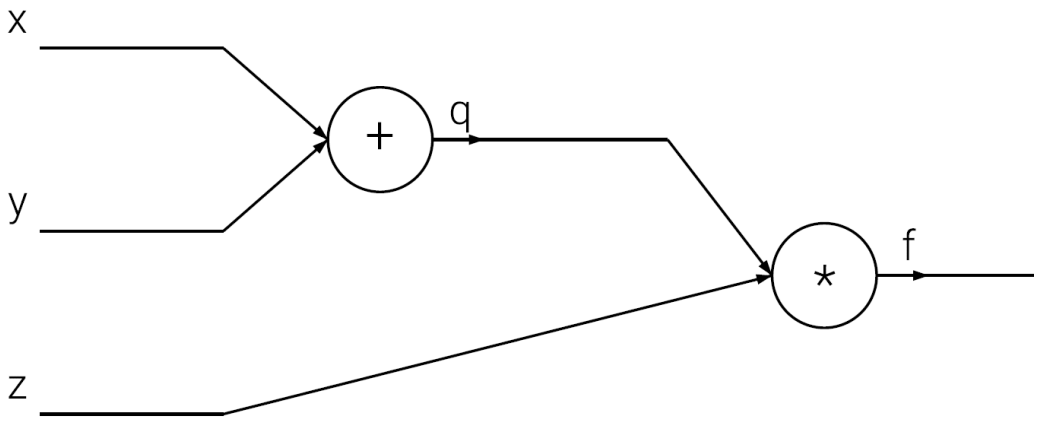
$$f(-2{,}5, -4) = (-2 + 5) \cdot -4 = -12$$

$$\nabla f = [-4, -4{,}3]$$

Backpropagation

Computational graph

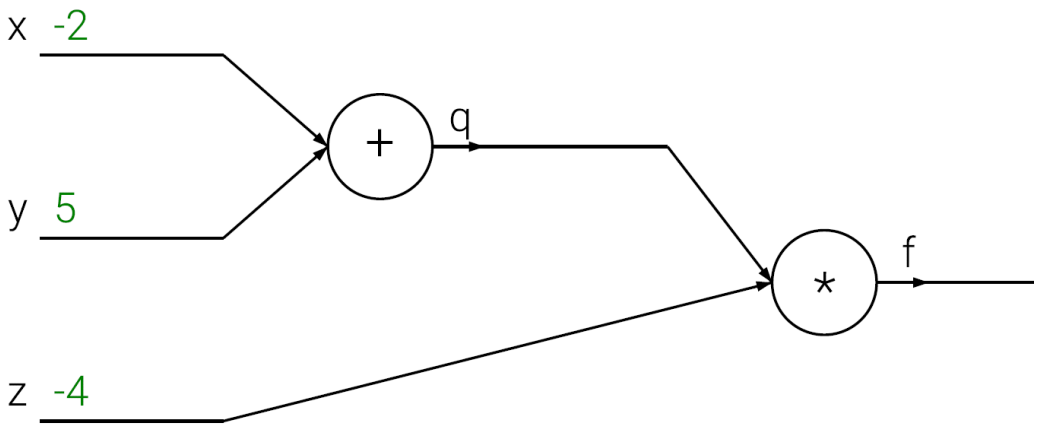
- This computation can be visualized with a circuit diagram:



Backpropagation

Computational graph

- This computation can be visualized with a circuit diagram:

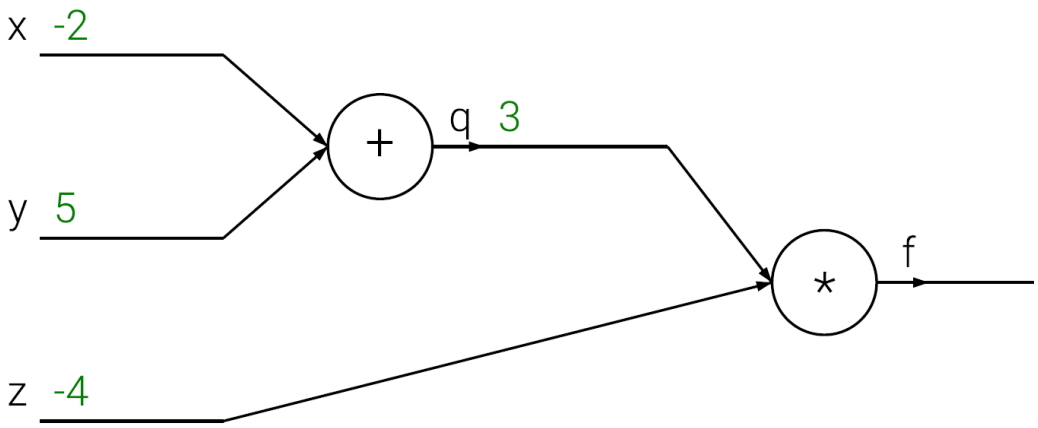


Forward pass

Backpropagation

Computational graph

- This computation can be visualized with a circuit diagram:

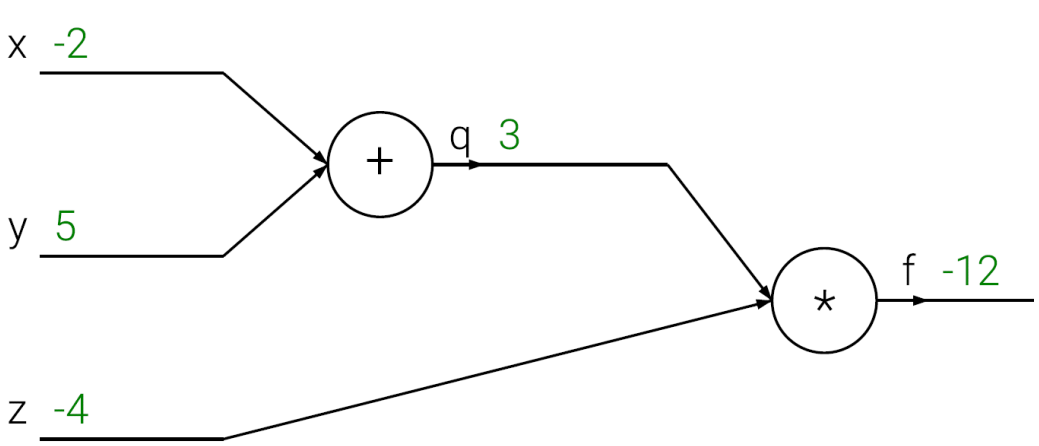


Forward pass

Backpropagation

Computational graph

- This computation can be visualized with a circuit diagram:

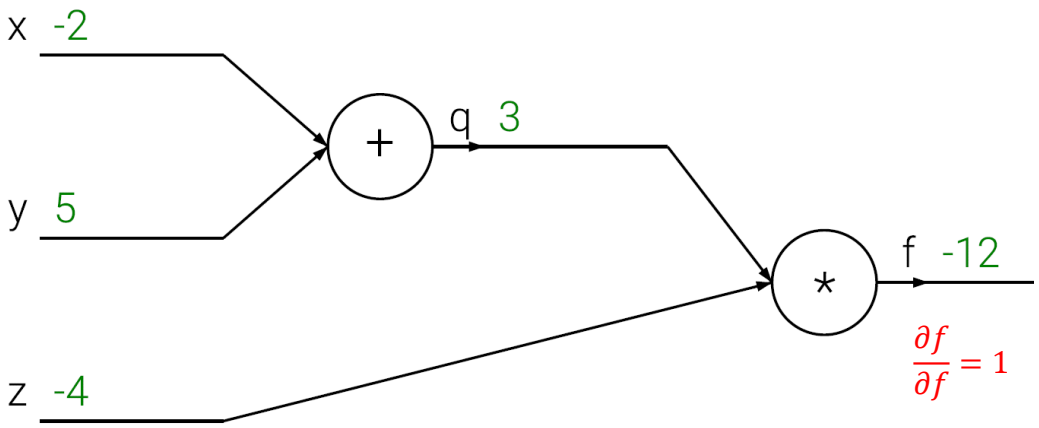


Forward pass

Backpropagation

Computational graph

- This computation can be visualized with a circuit diagram:

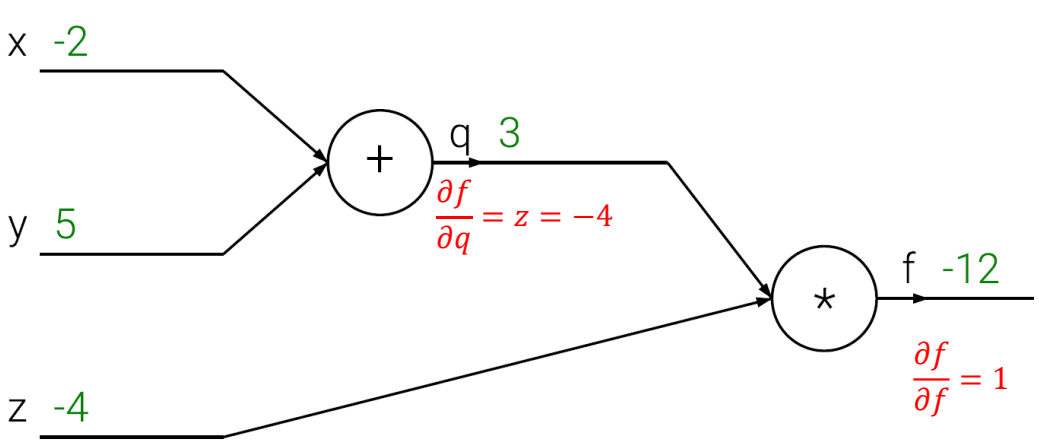


Backward pass

Backpropagation

Computational graph

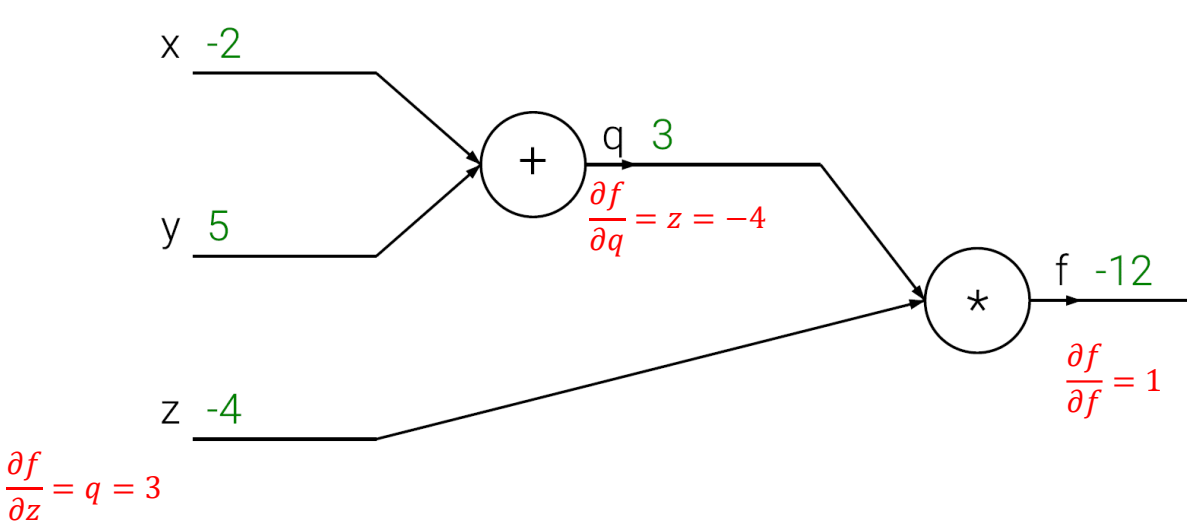
- This computation can be visualized with a circuit diagram:



Backpropagation

Computational graph

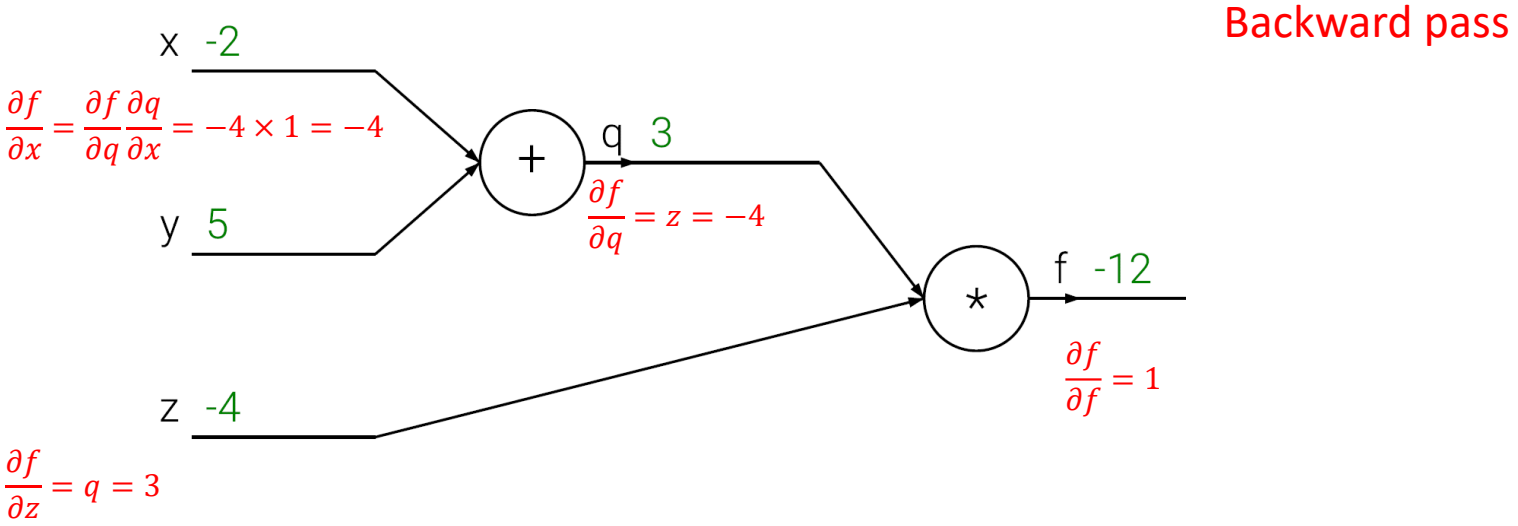
- This computation can be visualized with a circuit diagram:



Backpropagation

Computational graph

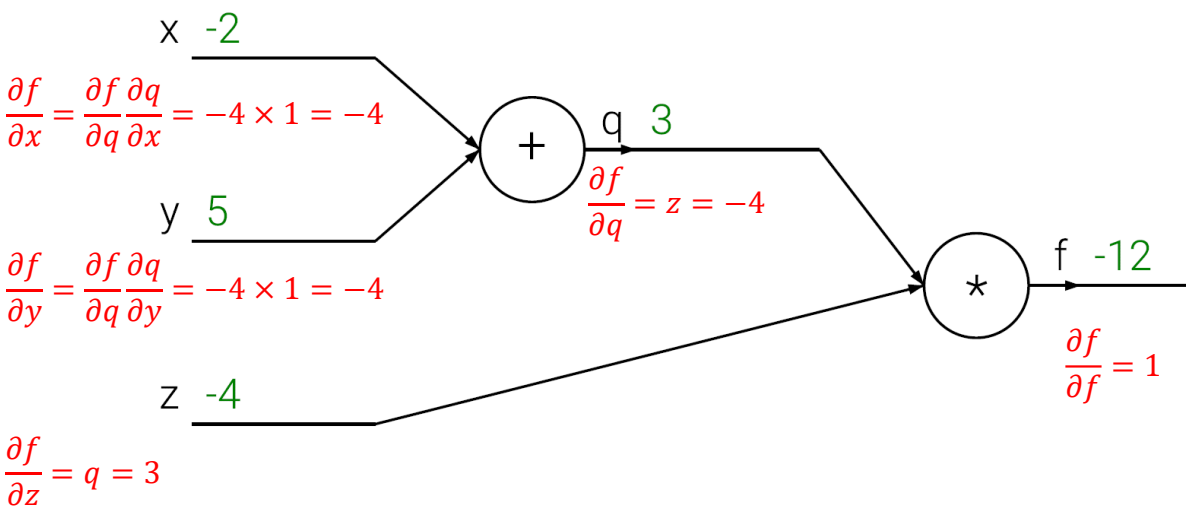
- This computation can be visualized with a circuit diagram:



Backpropagation

Computational graph

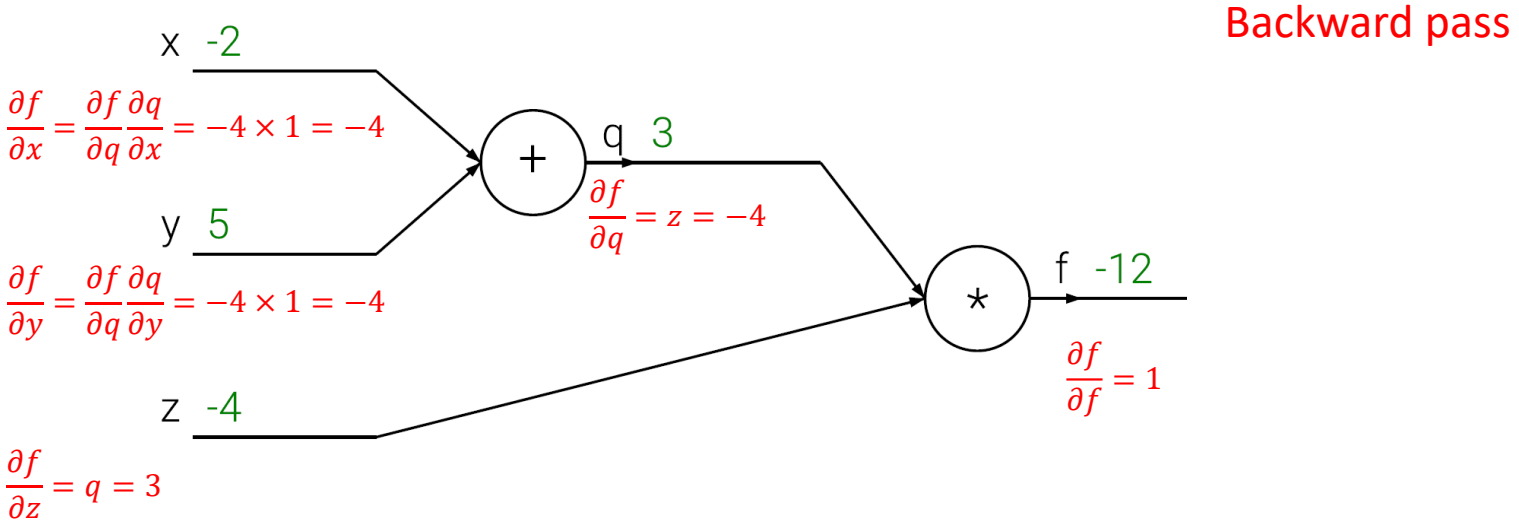
- This computation can be visualized with a circuit diagram:



Backpropagation

Computational graph

- This computation can be visualized with a circuit diagram:

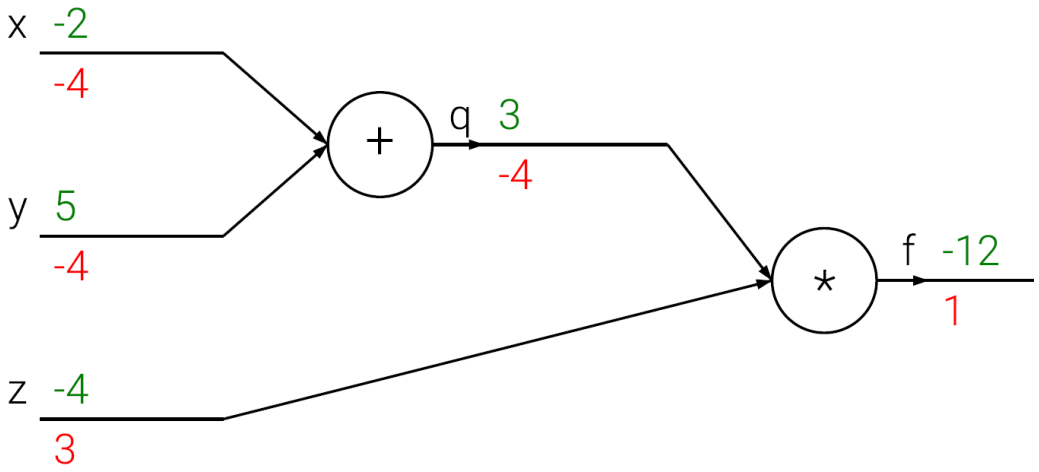


- Backward pass perform **backpropagation** – recursively applies the chain rule to compute the gradients

Backpropagation

Computational graph

- This computation can be visualized with a circuit diagram:



$f(-2{,}5, -4) = (-2 + 5) \cdot -4 = -12$ → Forward pass result

$\nabla f = [-4, -4{,}3]$ → Backward pass result (Backpropagation)

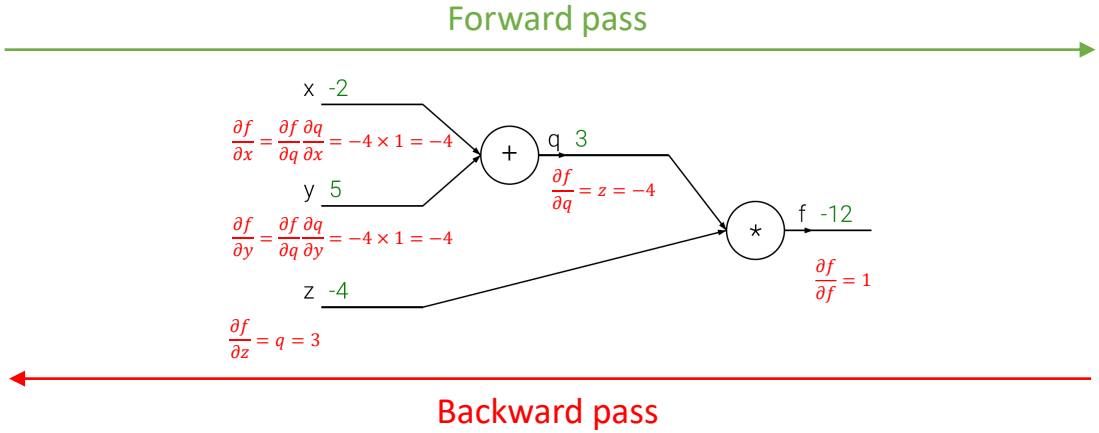
Backpropagation

Intuitive understanding of backpropagation

- Each gate in a circuit diagram compute two things:

  1. Its output value (forward path)

  2. Local gradient of its output with respect to its input

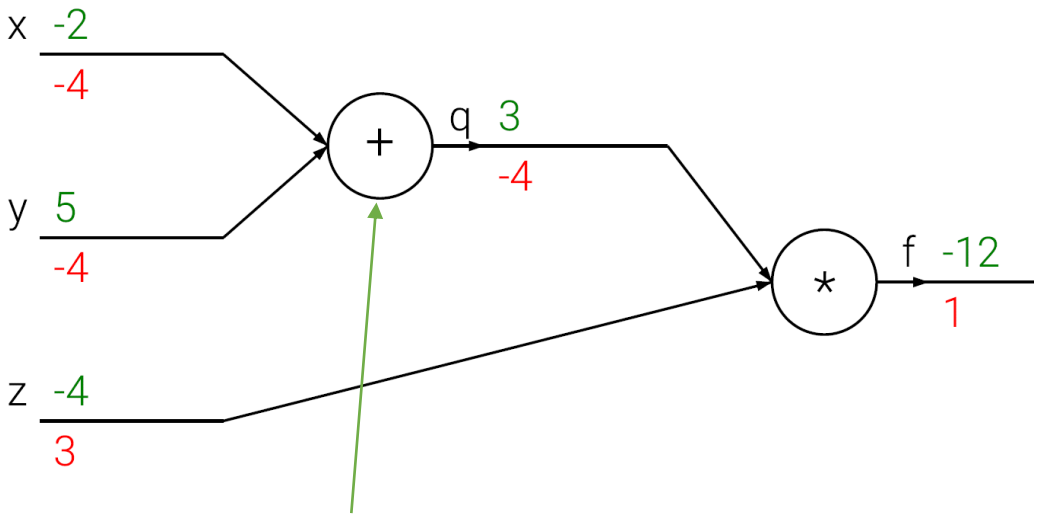


- Gates do this independently, without knowing the full circuit structure.
- During the backward pass, each gate receives the gradient of its output on the final output and:
  - Applies the chain rule to pass the gradient to its inputs by multiplying with local gradients.

Backpropagation

Intuitive understanding of backpropagation

- In this example,

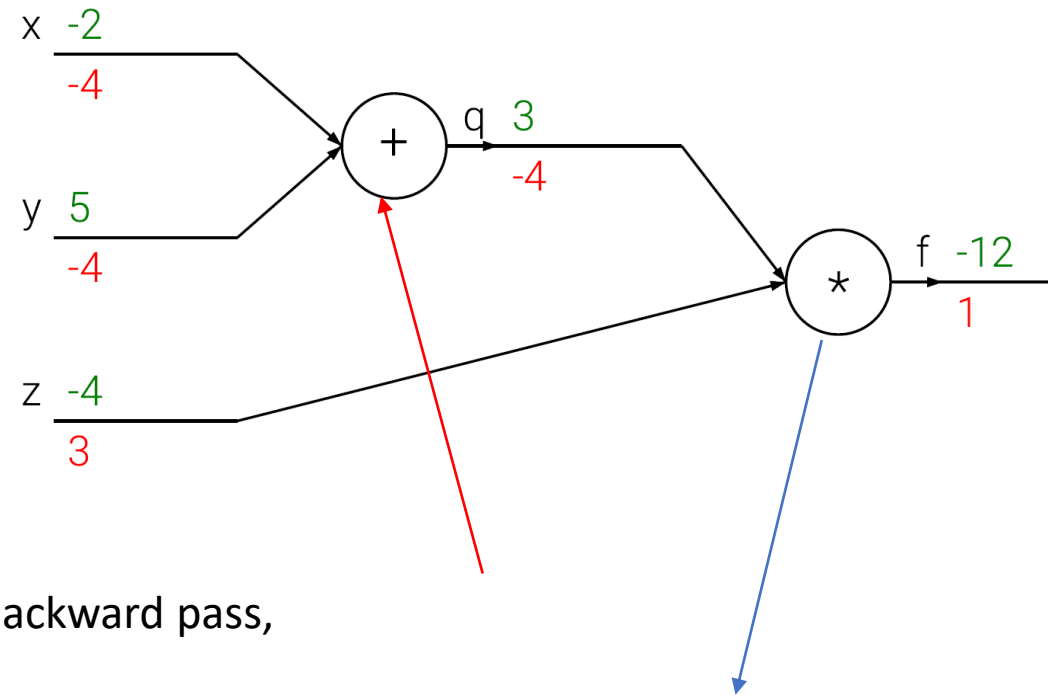


  - During forward pass:
    - Add gate receives input [-2,5] and output 3
    - Its local gradient for both input is +1 (since it's just addition)
  - The full circuit computes a final value -12

Backpropagation

Intuitive understanding of backpropagation

- In this example,

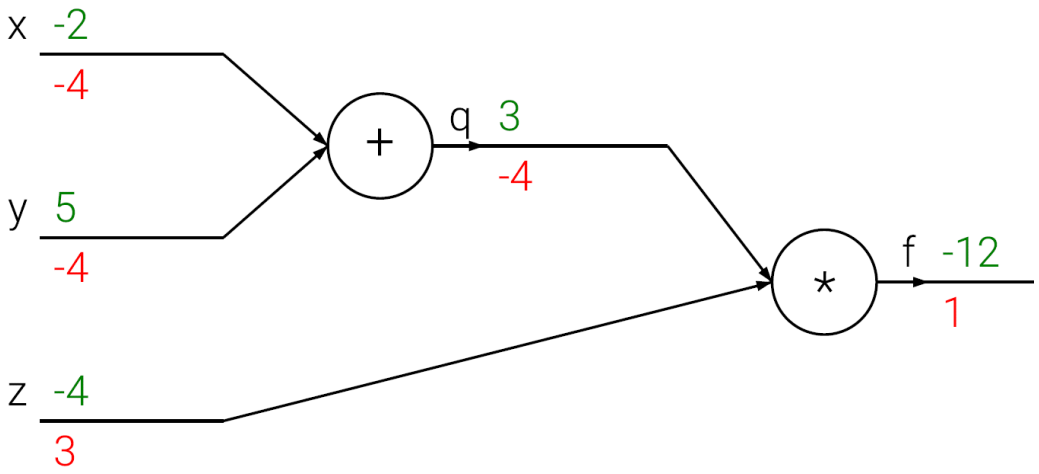


- During the backward pass,
  - Add gate receives a gradient of -4 from multiply gate
  - Applying the chain rule, the add gate multiplies this -4 with its local gradients +1 → resulting in gradients of -4 for both inputs.

Backpropagation

Intuitive understanding of backpropagation

- In this example,



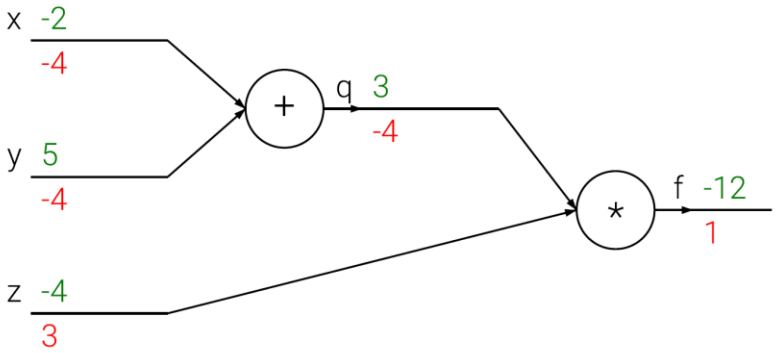
- This makes intuitive sense:
  - Decreasing x or y would lower the add gate's output, which would increase the output of the multiply gate — aligning with the goal of increasing the overall result.

- **Backpropagation** can be viewed as **gates "communicating" via gradients**, telling each other how their outputs should change (and by how much) to improve the final output.

Backpropagation

Torch auto-differentiation

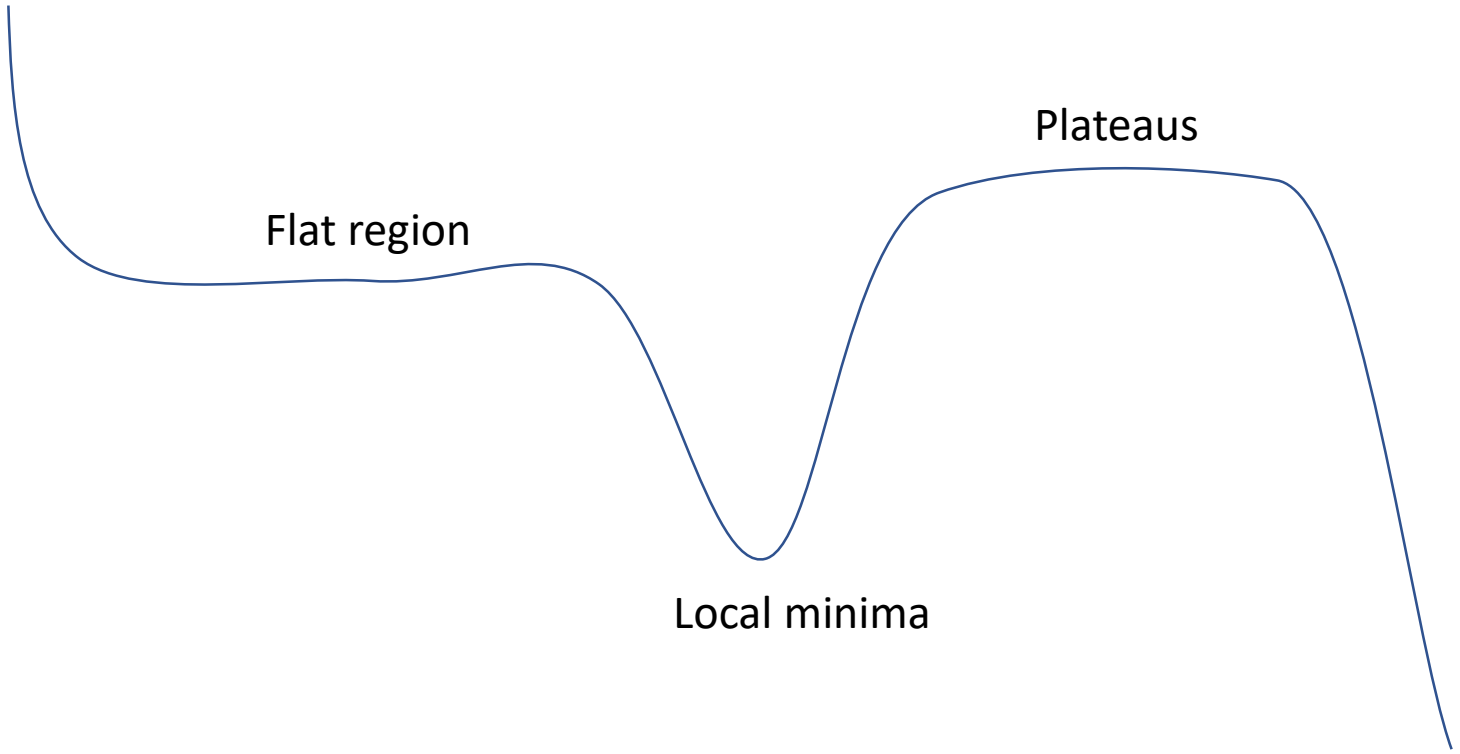
```
import torch

x = torch.tensor(-2.0, requires_grad=True)
y = torch.tensor(5.0, requires_grad=True)
z = torch.tensor(-4.0, requires_grad=True)
q = x + y
f = q * z

f.backward()

print(x.grad)
print(y.grad)
print(z.grad)
```



Optimizers

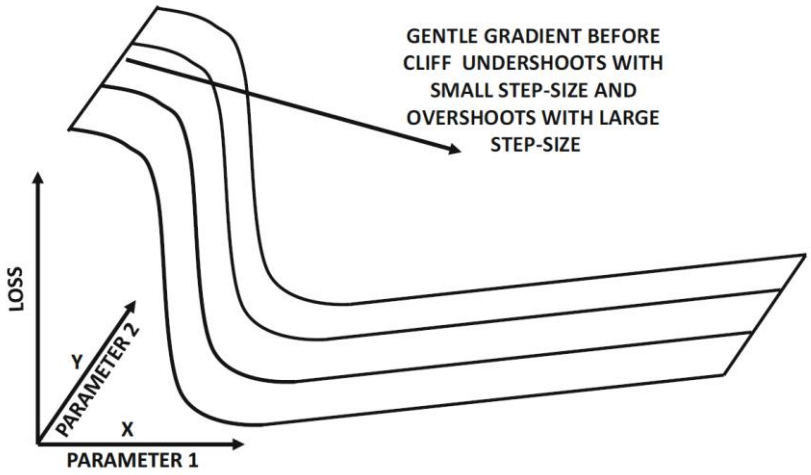
Challenges in Neural Network Optimization

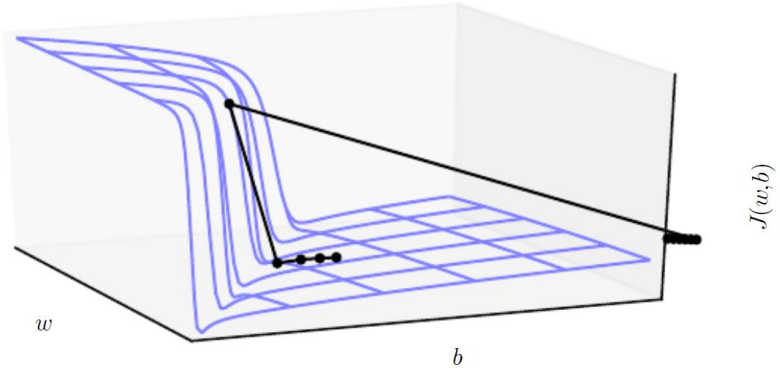
Local minima, Plateaus, Saddle Points and Other Flat Regions



Challenges in Neural Network Optimization

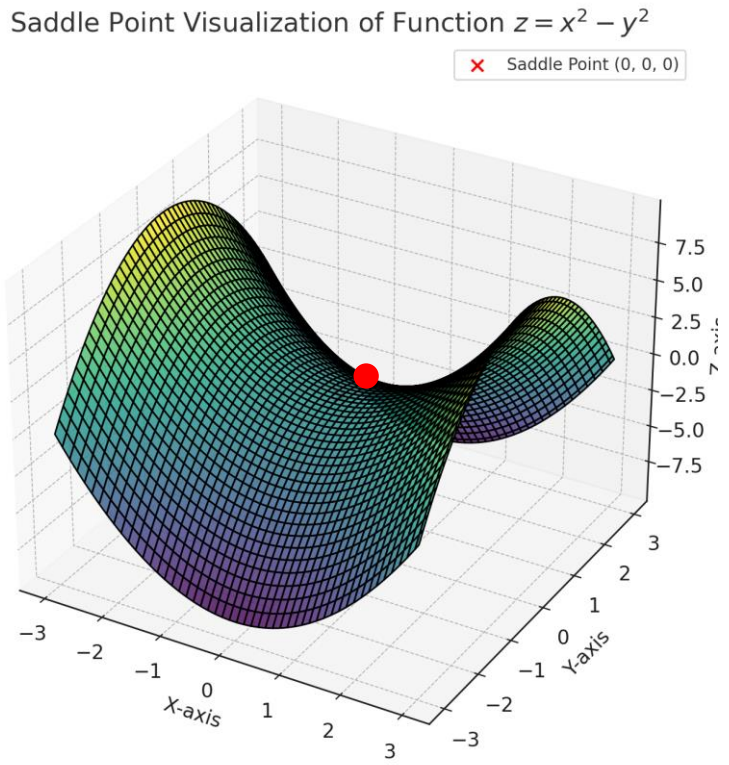
Cliff and exploding gradient





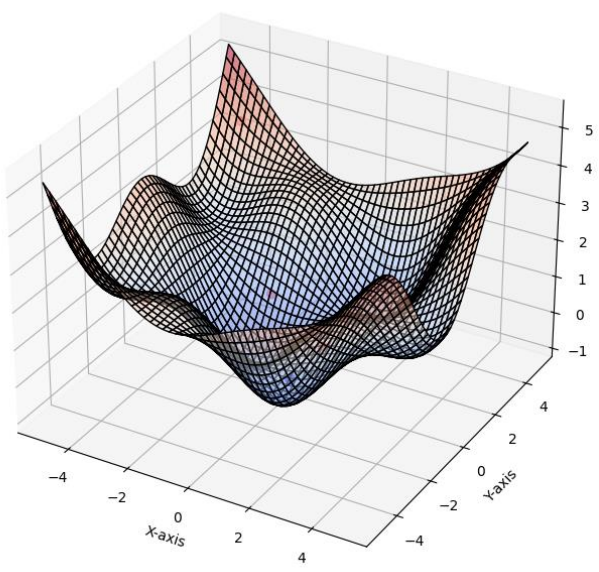
Challenges in Neural Network Optimization

Saddle point



Challenges in Neural Network Optimization

Poor Correspondence between Local and Global Structure



Limitation of Gradient descent

Gradient descent – Limitation 1

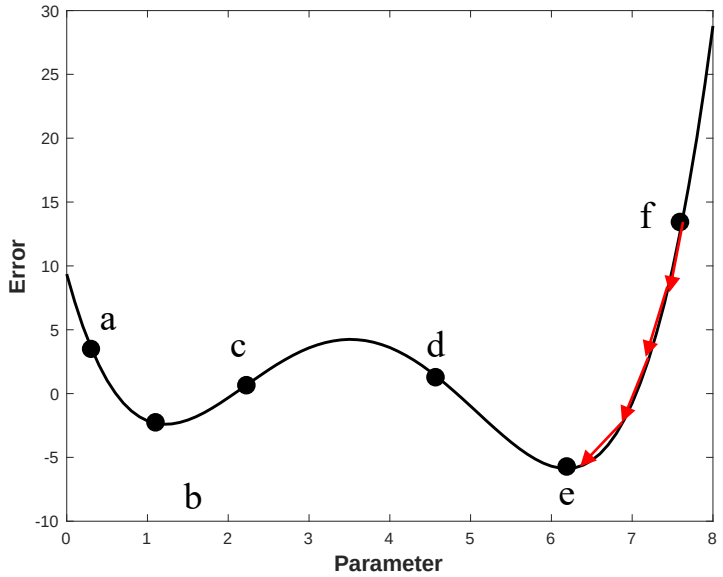
$$w_{new} = w_{old} - \alpha \frac{dL}{dw}$$

- Where,
  - L: Loss
  - w: parameter
  - α: learning rate – size of step

Defined by

$$L = \frac{1}{n}\sum_{x} f(x)$$

$$x = \{x | x \in Dataset\}$$

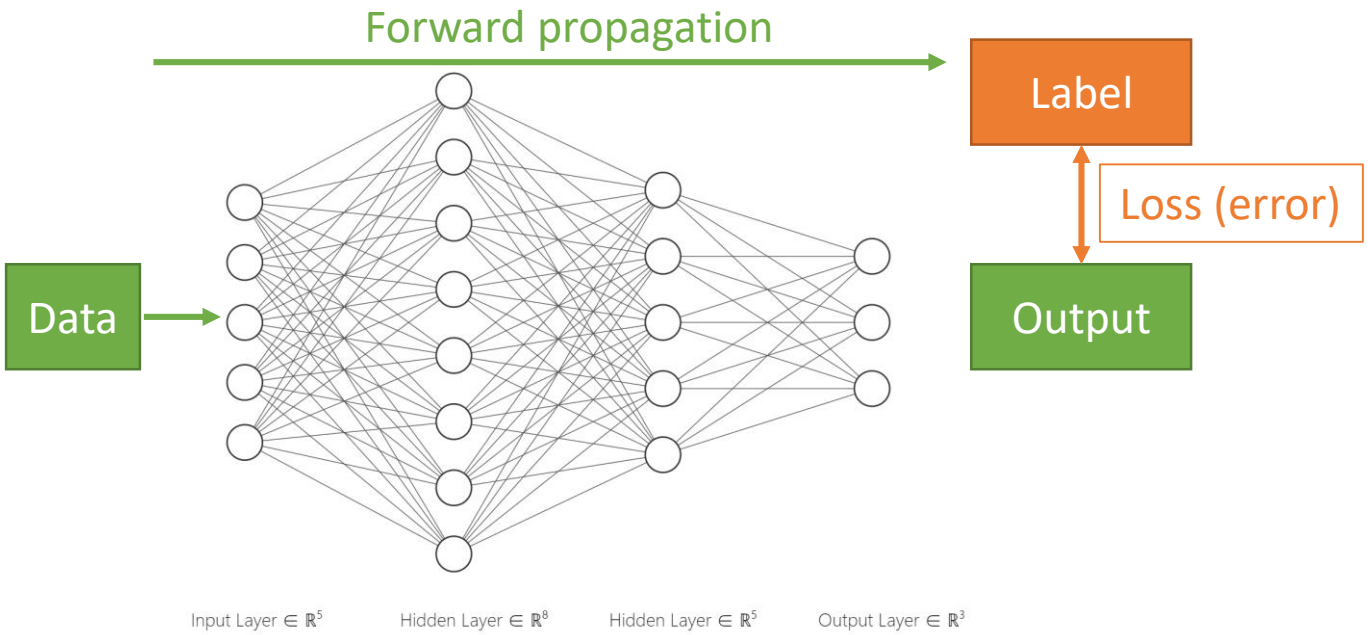


1. Calculate gradient
2. Move to opposite direction of gradient
3. Iterate

Limitation of Gradient descent

Gradient descent – Limitation 1

- In gradient descent, calculating the loss requires making predictions on the entire training dataset.
  → The model's weights are updated only once per epoch
  → Slow convergence



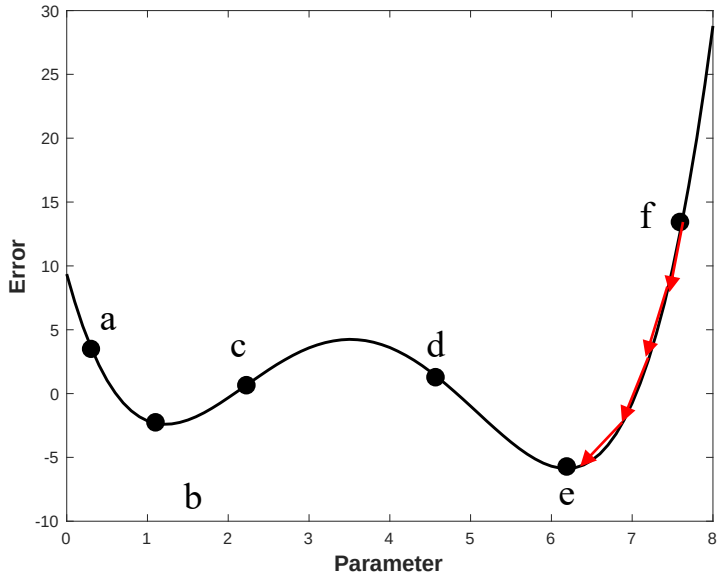
Limitation of Gradient descent

Gradient descent – Limitation 2

$$w_{new} = w_{old} - \alpha \frac{dL}{dw}$$

- Where,
  - L: Loss
  - w: parameter
  - α: learning rate – size of step

Only the gradient at the current weight is considered for the update.



1. Calculate gradient
2. Move to opposite direction of gradient
3. Iterate

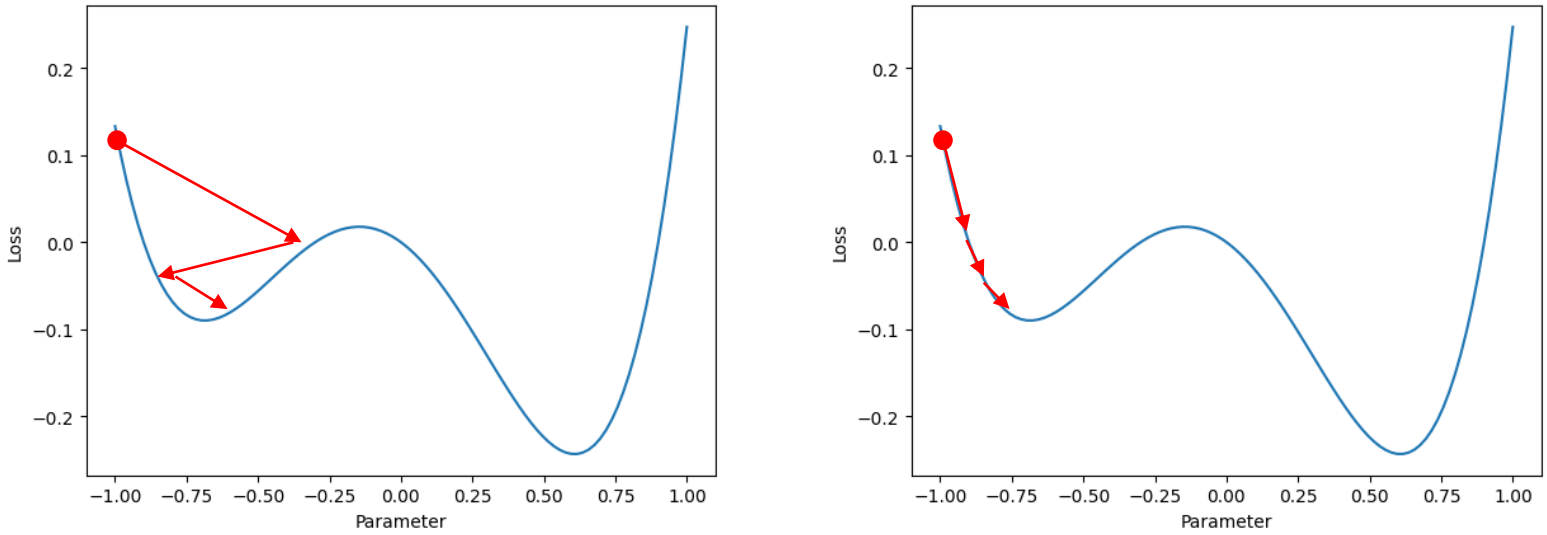
Limitation of Gradient descent

Gradient descent – Limitation 2

$$w_{new} = w_{old} - \alpha \frac{dL}{dw}$$

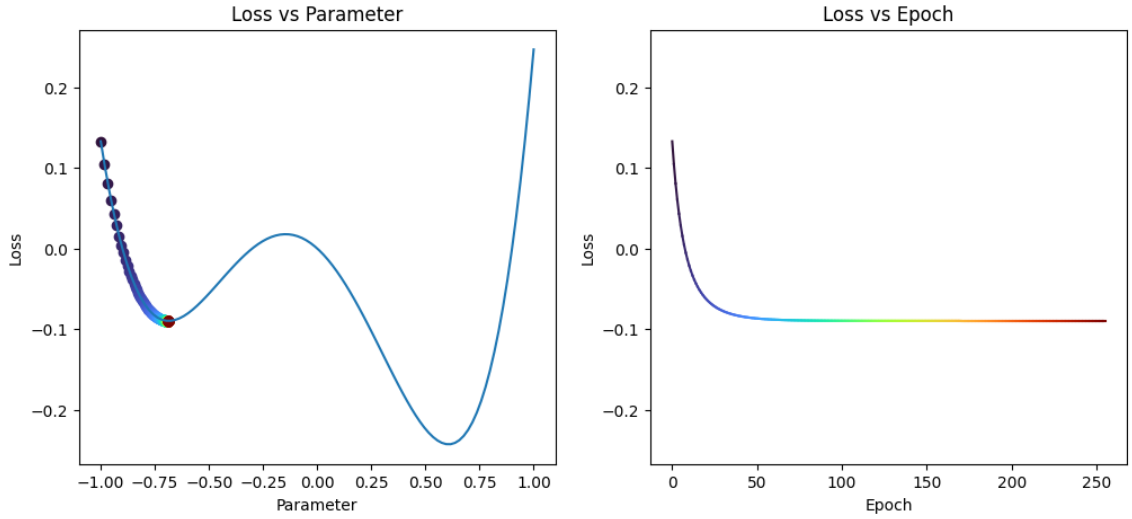
Only the gradient at the current weight is considered for the update.

High chance to get stuck in local minima or saddle points



Limitation of Gradient descent

Gradient descent – Limitation 3 (Sensitive to learning rate)



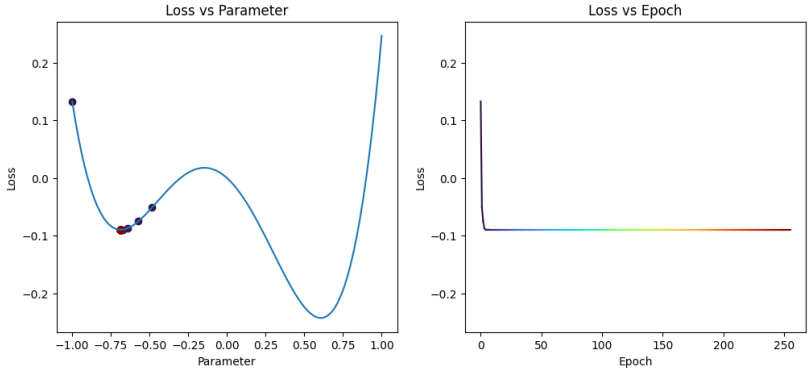
$$w \leftarrow w - \alpha \frac{\partial L}{\partial w}$$

$$L = (w - 0.9) \cdot w \cdot (w + 0.3) \cdot (w + 0.9)$$
$$w_{init} = -0.1$$
$$\alpha = 0.01$$

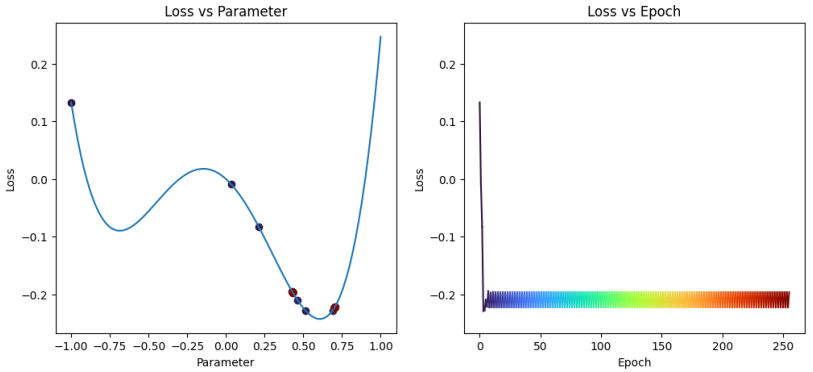
Limitation of Gradient descent

Gradient descent – Limitation 3 (Sensitive to learning rate)

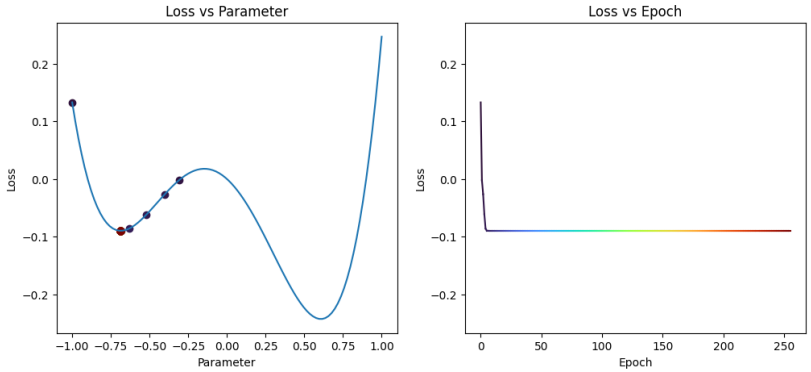
Learning rate 0.3



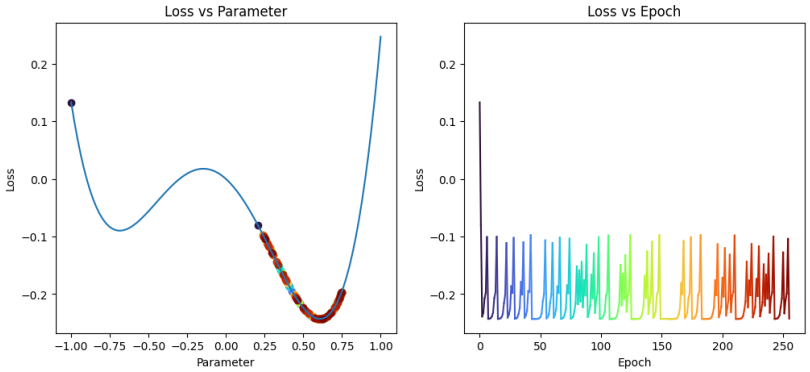
Learning rate 0.6



Learning rate 0.4



Learning rate 0.7



Limitation of Gradient descent

Gradient descent

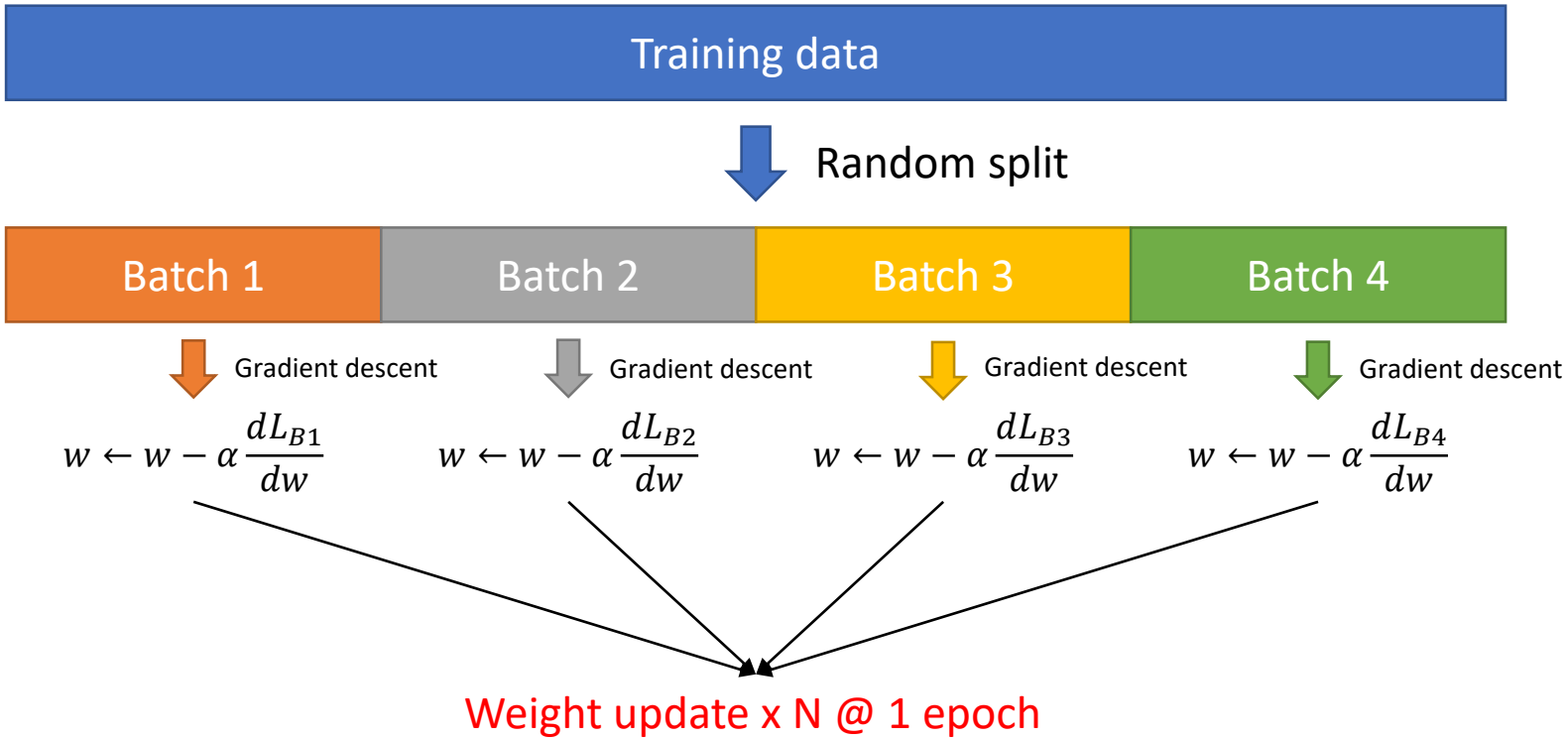
$$w_{new} = w_{old} - \alpha \frac{dL}{dw}$$

- We need more advanced optimizer.

Stochastic gradient descent

Stochastic gradient descent (SGD)

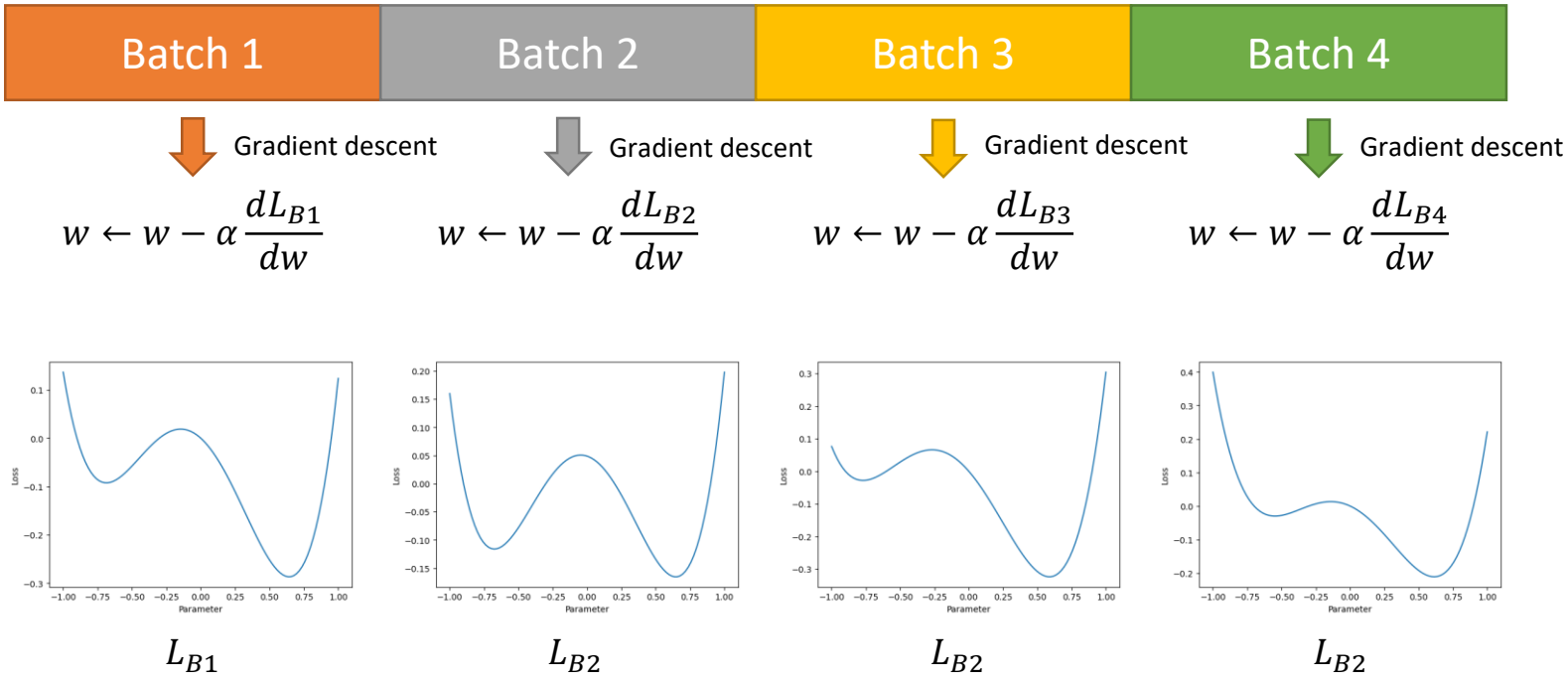
- Using a subset of data ([mini-]batch) to update weight
- Multiple updates within one epoch → Frequent update & Faster convergence
- Tensorflow's default strategy to update weight



Stochastic gradient descent

Limitation - Noisy update

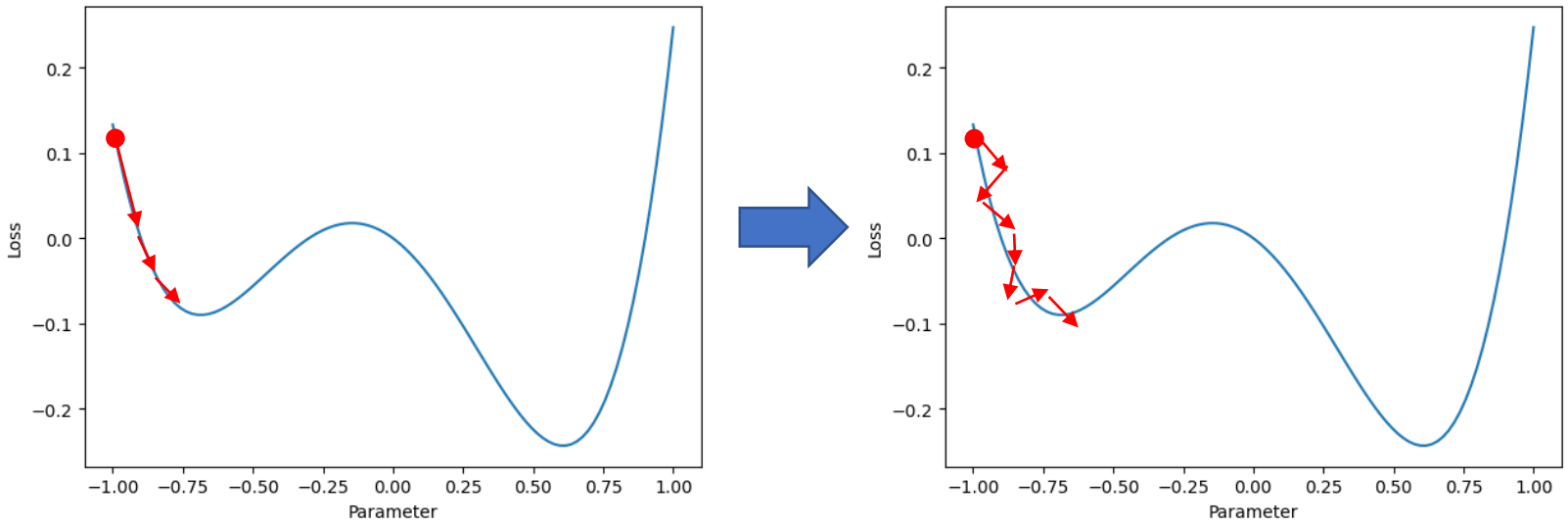
- Since SGD updates the weights based on the gradients from a random subset of the data, these updates can be noisy.
- It can cause the optimization path to oscillate and make it harder to converge smoothly



Stochastic gradient descent

Limitation - Impact of batch size

- **Small** batch size
  - Noise in gradient updates increases
    → Rather, too small batch may slow down the convergence speed.
  - Paradoxically, noisy updates may help avoid local minima.
  - Also may cause overfitting on specific sample.



Stochastic gradient descent

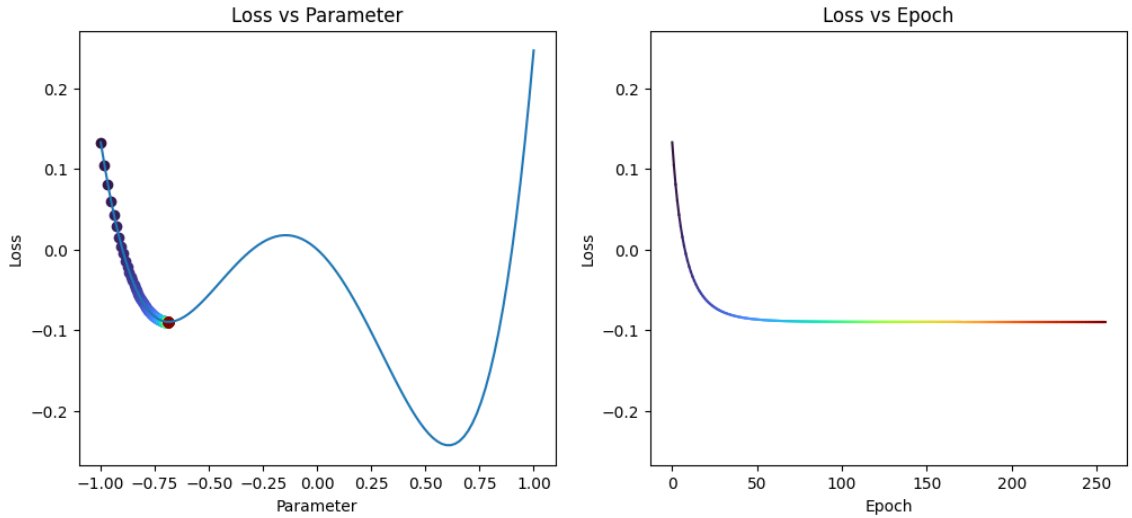
Limitation - Impact of batch size

- **Large** batch size
  - Too large batch size
    → make the optimizer's behavior similar to full-batch gradient descent
  - Large batch sizes require more memory and computational power, which can slow down training and reduce efficiency.

Stochastic gradient descent

Stochastic gradient descent

- Even though SGD accelerates the updating and training process, it still has a high chance of getting stuck in local minima



Stochastic gradient descent

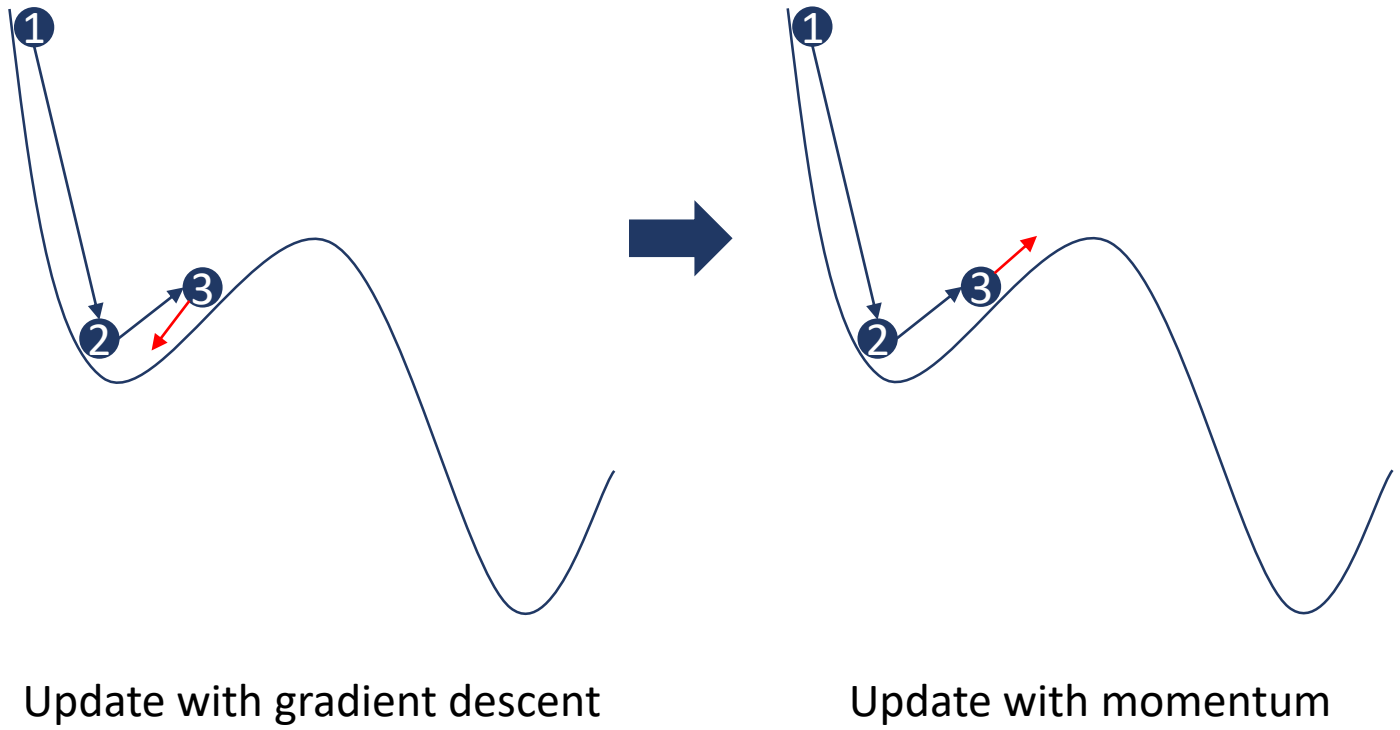
Python implementation

```
class SoftmaxLinear(nn.Module):
    def __init__(self, input_dim=28*28, num_classes=10):
        super().__init__()
        self.fc = nn.Linear(input_dim, num_classes)

    def forward(self, x):
        logits = self.fc(x)
        return logits

model = SoftmaxLinear().to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

for epoch in range(0, 100):
    model.train()
    total_loss = 0.0

    for x, y in train_loader:
        x, y = x.to(device), y.to(device)

        optimizer.zero_grad()
        logits = model(x)
        loss = criterion(logits, y)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()


    print(f"Epoch [{epoch+1}] Loss: {total_loss/len(train_loader):.4f}")
```

Momentum

Momentum

- Technique to improve gradient descent by accumulating a moving average of past gradients



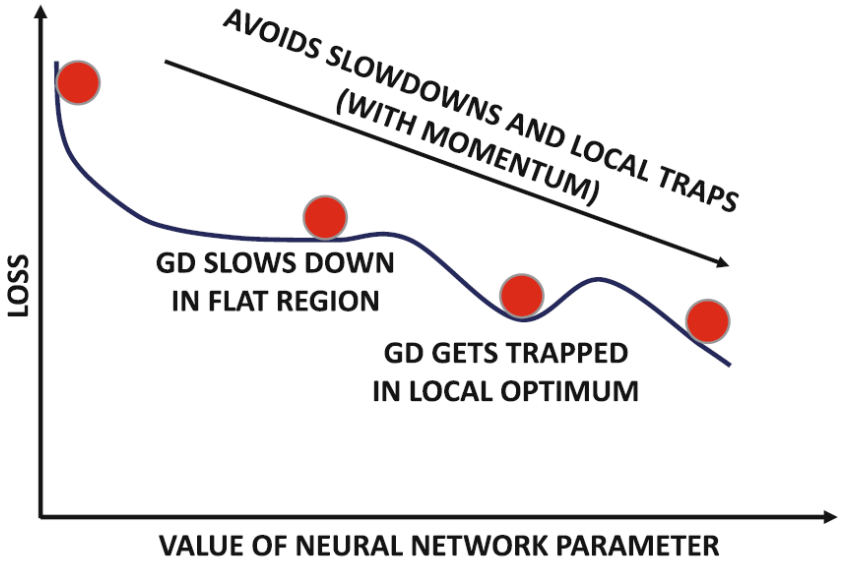
Update with gradient descent

Update with momentum

Momentum

Momentum

- Technique to improve gradient descent by accumulating a moving average of past gradients



Charu C. Aggarwal, "Neural Networks and Deep Learning: A Textbook"

Momentum

Momentum

- **Velocity term**

$$v_0 = 0$$

$$v_t = \gamma v_{t-1} + \alpha \frac{\partial L}{\partial w}$$

  - v: velocity, accumulated gradient
  - γ: momentum ($0 < \gamma < 1$), how much of the past gradient is carried forward, normally 0.8~0.99

- **Weight update**

$$w_{t+1} \leftarrow w_t - v_t$$

  - Where w_t is weight of time t

Momentum

Momentum

- **Velocity term**

$$v_0 = 0$$

$$v_t = \gamma v_{t-1} + \alpha \frac{\partial L}{\partial w}$$

Inducing momentum
+ Reinforcement the gradient that continues in the same direction.

  - v: velocity, accumulated gradient
  - γ: momentum ($0 < \gamma < 1$), how much of the past gradient is carried forward, normally 0.8~0.99

- **Weight update**

$$w_{t+1} \leftarrow w_t - v_t$$

  - Where w_t is weight of time t

Momentum

Momentum

- **Weight update flow**

Loop	Velocity	Weight update
1	$v_1 = \gamma v_0 + \alpha \nabla L(w_1)$ $= \alpha \nabla L(w_1)$	$w_2 \leftarrow w_1 - v_1$ $= w_1 - \alpha \nabla L(w_1)$
2	$v_2 = \gamma v_1 + \alpha \nabla L(w_2)$ $= \gamma \alpha \nabla L(w_1) + \alpha \nabla L(w_2)$ $= \alpha \left(\gamma \nabla L(w_1) + \nabla L(w_2) \right)$	$w_3 \leftarrow w_2 - v_2$ $= w_2 - \alpha \left(\gamma \nabla L(w_1) + \nabla L(w_2) \right)$
3	$v_3 = \gamma v_2 + \alpha \nabla L(w_1)$ $= \gamma^2 \alpha \nabla L(w_1) + \gamma \alpha \nabla L(w_2) + \alpha \nabla L(w_3)$ $= \alpha \left(\gamma^2 \nabla L(w_1) + \gamma \nabla L(w_2) + \nabla L(w_3) \right)$	$w_3 \leftarrow w_3 - v_3$ $= w_3 - \alpha \left(\gamma^2 \nabla L(w_1) + \gamma \nabla L(w_2) + \nabla L(w_3) \right)$
⋮	⋮	⋮

- $\nabla L = \frac{\partial L}{\partial w}$

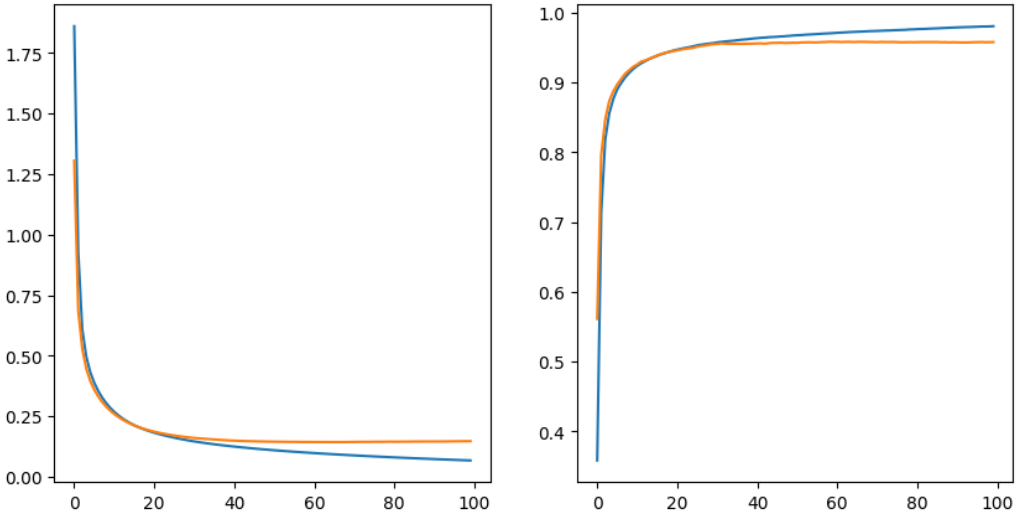
Momentum

Python implementation

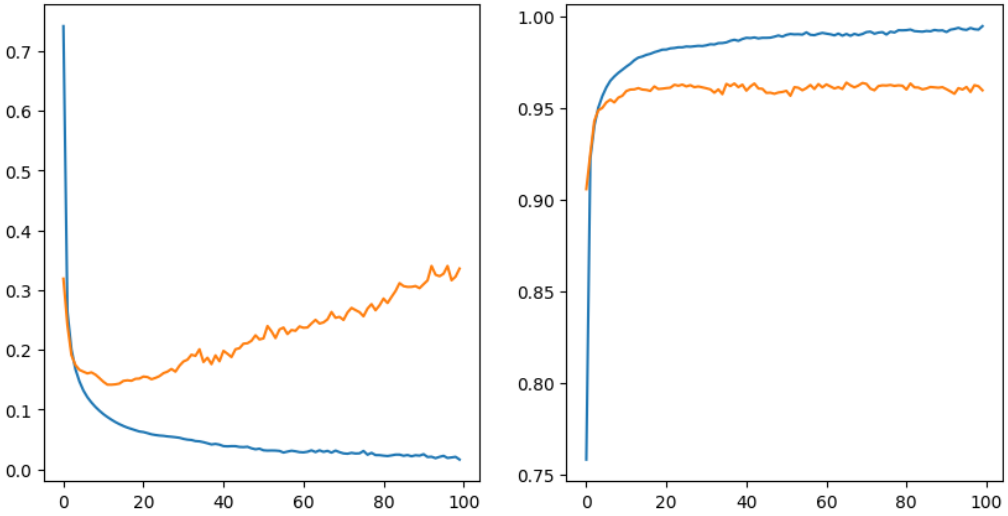
```
class SoftmaxLinear(nn.Module):
    def __init__(self, input_dim=28*28, num_classes=10):
        super().__init__()
        self.fc = nn.Linear(input_dim, num_classes)

    def forward(self, x):
        logits = self.fc(x)
        return logits

model = SoftmaxLinear().to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)

for epoch in range(0, 100):
    model.train()
    total_loss = 0.0

    for x, y in train_loader:
        x, y = x.to(device), y.to(device)

        optimizer.zero_grad()
        logits = model(x)
        loss = criterion(logits, y)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()


    print(f"Epoch [{epoch+1}] Loss: {total_loss/len(train_loader):.4f}")
```

Momentum

Python implementation



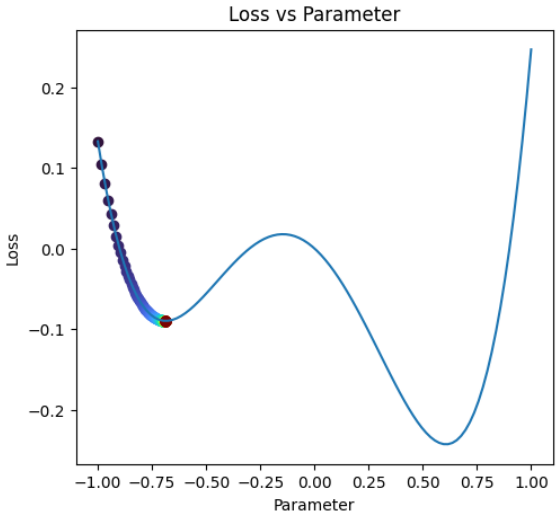
SGD

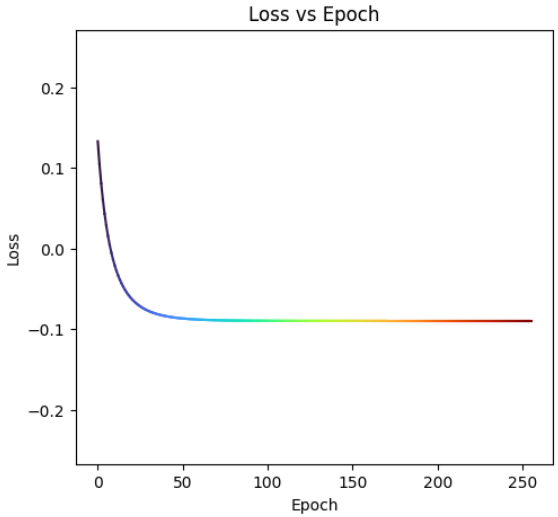


SGD + Momentum
- More fast progress

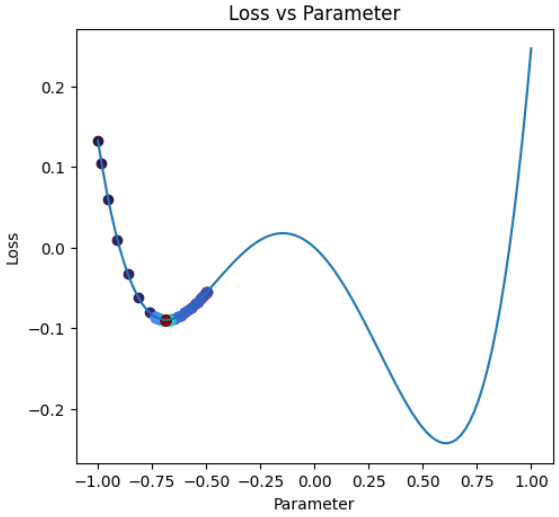
Momentum

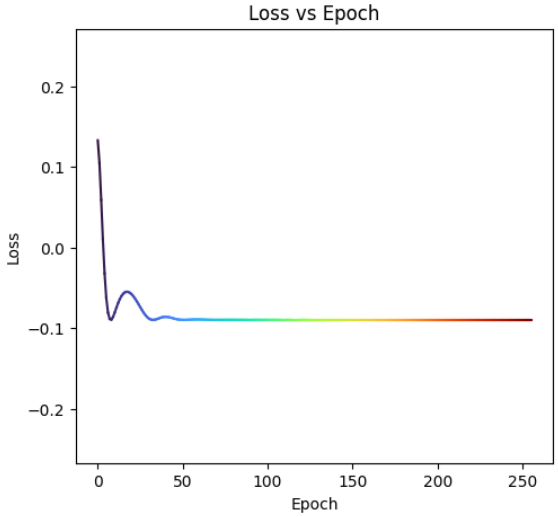
Momentum





Learning rate 0.01
Momentum 0.0

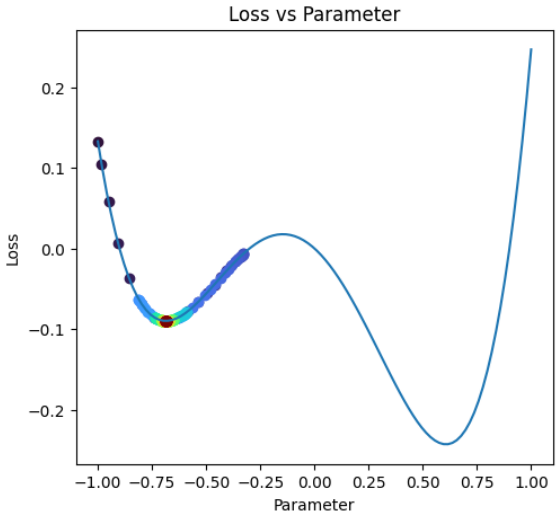


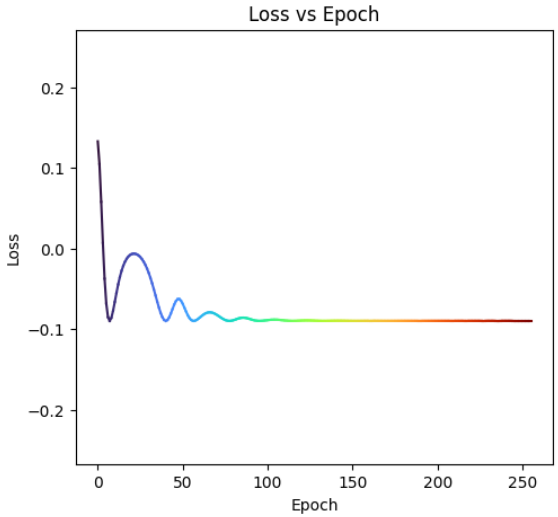


Learning rate 0.01
Momentum 0.9

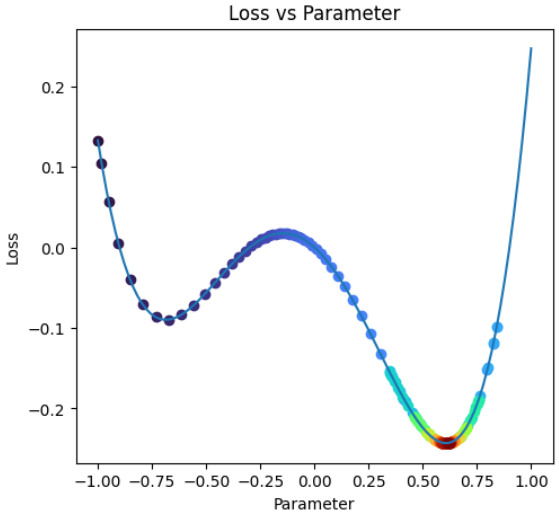
Momentum

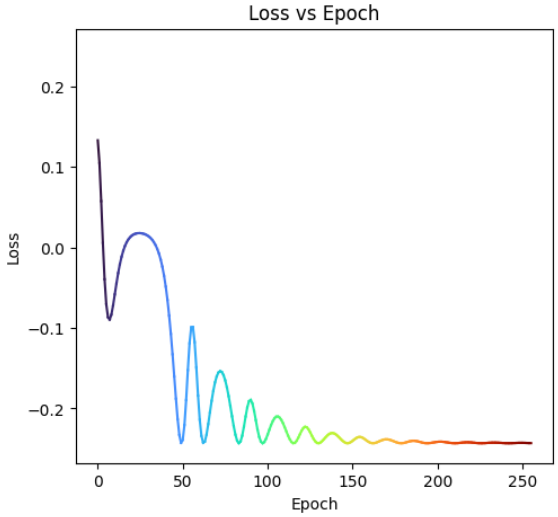
Momentum





Learning rate 0.01
Momentum 0.95

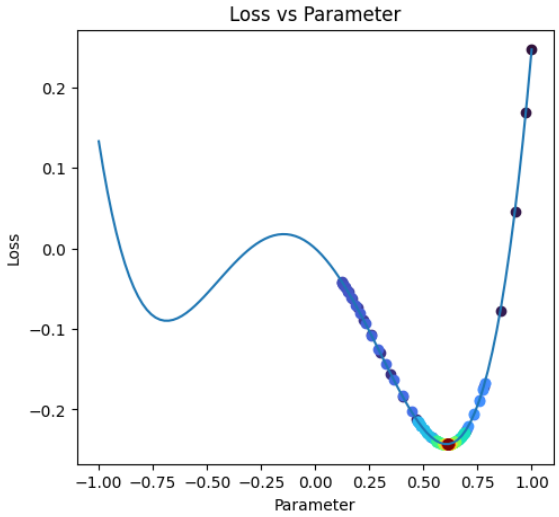


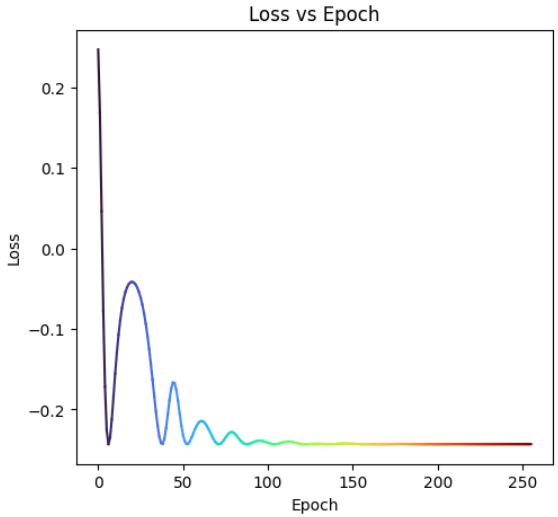


Learning rate 0.01
Momentum 0.97

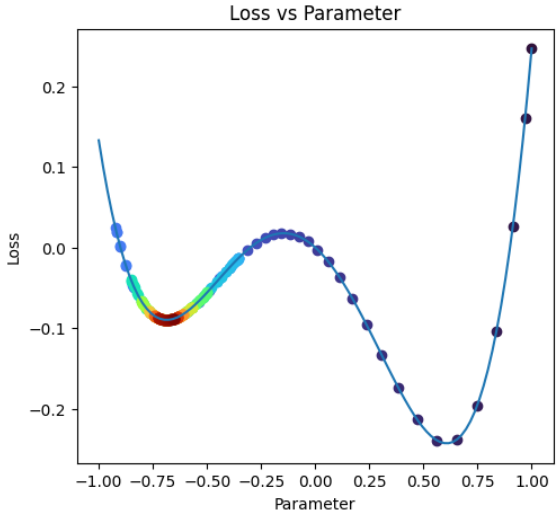
Momentum

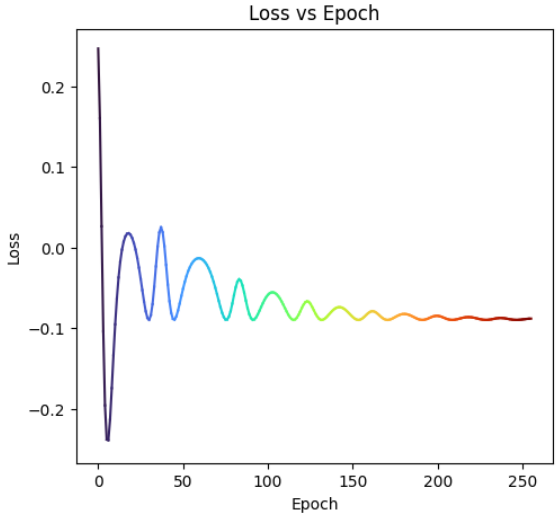
Momentum





Initial weight = 1.0
Learning rate 0.01
Momentum 0.95





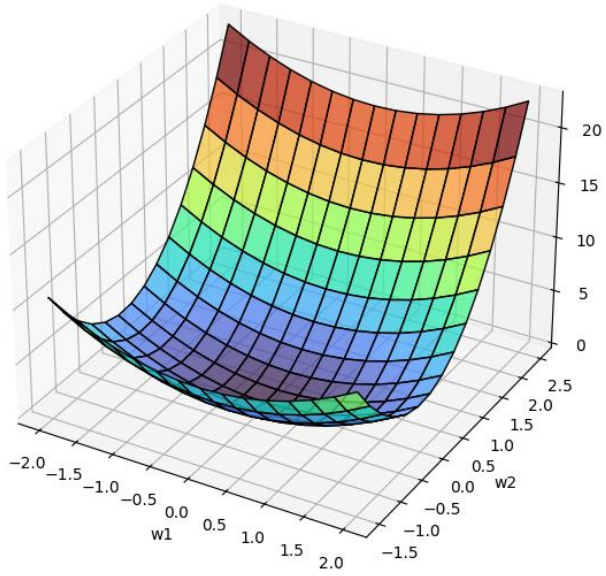
Initial weight = 1.0
Learning rate 0.01
Momentum 0.98

(Failure)

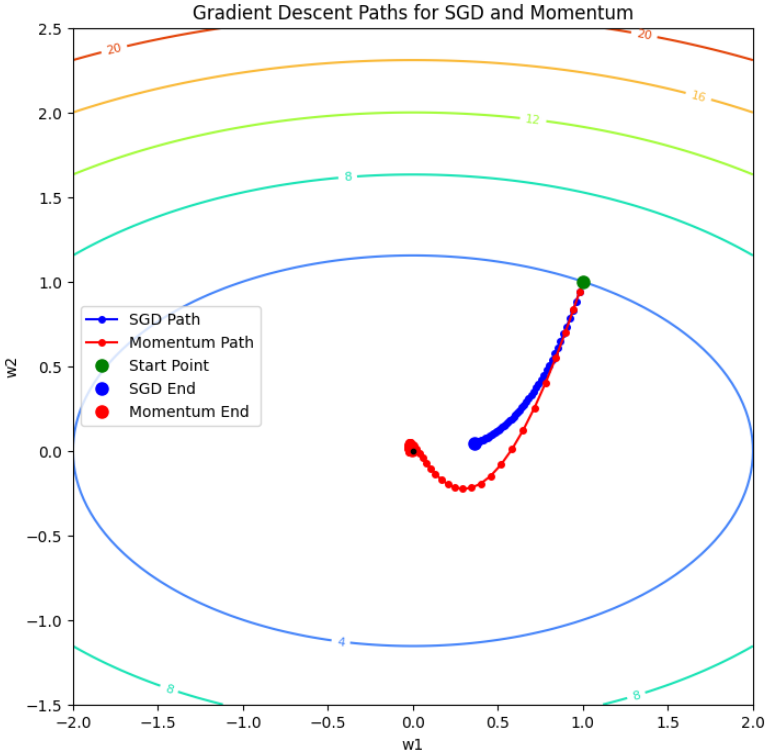
Momentum

SGD vs Momentum

Loss space



$$L = w_1^2 + 3w_2^2$$



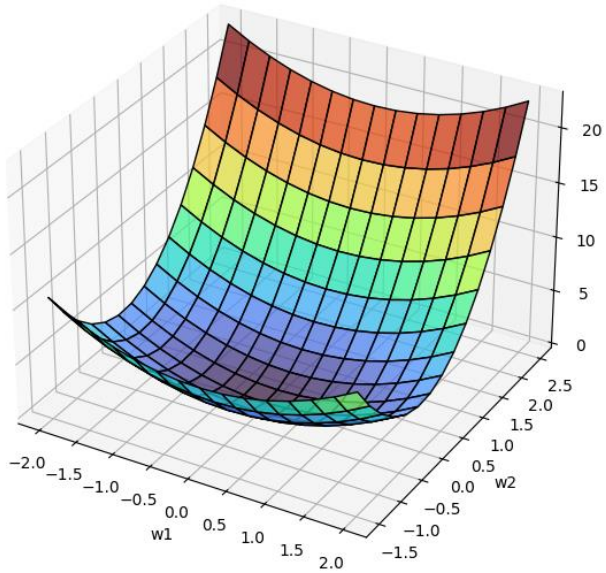
Learning rate 0.01, Momentum 0.8
Iteration 50

Momentum

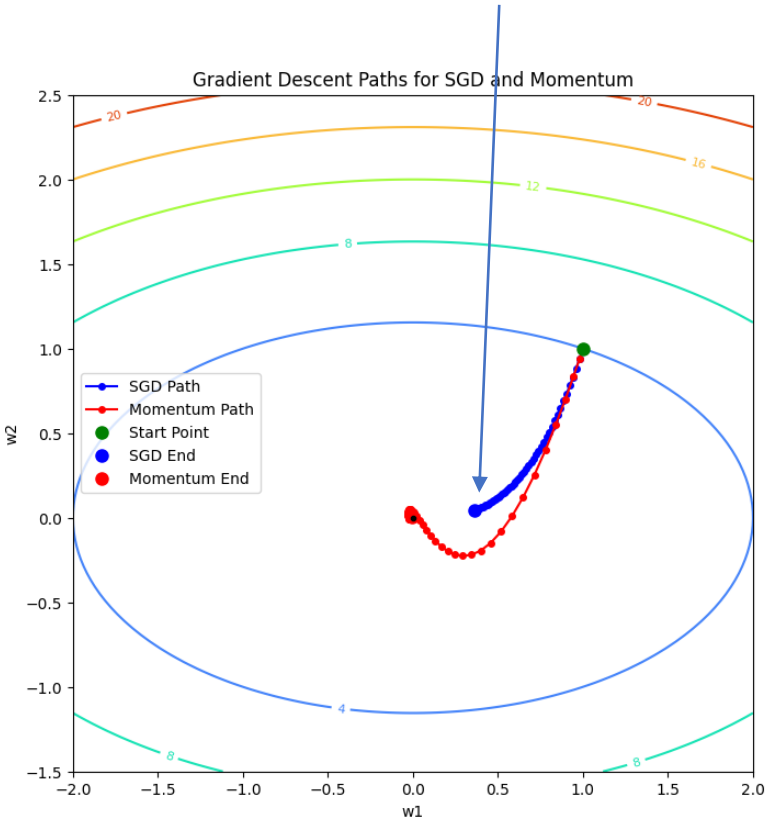
SGD vs Momentum

The size of updates by SGD decreases too much as it gets closer to the optimal point.

Loss space



$$L = w_1^2 + 3w_2^2$$



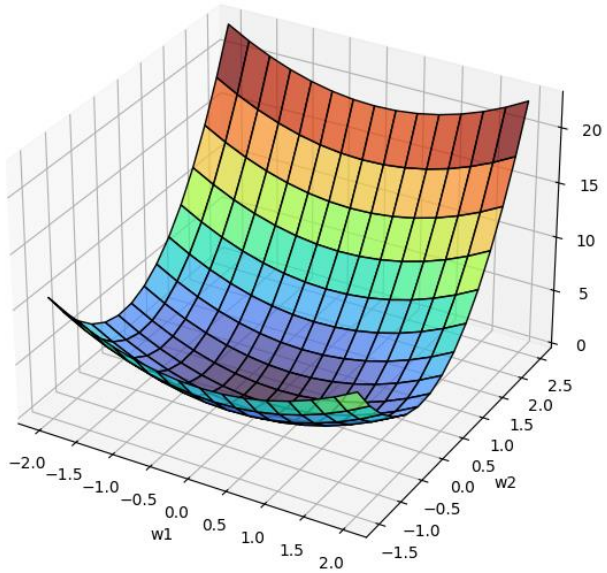
Learning rate 0.01, Momentum 0.8
Iteration 50

AdaGrad

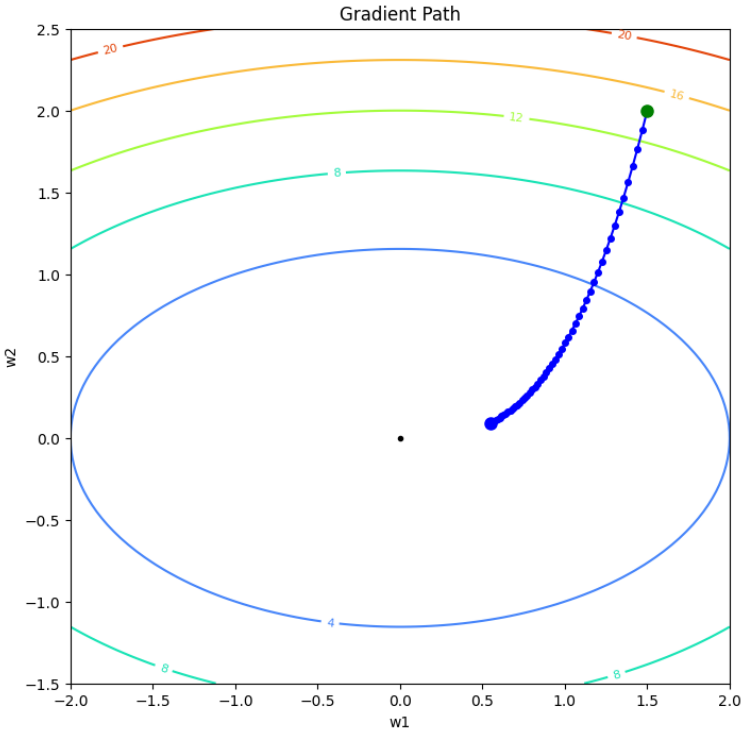
AdaGrad (Adaptive Gradient)

- Is it appropriate to apply the same learning rate to all weights?

Loss space



$$L = w_1^2 + 3w_2^2$$



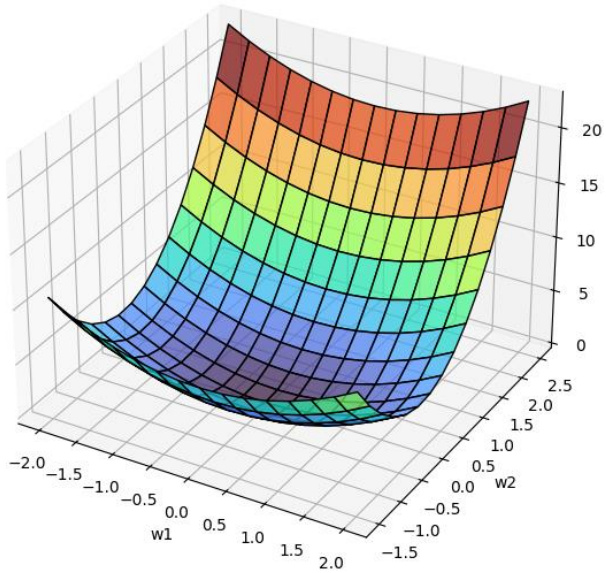
Duchi, John, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of machine learning research* 12.7 (2011).

AdaGrad

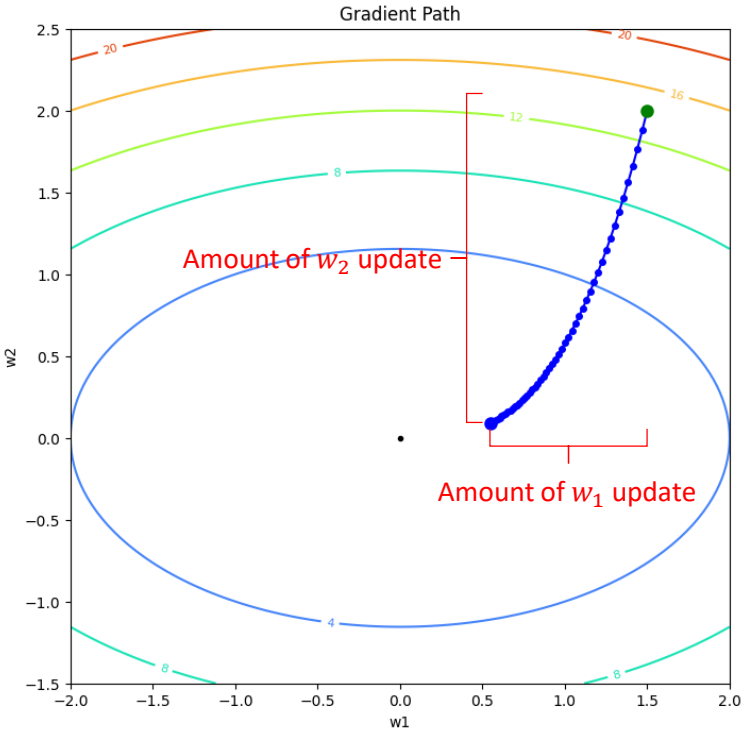
AdaGrad (Adaptive Gradient)

- Is it appropriate to apply the same learning rate to all weights?

Loss space



$$L = w_1^2 + 3w_2^2$$



Remained distance to optimal point: $w_1 > w_2$

~~Gradient size:$w_1 < w_2$~~

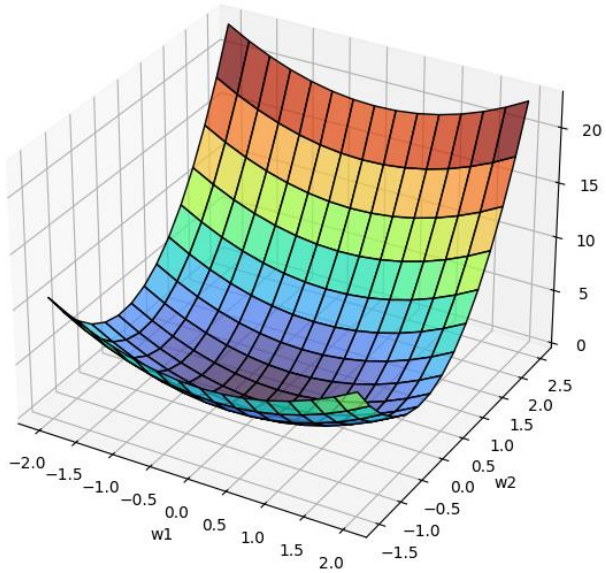
~~w_1 will be updated less than w_2~~

AdaGrad

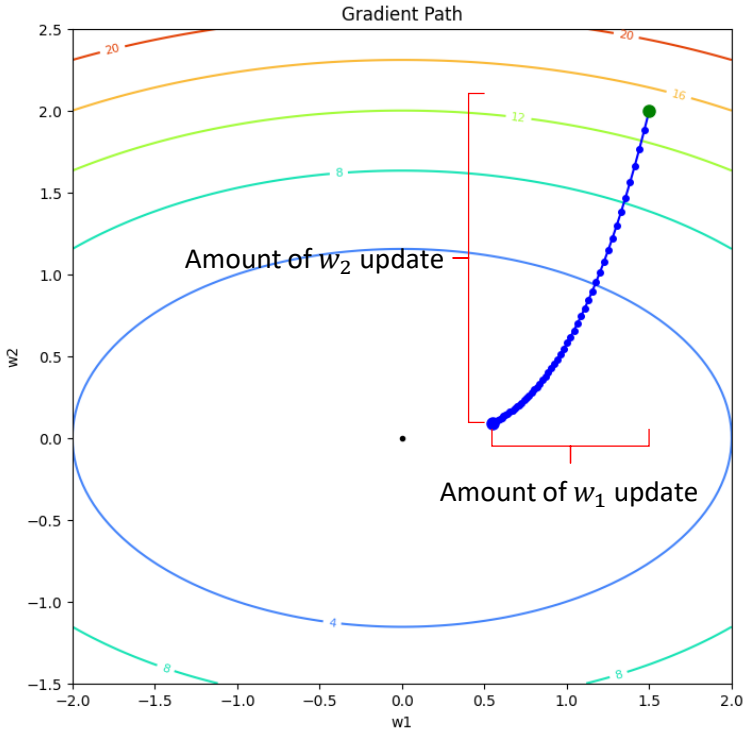
AdaGrad (Adaptive Gradient)

- Is it appropriate to apply the same learning rate to all weights?
  → We need to adjust learning rate for each parameter adaptively

Loss space



$$L = w_1^2 + 3w_2^2$$



Remained distance to optimal point: $w_1 > w_2$
Gradient size: $w_1 < w_2$
w_1 will be updated less than w_2

AdaGrad

AdaGrad (Adaptive Gradient)

- Divide accumulated squared gradients
- Sum of squared gradients for each weights

$$G_t = G_{t-1} + g_t^2$$

  - $g_t = \nabla L(w_t)$

- Weight update

$$w_{t+1} \leftarrow w_t - \frac{\alpha}{\sqrt{G_t + \varepsilon}} \cdot g_t$$

  - ε: small number (e.g., 10^{-8})

Duchi, John, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of machine learning research* 12.7 (2011).

AdaGrad

AdaGrad (Adaptive Gradient)

- Pseudo code

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
τ (initial accumulator value), η (lr decay)

initialize : $state\_sum_0 \leftarrow \tau$

for $t = 1$ **to** ... **do**

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$\tilde{\gamma} \leftarrow \gamma/(1 + (t-1)\eta)$

if $\lambda \neq 0$

$g_t \leftarrow g_t + \lambda\theta_{t-1}$

$state\_sum_t \leftarrow state\_sum_{t-1} + g_t^2$

$$\theta_t \leftarrow \theta_{t-1} - \tilde{\gamma}\frac{g_t}{\sqrt{state\_sum_t} + \epsilon}$$

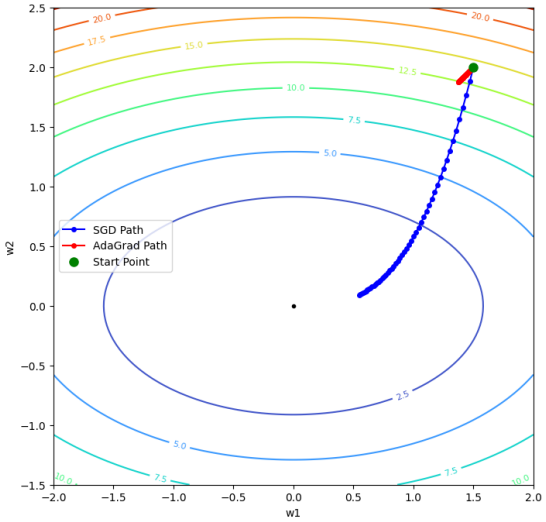
return θ_t

https://pytorch.org/docs/stable/generated/torch.optim.Adagrad.html#torch.optim.Adagrad

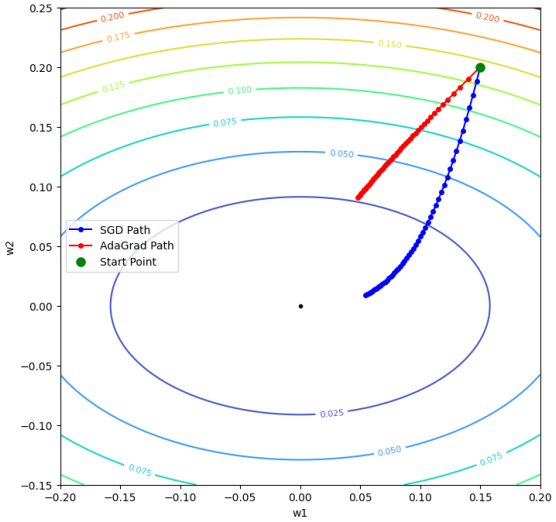
AdaGrad

AdaGrad VS SGD

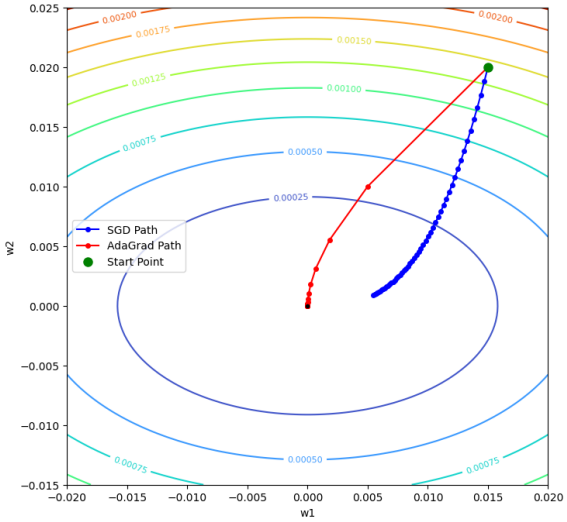
- $\alpha = 0.01, iter = 50, \varepsilon = 1e-8$



$\boldsymbol{w}_{init} = (1.5, 2.0)$



$\boldsymbol{w}_{init} = (0.15, 0.2)$



$\boldsymbol{w}_{init} = (0.015, 0.02)$

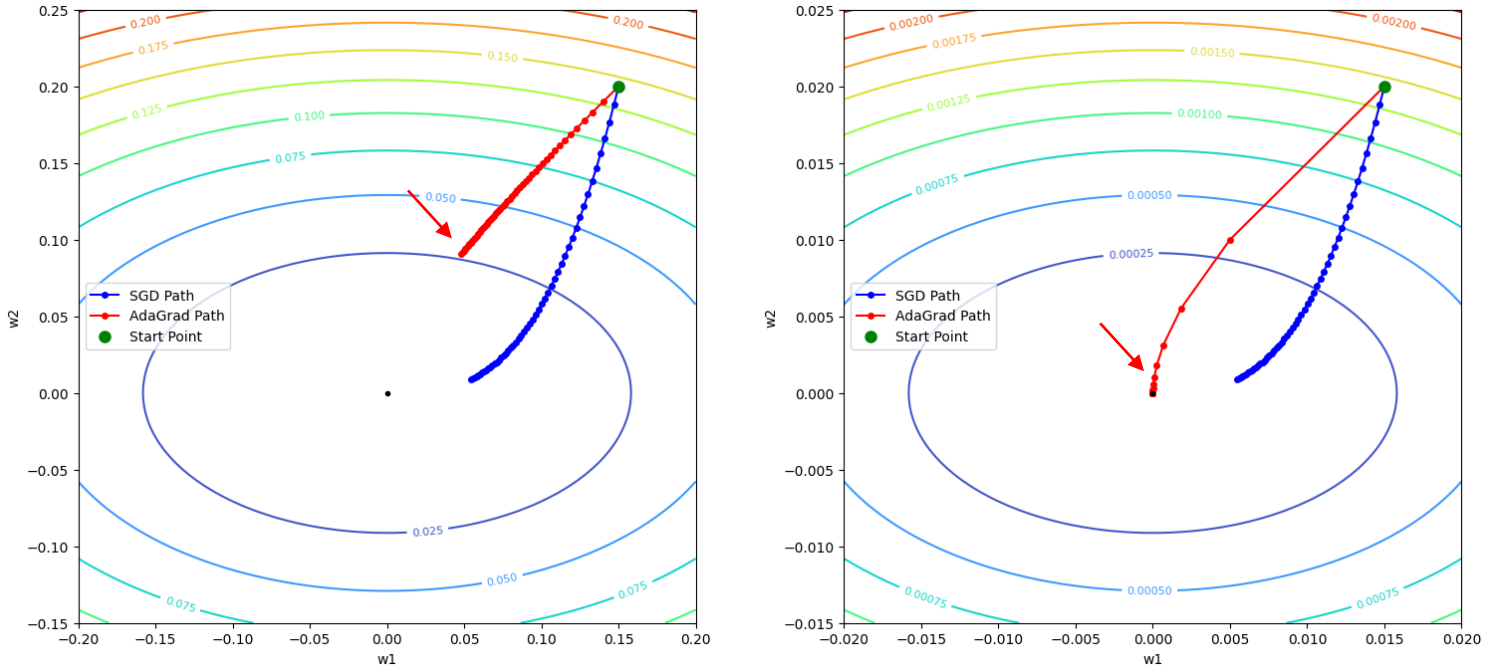
→ More effective when weight is small!

AdaGrad

Limitation of AdaGrad

- **Learning Rate Decay Issue**: Accumulated squared gradients keep increasing, causing the effective learning rate to keep decreasing.

- **Slow Convergence in Later Stages**: As training progresses, the learning rate can become too small, leading to slow convergence.



RMSprop

RMSprop (Root Mean Square Propagation)

- Improved version of AdaGrad
- Using a exponential moving average of squared gradients rather than the cumulative sum
- Reference:
  - Hinton, G. (2012). Neural Networks for Machine Learning [Coursera video lectures]. Lecture 6e.

RMSprop

RMSprop (Root Mean Square Propagation)

- Exponential moving average of gradients

$$h_t = \gamma h_{t-1} + (1-\gamma) g_t^2$$

  - $g_t = \nabla L(w_t)$
  - γ: decay rate (usually around 0.9)

- Weight update

$$w_{t+1} \leftarrow w_t - \frac{\alpha}{\sqrt{h_t + \varepsilon}} \cdot g_t$$

  - ε: small number (e.g., 10^{-8})

RMSprop

RMSprop (Root Mean Square Propagation)

- Pseudo code

---

input : α (alpha), γ (lr), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), μ (momentum), *centered*
initialize : $v_0 \leftarrow 0$ (square average), $\mathbf{b}_0 \leftarrow 0$ (buffer), $g_0^{ave} \leftarrow 0$

---

for $t = 1$ **to** ... **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  $if\ \lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $v_t \leftarrow \alpha v_{t-1} + (1-\alpha)g_t^2$
  $\tilde{v}_t \leftarrow v_t$
  $if\ centered$
    $g_t^{ave} \leftarrow g_{t-1}^{ave}\alpha + (1-\alpha)g_t$
    $\tilde{v}_t \leftarrow \tilde{v}_t - \left(g_t^{ave}\right)^2$
  $if\ \mu > 0$
    $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + g_t/\left(\sqrt{\tilde{v}_t} + \epsilon\right)$
    $\theta_t \leftarrow \theta_{t-1} - \gamma\mathbf{b}_t$
  $else$
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t/\left(\sqrt{\tilde{v}_t} + \epsilon\right)$

---

return θ_t

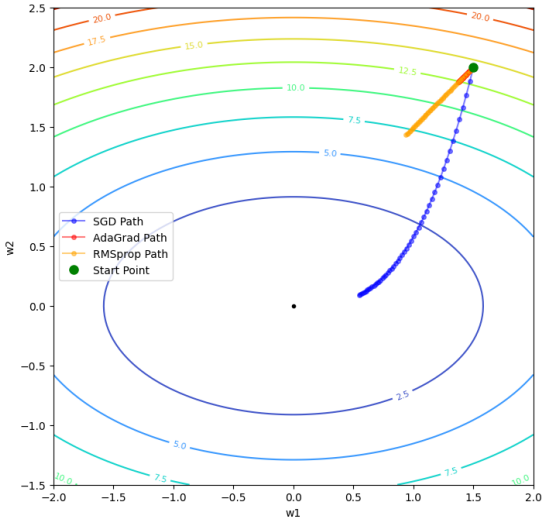
---

https://pytorch.org/docs/stable/generated/torch.optim.RMSprop.html#torch.optim.RMSprop

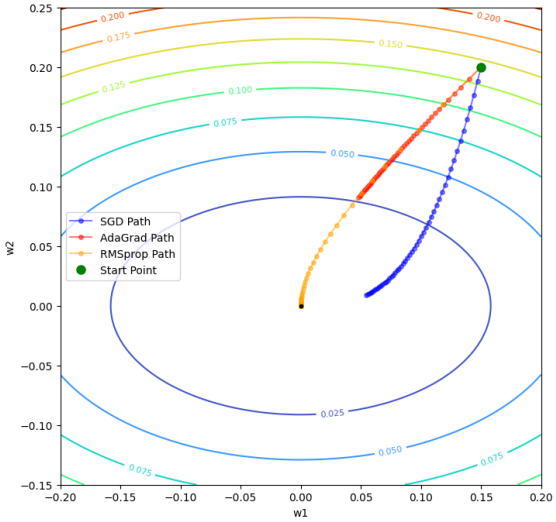
RMSprop

RMSprop VS AdaGrad VS SGD

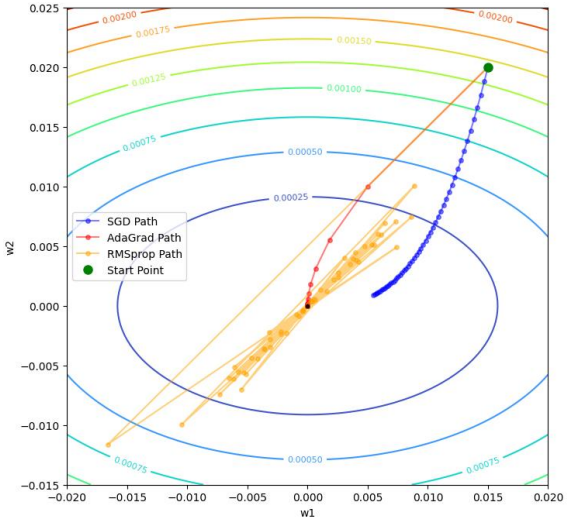
- $\alpha = 0.01, iter = 50, \varepsilon = 1e-8$



$\boldsymbol{w}_{init} = (1.5, 2.0)$



$\boldsymbol{w}_{init} = (0.15, 0.2)$



$\boldsymbol{w}_{init} = (0.015, 0.02)$

→ Faster than AdaGrad!

Can oscillate around optimal point
(especially, sharp local minima)

RMSprop

Limitation

- Lacks the speed and stability benefits of momentum-based methods, which help reduce oscillations, especially in narrow ravines

Adam

Adam (Adaptive Moment Estimation)

- Designed to combine the advantages of **Momentum** (for speed and stability) and **RMSprop** (for adaptive learning rates).

- Aim: To provide an efficient, adaptive, and robust optimizer that performs well across a wide range of tasks.

- Reference

  - Kingma, Diederik P. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

Adam

Foundation

- **Momentum**
  - Accumulates past gradients to improve speed and reduce oscillations + avoid local minima (utilizing first moment)

$$v_t = \gamma v_{t-1} + \alpha \frac{\partial L}{\partial w}$$
$$w_{t+1} \leftarrow w_t - v_t$$

- **RMSprop**
  - Adjusts learning rates adaptively based on the moving average of squared gradients (utilizing second moment)

$$h_t = \gamma h_{t-1} + (1-\gamma) g_t^2$$
$$w_{t+1} \leftarrow w_t - \frac{\alpha}{\sqrt{h_t + \varepsilon}} \cdot g_t$$

Adam

Formulation

- **First moment (m_t)**

$$m_0 = 0$$
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

  - β_1: decay rate for first moment (default 0.9)

- **Second moment (v_t)**

$$v_0 = 0$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

  - β_2: decay rate for second moment (default 0.999)

Adam

Formulation

- **First moment (m_t)**

$$m_0 = 0$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

  - β_1: decay rate for first moment (default 0.9)

We hope ratio of g_t to be 0.1, and past moment to be 0.9

But,
$m_1 = 0 + 0.1g_1 \rightarrow 0.1g_1$
$m_2 = 0.9 \cdot (0 + 0.1g_1) + 0.1g_2$
$\rightarrow past: g_2 = 9:10$
$m_3 = \cdots$
→ not ideal

- **Second moment (v_t)**

$$v_0 = 0$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

  - β_2: decay rate for second moment (default 0.999)

We hope ratio of g_t^2 to be 0.001, and past moment to be 0.999

But,
$v_1 = 0 + 0.001g_1^2 \rightarrow 0.001g_1^2$
v_2
$= 0.999 \cdot (0 + 0.1g_1) + 0.001g_2^2$
$\rightarrow past: g_2^2 = 999:10$
$v_3 = \cdots$
→ not ideal

Adam

Formulation

- **Bias correction**

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- At the first time step, this correction makes $\hat{m}_1 = m_1$ and $\hat{v}_1 = v_1$, ensuring the initial estimates reflect the true gradient and variance.
- The bias correction brings $\hat{m}_t$, $\hat{v}_t$ closer to their "ideal" values, allowing the optimizer to be more accurate in adjusting learning rates from the start.
- The effect of bias correction exponentially decreases as t grows, making the correction factor nearly unnecessary in later steps.

Adam

Adam

- **Weight update**

$$w_{t+1} \leftarrow w_t - \frac{\alpha}{\sqrt{\hat{v}_t + \varepsilon}} \hat{m}_t$$

Where,

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$m_0 = 0, v_0 = 0$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Adam

Adam

- **Weight update**

$$w_{t+1} \leftarrow w_t - \frac{\alpha}{\sqrt{\hat{v}_t + \varepsilon}} \hat{m}_t$$

Where,

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$ → For momentum

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$ → For adaptive learning rate

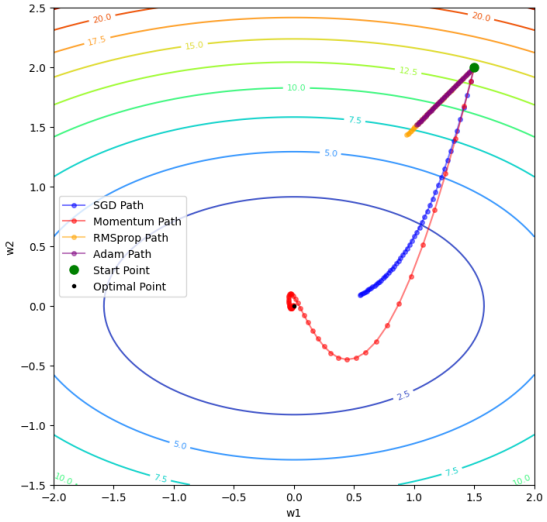
$$m_0 = 0, v_0 = 0$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

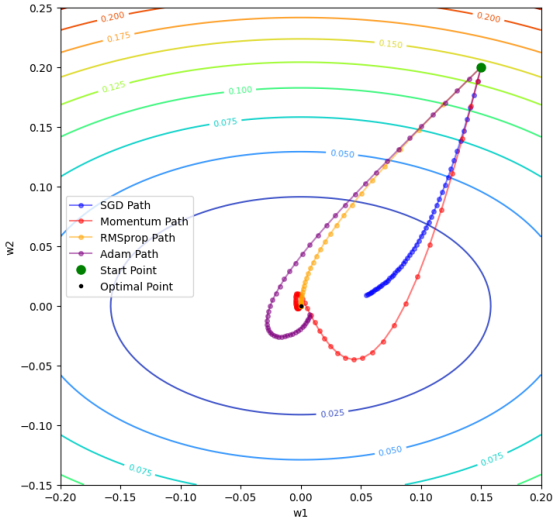
Adam

SGD vs Momentum vs RMSprop vs Adam

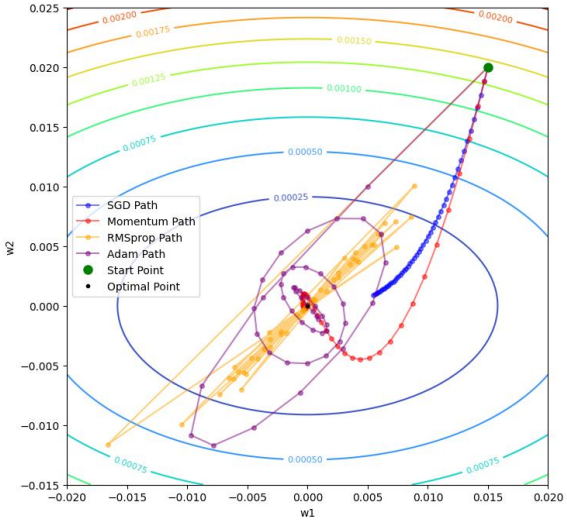
- $\alpha = 0.01, iter = 50, \varepsilon = 1e-8$



$\boldsymbol{w}_{init} = (1.5, 2.0)$



$\boldsymbol{w}_{init} = (0.15, 0.2)$



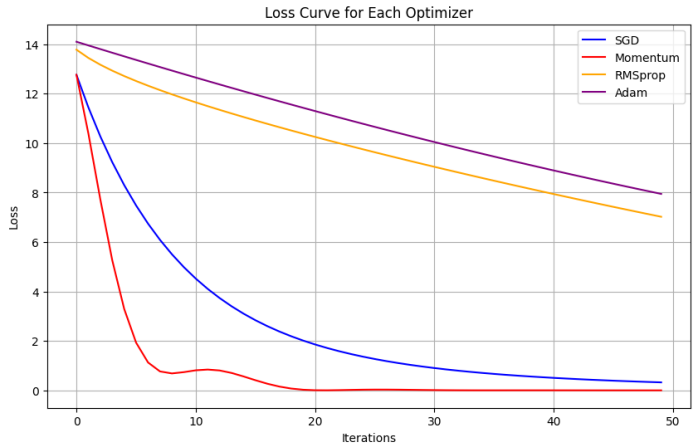
$\boldsymbol{w}_{init} = (0.015, 0.02)$

Adam Stable then RMS prop

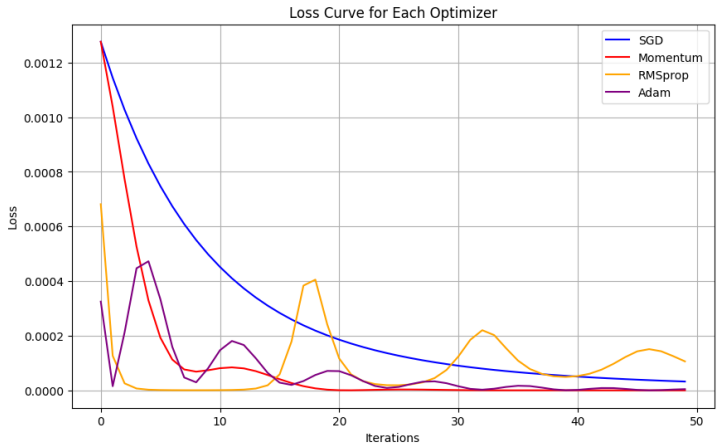
Note. Appropriate learning rate and epochs differ between optimizers

Adam

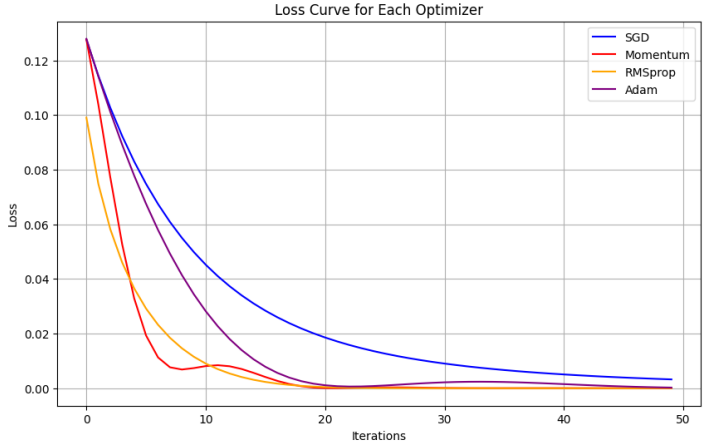
SGD vs Momentum vs RMSprop vs Adam



$\boldsymbol{w}_{init} = (1.5, 2.0)$



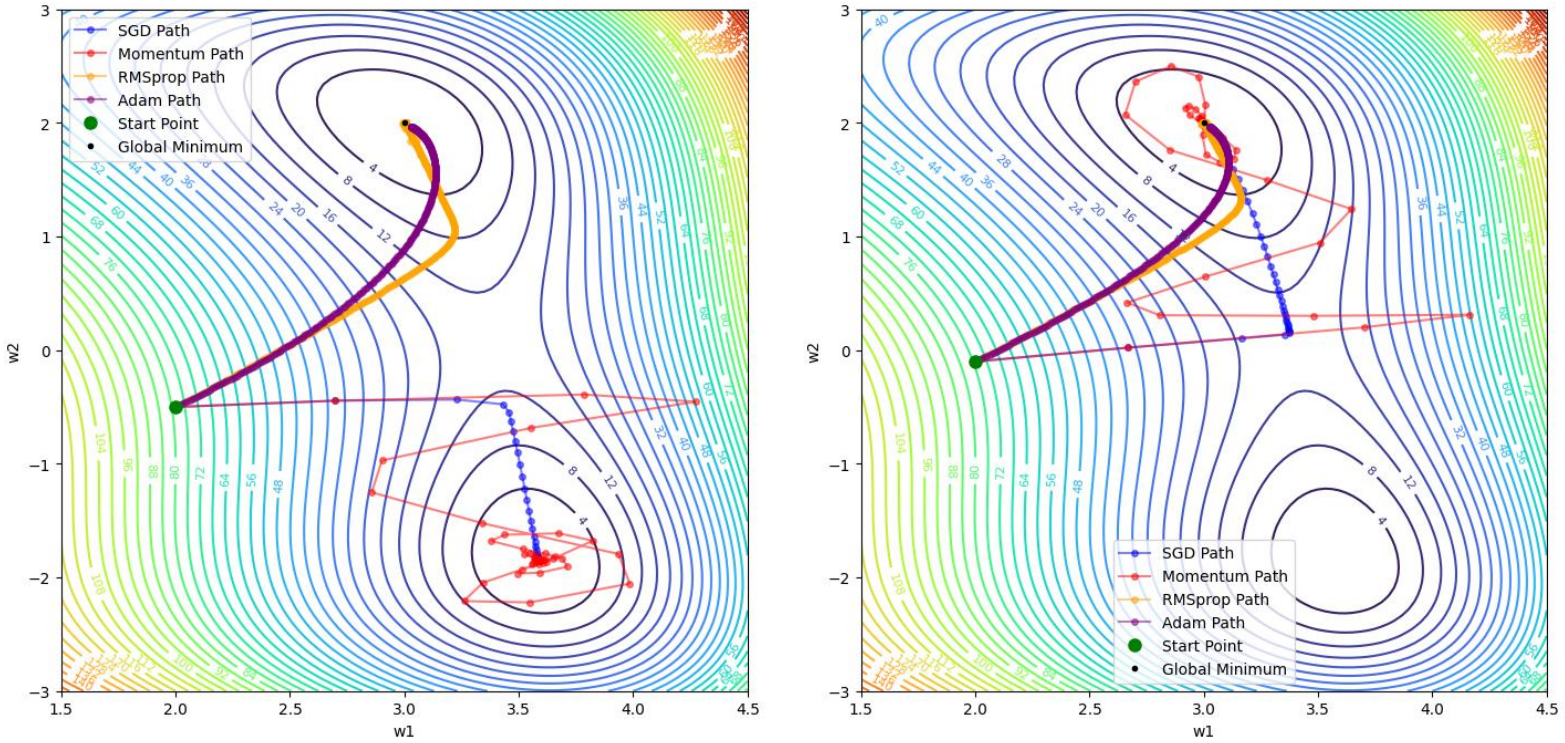
$\boldsymbol{w}_{init} = (0.015, 0.02)$



$\boldsymbol{w}_{init} = (0.15, 0.2)$

Adam

Different end point according to start point



Python implementation

Adagrad

- https://docs.pytorch.org/docs/stable/generated/torch.optim.Adagrad.html

```
class SoftmaxLinear(nn.Module):
    def __init__(self, input_dim=28*28, num_classes=10):
        super().__init__()
        self.fc = nn.Linear(input_dim, num_classes)

    def forward(self, x):
        logits = self.fc(x)
        return logits

model = SoftmaxLinear().to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adagrad(model.parameters(), lr=0.1)

for epoch in range(0, 100):
    model.train()
    total_loss = 0.0

    for x, y in train_loader:
        x, y = x.to(device), y.to(device)

        optimizer.zero_grad()
        logits = model(x)
        loss = criterion(logits, y)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()


    print(f"Epoch [{epoch+1}] Loss: {total_loss/len(train_loader):.4f}")
```

Python implementation

RMSprop

- https://docs.pytorch.org/docs/stable/generated/torch.optim.RMSprop.html

```
class SoftmaxLinear(nn.Module):
    def __init__(self, input_dim=28*28, num_classes=10):
        super().__init__()
        self.fc = nn.Linear(input_dim, num_classes)

    def forward(self, x):
        logits = self.fc(x)
        return logits

model = SoftmaxLinear().to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.RMSprop(model.parameters(), lr=0.1)

for epoch in range(0, 100):
    model.train()
    total_loss = 0.0

    for x, y in train_loader:
        x, y = x.to(device), y.to(device)

        optimizer.zero_grad()
        logits = model(x)
        loss = criterion(logits, y)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()


    print(f"Epoch [{epoch+1}] Loss: {total_loss/len(train_loader):.4f}")
```

Python implementation

Adam

- https://docs.pytorch.org/docs/stable/generated/torch.optim.Adam.html

```
class SoftmaxLinear(nn.Module):
    def __init__(self, input_dim=28*28, num_classes=10):
        super().__init__()
        self.fc = nn.Linear(input_dim, num_classes)

    def forward(self, x):
        logits = self.fc(x)
        return logits

model = SoftmaxLinear().to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.1)

for epoch in range(0, 100):
    model.train()
    total_loss = 0.0

    for x, y in train_loader:
        x, y = x.to(device), y.to(device)

        optimizer.zero_grad()
        logits = model(x)
        loss = criterion(logits, y)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()


    print(f"Epoch [{epoch+1}] Loss: {total_loss/len(train_loader):.4f}")
```

Learning curve

Where is optimal point?