Artificial Neural Networks

Course-6

Gavrilut Dragos

AGENDA FOR TODAY

- Basic Statistical Notions
- Optimizations (overview)
- Weight initialization
- Loss Functions
- Computation Graph
- Backpropagation in Pytorch
- Stochastic Gradient Descent

Basic Statistical Notions

Basic Statistical Notions

Let's discuss some common statistical notions:

Mean

Given a vector v = [v_1, v_2, … v_n] , then the mean of v (often denoted by the letter: μ is):

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}, n > 0, mean(v) = \mu = \frac{\sum_{i=1}^{n}(v_i)}{n}$$

Example:

$$v = \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}, mean(v) = \mu = \frac{10 + 20 + 30}{3} = \frac{60}{3} = 20$$

Basic Statistical Notions

Let's discuss some common statistical notions:

Standard deviation

Given a vector v = [v_1, v_2, … v_n] , then the standard deviation for v (often denoted by the letter: σ is):

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}, n > 0, stdev(v) = \sigma = \sqrt{\frac{1}{n} \times \sum_{i=1}^{n} (v_i - \mu)^2}, \mu = \frac{\sum_{i=1}^{n}(v_i)}{n}$$

Basic Statistical Notions

Let's discuss some common statistical notions:

Standard deviation (example)

$$v = \begin{bmatrix} 2 \\ 4 \\ 5 \\ 6 \\ 8 \end{bmatrix}, \mu = \frac{2+4+5+6+8}{5} = \frac{25}{5} = 5$$

$$stdev(v) = \sigma = \sqrt{\frac{1}{n} \times \sum_{i=1}^{n} (v_i - \mu)^2} = \sqrt{\frac{1}{5} \times \left((2-5)^2 + (4-5)^2 + (5-5)^2 + (6-5)^2 + (8-5)^2\right)}$$

$$= \sqrt{\frac{1}{5} \times \left((-3)^2 + (-1)^2 + (0)^2 + (1)^2 + (3)^2\right)} = \sqrt{\frac{1}{5} \times (9+1+0+9+1)} = \sqrt{\frac{20}{5}} = \sqrt{4} = 2$$

Basic Statistical Notions

Let's discuss some common statistical notions:

Standard deviation (Bessel's correction)

Given a vector v = [v_1, v_2, … v_n] , then the standard deviation for v (often denoted by the letter: σ is):

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}, n > 1, stdev(v) = \sigma = \sqrt{\frac{1}{n-1} \times \sum_{i=1}^{n} (v_i - \mu)^2}, \mu = \frac{\sum_{i=1}^{n}(v_i)}{n}$$

Bessel's correction uses "n-1" instead of "n" as a divider for the standard deviation formula.

Basic Statistical Notions

Let's discuss some common statistical notions:

Variance

Given a vector v = [v_1, v_2, … v_n] , then the variance for that vector is:

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}, variance(v) = \sigma^2, where\ \sigma = standard\ deviation$$

The same formula applies if we consider σ computed with Bessel correction.

Optimizations (overview)

Optimizations (overview)

Optimizations are crucial for improving the performance, speed, cost-effectiveness, and accessibility of neural networks, making them more practical and effective for a wide range of applications.

However, the term optimization can mean different things when it come to neuronal networks.

Optimizations (overview)

However, the term optimization can mean different things when it come to neural networks:

1. **Performance**: Optimizations help improve the accuracy and efficiency of neural networks. By refining the network architecture or training process, models can learn better representations of the data, leading to improved predictions or classifications.
2. **Speed**: A well-optimized neural network can process information faster. This is in particular relevant for applications that require real-time analysis, such as autonomous vehicles or high-frequency trading systems.

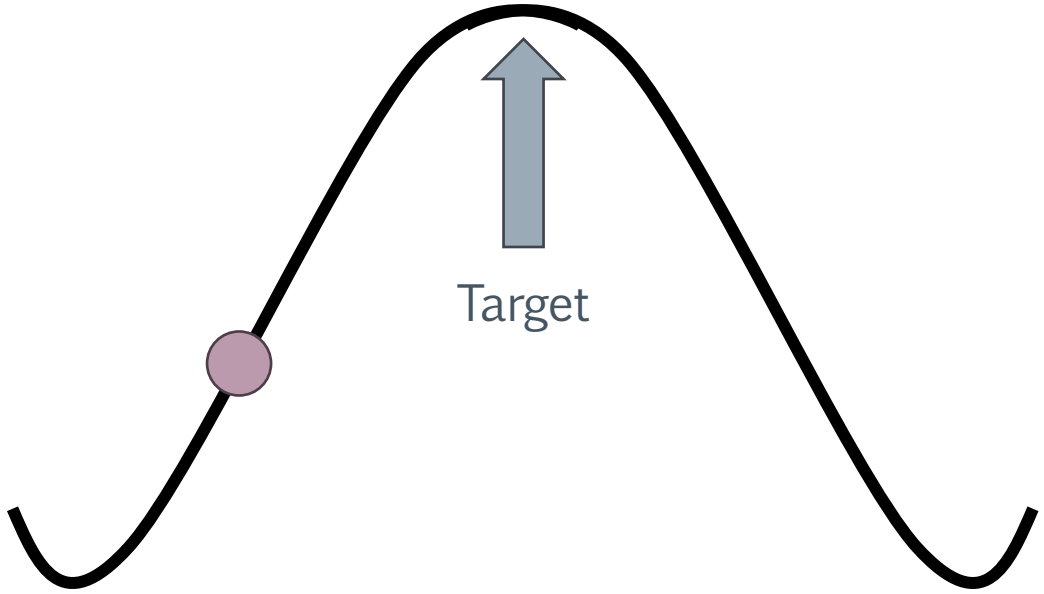
Optimizations (overview)

However, the term optimization can mean different things when it come to neuronal networks:

3. **Resource Utilization:** Optimizations can reduce the computational resources required, making it possible to run on devices with limited processing power, such as smartphones or embedded systems.
4. **Cost:** Building and using a neuronal network comes with a cost. Optimization can reduce this cost making a neuronal network more feasible to use in practice.
5. **Accessibility**: By making neural networks more efficient, optimizations can make advanced AI technologies more accessible to organizations with limited resources.

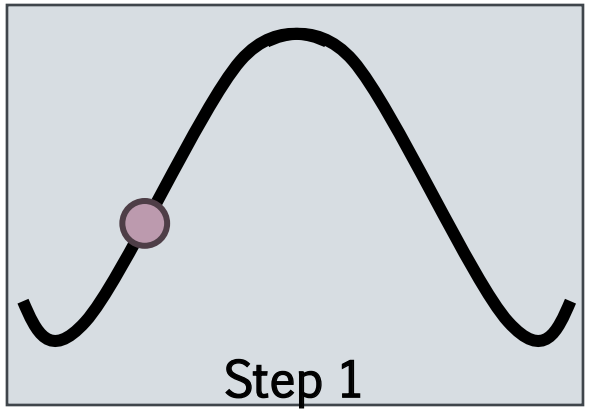
Optimizations (overview)

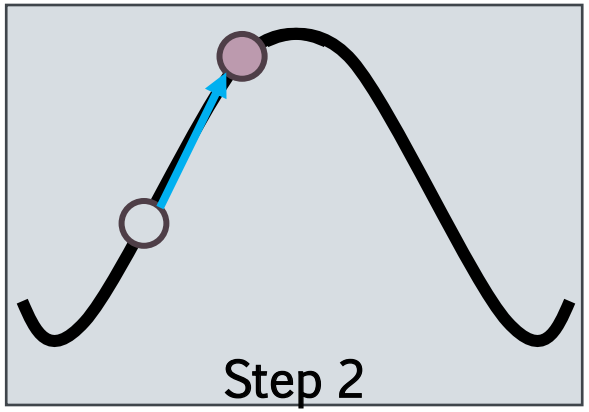
- Let's analyze the following case (to better understand the need of optimizations). Our task is to move the pink circle to the top, but we can only move with α centimeters per round.

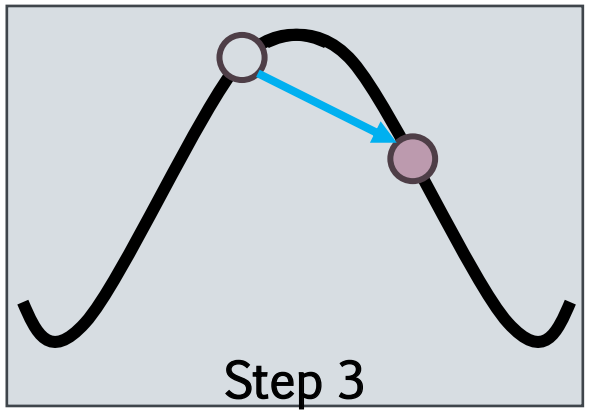


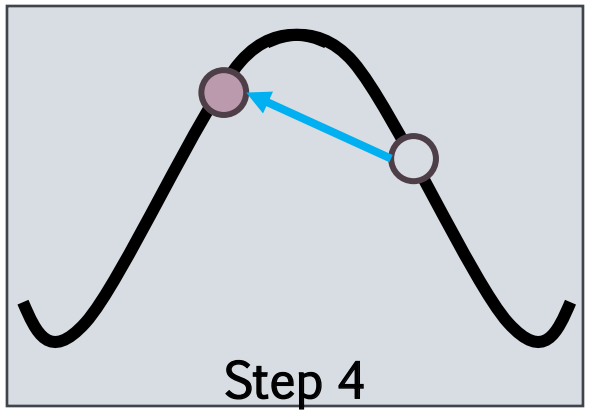
Optimizations (overview)

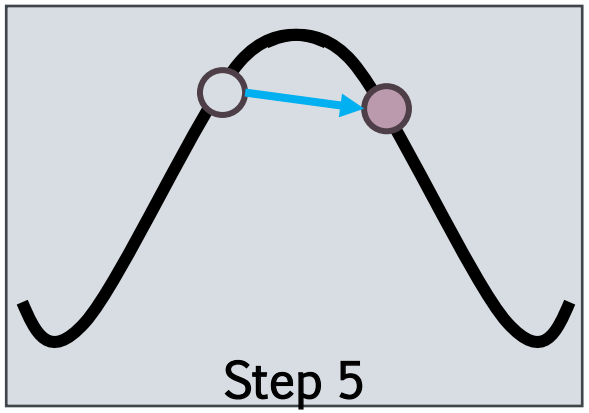
› α=a larger value (steps are big).

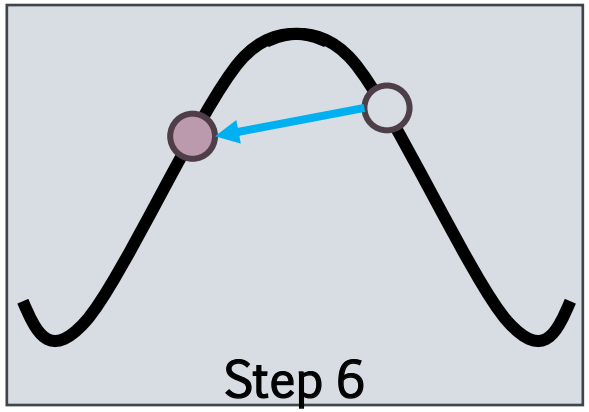






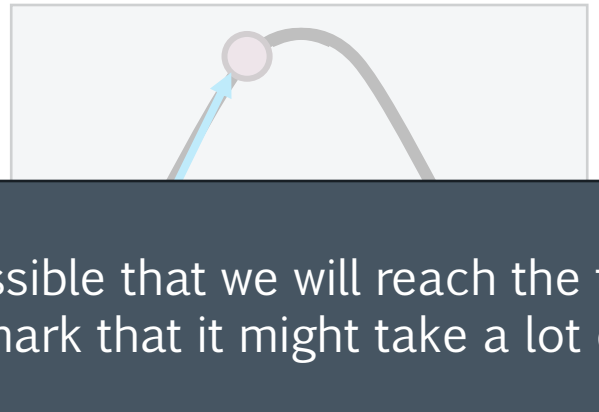




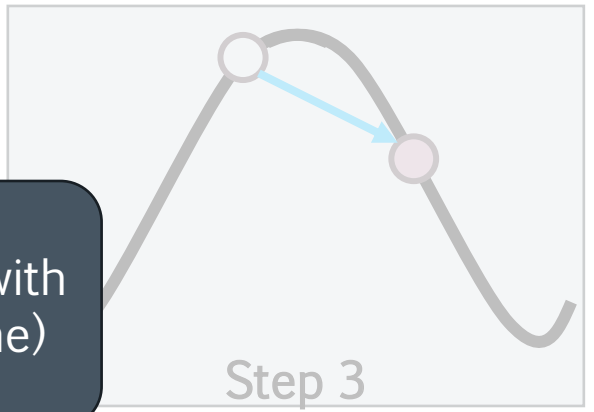


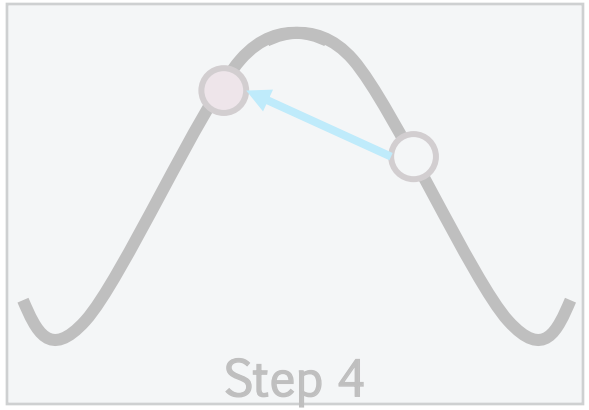
Optimizations (overview)

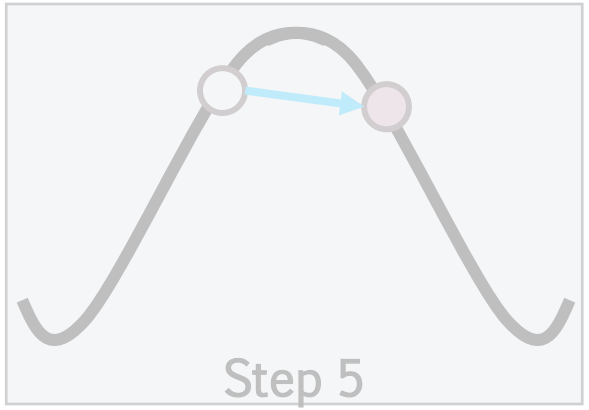
› α=a larger value (steps are big).

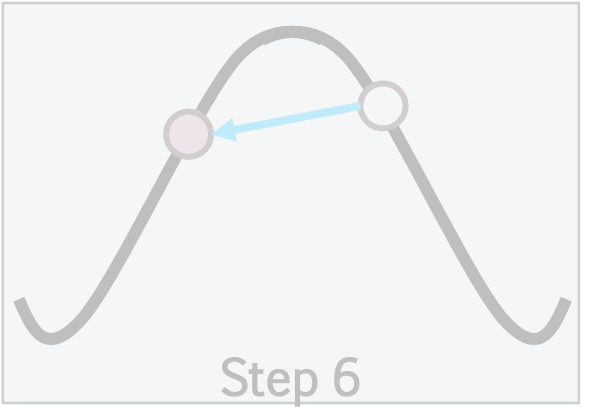


It is possible that we will reach the top (with the remark that it might take a lot of time)



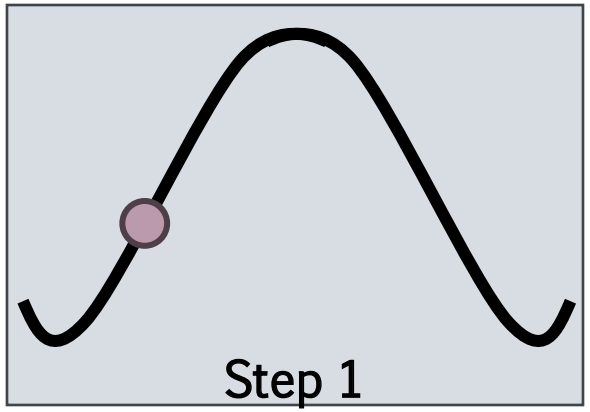


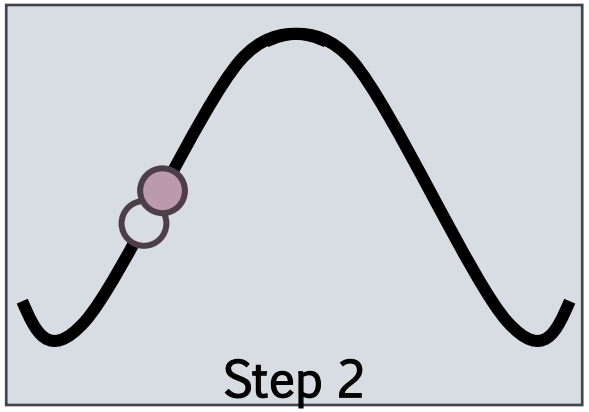


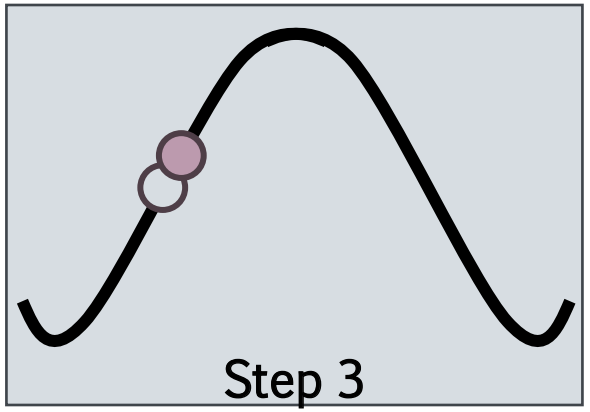


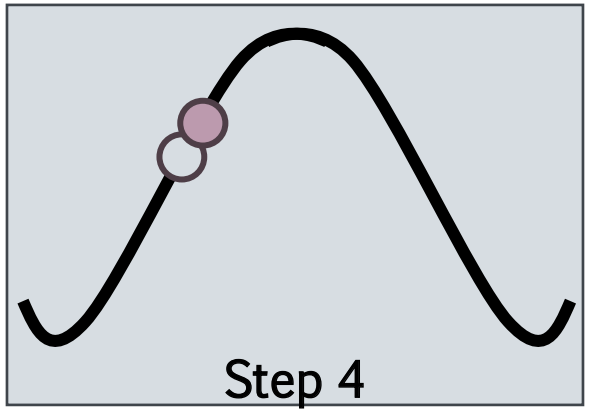
Optimizations (overview)

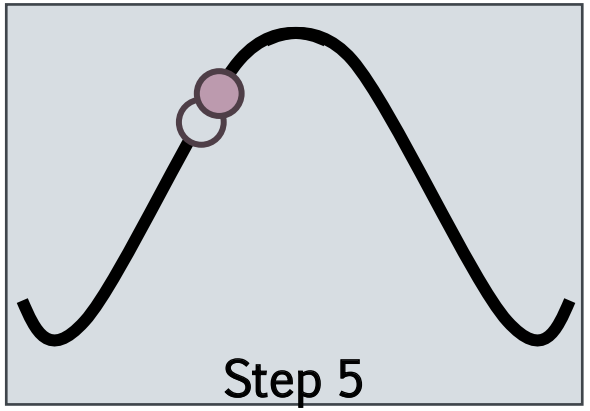
› α=a small value (steps are small).

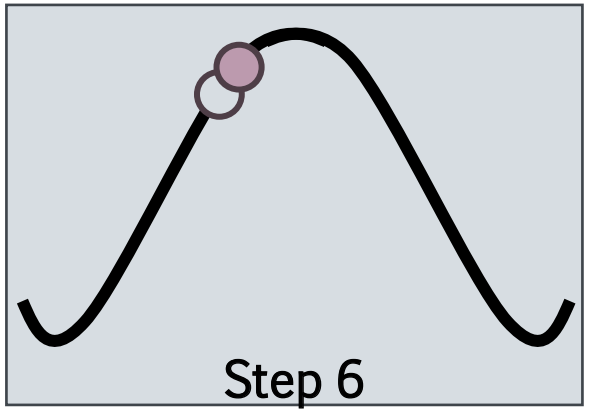






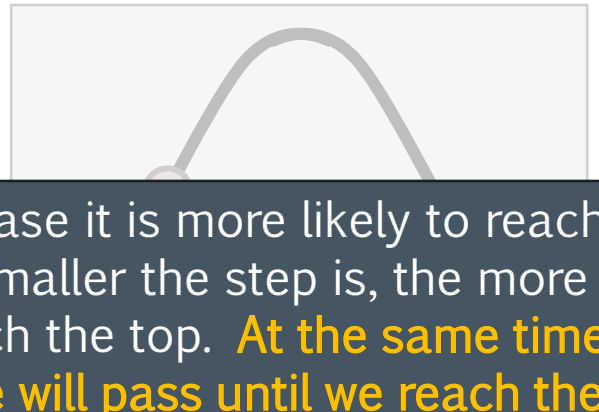




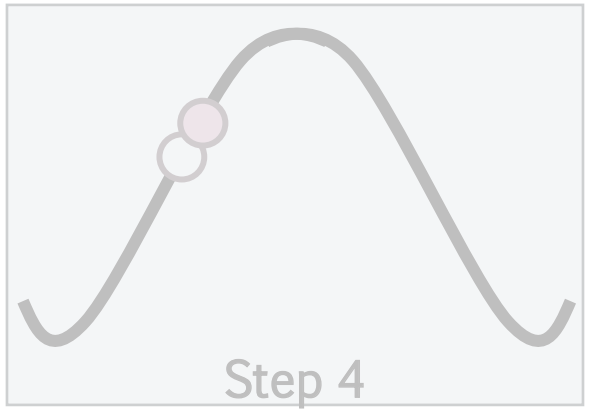


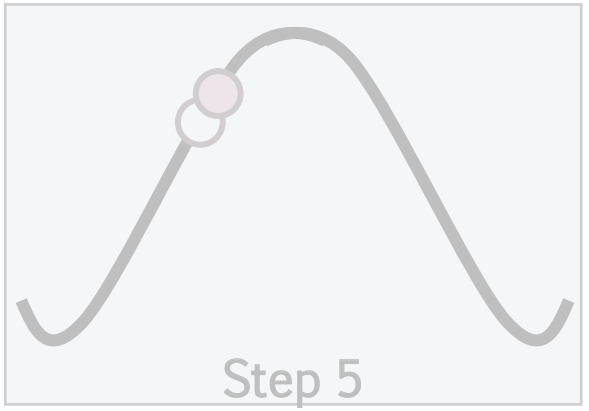
Optimizations (overview)

› α=a small value (steps are small).



In this case it is more likely to reach the top. And the smaller the step is, the more probable it is to reach the top. At the same time the more time will pass until we reach the top.

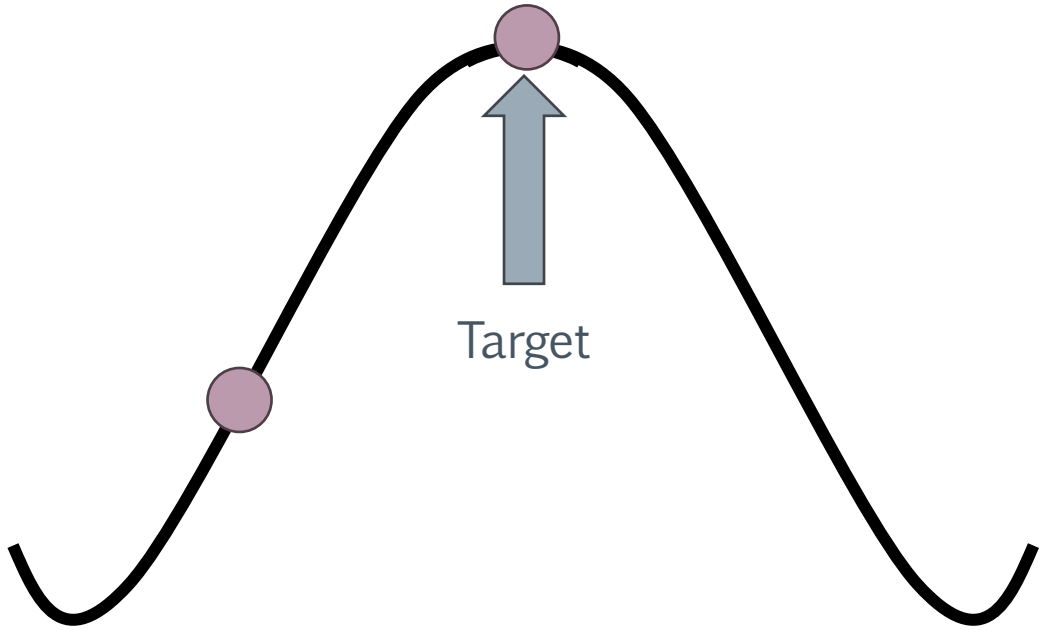




Optimizations (overview)

› So … the actual question is:

How do we find the BEST step that will lead us to the top in the shortest amount of time ?



Optimizations (overview)

› **So, what can we optimize at a neuronal network ?**

1. Weight Initialization
2. Gradient Descent Variants
3. Regularization Techniques
4. Learning Rate Schedules
5. Activation and Loss functions
6. Hyperparameter Optimization
7. Quantization and Pruning

Weight initialization

Weight Initialization

Weight initialization is an important step in training neural networks as it involves setting the initial values of the weights (and biases) before the training process begins.

Proper weight initialization can significantly impact the efficiency and convergence of the training process.

Weight Initialization

Why is weight initialization relevant:

- **Avoiding Symmetry:** If all weights are initialized with the same value, then all neurons will learn the same features during training. Proper initialization breaks this symmetry.
- **Preventing Vanishing/Exploding Gradients**: If weights are too small, gradients might become too small (vanish) as they propagate back through the network, making learning very slow or stagnant. If weights are too large, gradients can become too large (explode), leading to erratic updates and divergence.
- **Speeding Up Convergence**: Good initial weights can place the network in a region of the parameter space where it learns more quickly, speeding up convergence.

Weight Initialization

- As a general concept, in Pytorch, weight initialization is done in the `__init__` method of a neuronal network class:

```
import torch
import torch.nn as nn

class MyNet(nn.Module):
    def __init__(self):
        super(MyNet, self).__init__()
        self.fc1 = nn.Linear(10, 5)
        self.fc2 = nn.Linear(5, 2)
        # Add weight initialization here

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
net = MyNet()
```

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Uniform distribution

PyTorch class: torch.nn.init.uniform_
Pytorch definition: uniform_(tensor,a=0,b=1)

A type of initialization where every value from a to b has the same probability of being drawn: $P(x|a,b) = \frac{1}{b-a}, a \leq x \leq b$. Other characteristics:

- The mean of this type of distribution is close to $\frac{a+b}{2}$
- The variance of this type of distribution is close to $\frac{(b-a)^2}{12}$
- The standard deviation of this type of distribution is close to $\sqrt{\frac{(b-a)^2}{12}}$

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Uniform distribution

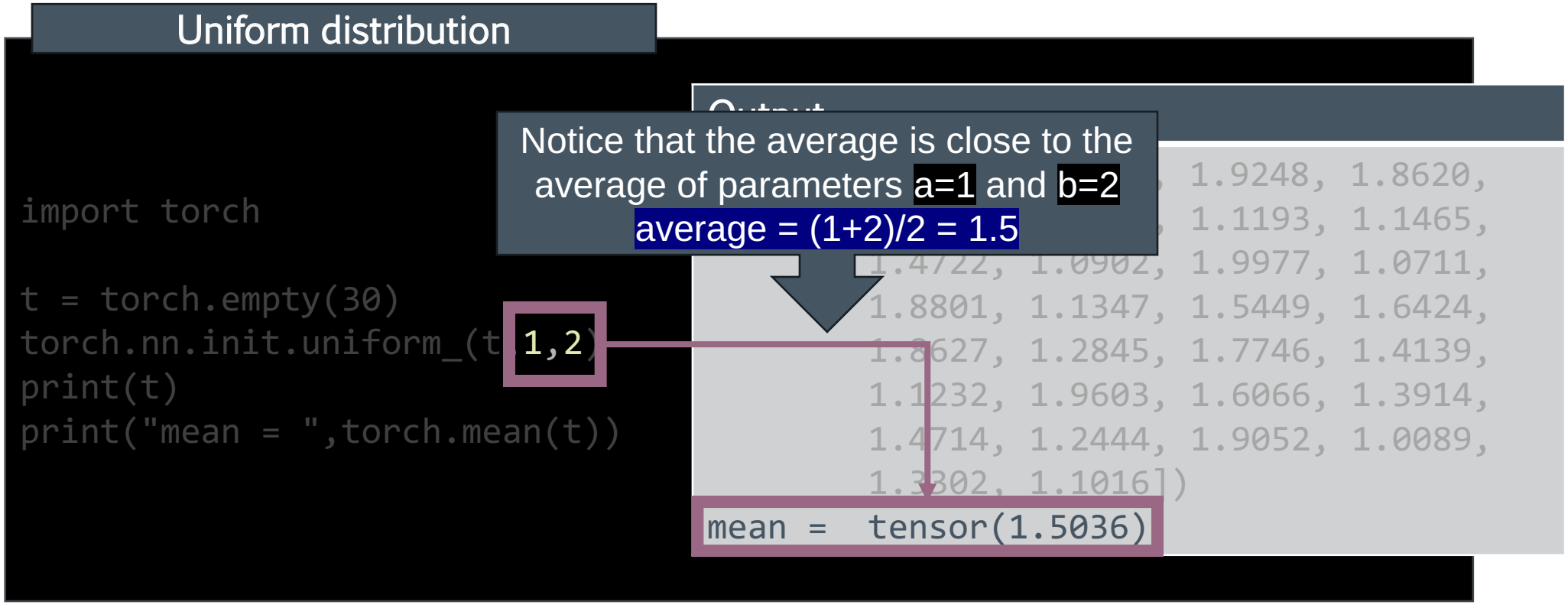
```
import torch

t = torch.empty(30)
torch.nn.init.uniform_(t,1,2)
print(t)
print("mean = ",torch.mean(t))
```

Output

```
tensor([1.9696, 1.7306, 1.9248, 1.8620,
        1.2054, 1.8369, 1.1193, 1.1465,
        1.4722, 1.0902, 1.9977, 1.0711,
        1.8801, 1.1347, 1.5449, 1.6424,
        1.8627, 1.2845, 1.7746, 1.4139,
        1.1232, 1.9603, 1.6066, 1.3914,
        1.4714, 1.2444, 1.9052, 1.0089,
        1.3302, 1.1016])
mean =  tensor(1.5036)
```

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

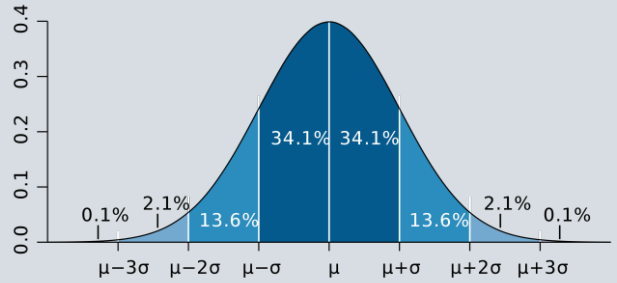


Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Normal distribution

PyTorch class: torch.nn.init.normal_
Pytorch definition: normal_(tensor,mean=0.0,std=1.0)



A type of initialization (bell shaped) where the probability for a value to be extracted is:

$$P(x|\mu,\sigma) = \frac{1}{\sqrt{2 \times \pi \times \sigma^2}} \times e^{-\frac{(x-\mu)^2}{2\times\sigma^2}},$$

$$where\ \mu = mean, \sigma = standard\ deviation\ and\ \pi \cong 3.14$$

* Image from https://en.wikipedia.org/wiki/Normal_distribution

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Normal distribution

```
import torch

t = torch.empty(30)
torch.nn.init.normal_(t,1,2)
print(t)
print("mean = ",torch.mean(t))
```

Output

```
tensor([ 0.6243,  2.1053,  0.8221,  4.2966,
         1.7287,  2.6475,  0.7444,  1.3841,
         0.0950,  3.2870,  1.3175,  1.4677,
        -2.1269,  0.3023,  0.3121, -1.3098,
         1.5830,  3.7458, -1.2519,  0.3864,
         2.1482,  1.9378,  2.0775,  1.1237,
         0.6475, -2.3913,  2.5435,  1.9884,
        -0.5488, -0.6369])
median =  tensor(1.0350)
```

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Normal distribution

```
import torch

t = torch.empty(30)
torch.nn.init.normal_(t,1,2)
print(t)
print("mean = ",torch.mean(t))
```

Output

```
tensor([ 0.6243,  2.1053,  0.8221,  4.2966,
         0.7444,  1.3841,
         1.3175,  1.4677,
        -2.1269,  0.3023,  0.3121, -1.3098,
         1.5830,  3.7458, -1.2519,  0.3864,
         2.1482,  1.9378,  2.0775,  1.1237,
         0.6475, -2.3913,  2.5435,  1.9884,
        -0.5488, -0.6369])
median =  tensor(1.0350)
```

Notice that the average is close to the provided mean: 1

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Normal (truncated) distribution

PyTorch class: torch.nn.init.trunc_normal_
Pytorch definition: trunc_normal_(tensor,mean=0.0,std=1.0,a=-2,b=2)

This is a normal distribution where we make sure that every element is within a and b.
The logic behind this is that we extract a value base on a normal distribution, then we check [a..b] interval and if that value is not within the interval, we extract again.
This method works best if $a \leq mean \leq b$

Weight Initialization

- Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Normal (truncated) distribution

```
import torch

t = torch.empty(30)
torch.nn.init.trunc_normal_(t,1,2,0.5,1.5)
print(t)
```

Output

```
tensor([1.1023, 0.5422, 1.0869, 0.9193, 0.8733, 1.0830, 0.8314, 1.4435, 1.1601,
        1.1981, 0.7829, 0.6048, 0.8016, 0.7796, 1.1722, 1.4447, 0.6168, 1.4766,
        0.9913, 1.4145, 1.1465, 0.8821, 0.7472, 1.2457, 0.5646, 1.0388, 0.5824,
        1.3114, 0.7116, 1.0906])
```

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Normal (truncated) distribution

```
import torch

t = torch.empty(30)
torch.nn.init.trunc_normal_(t,1,2,0.5,1.5)
print(t)
```

In this case we requested a distribution with the mean=1, a standard deviation=2 and all elements should be in the interval [0.5..1.5]

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Xavier (Glorot) uniform distribution

PyTorch class: torch.nn.init.xavier_uniform_
Pytorch definition: xavier_uniform_(tensor, gain=1.0)

This is a uniform distribution, where “a” (inferior limit) and “b” (superior limit) are computed in the following way:

$$a = -gain \times \sqrt{\frac{6}{fan_{in} + fan_{out}}}, b = -a,$$

$$fan_{in} = number\ of\ input\ units\ from\ a\ layer,$$
$$fan_{out} = number\ of\ output\ units\ from\ a\ layer,$$
$$gain = a\ scaling\ factor$$

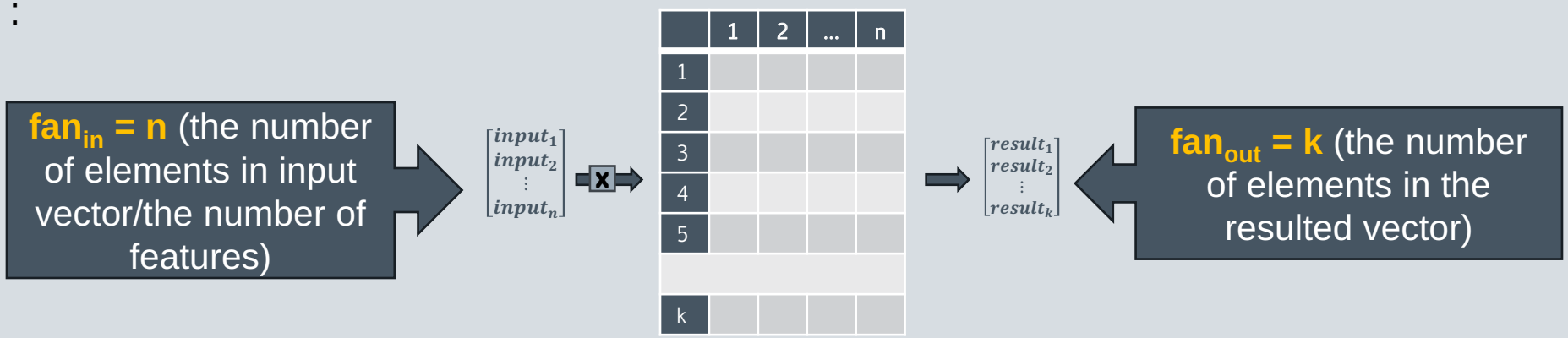
Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Xavier (Glorot) uniform distribution

PyTorch class: torch.nn.init.xavier_uniform_
Pytorch definition: xavier_uniform_(tensor, gain=1.0)

To get a better understanding on what fan_{in} and fan_{out} are lets analyze this image.
:



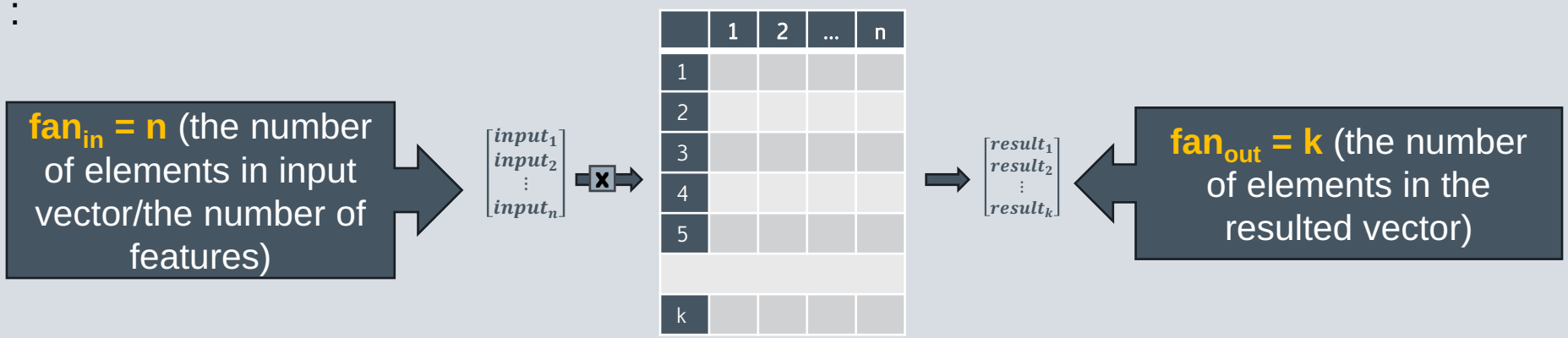
Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Xavier (Glorot) uniform distribution

PyTorch class: torch.nn.init.xavier_uniform_
Pytorch definition: xavier_uniform_(tensor, gain=1.0)

To get a better understanding on what fan_{in} and fan_{out} are lets analyze this image.
:



Weight Initialization

- Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Xavier (Glorot) uniform distribution

PyTorch class: torch.nn.init.xavier_uniform_
Pytorch definition: xavier_uniform_(tensor, gain=1.0)

The choice of this particular range for the uniform distribution is based on maintaining a balance between the variance of the inputs and the variance of the outputs. The factor of 6 in the numerator comes from a derivation involving the variance of the weights and the assumption of a linear activation function.

When using nonlinear activations like **sigmoid** or **tanh**, this initialization allows for gradients to flow more effectively during the initial stages of training. It helps in achieving a faster and more stable convergence.

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Xavier (Glorot) uniform distribution

```
import torch

t = torch.empty(4,3)
torch.nn.init.xavier_normal_(t)
print(t)
```

Output

```
tensor([[ 0.3323,  0.7760, -0.8714],
        [-0.5664,  0.4431, -0.9121],
        [ 0.8657,  0.0991,  0.8273],
        [ 0.7294, -0.6867,  0.6926]])
```

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Xavier (Glorot) uniform distribution

```
import torch

t = torch.empty(10)
torch.nn.init.xavier_normal_(t)
print(t)
```

Notice that a tensor must have **2 dimensions** to be used with this initialization method.

Error

```
Traceback (most recent call last):
  File "e:\Lucru\RN\teste\a.py", line 4, in <module>
    torch.nn.init.xavier_uniform_(t)
.....
    raise ValueError("Fan in and fan out can not be computed for tensor with fewer than 2 dimensions")
ValueError: Fan in and fan out can not be computed for tensor with fewer than 2 dimensions
```

Weight Initialization

- Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Xavier (Glorot) uniform distribution

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

layer = nn.Linear(10, 5)
init.xavier_uniform_(layer.weight)
```

- This is the most common usage of Xavier initialization (directly with a linear layer)

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Xavier (Glorot) normal distribution

PyTorch class: torch.nn.init.xavier_normal_
Pytorch definition: xavier_normal_(tensor, gain=1.0)

This is a normal distribution, where “mean” is set to 0, and the standard distribution (σ) is computed in the following way:

$$standard\ distribution = \ \sigma = gain \times \sqrt{\frac{2}{fan_{in} + fan_{out}}}$$

$$fan_{in} = number\ of\ input\ units\ from\ a\ layer,$$
$$fan_{out} = number\ of\ output\ units\ from\ a\ layer,$$
$$gain = a\ scaling\ factor$$

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Xavier (Glorot) normal distribution

PyTorch class: torch.nn.init.xavier_normal_
Pytorch definition: xavier_normal_(tensor, gain=1.0)

In practice, Xavier normal initialization is often chosen for layers in a network that:

- Use **tanh** or **sigmoid** activation functions
- As part of a deep network, where maintaining consistent variance across layers is important for effective gradient propagation

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Xavier (Glorot) normal distribution

```
import torch

t = torch.empty(4,3)
torch.nn.init.xavier_normal_(t)
print(t)
```

Output

```
tensor([[ 0.9750,  0.5690, -0.9753],
        [ 0.5845,  0.5841,  0.4461],
        [-0.1557,  0.7241, -0.3156],
        [-0.3469, -0.4621,  0.4724]])
```

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Kaiming (He) uniform distribution

PyTorch class: torch.nn.init.kaiming_uniform_
Pytorch definition: kaiming_uniform_(tensor, a=0.0, fan_{mode}="fan_in")

This is a uniform distribution, where "a" (inferior limit) and "b" (superior limit) are computed in the following way:

$$a = -gain \times \sqrt{\frac{3}{fan_{mode}}}, b = -a,$$

$$fan_{mode} = can\ be\ either\ fan_{in}\ or\ fan_{out}$$

$$gain = a\ constant\ computed\ base\ on\ the\ activation\ function\ used$$

See https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.calculate_gain for more details.

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Kaiming (He) uniform distribution

PyTorch class: torch.nn.init.kaiming_uniform_
Pytorch definition: kaiming_uniform_(tensor, a=0.0, fan_{mode}="fan_in")

ReLU activation functions have the property that they output zero for any negative input. This characteristic can potentially lead to "dead neurons" during training where, if a neuron's weights are initialized poorly, it may only output zeros and, consequently, never update its weights during backpropagation because the gradient will be zero.

The idea behind He initialization is to initialize the weights so that the variance of the activations remains the same across every layer. This helps avoid the gradients from exploding or vanishing during the training process when using ReLU activations.

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Kaiming (He) uniform distribution

PyTorch class: torch.nn.init.kaiming_uniform_
Pytorch definition: kaiming_uniform_(tensor, a=0.0, fan_{mode}="fan_in")

When to Use Kaiming Initialization:

- For layers before a ReLU activation (or its variants).
- In deep networks to prevent the vanishing/exploding gradient problem.
- When training models that need to converge faster and more reliably.

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Kaiming (He) uniform distribution

```
import torch

t = torch.empty(4,3)
torch.nn.init.kaiming_uniform_(t)
print(t)
```

Output

```
tensor([[ 0.9508,  0.1249,  0.5104],
        [-0.5183,  0.1223,  1.1484],
        [-1.1301, -0.3429,  0.6953],
        [ 0.5335,  1.3128, -1.1175]])
```

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Kaiming (He) normal distribution

PyTorch class: torch.nn.init.kaiming_normal_
Pytorch definition: kaiming_normal_(tensor, a=0.0, fan_{mode}="fan_in")

This is a normal distribution, where "mean" is set to 0, and the standard distribution (σ) is computed in the following way:

$$standard\ distribution = \ \sigma = \frac{gain}{\sqrt{fan_{mode}}}$$

$$fan_{mode} = can\ be\ either\ fan_{in}\ or\ fan_{out}$$
$$gain = a\ constant\ computed\ base\ on\ the\ activation\ function\ used$$

See https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.calculate_gain for more details.

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Kaiming (He) normal distribution

PyTorch class: torch.nn.init.kaiming_normal_
Pytorch definition: kaiming_normal_(tensor, a=0.0, fan_{mode}="fan_in")

In practice, Kaiming(He) normal initialization is often chosen for layers in a network that:

- Deep Neuronal Networks
- Convolutional Neuronal Networks
- Fully Connected Layers that use ReLU
- Recurrent Neuronal Networks
- Transfer Learning and Fine-Tuning
- Models with ReLU Derivatives

Weight Initialization

› Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.

Kaiming (He) normal distribution

```
import torch

t = torch.empty(4,3)
torch.nn.init.kaiming_normal_(t)
print(t)
```

Output

```
tensor([[ 0.6886, -0.1788, -0.7911],
        [-0.1446, -0.2412, -0.1562],
        [-0.4810,  0.7646, -0.1298],
        [ 0.6292,  1.4895, -1.2318]])
```

Weight Initialization

- Furthermore, **PyTorch** contains a sub-module (called ***init***) that already implements several methods for different type of initialization.
- Other initialization methods:
  - **torch.nn.init.ones_(tensor)** → all weights are initialized with value 1
  - **torch.nn.init.zeros_(tensor)** → all weights are initialized with value 0
  - **torch.nn.init.constant_(tensor, val)** → all weights are initialized with a constant value (val)
  - ...

Weight Initialization

- Finally, lets see how we can use this to create a neuronal network:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        # create 3 layes
        self.fc1 = nn.Linear(64, 128)  # First hidden layer
        self.fc2 = nn.Linear(128, 256) # Second hidden layer
        self.fc3 = nn.Linear(256, 10)  # Output layer

        # Initialize weights using Kaiming Normal initialization
        nn.init.kaiming_normal_(self.fc1.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.fc2.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.fc3.weight, nonlinearity='relu')

    def forward(self, input):
       # perform feed forward step
```

Loss Functions

Loss Functions

- Loss functions in PyTorch contain the implementation of one step from the backpropagation mechanism (computing the gradients).
- The following loss functions are implemented:
  - Mean Square Error (torch.nn.MSELoss)
  - Cross Entropy (torch.nn.CrossEntropyLoss)
  - Negative log likelihood loss (torch.nn.NLLLoss)
  - Binary cross entropy loss (torch.nn.BCELoss)
  - Mean absolute error loss (torch.nn.L1Loss)
- Currently PyTorch has more than 25 predefined loss function.

Loss Functions

› The regular usage of a loss function is as follows:

1. Create a loss function (based on the existing one or as something derived from the Module)
2. Call the loss function with two tensors (one that reflects the prediction, and another one that reflects the expected target). The output of this operation will be another tensor that contains the computed loss between those two tensors
3. Call the method .backward() on the tensor obtained in step 2. This will compute the gradients and update the prediction tensor with them (the gradients). It is important for all trainable parameters to have **requires_grad** set to True for weights and biases (applicable only to leafs in the computation graph

Loss Functions

Let's see an example:

```
import torch

# create a prediction and expected (target) tensors
prediction = torch.tensor([1., 2., 3.], requires_grad=True)
expected = torch.tensor([4., 5., 6.])

# create a loss function (we will use MSE)
loss_function = torch.nn.MSELoss()

# Compute the loss
loss = loss_function(prediction, expected)
print("Loss = ",loss)
print("Before computing the gradients = ",prediction.grad)

# Compute the gradients
loss.backward()
print("Gradients = ",prediction.grad)
```

Output

```
Loss =  tensor(9., grad_fn=<MseLossBackward0>)
Before computing the gradients =  None
Gradients =  tensor([-2., -2., -2.])
```

Loss Functions

Let's see an example:

```
import torch

# create a prediction and expected (target) tensors
prediction = torch.tensor([1., 2., 3.], requires_grad=True)
expected = torch.tensor([4., 5., 6.])

# create a loss function (we will use MSE)
loss_function = torch.nn.MSELoss()

# Compute the loss
loss = loss_function(prediction, expected)
print("Loss = ",loss)
print("Before computing the gradients = ",prediction.grad)

# Compute the grad
loss.backward()
print("Gradients = ",prediction.grad)
```

Output

```
Loss =  tensor(9., grad_fn=<MseLossBackward0>)
Before computing the gradients =  None
Gradients =  tensor([-2., -2., -2.])
```

$$MSE = \frac{(1-4)^2+(2-5)^2+(3-6)^2}{3} = \frac{9+9+9}{3} = \frac{27}{3} = 9$$

Loss Functions

Let's see an example:

```
import torch

# create a prediction and expected (target) tensors
prediction = torch.tensor([1., 2., 3.], requires_grad=True)
expected = torch.tensor([4., 5., 6.])

# create a loss function (we will use MSE)
loss_function = torch.nn.MSELoss()

# Compute the lo
loss = loss_func
print("Loss = ",loss)
print("Before computing the gradients = ",prediction.grad)

# Compute the gradients
loss.backward()
print("Gradients = ",prediction.grad)
```

Output

```
Loss =  tensor(9., grad_fn=<MseLossBackward0>)
Before computing the gradients =  None
Gradients =  tensor([-2., -2., -2.])
```

Notice that parameter .grad (from gradient) is None (meaning that by default no gradients were computed for the prediction tensor).

Loss Functions

Let's see an example:

```
import torch

# create a prediction and expected (target) tensors
prediction = torch.tensor([1., 2., 3.], requires_grad=True)
expected = torch.tensor([4., 5., 6.])

# create a loss function (we will use MSE
loss_funct

# Compute
loss = los
print("Los
print("Bef

# Compute the gradients
loss.backward()
print("Gradients = ",prediction.grad)
```

Output

```
Loss =  tensor(9., grad_fn=<MseLossBackward0>)
Before computing the gradients =  None
Gradients =  tensor([-2., -2., -2.])
```

When **loss.backward()** is called, the gradients are computed and stored in the prediction.grad data member. The loss tensor keeps an internal reference to the ***prediction*** and ***target/expected*** tensors that were used to initialize the loss function. As such, when the .backward() method is being called, the loss function can update the ***prediction*** tensor.

Loss Functions

Let's see an example:

```
import torch

# create a prediction and expected (target) tensors
prediction = torch.tensor([1., 2., 3.], requires_grad=False
expected = torch.tensor([4., 5., 6.])

# create a loss function (we will use MSE)
loss_function = torch.nn.M

# Compute the loss
loss = loss_function(prediction, expected)
print("Loss = ",loss)
print("Before computin

# Compute the gradient
loss.backward()
print("Gradients = ",p
```

You need to enable gradient support for tensors where you want to update based on gradients (for example tensor that store weights).

Error

```
Loss =   tensor(9.)
Before computing the gradients =   None
Traceback (most recent call last):
   File "e:\Lucru\RN\teste\a.py", line 16, in <module>
      loss.backward()
...
RuntimeError: element 0 of tensors does not require grad and does not have a grad_fn
```

Loss Functions

Let’s see another example (with cross entropy function).

```
import torch

prediction = torch.tensor([1., 2., 3.], requires_grad=True)
expected = torch.tensor([4., 5., 6.])

loss_function = torch.nn.CrossEntropyLoss()
loss = loss_function(prediction, expected)
loss.backward()

print("loss = ",loss)
print("Gradients = ",prediction.grad)
```

Output

```
loss =  tensor(19.1141, grad_fn=<DivBackward1>)
Gradients =  tensor([-2.6495, -1.3291,  3.9786])
```

Computation Graph

Computation Graph

Let's analyze the following example:

```
import torch

input = torch.tensor([1., 2., 3., 2., 1.])
expected = torch.tensor([4., 5., 6.])
net = torch.nn.Linear(5,3)
output = net(input)

loss_function = torch.nn.CrossEntropyLoss()
loss = loss_function(output, expected)

print("Before ", net.weight.grad)
loss.backward()
print("Gradients", net.weight.grad)
```

Output

```
Before  None
Gradients tensor([[-0.0033, -0.0065, -0.0098, -0.0065, -0.0033],
                  [ 0.8073,  1.6146,  2.4219,  1.6146,  0.8073],
                  [-0.8040, -1.6080, -2.4121, -1.6080, -0.8040]])
```

If we provided the ***output*** and ***expected*** to the loss function, how come the **net** variable (*grad* parameter) has changed ?

Computation Graph

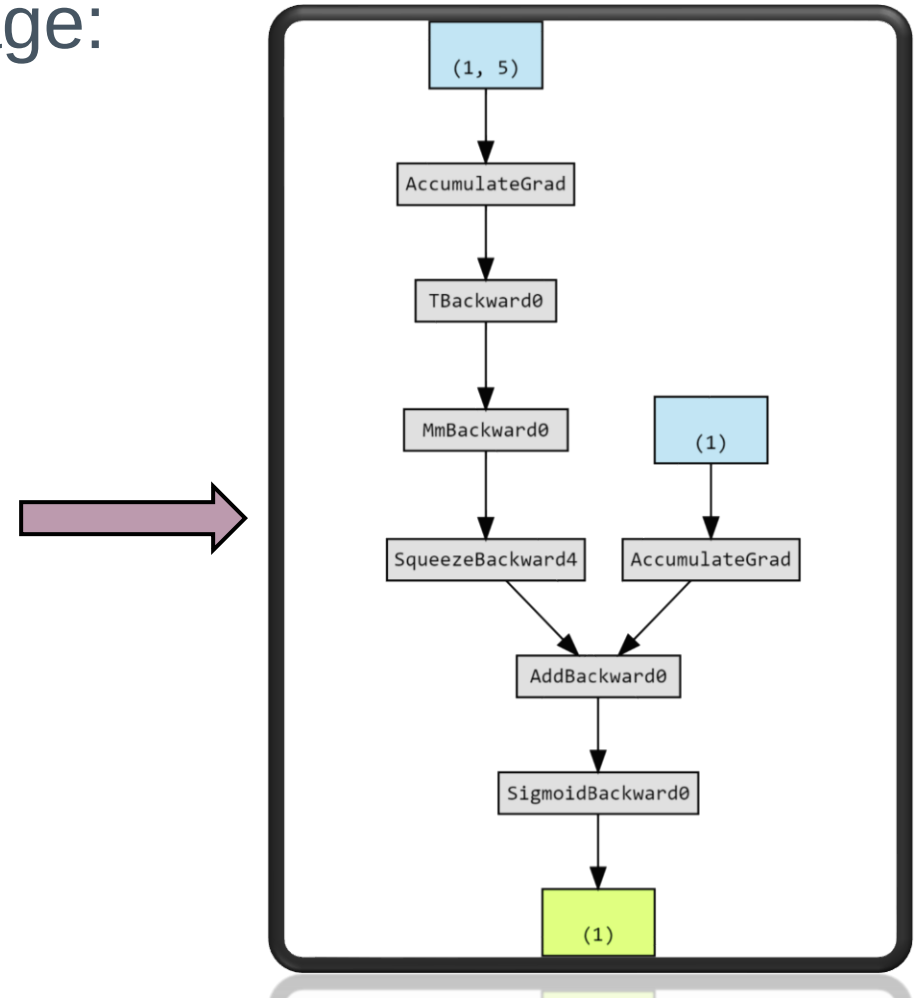
The answer is that ***PyTorch*** secretly build a list of all layers that need to be updated when the feed forward step is being build.

This list is also called a computational graph and allows other components (such as optimizers) to access the entire network.

Computation Graph

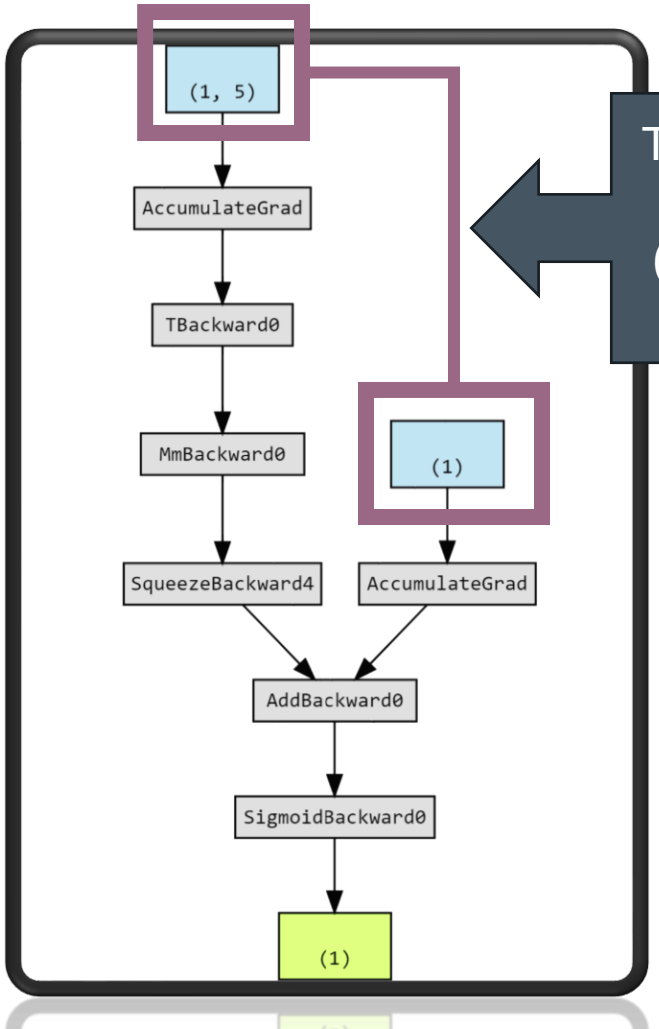
Let's see a very simple example of a computation graph. We will draw it using ***torchviz*** package:

```
import torch, torchviz

net = torch.nn.Sequential(
    torch.nn.Linear(5,1),
    torch.nn.Sigmoid()
)

input = torch.tensor([1.,2.,3.,4.,5.])
output = net(input)
dot = torchviz.make_dot(output)
dot.save('comp_graph.dot')
```



Computation Graph

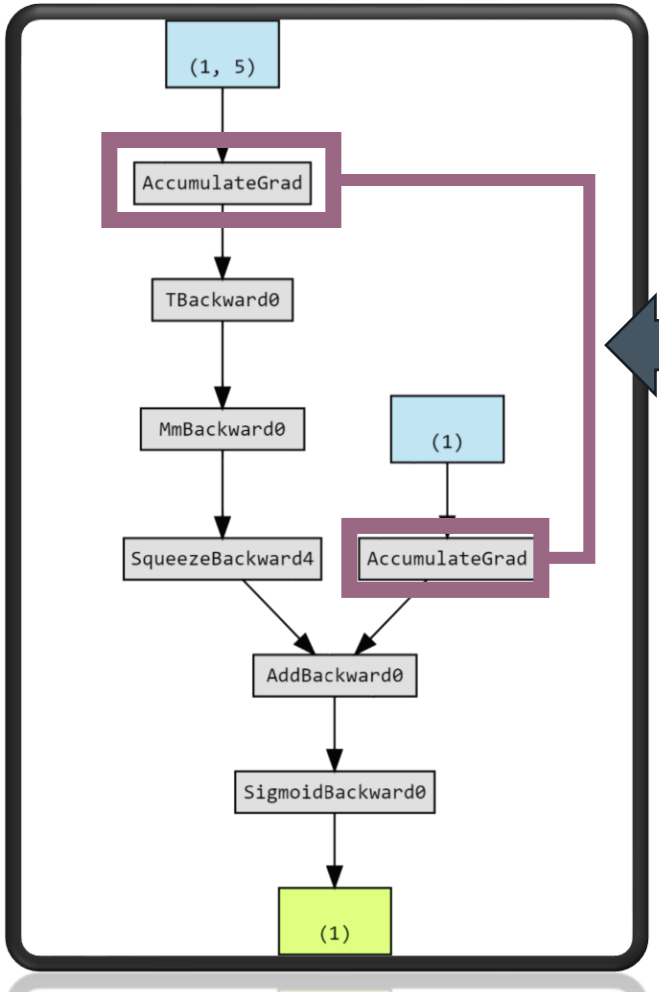
Let's see break down this diagram:



These nodes are in fact the tensors (as we have a very simple neuronal net), we only have an **input tensor** (with the shape (1,5)) and a **output tensor** that yields a scalar value (denote by the shape (1))

Computation Graph

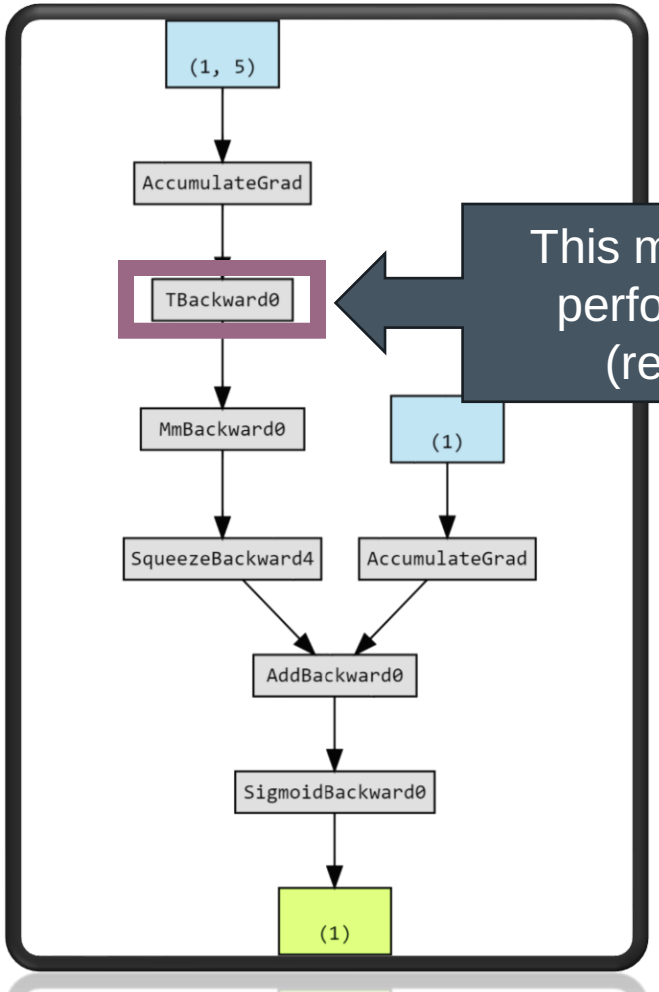
Let's see break down this diagram:



AccumulateGrad nodes are where the gradients are accumulated into the .grad attribute of the tensors during the backward pass.

Computation Graph

Let's see break down this diagram:

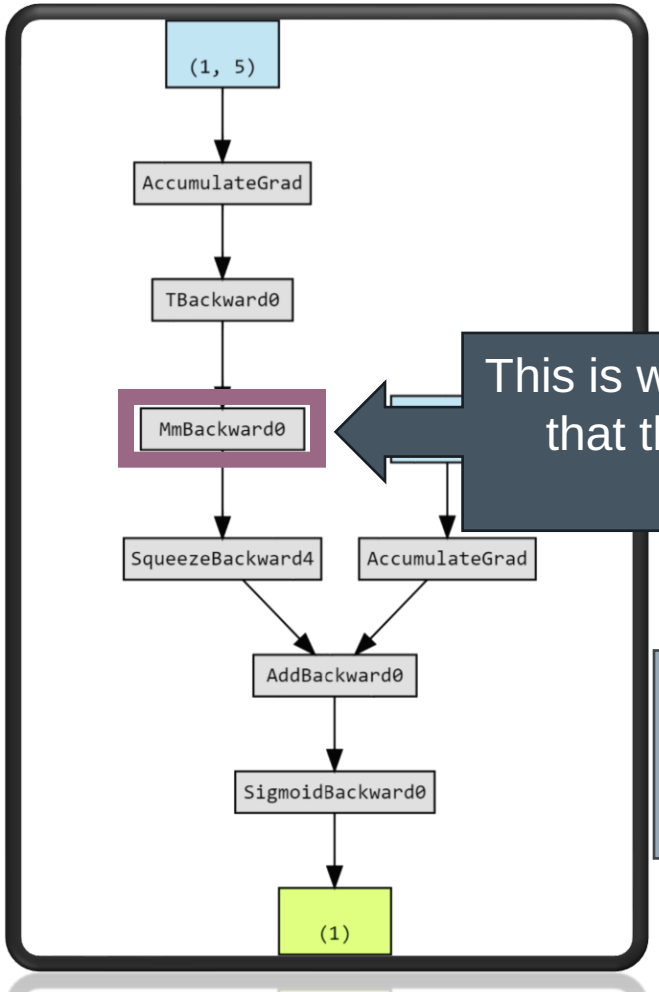


This means that a transpose operation is going to be performed over the matrix in the feed forward step (required to perform the matrix multiplication).

On the backpropagation step, this is where the gradient of the loss with respect to the transposed tensor is computed.

Computation Graph

Let's see break down this diagram:

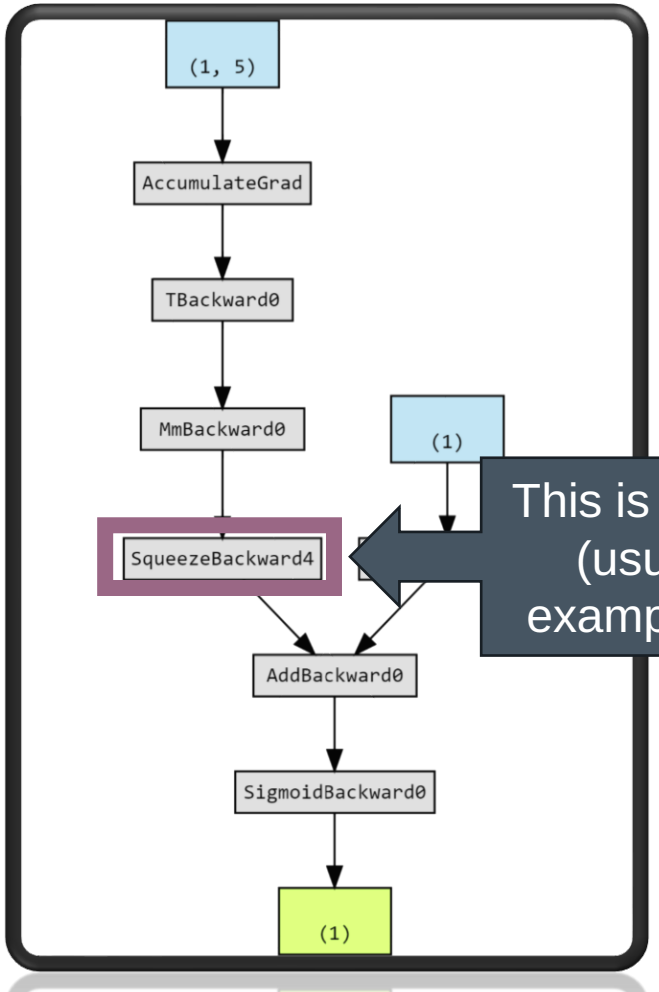


This is where the matrix multiplication happens (meaning that the input is multiplied with the weights from the tensor of size (5,1) – after transposing)

On the backpropagation step, this is where the gradients with respect to the input matrices of the matrix multiplication are going to be calculated.

Computation Graph

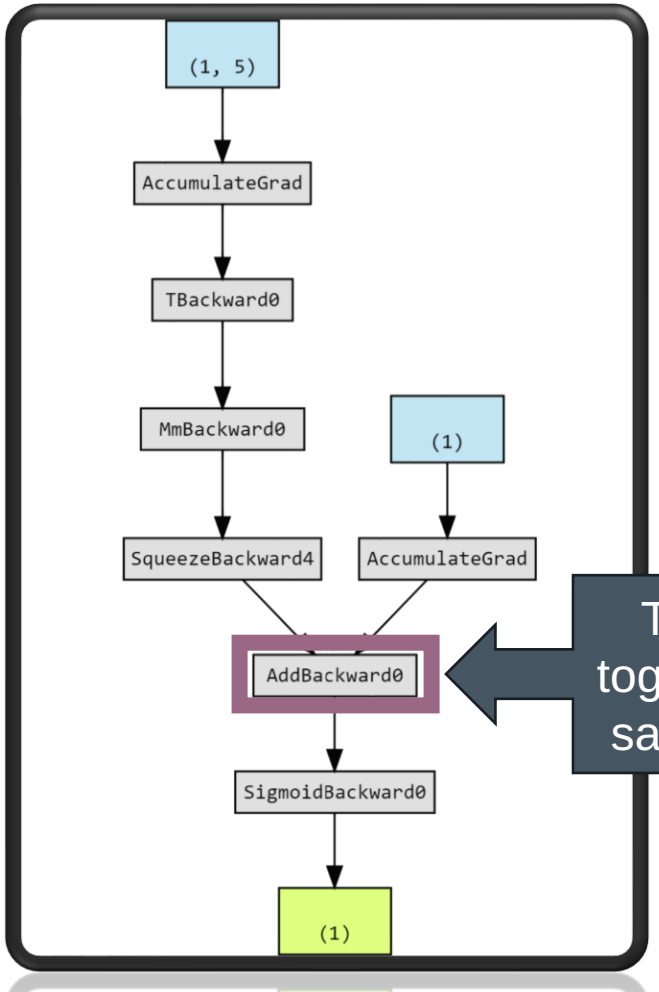
Let's see break down this diagram:



This is more a step that normalizes the shape of the result (usually by reducing dimension of one element). For example, a shape of (100,1,1,1) will be reduced to (100).

Computation Graph

Let's see break down this diagram:

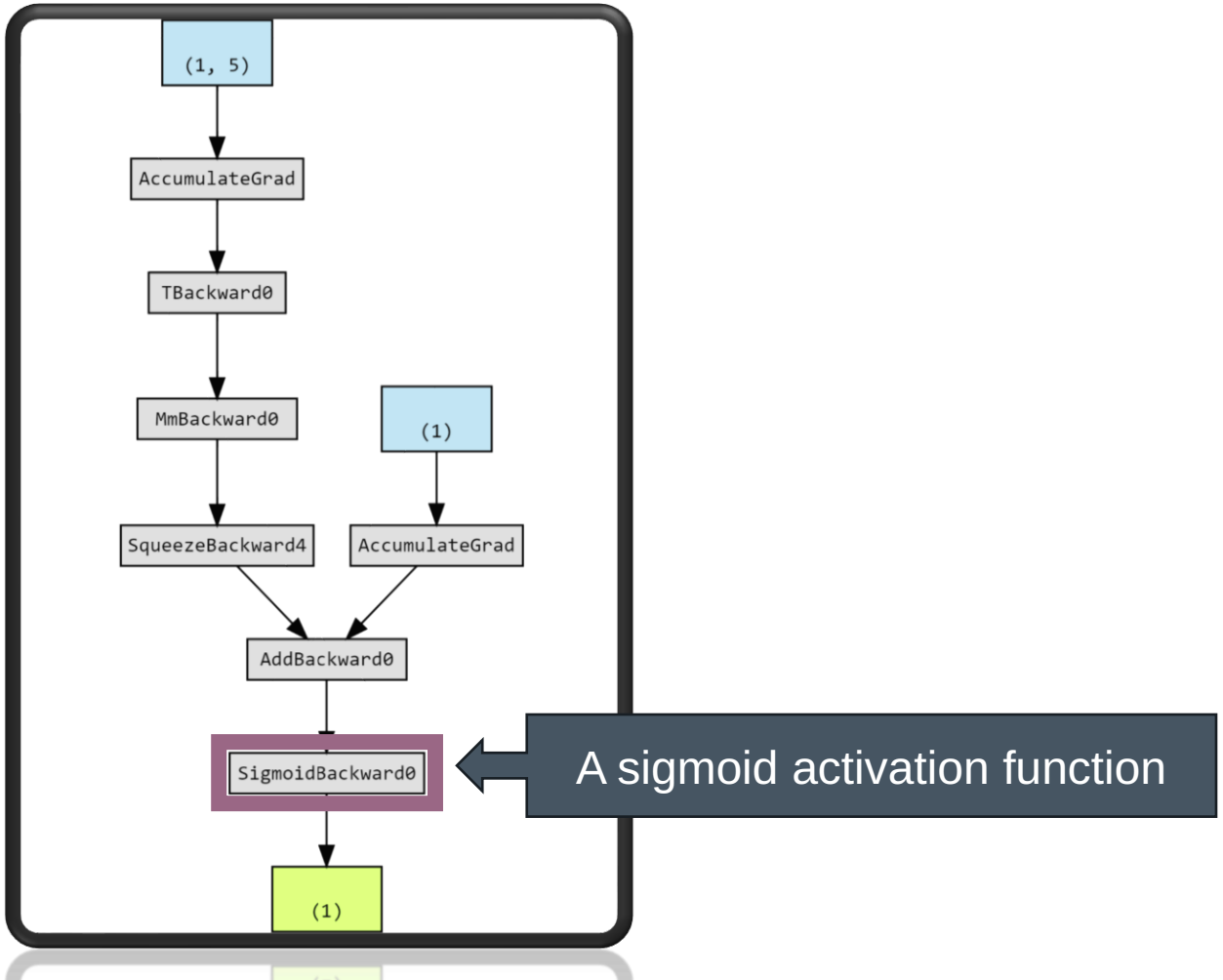


In the backpropagation test, pass the gradient from the loss unchanged to both operands of the addition since the derivative of an addition operation with respect to either input is 1.

This indicates a point where two tensors were added together. Tensor addition is element-wise and implies the same shape (hence the squeeze operation before this).

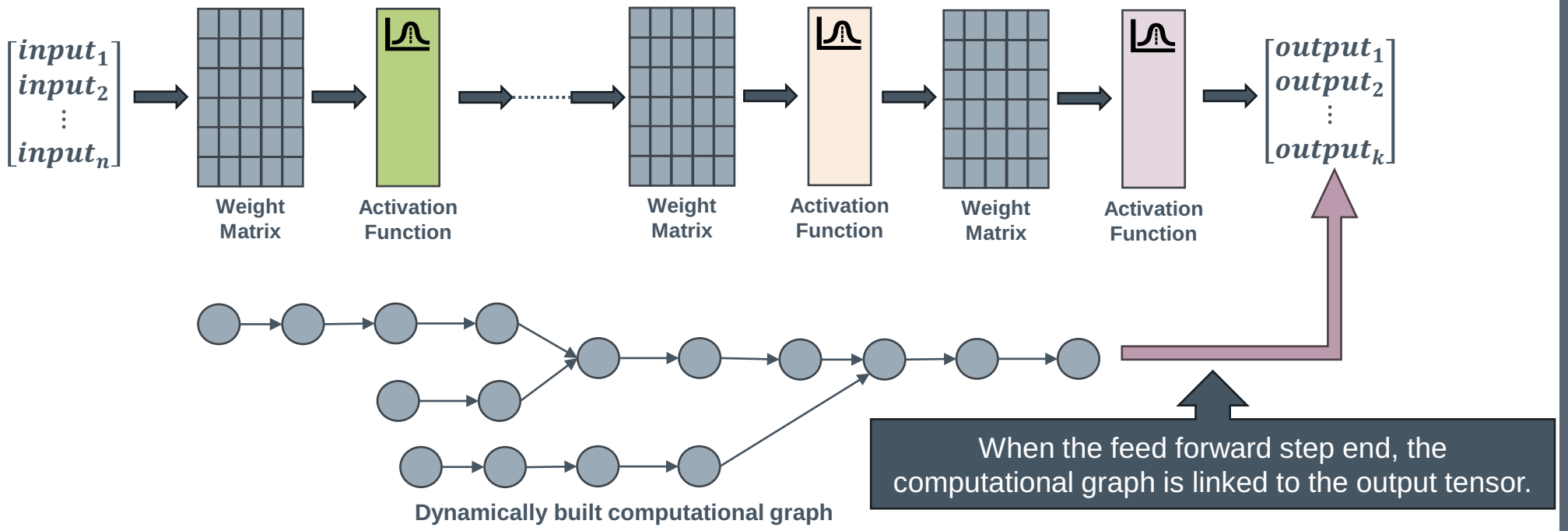
Computation Graph

Let's see break down this diagram:



Computation Graph

The computation graph is being build dynamically as operation are performed / computed. Internally, the output tensor from the feed forward step has a link (reference) to the dynamic computation graph.



Backpropagation in Pytorch

Backpropagation in Pytorch

Let's review the backpropagation process:

1. **Compute the output** of the current model given an input (an input can be one item or multiple items)
2. **Compute the loss** using the output obtained on the previous step and the expected value (if the input is represented by multiple items, the computed loss is the average of the loss from all items)
3. **Compute the delta gradients** for each layer
4. **Update weights** using from the computed delta gradients from the previous layer.

Backpropagation in Pytorch

Let's see how this flow looks like in PyTorch:

```
all_weights = model.parameters()
optimizer = optim.<Name>(all_weights, …)
```

Setup Phase

First, we need to create an **optimizer** (this is just a fancy word for a piece of code that knows how to compute gradients and to some automated steps).

Example of optimizer: **gradient descent**

Regardless of the chosen optimizer, **a reference to all weights** (from all layers in the model) has to be provided). Any object derived from Module has a method called .parameters() that provides a list of all weights (tensors) that support gradients (**requires_grad=True**) and as a result, can be adjusted.

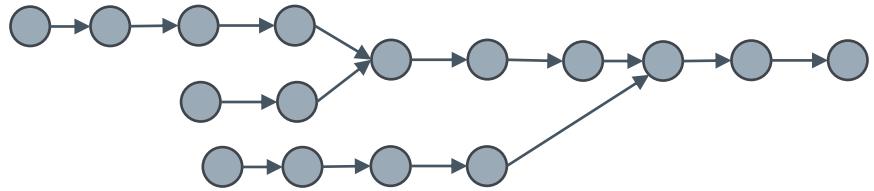
Backpropagation in Pytorch

Let's see how this flow looks like in PyTorch:

```
all_weights = model.parameters()
optimizer = optim.<Name>(all_weights, …)

output = model(input)
```

1. Compute the output

Compute the output of the model (given an input). The output variable is a tensor, that has the entire computation graph linked.



Dynamically built computational graph

Backpropagation in Pytorch

Let's see how this flow looks like in PyTorch:

```
all_weights = model.parameters()
optimizer = optim.<Name>(all_weights, …)


output = model(input)

loss = loss_function(output, expected)
```

2. Compute the loss

Compute the loss using the output from the previous step and the expected value for the input that was used to obtained the output.

Backpropagation in Pytorch

Let's see how this flow looks like in PyTorch:

```
all_weights = model.parameters()
optimizer = optim.<Name>(all_weights, …)


output = model(input)

loss = loss_function(output, expected)

optimizer.zero_grad()
loss.backward()
```

3. Compute the delta gradients

Since these steps are repeated, we need first to clear all previously computed gradients (the parameter **.grad**) from each tensor that resides in computational graph (that is linked by the output tensor).

After we do this, we can compute all delta gradients and we can store them into the **.grad** parameter from each tensor. This is done via the **loss.backward()** method.

Backpropagation in Pytorch

Let's see how this flow looks like in PyTorch:

```
all_weights = model.parameters()
optimizer = optim.<Name>(all_weights, …)


output = model(input)

loss = loss_function(output, expected)

optimizer.zero_grad()
loss.backward()

optimizer.step()
```

4. Update weights

In this step, the optimizer (that has access to all weights) has to update the weights based on the delta computed in the previous steps.

Backpropagation in Pytorch

Let's put all of these together and created a simple training:

```python
import torch

# Instantiate the network
model = torch.nn.Sequential(torch.nn.Linear(8,4),torch.nn.Linear(4,1))
loss_function = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Simulate some random input data and labels
inputs = torch.randn(64, 8)
labels = torch.randn(64, 1)

# Training loop for 10 iterations
for epoch in range(10):
    outputs = model(inputs)
    loss = loss_function(outputs, labels)
    print("Epoch=",epoch," loss=",loss)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Output

```
Epoch= 0  loss= tensor(0.7550, grad_fn=<MseLossBackward0>)
Epoch= 1  loss= tensor(0.7522, grad_fn=<MseLossBackward0>)
Epoch= 2  loss= tensor(0.7495, grad_fn=<MseLossBackward0>)
Epoch= 3  loss= tensor(0.7469, grad_fn=<MseLossBackward0>)
Epoch= 4  loss= tensor(0.7445, grad_fn=<MseLossBackward0>)
Epoch= 5  loss= tensor(0.7422, grad_fn=<MseLossBackward0>)
Epoch= 6  loss= tensor(0.7399, grad_fn=<MseLossBackward0>)
Epoch= 7  loss= tensor(0.7378, grad_fn=<MseLossBackward0>)
Epoch= 8  loss= tensor(0.7357, grad_fn=<MseLossBackward0>)
Epoch= 9  loss= tensor(0.7337, grad_fn=<MseLossBackward0>)
```

Backpropagation in Pytorch

Let's put all of these together and created a simple training:

```
import torch

# Instantiate the network
model = torch.nn.Sequential(torch.nn.Linear(8,4),torch.nn.Linear(4,1))
loss_function = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Simulate some data
inputs = torch.randn(100, 8)
labels = torch.randn(100, 1)

# Training loop for 10 iterations
for epoch in range(10):
    outputs = model(inputs)
    loss = loss_function(outputs, labels)
    print("Epoch=",epoch," loss=",loss)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

A model with 3 layers:
- input layer (8 neurons)
- hidden layer (4 neurons)
- output layer (one neuron)

Backpropagation in Pytorch

Let's put all of these together and created a simple training:

```
import torch

# Instantiate the network
model = torch.nn.Sequential(torch.nn.Linear(8,4),torch.nn.Linear(4,1))
loss_function = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

inputs = torch.randn(64, 8)  # Assuming input and labels
labels = torch.randn(64, 1)

# Training loop for 10 iterations
for epoch in range(10):
    outputs = model(inputs)
    loss = loss_function(outputs, labels)
    print("Epoch=",epoch," loss=",loss)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Loss function: MSE (Mean Square Error)

Backpropagation in Pytorch

Let's put all of these together and created a simple training:

```
import torch

# Instantiate the network
model = torch.nn.Sequential(torch.nn.Linear(8,4),torch.nn.Linear(4,1))
loss_function = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Simulate some random input data and labels
inputs = torch.randn(100, 8)
labels = torch.randn(100, 1)

# Training loop for 10 iterations
for epoch in range(10):
    outputs = model(inputs)
    loss = loss_function(outputs, labels)
    print("Epoch=",epoch," loss=",loss)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Optimizer: **SGD** (Stochastic Gradient Descent) with a learning rate of 0.01

Backpropagation in Pytorch

Let's put all of these together and created a simple training:

```
import torch

# Instanti
model = to                                          near(4,1))
loss_funct
optimizer

# Simulate some random input data                   labels
inputs = torch.randn(64, 8)
labels = torch.randn(64, 1)

# Training loop for 10 iterations
for epoch in range(10):
    outputs = model(inputs)
    loss = loss_function(outputs, labels)
    print("Epoch=",epoch," loss=",loss)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Notice that even if the inputs and the outputs consists in 64 entries, the computed loss contains one entry (the average of all losses computed over the 64 resulted outputs and labes).

Output

```
Epoch= 0  loss= tensor(0.7550, grad_fn=<MseLossBackward0>)
Epoch= 1  loss= tensor(0.7522, grad_fn=<MseLossBackward0>)
Epoch= 2  loss= tensor(0.7495, grad_fn=<MseLossBackward0>)
Epoch= 3  loss= tensor(0.7469, grad_fn=<MseLossBackward0>)
Epoch= 4  loss= tensor(0.7445, grad_fn=<MseLossBackward0>)
Epoch= 5  loss= tensor(0.7422, grad_fn=<MseLossBackward0>)
Epoch= 6  loss= tensor(0.7399, grad_fn=<MseLossBackward0>)
Epoch= 7  loss= tensor(0.7378, grad_fn=<MseLossBackward0>)
Epoch= 8  loss= tensor(0.7357, grad_fn=<MseLossBackward0>)
Epoch= 9  loss= tensor(0.7337, grad_fn=<MseLossBackward0>)
```

Stochastic Gradient Descent

Stochastic Gradient Descent

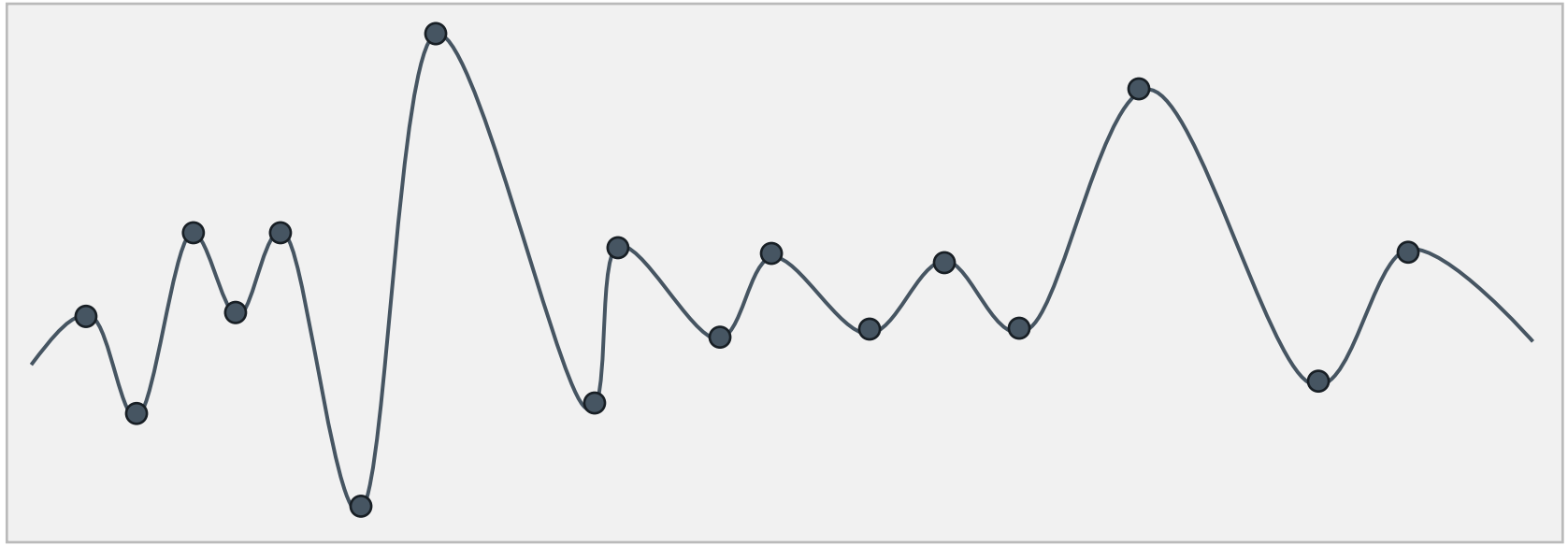
As a general observation, we can say that gradient descent implies updating weights in the following way:

$$w_{t+1} = w_t + \alpha \times \nabla Loss(w_t), where$$

$$w_t = weights\ at\ the\ moment\ t$$
$$w_{t+1} = weights\ at\ the\ moment\ t + 1,$$
$$\alpha = learning\ rate,$$
$$\nabla Loss(w_t) = gradient\ of\ the\ loss\ function\ with\ respect\ to\ the\ w_t$$

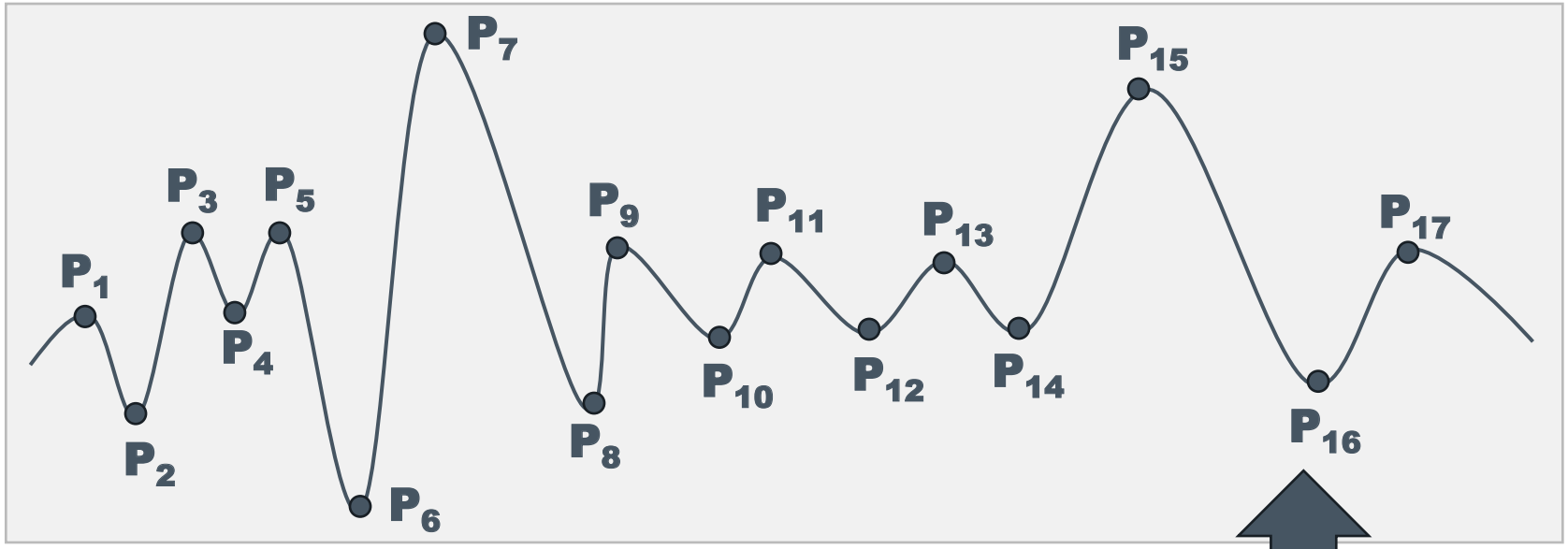
Stochastic Gradient Descent

We know that this rule converges to a local minimum, but this exact local minimum could be a problem. Let's consider the following graphic obtained via Lagrange interpolation:



Stochastic Gradient Descent

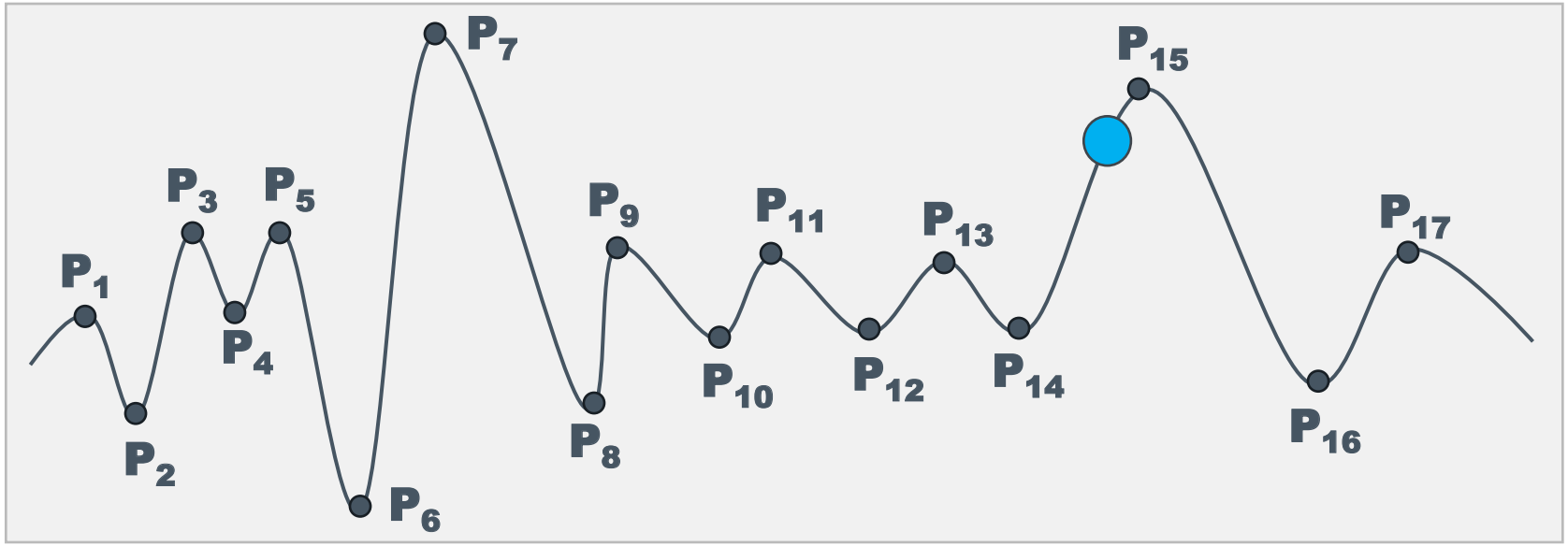
We know that this rule converges to a local minimum, but this exact local minimum could be a problem. Let's consider the following graphic obtained via Lagrange interpolation:



We can also say that the **dataset** used to obtain this graph is form out of 17 points (P_1 to P_{17}) used for Lagrange interpolation.

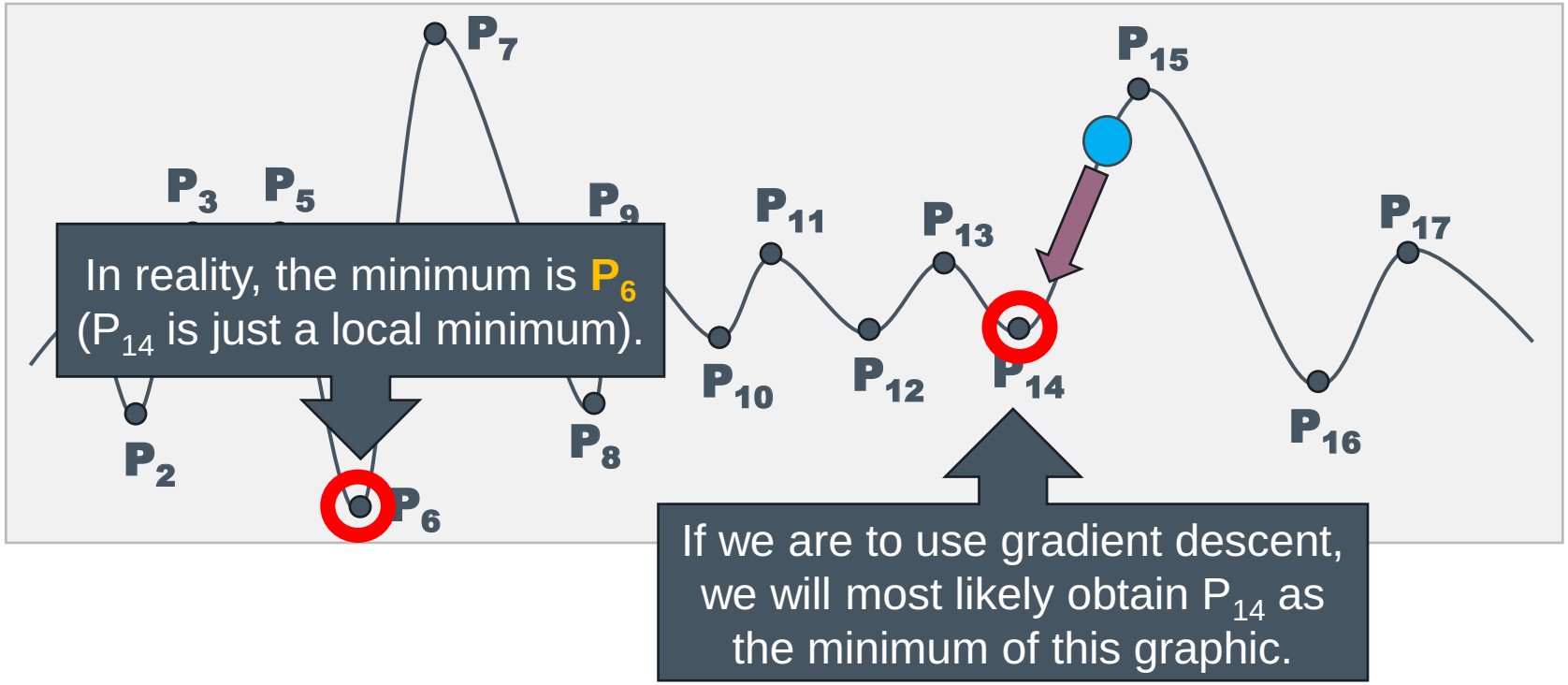
Stochastic Gradient Descent

Now let's consider that we have a blue dot (located somewhere between P_{14} and P_{15} and we want to find the minimum of this graphic starting from the blue dot position:



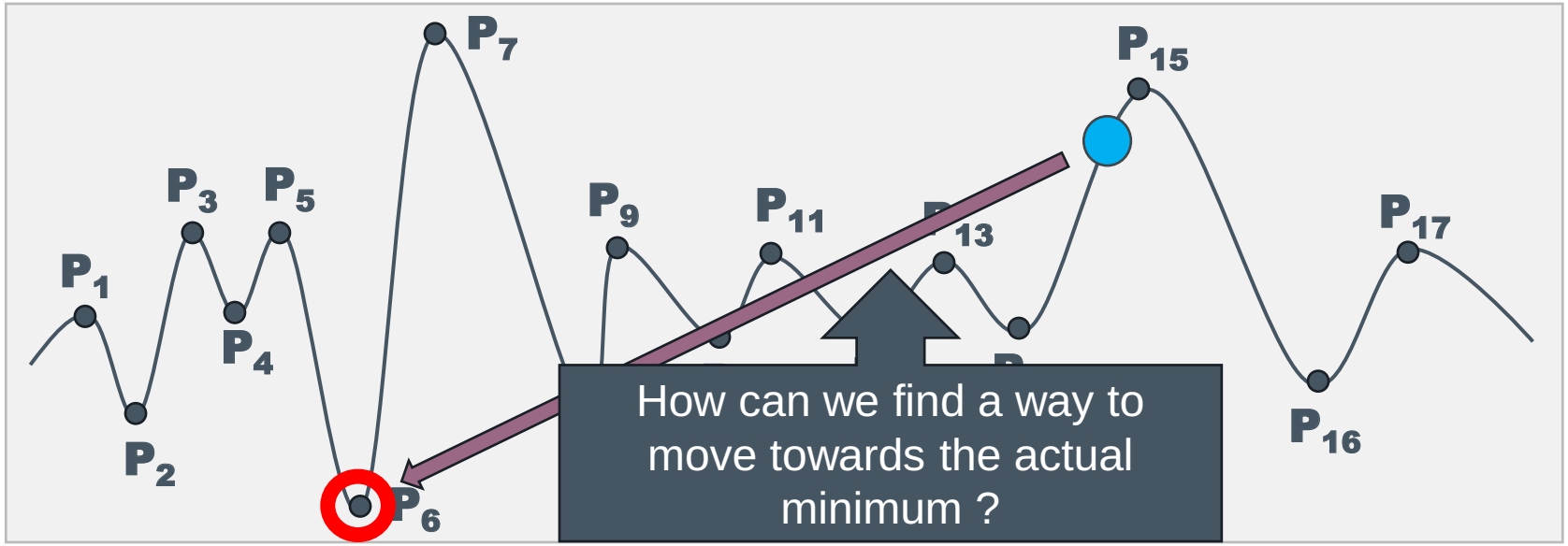
Stochastic Gradient Descent

Now let's consider that we have a blue dot (located somewhere between P_{14} and P_{15} and we want to find the minimum of this graphic starting from the blue dot position:



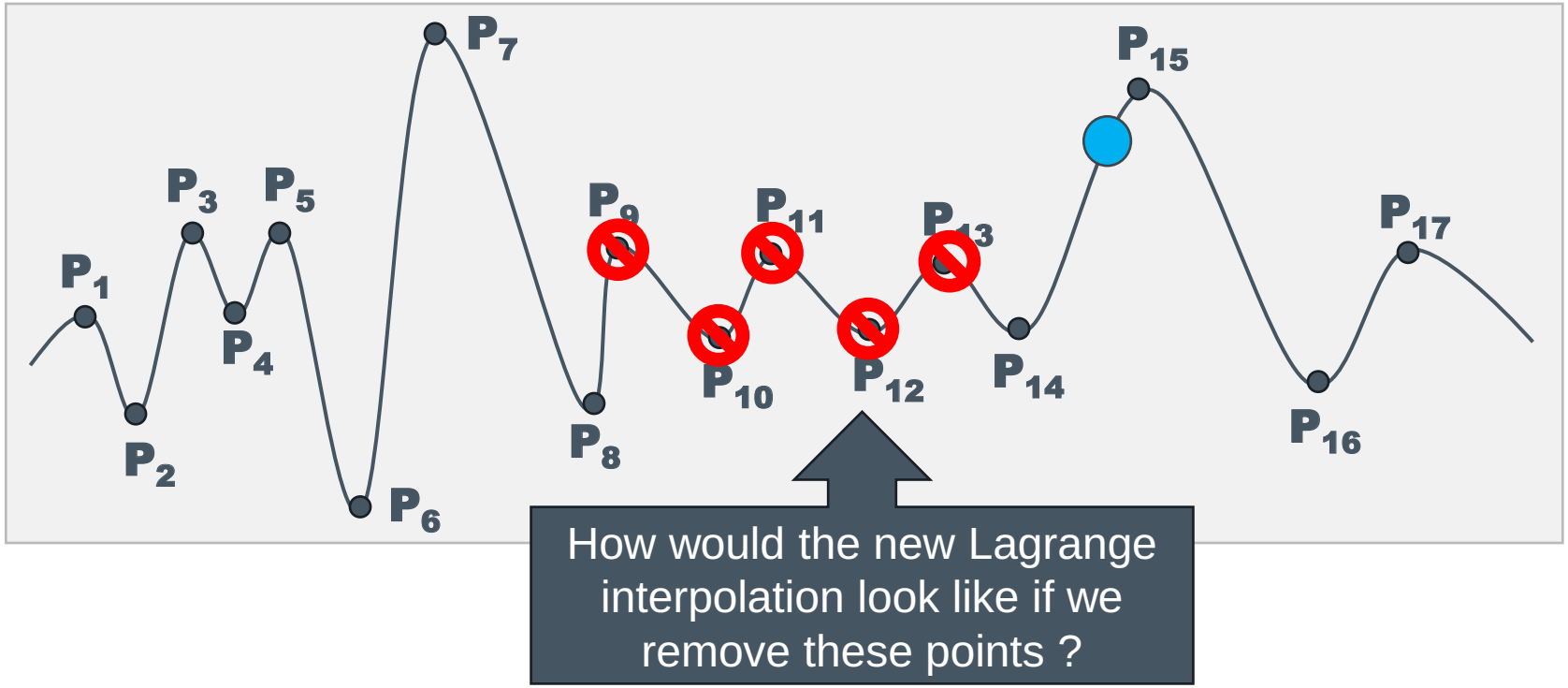
Stochastic Gradient Descent

Now let's consider that we have a blue dot (located somewhere between P_{14} and P_{15} and we want to find the minimum of this graphic starting from the blue dot position:



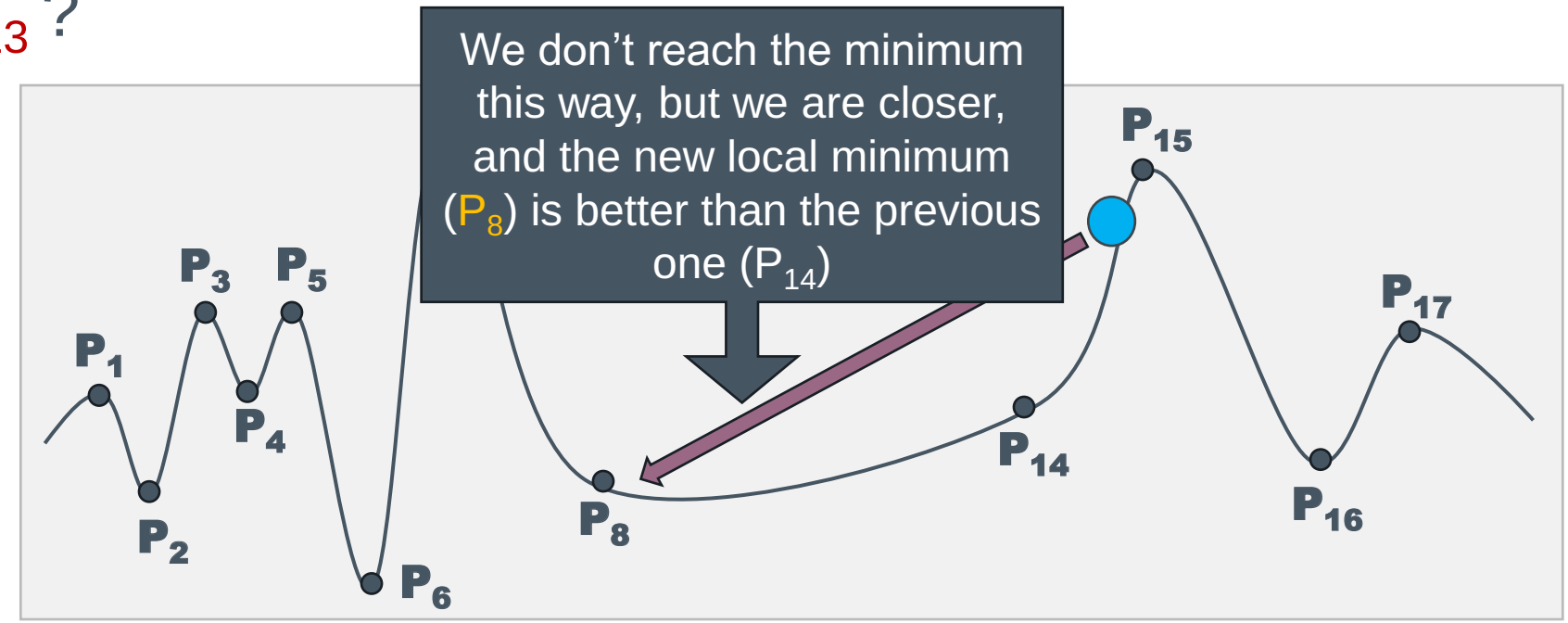
Stochastic Gradient Descent

What would happen if we would NOT take into consideration all 17 points ? What if we remove from the data set points P_9 to P_{13} ?



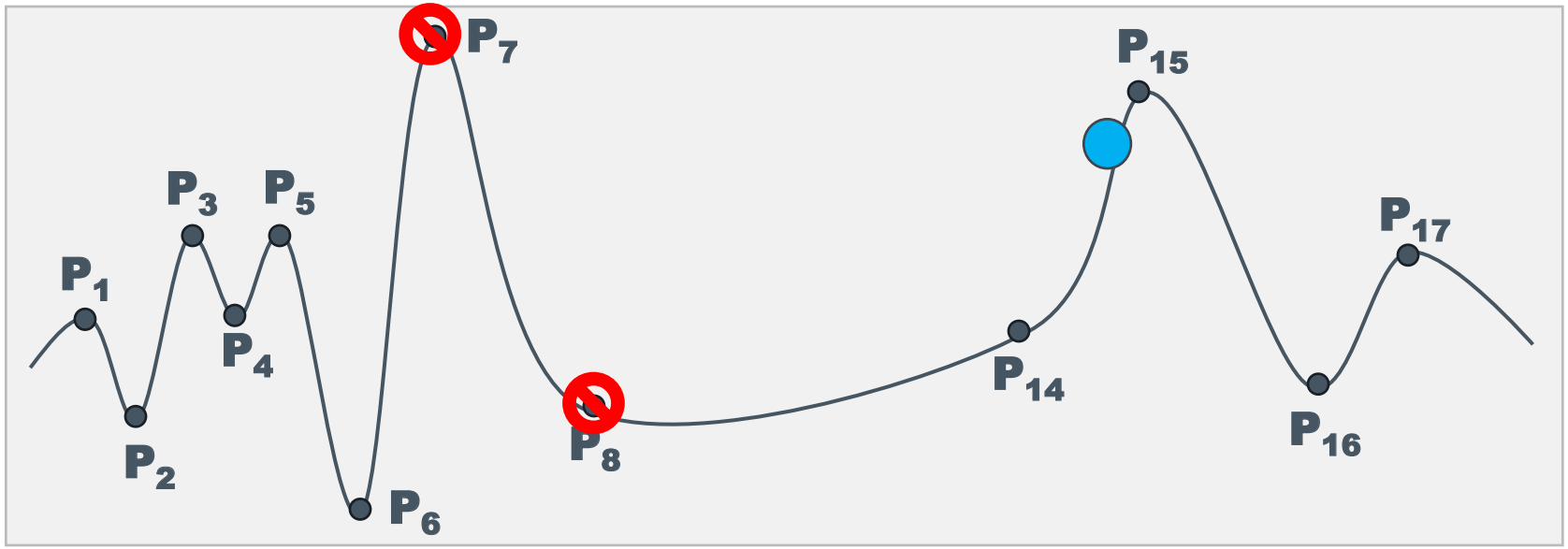
Stochastic Gradient Descent

What would happen if we would NOT take into consideration all 17 points ? What if we remove from the data set points P_9 to P_{13} ?



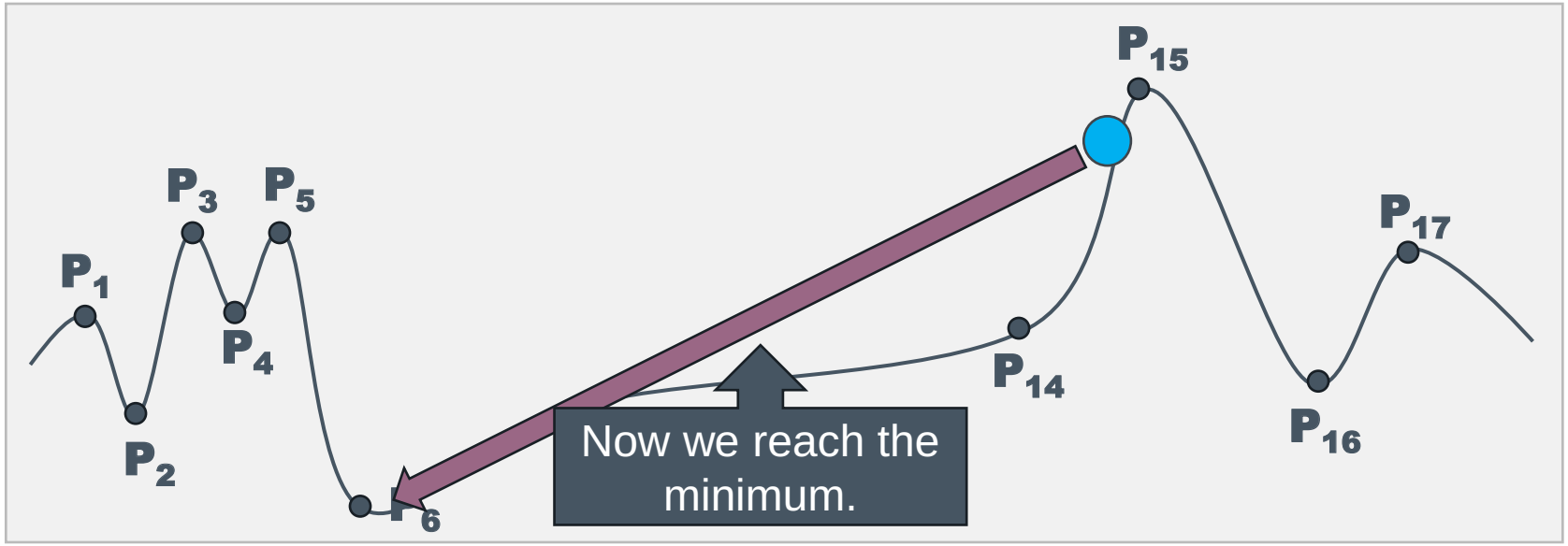
Stochastic Gradient Descent

But what if we remove even mode points from the data set – this time let's remove P_8 and P_7



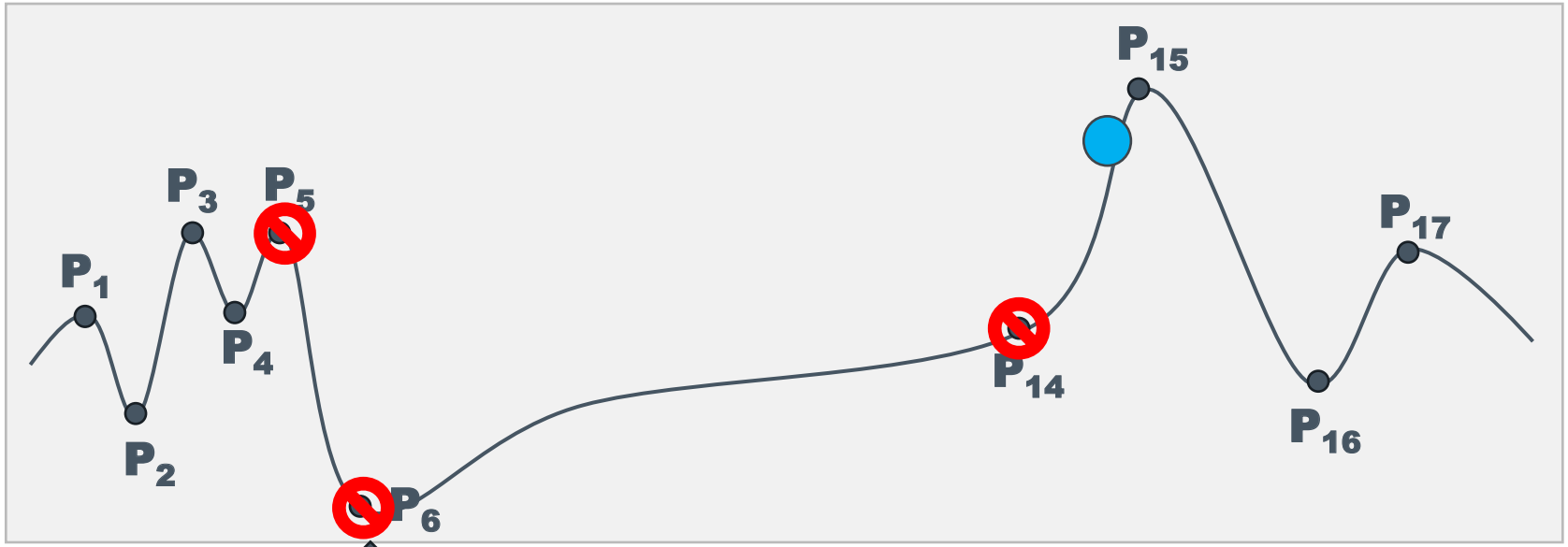
Stochastic Gradient Descent

But what if we remove even mode points from the data set – this time let's remove P_8 and P_7



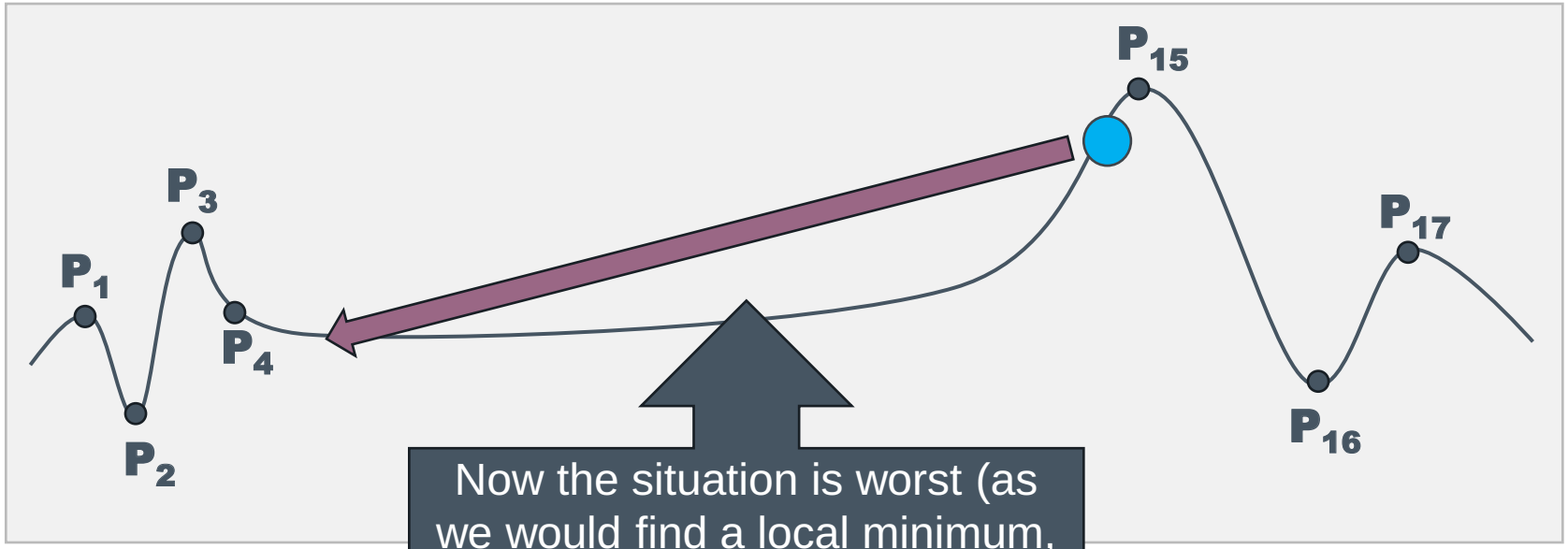
Stochastic Gradient Descent

But what if we decide to remove even more (lets say points P_5 , P_6 and P_{14} ?



Stochastic Gradient Descent

But what if we decide to remove even more (lets say points P_5 , P_6 and P_{14} ?



Stochastic Gradient Descent

So … what is the conclusion of this experiment:

1. Using the **entire dataset** might block our optimization towards a local minimum
2. Using a smaller part (**a batch**) of a data set is more likely to move out optimization to a better minimum (maybe to the actual minimum of the function).
3. Using a **too smaller part** (e.g., a batch of one element) might also be a bad idea as it will force also move towards another local minimum.

Stochastic Gradient Descent

This is how SGD (**S**tochastic **G**radient **D**escent) have appeared. The formula for the gradient descent is the same. What we change is the number of samples (batch size) that we used for the optimization step.

In other words, instead of computing the gradient descent over the entire dataset, we split the entire set into smaller batches (e.g., for example 64 entries) and we compute the gradient descent for each one of them.

This way we have a higher chance of locating the minimum.

Q & A