

Artificial Neural Networks

Course-7

Gavrilit Dragos

rev 1

π

AGENDA FOR TODAY

- › Space & memory considerations
- › Momentum
- › Nesterov Accelerated Gradient
- › Resilient Backpropagation
- › Adaptive Gradient Algorithm
- › Root Mean Square Propagation
- › Adaptive Moment Estimation
- › Nesterov-accelerated Adaptive Moment Estimation
- › AdaDelta
- › Dropout

Space & memory considerations

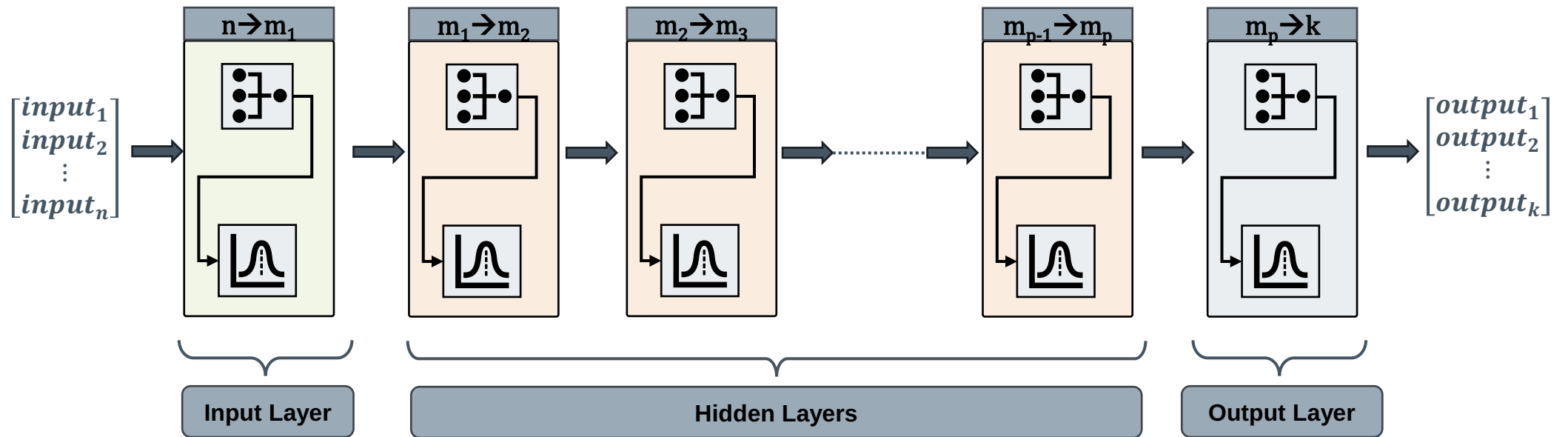
Space & memory considerations

Before we go into other optimization methods, let's discuss a little bit a couple of things that relates to the memory needed for a neural network.

This will come in handy later where some optimization come with a hidden price.

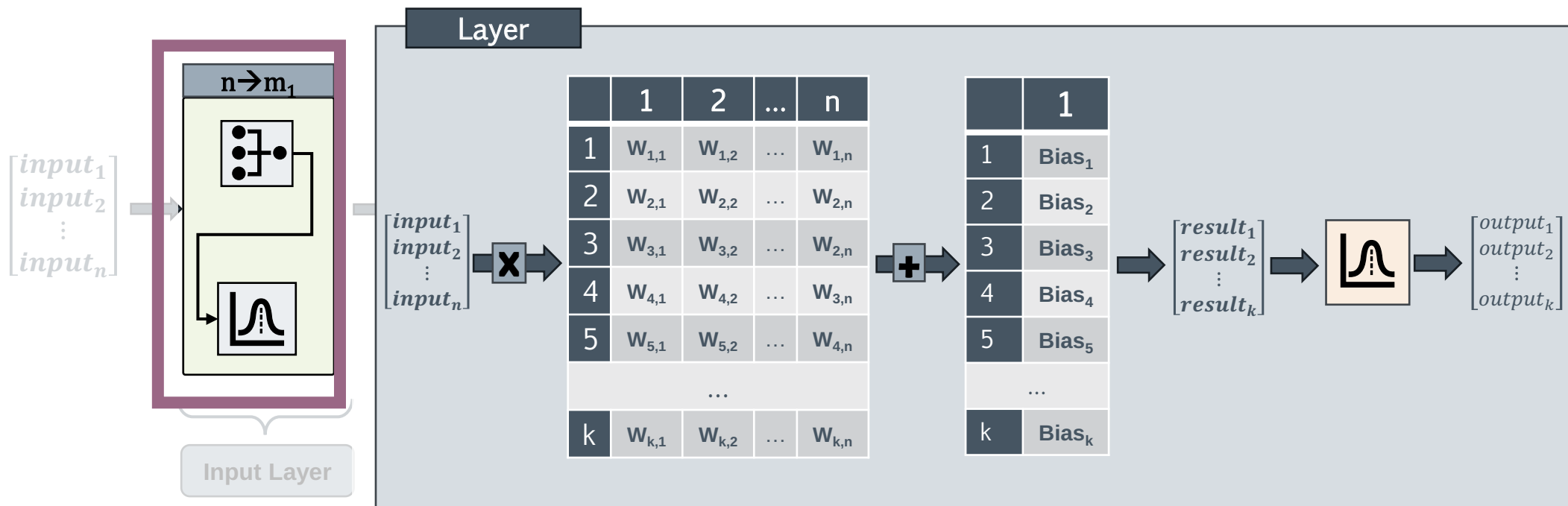
Space & memory considerations

Let's see how a neural network looks like:



Space & memory considerations

Let's see how a neural network looks like:



So ... as a general notion, we can say that the size of a layer (based on this description) is:

$$sizeof(Layer) = sizeof(type) \times (n \times k + k)$$

Space & memory considerations

Let's see some examples:

```
model = torch.nn.Linear(8,4)
```

One layer, no bias and let's consider float32 as data type, then:

$$\begin{aligned} \text{sizeof}(\text{model}) &= 8 \times 4 \times \text{sizeof}(\text{float32}) = 8 \times \\ &4 \times 4 = 128 \text{ bytes} \end{aligned}$$

```
model = torch.nn.Linear(  
    8,  
    4,  
    bias=True)
```

One layer, with bias data type is float 64:

$$\begin{aligned} \text{sizeof}(\text{model}) &= \text{sizeof}(\text{weights}) + \text{sizeof}(\text{bias}) \\ &= 8 \times 4 \times \text{sizeof}(\text{float64}) + 4 \times \text{sizeof}(\text{float64}) \\ &= 8 \times 4 \times 8 + 4 \times 8 = 288 \text{ bytes} \end{aligned}$$

```
model = Sequential (  
    Linear(8,4,bias=True),  
    Linear(4,1)  
)
```

Two layer, with bias data type is float 32:

$$\begin{aligned} \text{sizeof}(\text{model}) &= \text{sizeof}(\text{1}^{\text{st}} \text{ linear}) + \text{sizeof}(\text{2}^{\text{nd}} \text{ linear}) \\ \text{sizeof}(\text{1}^{\text{st}} \text{ linear}) &= 8 \times 4 \times \text{sizeof}(\text{float32}) + 4 \times \text{sizeof}(\text{float32}) \\ &= 8 \times 4 \times 4 + 4 \times 4 = 144 \text{ bytes} \\ \text{sizeof}(\text{2}^{\text{nd}} \text{ linear}) &= 4 \times 1 \times \text{sizeof}(\text{float32}) = 4 \times 4 = 16 \text{ bytes} \\ \text{sizeof}(\text{model}) &= 144 + 16 = 160 \text{ bytes} \end{aligned}$$

Space & memory considerations

The number of parameters of a model is the total number of weights and biases from the model. For example:

```
model = Sequential (  
    Linear(8,4,bias=True),  
    Linear(4,1)  
)
```

$$\text{params}(\text{model}) = \text{params}(\text{1}^{\text{st}} \text{ linear}) + \text{params}(\text{2}^{\text{nd}} \text{ linear})$$

$$\text{params}(\text{1}^{\text{st}} \text{ linear}) = 8 \times 4 + 4 = 8 \times 4 + 4 = 36 \text{ parameters}$$

$$\text{params}(\text{2}^{\text{nd}} \text{ linear}) = 4 \times 1 = 4 \text{ parameters}$$

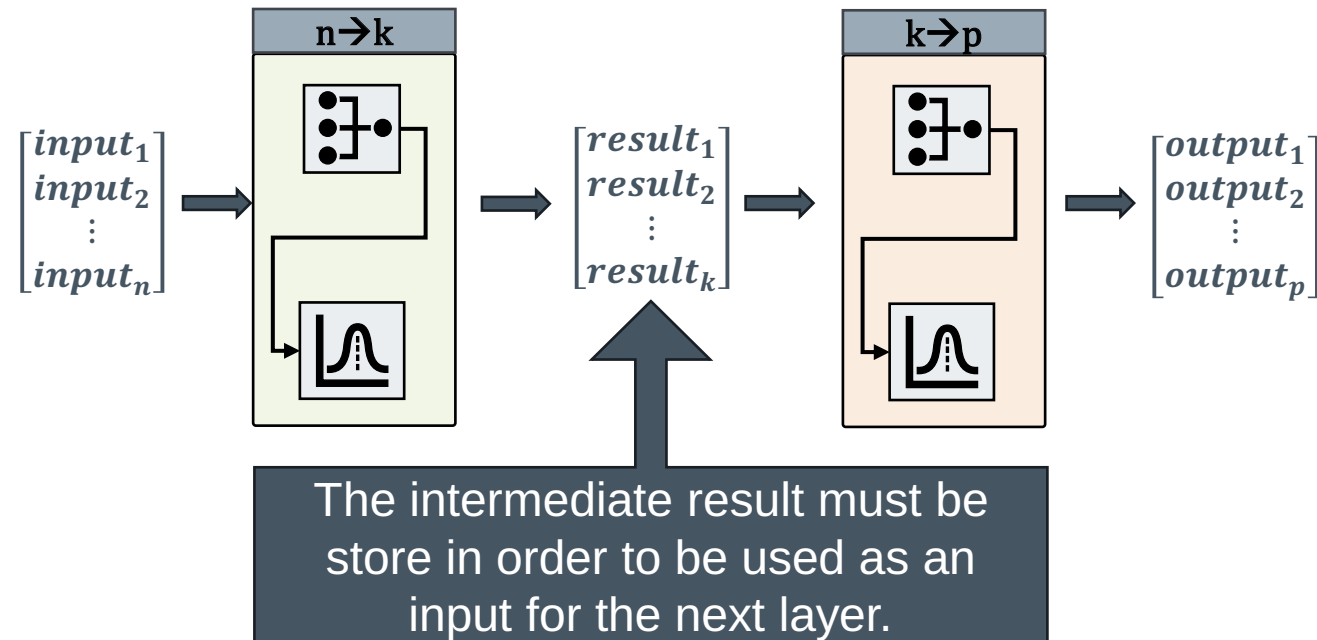
$$\text{params}(\text{model}) = 36 + 4 = 40 \text{ parameters}$$

This means that when you hear that a LLM model has let's say 7B parameters, this means that the sum of the number of parameters from each layer is 7 billions.

Space & memory considerations

It's also important to understand that the output of each layer need also to be stored in order to use it for:

- Feed-forward step (to compute the next layer)
- Backpropagation step (to compute the gradients)



Space & memory considerations

Depending on the batch size, the size for intermediate outputs can grow a lot (as it would in multiplied by the number of elements from the batch).

It is important to note that on the feedforward step we don't need to keep an intermediate output more than it is required to compute the next layer. Furthermore, the memory associated with that intermediate output can be reused.

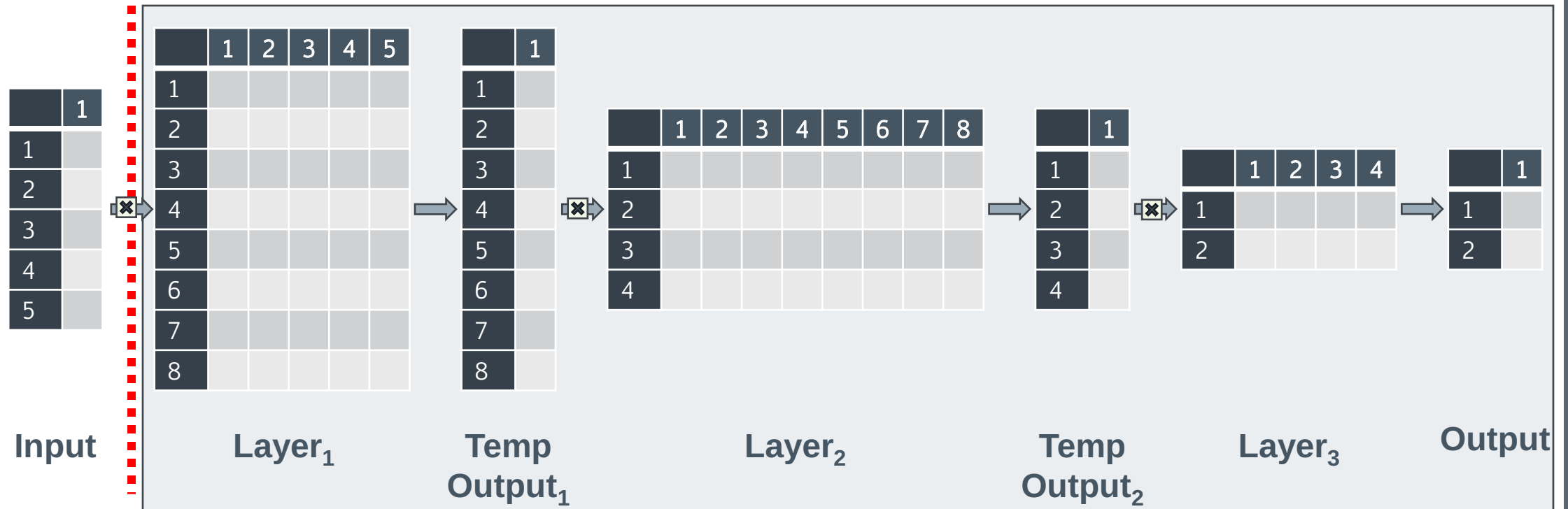
Space & memory considerations

Let's see an example:

- Layer 1 \rightarrow Linear(5,8)
- Layer 2 \rightarrow Linear(8,4)
- Layer 3 \rightarrow Linear(4,2)

Size required for feed forward:

$$\text{size} = 5 \times 8 + 1 \times 8 + 8 \times 4 + 1 \times 4 + 4 \times 2 + 1 \times 2 = 94$$



Space & memory considerations

Let's see an example:

- Layer 1 \rightarrow Linear(5,8)
- Layer 2 \rightarrow Linear(8,4)
- Layer 3 \rightarrow Linear(4,2)

Number of values to store = 94

For **float32** the space required is $94 \times 4 = 376$ bytes

If for example our batch is formed out of 100 elements then the space needed is $376 \times 100 = 37600$ bytes

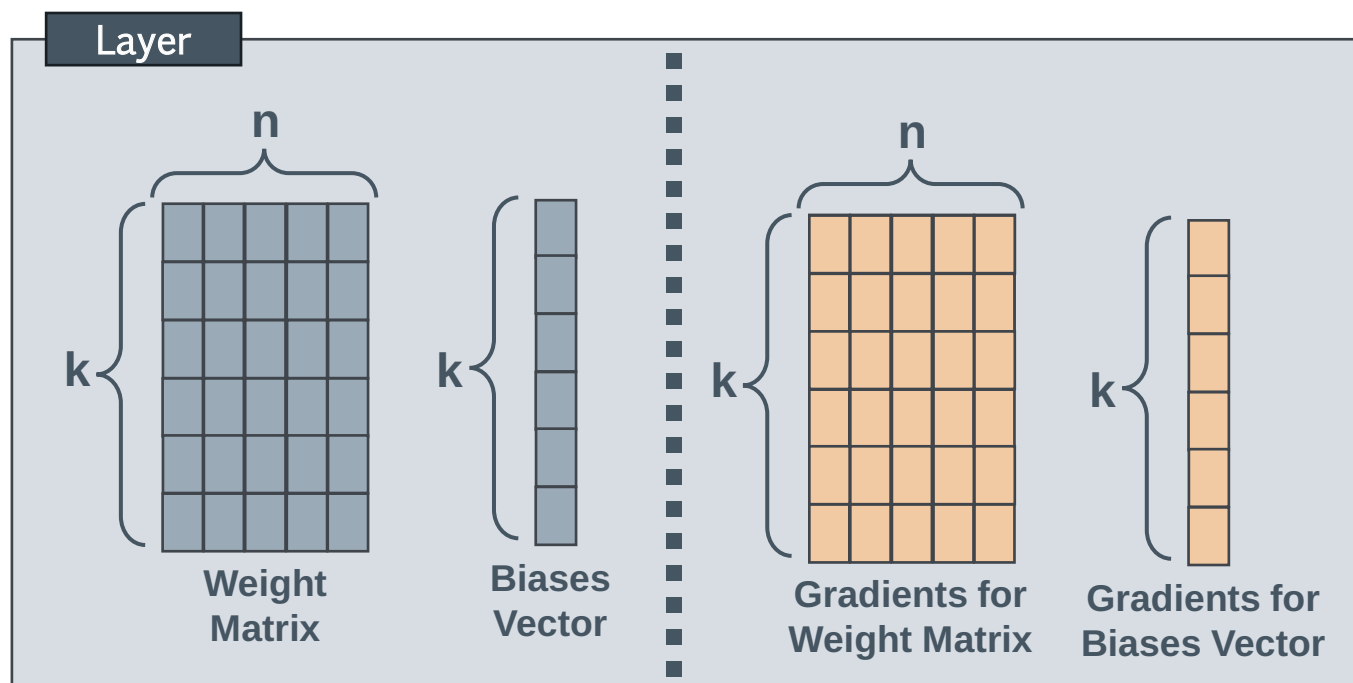
Space & memory considerations

But these consideration only refer to the inference part (feed forward part). Meaning that if you just want to test the model, this is the amount of memory that you will need.

When it comes to training however, the situation is quite different, as depending on various optimization the size of the model might change significantly.

Space & memory considerations

Using gradient descent implies that we first compute (**and store**) the gradients and then we update the weight and bias matrixes. This means that the actual representation of a layer would look like this:



These gradients are in fact the parameter `.grad` from a tensor

The gradients matrix and vector are cleaned (zeroed) whenever `optimizer.zero_grad()` is being called.

This also means that the training implies a size **twice as large** as the actual size of the model.

Space & memory considerations

Let's see an example:

```
import torch


model = torch.nn.Linear(4,1, bias=True)
loss_function = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
inputs = torch.randn(64, 4)
labels = torch.randn(64, 1)

# one iteration for training (so that we have gradients computed)
outputs = model(inputs)
loss = loss_function(outputs, labels)
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

```
print("Weights      =", model.weight.data)
print("Bias         =", model.bias.data)
print("Grad for Weights=", model.weight.grad)
print("Grad for Biases =", model.bias.grad)
```

Output

```
Weights      = tensor([[ -0.1739,  0.1598,  0.3838,  0.4105]])
Bias         = tensor([0.3439])
Grad for Weights= tensor([[0.0343, 0.6794, 0.6721, 0.7623]])
Grad for Biases = tensor([1.2795])
```



Momentum

Momentum

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

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

w_t = weights at the moment t

w_{t+1} = weights at the moment $t + 1$,

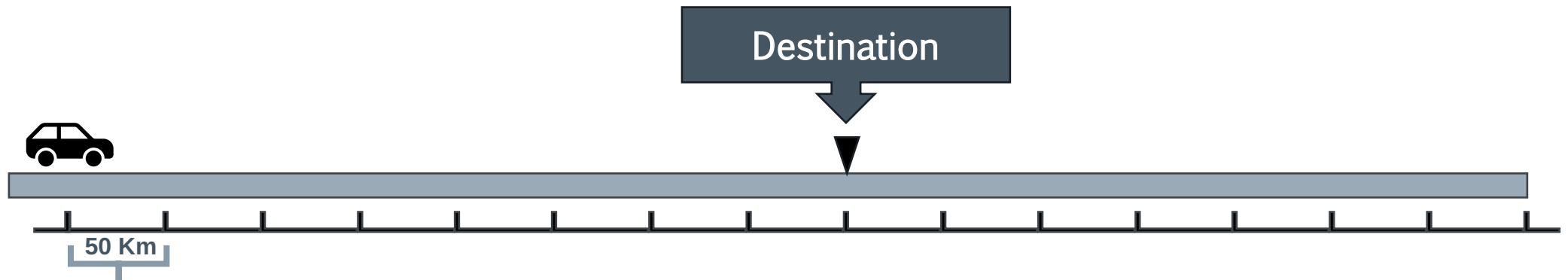
α = learning rate,

$\nabla \text{Loss}(w_t)$ = gradient of the loss function with respect to the w_t

Momentum

But updating the weights in this way can be ineffective. Let's consider the following scenario:

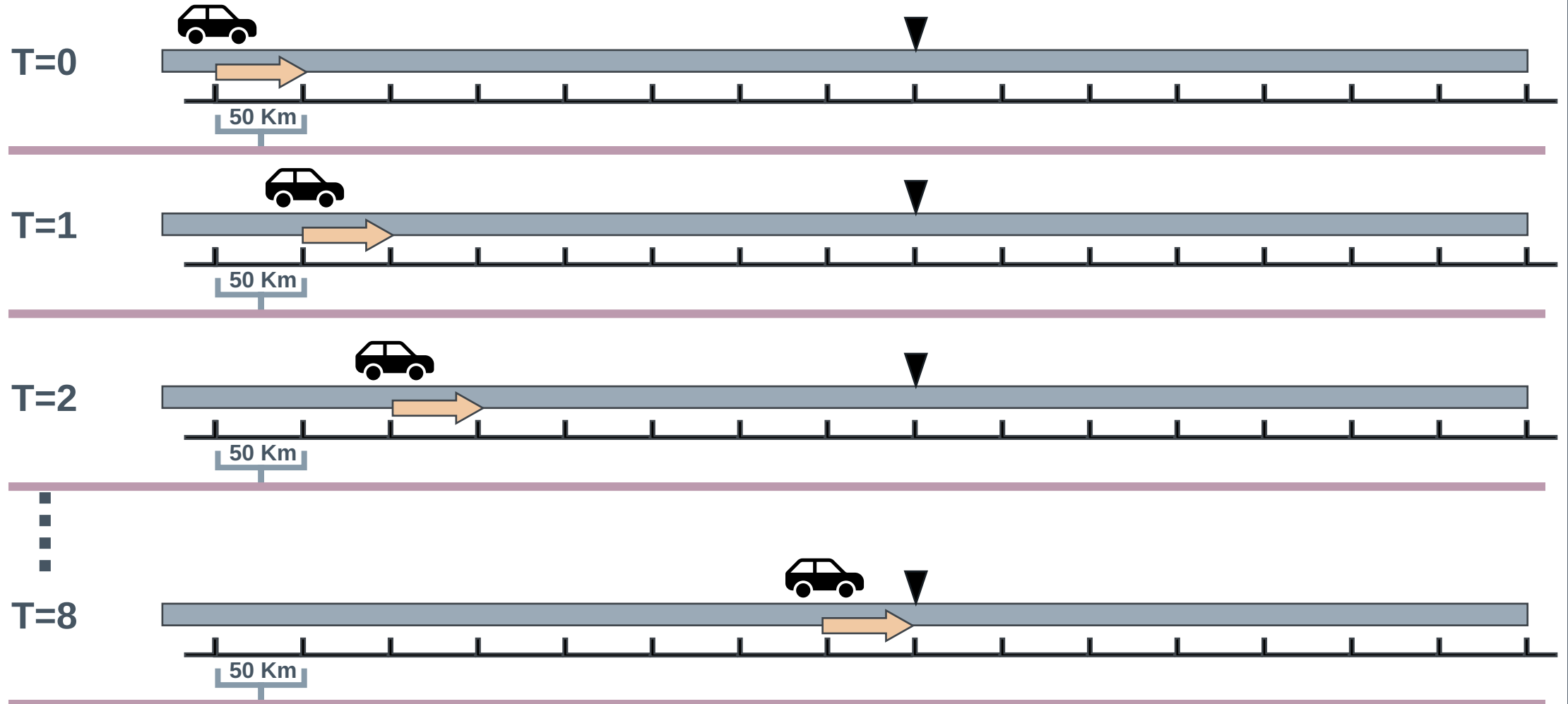
- we have a car that goes towards a destination with the speed of 50 Km/hour.
- the exact distance to the destination is unknown, however, we know if we are moving towards the destination or away from it.
- we always travel for one hour, then we evaluate if we are closer or not from to the destination. At this point we can also change the speed for the next hour.



π

Momentum

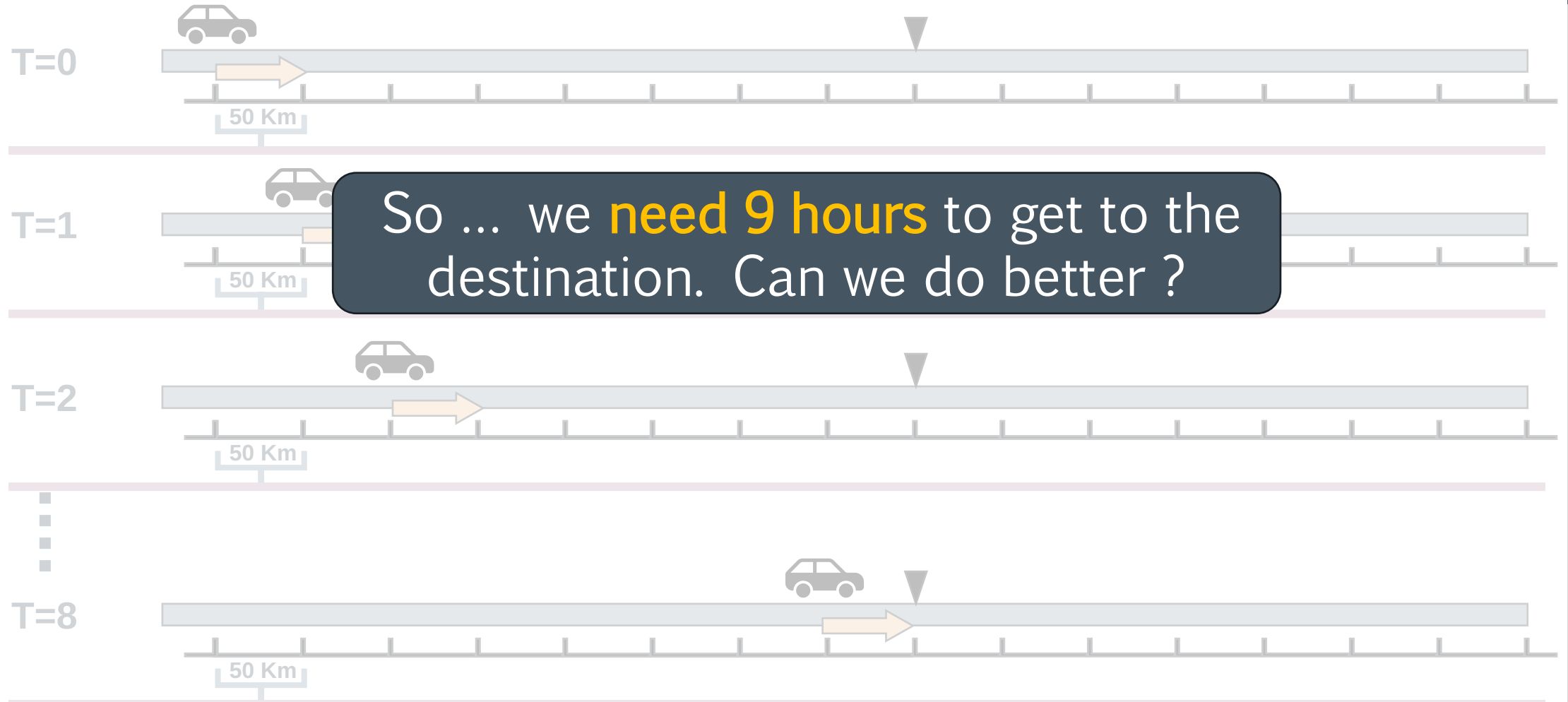
Scenario A: we never change the speed:



π

Momentum

Scenario A: we never change the speed:



Momentum

Scenario B: let's try a different algorithm. Remember that after every hour, the car can adjust its speed and knows if its direction is towards the destination or away from it.

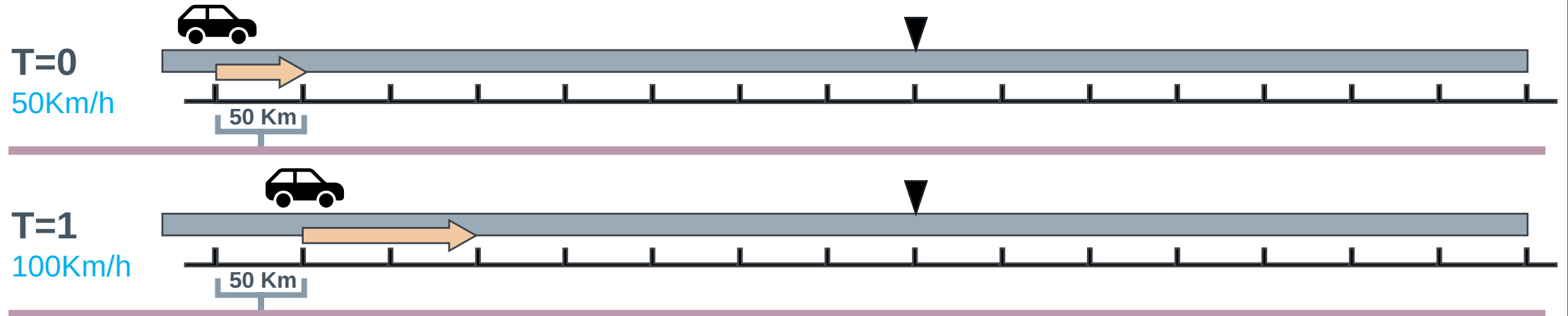
With this in mind, let's try the following algorithm – after every hour we adjust the speed in the following way:

- › if we move towards our destination, then we increase the speed with 50 Km/hours
- › If we move away from our destination, then we decrease the speed by 50 Km/hours and change the direction to face the destination point again.

π

Momentum

Scenario B: increase or decrease speed by 50 Km/hour

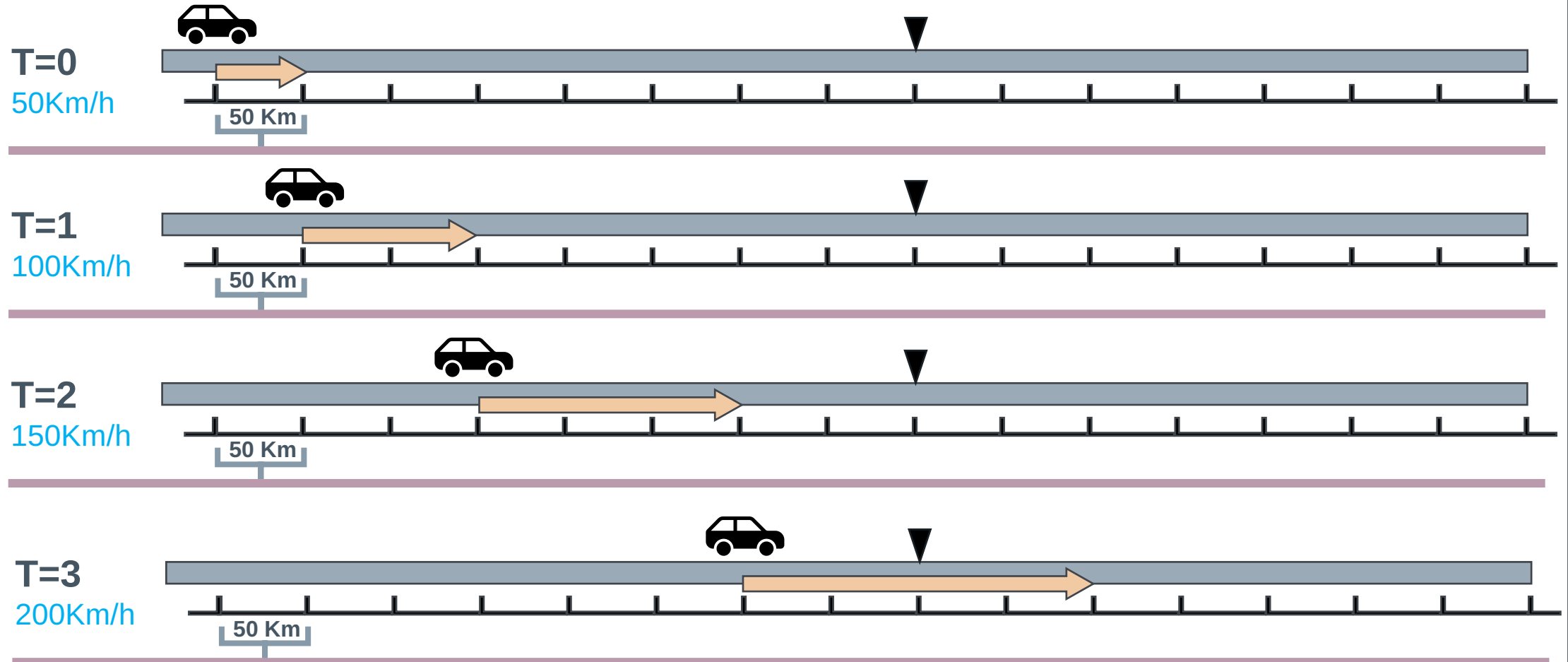


At this point we notice that we maintain the same direction as from the previous hour, and we are still moving towards our destination, so for the next hour we will increase the speed with 50 Km/h

π

Momentum

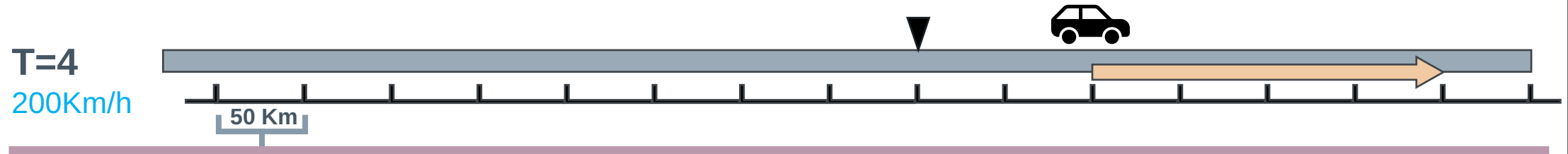
Scenario B: increase or decrease speed by 50 Km/hour



π

Momentum

Scenario B: increase or decrease speed by 50 Km/hour

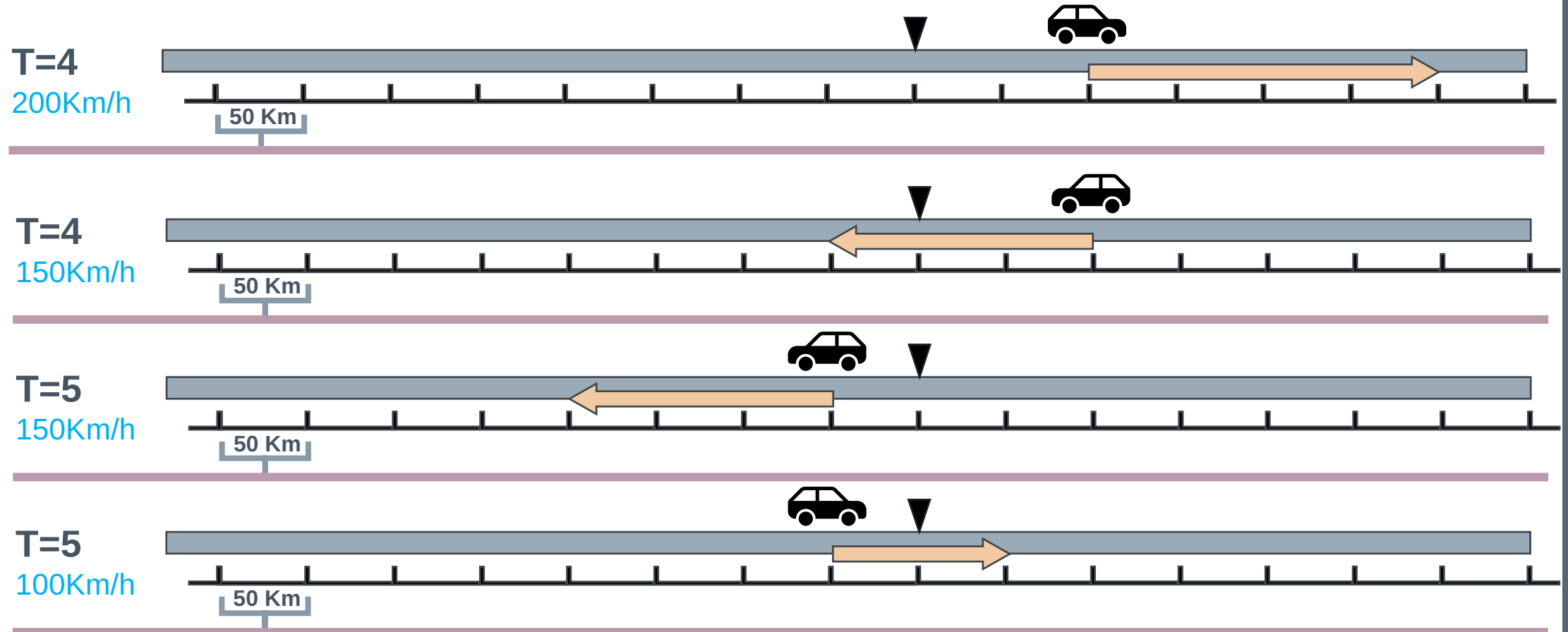


At this point if we continue on this direction, we will move away from our destination. As such, according to how our algorithm was presented, we will need to decrease the speed and change the direction.

π

Momentum

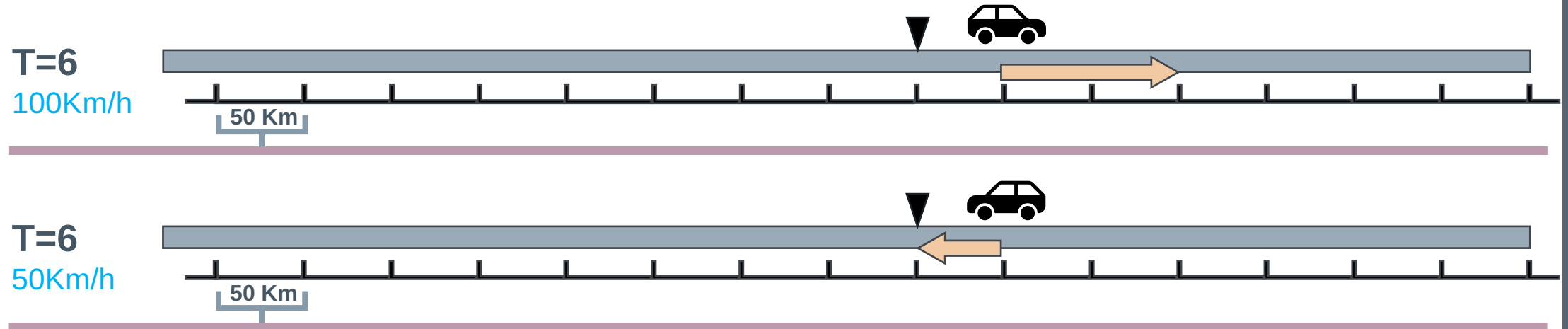
Scenario B: increase or decrease speed by 50 Km/hour



π

Momentum

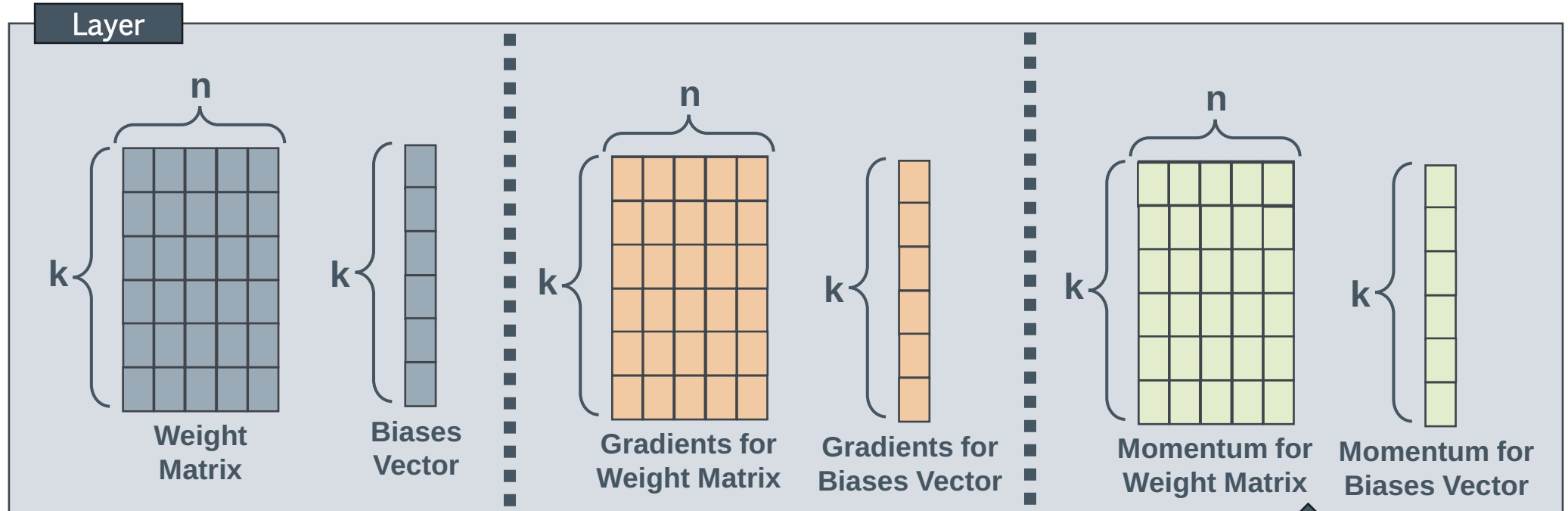
Scenario B: increase or decrease speed by 50 Km/hour



This mean that we could reach the destination in **7 hours** if we use this algorithm (that is more than 20% faster than in a previous case).

Momentum

This type of optimization is called ***Momentum*** and it implies some changes in the layer content and update computations:



In other words, we **triple** the size of a layer by adding a new matrix and bias vector.

Momentum

The momentum is computed in the following way:

$$m_t = \mu \times m_{t-1} + \nabla \text{Loss}(w_t), \text{ where}$$

m_t = current momentum,

m_{t-1} = previous momentum (with $m_0 = 0$)

μ = momentum coefficient,

$\nabla \text{Loss}(w_t)$ = gradient of the loss function with respect to the w_t

and the gradient update rule (the optimization rule) is:

$$w_{t+1} = w_t - \alpha \times m_t, \text{ where}$$

w_t = weights at the moment t

w_{t+1} = weights at the moment $t + 1$,

α = learning rate,

Momentum

The momentum coefficient (μ) is a value from bigger than 0.01 and smaller than 1. Notice that if μ is 0, we get the same equation as the one from gradient descent. Common values for μ are:

- **0.9**: This is usually a default choice for μ . It indicates that 90% of the previous momentum is retained in the current update.
- **0.99**: Sometimes, a higher value like 0.99 is used, especially in cases where a smoother convergence is desired. This retains 99% of the previous momentum.
- **Between 0.5 and 0.9**: In some cases, values in this range are chosen, especially if there's a need to reduce the influence of past gradients to make the optimization more responsive to recent changes.

Momentum

The momentum coefficient (μ) is a value from bigger than 0.01 and smaller than 1. Notice that if μ is 0, we get the same equation as the one from gradient descent. Common values for μ are:

- **0.9**: This is usually a default choice for μ . It indicates that 90% of the previous momentum is retained in the current update.
- **0.99**: Sometimes, a higher value like 0.99 is used, especially in cases where a smoother convergence is desired. This retains 99% of the previous momentum.
- **Between 0.5 and 0.9**: In some cases, values in this range are chosen, especially if there's a need to reduce the influence of past gradients to make the optimization more responsive to recent changes.

Momentum

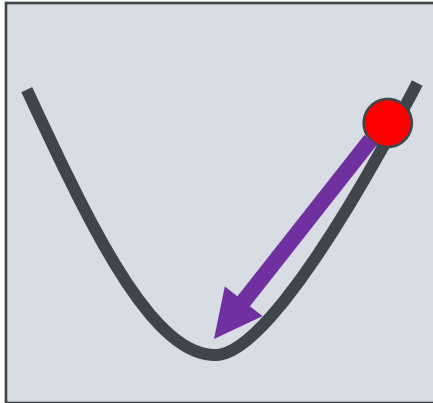
The momentum is effective to navigate through scenarios where local minimums are frequent. The most important takeaways (related to how momentum works) are:

- › **Increased Step Size:** If the optimization path is consistent (i.e., the gradients point in the same general direction), momentum builds up and increases the step size, accelerating convergence.
- › **Decreased Step Size:** If the gradients change direction, momentum helps to stabilize the updates by reducing the step size, preventing drastic changes in the path.

Also keep in mind the momentum triples the size of one layer (if memory concerns are relevant, this should be taken under considerations).

Momentum

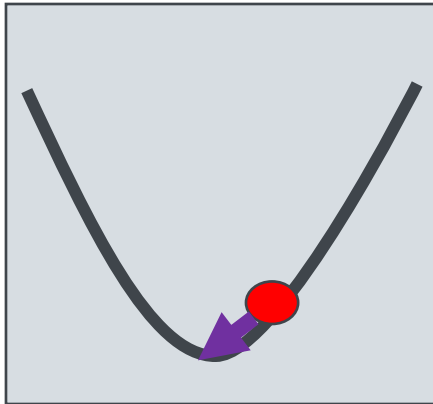
Let's discuss a couple of advantages of momentum. Let's consider the following minimum that we want to reach with gradient descent:



In this case, it is clear that the slope is quite high and as a result, the gradient descent will converge **closer to the minimum** quickly from this position.

Momentum

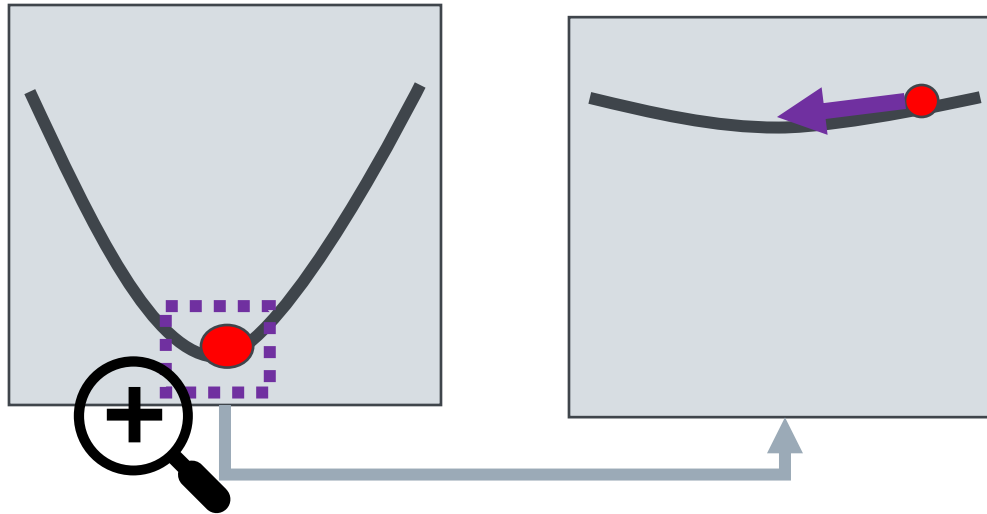
Let's discuss a couple of advantages of momentum. Let's consider the following minimum that we want to reach with gradient descent:



The closer we get to a local minimum, **the slower the gradient descent advances towards** that minimum.

Momentum

Let's discuss a couple of advantages of momentum. Let's consider the following minimum that we want to reach with gradient descent:



When we almost reach the minimum, the gradients are close to 0. As a result, the model advances really slow towards the minimum. However, using momentum can help in this cases and speed up the convergence process.

Momentum

Using momentum with small learning rates is recommended:

- Small learning rates make the **updates to the parameters more gradual and controlled**, reducing the risk of overshooting the minimum of the loss function.
- Momentum helps to accelerate the optimization by accumulating a fraction of past gradients. This means even with a small learning rate, the accumulation effect of momentum can lead to significant updates, especially if the gradients are consistently pointing in the same direction over multiple iterations.
- The combination of a small learning rate and momentum allows for a **balance between exploration** (navigating through the loss landscape to avoid local minima) **and exploitation** (fine-tuning the parameters to reach the lowest point in a minimum).
- In scenarios where larger learning rates cause **oscillations around minima, adding momentum while keeping the learning rate small can smooth out these oscillations**, leading to more stable convergence

Momentum

In Pytorch, there isn't a special optimizer for momentum (keep in mind that if **momentum coefficient** is 0, then momentum equation is the same as with gradient descent).

Instead, SGD optimizer provides a way to use momentum through its parameters:

```
torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0)
```

* torch.optim.SGD has multiple parameters (we will go through some of them once we discuss other optimization techniques)

Momentum

So ... what is the role of the dampening parameter (τ) from the SGD constructor ?

```
torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0)
```

First of all, it implies changing the formula for how momentum is computed in the following way:

$$m_t = \mu \times m_{t-1} + (1 - \tau) \times \nabla \text{Loss}(w_t), \text{ where}$$

m_t = current momentum,
 m_{t-1} = previous momentum (with $m_0 = 0$)
 μ = momentum coefficient,
 τ = dampening coefficient,
 $\nabla \text{Loss}(w_t)$ = gradient of the loss function with respect to the w_t

Notice that if $\tau = 0$ we have the Momentum formula !

Momentum

- › The dampening parameter (τ) essentially controls how much of the current gradient is used to update the momentum term.
- › A higher dampening value means that the momentum term relies more on the accumulated (past) gradients and less on the current gradient. This can lead to smoother updates but may also increase the risk of relying too heavily on past information, potentially leading to slower convergence.
- › A smaller value implies that the current gradient is more relevant when computing the momentum than the accumulated (past) gradients.

Momentum

The dampening parameter in Stochastic Gradient Descent with momentum is typically used in scenarios where you want to control the impact of the current gradient on the momentum term, such as:

- **Fine-Tuning Optimization Dynamics:**
- **Dealing with Noisy Gradients:**
- **Adjusting Learning Behavior:**
- **Experimentation and Hyperparameter Tuning:**
- **Specific Model Training Scenarios:**
- **Complementing Learning Rate Schedules:**

Momentum

Let's see some examples on how to instantiate a momentum optimizer with different parameter in Pytorch:

```
# Gradient Descent
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Momentum (coefficient=0.9)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

# Momentum (coefficient=0.99)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.99)

# Momentum (coefficient=0.99, dampening=25%)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.99, dampening=0.25)
```


Momentum

The dampening parameter in Stochastic Gradient Descent with momentum is typically used in scenarios where you want to control the impact of the current gradient on the momentum term, such as:

- **Fine-Tuning Optimization Dynamics:**
- **Dealing with Noisy Gradients:**
- **Adjusting Learning Behavior:**
- **Experimentation and Hyperparameter Tuning:**
- **Specific Model Training Scenarios:**
- **Complementing Learning Rate Schedules:**

Nesterov Accelerated Gradient

Nesterov Accelerated Gradient

Another optimization technique is Nesterov Accelerated Gradient (NAG), that is an adaptation of the momentum optimization technique.

The basic idea of NAG is to make a smarter use of the momentum term by first making a big step in the direction of the previous momentum, and then correcting this step based on the gradient at the new (look-ahead) position.

Nesterov Accelerated Gradient

The NAG is computed in the following way:

$$m_t = \mu \times m_{t-1} + \nabla \text{Loss}(w_t - \mu \times m_{t-1}), \text{ where}$$

m_t = current momentum,

m_{t-1} = previous momentum (with $m_0 = 0$)

μ = momentum coefficient,

$\nabla \text{Loss}(w_t - \mu \times m_{t-1})$ = gradient of the loss function with respect to $w_t - \mu \times m_{t-1}$

and the gradient update rule (the optimization rule) is:

$$w_{t+1} = w_t - \alpha \times m_t, \text{ where}$$

w_t = weights at the moment t

w_{t+1} = weights at the moment $t + 1$,

α = learning rate,

Nesterov Accelerated Gradient

The NAG is computed in the following way:

$$m_t = \mu \times m_{t-1} + \nabla \text{Loss}(w_t - \mu \times m_{t-1}), \text{ where}$$

m_t = current momentum,
 m_{t-1} = previous momentum (with $m_0 = 0$)
 μ = momentum coefficient

$\nabla \text{Loss}(w_t -$
 and the grad

Notice that the only difference between NAG and momentum is how the loss function is being calculated. While momentum is using $\nabla \text{Loss}(w_t)$, NAG first computes an intermediate parameter position based on the accumulated momentum from the previous step ($w_t - \mu \times m_{t-1}$), and then computes the gradient.

$$w_{t+1} = w_t - \alpha \times m_t, \text{ where}$$

w_t = weights at the moment t
 w_{t+1} = weights at the moment $t + 1$,
 α = learning rate,

Nesterov Accelerated Gradient

The 'look-ahead' step is the key to NAG's effectiveness. By calculating the gradient at the look-ahead position, NAG anticipates the future landscape of the loss function.

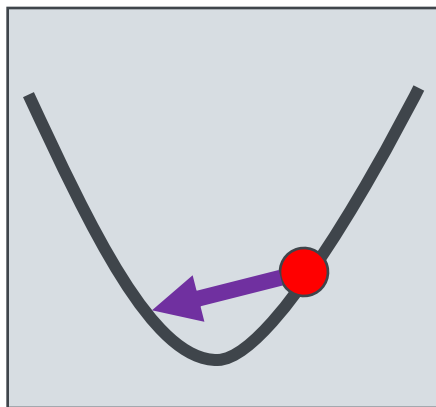
If this look-ahead step results in a parameter position that is in a less favorable direction (e.g., heading towards a higher loss), the gradient computed there will correct this, forcing the updates towards a more optimal position.

This is particularly useful in scenarios where the loss surface has sharp curves or local minima, as it allows NAG to adjust its trajectory more intelligently than standard momentum, which only looks at the past gradients.

Nesterov Accelerated Gradient

As a general idea:

- › NAG often converges faster and more effectively than standard momentum, especially in complex, high-dimensional spaces typical in deep learning.
- › It is more stable and less prone to the problem of **overshooting the minimum**, a common issue with algorithms that rely heavily on momentum.



Overshooting happens when the update steps during the optimization process are too large, causing the algorithm to miss the minimum of the loss function and jump over to the other side of the curve. This can happen **if the learning rate is set too high** or if the **momentum term accumulates excessively without sufficient correction**.

Nesterov Accelerated Gradient

In Pytorch, since this is but an adaptation of momentum, there is no specific optimizer for NAG. However, there is another parameter that can be used when creating a SGD that implies using a NAG technique: **nesterov**

```
torch.optim.SGD(params, lr=<required parameter>,  
                momentum=0,  
                dampening=0,  
                nesterov=False)
```

It's also important to notice that you still need to select a momentum parameter and you can also use the dampening property as well.

Resilient Backpropagation

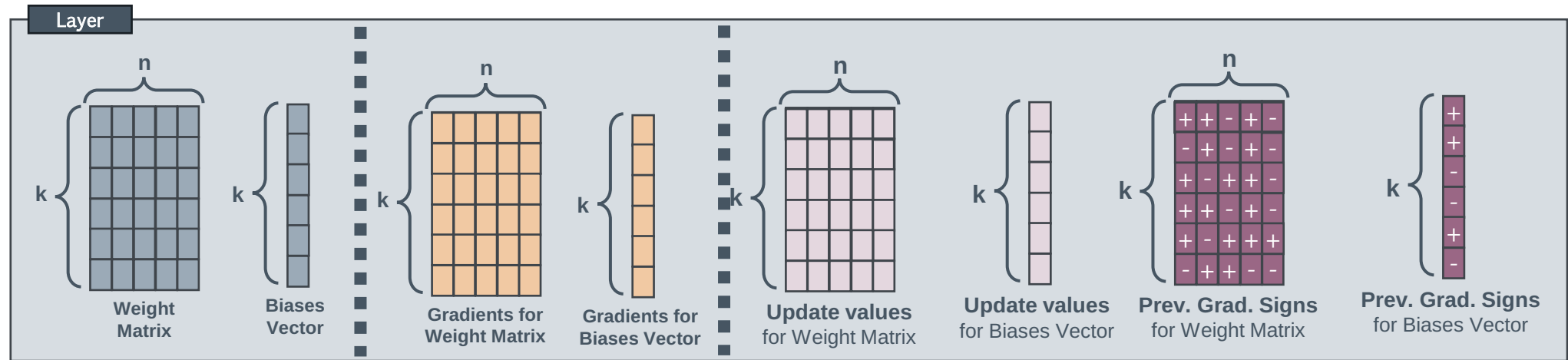
Resilient Backpropagation (Rprop)

Resilient backpropagation (Rprop) is an optimization algorithm designed to eliminate some of the issues faced by gradient descent methods.

RProp focuses solely on the direction of the gradient and not its magnitude. This means that the update is performed based on the sign of the gradient for each individual weight and bias in the network.

Resilient Backpropagation (Rprop)

Just like momentum, a separate set of values (update values) and the signed of the previous gradient have to be stored for each weight and bias from the network



In other words, we at least **triple** the size of a layer by adding a two new matrixes and two new bias vector. The sign matrixes and bias vector can be optimized (by using **bit-sets**)

Resilient Backpropagation (Rprop)

The Rprop algorithm is being performed in 3 steps:

1. Update the weights

$$w_{t+1} = w_t - U_t \times \text{sign}(\nabla \text{Loss}(w_t)), \text{ where}$$

w_t = weights at the moment t

w_{t+1} = weights at the moment $t + 1$,

U_t = Update values ($U_0 = 0$),

$\nabla \text{Loss}(w_t)$ = gradient of the loss function with respect to the w_t

2. Update the values

$$U_{i,j} = \begin{cases} U_{i,j} \times \eta^+, & \text{sign}(\nabla \text{Loss}(w_{i,j})) = SGN_{i,j} \\ U_{i,j} \times \eta^-, & \text{sign}(\nabla \text{Loss}(w_{i,j})) \neq SGN_{i,j} \end{cases} \quad \text{where}$$

η^+ = an increase coefficient ($\eta^+ > 1$), η^- = a decrease coefficient ($0 < \eta^- < 1$)

$SGN_{i,j}$ = sign of the previous gradient for element (i,j) from weight matrix,

3. Store the current gradient signs

$$SGN_{i,j} = \text{sign}(\nabla \text{Loss}(w_{i,j}))$$

Resilient Backpropagation (Rprop)

The η^+ and η^- typically have values as follows:

- $\eta^+ > 1$, a typical value is 1.2 (increase by 20%)
- $\eta^- < 1$, a typical value is 0.5 (decrease by 50%)

In practice, there are also some limits that are being set up so that the update values don't go outside a specific range. These limits are:

- Minimum value: $1e-06 = 0.00000001$
- Maximum value: 50

Resilient Backpropagation (Rprop)

The η^+ and η^- typically have values as follows:

- $\eta^+ > 1$, a typical value is 1.2 (increase by 20%)
- $\eta^- < 1$, a typical value is 0.5 (decrease by 50%)

In practice, there are also some limits that are being set up so that the update values don't go outside a specific range. These limits are:

- Minimum value: $1e-06 = 0.00000001$
- Maximum value: 50

Resilient Backpropagation (Rprop)

Advantages of Rprop:

1. Eliminates Learning Rate:

2. Scale-Invariant Updates:

- The updates do not depend on the magnitude of the gradient, only on the sign.

3. Faster Convergence:

- RProp can converge faster than standard gradient descent because it dynamically adapts the step sizes for each weight, potentially avoiding problems with choosing an appropriate learning rate.

4. Stable Convergence:

- It tends to be more stable and less sensitive to the specific topology of the error landscape, as it is not as affected by small, noisy gradients or very steep gradients.

5. Robust to Initial Parameters:

Resilient Backpropagation (Rprop)

Disadvantages of Rprop:

› Not Suitable for Mini-Batch Learning:

- RProp is generally not well-suited for mini-batch learning, as it was designed for full-batch training. Its updates can be too aggressive when gradients are estimated from small subsets of the data.

› No Momentum

› Poor Online Learning Performance

› Parameter Oscillation

- If the gradient sign changes frequently, RProp can lead to oscillations in parameter updates, especially near local minima, potentially causing the model to miss the minima.

Resilient Backpropagation (Rprop)

In PyTorch there is an optimizer defined for Rprop:

```
torch.optim.Rprop(params, etas=(0.5, 1.2))
```

Where:

› etas tuple corresponds to η^+ and η^- values

And a simple example:

```
# Rproc (eta-minus = 0.75, eta-plus = 1.1)  
optimizer = torch.optim.Rprop(model.parameters(), etas=(0.75, 1.1))
```

Adaptive Gradient Algorithm

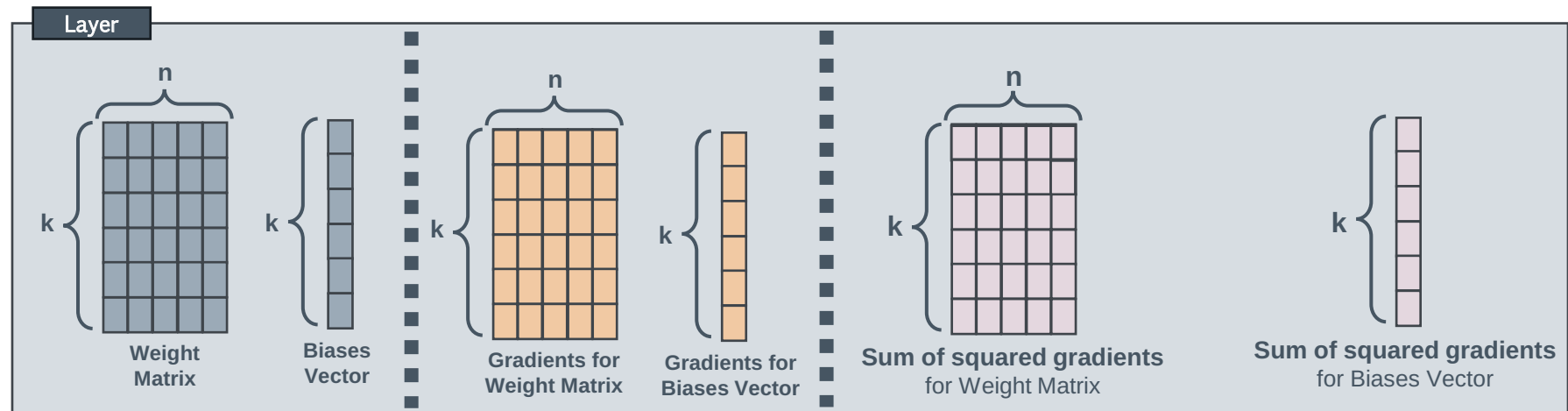
Adaptive Gradient Algorithm (AdaGrad)

Adaptive Gradient Algorithm (AdaGrad) is an optimization method that provides an adaptive learning rate for each parameter.

The learning rate for each parameter is adapted by scaling it with the inverse square root of the sum of all past squared gradients for that parameter. Parameters with larger past gradients receive a smaller learning rate, and the ones with smaller past gradients receive a bigger learning rate.

Adaptive Gradient Algorithm (AdaGrad)

In case of AdaGrad, we need to store for each parameter is the sum of the squares of the gradients. This sum is then used to scale the global learning rate for each parameter individually during the update step.



In other words, we **triple** the size of a layer by adding a matrix and vector for each later

Adaptive Gradient Algorithm (AdaGrad)

The update equation for AdaGrad is:

$$w_{i,j}^{t+1} = w_{i,j}^t - \frac{\eta}{\sqrt{\text{Sum}_{i,j}^t + \varepsilon}} \times \nabla \text{Loss} \left(w_{i,j}^t \right),$$
$$\text{Sum}_{i,j}^t = \text{Sum}_{i,j}^{t-1} + \nabla \text{Loss} \left(w_{i,j}^t \right)^2$$

$w_{i,j}^t$ = weight (i,j) in the weight matrix at the moment t

$w_{i,j}^{t+1}$ = weight (i,j) in the weight matrix at the moment t + 1

$\text{Sum}_{i,j}^t$ = Square sum of all gradients for element (i,j) in the weight matrix at the moment t

$\text{Sum}_{i,j}^{t-1}$ = Square sum of all gradients for element (i,j) in the weight matrix at the moment t - 1

η = a global learning rate,

ε = a very small but bigger than 0 value (needed to ensure that the denominator > 0)

$\nabla \text{Loss} \left(w_{i,j}^t \right)$ = gradient of the loss function with respect to the $w_{i,j}^t$

Adaptive Gradient Algorithm (AdaGrad)

Advantages:

- AdaGrad can be less sensitive to the global learning rate η because it makes fine-scale adjustments to each parameter.
- It's also beneficial for dealing with sparse data and features that have different frequencies of updates.

Disadvantages:

- One of the main drawbacks of AdaGrad is that the accumulated squared gradients in the denominator can become very large over time, causing the learning rate to shrink to a point where it changes the weights very little. This usually translates that after a while the algorithm “stop learning”.

Adaptive Gradient Algorithm (AdaGrad)

In PyTorch there is an optimizer defined for AdaGrad:

```
torch.optim.Adagrad(params, lr=0.01, initial_accumulator_value=0, eps=1e-10)
```

Where:

- › **lr** corresponds to the global learning rate (η)
- › **eps** is that small value (ε) used to prevent division by 0 if the denominator reaches 0 value

And a simple example:

```
# Adagrad (learning rate = 0.2)  
optimizer = torch.optim.Adagrad(model.parameters(), lr=0.2)
```

Root Mean Square Propagation

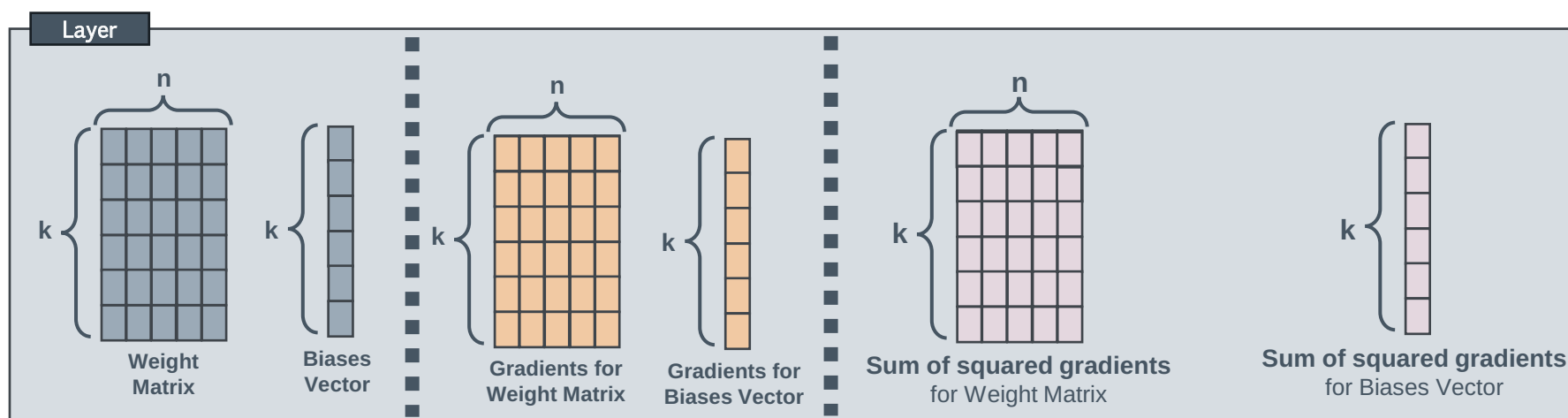
Root Mean Square Propagation (RMSprop)

Root Mean Square Propagation (RMSprop) is an adaptive learning rate optimization algorithm, based on AdaGrad and design to fix some of the problems AdaGrad has.

RMSprop modifies the gradient accumulation process of AdaGrad by using an *exponentially weighted moving average* of the squared gradients. This means it doesn't accumulate all past squared gradients but instead gives more weight to the more recent gradients.

Root Mean Square Propagation (RMSprop)

Just like in the case of AdaGrad, we need to store for each parameter is the sum of the squares of the gradients. This sum is then used to scale the global learning rate for each parameter individually during the update step.



In other words, we **triple** the size of a layer by adding a matrix and vector for each later

Root Mean Square Propagation (RMSprop)

The update equation for RMSprop is:

$$w_{i,j}^{t+1} = w_{i,j}^t - \frac{\eta}{\sqrt{\text{Sum}_{i,j}^t + \varepsilon}} \times \nabla \text{Loss} \left(w_{i,j}^t \right),$$

$$\text{Sum}_{i,j}^t = \gamma \times \text{Sum}_{i,j}^{t-1} + (1 - \gamma) \times \left(\nabla \text{Loss} \left(w_{i,j}^t \right)^2 \right)$$

$w_{i,j}^t$ = weight (i,j) in the weight matrix at the moment t

$w_{i,j}^{t+1}$ = weight (i,j) in the weight matrix at the moment t + 1

$\text{Sum}_{i,j}^t$ = Square sum of all gradients for element (i,j) in the weight matrix at the moment t

$\text{Sum}_{i,j}^{t-1}$ = Square sum of all gradients for element (i,j) in the weight matrix at the moment t - 1

η = a global learning rate,

γ = a decay rate (a parameter between 0 and 1, usually 0.9)

ε = a very small but bigger than 0 value (needed to ensure that the denominator > 0)

$\nabla \text{Loss} \left(w_{i,j}^t \right)$ = gradient of the loss function with respect to the $w_{i,j}^t$

Root Mean Square Propagation (RMSprop)

Advantages:

- RMSprop avoids the problem of a rapidly diminishing learning rate encountered in AdaGrad by using a moving average (newer values count more than older ones).
- By using the moving average of the squared gradients, RMSprop automatically scales the learning rate with respect to the history of the gradients.

Disadvantages:

- Often the decay rate needs to be tuned (value of 0.9 is usually OK but for different cases different values are needed)..
- The performance of RMSprop can be sensitive to the initialization of Sum matrix and the (ϵ) value
- RMSprop does not really provides guarantee of convergence.

Root Mean Square Propagation (RMSprop)

In PyTorch there is an optimizer defined for RMSprop:

```
torch.optim.RMSprop(params, lr=0.01, initial_accumulator_value=0, eps=1e-10, alpha=0.99)
```

Where:

- › **lr** corresponds to the global learning rate (η)
- › **eps** is that small value (ε) used to prevent division by 0 if the denominator reaches 0 value
- › alpha corresponds to the decay rate (γ)

And a simple example:

```
# RMSprop (learning rate = 0.01, decay rate = 0.9)  
optimizer = torch.optim.RMSprop(model.parameters(), lr=0.01, alpha=0.9)
```

Adaptive Moment Estimation

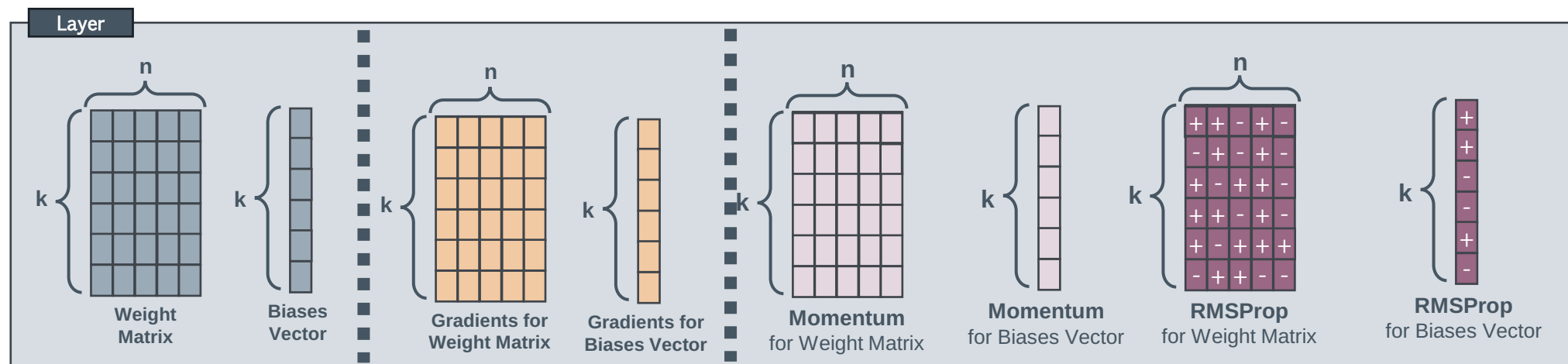
Adaptive Moment Estimation (ADAM)

Addaptive Moment Estimation (ADAM) is an optimization algorithm that combines ideas from both Momentum and RMSprop to update network weights.

Adam calculates an exponential moving average of the gradient and the squared gradient.

Adaptive Moment Estimation (ADAM)

In case of Adam we need to store two matrixes (one for momentum) and another one for the RMSProp-like computations.



In other words, we at **quadruple** the size of a layer by adding a two new matrixes and two new bias vector.

Adaptive Moment Estimation (ADAM)

Let's see the equation for the momentum (also called the first momentum):

$$m_t = \beta_1 \times m_{t-1} + (1 - \beta_1) \times \nabla \text{Loss}(w_t), \text{ where}$$

m_t = current momentum,

m_{t-1} = previous momentum (with $m_0 = 0$)

β_1 = exponential decay rate for the first moment estimates, typically around 0.9,

$\nabla \text{Loss}(w_t)$ = gradient of the loss function with respect to the w_t

Since m_0 is instantiated with zeros, m_t is biased toward zero, especially during the initial time steps or if β_1 is close to 1. To correct this, we will use bias-corrected first moment estimate:

$$\text{bias corrected} = \hat{m}_t = \frac{m_t}{1 - (\beta_1)^t}, \text{ where}$$

$(\beta_1)^t$ = β_1 at the power of t (the bigger t is the less important the denominator is)

Adaptive Moment Estimation (ADAM)

Let's see the equation for the RMSprop (also called the second momentum):

$$r_t = \beta_2 \times r_{t-1} + (1 - \beta_2) \times (\nabla \text{Loss}(w_t))^2, \text{ where}$$

r_t = second momentum (rmsprop) computed at time t ,
 r_{t-1} = second momentum (rmsprop) computed at time $t - 1$ (with $r_0 = 0$)
 β_2 = exponential decay rate for the second moment estimates, typically 0.999,
 $\nabla \text{Loss}(w_t)$ = gradient of the loss function with respect to the w_t

Since r_0 is instantiated with zeros, r_t is biased toward zero, especially during the initial time steps or if β_2 is close to 1. To correct this, we will use bias-corrected first moment estimate:

$$\text{bias corrected} = \hat{r}_t = \frac{r_t}{1 - (\beta_2)^t}, \text{ where}$$

$(\beta_2)^t = \beta_2$ at the power of t (the bigger t is the less important the denominator is)

Adaptive Moment Estimation (ADAM)

The update formula is the following:

$$w_{t+1} = w_t - \eta \times \frac{\hat{m}_t}{\sqrt{\hat{r}_t + \varepsilon}}, \text{ where}$$

w_t = weights at the moment t

w_{t+1} = weights at the moment $t + 1$,

η = learning rate

ε = a small value to prevent division by zero,

\hat{m}_t = first momentum (with bias corection),

\hat{r}_t = second momentum (with bias corection)

Adaptive Moment Estimation (ADAM)

Advantages:

- Adam adjusts the learning rate for each parameter individually based on estimates of first (mean) and second (uncentered variance) moments of the gradients.
- efficient for problems with large datasets and high-dimensional parameter spaces, which is typical in deep learning.
- well-suited for problems with sparse gradients (e.g., Natural Language Processing and Computer Vision tasks)
- Has a bias corrections to the first and second moment estimates, which counteract the biases toward zero that might occur especially in the initial time steps.
- requires less tuning of the hyperparameters (ADAM can often be used out of the box with default settings and it produces good results).

Adaptive Moment Estimation (ADAM)

Disadvantages:

- Sensitive to Initial Learning Rate.
- Potential for Overfitting (due to its rapid convergence)

Adaptive Moment Estimation (ADAM)

In PyTorch there is an optimizer defined for ADAM:

```
torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08,)
```

Where:

- › **lr** corresponds to the global learning rate (η)
- › **eps** is that small value (ε) used to prevent division by 0 if the denominator reaches 0 value
- › **betas** corresponds to exponential decay rates (β_1 and β_2)

And a simple example:

```
# Adam with default parameters  
optimizer = torch.optim.Adam(model.parameters())
```

Nesterov-accelerated Adaptive Moment Estimation

Nesterov-accelerated Adaptive Moment Estimation

Nesterov-accelerated Adaptive Moment Estimation (NADAM), is an optimization algorithm that combines the ideas of Nesterov momentum with Adam. It's essentially Adam with Nesterov momentum integrated into the moment estimates.

NADAM incorporates the Nesterov momentum by modifying the way the first moment (the moving average of the gradients) is calculated. Instead of using the current gradient to update the first moment as in Adam, NADAM uses the lookahead gradient, as is done in Nesterov accelerated gradient (NAG).

Nesterov-accelerated Adaptive Moment Estimation

NADAM uses the same update formulas as ADAM for m_t, r_t and \hat{r}_t . The weights are also updated in a similar way:

$$w_{t+1} = w_t - \eta \times \frac{\hat{m}_t}{\sqrt{\hat{r}_t} + \varepsilon}$$

but parameter \hat{m}_t is computed in a different way:

$$\text{bias corrected} = \hat{m}_t = \beta_1 \times \frac{m_t}{1 - (\beta_1)^t} + \frac{(1 - \beta_1)}{1 - (\beta_1)^t}$$

Nesterov-accelerated Adaptive Moment Estimation

Observations:

- NADAM maintains the same advantages and disadvantages as ADAM
- potentially faster convergence and better performance on certain problems
- It is particularly useful for tasks where the benefits of Nesterov momentum, which anticipates the future gradient, are relevant
- Like Adam, Nadam can sometimes converge rapidly to suboptimal solutions for certain kinds of problems, especially those with noisy or sparse gradients.
- It can be more sensitive to the choice of hyperparameters compared to Adam, due to the additional complexity introduced by the Nesterov term.

Nesterov-accelerated Adaptive Moment Estimation

In PyTorch there is an optimizer defined for ADAM:

```
torch.optim.NAdam(params, lr=0.002, betas=(0.9, 0.999), eps=1e-08, momentum_decay=0.004,)
```

Where:

- › **lr** corresponds to the global learning rate (η)
- › **eps** is that small value (ε) used to prevent division by 0 if the denominator reaches 0 value
- › **betas** corresponds to exponential decay rates (β_1 and β_2)

And a simple example:

```
# NAdam with default parameters  
optimizer = torch.optim.NAdam(model.parameters())
```

AdaDelta

AdaDelta

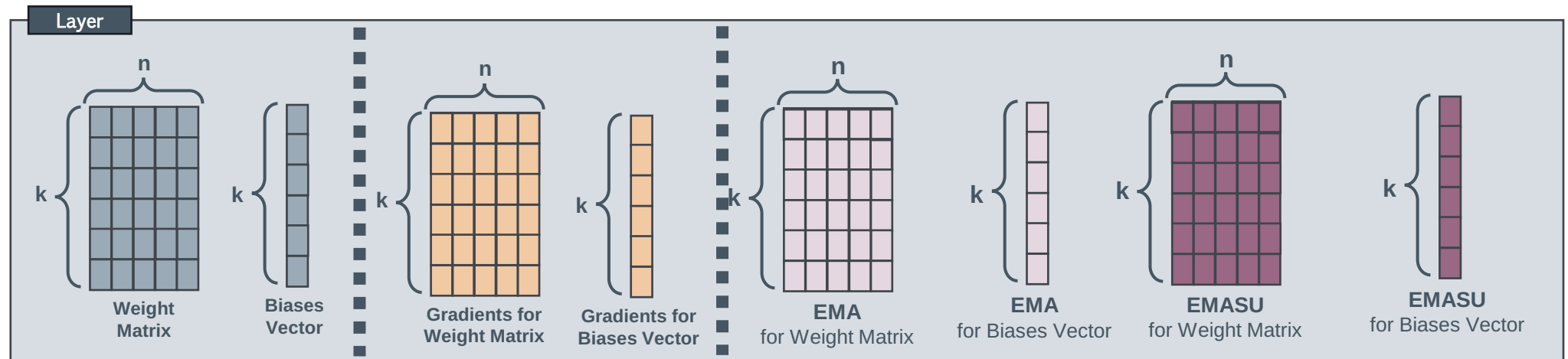
AdaDelta is an extension of AdaGrad that instead of accumulating all past squared gradients, it restricts the window of accumulated past gradients to a fixed size.

It was proposed to address the diminishing learning rates of AdaGrad, which can stop learning altogether too early in training. By maintaining the moving average of gradient information, AdaDelta continues to learn and adapt as training progresses.

A key advantage of AdaDelta is that it does not require an external learning rate. This method derives its own learning rate from the data.

AdaDelta

In case of AdaDelta we need to store for each parameter the exponential moving average (EMA) and the exponential moving average for square parameter update (EMASU).



In other words, we at **quadruple** the size of a layer by adding a two new matrixes and two new bias vector.

AdaDelta

The update formula is done in several steps:

1. Compute the Exponential Moving Average (EMA)

$$EMA_t = \rho \times EMA_{t-1} + (1 - \rho) \times (\nabla Loss(w_t))^2, \text{ where}$$

EMA_t = Exponential moving average at the moment t

EMA_{t-1} = Exponential moving average at the moment $t - 1$,

ρ = a decay rate,

$\nabla Loss(w_t)$ = gradient of the loss function with respect to the w_t

2. Compute the Root Mean Square (RMS)

$$RMS_t = \sqrt{EMA_t + \epsilon}, \text{ where}$$

ϵ = a very small value to keep $RMS_t > 0$ (usually $1e - 8$)

AdaDelta

The update formula is done in several steps:

3. Compute the Exponential Moving Average for squared updates (EMASU)

$$EMASU_t = \rho \times EMASU_{t-1} + (1 - \rho) \times (\Delta w_t)^2, \text{ where}$$

$EMASU_t$ = Exponential moving average for squared updates at the moment t
 $EMASU_{t-1}$ = Exponential moving average for squared updates at the moment $t - 1$,
 ρ = a decay rate,
 Δw_t = the delta updates for w_t

4. Compute the Root Mean Square for EMASU (RMSU)

$$RMSU_t = \sqrt{EMASU_t + \epsilon}, \text{ where}$$

ϵ = a very small value to keep $RMS_t > 0$ (usually $1e - 8$)

AdaDelta

Finally, the update formula is as follows:

$$w_{t+1} = w_t + \Delta w_t,$$

$$\Delta w_t = \frac{RMSU_{t-1}}{RMS_t} \times \nabla Loss(w_t)$$

AdaDelta

AdaDelta modifies this approach by considering only a window of recent gradients. This is done to prevent the learning rates from diminishing to the point where the model stops learning.

At the same time, instead of summing all past squared gradients, AdaDelta maintains an exponential moving average (EMA) of squared gradients. This provides a way to give more importance to recent gradients and less to older ones.

This means that parameters with larger gradients will have smaller updates, and vice versa.

AdaDelta

In PyTorch there is an optimizer defined for AdaDelta:

```
torch.optim.Adadelta(params, rho=0.9, eps=1e-08,)
```

Where:

- › **rho** corresponds to the decay parameter (ρ)
- › **eps** is that small value (ϵ) used to prevent division by 0 if the denominator reaches 0 value

And a simple example:

```
# Adadelta with default parameters  
optimizer = torch.optim.Adadelta(model.parameters())
```

Dropout

Dropout

Dropout is a regularization technique used in neural networks to prevent overfitting. It works by randomly deactivates a subset of neurons (units) within a layer with a certain probability (usually between 20% and 50%) at each training step or epoch. The "dropped-out" neurons do not contribute to the forward pass and do not participate in backpropagation. Thus, their contribution to the network's output is temporarily removed.

In practice, this effect is achieved by converting some values from the input matrix to 0 values (as a result, they will not participate to the computations).

Dropout

Let's see an example:

```
import torch

d = torch.nn.Dropout()
t = torch.randn(10)
res = d(t)
print("Tensor=", t)
print("After dropout=", res)
```

Output

```
Tensor= tensor([-0.8307,  1.8880,  0.3791, -0.1156, -1.7474,
                0.6103, -0.7784,  1.2726,  0.0262, -1.1152])
After dropout= tensor([ 0.0000,  3.7760,  0.0000, -0.2312,  0.0000,
                      1.2205, -1.5568,  0.0000,  0.0524, -2.2305])
```

The default rate for dropout is 0.5 (notice that approximatively 50% of the new values from the tensors are zeroed).

Dropout

Let's see an example:

```
import torch

d = torch.nn.Dropout(p=0.25, inplace=True)
t = torch.randn(10)
print("Tensor=", t)
d(t)
print("After dropout=", t)
```

Output

```
Tensor= tensor([-0.0321, -0.1080,  0.2825, -0.7310,  1.6960,
                -0.8184,  0.8193, -1.7210, -1.1479,  0.0763])
After dropout= tensor([-0.0428, -0.1440,  0.3766, -0.9747,  0.0000,
                      -1.0913,  1.0924, -2.2947, -1.5305,  0.0000])
```

In this case, the drop out rate is 25% and we do the change directly on the tensor (as such we don't need to create another tensor). Notice that we change the tensor "t" instead of creating another one.

Dropout

During **training**, dropout is used to randomly deactivate neurons. **At inference time** (when evaluating the model on validation or test data), dropout is not used, and the full network is utilized.

Dropout can be seen as a form of data augmentation in the latent space/features, and it is particularly helpful for types of data where we do not have as developed an intuition as with images for how we can augment the data (tabular data, text)

Advantages:

- Dropout has been shown to be highly effective at reducing overfitting in deep neural networks, leading to improved generalization and robustness.
- It is computationally inexpensive and can be easily implemented in most neural network architectures.

π

Q & A

