50.039 Theory and Practice of Deep Learning

W2-S2 Neural Networks, Initializers, Optimizers and other Good Practices

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

About this week (Week 2)

1. What are the **typical initializers for trainable parameters** in Neural Networks?
2. What is **symmetry** in a Neural Network and why is it essential to **break it** with proper initializers?
3. What is the **exploding gradient** problem?
4. How to **spot and fix an exploding gradient** problem?
5. What is the **vanishing gradient** problem?
6. How to **spot and fix a vanishing gradient** problem?
7. Why are **activation functions** needed in Neural Networks?
8. What is the **universal approximation theorem**?

About this week (Week 2)

9. What are typical **examples** of **activation functions**? Which activations should I be using?
10. What are typical examples of **advanced optimizers**? (Adagrad, RMSProp, Adam, etc.).
11. How to **implement** said **optimizers** manually?
12. What is a **stochastic gradient descent** and what are its **benefits**?
13. How to implement the **stochastic gradient descent** manually?
14. What is a **mini-batch gradient descent** and what are its **benefits**?
15. How to implement the **mini-batch gradient descent** manually?
16. How to **combine all optimizers concepts** into a great optimizer?

About this week (Week 2)

17. What is the **no free lunch theorem**?
18. What is the **train-test-validation split**?
    Why is it **good practice** to use an extra validation set?
19. What is the **early stopping** of optimizer concept?
    Why is it **good practice** to use it?
20. What are **saver** and **loader** functions?
    Why is it **good practice** to use them?
21. What are **other common good practices** when it comes to Neural Networks?
22. How to decide on **an appropriate number of layers and neurons**?

The problem with the GD algorithm

Let us consider the following function f:

$$f(x) = \frac{319}{8400}\,x^4 + \frac{43}{4200}x^3 - \frac{6799}{8400}\,x^2 - \frac{299}{840}\,x + 6$$

And let us assume we are trying to solve the optimization problem:

$$x^* = \arg\min_x f(x)$$

The problem with the GD algorithm

The derivative of f is:

$$f'(x) = \frac{1276}{8400}\,x^3 + \frac{129}{4200}\,x^2 - \frac{13598}{8400}\,x - \frac{299}{840}$$

Following from this result, we could use gradient descent to solve the previous optimization by using the following update rule:

$$x \leftarrow x - \alpha f'(x)$$

The problem with the GD algorithm

Our **vanilla gradient descent algorithm** is implemented by manually defining the objective function f and its derivative f'.

```python
def obj_fun(val):
    a4 = 319/8400
    a3 = 43/4200
    a2 = -6799/8400
    a1 = -299/840
    a0 = 6
    return a4*val**4 + a3*val**3 + a2*val**2 + a1*val + a0
```

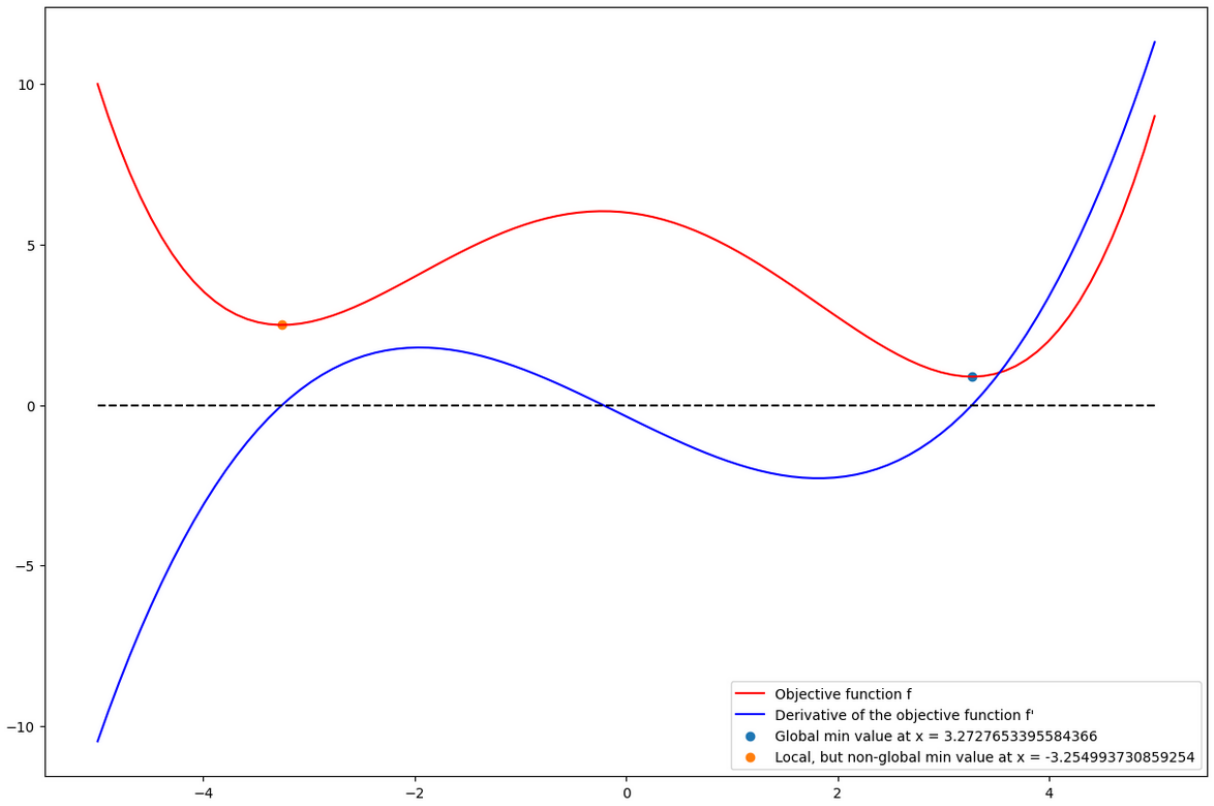
```python
def obj_fun_deriv(val):
    a4 = 319/8400
    a3 = 43/4200
    a2 = -6799/8400
    a1 = -299/840
    return 4*a4*val**3 + 3*a3*val**2 + 2*a2*val + a1
```

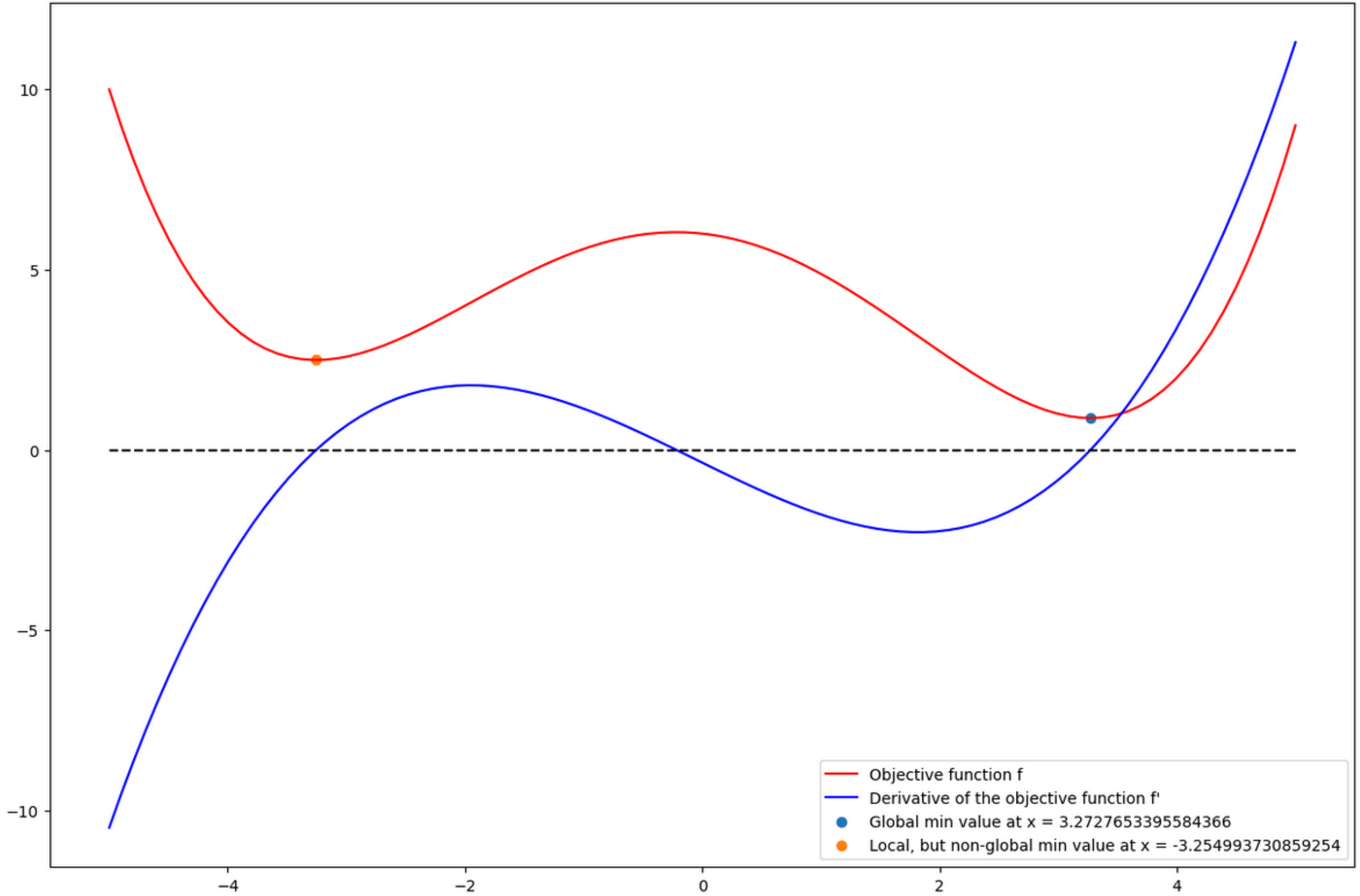
We can then implement our "**vanilla" gradient descent algorithm** for this optimization problem as shown below.

```python
def vanilla_gd(start_val, alpha = 0.1, n_iter = 100):
    val = start_val
    list_vals = [val]
    for iter_num in range(n_iter):
        # Compute gradient and update value
        val -= alpha*obj_fun_deriv(val)
        list_vals.append(val)
    return val, list_vals
```

The problem with the GD algorithm

The function f admits a **global minimum** at $x \approx 3.27$ and a **local (non-global) minimum** at $x \approx -3.25$, as shown in the figure below.




10
5
0
−5
−10
−4
−2
0
2
4
Objective function f
Derivative of the objective function f'
Global min value at x = 3.2727653395584366
Local, but non-global min value at x = -3.254993730859254

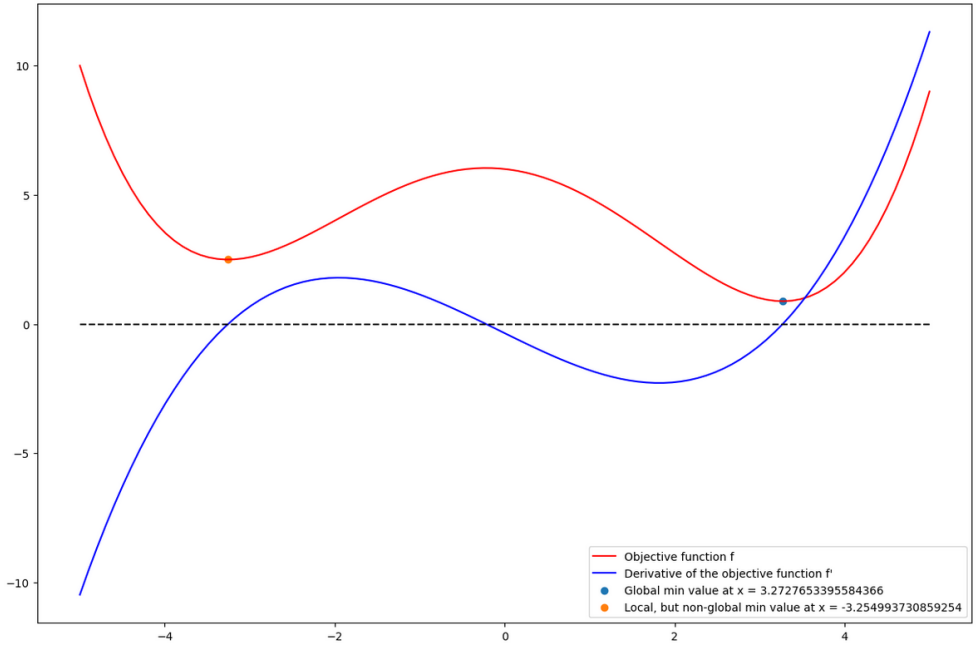
The problem with the GD algorithm

When using $x_0 = 5$ as a **starting point** for our **gradient descent algorithm**, we end up converging to the global minimum.

(Success!)

```
opt_val_vanilla_gd1, val_list_vanilla_gd1 = vanilla_gd(start_val = 5, \
                                                       alpha = 0.1, \
                                                       n_iter = 100)
print("Optimal, found by vanilla gd: ", opt_val_vanilla_gd1)
print("Global min: ", approx_min_x)
print("Local, non-global min: ", approx_min_x2)
```

```
Optimal, found by vanilla gd:  3.2727653395584366
Global min:  3.2727653395584366
Local, non-global min:  -3.254993730859254
```



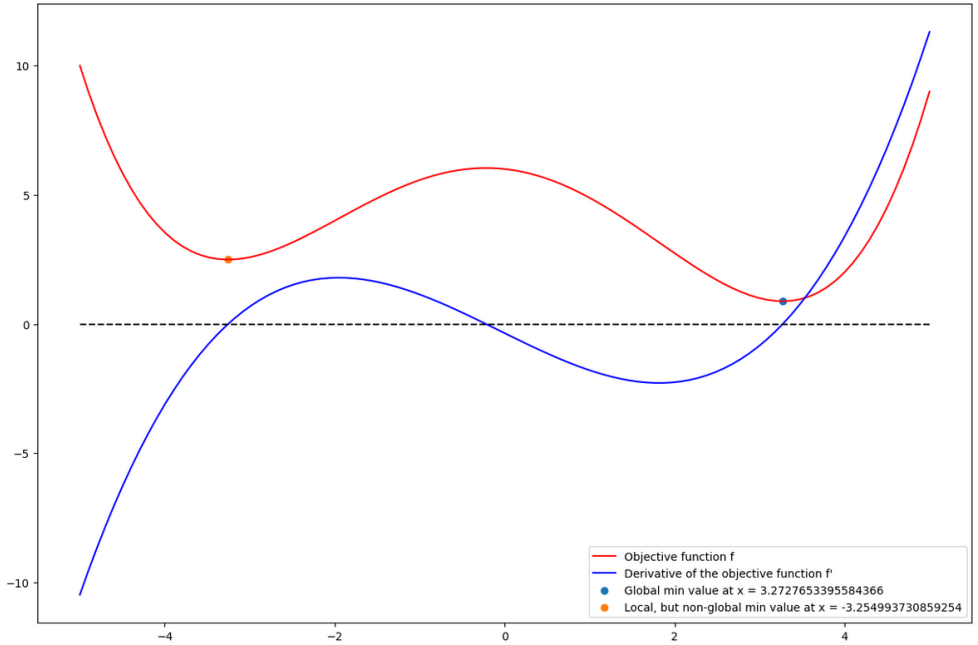
The problem with the GD algorithm

When using $x_0 = -5$ as a **starting point** for our **gradient descent**, however, we end up converging to the local (non-global) minimum.

Observation: Unfortunately, in the presence of local non-global minima, the GD algorithm might end up getting trapped and converging to the wrong minimum.

```
opt_val_vanilla_gd2, val_list_vanilla_gd2 = vanilla_gd(start_val = -5, \
                                                       alpha = 0.1, \
                                                       n_iter = 100)
print("Optimal, found by vanilla gd: ", opt_val_vanilla_gd2)
print("Global min: ", approx_min_x)
print("Local, non-global min: ", approx_min_x2)
```

```
Optimal, found by vanilla gd:  -3.254993730859254
Global min:  3.2727653395584366
Local, non-global min:  -3.254993730859254
```



The problem with the GD algorithm

When using $x_0 = -5$ as a **starting point** for our **gradient descent**, however, we end up converging to the local (non-global) minimum.

Observation: Unfortunately, in the presence of local non-global minima, the GD algorithm might end up getting trapped and converging to the wrong minimum.

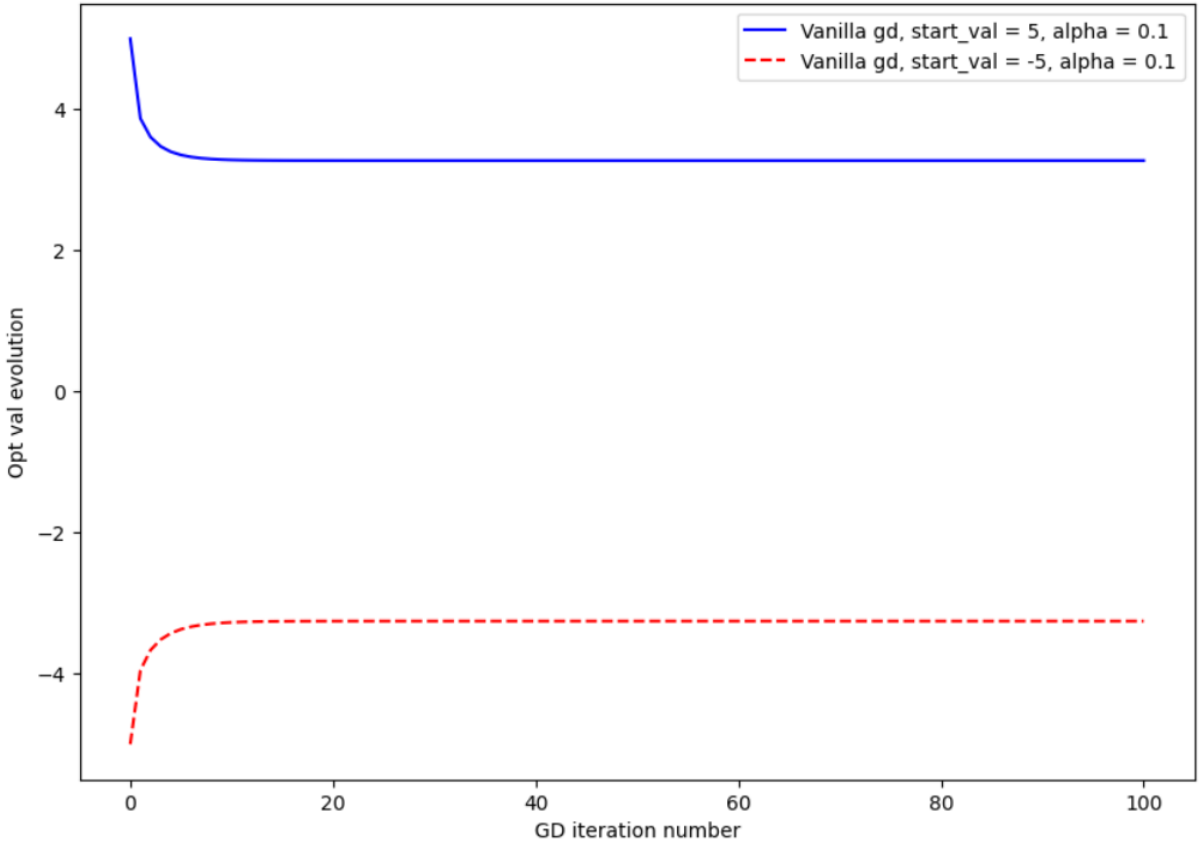
In the case of our Neural Networks:

- This could lead to choosing **suboptimal parameters W and b**, which are not going to minimize the loss function used for training,
- And therefore, lead to **a Neural Network that has not been optimally trained for its task**.

The problem with the GD algorithm

Remember

- **When the function is convex:** GD is guaranteed to converge to the global minimum, no matter the starting point.
- **When the function is not convex (and our function f is often not):** the starting point has an impact and might lead to a completely different outcome.
- Solutions exist but are not guaranteed to work.



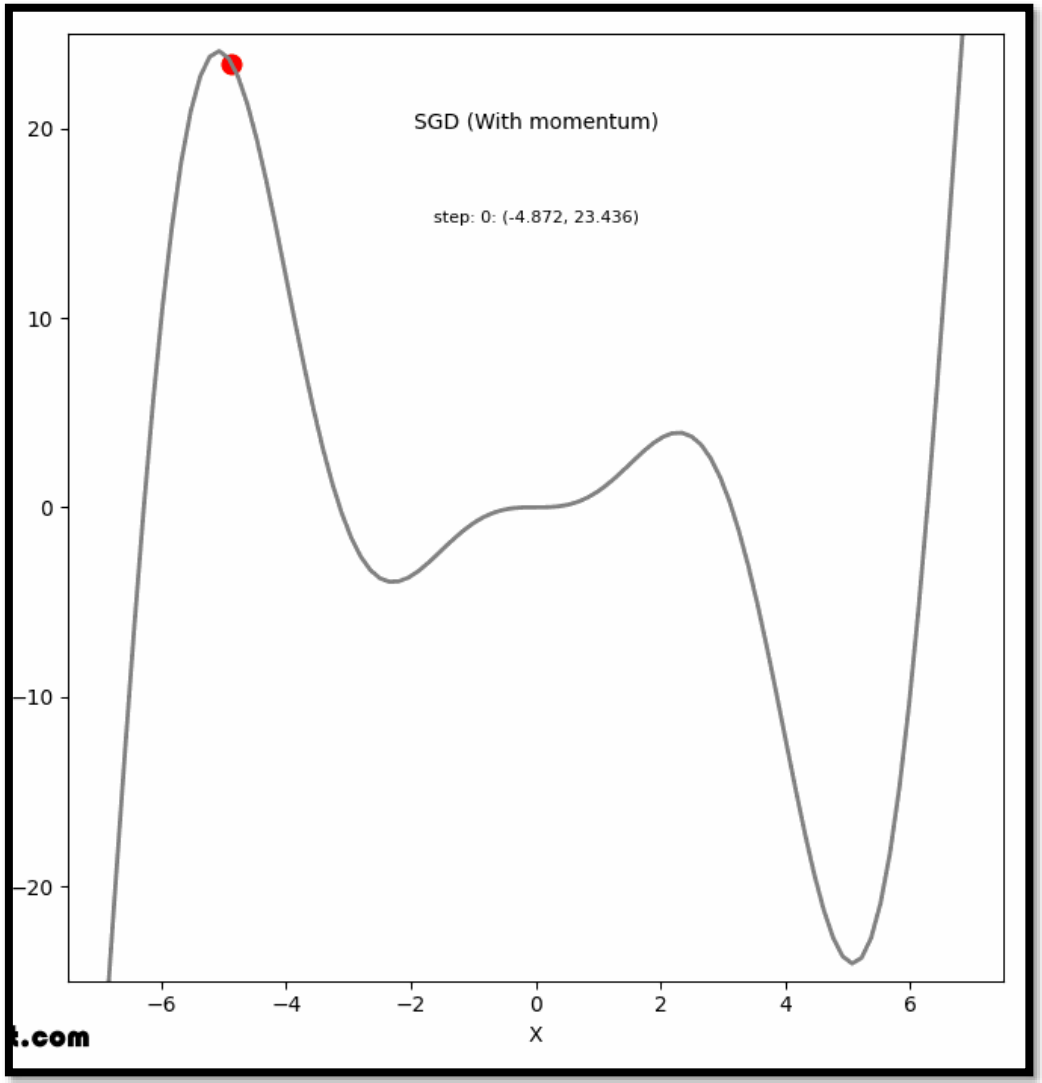
Adding momentum to gradient descent

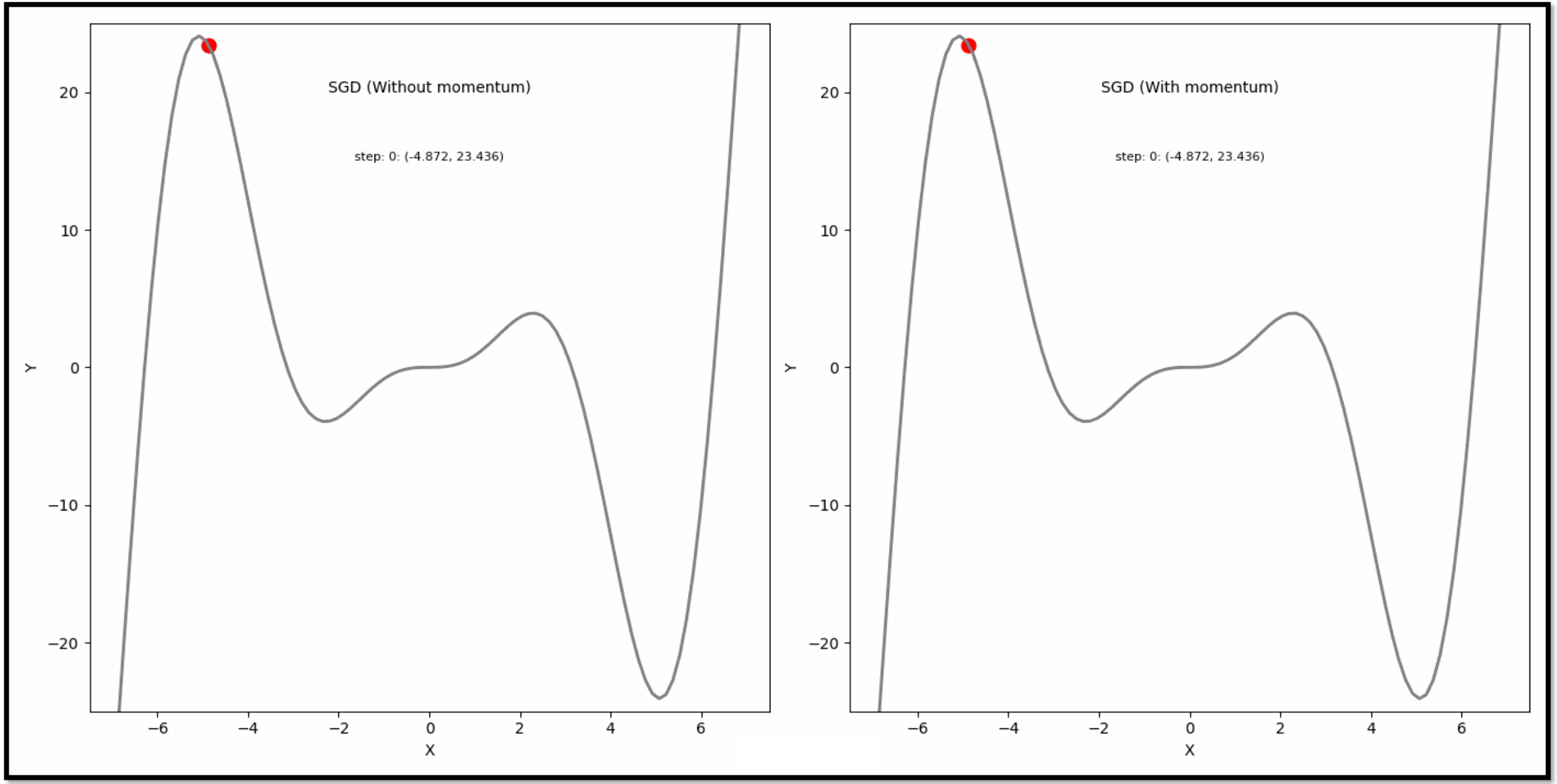
Definition (Gradient Descent with Added Momentum):

To alleviate this phenomenon, we propose to use **momentum**.

It is a method that helps accelerate the training process by updating the weights of the model by reusing values from the past gradients.

The idea is that, if a direction in the parameters space consistently leads to good reduction in the loss function, then that direction should be reinforced.




SGD (Without momentum)
step: 0: (-4.872, 23.436)
SGD (With momentum)
step: 0: (-4.872, 23.436)
Y
X
20
10
0
−10
−20
−6
−4
−2
0
2
4
6

Adding momentum to gradient descent

Momentum can be implemented by adding a term $g(f'_{prev})$ to the parameter update, for instance:

$$x \leftarrow x - \alpha f'(x) + \mu g(f'_{prev})$$

As before,

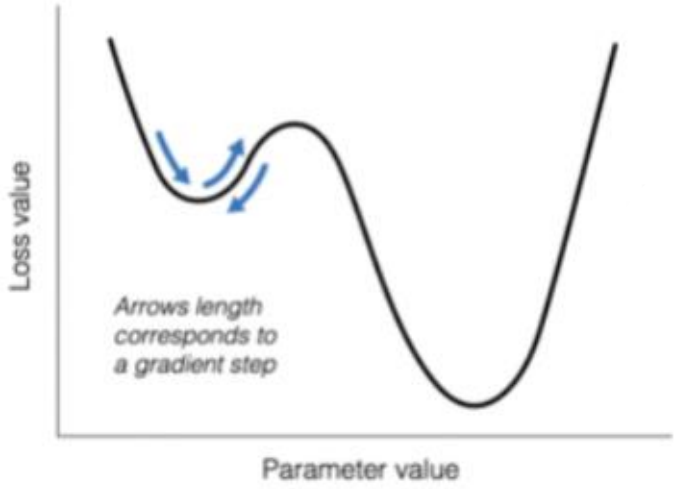
- α is a hyperparameter that controls the step size/learning rate at each iteration,
- And **the coefficient $\boldsymbol{\mu}$ is another hyperparameter that controls the influence of the momentum** $g(f'_{prev})$, which uses the history of previous gradient values f'_{prev}.

Adding momentum to gradient descent

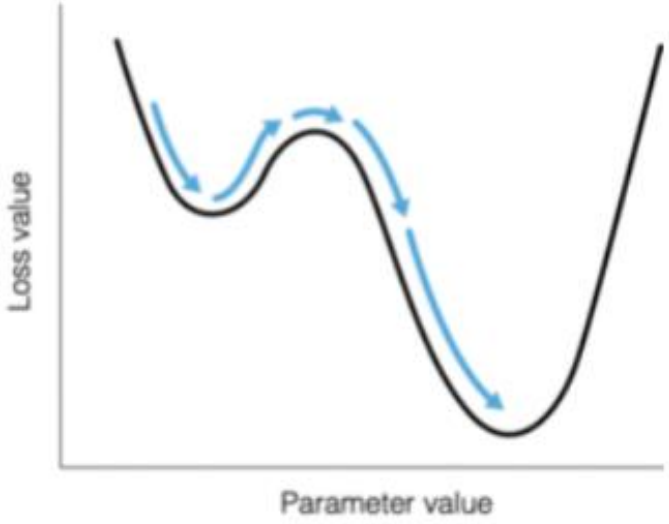
Using momentum can help the model converge faster and **escape local minima** and converge to global minima instead.

However, it can also cause the model to overshoot the global minimum and diverge.

So, just like with the learning rate, **it is important to tune the momentum hyperparameter μ carefully (ever heard of the NFL theorem?)**.



Insufficient Momentum



Sufficient Momentum

Adding momentum to gradient descent

Definition (Gradient Descent with Added Momentum):

Momentum is a method that helps accelerate the training process by updating the weights of the model by reusing values from the past gradients.

Momentum can be implemented in many ways, by incorporating the previous values of past gradients in some way. We show a possible implementation here.

```python
def momentum_gd(start_val, alpha = 0.1, mu = 0.1, n_iter = 100):
    val = start_val
    mom = 0
    list_vals = [val]
    for iter_num in range(n_iter):
        # Compute gradient
        grad = obj_fun_deriv(val)
        # Update momentum
        mom = mu*mom + alpha*grad
        # Update value
        val -= mom
        list_vals.append(val)

    return val, list_vals
```

Another possible implementation (Nesterov Momentum GD) is shown in bonus slides.

Adding momentum to gradient descent

Case #1 (good scenario): As before, if the starting point is good and not too much momentum is used, we will converge to the correct minimum.

```
opt_val_momentum_gd1, val_list_momentum_gd1 = momentum_gd(start_val = 5, \
                                                          alpha = 0.1, \
                                                          mu = 0.1, \
                                                          n_iter = 100)
print("Optimal, found by momentum gd: ", opt_val_momentum_gd1)
print("Global min: ", approx_min_x)
print("Local, non-global min: ", approx_min_x2)
```

```
Optimal, found by momentum gd:  3.272765339558436
Global min:  3.2727653395584366
Local, non-global min:  -3.254993730859254
```

Adding momentum to gradient descent

Case #2 (bad scenario): As before, if the starting point is bad and not too much momentum is used, we will end up converging to the incorrect minimum, like before.

```
opt_val_momentum_gd2, val_list_momentum_gd2 = momentum_gd(start_val = -5, \
                                                          alpha = 0.1, \
                                                          mu = 0.1, \
                                                          n_iter = 100)
print("Optimal, found by momentum gd: ", opt_val_momentum_gd2)
print("Global min: ", approx_min_x)
print("Local, non-global min: ", approx_min_x2)
```

```
Optimal, found by momentum gd:  -3.254993730859254
Global min:  3.2727653395584366
Local, non-global min:  -3.254993730859254
```

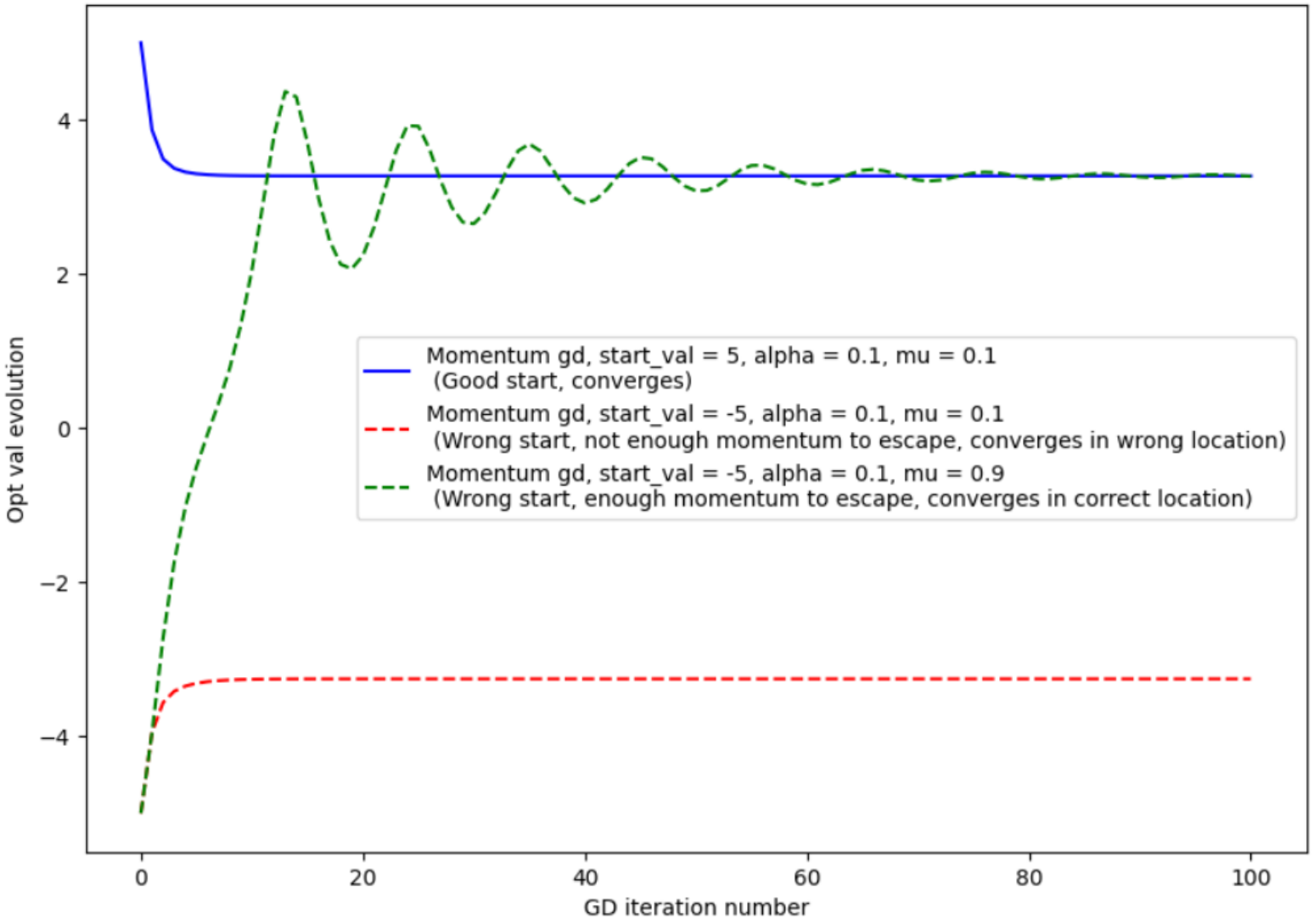
Adding momentum to gradient descent

Case #3 (bad scenario, momentum saves the day): However, if the starting point is bad and enough momentum is used, we might be able to escape the local minimum and converge to the correct minimum.

```
opt_val_momentum_gd3, val_list_momentum_gd3 = momentum_gd(start_val = -5, \
                                                          alpha = 0.1, \
                                                          mu = 0.9, \
                                                          n_iter = 100)
print("Optimal, found by momentum gd: ", opt_val_momentum_gd3)
print("Global min: ", approx_min_x)
print("Local, non-global min: ", approx_min_x2)
```

```
Optimal, found by momentum gd:  3.267681061691328
Global min:  3.2727653395584366
Local, non-global min:  -3.254993730859254
```

Adding momentum to gradient descent



Adding a decay to hyperparameters

Definition (Learning rate decay):

Learning rate decay is a technique used to **gradually reduce the learning rate** of gradient descent over iterations.

The idea is to **start with a relatively high learning rate, which can help the model to converge quickly in the early stages of training**, and then **gradually reduce the learning rate as training progresses**.

This can help the model to continue learning and improving even as it gets closer to a good solution. It can also help to address overfitting by preventing the model from making overly large updates to its parameters.

There are **several ways to implement learning rate decay**, for instance by reducing the learning rate by a fixed amount at regular intervals, updating the learning rate as $\alpha \leftarrow 0.99\alpha$, every N iterations.

Adding a decay to hyperparameters

A possible implementation of gradient descent with learning rate decay is shown below.

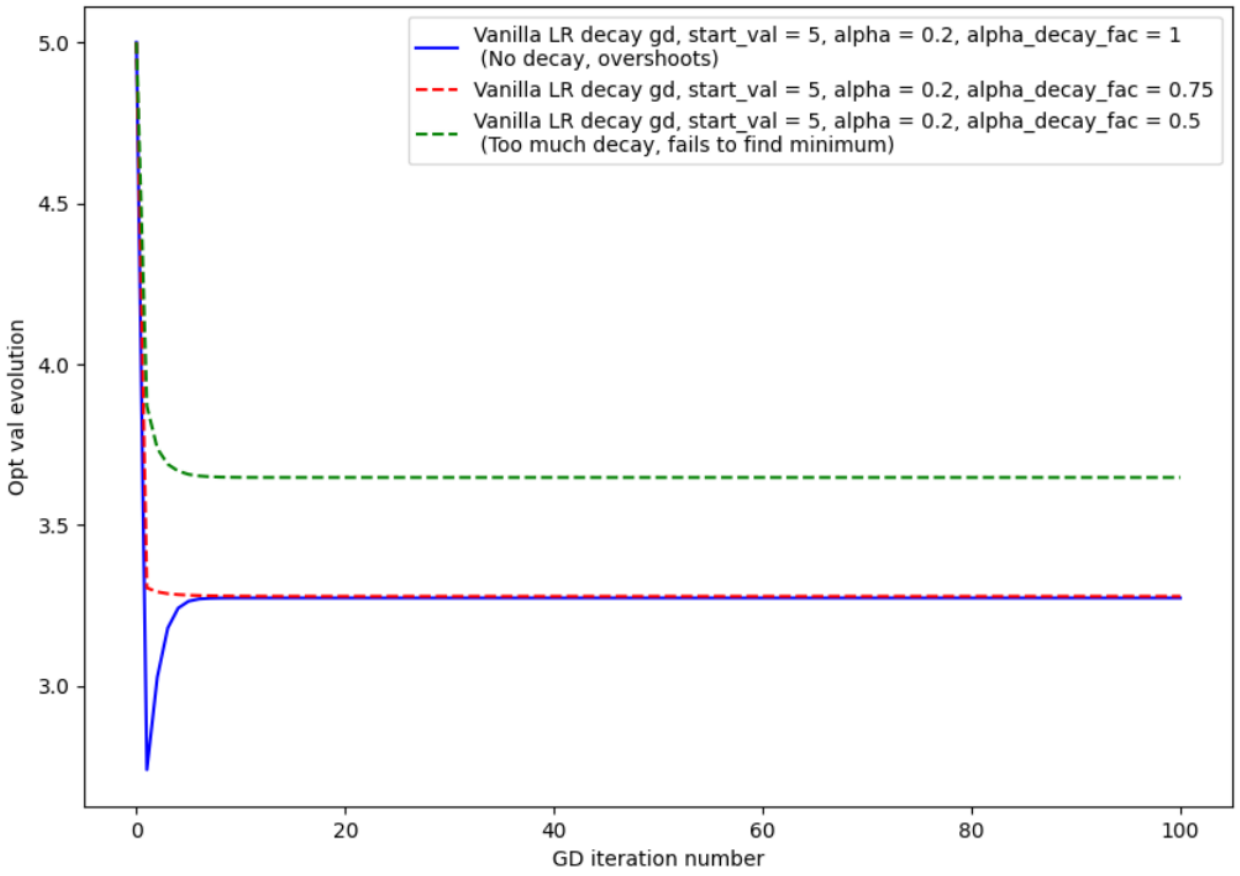
Notice how, on every iteration, the learning rate is updated and decayed as $\alpha \leftarrow K\alpha$, with K a decay factor between 0 and 1.

```python
def vanilla_gd_lrdecay(start_val, alpha = 0.1, alpha_decay_fac = 0.99, n_iter = 100):
    val = start_val
    list_vals = [val]
    for iter_num in range(n_iter):
        # Decay on LR
        alpha *= alpha_decay_fac
        # Update value
        val += -alpha*obj_fun_deriv(val)
        list_vals.append(val)
    return val, list_vals
```

Adding a decay to hyperparameters

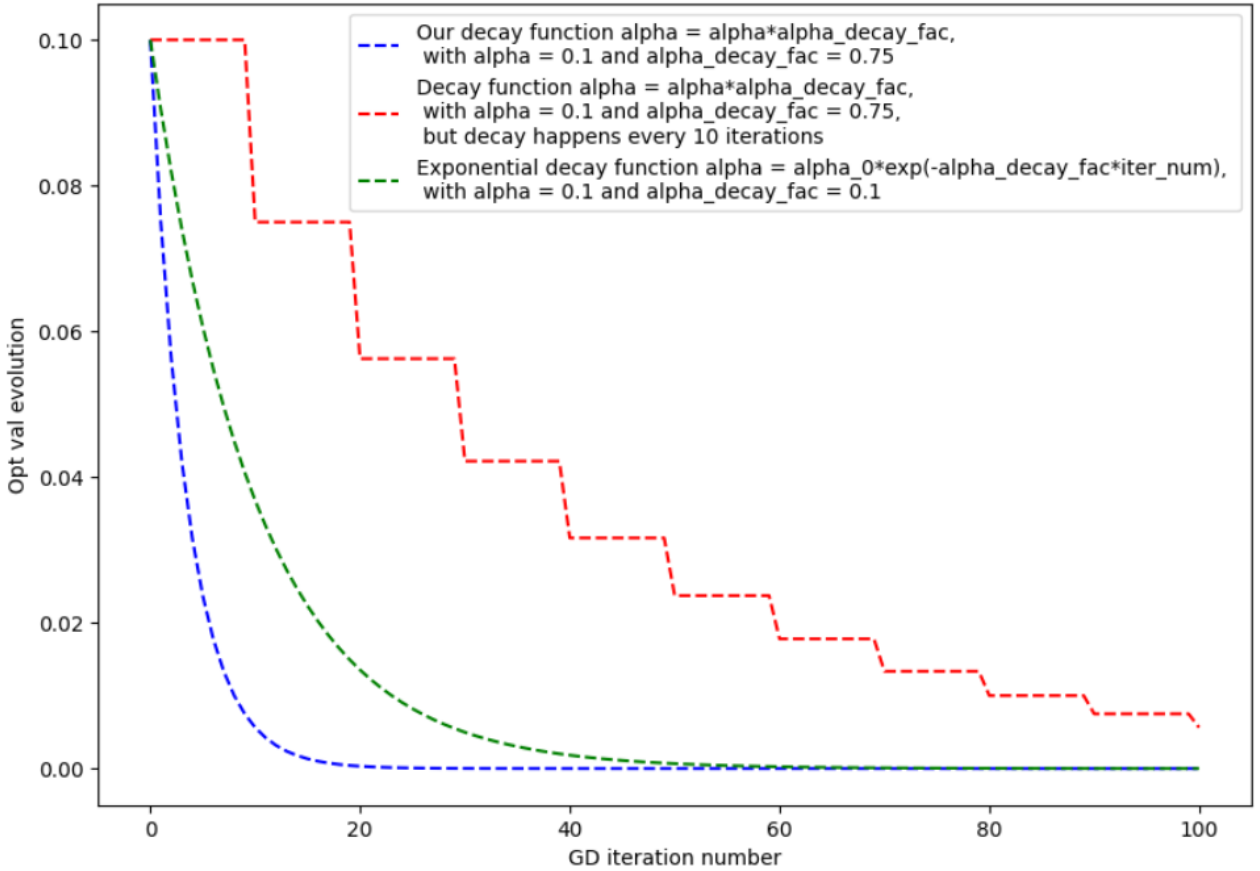
As before, the learning rate decay parameter K is a hyperparameter that requires to be adjusted carefully:

- a high decay might help not overshoot a minimum,
- but decaying too fast might make α go to small values too soon, leading to a wrong convergence point.



Adding a decay to hyperparameters

Different decay methods can be implemented, with different effects on the gradient descent algorithm: as before (NFL!), worth exploring.



Gradient-based learning rate control

Definition (Gradient-based learning rate control):

Other options for **controlling the learning rate** suggest to **use the value of the gradients to adjust the learning rate** accordingly.

Gradients are often used in learning rate formulas because they provide a measure of how much the parameters are changing with each update.

The magnitude of the gradients give us an indication of how much "learning" is done at each step. It can be used to adjust the learning rate accordingly.

In Notebook 5, we implement a possible learning rate control mechanism, using the gradients and the learning rate factor α_{cf}, below.

$$\alpha = \frac{1}{(1 + \alpha_{cf}\sqrt{|f'(x)|})}$$

Idea behind gradient-based LR control

Core idea: Gradient-based learning rate control is typically used to make the learning rate inversely proportional to the mean gradient value (or its mean squared value).

By doing so,

- **if the gradients are small** (i.e. the model is making small updates to its weights), the resulting learning rate will be high to prevent vanishing gradients.
- and **if the gradients are large** (i.e. the model is making large updates to its weights), the resulting learning rate will be low to prevent exploding gradients.

Gradient-based learning rate control

A possible gradient-based learning control scheme:

$$\alpha = \frac{1}{(1 + \alpha_{cf}\sqrt{|f'(x)|})}$$

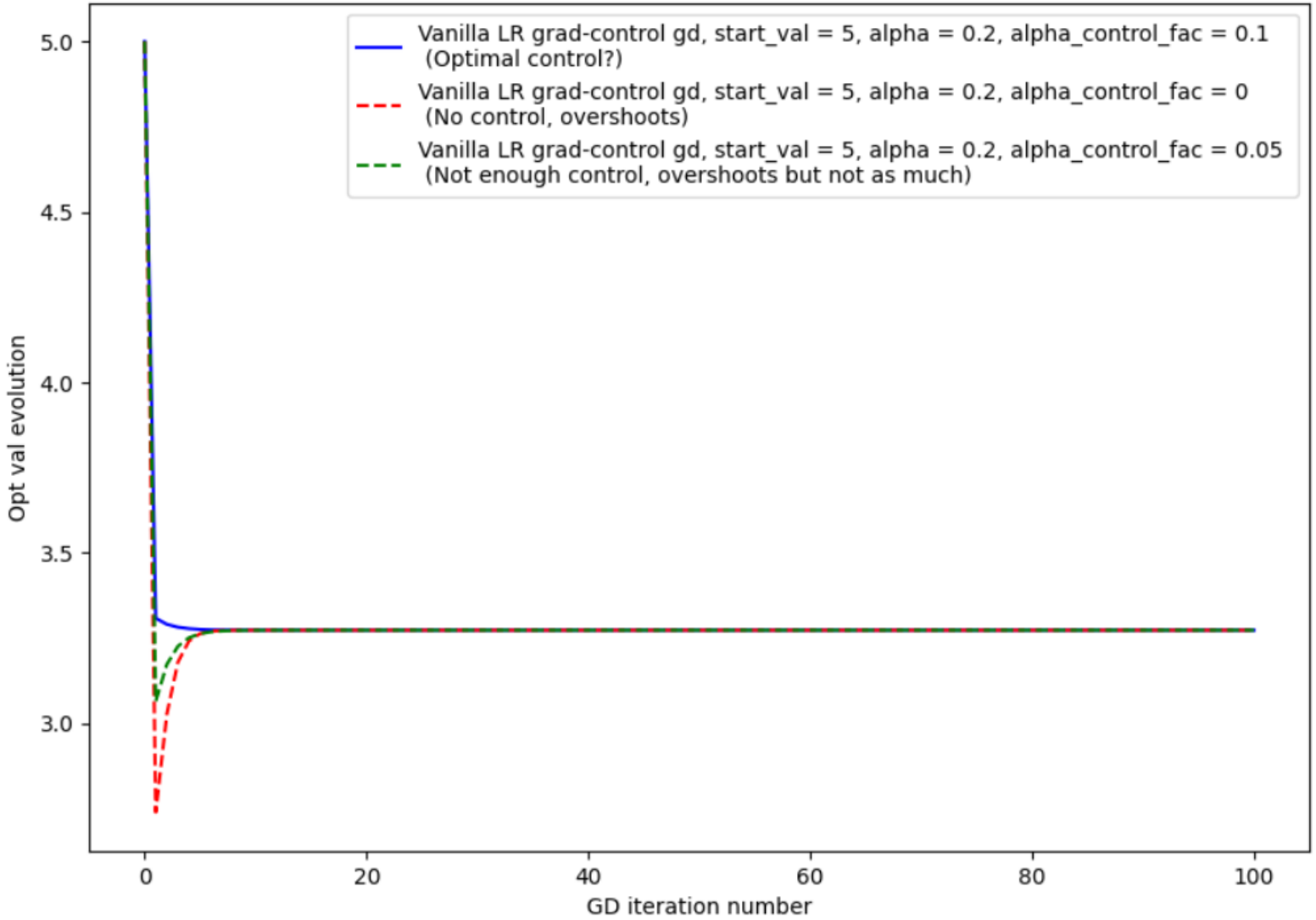
```
def vanilla_gd_gradlr(start_val, alpha = 0.1, alpha_control_fac = 0.99, n_iter = 100):
    val = start_val
    list_vals = [val]
    for iter_num in range(n_iter):
        # Compute gradients
        grad = obj_fun_deriv(val)
        # Gradient-based decay on LR
        alpha = 1/(1 + alpha_control_fac*np.sqrt(np.abs(grad)))
        # Update value
        val += -alpha*grad
        list_vals.append(val)
    return val, list_vals
```

Gradient-based learning rate control

Open question: What would change if I decided to replace the = sign in line 8 with a *= instead, as shown below?

```
def vanilla_gd_gradlr_v2(start_val, alpha = 0.1, alpha_control_fac = 0.99, n_iter = 100):
    val = start_val
    list_vals = [val]
    for iter_num in range(n_iter):
        # Compute gradients
        grad = obj_fun_deriv(val)
        # Gradient-based decay on LR
        alpha *= 1/(1 + alpha_control_fac*np.sqrt(np.abs(grad)))
        # Update value
        val += -alpha*grad
        list_vals.append(val)
    return val, list_vals
```

Gradient-based learning rate control



Gradient-based learning rate control

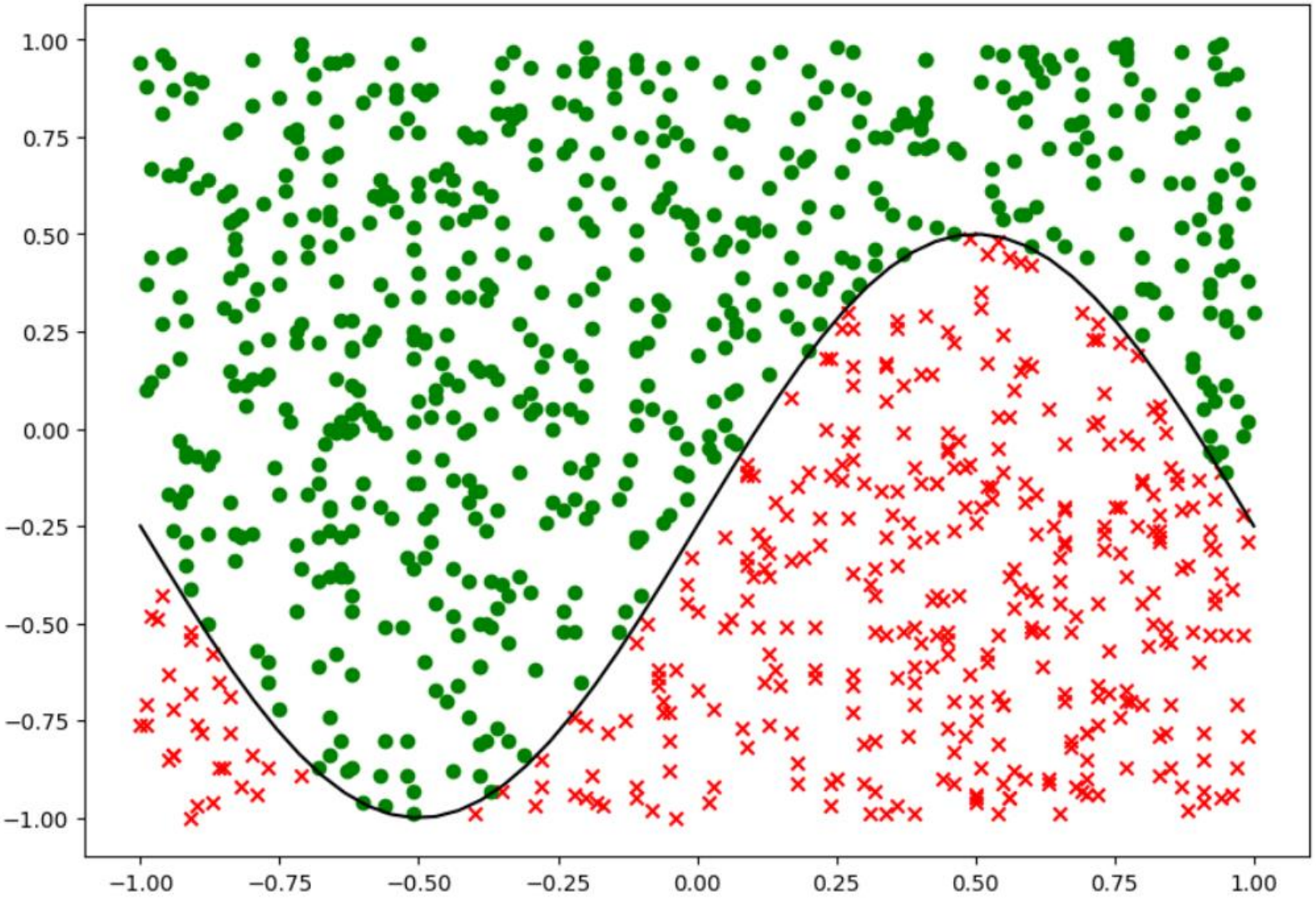
This can also help to prevent the model from making overly large updates, which can – in turn – help to improve generalization by preventing overfitting.

Most advanced gradient descent algorithms will typically combine

1. **A gradient-based learning rate (LR) control,**
2. **A learning rate (LR) decay,**
3. **And some momentum formula**.

Adjustments on the hyperparameters values for each component will be left to the human behind the keyboard!

Going back to our classification dataset



So far, training using “vanilla” gradient descent

```
    def backward(self, inputs, outputs, alpha = 1e-5):
        # Get the number of samples in dataset
        m = inputs.shape[0]

        # Forward propagate
        Z1 = np.matmul(inputs, self.W1)
        Z1_b = Z1 + self.b1
        A1 = self.sigmoid(Z1_b)
        Z2 = np.matmul(A1, self.W2)
        Z2_b = Z2 + self.b2
        A2 = self.sigmoid(Z2_b)

        # Compute error term
        dL_dA2 = -outputs/A2 + (1 - outputs)/(1 - A2)
        dL_dZ2 = dL_dA2*A2*(1 - A2)
        dL_dA1 = np.dot(dL_dZ2, self.W2.T)
        dL_dZ1 = dL_dA1*A1*(1 - A1)

        # Gradient descent update rules
        self.W2 -= (1/m)*alpha*np.dot(A1.T, dL_dZ2)
        self.W1 -= (1/m)*alpha*np.dot(inputs.T, dL_dZ1)
        self.b2 -= (1/m)*alpha*np.sum(dL_dZ2, axis = 0, keepdims = True)
        self.b1 -= (1/m)*alpha*np.sum(dL_dZ1, axis = 0, keepdims = True)
```

AdaGrad

Definition (AdaGrad optimizer [AdaGrad2011]):

AdaGrad is variation of gradient descent, which **adapts the learning rate for each parameter separately**, based on the historical gradient information for that parameter.

AdaGrad scales down the learning rate for parameters that have received a large number of updates, which can help prevent overfitting and improve generalization.

The gradient descent rule relies on variables G_W and G_b. Their values are defined as cumulated sum of the squared values of the gradients with respect to each parameter.

AdaGrad

For instance, G_{W_2} starts at 0, and then, on each iteration:

$$G_{W_2} = G_{W_2} + \left(\frac{\partial L}{\partial W_2}\right)^2$$

This is then used to adjust the learning rate α_{W_2} for the W_2 update rule:

$$\alpha_{W_2} = \frac{\alpha}{\sqrt{G_{W_2} + \epsilon}}$$

Here, the value ϵ is set to $1e^{-6}$ to prevent divisions by zero.

AdaGrad

The update rule for W_2 is then:

$$W_2 \leftarrow W_2 - \alpha_{W_2} \frac{\partial L}{\partial W_2}$$

The value of G_{W_2} therefore grows over time and it grows faster, if the present and the history past gradients we calculated are large.

Therefore, AdaGrad can be seen as a GD, which implements **both a LR decay** and **a LR gradient-based control**.

AdaGrad implementation

Keep history of gradient in a list called G_list, which will be updated on each iteration of backward
(Note that it could have also been an attribute for class)

Compute gradients and G_X Coefficients for each trainable parameter X

Update rule uses adjusted learning rate for each parameter and epsilon value.

```
def backward(self, inputs, outputs, alpha = 1e-5):
    # Get the number of samples in dataset
    m = inputs.shape[0]

    # Forward propagate
    Z1 = np.matmul(inputs, self.W1)
    Z1_b = Z1 + self.b1
    A1 = self.sigmoid(Z1_b)
    Z2 = np.matmul(A1, self.W2)
    Z2_b = Z2 + self.b2
    A2 = self.sigmoid(Z2_b)

    # Compute error term
    dL_dA2 = -outputs/A2 + (1 - outputs)/(1 - A2)
    dL_dZ2 = dL_dA2*A2*(1 - A2)
    dL_dA1 = np.dot(dL_dZ2, self.W2.T)
    dL_dZ1 = dL_dA1*A1*(1 - A1)

    # Gradient descent update rules
    grad_W2 = (-1/m)*np.dot(A1.T, dL_dZ2)
    grad_W1 = (-1/m)*np.dot(inputs.T, dL_dZ1)
    grad_b2 = (-1/m)*np.sum(dL_dZ2, axis = 0, keepdims = True)
    grad_b1 = (-1/m)*np.sum(dL_dZ1, axis = 0, keepdims = True)

    # Momentum and gradient decay/normalization for each parameter
    G_W2, G_W1, G_b2, G_b1 = self.G_list
    G_W2 += grad_W2**2
    G_W1 += grad_W1**2
    G_b2 += grad_b2**2
    G_b1 += grad_b1**2
    self.G_list = [G_W2, G_W1, G_b2, G_b1]

    # Gradient descent update rules
    eps = 1e-6
    self.W2 += alpha*grad_W2/(np.sqrt(G_W2 + eps))
    self.W1 += alpha*grad_W1/(np.sqrt(G_W1 + eps))
    self.b2 += alpha*grad_b2/(np.sqrt(G_b2 + eps))
    self.b1 += alpha*grad_b1/(np.sqrt(G_b1 + eps))

    # Update loss
    self.CE_loss(inputs, outputs)
```

RMSProp

Definition (RMSProp optimizer [RMSProp2012]):

RMSProp is a variation of AdaGrad, but it uses a **running average of the squared gradients to scale the learning rate** for each parameter, rather than using the sum of the squared gradients as in AdaGrad.

RMSProp can be less sensitive to the learning rate than AdaGrad, and it is often used in conjunction with other techniques such as momentum.

We will typically keep this idea in mind for later when designing the **Adam** optimizer.

RMSProp

Problem: The value G_{W_2} is no longer defined as the sum of all the squared values of the gradients with respect to each parameter.

This AdaGrad formula could prove problematic in practice as it would make the value of G_{W_2} grow forever.

It would sometimes end up exploding to infinity and would, in turn, make the LR decay too much, too fast.

RMSProp

Solution: RMSProp suggests, instead to use a mixed formula (known as **running average**), reusing the previous values of these parameters, mixed with the current squared values of the gradients, shown below.

$$G_{W_2} = (1 - \rho)\left(\frac{\partial L}{\partial W_2}\right)^2 + \rho\, G_{W_2}$$

The coefficient ρ is a hyperparameter, taking values between 0 and 1 and deciding the mixture to use in the running average.

RMSProp

As before, this G_{W_2} coefficient is then used to adjust the learning rate α_{W_2} for the W_2 update rule:

$$\alpha_{W_2} = \frac{\alpha}{\sqrt{G_{W_2} + \epsilon}}$$

The update rule for W_2 is then:

$$W_2 \leftarrow W_2 - \alpha_{W_2} \frac{\partial L}{\partial W_2}$$

As in AdaGrad, RMSProp can be seen as GD with a mixture of **LR decay** and **LR gradient-based control**.

RMSProp implementation

```python
def backward(self, inputs, outputs, alpha = 1e-5, rho = 0.1):
    # Get the number of samples in dataset
    m = inputs.shape[0]

    # Forward propagate
    Z1 = np.matmul(inputs, self.W1)
    Z1_b = Z1 + self.b1
    A1 = self.sigmoid(Z1_b)
    Z2 = np.matmul(A1, self.W2)
    Z2_b = Z2 + self.b2
    A2 = self.sigmoid(Z2_b)

    # Compute error term
    dL_dA2 = -outputs/A2 + (1 - outputs)/(1 - A2)
    dL_dZ2 = dL_dA2*A2*(1 - A2)
    dL_dA1 = np.dot(dL_dZ2, self.W2.T)
    dL_dZ1 = dL_dA1*A1*(1 - A1)

    # Compute gradients
    grad_W2 = (-1/m)*np.dot(A1.T, dL_dZ2)
    grad_W1 = (-1/m)*np.dot(inputs.T, dL_dZ1)
    grad_b2 = (-1/m)*np.sum(dL_dZ2, axis = 0, keepdims = True)
    grad_b1 = (-1/m)*np.sum(dL_dZ1, axis = 0, keepdims = True)

    # Momentum and gradient decay/normalization for each parameter
    G_W2, G_W1, G_b2, G_b1 = self.G_list
    G_W2 = rho*G_W2 + (1 - rho)*grad_W2**2
    G_W1 = rho*G_W1 + (1 - rho)*grad_W1**2
    G_b2 = rho*G_b2 + (1 - rho)*grad_b2**2
    G_b1 = rho*G_b1 + (1 - rho)*grad_b1**2
    self.G_list = [G_W2, G_W1, G_b2, G_b1]

    # Gradient descent update rules
    eps = 1e-6
    self.W2 += alpha*grad_W2/(np.sqrt(G_W2 + eps))
    self.W1 += alpha*grad_W1/(np.sqrt(G_W1 + eps))
    self.b2 += alpha*grad_b2/(np.sqrt(G_b2 + eps))
    self.b1 += alpha*grad_b1/(np.sqrt(G_b1 + eps))

    # Update loss
    self.CE_loss(inputs, outputs)
```

The only change compared to AdaGrad is in this formula and the addition of another hyperparameter ρ

Adam

Definition (Adam optimizer [Adam2015]):

Adam is a very popular variation of gradient descent that combines the ideas of momentum and RMSProp.

Adam is a widely used (possibly the most used?) optimizer, as **it can often achieve good performance with relatively little hyperparameter tuning.**

Adam

Definition (Adam optimizer [Adam2015]):

Adam is a very popular variation of gradient descent that combines the ideas of momentum and RMSProp.

Adam is a widely used (possibly the most used?) optimizer, as **it can often achieve good performance with relatively little hyperparameter tuning.**

Adam combines two things:

- **exponentially decaying averages of the past gradients** to scale the learning rate for each parameter,
- and **exponentially decaying average of the past squared gradients** to scale the learning rate for each parameter (as in RMSProp).

Adam

As before, in AdaGrad and RMSProp, **Adam** uses the same intuition on using running averages for gradients, but will do so by using two running averages instead of one.

For instance, for the trainable parameter W_2, we will have two variables, denoted V_{W_2} and S_{W_2}.

The V and S variables will be calculated for each trainable parameter.

They will later be used in the gradient descent formula.

Adam

The V and S values are defined following the logic in RMSProp, reusing the running average formula on both **gradients** and **squared gradients**:

$$V_{W_2} = (1 - \beta_1)\frac{\partial L}{\partial W_2} + \beta_1 V_{W_2}$$

$$S_{W_2} = (1 - \beta_2)\left(\frac{\partial L}{\partial W_2}\right)^2 + \beta_2 S_{W_2}$$

Adam

In Adam, we will have two hyperparameters, one for each component we described earlier:

- First, β_1 is an hyperparameter, whose value is set between 0 and 1, often set to 0.9.
- Second, β_2 is an hyperparameter, whose value is set between 0 and 1, often set to 0.999.

In general, default values for these parameters seem to perform well, but as in all things, NFL!

Adam

The values are then used to calculate the change C_{W_2} below:

$$C_{W_2} = \alpha \frac{V_{W_2}}{\sqrt{S_{W_2}} + \epsilon}$$

Note that ϵ is no longer in the square root. The update rule for W_2 is then:

$$W_2 \leftarrow W_2 - C_{W_2}$$

This new gradient descent update rule formula technically includes **LR decay**, **LR gradient-based control** and **momentum**.

Adam implementation

Compute all eight S and V variables (we have four trainable parameters).

Use update formulas as mentioned.

```
# Momentum and gradient decay/normalization for each parameter
V_W2, V_W1, V_b2, V_b1, S_W2, S_W1, S_b2, S_b1 = self.SV_list
V_W2 = beta1*V_W2 + (1 - beta1)*grad_W2
V_W1 = beta1*V_W1 + (1 - beta1)*grad_W1
V_b2 = beta1*V_b2 + (1 - beta1)*grad_b2
V_b1 = beta1*V_b1 + (1 - beta1)*grad_b1
V_W2_norm = V_W2/(1 - beta1**iteration_number)
V_W1_norm = V_W1/(1 - beta1**iteration_number)
V_b2_norm = V_b2/(1 - beta1**iteration_number)
V_b1_norm = V_b1/(1 - beta1**iteration_number)
S_W2 = beta2*S_W2 + (1 - beta2)*grad_W2**2
S_W1 = beta2*S_W1 + (1 - beta2)*grad_W1**2
S_b2 = beta2*S_b2 + (1 - beta2)*grad_b2**2
S_b1 = beta2*S_b1 + (1 - beta2)*grad_b1**2
S_W2_norm = S_W2/(1 - beta2**iteration_number)
S_W1_norm = S_W1/(1 - beta2**iteration_number)
S_b2_norm = S_b2/(1 - beta2**iteration_number)
S_b1_norm = S_b1/(1 - beta2**iteration_number)
self.SV_list = [V_W2, V_W1, V_b2, V_b1, S_W2, S_W1, S_b2, S_b1]

# Gradient descent update rules
eps = 1e-6
self.W2 += alpha*V_W2_norm/(np.sqrt(S_W2_norm) + eps)
self.W1 += alpha*V_W1_norm/(np.sqrt(S_W1_norm) + eps)
self.b2 += alpha*V_b2_norm/(np.sqrt(S_b2_norm) + eps)
self.b1 += alpha*V_b1_norm/(np.sqrt(S_b1_norm) + eps)
```

Adam implementation

Compute all eight S and V variables (we have four trainable parameters).

Use update formulas as mentioned.

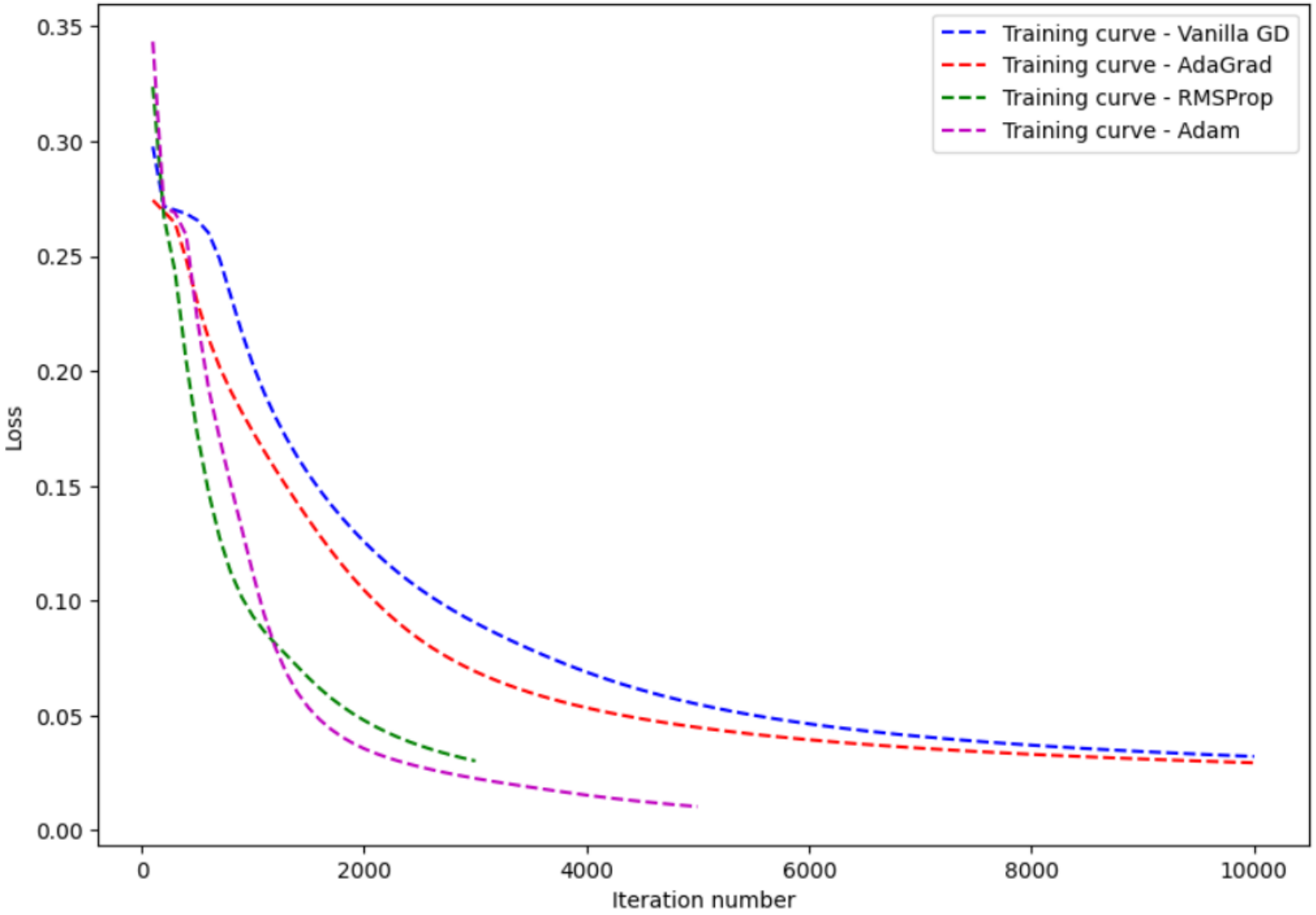
We have also implemented an optional normalization of the S and V coefficients.

This is mentioned in the Adam paper and it typically allows to include the iteration number (somewhat similar to learning rate decay).

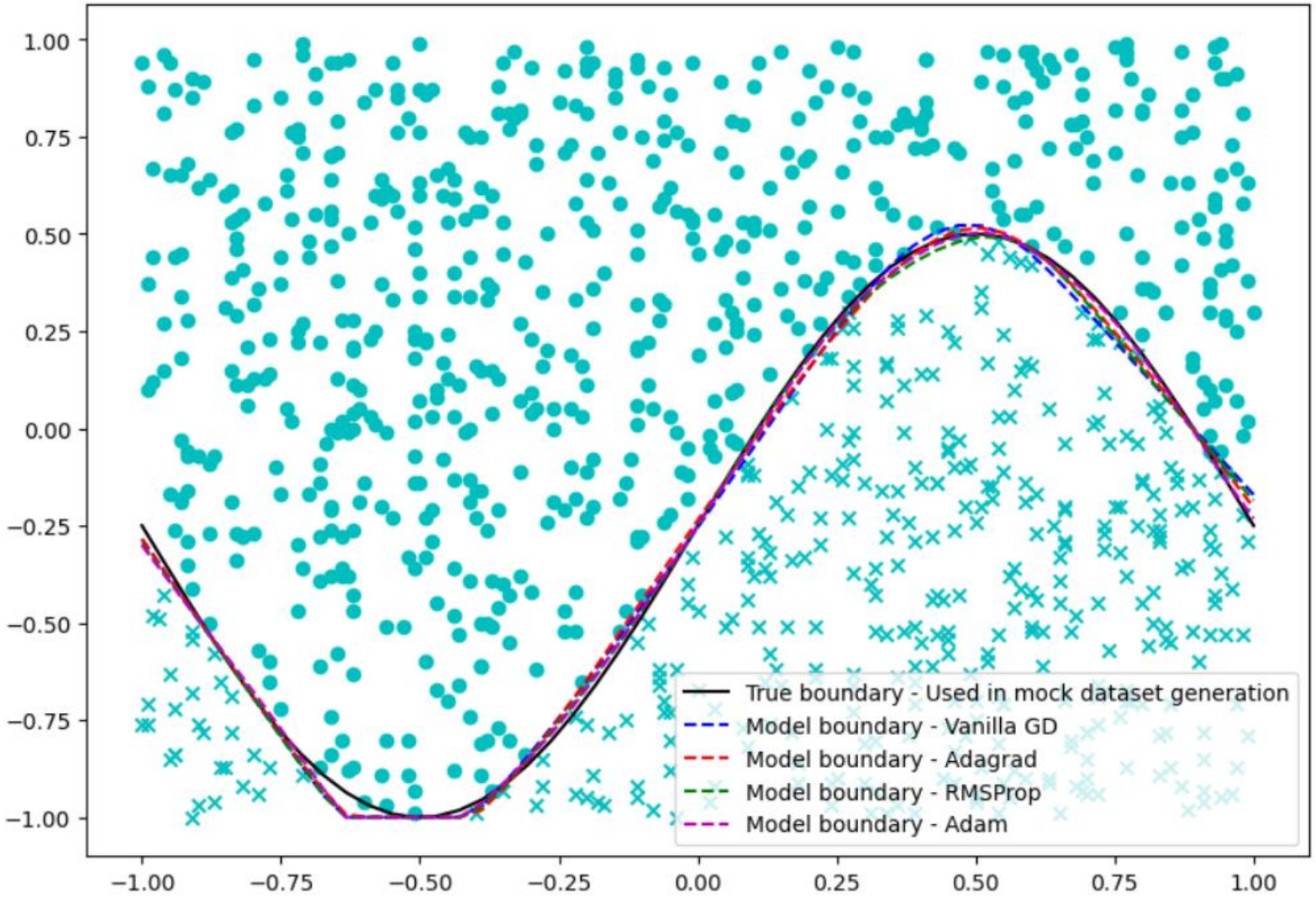
It could be freely omitted, but shown for curiosity.

```
# Momentum and gradient decay/normalization for each parameter
V_W2, V_W1, V_b2, V_b1, S_W2, S_W1, S_b2, S_b1 = self.SV_list
V_W2 = beta1*V_W2 + (1 - beta1)*grad_W2
V_W1 = beta1*V_W1 + (1 - beta1)*grad_W1
V_b2 = beta1*V_b2 + (1 - beta1)*grad_b2
V_b1 = beta1*V_b1 + (1 - beta1)*grad_b1
V_W2_norm = V_W2/(1 - beta1**iteration_number)
V_W1_norm = V_W1/(1 - beta1**iteration_number)
V_b2_norm = V_b2/(1 - beta1**iteration_number)
V_b1_norm = V_b1/(1 - beta1**iteration_number)
S_W2 = beta2*S_W2 + (1 - beta2)*grad_W2**2
S_W1 = beta2*S_W1 + (1 - beta2)*grad_W1**2
S_b2 = beta2*S_b2 + (1 - beta2)*grad_b2**2
S_b1 = beta2*S_b1 + (1 - beta2)*grad_b1**2
S_W2_norm = S_W2/(1 - beta2**iteration_number)
S_W1_norm = S_W1/(1 - beta2**iteration_number)
S_b2_norm = S_b2/(1 - beta2**iteration_number)
S_b1_norm = S_b1/(1 - beta2**iteration_number)
self.SV_list = [V_W2, V_W1, V_b2, V_b1, S_W2, S_W1, S_b2, S_b1]

# Gradient descent update rules
eps = 1e-6
self.W2 += alpha*V_W2_norm/(np.sqrt(S_W2_norm) + eps)
self.W1 += alpha*V_W1_norm/(np.sqrt(S_W1_norm) + eps)
self.b2 += alpha*V_b2_norm/(np.sqrt(S_b2_norm) + eps)
self.b1 += alpha*V_b1_norm/(np.sqrt(S_b1_norm) + eps)
```

Comparing all four optimizers



Comparing all four optimizers



PyTorch has many more optimizers

These days, most Neural Networks seem to rely on Adam.

- As with activation functions, there are a few more (niche) optimizers, implemented in PyTorch.
- Worth looking into: https://pytorch.org/docs/stable/optim.html#torch.optim.Optimizer

(Quick discussion, in bonus slides)

Algorithms

`Adadelta`	Implements Adadelta algorithm.
`Adagrad`	Implements Adagrad algorithm.
`Adam`	Implements Adam algorithm.
`AdamW`	Implements AdamW algorithm.
`SparseAdam`	Implements lazy version of Adam algorithm suitable for sparse tensors.
`Adamax`	Implements Adamax algorithm (a variant of Adam based on infinity norm).
`ASGD`	Implements Averaged Stochastic Gradient Descent.
`LBFGS`	Implements L-BFGS algorithm, heavily inspired by minFunc.
`NAdam`	Implements NAdam algorithm.
`RAdam`	Implements RAdam algorithm.

Just a question

We have a dataset of N samples, used for training. Our training procedure, so far consists of:

- Formulating predictions for all N samples in the dataset, using current model parameters,
- Using all N predictions to compute the value of the loss function L for current model parameters, essentially averaging errors over all predictions made on all samples,
- Backpropagation using the loss function we just calculated, using parameters update rules calculations and parameters adjustment.

Just a question

On each call of the backward() method, we would therefore perform **1 parameter update using all N samples in the dataset**.

This is called a **(Full) Batch Gradient Descent**.

→While this is a **slow and safe way to go**, do we really need to use all N samples in the dataset to perform one update of the parameters?

Or could we perform **more regular updates** to reduce the computational cost for each parameter update?

Stochastic Gradient Descent

Definition (Stochastic Gradient Descent [StoGD1951]):

Vanilla Gradient Descent would perform **1 parameter update using all N samples in the dataset**.

If we wanted to design the most frequent update scheme, it would probably consist of using of

- Formulating predictions **for a single sample, randomly drawn in dataset**, using current model parameters,
- Using this single prediction to compute the value of the loss function L for current model parameters,
- Backpropagation using the loss function we just calculated (parameters update rules calculations and parameters adjustment).

This procedure is called **Stochastic Gradient Descent**.

Stochastic GD: One update per sample?!

1 update using all N samples in dataset

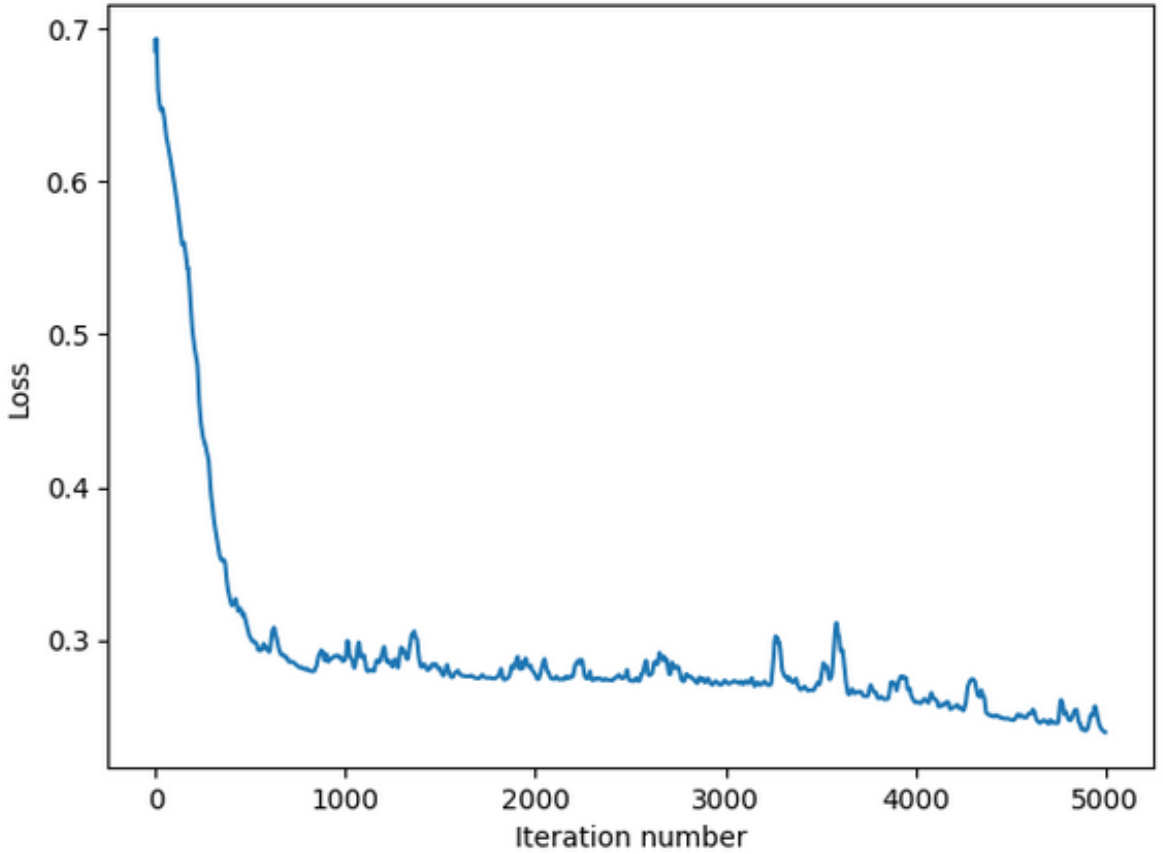
Stochastic Gradient Descent implementation

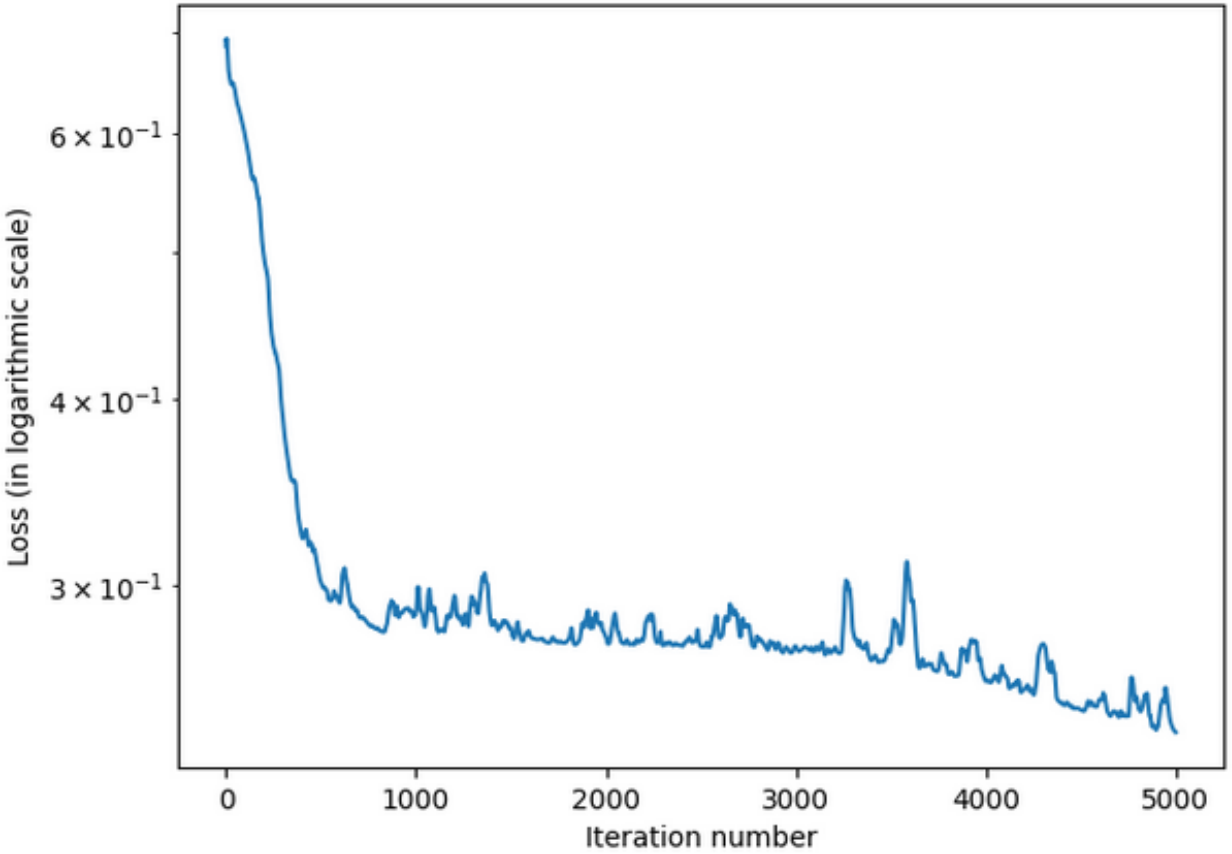
Implementing the **Stochastic Gradient Descent** requires to slightly change the trainer method of our class. **It will now draw a random sample in the dataset and use this single sample to perform the backward operations.**

```
    def train(self, inputs, outputs, N_max = 1000, alpha = 1e-5, beta1 = 0.9, beta2 = 0.999, delta = 1e-5, displa
        # Get number of samples
        M = inputs.shape[0]
        # List of losses, starts with the current loss
        self.losses_list = [self.CE_loss(inputs, outputs)]
        # Initialize G_list
        G_list = [0*self.W2, 0*self.W1, 0*self.b2, 0*self.b1, \
                  0*self.W2, 0*self.W1, 0*self.b2, 0*self.b1]
        # Repeat iterations
        for iteration_number in range(1, N_max + 1):

            # Stochastic GD on one randomly chosen sample
            indexes = np.random.randint(0, M)
            inputs_sub = np.array([inputs[indexes, :]])
            outputs_sub = np.array([outputs[indexes, :]])

            # Backpropagate
            G_list, loss = self.backward(inputs_sub, outputs_sub, G_list, iteration_number, alpha, beta1, beta2)
```

Stochastic Gradient Descent implementation

In general, slightly faster training, but a lot more erratic.





Stochastic Gradient Descent implementation

In general, slightly faster training, but a lot more erratic.

Reason: Most loss function are using mean/averaging error values over several samples, e.g. Mean Square Error.

- Using only one sample in the MSE formula will not lead to a good estimation of the Mean Square Error of the model on the dataset.
- Using all samples however leads to the best possible estimations but it is slow to compute.

Stochastic GD: worst estimation of the MSE, but fast to compute

What does the optimal middle ground consist of?

Batch GD: best estimation of MSE loss, but slow to compute.

Intuition behind Mini-batch Stochastic GD

What if we used a subset of N' $(with\ N' < N)$ randomly drawn samples from the dataset to perform one iteration of parameter update instead of

- The entire dataset as in **(Full) Batch GD**,
- Or only one sample as in **Stochastic GD**?

Definition (samples mini-batch):

We call a **samples mini-batch**, or simply **mini-batch**, a subset of $N'(with\ N' < N)$ randomly drawn samples from the dataset.

Stochastic Mini-Batch Gradient Descent

Definition (mini-batch stochastic gradient descent [MiniGD1952]):

The **mini-batch stochastic gradient descent** uses the following steps:

- Formulate predictions **for a single mini-batch of N' samples, randomly drawn in dataset**, using current model parameters,
- Using these N' predictions to compute the value of the loss function L for current model parameters,
- Backpropagation using the loss function we just calculated.

It is therefore our **middle ground**.

Stochastic GD: worst estimation of the MSE, but fast to compute

What does the optimal middle ground consist of?
→ Mini-batch stochastic GD!

Batch GD: best estimation of MSE loss, but slow to compute.

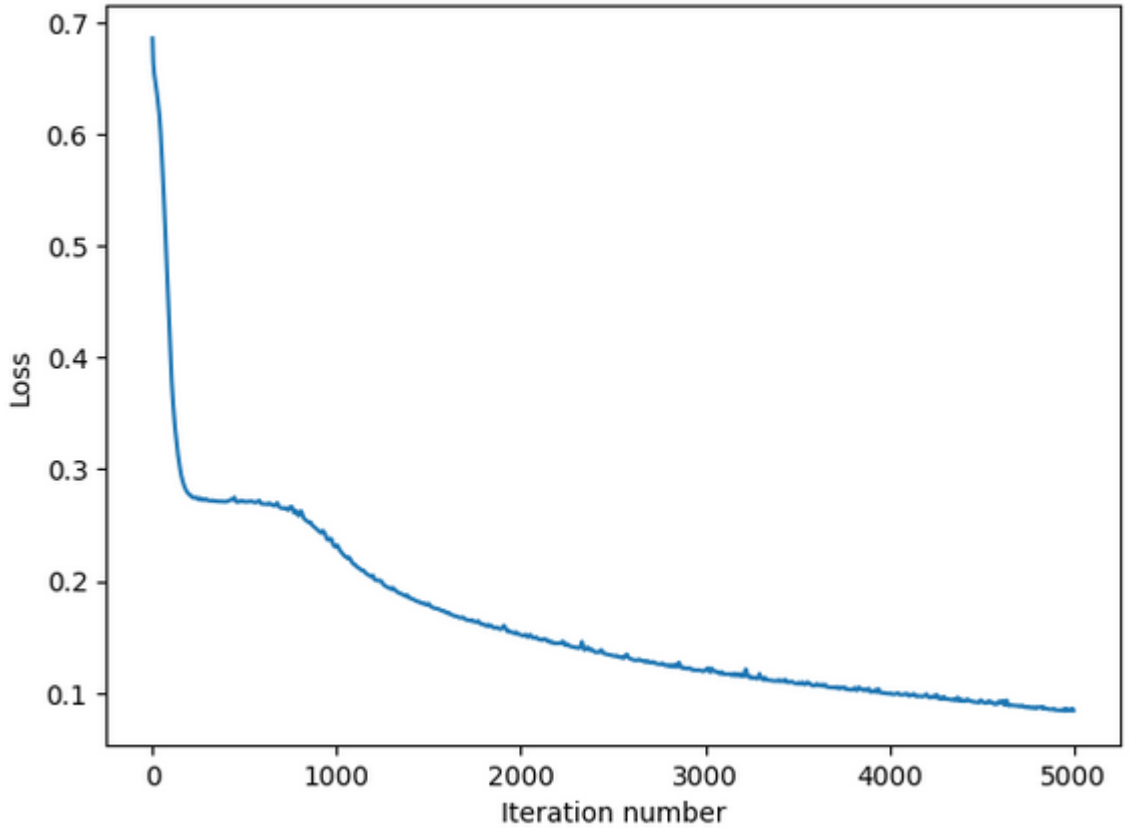
Mini-batch Stochastic GD implementation

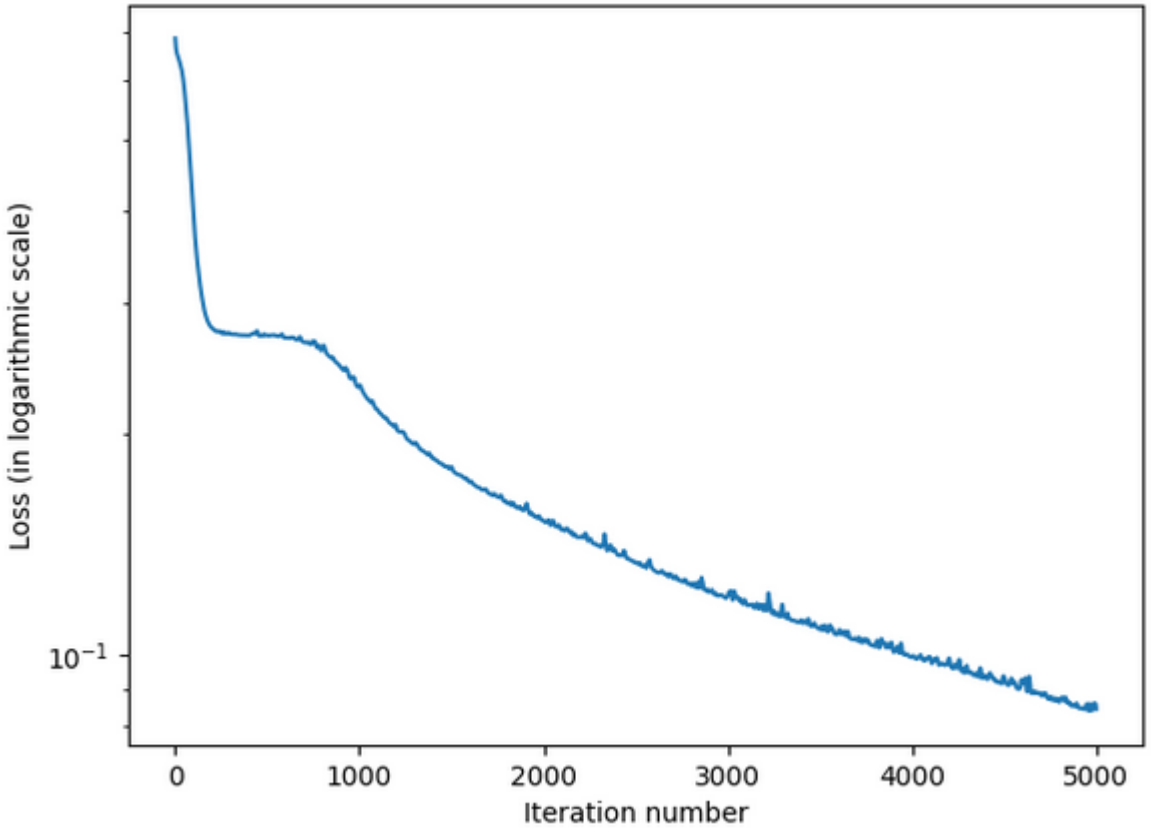
Our trainer **will now draw N' random sample in the dataset and use these to perform the backward operations.**

```python
    def train(self, inputs, outputs, N_max = 1000, alpha = 1e-5, beta1 = 0.9, beta2 = 0.999, \
              delta = 1e-5, batch_size = 100, display = True):
        # Get number of samples
        M = inputs.shape[0]
        # List of losses, starts with the current loss
        self.losses_list = [self.CE_loss(inputs, outputs)]
        # Initialize G_list
        G_list = [0*self.W2, 0*self.W1, 0*self.b2, 0*self.b1, \
                  0*self.W2, 0*self.W1, 0*self.b2, 0*self.b1]

        # Define RNG for stochastic minibatches
        rng = default_rng()

        # Repeat iterations
        for iteration_number in range(1, N_max + 1):

            # Select a subset of inputs and outputs with given batch size
            shuffler = rng.choice(M, size = batch_size, replace = False)
            inputs_sub = inputs[shuffler, :]
            outputs_sub = outputs[shuffler, :]

            # Backpropagate
            G_list, loss = self.backward(inputs_sub, outputs_sub, G_list, iteration_number, alpha, beta1, beta2)
```

Mini-batch Stochastic GD implementation

In general, good performance, might be erratic towards late iterations, but – in general – not as much as stochastic GD.





A quick note on batch sizes

A few remarks regarding batch sizes:

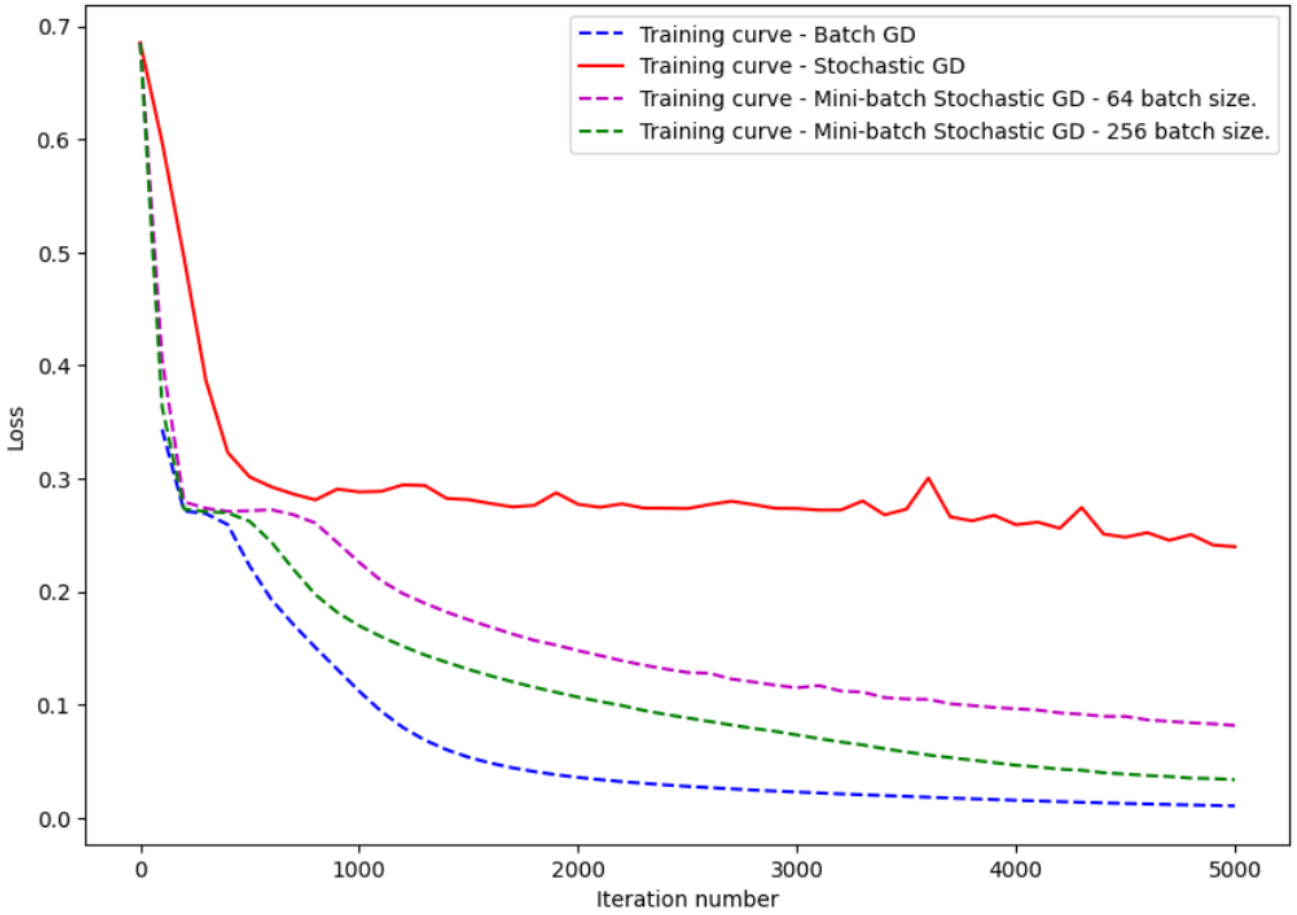
- It is often a good idea to choose a **batch size N'**, defined as a power of 2, between 32 and 512, that is $N' = \{32, 64, 128, 256, 512\}$.
- In general, **a larger batch size means slower computation but better training performance**.
- As in all things so far, **it will be about finding the correct balance!**

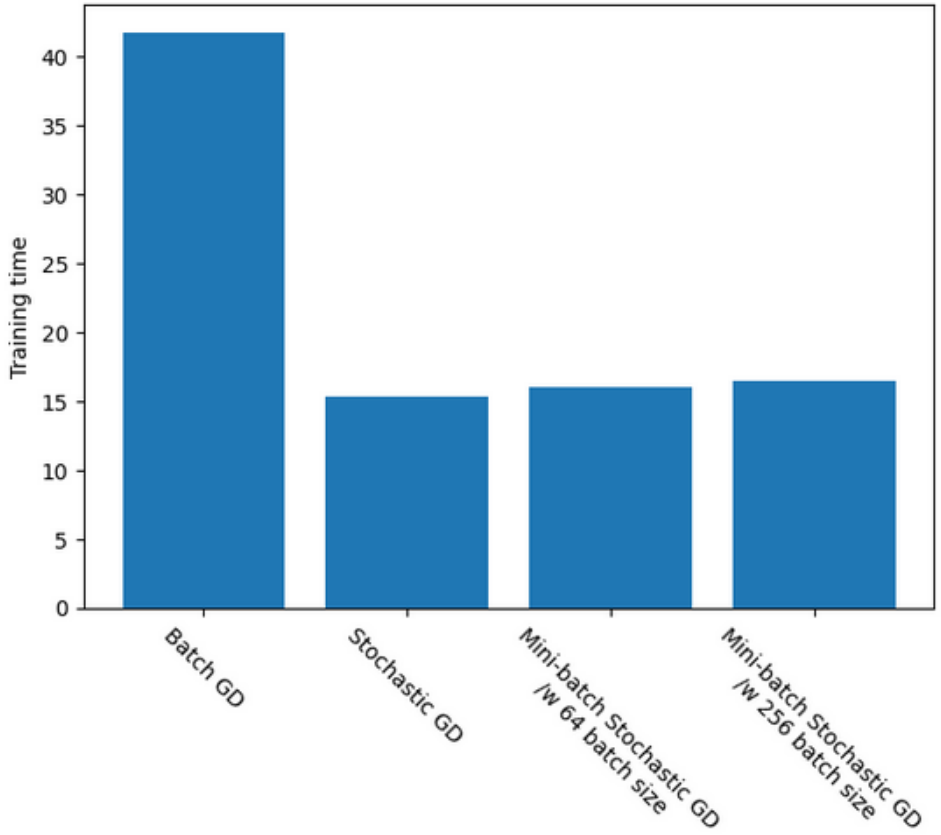
Stochastic GD: worst estimation of the MSE, but fast to compute

What does the optimal middle ground consists of?
→ Mini-batch stochastic GD!

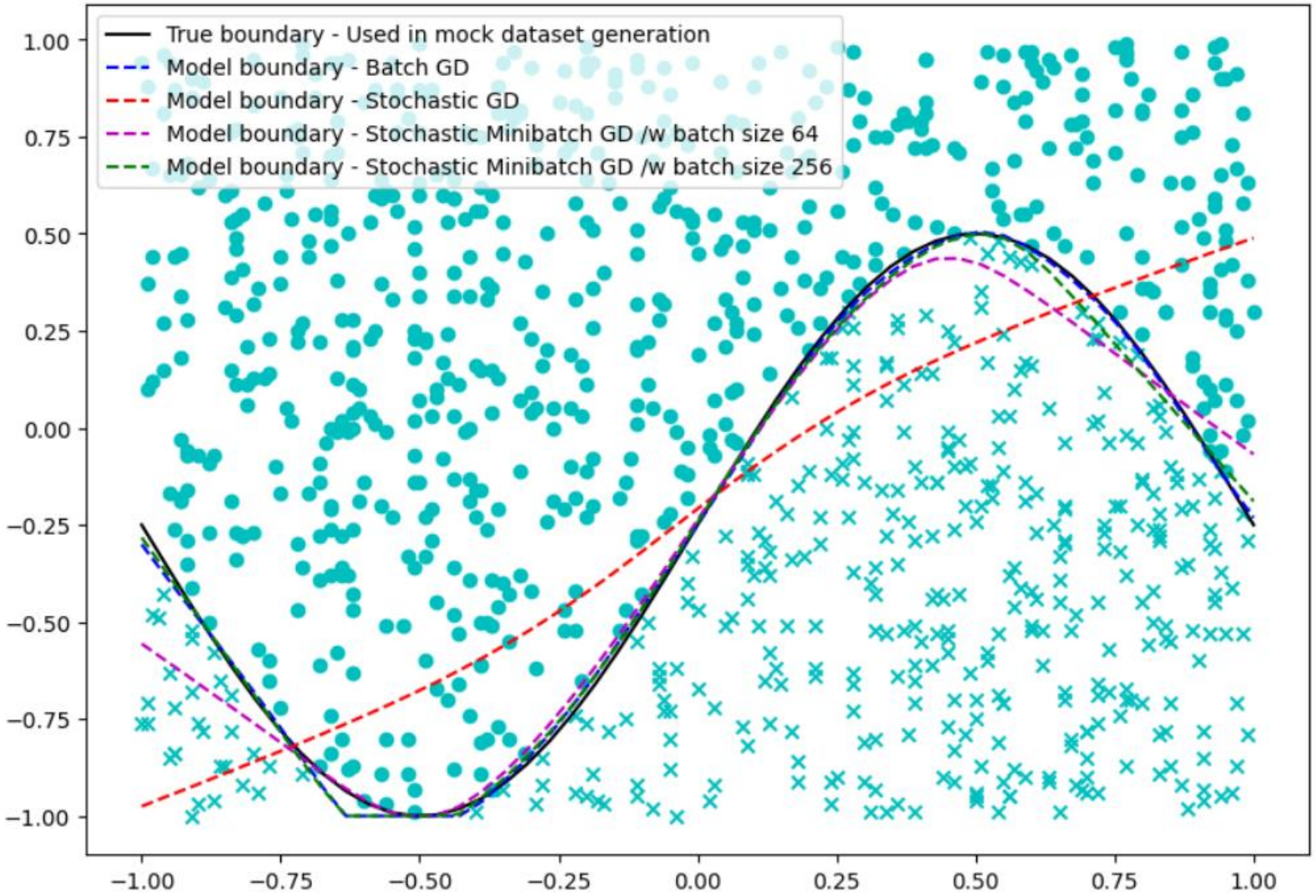
Batch GD: best estimation of MSE loss, but slow to compute.

Mini-batch Stochastic GD implementation





Mini-batch Stochastic GD implementation



Combining everything into a great optimizer

From now on, we will be:

- Using an **advanced optimizer** (like **Adam**), implementing **momentum** and **gradient-based learning rate control**,
- Using a **mini-batch stochastic gradient descent** procedure, drawing mini-batches to perform trainable parameter updates instead of using the entire dataset for each update,
- Trying some values for the different hyperparameters (learning rate, batch size, momentum/optimizers parameters, number of iterations, etc.). Often a good idea to start with recommended values and explore from there!

Combining everything into a great optimizer

In an upcoming lesson, we will implement more controls on the learning rate and gradient descent algorithm, for instance:

- **Learning rate decay:** decrease the value of learning rate over iterations like before. (Feel free to try implementing it for practice?)
- **Early stopping:** stop iterations of GD, if network has already achieved a good performance (Done a few times, more in next lecture).
- **Gradient clipping:** prevent gradient values from going above a certain threshold to avoid large modifications on a single iteration (Mentioned in previous lecture, will be implemented on W10 for other purposes).
- Etc.

Learn more about these topics

Out of class, supporting papers, for those of you who are curious.

- [AdaGrad2011] J. Duchi, E. **Hazan**, and Y. Singer, “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”, 2011.
- [RMSProp2012] G. **Hinton** “Neural Networks for Machine Learning”, 2012.
- [Adam2015] D. **Kingma** and J. **Ba** “Adam: A Method for Stochastic Optimization”, 2015

Learn more about these topics

Out of class, supporting papers, for those of you who are curious.

- [StoGD1951] H. Robbins and S. Monro, “A Stochastic Approximation Method”, 1951.
- [MiniGD1952] Kiefer and Wolfowitz, “On the Use of Stochastic Approximation Methods in Optimization and Control Problems”, 1952.
- Leon Bottou, Yoshua Bengio and Yann LeCun have also worked on implementations of the Mini-batch stochastic Gradient Descent for Neural Networks. https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Elad Hazan: Professor** at **Princeton University**, **Director** and **co-founder** of **Google AI Princeton**.
  https://www.ehazan.com
  https://scholar.google.com/citations?user=LnhCGNMAAAAJ&hl=fr
- **Diederik P. Kingma: Research Scientist** at **Google Brain.**
  http://dpkingma.com/
  https://scholar.google.nl/citations?user=yyIoQu4AAAAJ&hl=en
- **Jimmy Ba: Assistant Professor** at **University of Toronto.**
  https://jimmylba.github.io/
  https://scholar.google.ca/citations?user=ymzxRhAAAAAJ&hl=en

Learn more about these topics

Some extra (easy) reading and videos for those of you who are curious.

- [Analy_Vidhya_Optim] Learn more about optimizers. https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/
- [MLM_GD] Learn more about stochastic mini-batch gradient descent. https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/
- Is gradient descent the only way to train parameters anyway? An introduction to the new 2022 training algorithm, from Hinton, called the forward-forward method. https://bdtechtalks.com/2022/12/19/forward-forward-algorithm-geoffrey-hinton/

Other forms of GD with momentum exist

Definition (Gradient Descent with Nesterov Momentum):

Nesterov **accelerated gradient descent (NAG)** is a variant of momentum gradient descent that can help accelerate convergence and improve the optimization of deep learning models.

It does this by incorporating the concept of momentum, although **in a slightly different way** than before.

This could help the optimization algorithm to continue moving in the same direction even if the gradients change, but with **less oscillations than observed with the previous momentum.**

In my opinion, this is neither better or worse, than the previous formula, just different (have I mentioned the NFL theorem yet?).

Other forms of GD with momentum exist

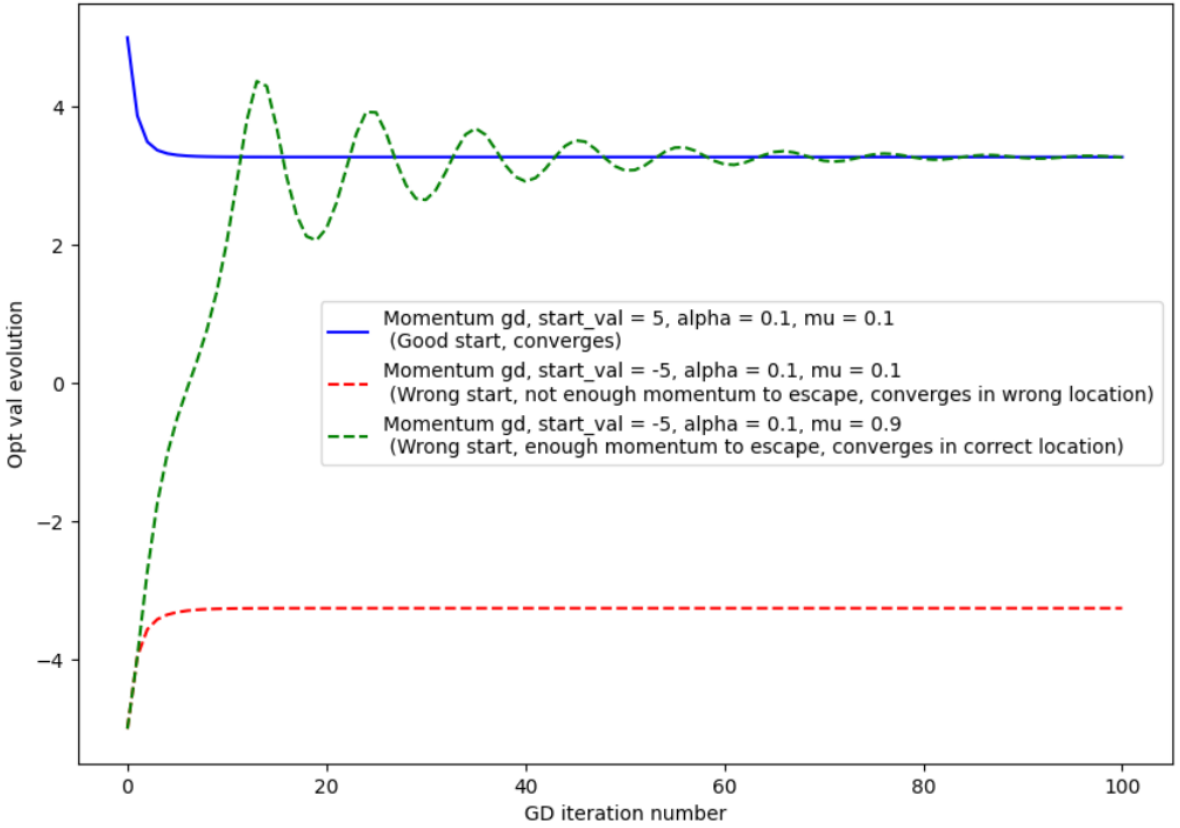
Definition (Gradient Descent with Nesterov Momentum):

Nesterov accelerated gradient descent (NAG) is a variant of momentum gradient descent that can help accelerate convergence and improve the optimization of deep learning models.

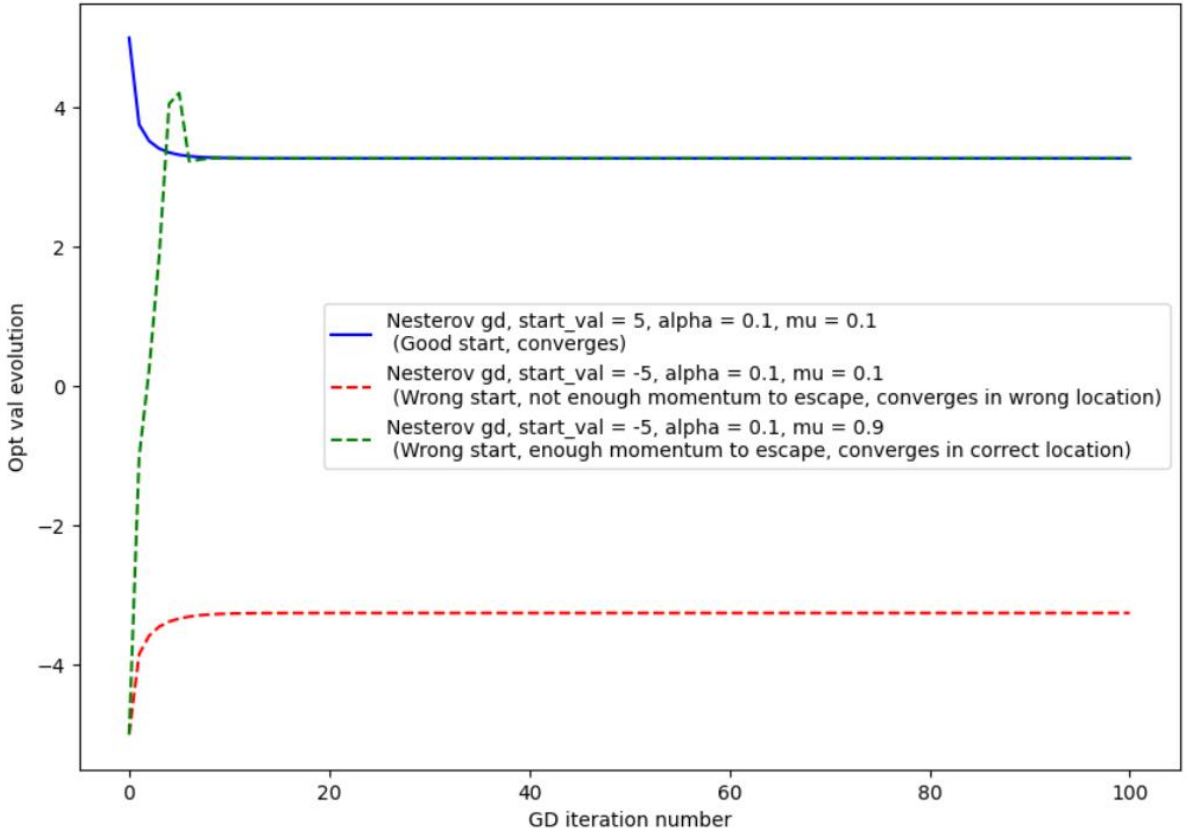
```python
def nesterov_gd(start_val, alpha = 0.1, mu = 0.1, n_iter = 100):
    val = start_val
    mom = 0
    list_vals = [val]
    for iter_num in range(n_iter):
        # Compute gradient and Nesterov momentum
        grad = -alpha*obj_fun_deriv(val)
        mom = mu*mom + grad
        # Update value
        val += mu*mom + grad
        list_vals.append(val)
    return val, list_vals
```

Comparing both momentum GD

Gradient Descent with Added Momentum



Nesterov accelerated gradient descent (NAG)



PyTorch has many more optimizers

These days, most Neural Networks seem to rely on Adam.

- As with activation functions, there are a few more (niche) optimizers, implemented in PyTorch.
- Worth looking into: https://pytorch.org/docs/stable/optim.html#torch.optim.Optimizer

(Quick discussion, in bonus slides)

Algorithms

`Adadelta`	Implements Adadelta algorithm.
`Adagrad`	Implements Adagrad algorithm.
`Adam`	Implements Adam algorithm.
`AdamW`	Implements AdamW algorithm.
`SparseAdam`	Implements lazy version of Adam algorithm suitable for sparse tensors.
`Adamax`	Implements Adamax algorithm (a variant of Adam based on infinity norm).
`ASGD`	Implements Averaged Stochastic Gradient Descent.
`LBFGS`	Implements L-BFGS algorithm, heavily inspired by minFunc.
`NAdam`	Implements NAdam algorithm.
`RAdam`	Implements RAdam algorithm.

PyTorch has many more optimizers

There are several recent optimizers that are worth looking into beyond Adam, typically **AdaBound** , **AMSGrad**, and **Lookahead**.

- **AdaBound** is an improved version of Adam that combines the advantages of Adam and the AdaGrad algorithm.
- **AMSGrad**, which stands for Adaptive Moment Estimation with Simulated Annealing and Gradient Descent, is an improved version of Adam that utilizes a special bounding mechanism to prevent overshooting and divergence.
- Finally, **Lookahead** is an algorithm that combines the advantages of several optimization techniques, such as Adam and momentum, to speed up the optimization process.

These are out of scope, but worth mentioning for the curious reader.