Practice of Deep Learning

Day 2, Part 1/4

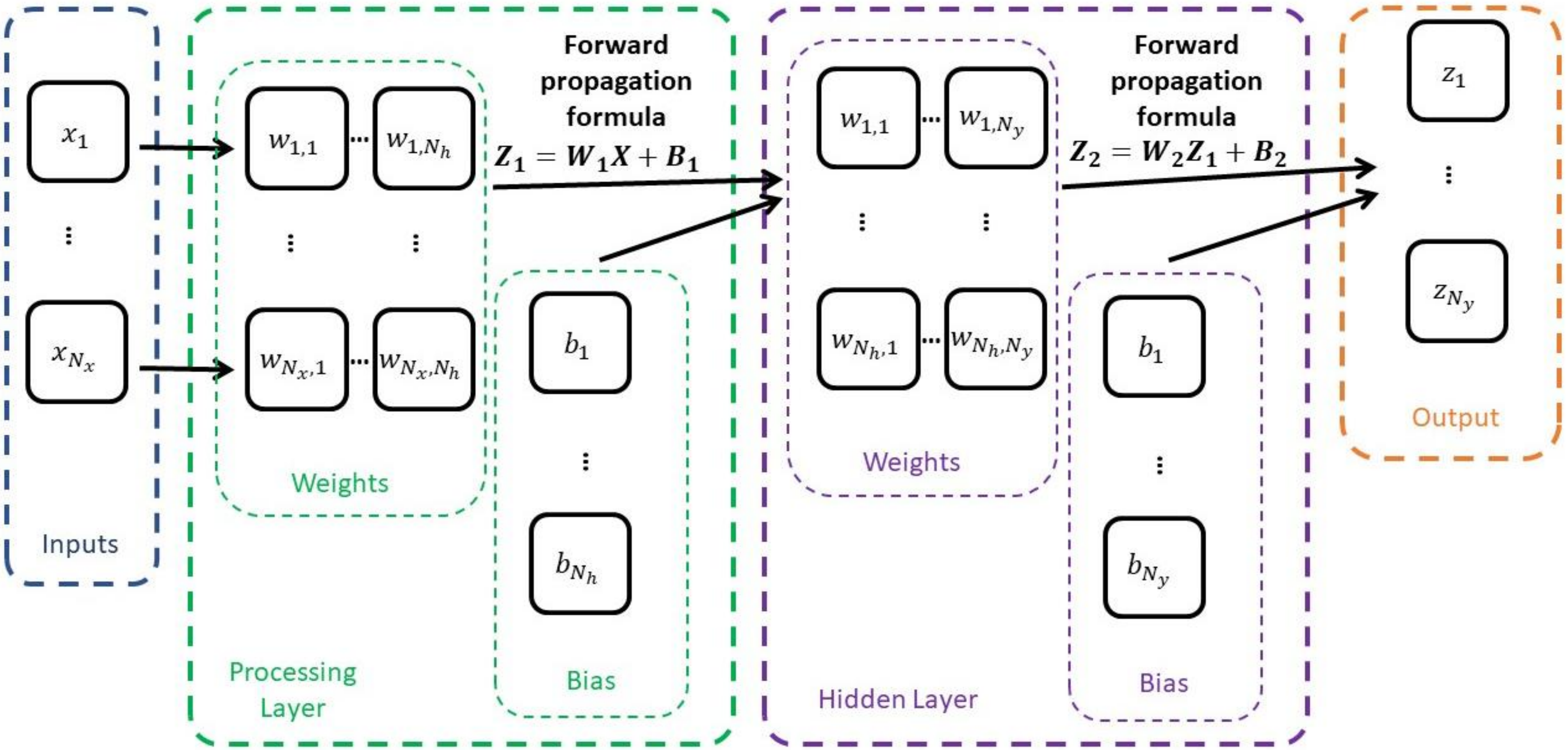
Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

About this lecture

1. What are the **typical initializers for trainable parameters**?
2. What is **symmetry** and why is it essential to **break it**?
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. What are **activation functions** and what is their use?
8. What is the **universal approximation theorem**?
9. What is the **no free lunch theorem**?

Our shallow NN with two processing layers



Our NN class

How it works:

- Weights and biases matrices are now initialized as normal random with zero mean and variance 0.1.
- Forward method for making predictions using current parameters.
- Loss function for performance evaluation.

```python
class ShallowNeuralNet():

    def __init__(self, n_x, n_h, n_y):
        # Network dimensions
        self.n_x = n_x
        self.n_h = n_h
        self.n_y = n_y
        # Weights and biases matrices
        self.W1 = np.random.randn(n_x, n_h)*0.1
        self.b1 = np.random.randn(1, n_h)*0.1
        self.W2 = np.random.randn(n_h, n_y)*0.1
        self.b2 = np.random.randn(1, n_y)*0.1
        # Loss, initialized as infinity before first calculation is made
        self.loss = float("Inf")

    def forward(self, inputs):
        # Wx + b operation for the first layer
        Z1 = np.matmul(inputs, self.W1)
        Z1_b = Z1 + self.b1
        # Wx + b operation for the second layer
        Z2 = np.matmul(Z1_b, self.W2)
        Z2_b = Z2 + self.b2
        return Z2_b

    def MSE_loss(self, inputs, outputs):
        # MSE loss function as before
        outputs_re = outputs.reshape(-1, 1)
        pred = self.forward(inputs)
        losses = (pred - outputs_re)**2
        self.loss = np.sum(losses)/outputs.shape[0]
        return self.loss
```

```
# Define neural network structure
n_x = 2
n_h = 4
n_y = 1
shallow_neural_net = ShallowNeuralNet(n_x, n_h, n_y)
print(shallow_neural_net.__dict__)
```

```
{'n_x': 2, 'n_h': 4, 'n_y': 1, 'W1': array([[-0.10476816,  0.18570216,  0.03204007, -0.10951262],
       [-0.13867874, -0.03539496, -0.02856421,  0.20592501]]), 'b1': array([[ 0.0232776 , -0.16122469,  0.00718537,  0.0666335
1]]), 'W2': array([[ 0.03321156],
       [-0.0336505 ],
       [ 0.04977554],
       [-0.1794089 ]]), 'b2': array([[0.03460341]]), 'loss': inf}
```

1. Initialize model
2. Forward to predict
3. Compute loss to evaluate model (it is bad, because trainable parameters have received random values!)

```
pred = shallow_neural_net.forward(inputs)
print(pred.shape)
print(outputs.shape)
print(pred[0:5])
print(outputs[0:5])
```

```
(100, 1)
(100, 1)
[[-23.24055489]
 [-31.54945952]
 [-12.75105332]
 [ -2.49026451]
 [-32.3803654 ]]
[[1.581913]
 [3.450274]
 [2.978769]
 [2.808258]
 [2.556398]]
```

```
loss = shallow_neural_net.MSE_loss(inputs, outputs)
print(loss)
```

```
677.625448852107
```

Backpropagation in our model

Update rules for W_2 and b_2

(Obtained after using chain rule to compute derivatives wrt. parameters for loss)

$$W_2 \leftarrow W_2 - \frac{2\epsilon\alpha}{M}(W_1 X + b_1)$$

$$b_2 \leftarrow b_2 - \frac{2\epsilon\alpha}{M}$$

```
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
    Z2 = np.matmul(Z1_b, self.W2)
    y_pred = Z2 + self.b2

    # Compute error term
    epsilon = y_pred - outputs

    # Compute the gradient for W2 and b2
    dL_dW2 = (2/m)*np.matmul(Z1_b.T, epsilon)
    dL_db2 = (2/m)*np.sum(epsilon, axis = 0, keepdims = True)

    # Compute the loss derivative with respect to the first layer
    dL_dZ1 = np.matmul(epsilon, self.W2.T)
```

Backpropagation in our model

Update rules for W_1 and b_1

(Obtained after using chain rule to compute derivatives wrt. parameters for loss)

$$W_1 \leftarrow W_1 - \frac{2\epsilon\alpha}{M} W_2 X$$

$$b_1 \leftarrow b_1 - \frac{2\epsilon\alpha}{M} W_2$$

```
# Compute the loss derivative with respect to the first layer
dL_dZ1 = np.matmul(epsilon, self.W2.T)

# Compute the gradient for W1 and b1
dL_dW1 = (2/m)*np.matmul(inputs.T, dL_dZ1)
dL_db1 = (2/m)*np.sum(dL_dZ1, axis = 0, keepdims = True)

# Update the weights and biases using gradient descent
self.W1 -= alpha*dL_dW1
self.b1 -= alpha*dL_db1
self.W2 -= alpha*dL_dW2
self.b2 -= alpha*dL_db2

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

Training procedure, in short.

Forward pass on dataset samples. Compute errors and loss function.

Use new loss value to compute gradients. Adjust parameters with backprop.

Repeat until convergence or max number of iterations, as in GD earlier.

Made that iterative process a trainer method for our class.

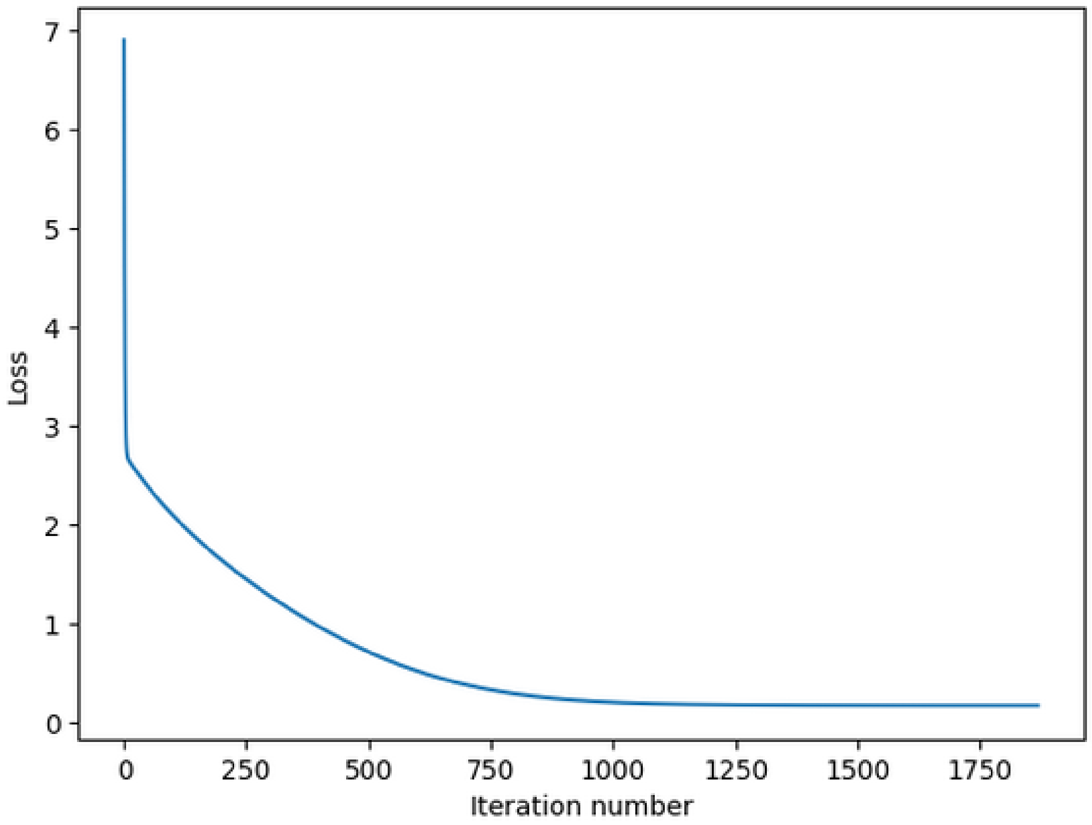
```
    def train(self, inputs, outputs, N_max = 1000, alpha = 1e-5, delta = 1e-5, display = True):
        # List of losses, starts with the current loss
        self.losses_list = [self.loss]
        # Repeat iterations
        for iteration_number in range(1, N_max + 1):
            # Backpropagate
            self.backward(inputs, outputs, alpha)
            new_loss = self.loss
            # Update losses list
            self.losses_list.append(new_loss)
            # Display
            if(display):
                print("Iteration {} - Loss = {}".format(iteration_number, new_loss))
            # Check for delta value and early stop criterion
            difference = abs(self.losses_list[-1] - self.losses_list[-2])
            if(difference < delta):
                if(display):
                    print("Stopping early - loss evolution was less than delta.")
                break
        else:
            # Else on for loop will execute if break did not trigger
            if(display):
                print("Stopping - Maximal number of iterations reached.")

    def show_losses_over_training(self):
        # Initialize matplotlib
        fig, axs = plt.subplots(1, 2, figsize = (15, 5))
        axs[0].plot(list(range(len(self.losses_list))), self.losses_list)
        axs[0].set_xlabel("Iteration number")
        axs[0].set_ylabel("Loss")
        axs[1].plot(list(range(len(self.losses_list))), self.losses_list)
        axs[1].set_xlabel("Iteration number")
        axs[1].set_ylabel("Loss (in logarithmic scale)")
        axs[1].set_yscale("log")
        # Display
        plt.show()
```

Trainer function for our model

```
1  losses_list = train(shallow_neural_net, inputs, outputs, N_max = 10000, alpha = 1e-5, delta = 1e-6, display = True)
```

```
Iteration 1 - Loss = 4.639845655363769
Iteration 2 - Loss = 3.6055739235388646
Iteration 3 - Loss = 3.1219544752264055
Iteration 4 - Loss = 2.891370725402625
Iteration 5 - Loss = 2.778663798354082
Iteration 6 - Loss = 2.721333641596964
Iteration 7 - Loss = 2.6901646643755135
Iteration 8 - Loss = 2.671407441095785
Iteration 9 - Loss = 2.6585626629544303
Iteration 10 - Loss = 2.648548914291606
Iteration 11 - Loss = 2.639902127093024
Iteration 12 - Loss = 2.6319255476288577
Iteration 13 - Loss = 2.6242871305644138
Iteration 14 - Loss = 2.6168284145839156
Iteration 15 - Loss = 2.60947364178999
Iteration 16 - Loss = 2.6021864900028118
Iteration 17 - Loss = 2.5949494536811937
Iteration 18 - Loss = 2.58775401064343
```



A bad initialization?

In our previous version of the model, we initialized our parameters randomly.

- While it seemed unimportant, this is absolutely essential.
- **Initializing all parameters as identical constants (e.g. setting all to zeroes) would in fact be a terrible idea.**
- But we have to try it at least once to understand why.

```python
class ShallowNeuralNet():

    def __init__(self, n_x, n_h, n_y):
        # Network dimensions
        self.n_x = n_x
        self.n_h = n_h
        self.n_y = n_y
        # Weights and biases matrices (randomly initialized)
        self.W1 = np.random.randn(n_x, n_h)*0.1
        self.b1 = np.random.randn(1, n_h)*0.1
        self.W2 = np.random.randn(n_h, n_y)*0.1
        self.b2 = np.random.randn(1, n_y)*0.1
        # Loss, initialized as infinity before first calculation is made
        self.loss = float("Inf")
```

```python
class ShallowNeuralNet_ConstantInit():

    def __init__(self, n_x, n_h, n_y, const_val):
        # Network dimensions
        self.n_x = n_x
        self.n_h = n_h
        self.n_y = n_y
        # Weights and biases matrices (all initialized as constant
        # matrices filled with 0.1 values)
        self.W1 = np.ones(shape = (n_x, n_h))*const_val
        self.b1 = np.ones(shape = (1, n_h))*const_val
        self.W2 = np.ones(shape = (n_h, n_y))*const_val
        self.b2 = np.ones(shape = (1, n_y))*const_val
        # Loss, initialized as infinity before first calculation is made
        self.loss = float("Inf")
```

Training the constant initialization model

```python
# Define and train neural network structure
n_x = 2
n_h = 4
n_y = 1
np.random.seed(37)
shallow_neural_net = ShallowNeuralNet_ConstantInit(n_x, n_h, n_y, 0.1)
shallow_neural_net.train(inputs, outputs, N_max = 10000, alpha = 1e-6, delta = 1e-6, display = True)
```

```
Iteration 1 - Loss = 618.4538740049891
W1, b1, W2, b2:
[[0.09938446 0.09938446 0.09938446 0.09938446]
 [0.09676805 0.09676805 0.09676805 0.09676805]]
[[0.09999521 0.09999521 0.09999521 0.09999521]]
[[0.09614772]
 [0.09614772]
 [0.09614772]
 [0.09614772]]
[[0.09995206]]
Iteration 501 - Loss = 1.3726340052732087
W1, b1, W2, b2:
[[0.09064391 0.09064391 0.09064391 0.09064391]
 [0.04434246 0.04434246 0.04434246 0.04434246]]
[[0.09992629 0.09992629 0.09992629 0.09992629]]
[[0.0145902]
 [0.0145902]
 [0.0145902]
 [0.0145902]]
```

Same values on all groups of weights

Same thing for bias vector

Keeps happening without discontinuity no matter the number of iterations…
This means all the neurons in a given layer are doing the same thing! (a.k.a. **symmetry** in the neural network)

Symmetry in the Neural Network

Definition (Symmetry):

Symmetry in a neural network is the **tendency of all neurons to have the same weights and processes (hence doing the same thing).**

As a **lack of diversity**, this can lead to a **lack of generalization** and can prevent the NN from learning complex patterns.

```
Iteration 1 - Loss = 618.4538740049891
W1, b1, W2, b2:
[[0.09938446 0.09938446 0.09938446 0.09938446]
 [0.09676805 0.09676805 0.09676805 0.09676805]]
[[0.09999521 0.09999521 0.09999521 0.09999521]]
[[0.09614772]
 [0.09614772]
 [0.09614772]
 [0.09614772]]
[[0.09995206]]
Iteration 501 - Loss = 1.372634005273208
W1, b1, W2, b2:
[[0.09064391 0.09064391 0.09064391 0.09064391]
 [0.04434246 0.04434246 0.04434246 0.04434246]]
[[0.09992629 0.09992629 0.09992629 0.09992629]]
[[0.0145902]
 [0.0145902]
 [0.0145902]
 [0.0145902]]
```

Symmetry in the Neural Network

Definition (Symmetry):

Symmetry in a neural network is the **tendency of all neurons to have the same weights and processes (hence doing the same thing).**

As a **lack of diversity**, this can lead to a **lack of generalization** and can prevent the NN from learning complex patterns.

- Essentially, this happened because **all weights and biases had the same starting point.**
- The backward process then updated the parameters identically, and they end up keeping the same values over the course of training.

→ Need to enforce diversity on these parameters from the start! Hence, some random initialization.

The random normal initialization

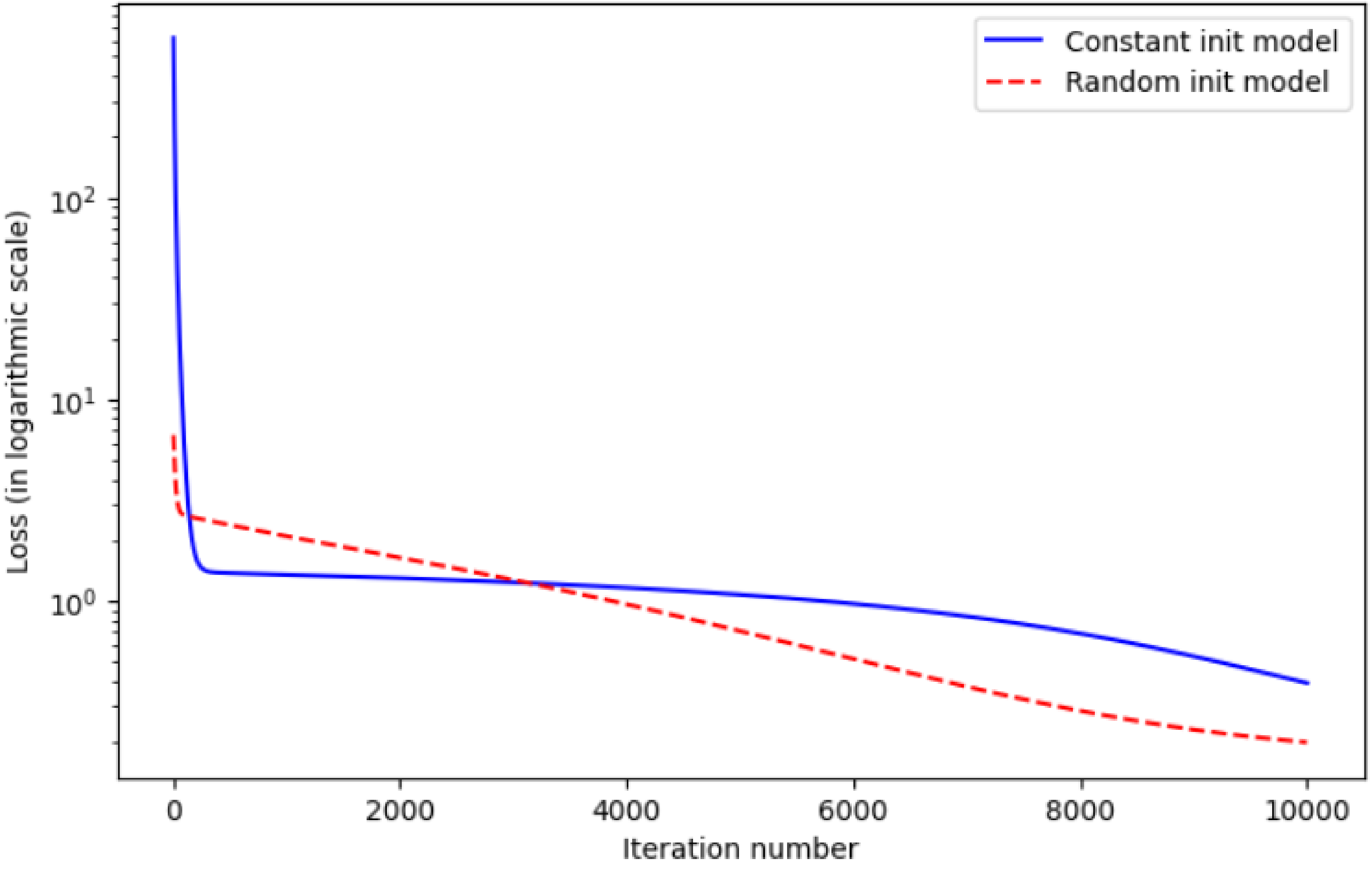
Definition (the random normal initialization [LeCun1998]**):**

The **normal random initializer** will setup **parameters randomly,** by sampling from a **normal distribution**, with a **mean 0** and a **standard deviation of 1** (or lower).

This initializer is **pretty straightforward** but can lead to **slow convergence or even divergence as the network gets deeper**.

```python
    def init_parameters_normal(self):
        # Weights and biases matrices (randomly initialized)
        self.W1 = np.random.randn(self.n_x, self.n_h)*0.1
        self.b1 = np.random.randn(1, self.n_h)*0.1
        self.W2 = np.random.randn(self.n_h, self.n_y)*0.1
        self.b2 = np.random.randn(1, self.n_y)*0.1
```

Using a random initialization is just better



The Xavier initialization

Definition (the Xavier initialization [Glorot2010]**):**

The **Xavier initializer** is based on a **Gaussian distribution**, with parameters initialized to zero-mean and a **variance** adjusted to the number of input/output parameters of the NN (e.g. $\frac{\mathbf{2}}{\boldsymbol{N_x+N_y}}$, with N_x being the input size and N_y the output size).

It is **an overall good initializer for most architectures**.

```python
    def init_parameters_xavier(self):
        # Weights and biases matrices (Xavier initialized)
        var = np.sqrt(2.0/(self.n_x + self.n_y))
        self.W1 = np.random.randn(self.n_x, self.n_h)*var
        self.b1 = np.random.randn(1, self.n_h)*var
        self.W2 = np.random.randn(self.n_h, self.n_y)*var
        self.b2 = np.random.randn(1, self.n_y)*var
```

Initializations variations and more

Many more initialization formulas exist:

- Xavier has random uniform and normal variations,
- Glorot proposed more initializations, same for He,
- Orthogonal initializations are sometimes useful but rare,
- Etc.

PyTorch has listed a lot of them, ready to use:
https://pytorch.org/docs/stable/nn.init.html

(Have a look at bonus slides if curious!)

No free lunch theorem

Very important note: What I listed earlier about initializers are empirical observations only, certainly not absolute rules.

- For this reason, it is a good idea to **try a few different initialization techniques and see which one works best** for your problem.
- Some recent research has also suggested that using a combination of different initialization techniques (e.g., Xavier initialization for some layers and He initialization for others) can improve performance.

In practice, this phenomenon is something that the machine learning community likes to call the “**No free lunch theorem**“.

No free lunch theorem

Definition (the No Free Lunch (or NFL) theorem):

The **no free lunch (NFL) theorem** for supervised machine learning is a theorem that essentially implies that **no single machine learning algorithm or tool is universally the best-performing algorithm for all problems**.

(https://en.wikipedia.org/wiki/No_free_lunch_theorem).

When in doubt about a supposedly "ground-breaking new technique beating all other techniques", **try them out**.

No free lunch theorem – season 1

Definition (the No Free Lunch (or NFL) theorem – Season 1):

The **no free lunch theorem** then implies that there is **no such thing "one initializer that works best for all models and problems".**

No choice: try some of them and see how it goes!

(A good read on this matter, also: https://www.deeplearning.ai/ai-notes/initialization/index.html)

When in doubt about a supposedly "ground-breaking new technique beating all other techniques", **try them out**.

A weird initializer and a new problem

Definition (our custom “weird” initializer):

Let us consider the **“weird” initializer** below. It uses a **random uniform distribution**, **scaling parameters based on the input and output sizes**, like in Xavier. If you check the bonus slides, you could in fact see this initializer as a **random uniform variation of Xavier**.

And yet, when we use it, something weird happens.

```
    def init_parameters_weird(self):
        # Weights and biases matrices (Weird? initialized)
        init_val = np.sqrt(6.0/(self.n_x + self.n_y))
        self.W1 = np.random.uniform(-init_val, init_val, (self.n_x, self.n_h))
        self.b1 = np.random.uniform(-init_val, init_val, (1, self.n_h))
        self.W2 = np.random.uniform(-init_val, init_val, (self.n_h, self.n_y))
        self.b2 = np.random.uniform(-init_val, init_val, (1, self.n_y))
```

```python
# Define and train neural network structure (Weird initialization)
n_x = 2
n_h = 4
n_y = 1
init_type = "Weird"
np.random.seed(37)
shallow_neural_net_weird = ShallowNeuralNet(n_x, n_h, n_y, init_type)
# Train and show final loss
shallow_neural_net_weird.train(inputs, outputs, N_max = 100, alpha = 1e-6, delta = 1e-6, display = True)
```

```
- Gradients:
[[ 0.0347825  -0.28526197 -0.24439365  0.18661634]
 [ 0.17610429 -1.44428536 -1.23736846  0.94484113]]
[[ 0.00026903 -0.00220639 -0.00189029  0.0014434 ]]
[[-0.64676422]
 [-0.55320161]
 [ 1.42348809]
 [-0.81598256]]
[[-0.00168314]]
- Parameters:
[[ 1.25722625 -0.1015457  -0.86890687  0.23163369]
 [ 0.33964944  0.52106421 -1.12164798  0.69431032]]
[[-0.61665631  0.71679298  0.82789654  0.3603433 ]]
[[-0.15983769]
 [ 1.31087797]
 [ 1.12307381]
 [-0.85756698]]
[[0.27139079]]
Iteration 1 - Loss = 5638943.191688167
```

Wow, that is a big loss value!?

```python
# Define and train neural network structure (Weird initialization)
n_x = 2
n_h = 4
n_y = 1
init_type = "Weird"
np.random.seed(37)
shallow_neural_net_weird = ShallowNeuralNet(n_x, n_h, n_y, init_type)
# Train and show final loss
shallow_neural_net_weird.train(inputs, outputs, N_max = 100, alpha = 1e-6, delta = 1e-6, display = True)
```

```
Iteration 26 - Loss = nan
- Gradients:
[[nan nan nan nan]
 [nan nan nan nan]]
[[nan nan nan nan]]
[[nan]
 [nan]
 [nan]
 [nan]]
[[nan]]
- Parameters:
[[nan nan nan nan]
 [nan nan nan nan]]
[[nan nan nan nan]]
[[nan]
 [nan]
 [nan]
 [nan]]
[[nan]]
```

And after iteration 26, we are getting NaNs values (Not a Number values) all over the place…?! These NaN values indicate that the numbers were so big that they could not be stored in memory. What is going on…?!

The exploding gradient problem

Definition (Exploding Gradients):

These **NaN** values are a typical symptom for a phenomenon called **exploding gradients**.

This typically occurs when the gradient descent rule has changes (in $\alpha \frac{\partial dL}{\partial W1}$ for instance) that are <u>far greater</u> than the current values in the matrices (e.g. W_1).

The values will explode to infinity and eventually become NaNs.

This typically happens with

- **unlucky initializations**,
- **when the learning rate α is too large for the given parameters**,
- and sometimes when **gradients are approximated with formulas that are a bit buggish**…

In order to avoid this problem, many techniques have been developed (something for later).

Unlucky initialization

Definition (Unlucky Initialization):

Unlucky Initialization is a typical example used to describe that a system's behaviour is impossible to predict due to the **random impact of its initial conditions**.

In some cases, the result of a training will be **drastically different** depending on the initial values used, which can lead to **vastly different outcomes for your model**.

In fact, in our previous “weird” initializer, we can resolve the exploding gradient/NaN problem… By just using a different seed for the initialization…!

Yup, we just had an **unlucky initialization**, as simple as that!

(BTW, that could be an empirical proof that the NFL theorem is indeed true!)

```python
# Define and train neural network structure (Weird initialization)
n_x = 2
n_h = 4
n_y = 1
init_type = "Weird"
np.random.seed(37)
shallow_neural_net_weird = ShallowNeuralNet(n_x, n_h, n_y, init_type)
# Train and show final loss
shallow_neural_net_weird.train(inputs, outputs, N_max = 100, alpha = 1e-6, delta = 1e-6, display = True)
```

```
Iteration 26 - Loss = nan
- Gradients:
[[nan nan nan nan]
 [nan nan nan nan]]
[[nan nan nan nan]]
[[nan]
 [nan]
 [nan]
 [nan]]
[[nan]]
- Parameters:
[[nan nan nan nan]
 [nan nan nan nan]]
[[nan nan nan nan]]
[[nan]
 [nan]
 [nan]
 [nan]]
[[nan]]
```

Seed 37 is an unlucky initialization…

```python
# Define and train neural network structure (Weird initialization)
n_x = 2
n_h = 4
n_y = 1
init_type = "Weird"
np.random.seed(17)
shallow_neural_net_weird2 = ShallowNeuralNet(n_x, n_h, n_y, init_type)
# Train and show final loss
shallow_neural_net_weird2.train(inputs, outputs, N_max = 500, alpha = 1e-6, delta = 1e-6, display = True)
```

```
- Gradients:
[[-9.86521735e-08 -7.36853902e-08 -8.11338502e-08  1.17729102e-07]
 [-3.95240319e-08 -2.95213335e-08 -3.25054864e-08  4.71670175e-08]]
[[-3.87197953e-08 -2.89206322e-08 -3.18440634e-08  4.62072611e-08]]
[[ 4.10683166e-07]
 [-1.03823853e-07]
 [ 3.91291750e-07]
 [ 7.42268171e-07]]
[[-1.96398395e-07]]
- Parameters:
[[-0.57336145  0.13297469 -1.0271704  -1.19171781]
 [ 0.65980609  0.57963967 -0.6216372   0.49082227]]
[[-1.30374448 -0.40183954  1.25928682 -1.24410091]]
[[ 0.19714925]
 [ 0.14725493]
 [ 0.16214014]
 [-0.23527311]]
[[0.14792844]]
Stopping early - loss evolution was less than beta on iteration 342.
```

Different seed…
Now that works?!

```python
# Showing final loss
print(shallow_neural_net_weird2.loss)
```

```
0.12422436714449628
```

The learning rate tradeoff

Definition (Learning rate trade-off):

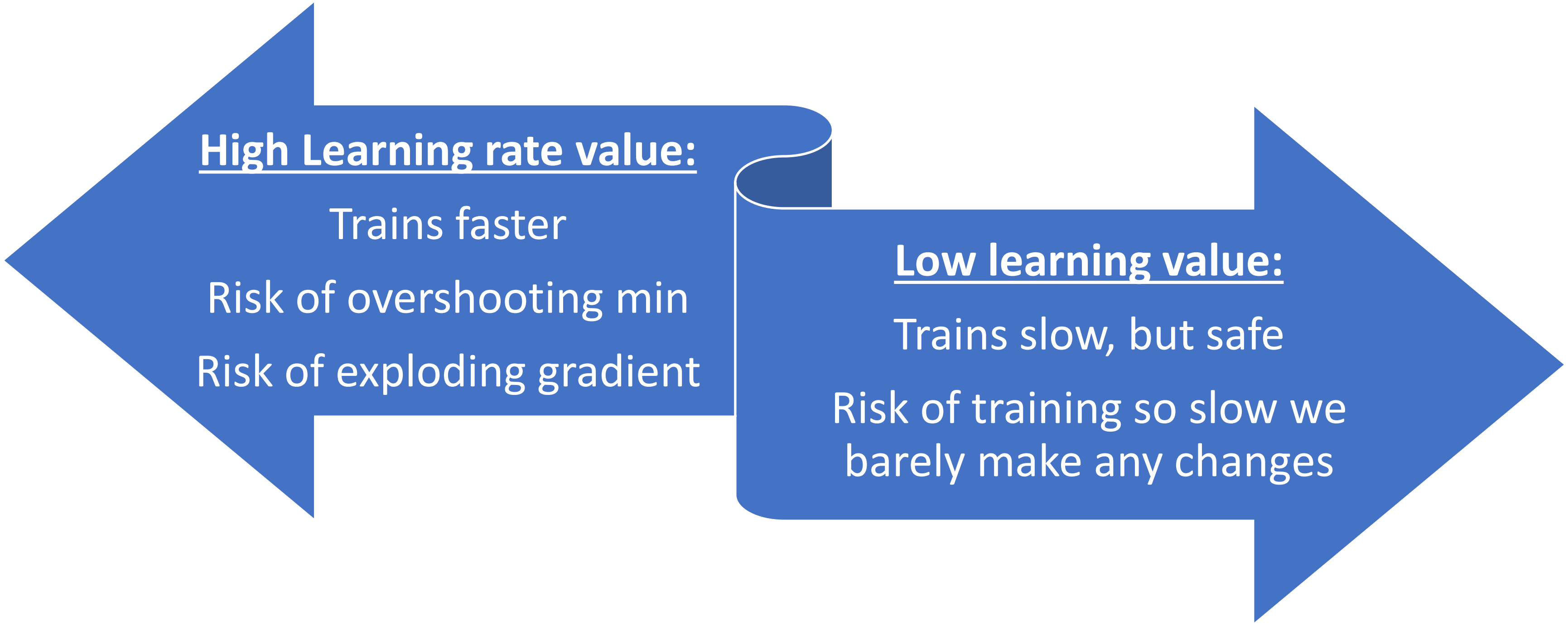
The **learning rate** is a **hyperparameter** which controls the step size taken by the algorithm in order to minimize the loss function during training.

- If the **learning rate is too small**, the algorithm will **take longer to converge**, but it can also prevent overshooting the global minimum. It might, however, lead to being stuck in non-optimum local minima.
- If the **learning rate is too high**, the algorithm **may converge faster**, but **it may lead to exploding gradients**.

The learning rate tradeoff

Definition (Learning rate trade-off):

There is therefore a **trade-off** on the deep learning rate, and as in all things with trade-offs, it is about finding the **optimal middle ground**.

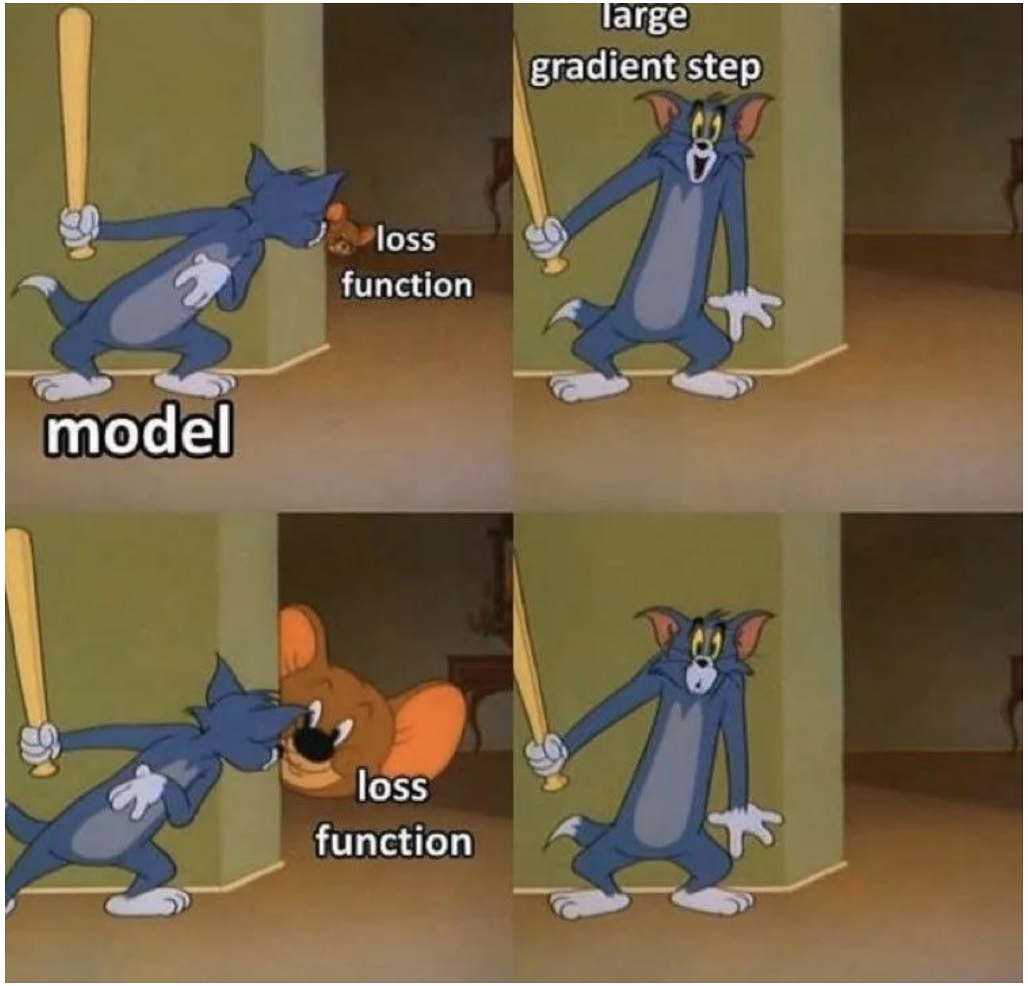


Triggering an exploding gradient on purpose

And, as a proof of that…

We can, in fact, **trigger an exploding gradient problem, on purpose, by using a very high learning rate value!**

Let us try a much higher value for the learning rate (e.g. using $1e^{-2}$ instead of $1e^{-6}$).



```
# Define and train neural network structure (random normal initialization)
n_x = 2
n_h = 4
n_y = 1
init_type = "Normal"
np.random.seed(37)
shallow_neural_net_normal1 = ShallowNeuralNet(n_x, n_h, n_y, init_type)
# Train and show final loss
shallow_neural_net_normal1.train(inputs, outputs, N_max = 1000, alpha = 1e-6, delta = 1e-6, display = True)
print(shallow_neural_net_normal1.loss)
```

```
- Gradients:
[[ 1.42356523e-05 -1.73577846e-06 -9.53259136e-07  1.24839272e-06]
 [ 1.32743203e-04 -1.61856153e-05 -8.88885652e-06  1.16408890e-05]]
[[ 1.27309997e-07 -1.55231348e-08 -8.52503385e-09  1.11644251e-08]]
[[ 3.86686371e-04]
 [ 2.71147622e-04]
 [ 8.05690592e-05]
 [-1.49929229e-04]]
[[2.45019007e-06]]
- Parameters:
[[-0.00544636  0.06743081  0.0346647  -0.13003462]
 [ 0.15185119  0.09898237  0.02776809 -0.04485894]]
[[ 0.09619662 -0.08275786  0.05346571  0.12283862]]
[[ 0.05195923]
 [-0.00633548]
 [-0.00347934]
 [ 0.00455655]]
[[0.14480251]]
Iteration 1 - Loss = 6.635227700991098
```

Normal initialization and low learning rate (1e-6)

This is fine…

```
# Show loss after training
print(shallow_neural_net_normal1.loss)
```

```
2.0907010654169245
```

```
# Define and train neural network structure (random normal initialization)
n_x = 2
n_h = 4
n_y = 1
init_type = "Normal"
np.random.seed(37)
shallow_neural_net_normal2 = ShallowNeuralNet(n_x, n_h, n_y, init_type)
# Train and show final loss
shallow_neural_net_normal2.train(inputs, outputs, N_max = 1000, alpha = 1e-2, delta = 1e-6, display = True)
print(shallow_neural_net_normal2.loss)
```

```
- Gradients:
[[ 0.14235652 -0.01735778 -0.00953259  0.01248393]
 [ 1.32743203 -0.16185615 -0.08888857  0.11640889]]
[[ 1.27309997e-03 -1.55231348e-04 -8.52503385e-05  1.11644251e-04]]
[[ 3.86686371]
 [ 2.71147622]
 [ 0.80569059]
 [-1.49929229]]
[[0.0245019]]
- Parameters:
[[-0.00544636  0.06743081  0.0346647  -0.13003462]
 [ 0.15185119  0.09898237  0.02776809 -0.04485894]]
[[ 0.09619662 -0.08275786  0.05346571  0.12283862]]
[[ 0.05195923]
 [-0.00633548]
 [-0.00347934]
 [ 0.00455655]]
[[0.14480251]]
Iteration 1 - Loss = 4354741.971946698
```

Learning rate too high leads to exploding gradients!

```
# Show loss after training
print(shallow_neural_net_normal2.loss)
```

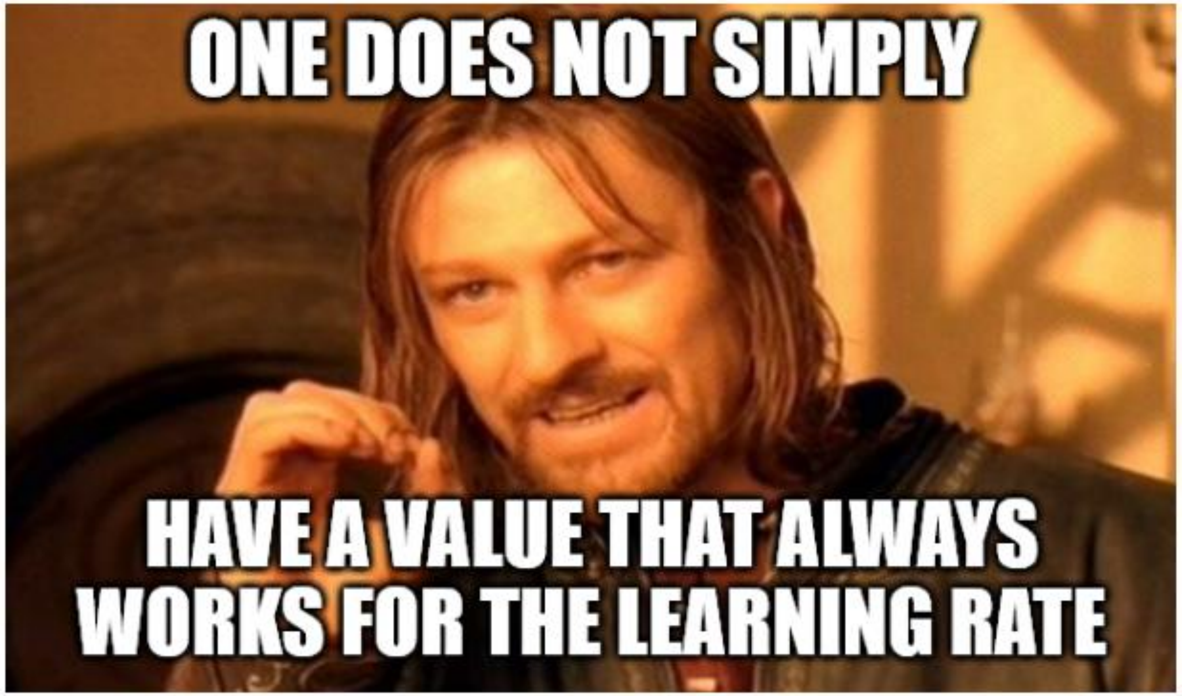
```
nan
```

No free lunch theorem, again

Definition (the No Free Lunch (or NFL) theorem, again):

The **no free lunch (NFL) theorem** also implies that there is **no such thing as "one value that works for all models" when it comes to hyperparameters such as the learning rate.**

No choice (again!): try some values and see how it goes!



About hyperparameters tuning

Definition (Hyperparameters and Hyperparameter tuning):

As we have seen, the **learning rate** is a **hyperparameter** that needs to be carefully chosen.

In fact, an important part of the practice of deep learning is **hyperparameter tuning.**

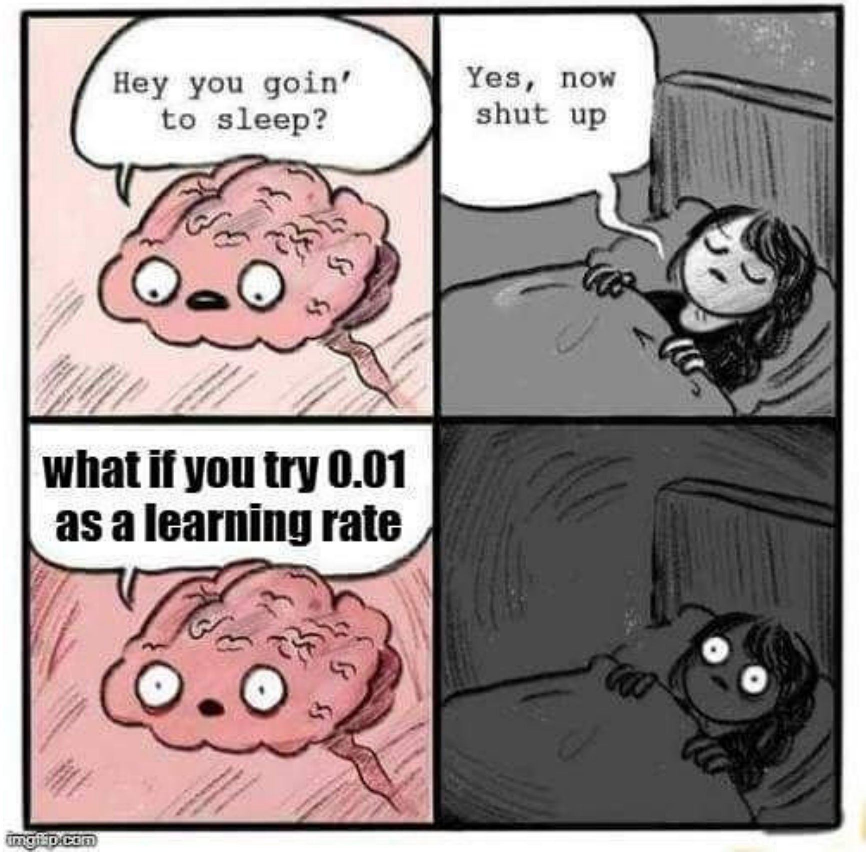
It consists, for instance, of **finding the optimal learning rate** that will allow the algorithm to **converge quickly and reliably**.

For a given model, there could be multiple **hyperparameters**:

- for instance, the **lambda** value in regularization is a hyperparameter…
- the **number of neurons** used in a given layer,
- the **number of layers** used in the neural network,
- the **initialization method for parameters**,
- Etc.

About hyperparameters tuning

Sadly, that is sometimes all it takes to make the difference between a model that successfully trains and a model that does not!



Controlling gradients to prevent explosions

Adjusting and trying different values for the learning rate is often good enough to prevent exploding gradients.

There are also other options, like:

- **Downscaling initial parameters values:** Divide initial parameter values by a certain factor to prevent large initial values on parameters.
- **Gradient clipping:** Set a max value for change on a single iteration and truncate any change that goes above that (to be discussed later).
- **Time evolution on learning rate:** Make the learning rate value change over time, make it large at the beginning and progressively decrease it over iterations (to be discussed later).

```python
# Define and train neural network structure (Weird initialization)
n_x = 2
n_h = 4
n_y = 1
init_type = "Weird"
np.random.seed(37)
shallow_neural_net_weird = ShallowNeuralNet(n_x, n_h, n_y, init_type)
# Train and show final loss
shallow_neural_net_weird.train(inputs, outputs, N_max = 100, alpha = 1e-6, delta = 1e-6, display = True)
```

```
Iteration 26 - Loss = nan
- Gradients:
[[nan nan nan nan]
 [nan nan nan nan]]
[[nan nan nan nan]]
[[nan]
 [nan]
 [nan]
 [nan]]
[[nan]]
- Parameters:
[[nan nan nan nan]
 [nan nan nan nan]]
[[nan nan nan nan]]
[[nan]
 [nan]
 [nan]
 [nan]]
[[nan]]
```

Back to our first exploding gradient example…

```python
# Define and train neural network structure (Weird initialization = some sort of a Uniform Xavier)
n_x = 2
n_h = 4
n_y = 1
init_type = "Weird"
np.random.seed(37)
shallow_neural_net_xavier2 = ShallowNeuralNet(n_x, n_h, n_y, init_type)
# Divide initial values by 10!
shallow_neural_net_xavier2.W1 /= 100
shallow_neural_net_xavier2.b1 /= 100
shallow_neural_net_xavier2.W2 /= 100
shallow_neural_net_xavier2.b2 /= 100
# Train and show final loss
shallow_neural_net_xavier2.train(inputs, outputs, N_max = 1000, alpha = 1e-6, delta = 1e-6, display = True)
print(shallow_neural_net_xavier2.loss)
```

```
- Gradients:
[[ 1.08643744e-06 -8.91020716e-06 -7.63367797e-06  5.82899367e-06]
 [ 4.17432348e-06 -3.42349093e-05 -2.93302128e-05  2.23962323e-05]]
[[ 8.17657760e-09 -6.70586248e-08 -5.74514080e-08  4.38692717e-08]]
[[-1.73842740e-05]
 [-1.29545695e-05]
 [ 3.51566967e-05]
 [-1.97254965e-05]]
[[-5.1155505e-06]]
- Parameters:
[[ 0.01257226 -0.00101546 -0.00868907  0.00231634]
 [ 0.00339649  0.00521064 -0.01121648  0.0069431 ]]
[[-0.00616656  0.00716793  0.00827897  0.00360343]]
[[-0.00159838]
 [ 0.01310878]
 [ 0.01123074]
 [-0.00857567]]
[[0.00271391]]
Iteration 1 - Loss = 7.036477890046345
```

Here, downscaling parameters during initialization fixed the exploding gradients issue, but it leads to slower training.

```python
# Show loss after training
print(shallow_neural_net_xavier2.loss)
```

```
2.2689392753336914
```

The vanishing gradient problem

Definition (Vanishing Gradients):

We just observed **exploding gradients**. Its counterpart exists and is called **vanishing gradients**.

This typically occurs when the gradient descent rule has changes (in $\alpha \frac{\partial dL}{\partial W1}$ for instance) that are <u>far smaller</u> than the current values in the matrices (e.g. W_1).

It might also happen for other reasons (to be discussed later).

- We can typically force the apparition of a vanishing gradient problem by forcing the parameters to be initialized as very small values (or even zeroes).
- We can also force the apparition of a vanishing gradient problem by using a very small learning rate α.
- We can then observe that most parameters remain unchanged during training as the changes are close, or even equal to, zero.

```python
# Define and train neural network structure (random normal initialization)
n_x = 2
n_h = 4
n_y = 1
init_type = "Normal"
np.random.seed(37)
shallow_neural_net_normal = ShallowNeuralNet(n_x, n_h, n_y, init_type)
# Train and show final loss
shallow_neural_net_normal.train(inputs, outputs, N_max = 100, alpha = 1e-6, delta = 1e-6, display = True)
print(shallow_neural_net_normal.loss)
```

```
- Gradients:
[[ 1.42356523e-05 -1.73577846e-06 -9.53259136e-07  1.24839272e-06]
 [ 1.32743203e-04 -1.61856153e-05 -8.88885652e-06  1.16408890e-05]]
[[ 1.27309997e-07 -1.55231348e-08 -8.52503385e-09  1.11644251e-08]]
[[ 3.86686371e-04]
 [ 2.71147622e-04]
 [ 8.05690592e-05]
 [-1.49929229e-04]]
[[2.45019007e-06]]
- Parameters:
[[-0.00544636  0.06743081  0.0346647  -0.13003462]
 [ 0.15185119  0.09898237  0.02776809 -0.04485894]]
[[ 0.09619662 -0.08275786  0.05346571  0.12283862]]
[[ 0.05195923]
 [-0.00633548]
 [-0.00347934]
 [ 0.00455655]]
[[0.14480251]]
Iteration 1 - Loss = 6.635227700991098
```

This is fine.

```python
# Show final loss
print(shallow_neural_net_normal.loss)
```

```
2.654115820074732
```

```
# Define and train neural network structure (Zero initialization)
n_x = 2
n_h = 4
n_y = 1
init_type = "Zero"
np.random.seed(37)
shallow_neural_net_zero = ShallowNeuralNet(n_x, n_h, n_y, init_type)
# Train and show final loss
shallow_neural_net_zero.train(inputs, outputs, N_max = 100, alpha = 1e-6, delta = 1e-6, display = True)
print(shallow_neural_net_zero.loss)
```

```
- Gradients:
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[0. 0. 0. 0.]]
[[0.]
 [0.]
 [0.]
 [0.]]
[[-4.9531058e-06]]
- Parameters:
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[0. 0. 0. 0.]]
[[0.]
 [0.]
 [0.]
 [0.]]
[[0.]]
Iteration 1 - Loss = 6.641131615309766
```

Initializing parameters as zeroes is seriously problematic and the model just does not train.

Why? Look at your Gradient Descent update rules and see what happens if W1, Z2, b1 and b2 are zeroes?

$$\boxed{W_2 \leftarrow W_2 - \frac{2\epsilon\alpha}{M}(W_1 X + b_1)}$$

```
# Show final loss
print(shallow_neural_net_zero.loss)
```

```
6.638703310983079
```

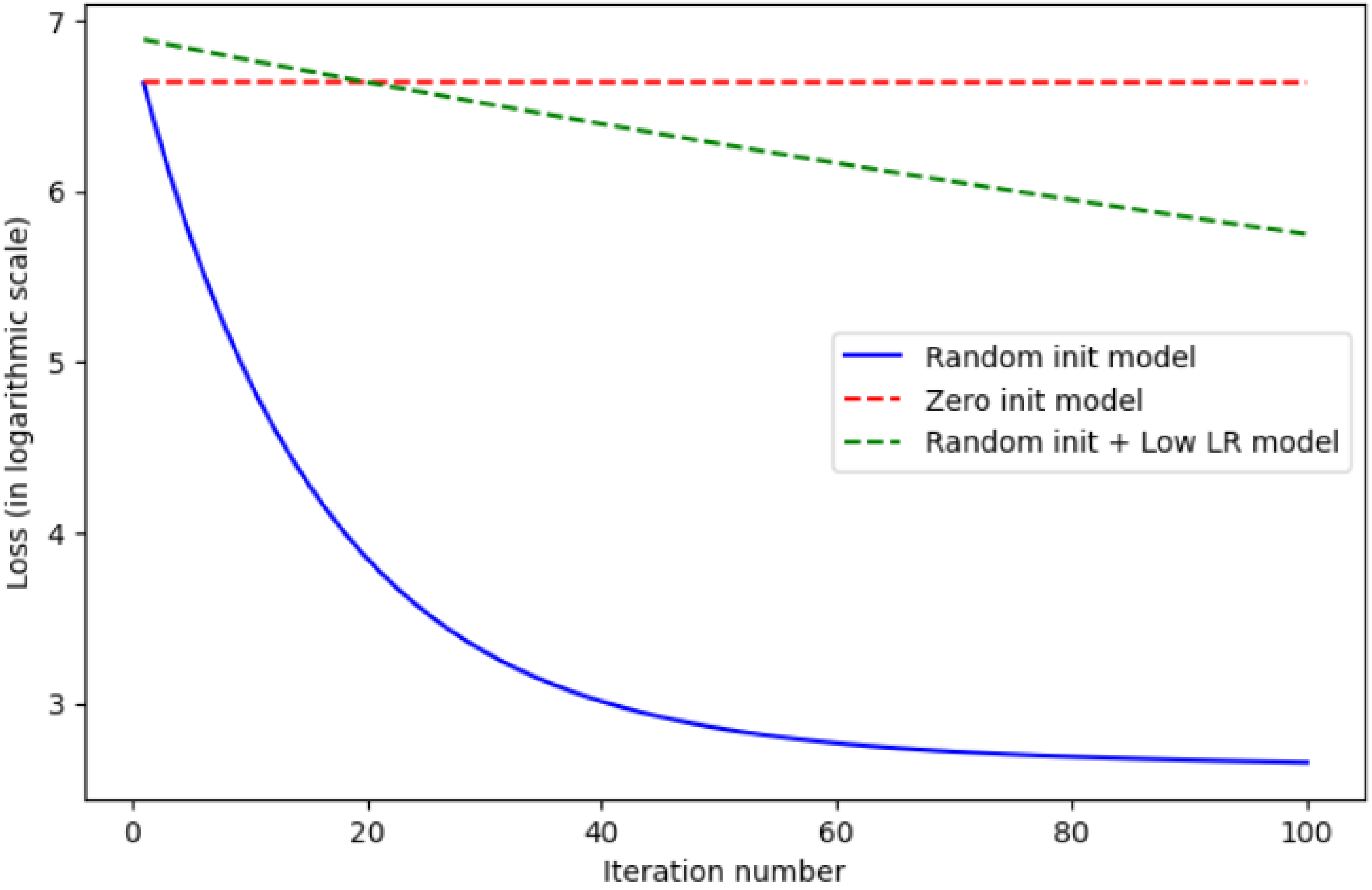
```python
# Define and train neural network structure
# (random normal initialization,
# but very low learning rate alpha)
n_x = 2
n_h = 4
n_y = 1
init_type = "Normal"
np.random.seed(37)
shallow_neural_net_normal_zerolr = ShallowNeuralNet(n_x, n_h, n_y, init_type)
# Train and show final loss
shallow_neural_net_normal_zerolr.train(inputs, outputs, N_max = 100, alpha = 5e-8, delta = 1e-6, display = True)
print(shallow_neural_net_normal_zerolr.loss)
```

```
- Gradients:
[[ 7.11782617e-07 -8.67889228e-08 -4.76629568e-08  6.24196361e-08]
 [ 6.63716016e-06 -8.09280764e-07 -4.44442826e-07  5.82044450e-07]]
[[ 6.36549987e-09 -7.76156741e-10 -4.26251693e-10  5.58221254e-10]]
[[ 1.93343185e-05]
 [ 1.35573811e-05]
 [ 4.02845296e-06]
 [-7.49646145e-06]]
[[1.22509503e-07]]
- Parameters:
[[-0.00544636  0.06743081  0.0346647  -0.13003462]
 [ 0.15185119  0.09898237  0.02776809 -0.04485894]]
[[ 0.09619662 -0.08275786  0.05346571  0.12283862]]
[[ 0.05195923]
 [-0.00633548]
 [-0.00347934]
 [ 0.00455655]]
[[0.14480251]]
Iteration 1 - Loss = 6.887598572349596
```

Setting up a learning rate that is too small might seriously hinder the training (a.k.a. vanishing gradients). And sometimes even lead to a “convergence” to wrong minimum.

```python
# Show final loss
print(shallow_neural_net_normal_zerolr.loss)
```

```
5.748091602996055
```



7
6
5
4
3
0
20
40
60
80
100
Loss (in logarithmic scale)
Iteration number
Random init model
Zero init model
Random init + Low LR model

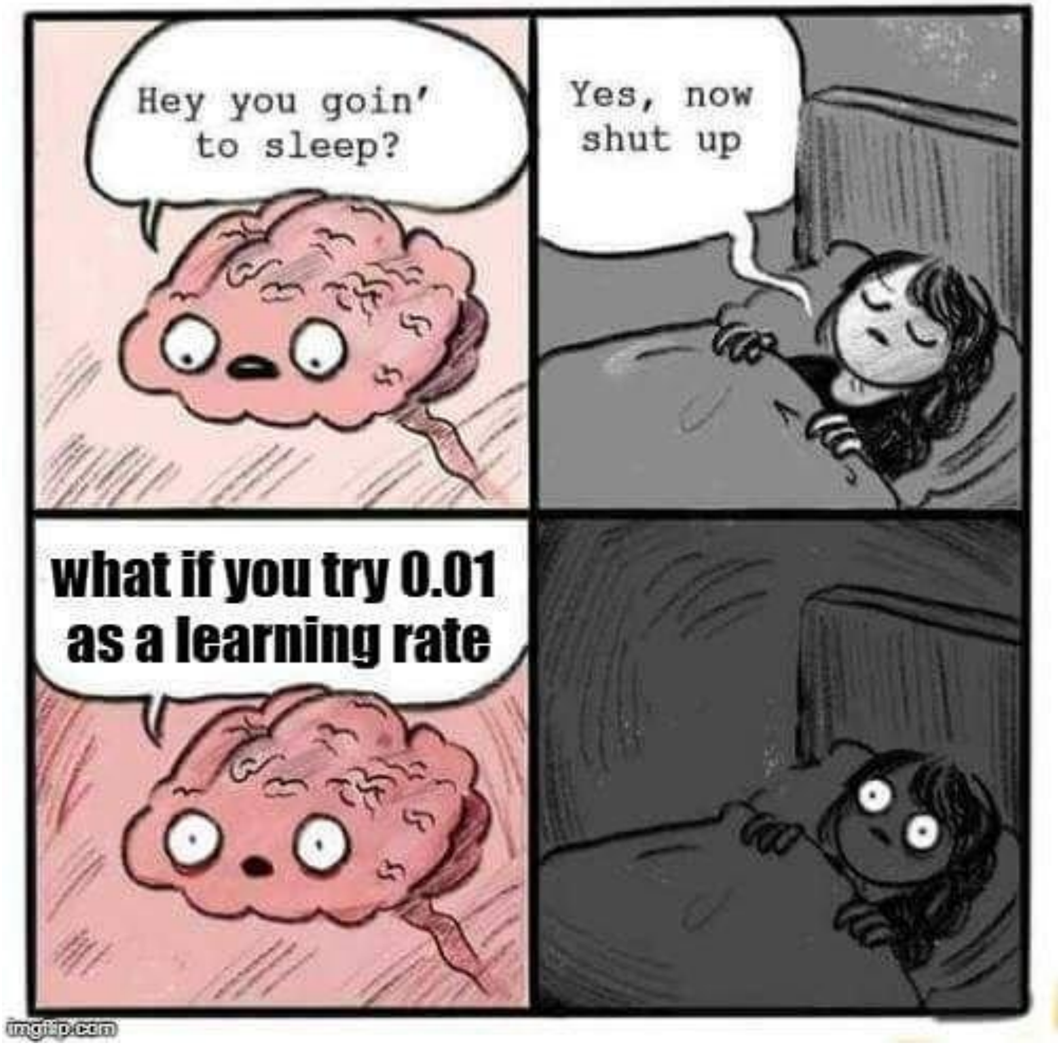
Remember, the learning rate tradeoff

Definition (Hyperparameters and hyperparameter tuning):

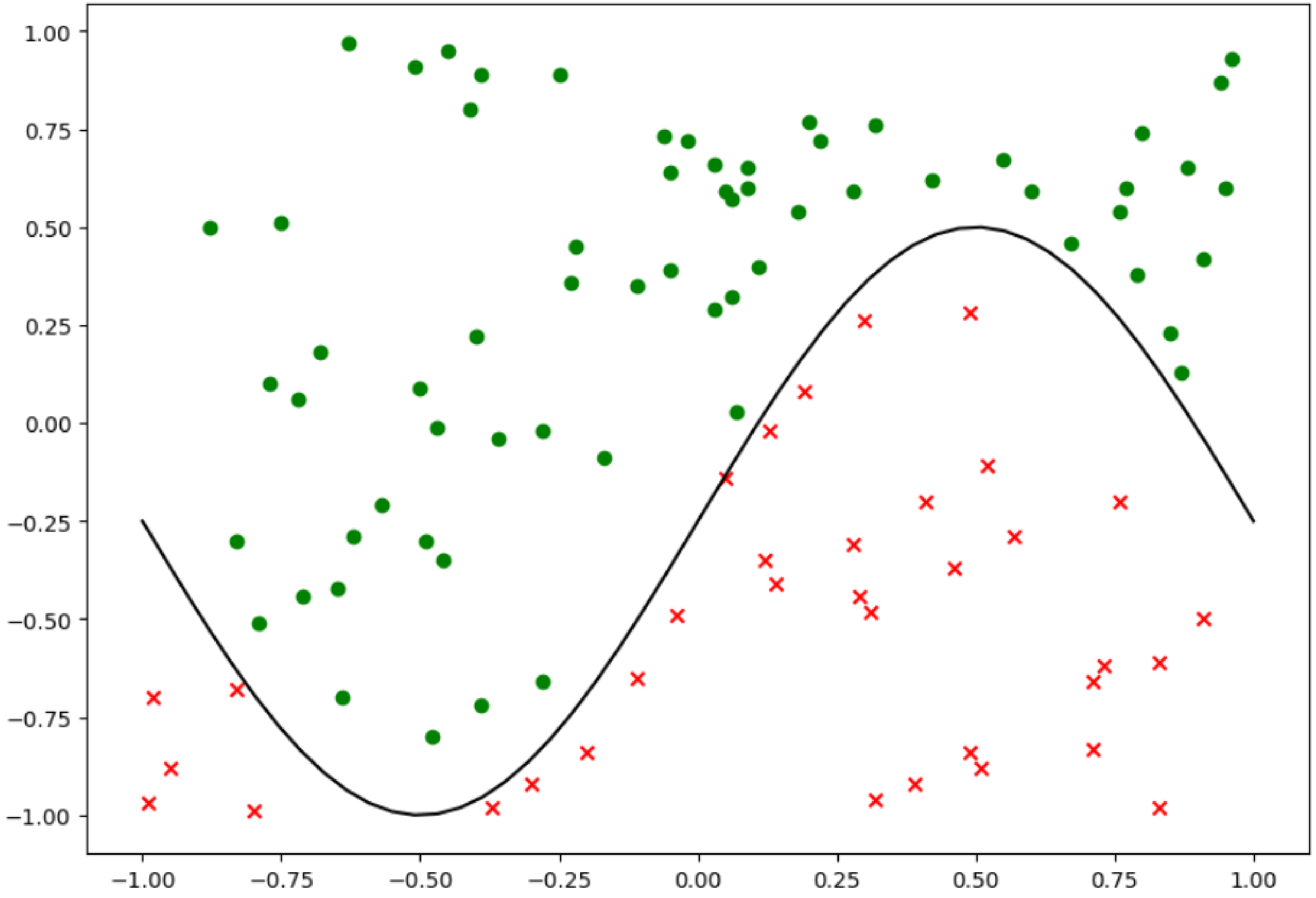
As we have seen, the **learning rate** is a **hyperparameter** that needs to be carefully chosen.

Thus, an important part of deep learning is **hyperparameter tuning.**

It consists, for instance, of **finding the optimal learning rate** α that will allow the gradient descent to **converge quickly and reliably**.



New dataset! (An abstract one this time)



New dataset! (An abstract one this time)

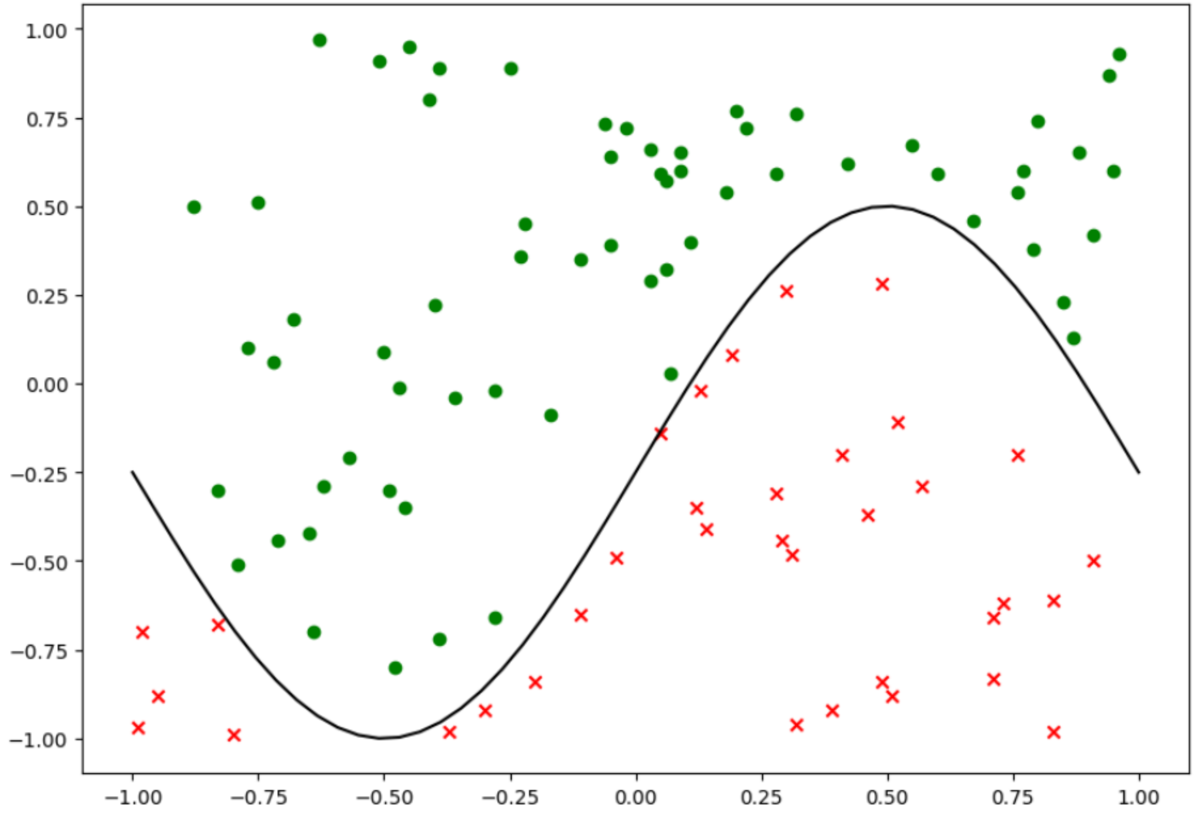
Each sample consists of two randomly drawn input values (x_1, x_2), with

$$-1 < x_1, x_2 < 1.$$

Black line serves as (non-linear) frontier between both classes, with equation.

$$f(x) = -\frac{1}{4} + \frac{3}{4}\sin(x\pi)$$

Two classes (green and red): Green (class 1) if $x_2 > f(x_1)$ and red (class 0) otherwise.



New dataset!

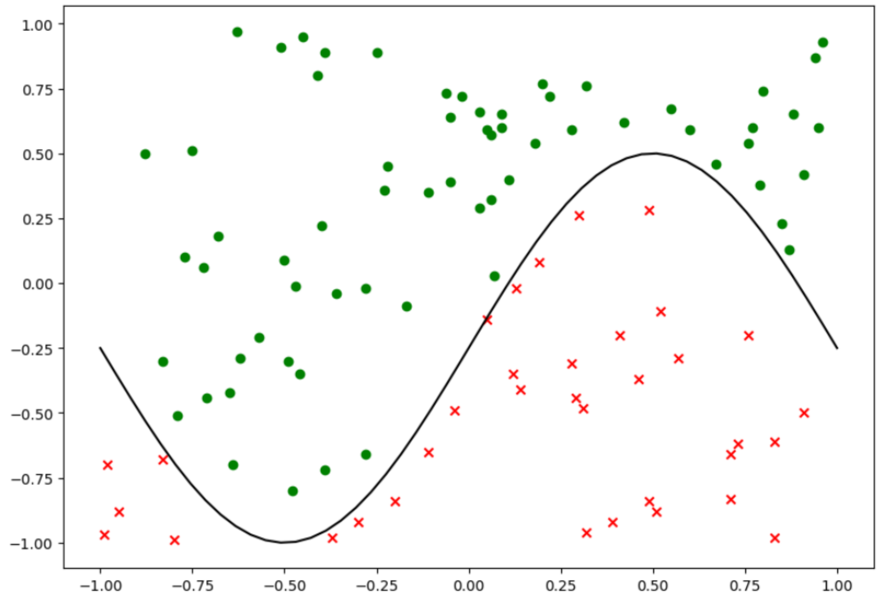
We can generate some entries using the helper functions here.

Basically, drawing pairs of values randomly and checking if above or below the sine line.

```python
# All helper functions
eps = 1e-5
min_val = -1 + eps
max_val = 1 - eps
def val(min_val, max_val):
    return round(np.random.uniform(min_val, max_val), 2)
def class_for_val(val1, val2):
    k = np.pi
    return int(val2 >= -1/4 + 3/4*np.sin(val1*k))
n_points = 100
def create_dataset(n_points, min_val, max_val):
    val1_list = np.array([val(min_val, max_val) for _ in range(n_points)])
    val2_list = np.array([val(min_val, max_val) for _ in range(n_points)])
    inputs = np.array([[v1, v2] for v1, v2 in zip(val1_list, val2_list)])
    outputs = np.array([class_for_val(v1, v2) for v1, v2 in zip(val1_list, val2_list)]).reshape(n_points, 1)
    return val1_list, val2_list, inputs, outputs
```

```python
# Generate dataset
np.random.seed(47)
val1_list, val2_list, inputs, outputs = create_dataset(n_points, min_val, max_val)
# Check a few entries of the dataset
print(val1_list.shape)
print(val2_list.shape)
print(inputs.shape)
print(outputs.shape)
print(inputs[0:5, :])
print(outputs[0:5])
```

```
(100,)
(100,)
(100, 2)
(100, 1)
[[-0.77  0.1 ]
 [ 0.95  0.6 ]
 [ 0.46 -0.37]
 [-0.3  -0.92]
 [ 0.42  0.62]]
[[1]
 [1]
 [0]
 [0]
 [1]]
```



Adjusting our shallow NN

We can adjust our shallow neural network to the task by:

- Using the same two layers as before.
- Simply adding a clipping to the final value of the forward method to ensure values between 0 and 1.
- Replace MSE with our cross-entropy loss function (as in Log. Reg.).
- Backward and trainer methods are left untouched.

```
class ShallowNeuralNet():

    def __init__(self, n_x, n_h, n_y):
        # Network dimensions
        self.n_x = n_x
        self.n_h = n_h
        self.n_y = n_y
        # Initialize parameters
        self.init_parameters_normal()
        # Loss, initialized as infinity before first calculation is made
        self.loss = float("Inf")

    def init_parameters_normal(self):
        # Weights and biases matrices (randomly initialized)
        self.W1 = np.random.randn(self.n_x, self.n_h)*0.1
        self.b1 = np.random.randn(1, self.n_h)*0.1
        self.W2 = np.random.randn(self.n_h, self.n_y)*0.1
        self.b2 = np.random.randn(1, self.n_y)*0.1

    def forward(self, inputs):
        # Wx + b operation for the first layer
        Z1 = np.matmul(inputs, self.W1)
        Z1_b = Z1 + self.b1
        # Wx + b operation for the second layer
        Z2 = np.matmul(Z1_b, self.W2)
        Z2_b = Z2 + self.b2
        #  Adding clipping to keep prediction values
        # between 0 and 1 (see loss function!)
        pred = np.clip(Z2_b, 0, 1)
        return pred

    def CE_loss(self, inputs, outputs):
        # Cross Entropy loss function as before
        outputs_re = outputs.reshape(-1, 1)
        pred = self.forward(inputs)
        eps = 1e-5
        losses = outputs*np.log(pred + eps) + (1 - outputs)*np.log(1 - pred + eps)
        self.loss = -np.sum(losses)/outputs.shape[0]
        return self.loss

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

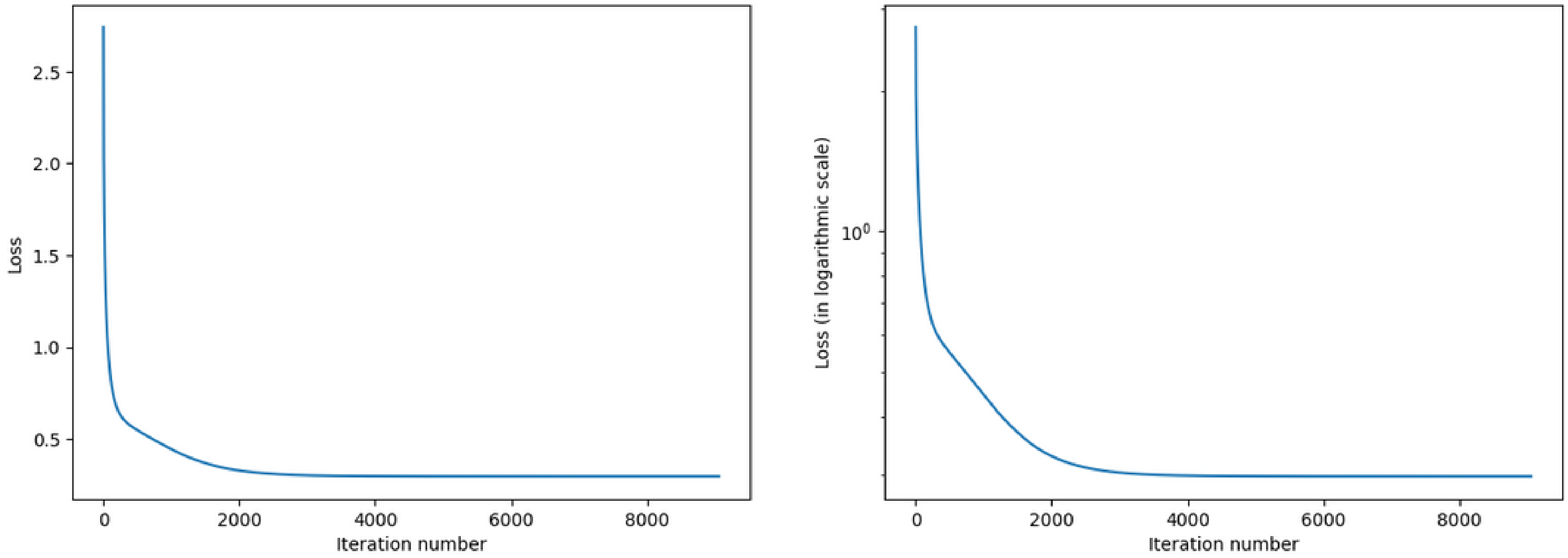
It trains just fine! (or does it?)

Training curves looking nice, but does that mean the model is good?

```
# Define and train neural network structure (no activation)
n_x = 2
n_h = 10
n_y = 1
np.random.seed(37)
shallow_neural_net = ShallowNeuralNet(n_x, n_h, n_y)
# Train and show final loss
shallow_neural_net.train(inputs, outputs, N_max = 10000, alpha = 5e-3, delta = 1e-8, display = False)
print(shallow_neural_net.loss)
```

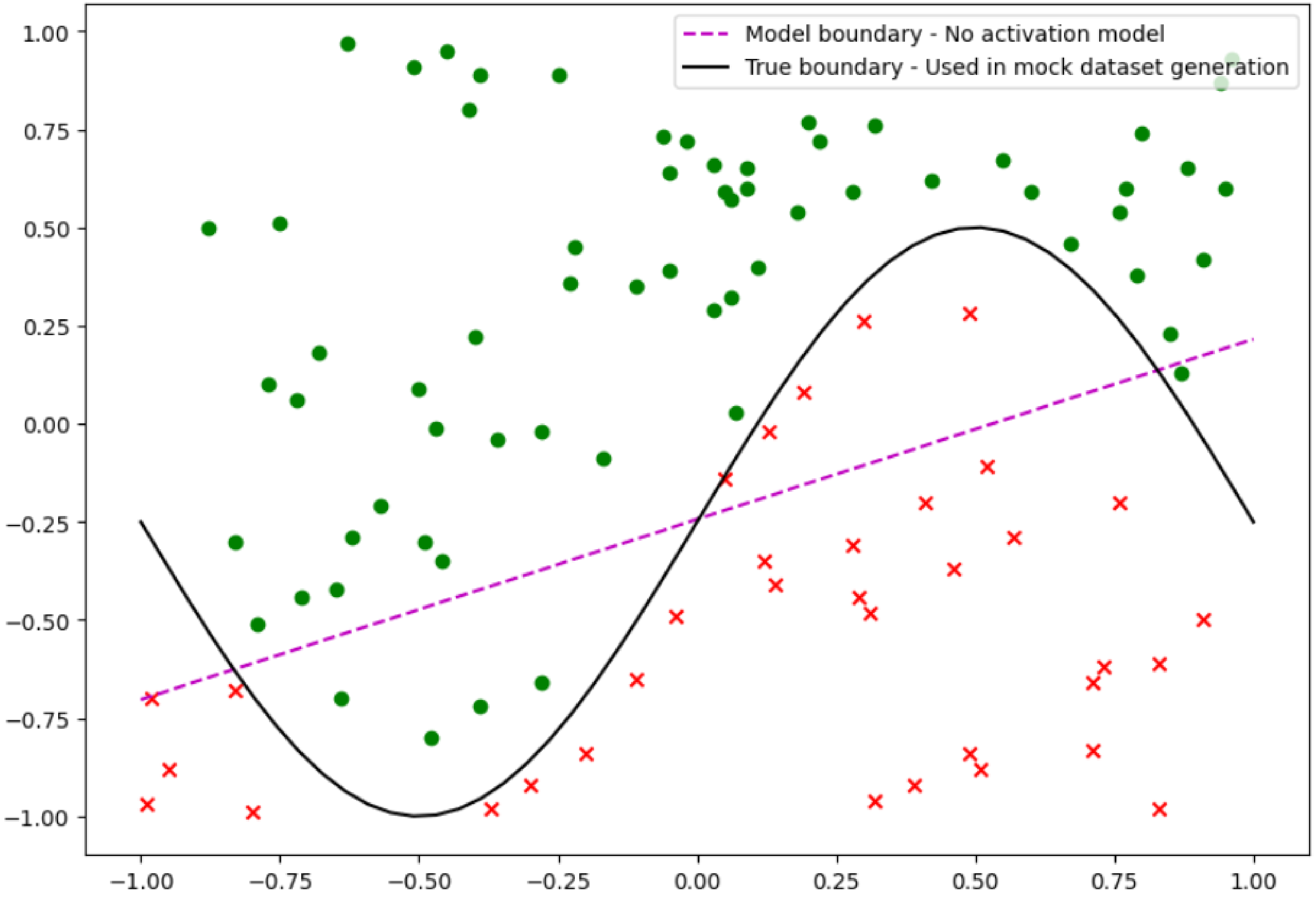
```
0.29783823594717523
```

```
shallow_neural_net.show_losses_over_training()
```



It trains just fine! (or does it?)

Errrm… NOPE!



Linear layers necessarily lead to linear boundary

As shown in Notebook,

- Our model consists of two linear layers, with two $WX + B$ operations in a row.
- The result is linear also, see the reorganised y_{pred} equation.
- Boundary for model is therefore linear in x_1 and x_2!

To demonstrate, let us consider that $n_x = 2$, $n_h = 4$ and $n_y = 1$. In this configuration, we have:

$$W_1 = \begin{pmatrix} w_{1,1}^{(1)}, & w_{1,2}^{(1)}, & w_{1,3}^{(1)}, & w_{1,4}^{(1)} \\ w_{2,1}^{(1)}, & w_{2,2}^{(1)}, & w_{2,3}^{(1)}, & w_{2,4}^{(1)} \end{pmatrix}$$

$$b_1 = \begin{pmatrix} b_{1,1}^{(1)}, & b_{1,2}^{(1)}, & b_{1,3}^{(1)}, & b_{1,4}^{(1)} \end{pmatrix}$$

$$W_2 = \begin{pmatrix} w_{1,1}^{(2)} \\ w_{1,2}^{(2)} \\ w_{1,3}^{(2)} \\ w_{1,4}^{(2)} \end{pmatrix}$$

$$b_2 = \begin{pmatrix} b_{1,1}^{(2)} \end{pmatrix}$$

After the first operation $Z_1 = XW_1 + b_1$, we have:

$$Z_1 = \begin{pmatrix} w_{1,1}^{(1)}x_1 + w_{2,1}^{(1)}x_2 + b_{1,1}^{(1)}, & w_{1,2}^{(1)}x_1 + w_{2,2}^{(1)}x_2 + b_{1,2}^{(1)}, & w_{1,3}^{(1)}x_1 + w_{2,3}^{(1)}x_2 + b_{1,3}^{(1)}, & w_{1,4}^{(1)}x_1 + w_{2,4}^{(1)}x_2 + b_{1,4}^{(1)} \end{pmatrix}$$

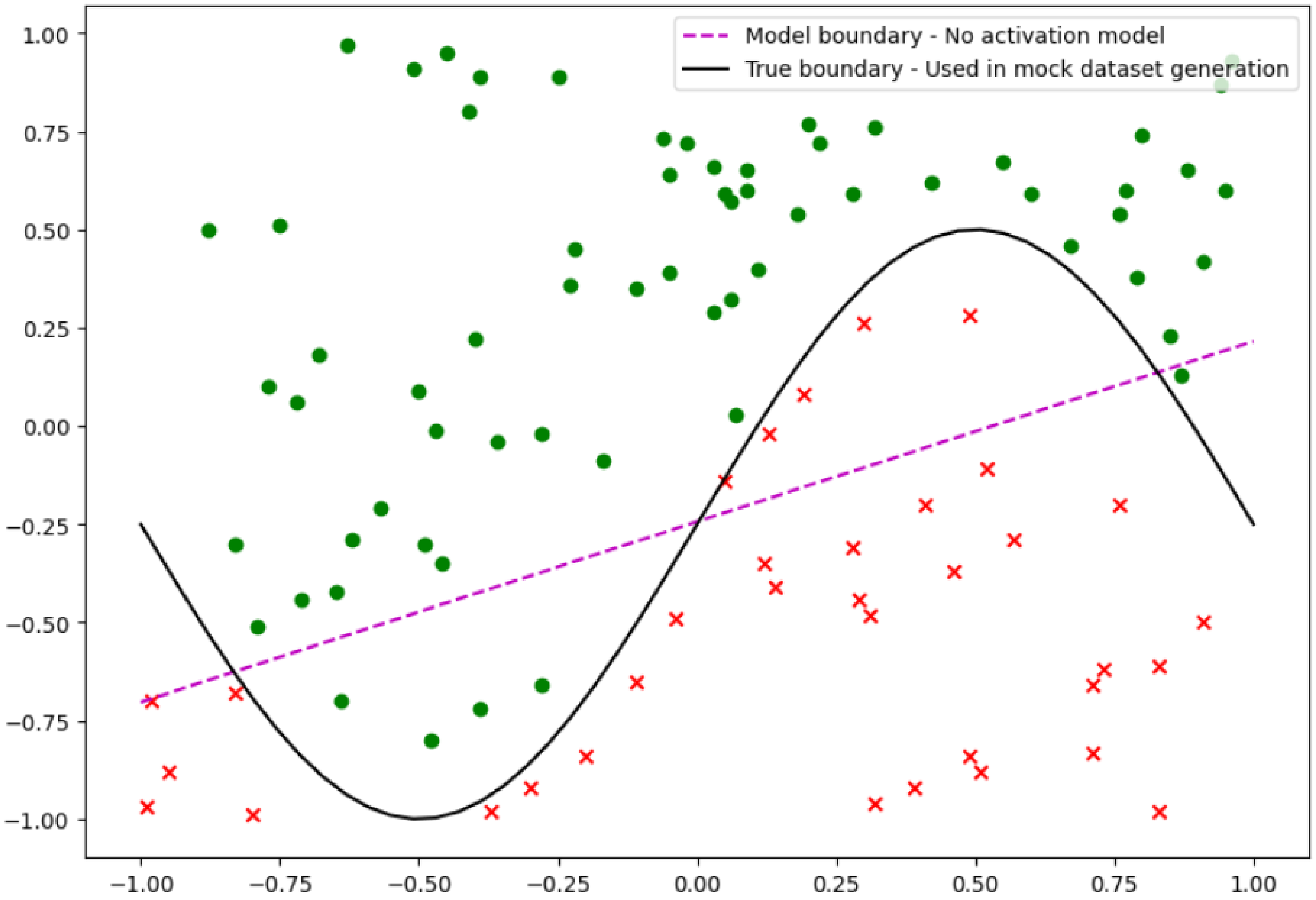
After the second operation, the predicted price y_{pred} is given by $y_{pred} = Z_1 W_2 + b_2$

$$y_{pred} = \left(\sum_{k=1}^{4} w_{1,k}^{(1)} w_{1,k}^{(2)} \right) x_1 + \left(\sum_{k=1}^{4} w_{2,k}^{(1)} w_{1,k}^{(2)} \right) x_2 + \left(\sum_{k=1}^{4} b_{1,k}^{(1)} w_{1,k}^{(2)} \right) + b_{1,1}^{(2)}$$

Trains just fine! (or does it?)

Well, that is going to be a problem, because the black boundary is definitely not linear…

Or, in other words, the data is **not linearly separable.**



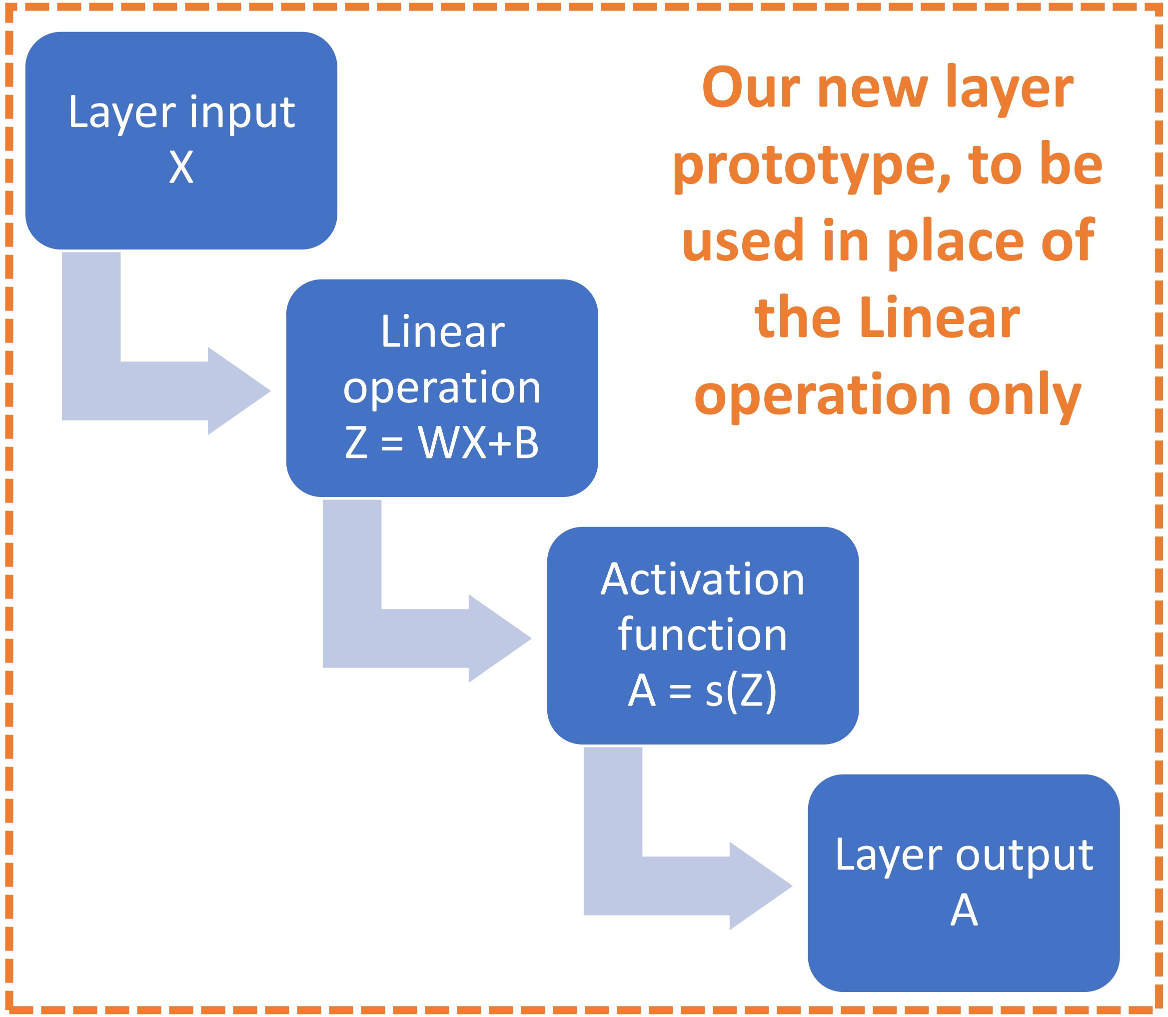
Activation functions and introducing non-linearity in the Neural Networks

Definition (Activation functions and non-linearity):

Activation functions are an important component of neural networks because they **introduce non-linearity** to the model.

This is important because most real-world data is non-linear.

Simply done by **adding an extra operation** after the linear one, for instance **our sigmoid function**.



Adding sigmoid activations

We can update our model by:

- Adding a sigmoid method, implementing the sigmoid operation from earlier.
- Adding sigmoid operations after each linear operation in the forward method.
- Our cross-entropy loss method remains unchanged, as it will simply use the updated forward procedure.

```python
class ShallowNeuralNet_WithAct():

    def __init__(self, n_x, n_h, n_y):
        # Network dimensions
        self.n_x = n_x
        self.n_h = n_h
        self.n_y = n_y
        # Initialize parameters
        self.init_parameters_normal()
        # Loss, initialized as infinity before first calculation is made
        self.loss = float("Inf")

    def init_parameters_normal(self):
        # Weights and biases matrices (randomly initialized)
        self.W1 = np.random.randn(self.n_x, self.n_h)*0.1
        self.b1 = np.random.randn(1, self.n_h)*0.1
        self.W2 = np.random.randn(self.n_h, self.n_y)*0.1
        self.b2 = np.random.randn(1, self.n_y)*0.1

    def sigmoid(self, val):
        return 1/(1 + np.exp(-val))

    def forward(self, inputs):
        # Wx + b operation for the first layer
        Z1 = np.matmul(inputs, self.W1)
        Z1_b = Z1 + self.b1
        A1 = self.sigmoid(Z1_b)
        # Wx + b operation for the second layer
        Z2 = np.matmul(A1, self.W2)
        Z2_b = Z2 + self.b2
        y_pred = self.sigmoid(Z2_b)
        return y_pred

    def CE_loss(self, inputs, outputs):
```

Adding sigmoid activations

Careful now…

- **As the forward procedure has changed, we would have to update the backward propagation…**

```
class ShallowNeuralNet_WithAct():

    def __init__(self, n_x, n_h, n_y):
        # Network dimensions
        self.n_x = n_x
        self.n_h = n_h
        self.n_y = n_y
        # Initialize parameters
        self.init_parameters_normal()
        # Loss, initialized as infinity before first calculation is made
        self.loss = float("Inf")

    def init_parameters_normal(self):
        # Weights and biases matrices (randomly initialized)
        self.W1 = np.random.randn(self.n_x, self.n_h)*0.1
        self.b1 = np.random.randn(1, self.n_h)*0.1
        self.W2 = np.random.randn(self.n_h, self.n_y)*0.1
        self.b2 = np.random.randn(1, self.n_y)*0.1

    def sigmoid(self, val):
        return 1/(1 + np.exp(-val))

    def forward(self, inputs):
        # Wx + b operation for the first layer
        Z1 = np.matmul(inputs, self.W1)
        Z1_b = Z1 + self.b1
        A1 = self.sigmoid(Z1_b)
        # Wx + b operation for the second layer
        Z2 = np.matmul(A1, self.W2)
        Z2_b = Z2 + self.b2
        y_pred = self.sigmoid(Z2_b)
        return y_pred
```

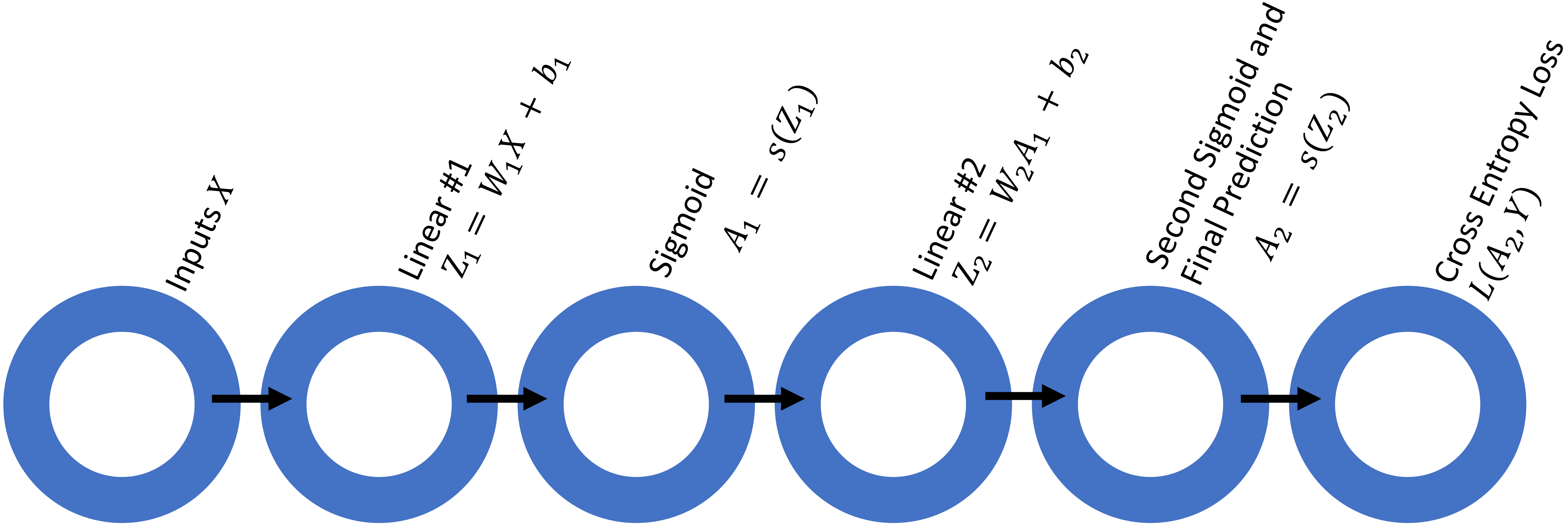
Adding sigmoid activations

Careful now…

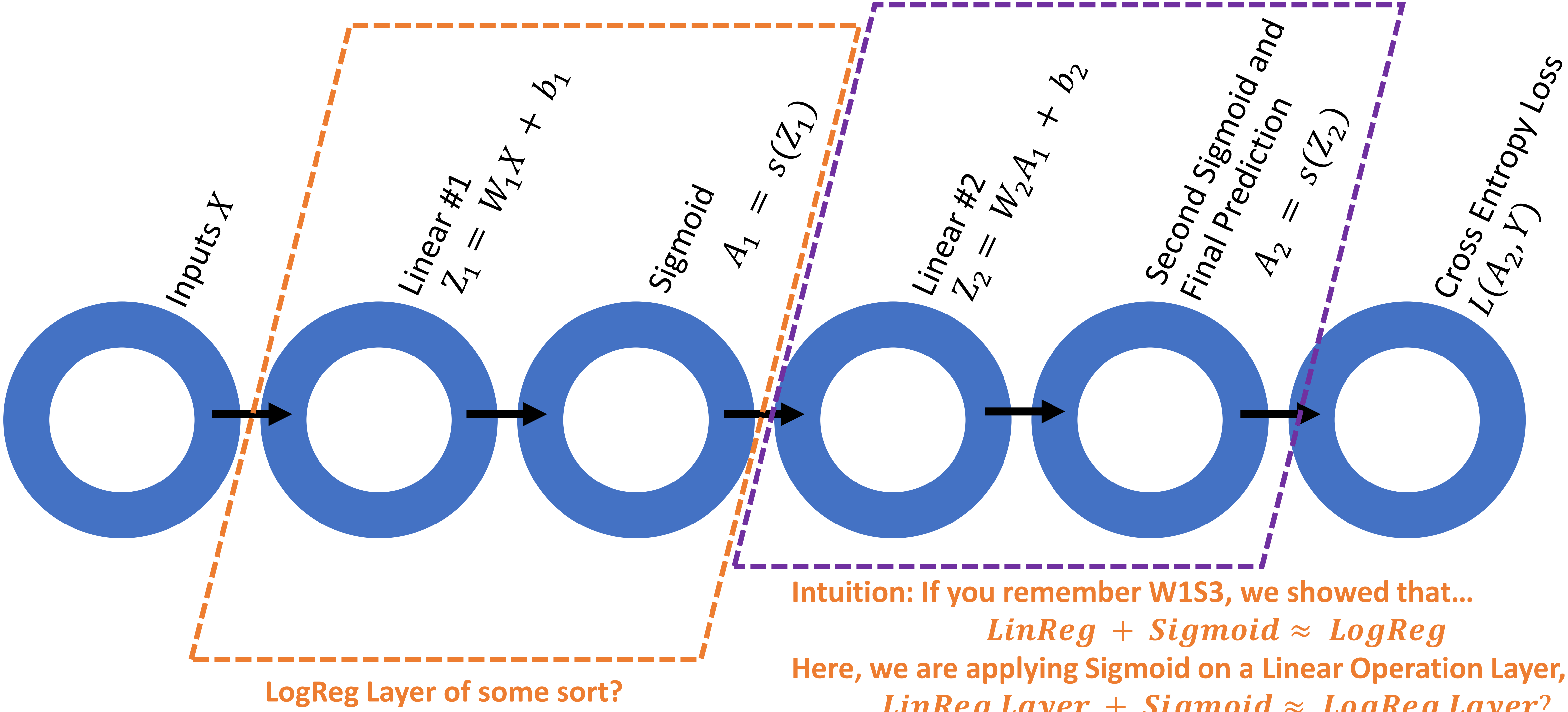
- **As the forward procedure has changed, we would have to update the backward propagation…**
- This means we have to go back to using the chain rule manual differentiation procedure, account for the new sigmoid operations, and get the new update rules to use in the backward method…

When you realize you have to re-do the entire backpropagation calculation and implementation to account for a small change of an activation function

Step 1: Drawing a computation graph



Step 1: Drawing a computation graph



Intuition: If you remember W1S3, we showed that…

$$LinReg + Sigmoid \approx LogReg$$

Here, we are applying Sigmoid on a Linear Operation Layer, so…

$$LinReg\ Layer + Sigmoid \approx LogReg\ Layer?$$

Step 2: Recall the forward equations

Our forward method gives:

$$Z_1 = W_1 X + b_1$$

$$A_1 = s(Z_1)$$

$$Z_2 = W_2 A_1 + b_2$$

$$A_2 = s(Z_2)$$

$$L = \frac{-1}{N} \sum_{i}^{N} Y\, ln(A_2) + (1 - Y) ln(1 - A_2)$$

Step 3: Use the chain rule to backpropagate

Retrieving the gradient descent update rules takes a few steps and requires some organizing…

- First, recall that our loss function is defined as

$$L = \frac{-1}{N}\sum_{i}^{N} Y\, ln(A_2) + (1-Y) ln(1-A_2)$$

Therefore, we have

$$\frac{\partial L}{\partial A_2} = \frac{1}{N}\left(-\frac{Y}{A_2} + \frac{1-Y}{1-A_2}\right)$$

Step 3: Use the chain rule to backpropagate

- Then, recall that

$$A_2 = s(Z_2)$$

We can then prove that

$$s'(X) = s(X)\big(1 - s(X)\big)$$

Using the chain rule, we then obtain

$$\frac{\partial L}{\partial Z_2} = \frac{\partial L}{\partial A_2}\frac{\partial A_2}{\partial Z_2} = \frac{\partial L}{\partial A_2} A_2(1 - A_2)$$

Step 3: Use the chain rule to backpropagate

- Let us continue

$$Z_2 = W_2 A_1 + b_2$$

Using the chain rule, we then obtain

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial Z_2} \frac{\partial Z_2}{\partial W_2} = \frac{\partial L}{\partial Z_2} A_1$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial Z_2} \frac{\partial Z_2}{\partial b_2} = \frac{\partial L}{\partial Z_2}$$

Step 3: Use the chain rule to backpropagate

- Again…

$$Z_2 = W_2 A_1 + b_2$$

Using the chain rule and our sigmoid derivative, we then obtain

$$\frac{\partial L}{\partial A_1} = \frac{\partial L}{\partial Z_2} \frac{\partial Z_2}{\partial A_1} = \frac{\partial L}{\partial Z_2} W_2$$

Step 3: Use the chain rule to backpropagate

- Almost there…

$$Z_2 = W_2 A_1 + b_2$$
$$A_1 = s(Z_1)$$

Using the chain rule and our sigmoid derivative, we then obtain

$$\frac{\partial L}{\partial A_1} = \frac{\partial L}{\partial Z_2}\frac{\partial Z_2}{\partial A_1} = \frac{\partial L}{\partial Z_2} W_2$$

$$\frac{\partial L}{\partial Z_1} = \frac{\partial L}{\partial A_1}\frac{\partial A_1}{\partial Z_1} = \frac{\partial L}{\partial A_1} A_1(1 - A_1)$$

Step 3: Use the chain rule to backpropagate

- Finally…!

$$Z_1 = W_1 X + b_1$$

Using the chain rule and our sigmoid derivative, we then obtain

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial Z_1}\frac{\partial Z_1}{\partial W_1} = \frac{\partial L}{\partial Z_1}X$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial Z_1}\frac{\partial Z_1}{\partial b_1} = \frac{\partial L}{\partial Z_1}$$

Backward Method Update

Reusing our derivatives formulas <u>carefully</u> gives the GD update rules to use for the backward method.

- Notice how we did not rush and computed each term, one operation at a time, to avoid making mistakes.
- Take it slow!

```
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)

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

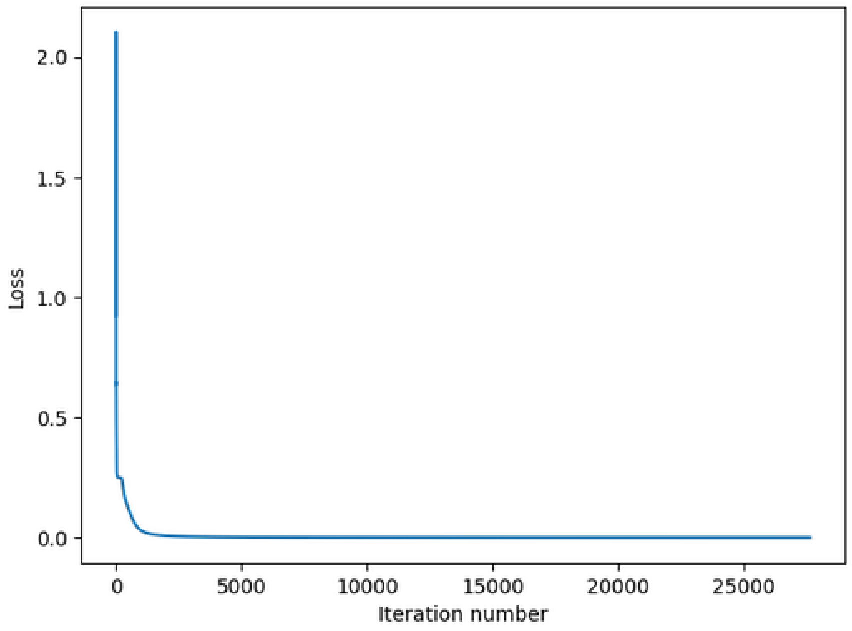
Restricted

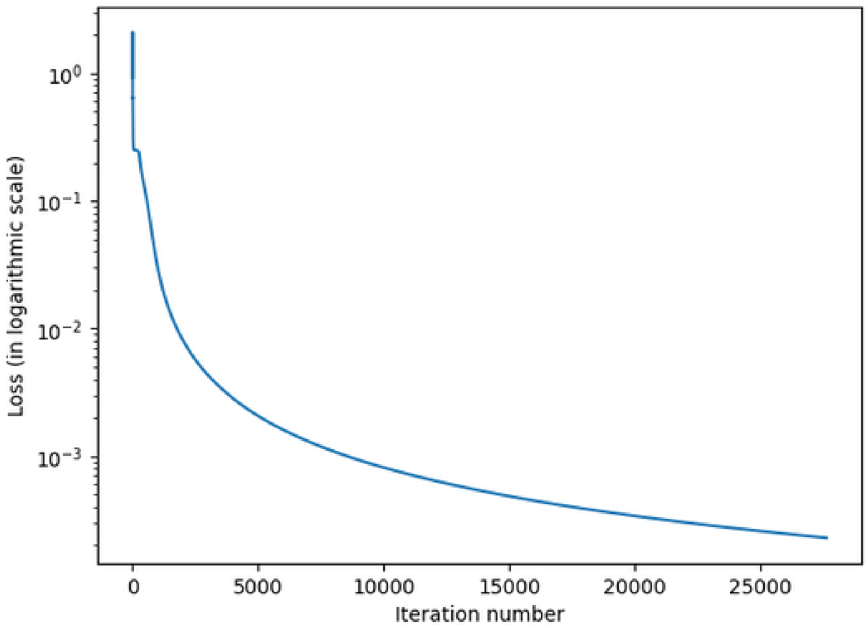
Trains just fine! (Much lower loss BTW!)

```
# Define and train neural network structure (with activation)
n_x = 2
n_h = 10
n_y = 1
np.random.seed(37)
shallow_neural_net_act = ShallowNeuralNet_WithAct(n_x, n_h, n_y)
# Train and show final loss
shallow_neural_net_act.train(inputs, outputs, N_max = 100000, alpha = 5, delta = 1e-8, display = False)
print(shallow_neural_net_act.loss)
```

```
0.00022943092333062472
```

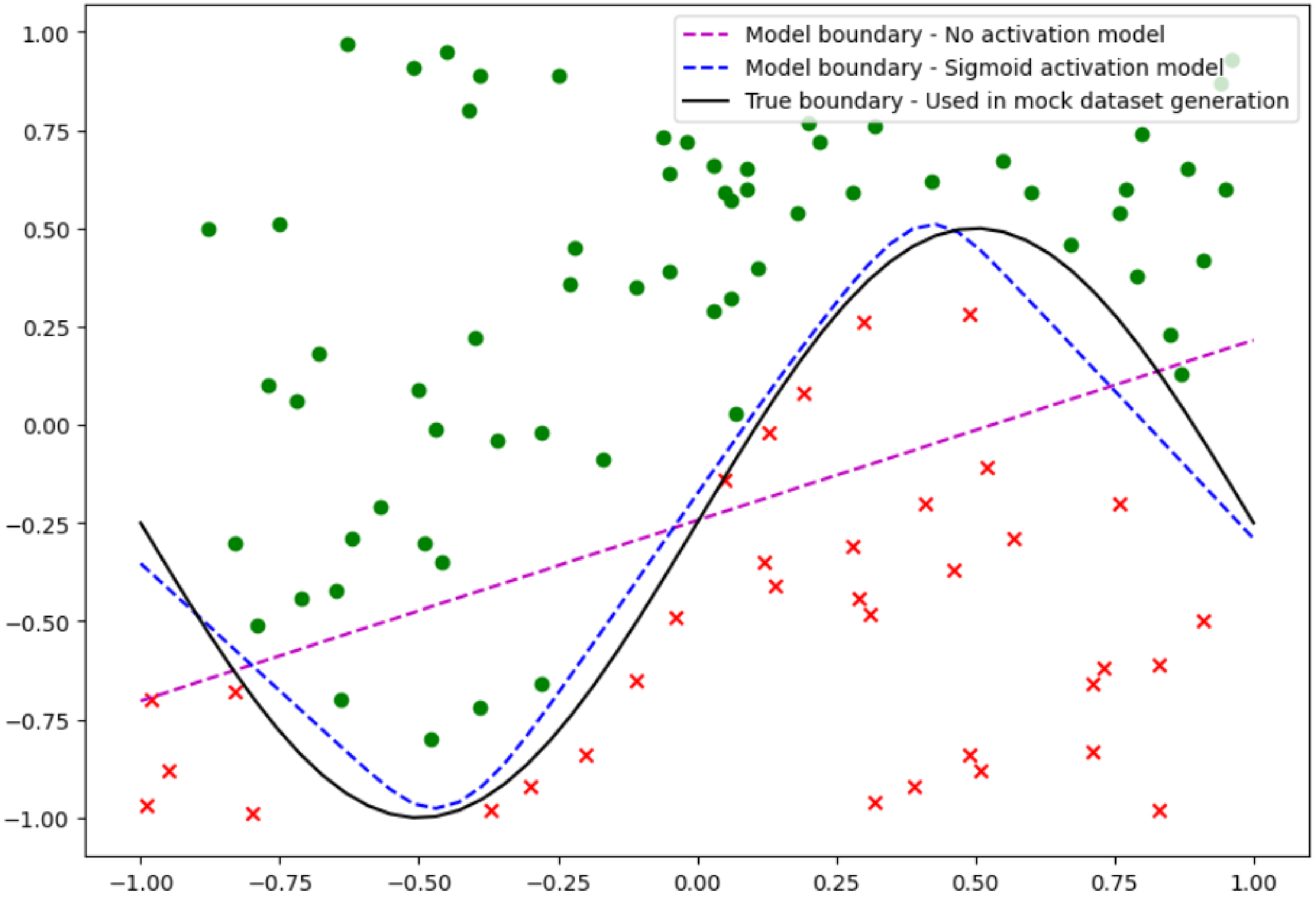
```
shallow_neural_net_act.show_losses_over_training()
```





Trains just fine!

Using the sigmoid activation functions helped the network **create some non-linearity!**
(Blue curve looking good and correctly separating green from red!)



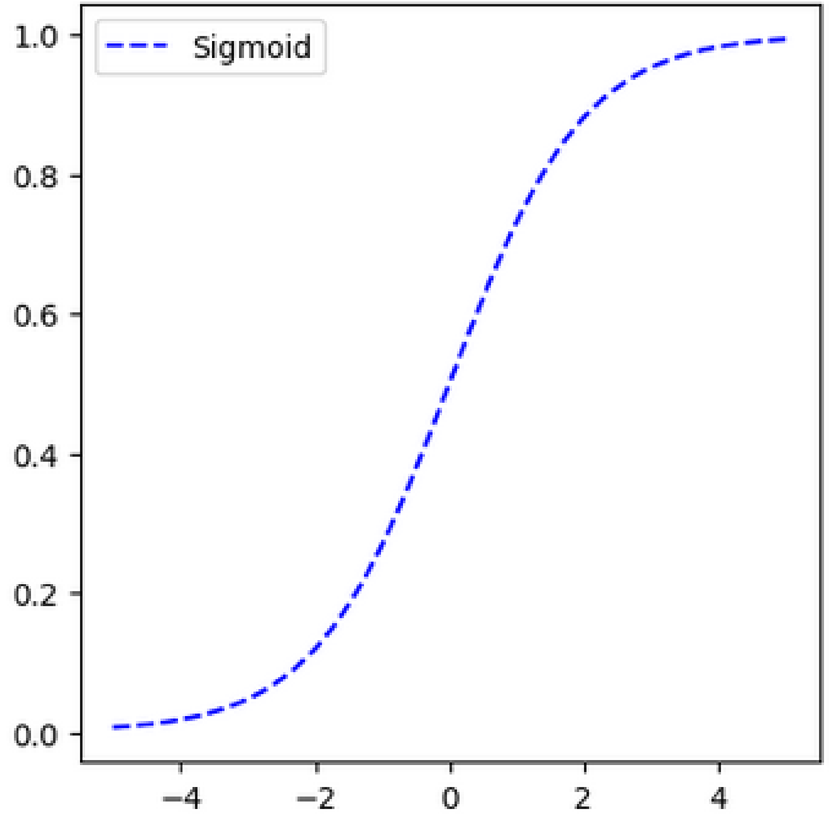
Examples of activations: **Sigmoid**

Definition (Sigmoid function):

$$s(x) = \frac{1}{1 + \exp(-x)}$$

The **sigmoid** **activation function** is often used in the **output layer of a binary classification model**, because the output will then have values between 0 and 1, and can therefore be interpreted as a **probability**.

```
def sigmoid(val):
    return 1/(1 + np.exp(-val))
```



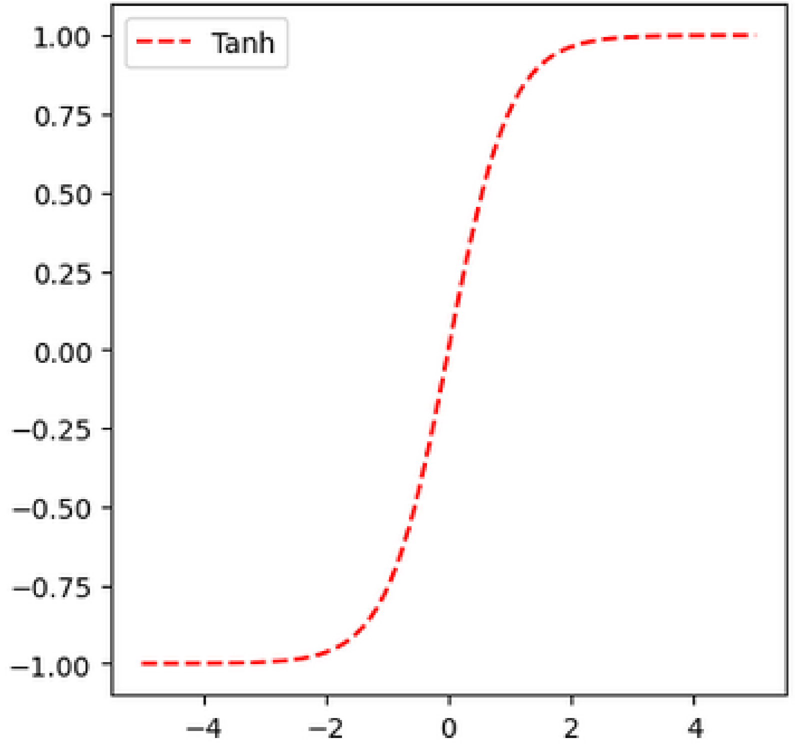
Examples of activations: **Hyperbolic tangent**

Definition (Tanh function):

The **tanh (short for "hyperbolic tangent") activation function** is somewhat similar to the sigmoid function, but it maps values to a range between -1 and 1.

Like the sigmoid function, it is often used in the output layer of a classification model.

```python
def tanh(val):
    return np.tanh(val)
```



About Sigmoid and Tanh functions

Some observations about Sigmoid and Tanh functions (empirical, subject to No Free Lunch!)

- Those activation functions were widely used prior to ReLU.
- People were very comfortable with those as they were reminiscent of Logistic Regression and they are simply differentiable.
- The problem with those is that being squeezed between [0, 1] or [-1, 1], we can have a hard time training deep networks, as the gradient will tend to vanish when neurons want to output values close to the interval bounds (0, 1, and -1) as layer outputs.
- Indeed, derivatives of these activations are closer to 0 when sigmoid/tanh produce values close their interval bounds!

Examples of activations: **ReLU**

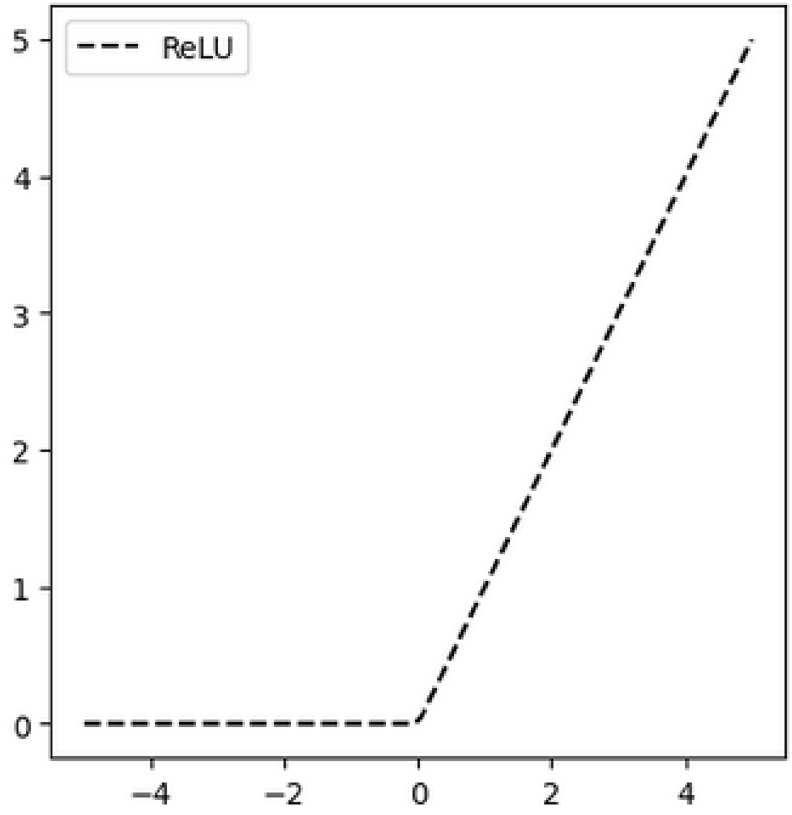
Definition (ReLU function):

The **ReLU** (short for "**Rectified Linear Unit**") **activation function** is a function that maps any input value less than 0 to 0, and any input value greater than or equal to 0 to the input value itself.

It is widely used because it is computationally efficient and does not saturate (i.e., "die") like some previous activation functions.

From [Glorot2011].

```python
def ReLU(val):
    return np.maximum(0, val)
```



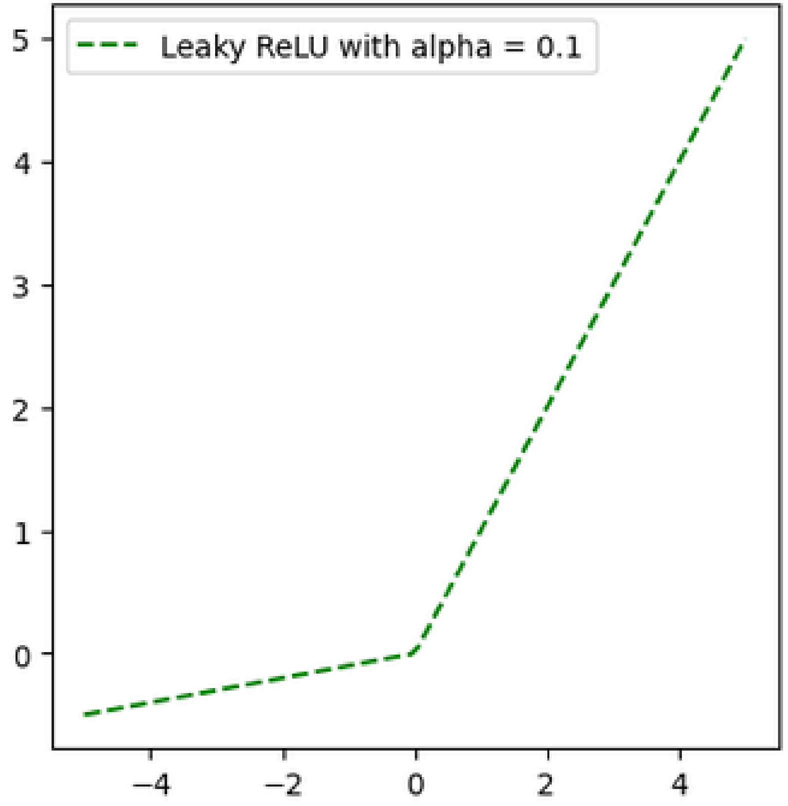
Examples of activations: **Leaky ReLU**

Definition (Leaky ReLU function):

The **leaky ReLU activation** function is similar to the ReLU function, but it allows a **small negative slope** for input values less than 0.

This can help to alleviate the "dying ReLU" problem, where some neurons in the network "die" when they produce negative values and might no longer respond to input.

```
def leaky_ReLU(val, alpha = 0.1):
    return np.where(val > 0, val, alpha*val)
```

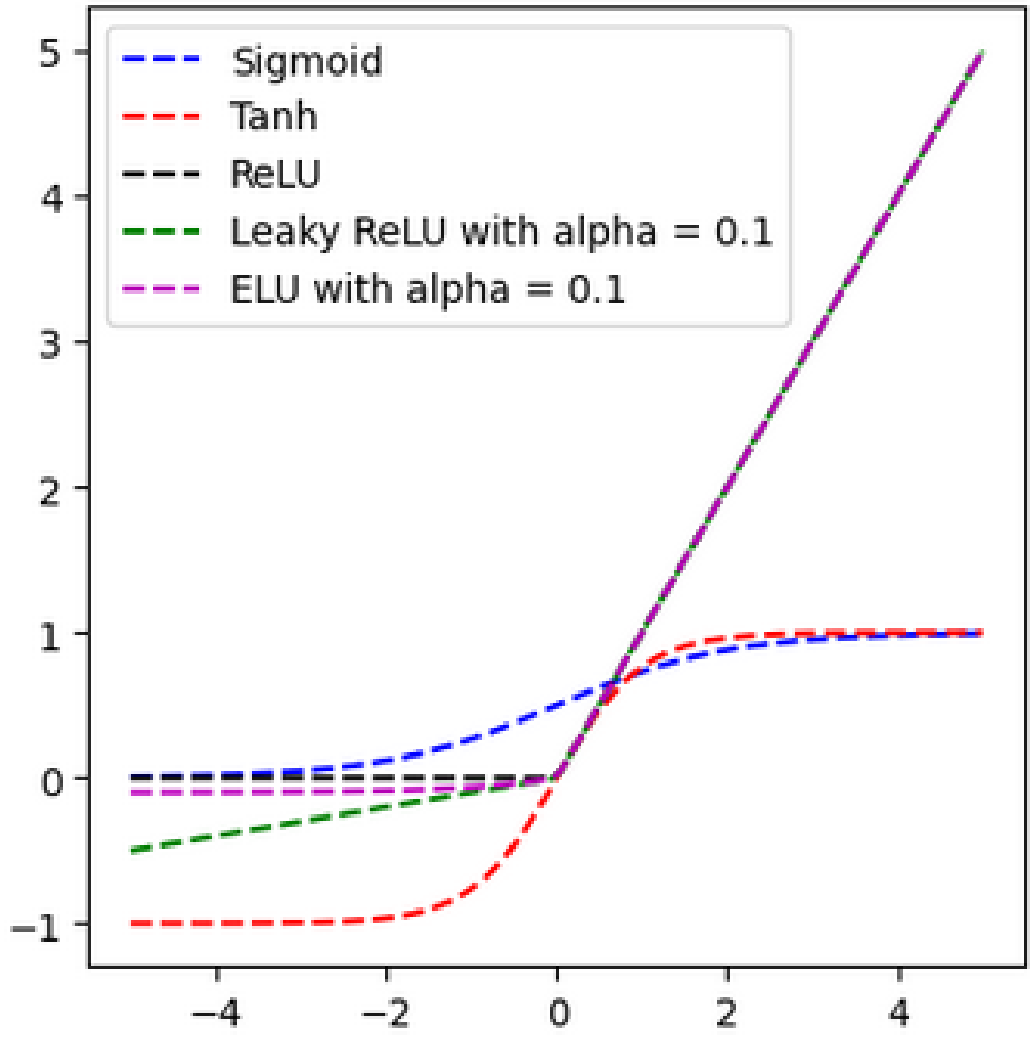


A note on activation functions

We have listed the most common activation functions, which are used most of the time.

Keep in mind that **many more activation functions exist**, some of them being very advanced or niche, but worth keeping an eye on. https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity

(More in bonus slides!)



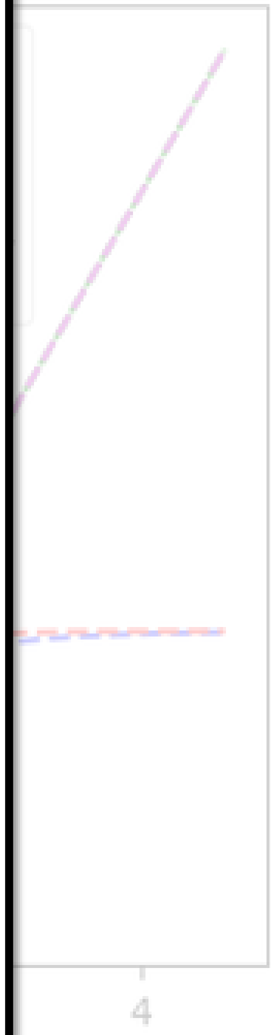
A n

We k
activ
used

Keep
activ
of th
wort
look
http
n.htm
weig

Non-linear Activations (weighted sum, nonlinearity)

`nn.ELU`	Applies the Exponential Linear Unit (ELU) function, element-wise, as described in the paper: Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs).
`nn.Hardshrink`	Applies the Hard Shrinkage (Hardshrink) function element-wise.
`nn.Hardsigmoid`	Applies the Hardsigmoid function element-wise.
`nn.Hardtanh`	Applies the HardTanh function element-wise.
`nn.Hardswish`	Applies the Hardswish function, element-wise, as described in the paper: Searching for MobileNetV3.
`nn.LeakyReLU`	Applies the element-wise function:
`nn.LogSigmoid`	Applies the element-wise function:



The universal approximation theorem

Definition (the Universal Approximation Theorem):

Mathematically speaking, **any neural network architecture aims at finding the mathematical function $y = f(x)$ that can map some given inputs x to outputs y.** This function $f(x)$ can be arbitrarily complex, based on the dataset and task at hand.

The **Universal Approximation Theorem** states that **Neural Networks have a kind of universality property.**

In Layman terms, **no matter what $f(x)$ might be, there is a neural network that can approach the function $f(x)$** with a given precision!

As such, this is the most important theorem of Deep Learning!

The universal approximation theorem

Definition (the Universal Approximation Theorem):

Mathematically speaking, any neural network architecture aims at finding the mathematical function $y = f(x)$ that can map some given inputs x to outputs y. This function $f(x)$ can be arbitrarily complex, based on the dataset and task at hand.

The Universal Approximation Theorem states that Neural Networks have a kind of universality property.

In Layman terms, no matter what $f(x)$ might be, there is a neural network that can approach the function $f(x)$ with a given precision!

As such, this is the most important theorem of Deep Learning!

Two caveats, however:

- The model in question might be unfeasibly large ("slightly" annoying…!)

- And the resulting model is not even guaranteed to generalize (still our main goal last time I checked)

Learn more about these topics

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

- [LeCun1998] Y. **LeCun**, Y. **Bengio**, and G. **Hinton**, “Efficient backprop. In Neural networks: Tricks of the trade”, 1998.
- [Glorot2010] X. **Glorot**, and Y. **Bengio**, “Understanding the difficulty of training deep feedforward neural networks”, 2010.
- [He2015] K. **He**, X. Zhang, S. Ren, & J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.”, 2015.

Learn more about these topics

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

- [LeCun1998bis] Y. **LeCun**, L. **Bottou**, Y. **Bengio**, and P. Haffner "Gradient-based learning applied to document recognition.", 1998.
- [Glorot2011] X. **Glorot**, A. Bordes, and Y. **Bengio**, "Deep Sparse Rectifier Neural Networks", 2011.
- [Dubey2022] S. R. Dubey et al., "Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark", 2022.

Learn more about these topics

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

- **Yann** **LeCun**: **Formerly, chief AI Scientist** at **Facebook/Meta** and **Silver Professor** at the **New York University**, **another one of the three Godfathers of Deep Learning** and **2018 Turing Award** winner (highest distinction in Computer Science).
  http://yann.lecun.com/
  https://scholar.google.com/citations?user=WLN3QrAAAAAJ&hl=fr
- **Yoshua** **Bengio**: **Professor** at **University of Montreal**, **last of the three Godfathers of Deep Learning** and **2018 Turing Award** winner (highest distinction in Computer Science).
  https://yoshuabengio.org
  https://scholar.google.com/citations?user=kukA0LcAAAAJ&hl=fr

Learn more about these topics

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

- **Xavier Glorot:** Formerly, **DeepMind**. **Now, Computer Science Teacher** at Cégep Édouard-Montpetit https://scholar.google.com/citations?user=_WnkXlkAAAAJ&hl=fr
- **Kaiming He: Former researcher** at **Facebook AI Research (FAIR/Meta AI).** Now, **associate professor at MIT** and **DeepMind**. https://scholar.google.com/citations?user=DhtAFkwAAAAJ&hl=en

Learn more about these topics

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

- **Leon Bottou:** Works at **FAIR/Meta AI**.
  https://leon.bottou.org/
  https://scholar.google.fr/citations?user=kbN88gsAAAAJ&hl=fr
- **FAIR/Meta AI**: Facebook Artificial Intelligence Research, now Meta Artificial Intelligence is an academic research laboratory focused on generating knowledge for the AI community.
  https://ai.facebook.com/
  https://twitter.com/MetaAI?ref_src=twsrc%5Egoogle%7Ctwcamp%5Eserp%7Ctwgr%5Eauthor

The He initialization

Definition (the He initialization [LeCun1998]**):**

The **He initializer** is similar to the **Xavier initializer**, but with a **variance** $\frac{\mathbf{1}}{\boldsymbol{N_{in}+N_{out}}}$, where $\boldsymbol{N_{in}}$ (resp. $\boldsymbol{N_{out}}$) is the size of each layer input (resp. output). Each layer is therefore initialized with its own variance.

It seems to **improve performance when working with deeper networks**.

```python
    def init_parameters_he(self):
        # Weights and biases matrices (He initialized)
        range1 = np.sqrt(4/(self.n_x + self.n_h))
        self.W1 = np.random.randn(self.n_x, self.n_h)*range1
        self.b1 = np.random.randn(1, self.n_h)*range1
        range2 = np.sqrt(4/(self.n_h + self.n_y))
        self.W2 = np.random.randn(self.n_h, self.n_y)*range2
        self.b2 = np.random.randn(1, self.n_y)*range2
```

The LeCun initialization

Definition (the LeCun initialization [LeCun1998bis]):

The **LeCun initializer** is **based on a random normal distribution**, with a variance of $\sqrt{\frac{1}{N_{in}}}$, with N_{in} being the number of inputs of each layer.

This initializer is reported to be particularly useful for **architectures with sigmoid and tanh activation functions.**

```
    def init_parameters_lecun(self):
        # Weights and biases matrices (LeCun initialized)
        range1 = np.sqrt(1/self.n_x)
        self.W1 = np.random.randn(self.n_x, self.n_h)*range1
        self.b1 = np.zeros((1, self.n_h))
        range2 = np.sqrt(1/self.n_h)
        self.W2 = np.random.randn(self.n_h, self.n_y)*range2
        self.b2 = np.zeros((1, self.n_y))
```

Initializations variations and more

Many more initialization formulas exist:

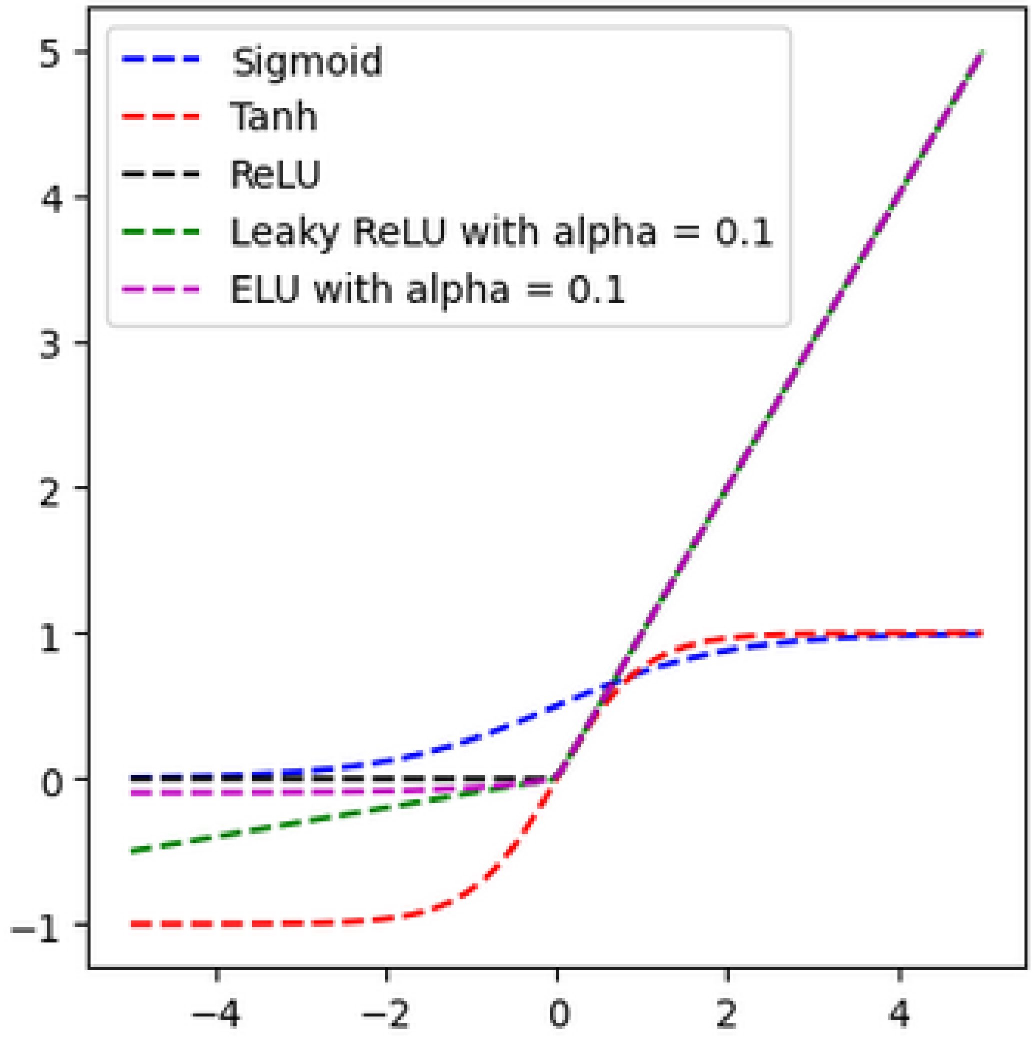
- Xavier has random uniform and normal variations (same for He),
- Glorot proposed more initializations,
- Orthogonal initializations are sometimes useful but rare,
- Variance scaling initialization also exists but rarely used,
- Etc.

PyTorch has listed a lot of them, ready to use:
https://pytorch.org/docs/stable/nn.init.html

A note on activation functions

We have listed the most common activation functions, which are used most of the time.

Keep in mind that **many more activation functions exist**, some of them being very advanced or niche, but worth keeping an eye on. https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity



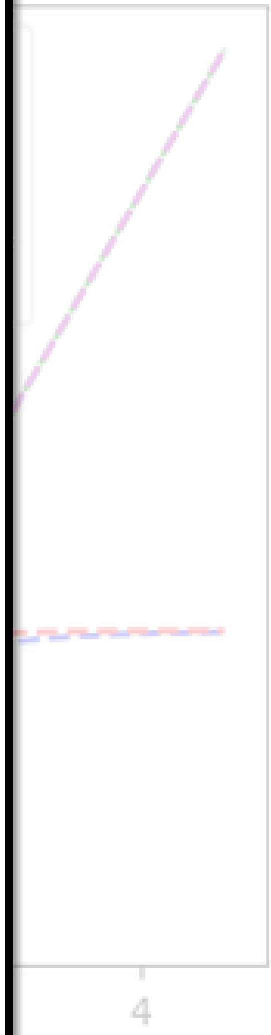
A n

We h
activ
used

Keep
activ
of th
wort
look
http
n.htm
weig

Non-linear Activations (weighted sum, nonlinearity)

`nn.ELU`	Applies the Exponential Linear Unit (ELU) function, element-wise, as described in the paper: Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs).
`nn.Hardshrink`	Applies the Hard Shrinkage (Hardshrink) function element-wise.
`nn.Hardsigmoid`	Applies the Hardsigmoid function element-wise.
`nn.Hardtanh`	Applies the HardTanh function element-wise.
`nn.Hardswish`	Applies the Hardswish function, element-wise, as described in the paper: Searching for MobileNetV3.
`nn.LeakyReLU`	Applies the element-wise function:
`nn.LogSigmoid`	Applies the element-wise function:



From (Leaky) ReLU to more advanced LU

- Using **ReLU**, we can train deeper models, but the gradient would still die for negative numbers due to the zeroing in $x < 0$.
- Numerous Activation Functions were created to address this problem such as **LeakyReLU** and **PReLU**, but were still subject to issues related to gradients.
- The most notable one would be the **Exponential Linear Unit** (**ELU**) functions.
- Exponential behaviour sped up learning by bringing the normal gradient closer to the unit natural gradient because of a reduced bias shift effect.

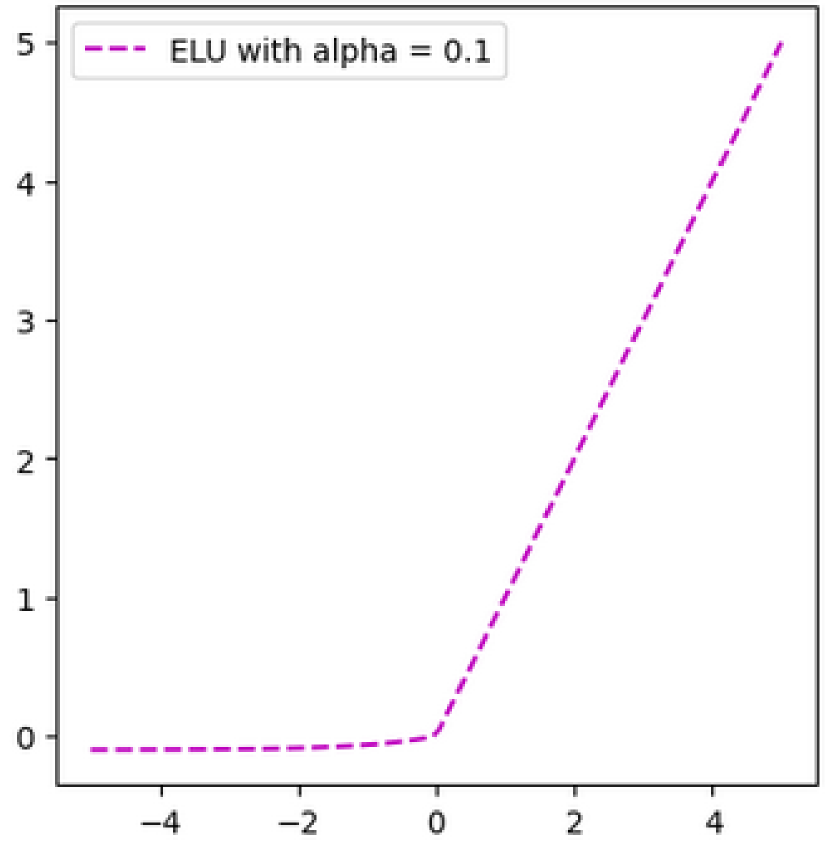
Examples of activations: **ELU**

```
def ELU(val, alpha = 0.1):
    return np.where(val > 0, val, alpha*(np.exp(val) - 1))
```

Definition (ELU function):

The **ELU** (short for "**exponential linear unit**") **activation function** is similar to the ReLU function, but it has a negative slope for input values less than 0.

As we mentioned in the leaky ReLU function, ELU can help to alleviate the "dying ReLU" problem.



A note on more advanced activation functions

- More recent activation functions, such as **Swish** and **Mish**, are also worth looking into.
- Some functions like **PReLU** even suggest to use activation functions with **learnable/trainable parameters**.
- Those **adaptive/trainable activation functions** allow for different neurons to learn different activation functions for richer learning while adding parametric complexity to the networks (see [Dubey2022] for a good benchmark on activation functions).
- More in HW next week?

A note on more advanced activation functions

- In addition, the class of **Gated Linear Unit** (or **GLU**) has been studied quite a bit in Natural Language Processing architectures and they control what information is passed up to the following layer using gates mechanisms similar to the ones found in LSTMs.
- More on this on Week 6, stay tuned for more!