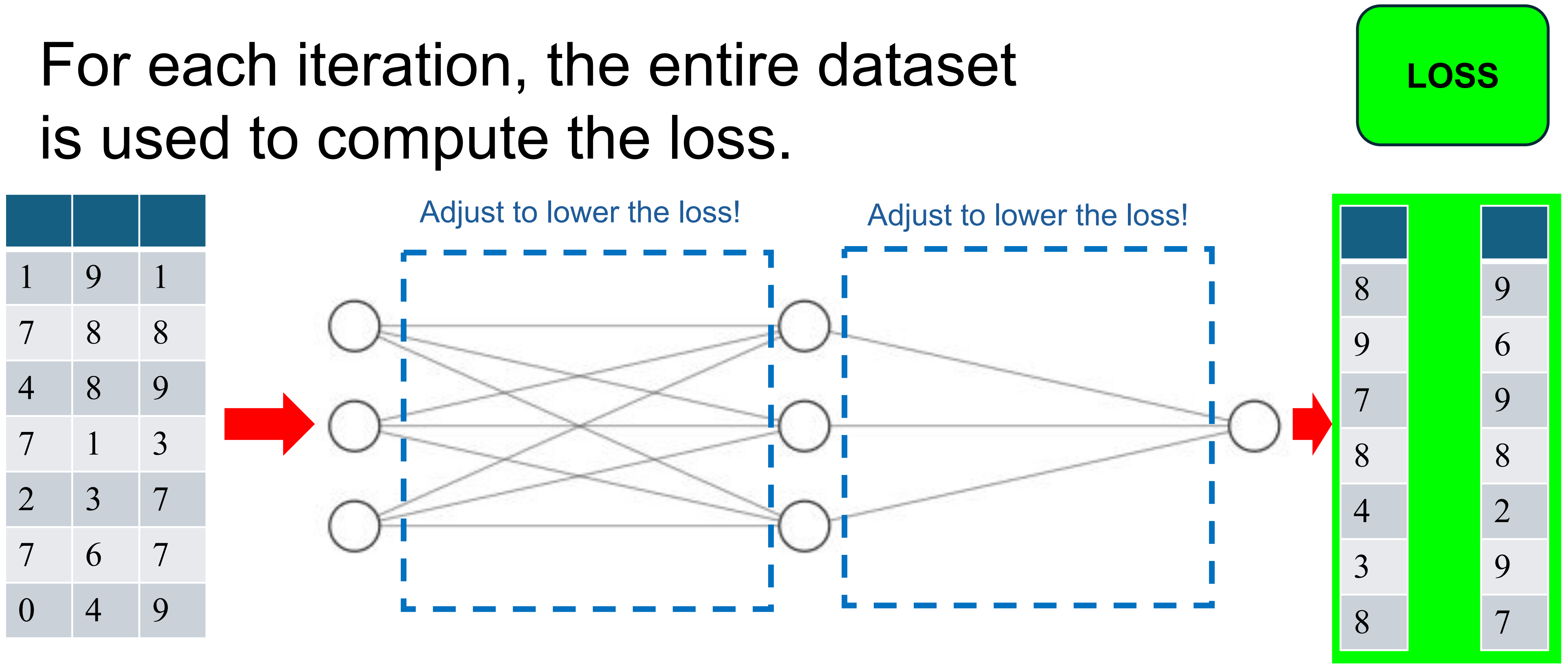
Neural Networks, Part 2

Original Slides by:
Courtney Anne Ngo
Daniel Stanley Tan, PhD
Arren Antiquioa

Updated (AY 2024 – 2025 T1) by:
Thomas James Tiam-Lee, PhD

Gradient Descent

- For each iteration, the entire dataset is used to compute the loss.



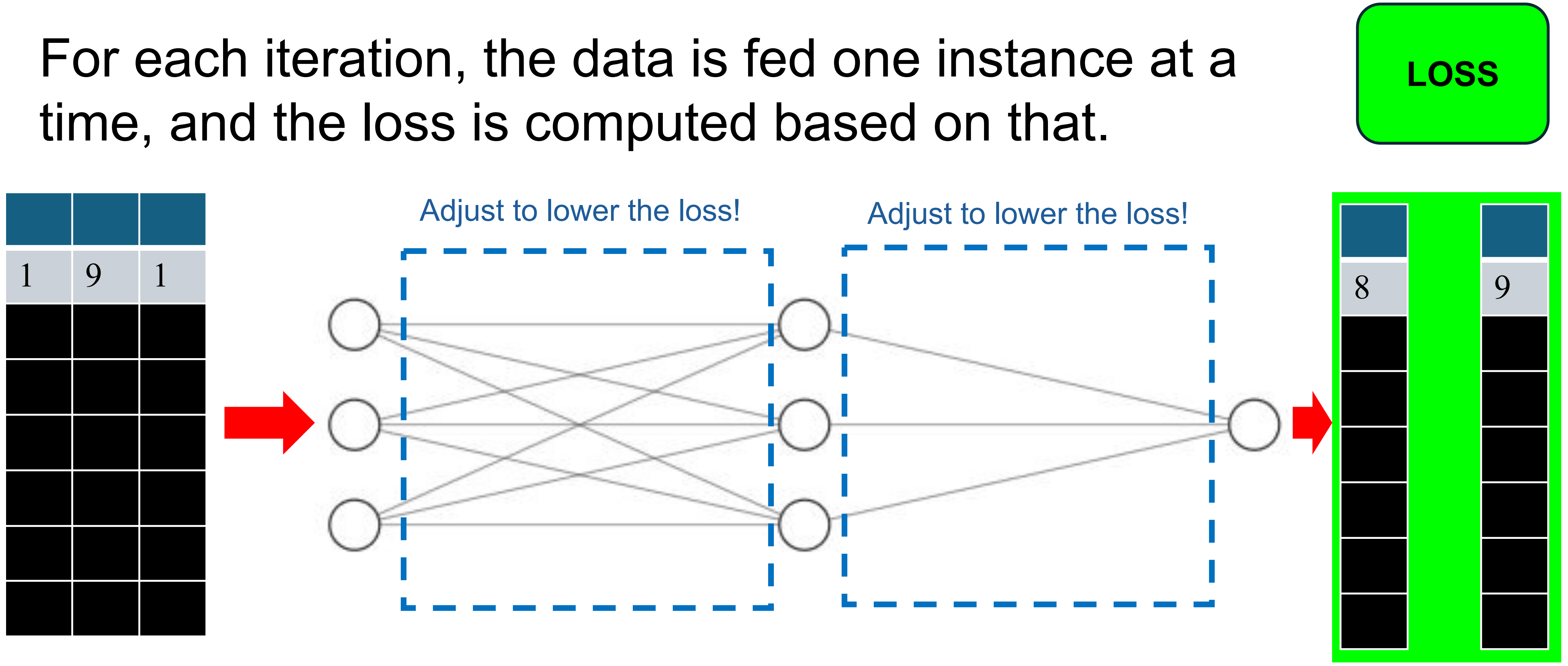
1	9	1
7	8	8
4	8	9
7	1	3
2	3	7
7	6	7
0	4	9

8	9
9	6
7	9
8	8
4	2
3	9
8	7

- Problem: computationally expensive if the dataset is large!

Stochastic Gradient Descent

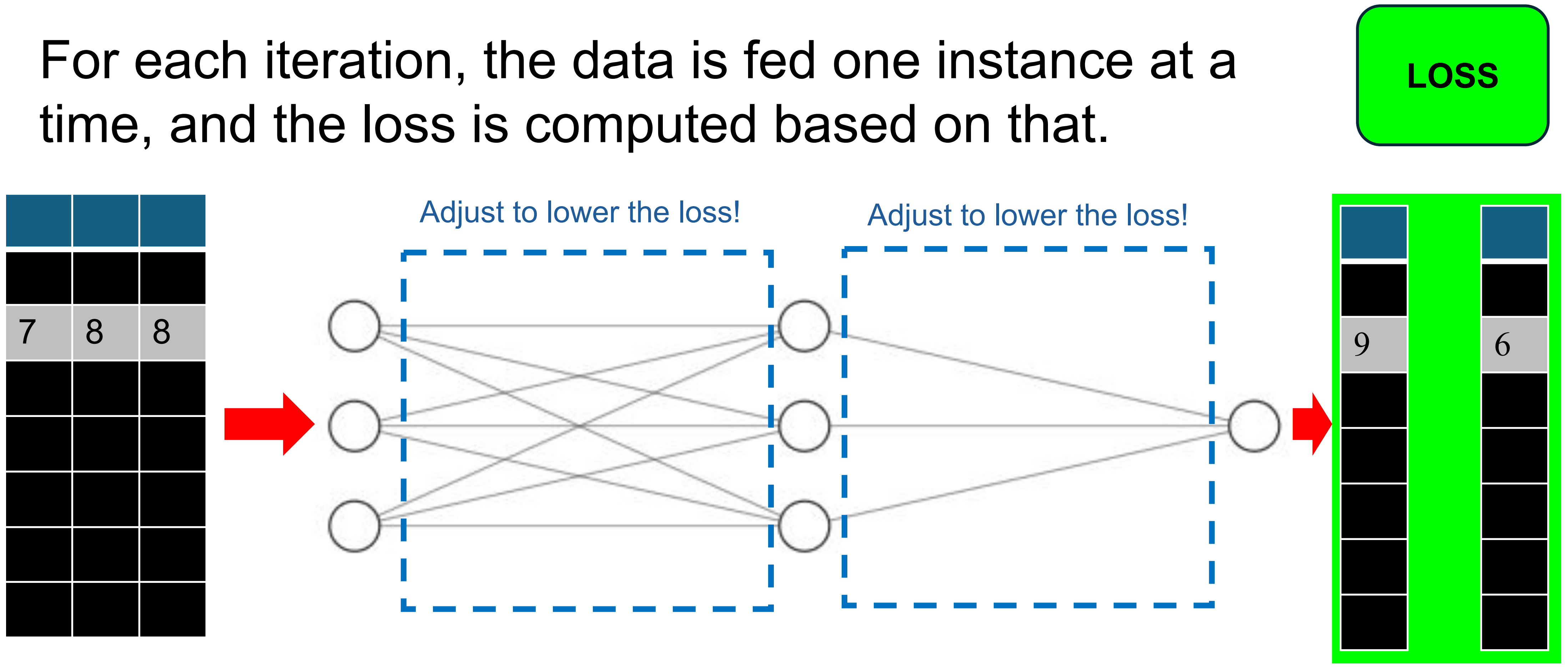
- For each iteration, the data is fed one instance at a time, and the loss is computed based on that.



Iteration 1

Stochastic Gradient Descent

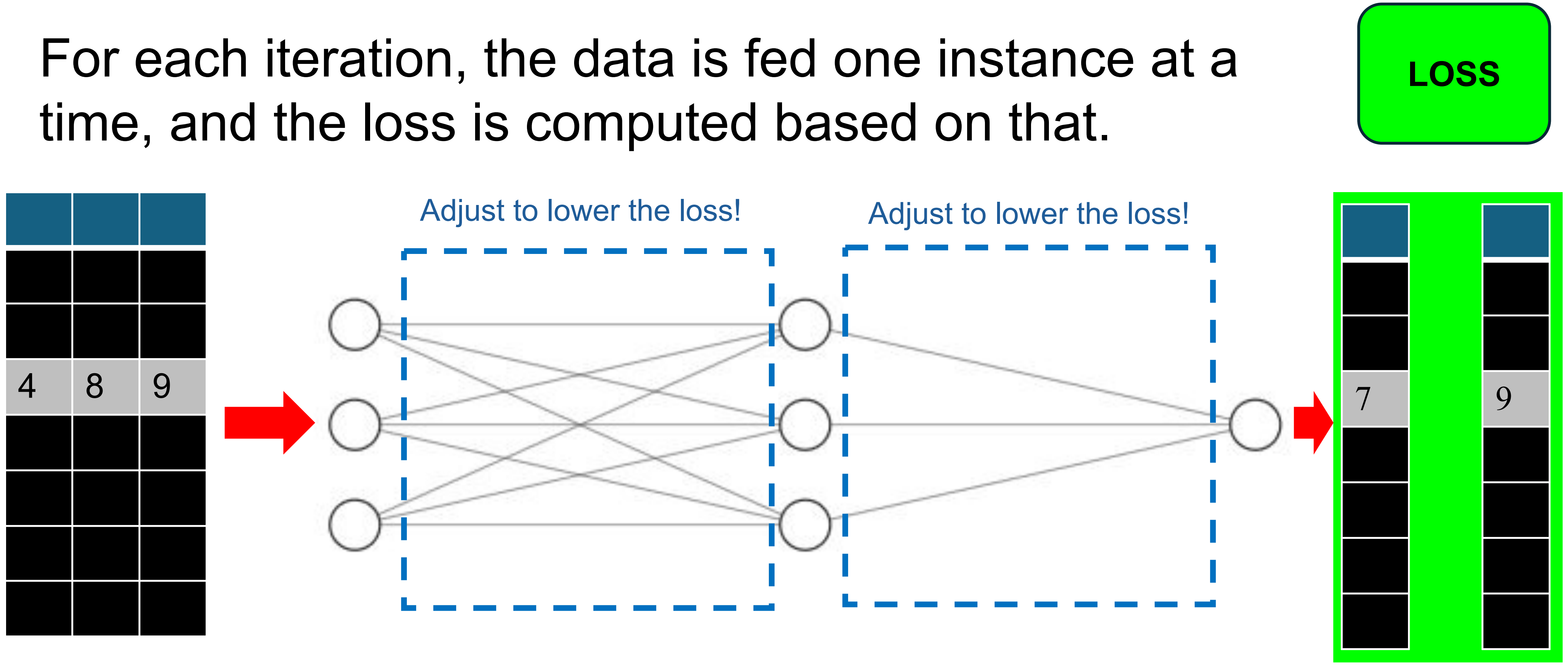
- For each iteration, the data is fed one instance at a time, and the loss is computed based on that.



Iteration 2

Stochastic Gradient Descent

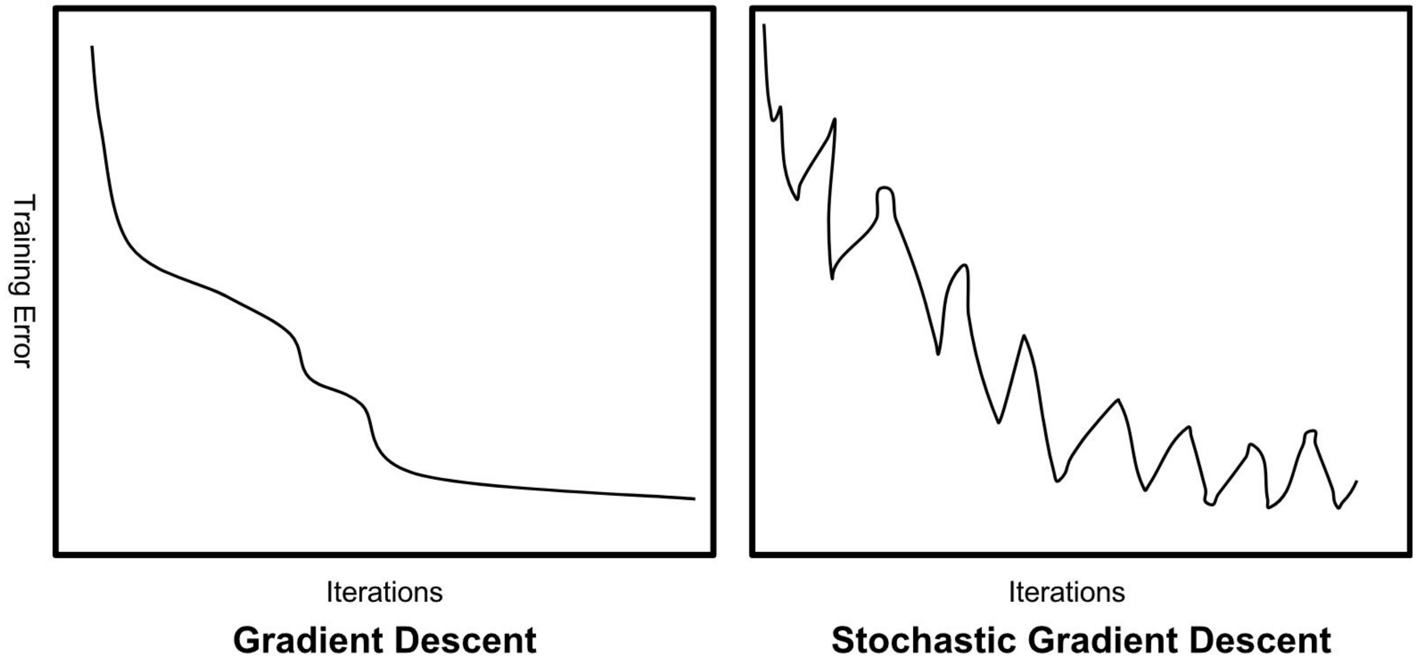
- For each iteration, the data is fed one instance at a time, and the loss is computed based on that.



Iteration 3, and so on…

Stochastic Gradient Descent

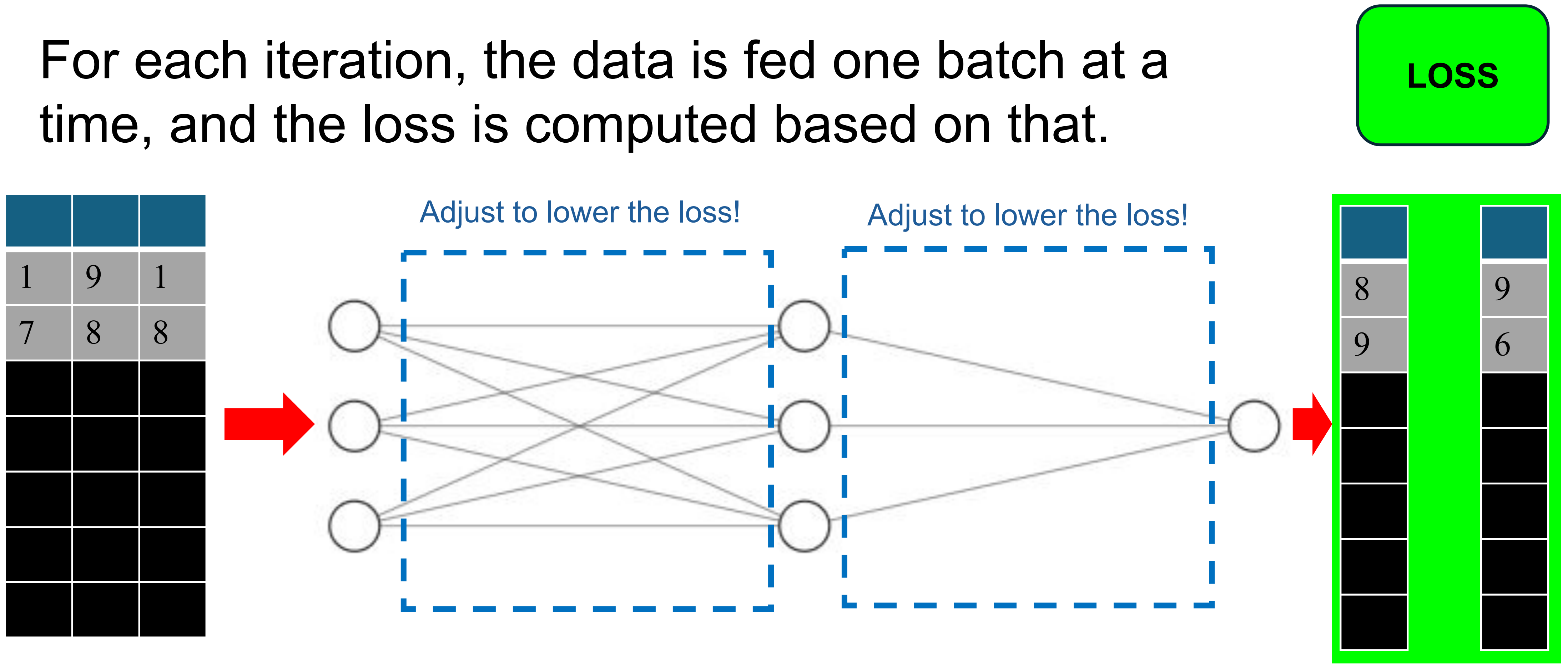
- When all the instances have been fed, this is known as one **epoch**.
- You must run **many epochs** in stochastic gradient descent.
- **Will this still give an accurate result?**
  - Yes, it turns out that this can still approximate the minimum of the loss function quite well.



Source: cs760 University of Wisconsin (https://pages.cs.wisc.edu/~spehlmann/cs760/_site/project/2017/05/04/intro.html)

Mini-Batch Gradient Descent

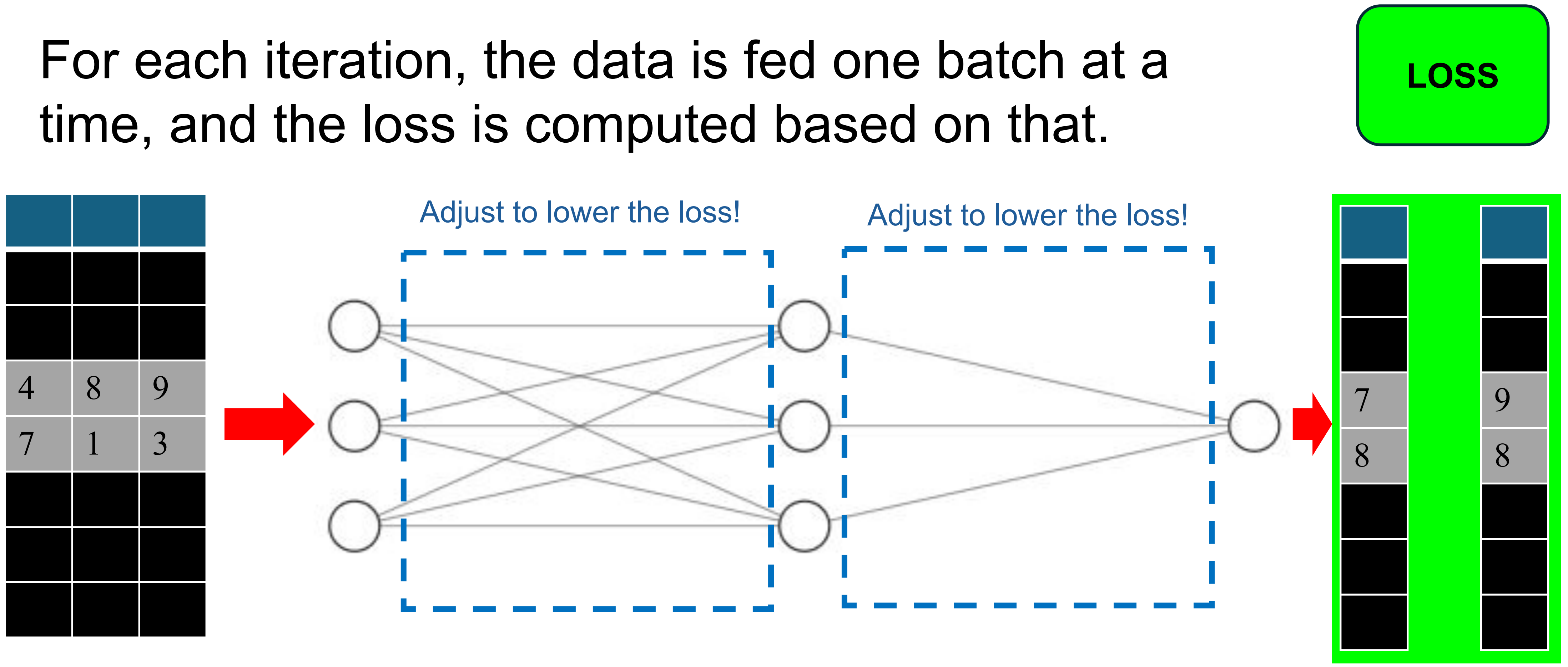
- For each iteration, the data is fed one batch at a time, and the loss is computed based on that.



Iteration 1 (batch size = 2)

Mini-Batch Gradient Descent

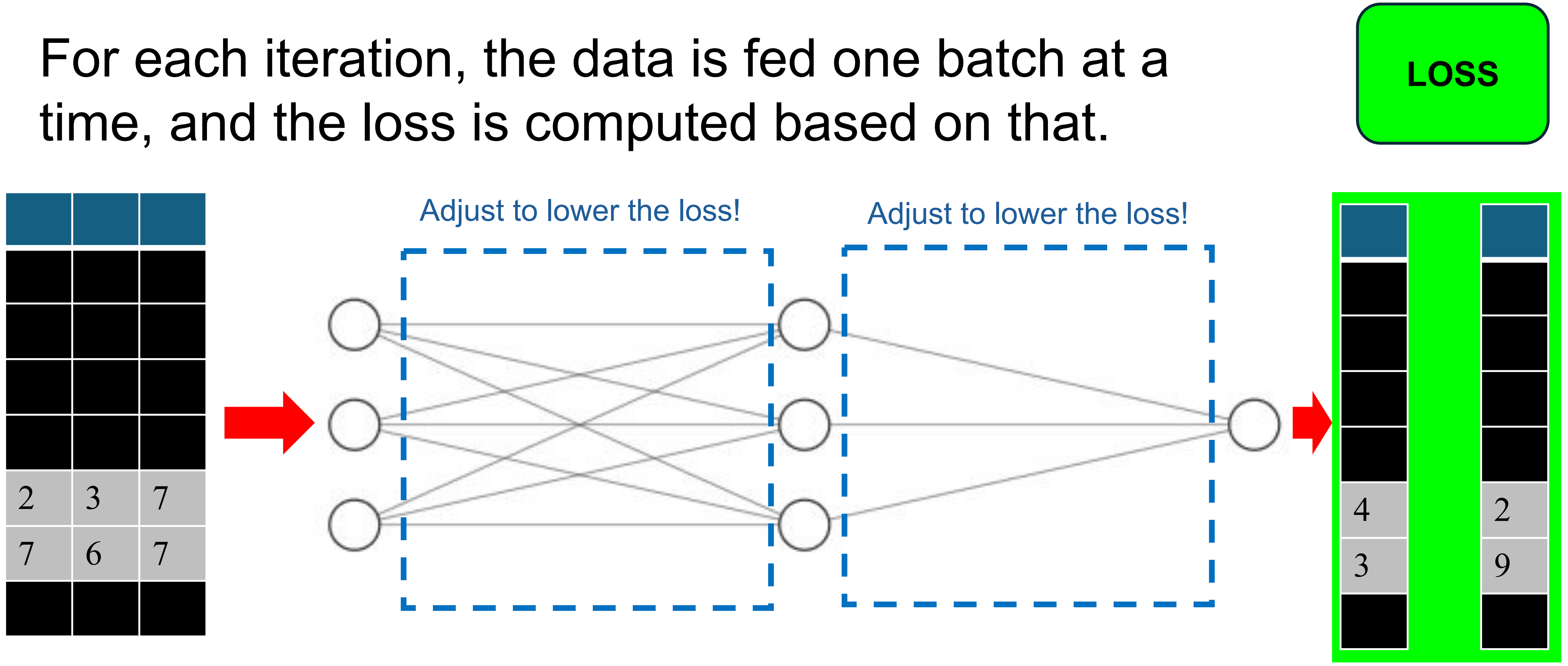
- For each iteration, the data is fed one batch at a time, and the loss is computed based on that.



Iteration 2 (batch size = 2)

Mini-Batch Gradient Descent

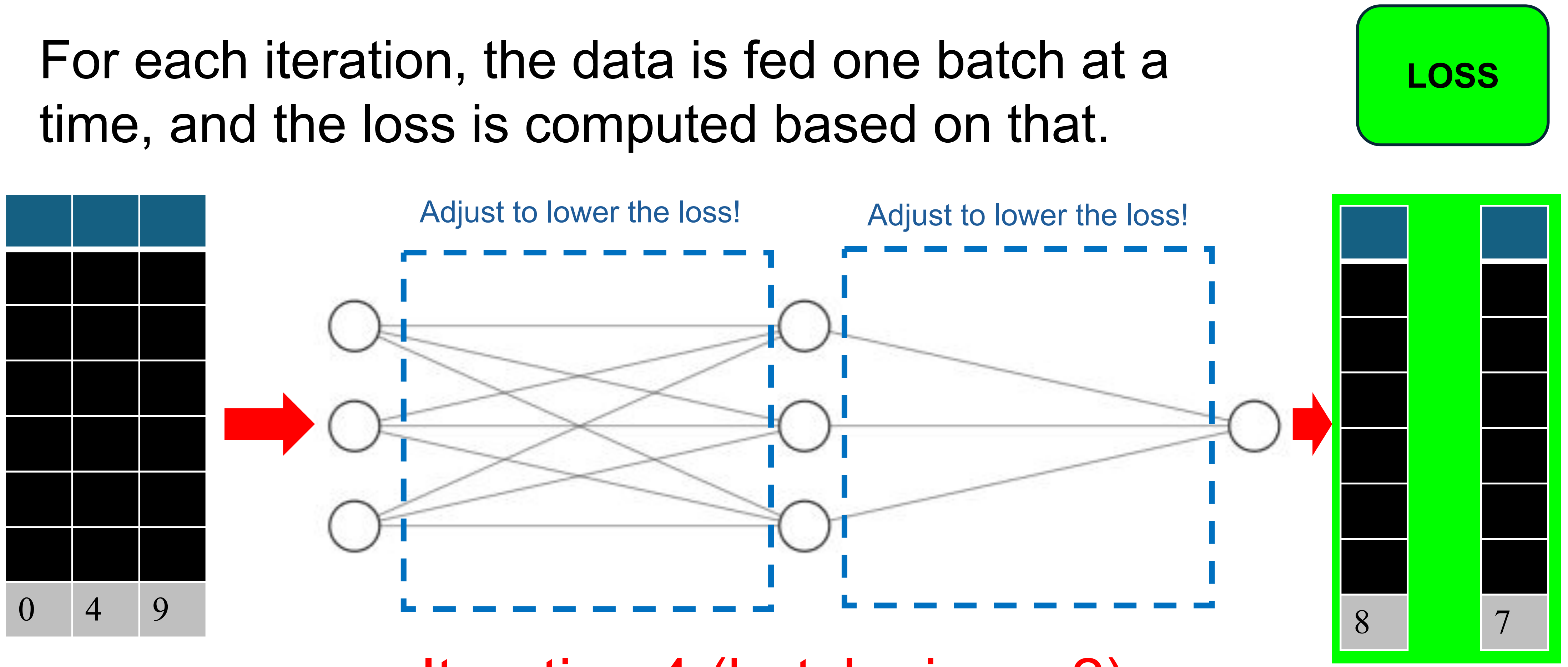
- For each iteration, the data is fed one batch at a time, and the loss is computed based on that.



Iteration 3 (batch size = 2)

Mini-Batch Gradient Descent

- For each iteration, the data is fed one batch at a time, and the loss is computed based on that.



Iteration 4 (batch size = 2)

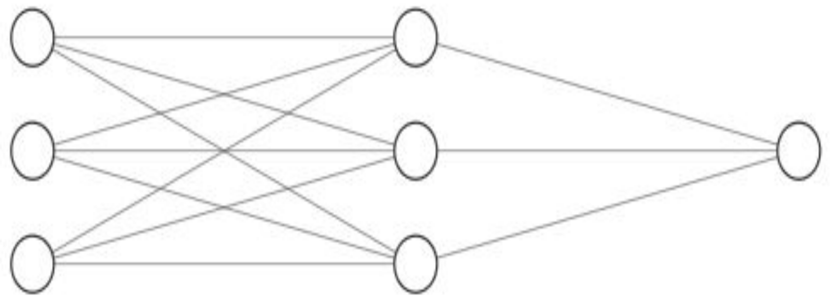
Note: if the instances is not divisible by the batch size, you can choose your own scheme of how to handle it.

Stochastic, Mini-Batch Gradient Descent

Gradient Descent	Stochastic GD	★ Mini-Batch GD
Updates θ based on gradient of the **whole dataset**	Updates θ based on gradient of **one data instance**	Updates θ based on gradient of a **subset of the whole dataset**
Runs slow	Runs fast, but jittery	Runs faster than GD, and path is not as jittery as stochastic GD
1 iter = N instances N = size of whole dataset	1 iter = 1 instance	1 iter = M instances, M = size of batch

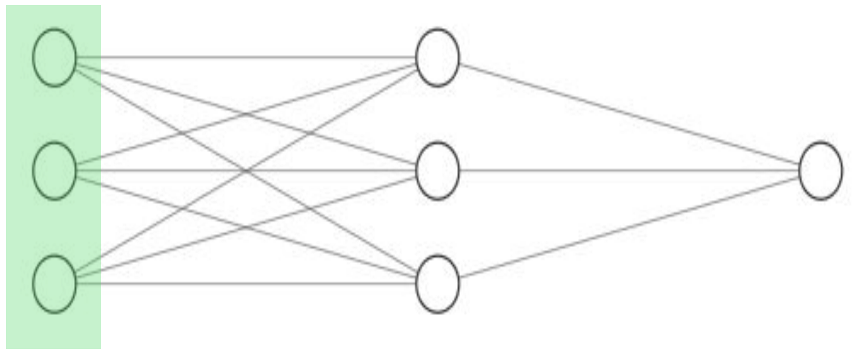
Designing the NN Architecture

- How do you decide:
    - The number of input neurons?
    - The number of hidden layers?
    - The number of neurons in each hidden layer?
    - The activation function in each hidden layer?
    - The number of output neurons?
    - The activation function of the output layer?



Designing the NN Architecture

- How do you decide:
  - The number of input neurons?
  - The number of hidden layers?
  - The number of neurons in each hidden layer?
  - The activation function in each hidden layer?
  - The number of output neurons?
  - The activation function of the output layer?



Input Neurons

- Number of input neurons = **number of dimensions of the feature vector (a.k.a. number of features)**

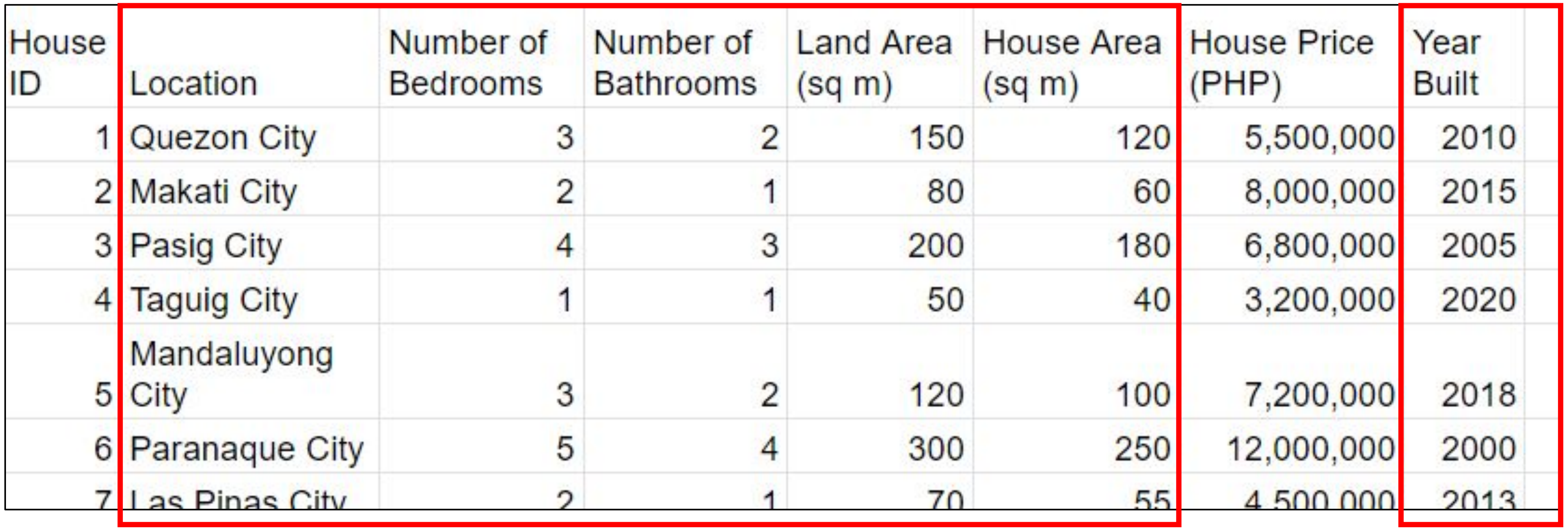
- Depends on the formulation of your task

- The number of input neurons must be **fixed**!

Input Neurons

- **Goal:** We want to make a system that can estimate the price of a house. We identified our features as follows:
- Location, number of bedrooms, number of bathrooms, land area, house area, year built
- How many input neurons should there be?

Input Neurons



House ID	Location	Number of Bedrooms	Number of Bathrooms	Land Area (sq m)	House Area (sq m)	House Price (PHP)	Year Built
1	Quezon City	3	2	150	120	5,500,000	2010
2	Makati City	2	1	80	60	8,000,000	2015
3	Pasig City	4	3	200	180	6,800,000	2005
4	Taguig City	1	1	50	40	3,200,000	2020
5	Mandaluyong City	3	2	120	100	7,200,000	2018
6	Paranaque City	5	4	300	250	12,000,000	2000
7	Las Pinas City	2	1	70	55	4,500,000	2013

- Structured data: easy, just use the number of features
- **6 input neurons**

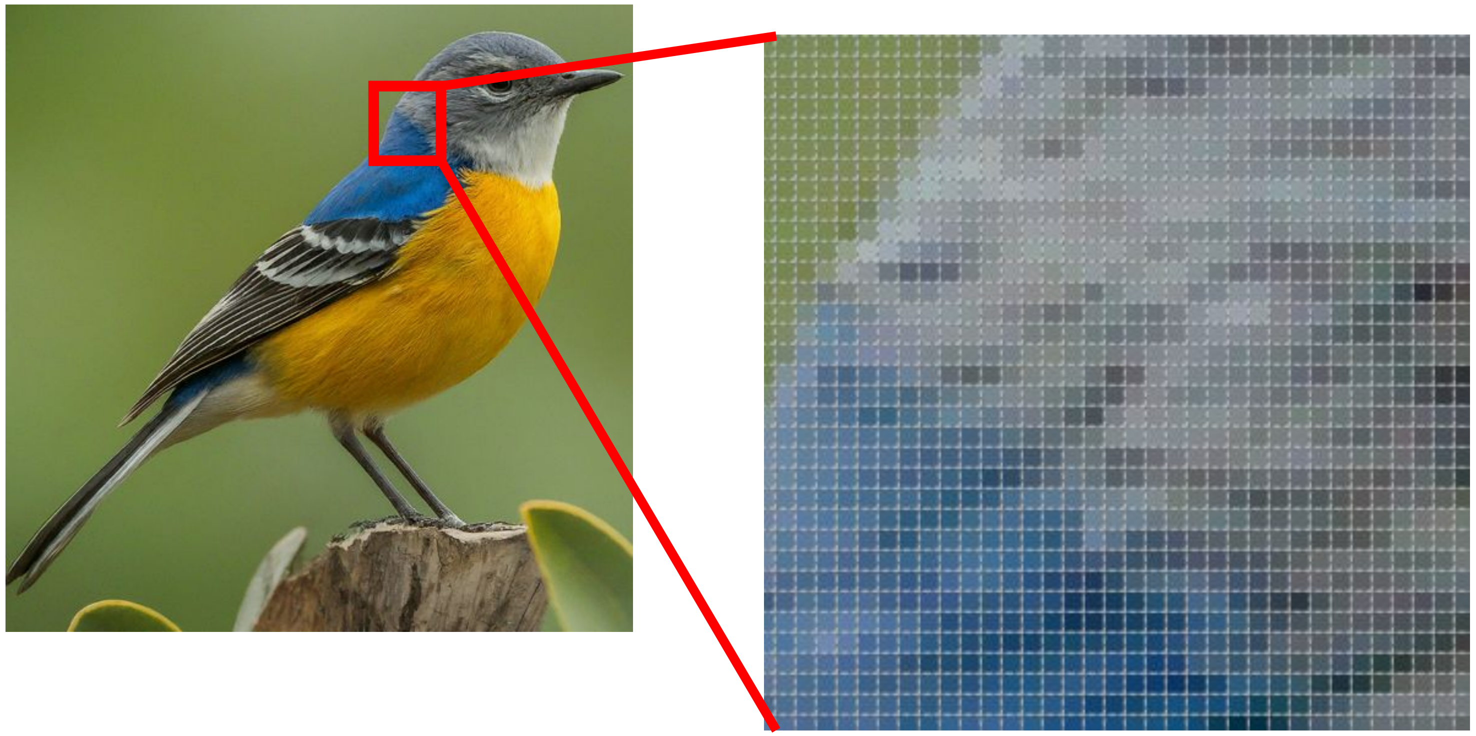
Input Neurons

- **Goal:** We want to make a system that can automatically detect if an animal image is a photo of a mammal, reptile, or bird.
- How many input neurons should there be?

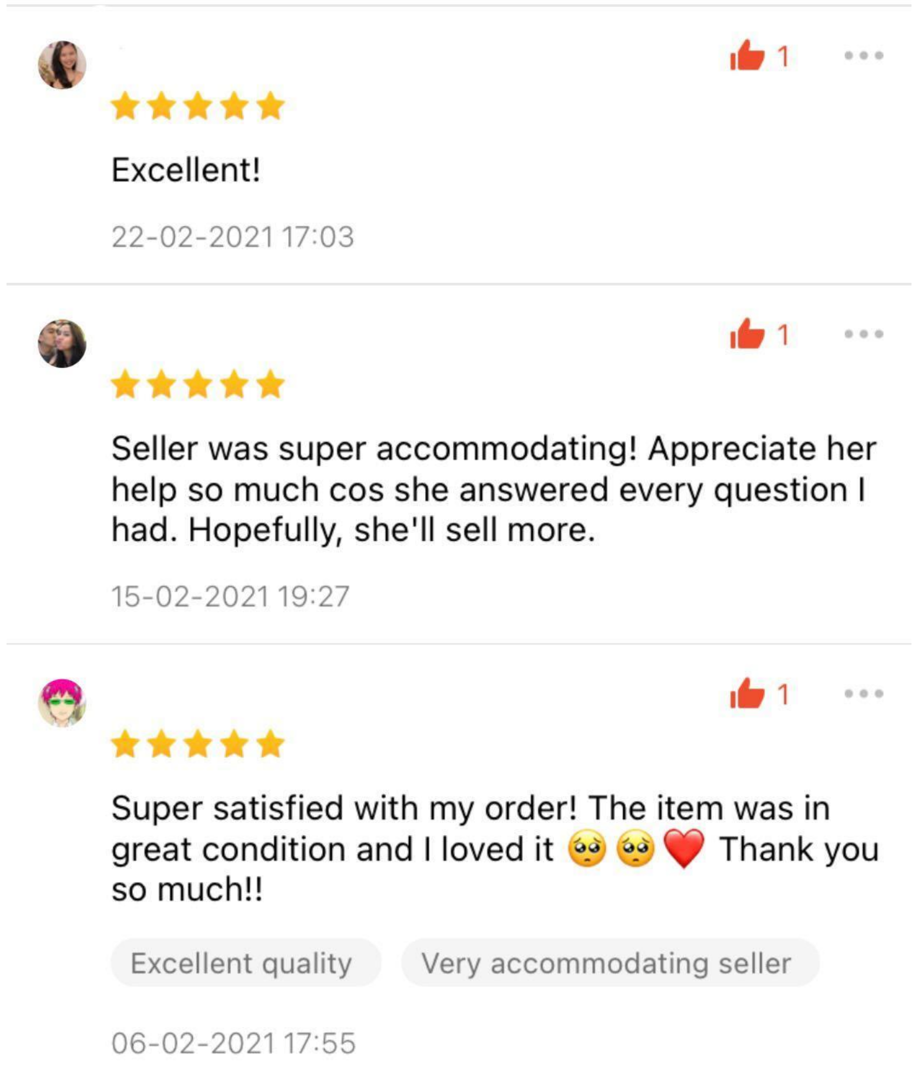
Input Neurons

- **Possible feature representation:**

- Resize all images to 100x100, then represent each pixel with 3 values (R, G, B).
- **30,000 input neurons!**



Input Neurons


1

Excellent!

22-02-2021 17:03

1

Seller was super accommodating! Appreciate her help so much cos she answered every question I had. Hopefully, she'll sell more.

15-02-2021 19:27

1

Super satisfied with my order! The item was in great condition and I loved it 🥺🥺❤️ Thank you so much!!

Excellent quality

Very accommodating seller

06-02-2021 17:55

- **Goal:** We want to make a system that gets the review text of a comment and classify whether it is a positive review or a negative review.

- How many input neurons should there be?

Input Neurons

- **Possible feature representation:**
- **Bag of words**
  - Each feature is a vocabulary in a predefined dictionary, and the value represents how many times that word appears in the text.

The product was very very good!

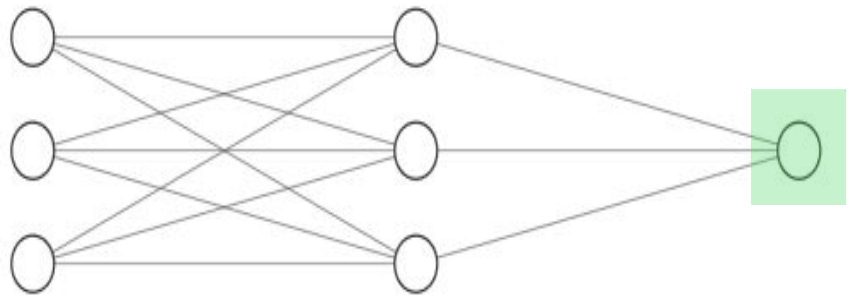
able	bad	carrot	good	great	product	sad	very	zebra	…..
0	0	0	1	0	1	0	2	0	…..

- **n input neurons**, where n is the number of words in the dictionary!

Note: State-of-the-art approaches now use another kind of feature representation called a **sentence embedding**. Try to read up on this concept and determine how many input neurons will this be!

Designing the NN Architecture

- How do you decide:



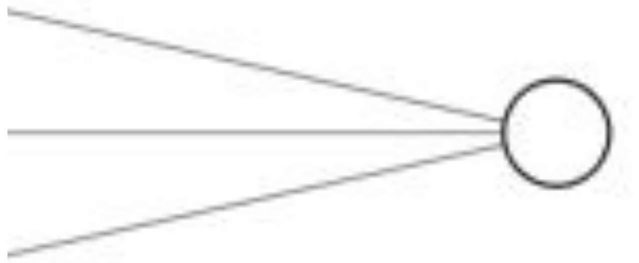
  - The number of input neurons?
  - The number of hidden layers?
  - The number of neurons in each hidden layer?
  - The activation function in each hidden layer?
  - The number of output neurons?
  - The activation function of the output layer?

Output Neurons

- **Goal:** We want to make a system that can estimate the price of a house. We identified our features as follows:
- Location, number of bedrooms, number of bathrooms, land area, house area, year built
- How many output neurons should there be?

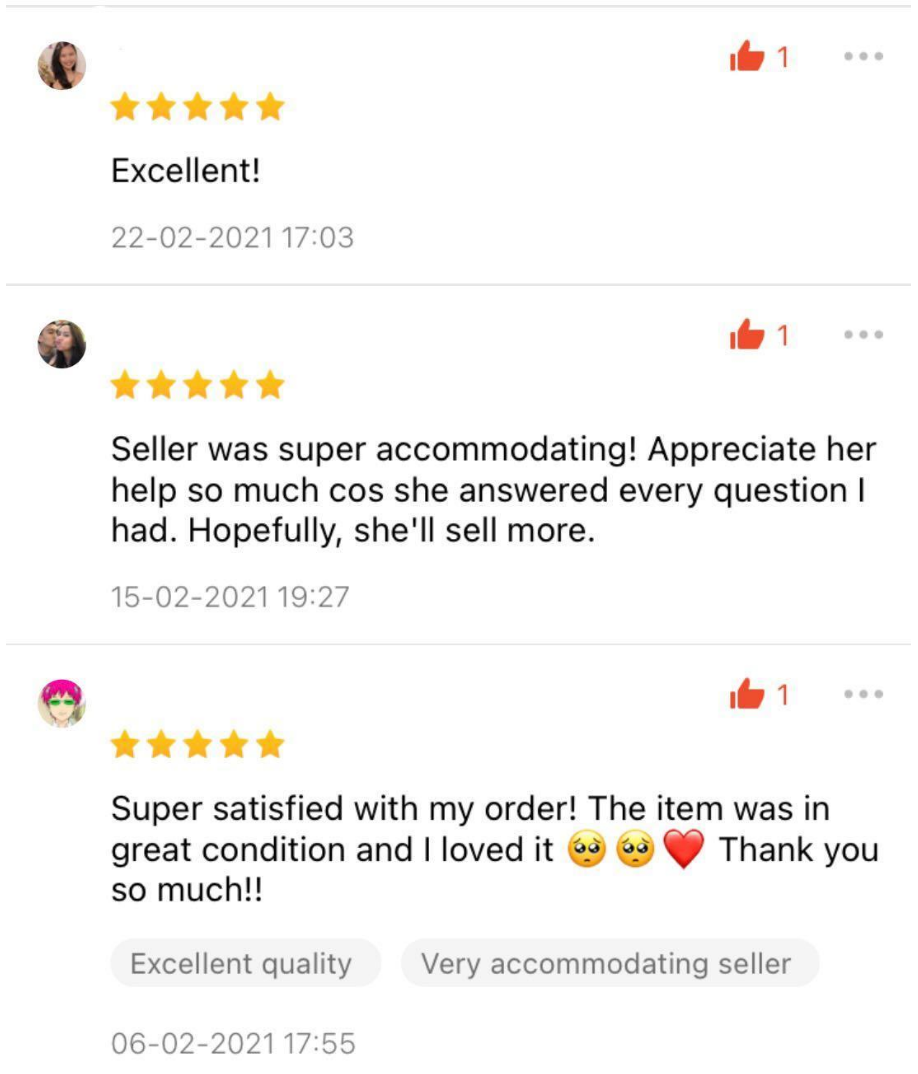
Output Neurons

- For **regression** tasks, use only one output neuron with **no activation function** (sometimes also called a **linear activation function**)



This neuron can output any real number value, which now represents our prediction.

Output Neurons



★★★★★

Excellent!

22-02-2021 17:03

★★★★★

Seller was super accommodating! Appreciate her help so much cos she answered every question I had. Hopefully, she'll sell more.

15-02-2021 19:27

★★★★★

Super satisfied with my order! The item was in great condition and I loved it 🥺🥺❤️ Thank you so much!!

Excellent quality | Very accommodating seller

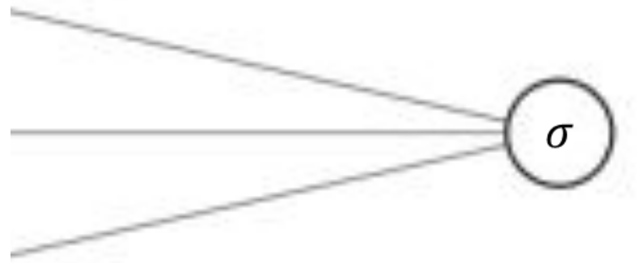
06-02-2021 17:55

- **Goal:** We want to make a system that gets the review text of a comment and classify whether it is a positive review or a negative review.

- How many output neurons should there be?

Output Neurons

- For **binary classification** tasks, use only one output neuron with **sigmoid activation function**)



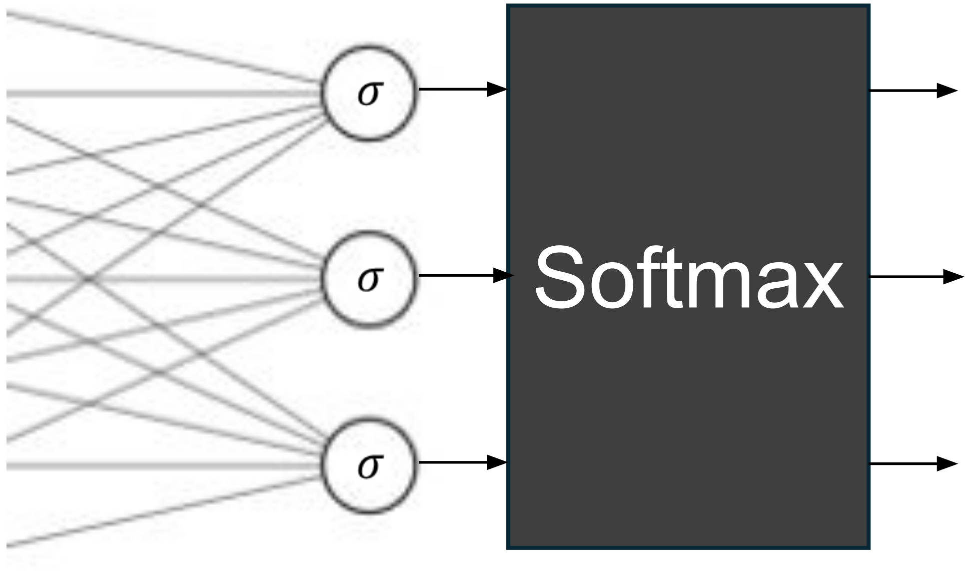
This neuron can output a value between 0 and 1, and can be interpreted as the predicted **probability that the instance belongs to the positive class**

Output Neurons

- **Goal:** We want to make a system that can automatically detect if an animal image is a photo of a mammal, reptile, or bird.
- How many output neurons should there be?

Output Neurons

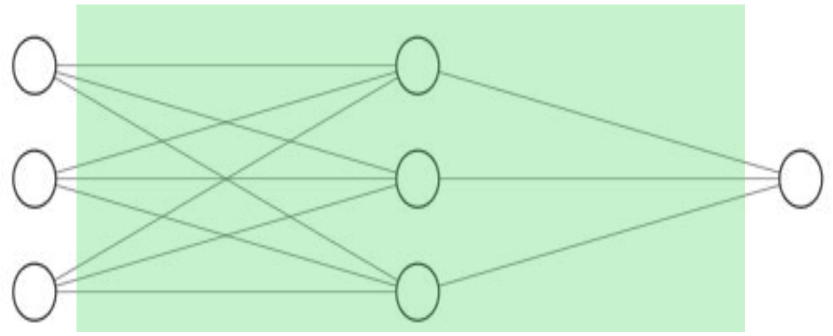
- For **multiclass classification** tasks, use k output neurons with **sigmoid activation functions**, then apply a **softmax layer** after that (k is the number of classes).



This setup will output k values representing the predicted **probability distribution** across the k classes.

Designing the NN Architecture

- How do you decide:
  - The number of input neurons?
  - The number of hidden layers?
  - The number of neurons in each hidden layer?
  - The activation function in each hidden layer?
  - The number of output neurons?
  - The activation function of the output layer?



Hidden Layers

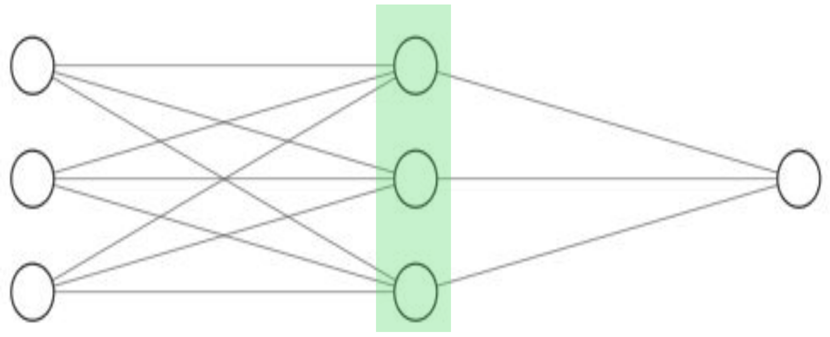
- No universal way to determine the best number of hidden layers and the number of neurons in each hidden layer.

- Need to try out different combinations and see what works best (**hyperparameter tuning**)!

- Things to keep in mind:
  - Start with something simple
  - Look at related literature on the domain you are working on to get a good starting point

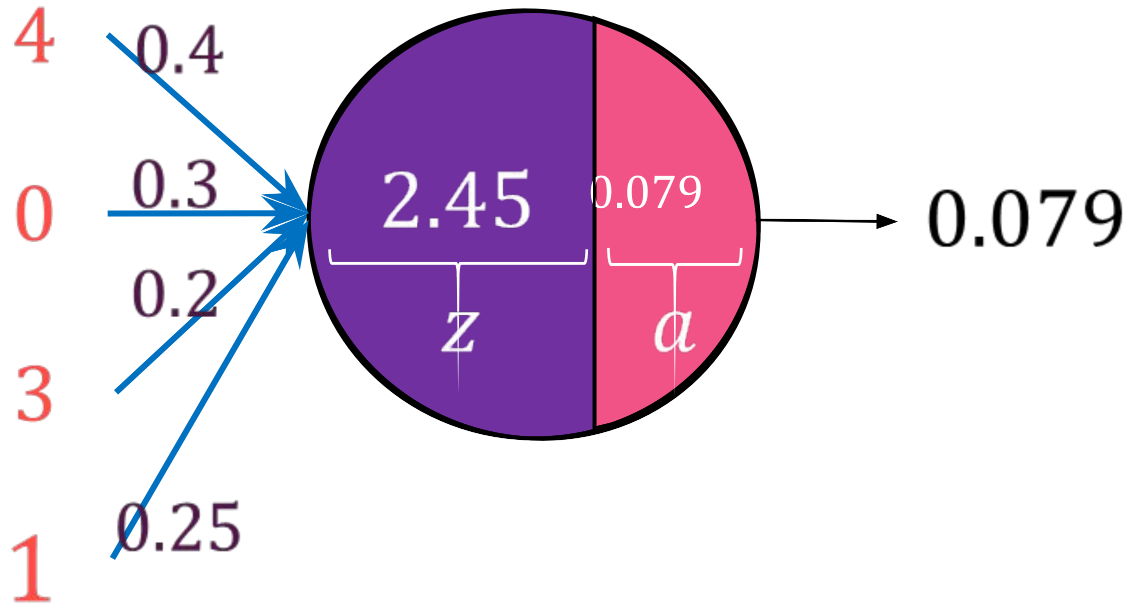
Designing the NN Architecture

- How do you decide:
  - The number of input neurons?
  - The number of hidden layers?
  - The number of neurons in each hidden layer?
  - The activation function in each hidden layer?
  - The number of output neurons?
  - The activation function of the output layer?



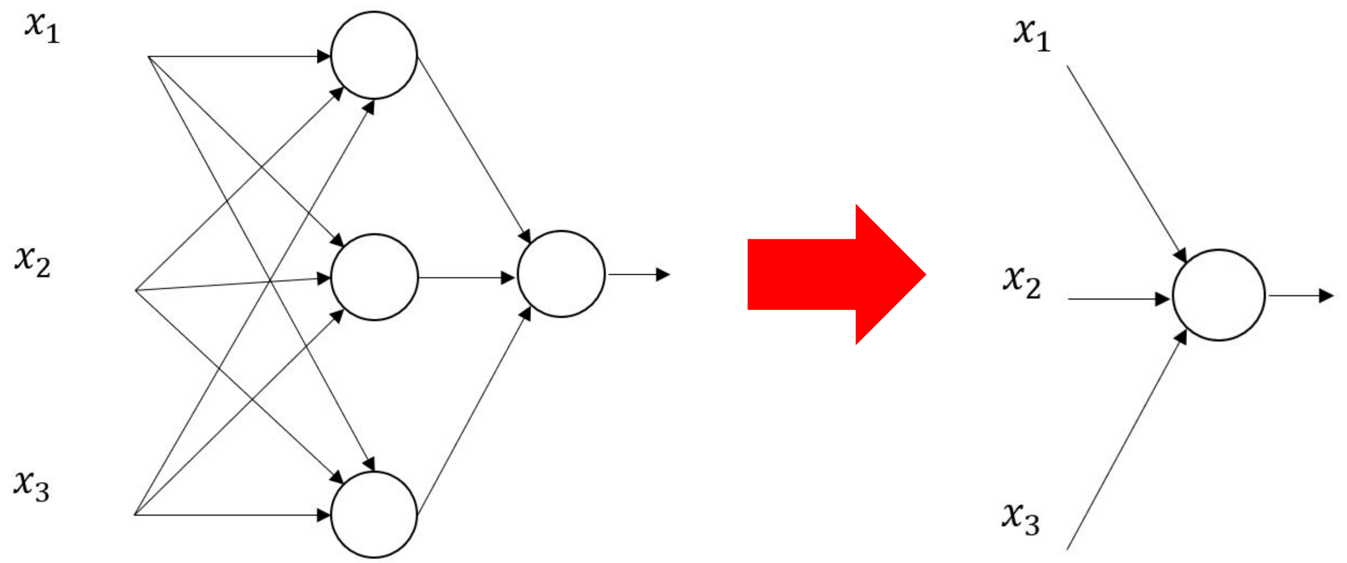
Activation Functions / Non-Linearities

- Determines how the a is computed
  - Sigmoid function (what we have been using so far)
  - Tanh function
  - RelU function
  - and others…



Why Activation is Neeeded?

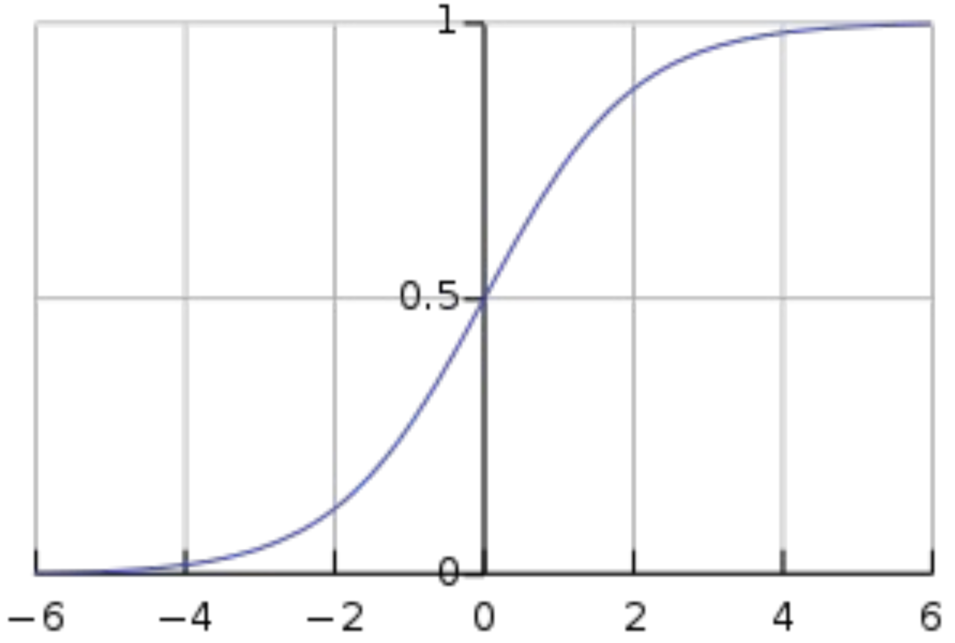
- If we don't use activation function, also referred to as "linear activation function", then mathematically, the network **devolves into simple linear regression**.



- Activation functions provide the **non-linearity**!

Sigmoid Function

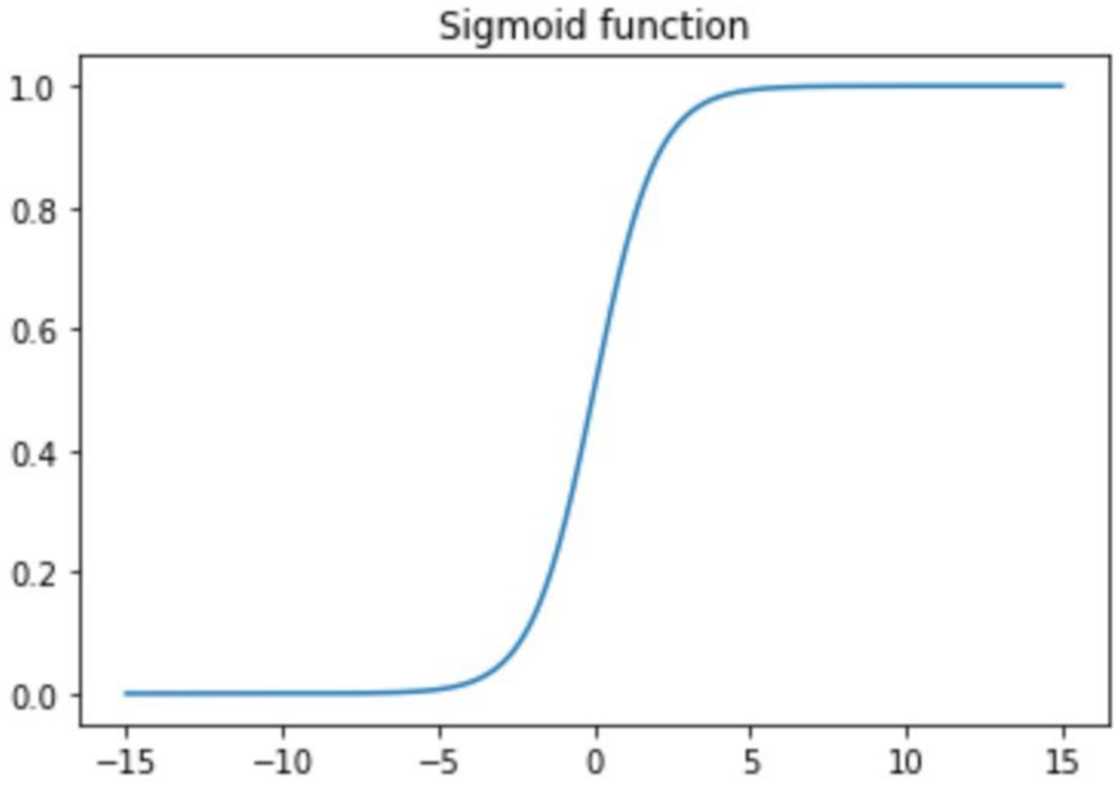
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

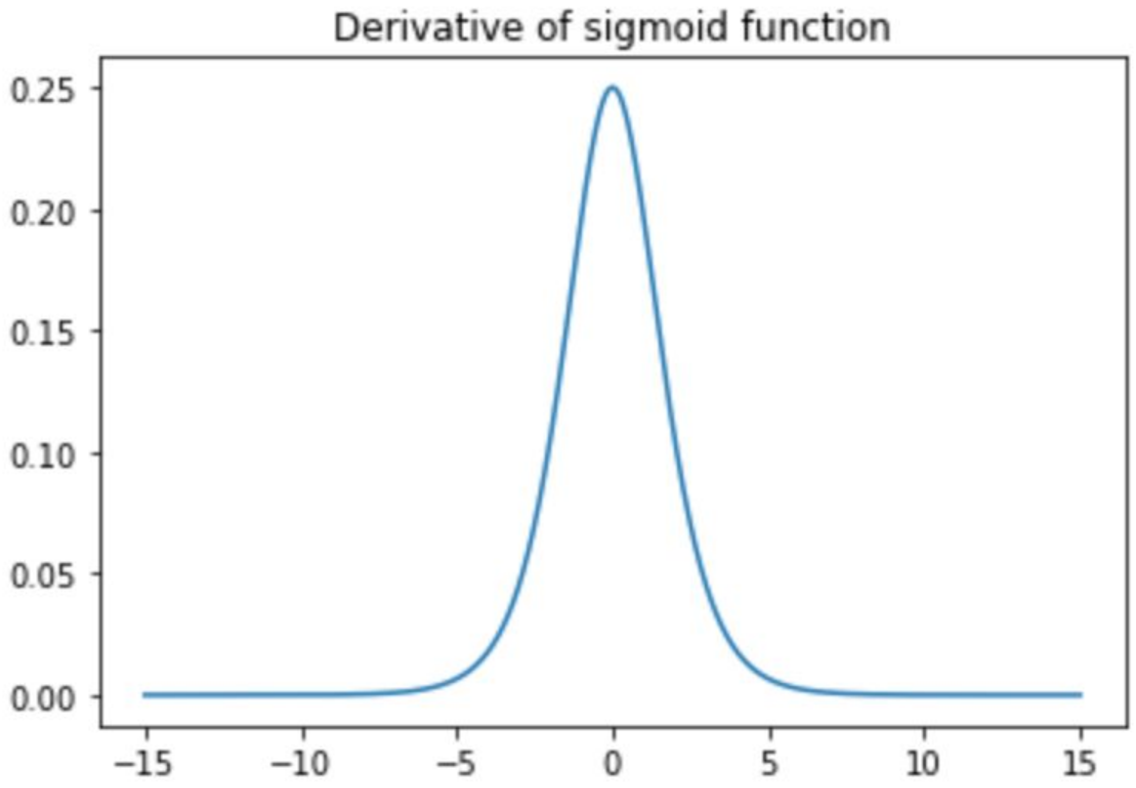


Range: $(0,1)$

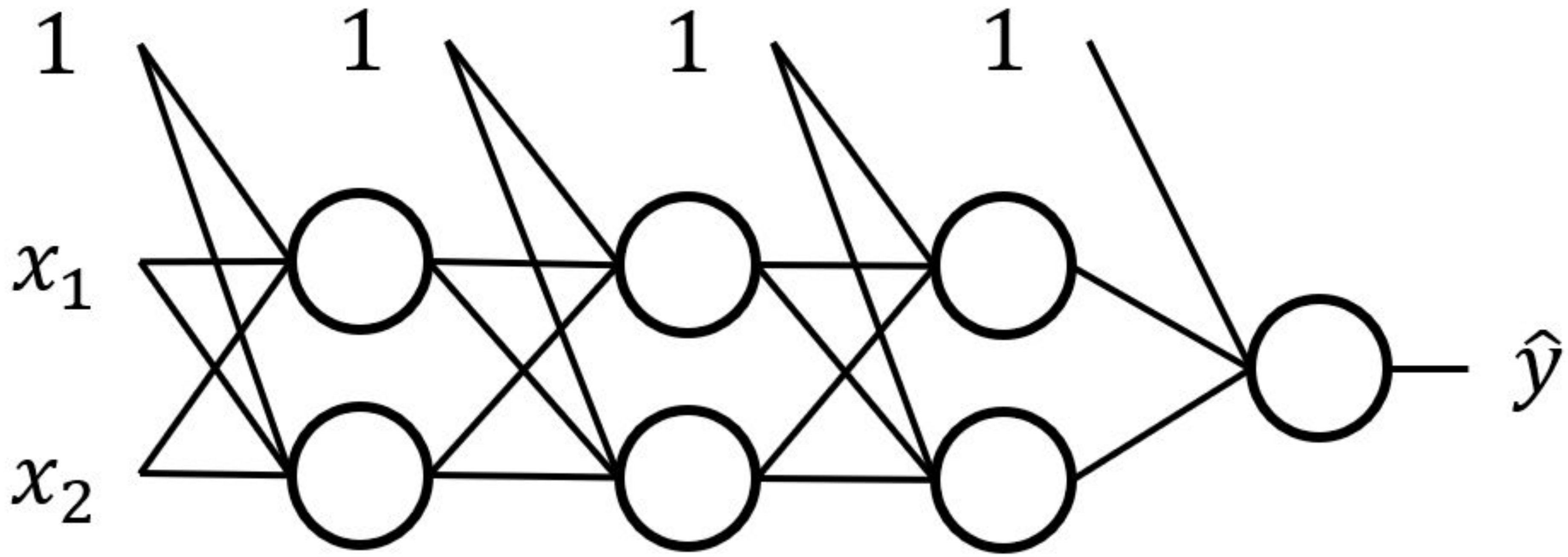
- Problems:
  - Saturated near 0 or 1
    - Saturation “kills” the neurons
  - e^{-x} is expensive to compute
  - Outputs are not zero-centered

Sigmoid Function





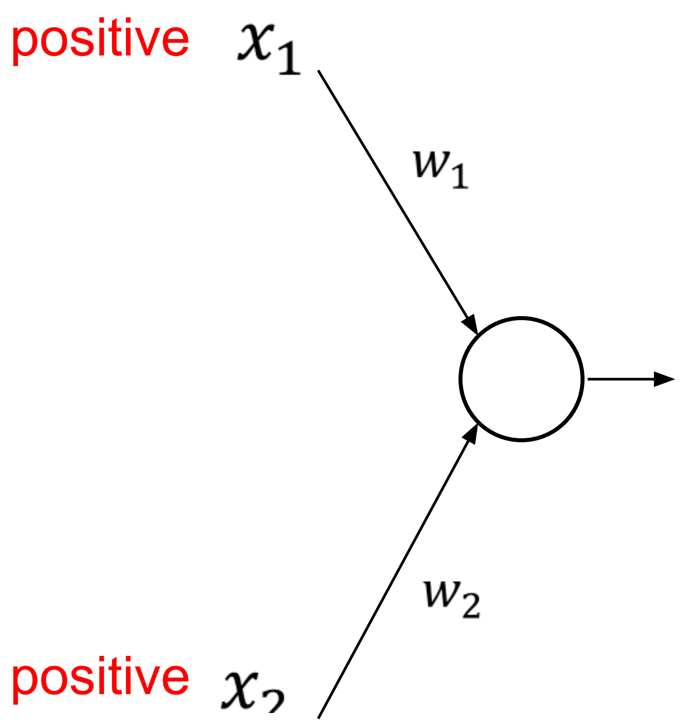
The “Vanishing Gradients” Problem



- For neurons with outputs close to 0 or 1, the gradient is a small value, which is reduced further when chain rule is applied, causing the network to have very minimal adjustments and “learning”.

Zero-Centered Outputs

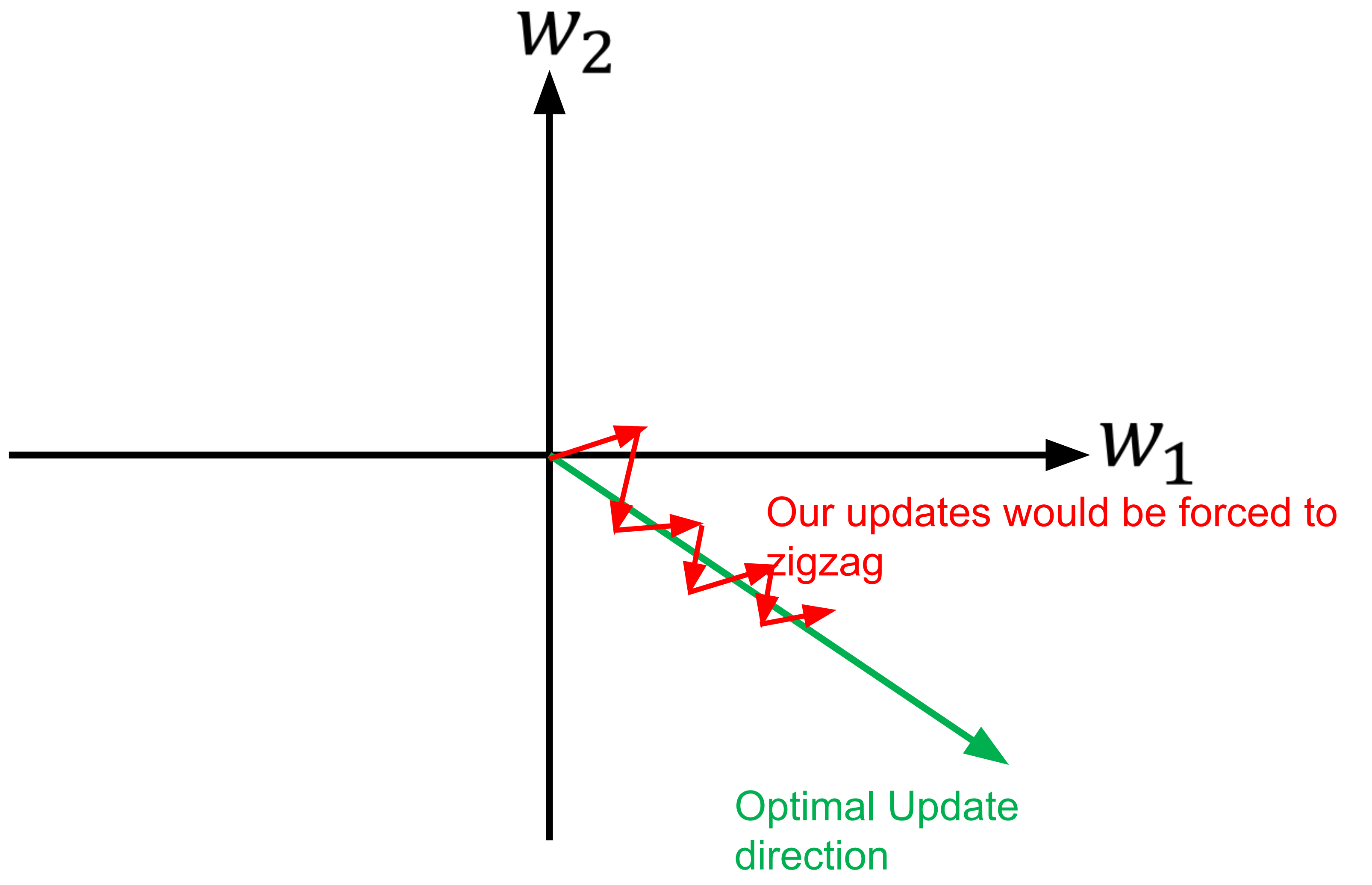
- If the inputs to a neuron are all positive, the gradients of the weights will either be **all positive or all negative**.



- Therefore, two things can happen in gradient descent:
  - All weights will increase
  - All weights will decrease

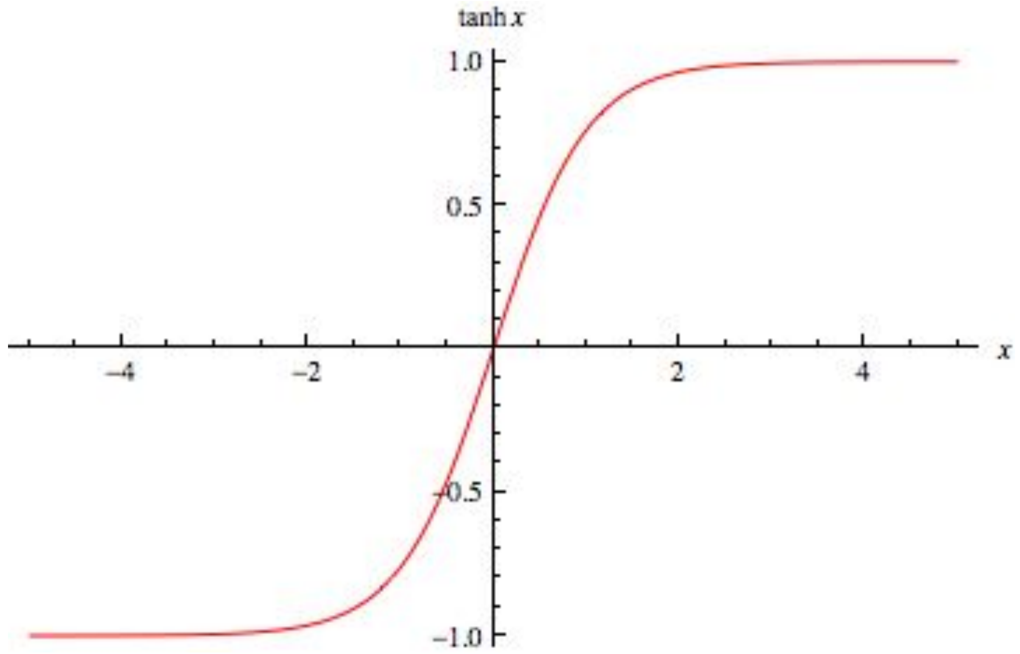
- Why is this a problem?

Zero-Centered Outputs



Tanh Function

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

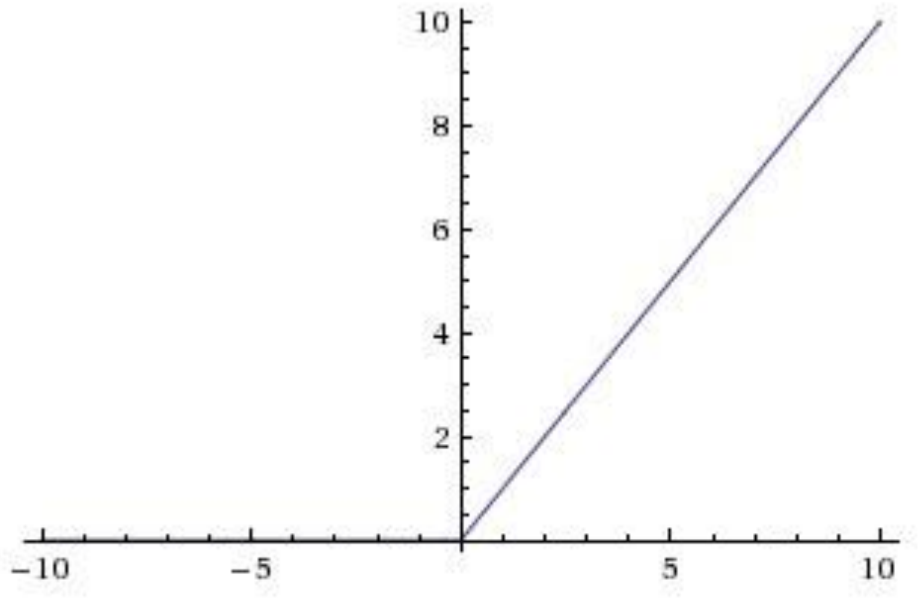


Range: $(-1,1)$

- Problems:
  - Saturated near 0 or 1
    - Saturation "kills" the neurons
  - e^{-x} is expensive to compute
- Advantages
  - Outputs are zero-centered

ReLU Function

$$\mathrm{ReLU}(x) = \max(0, x)$$

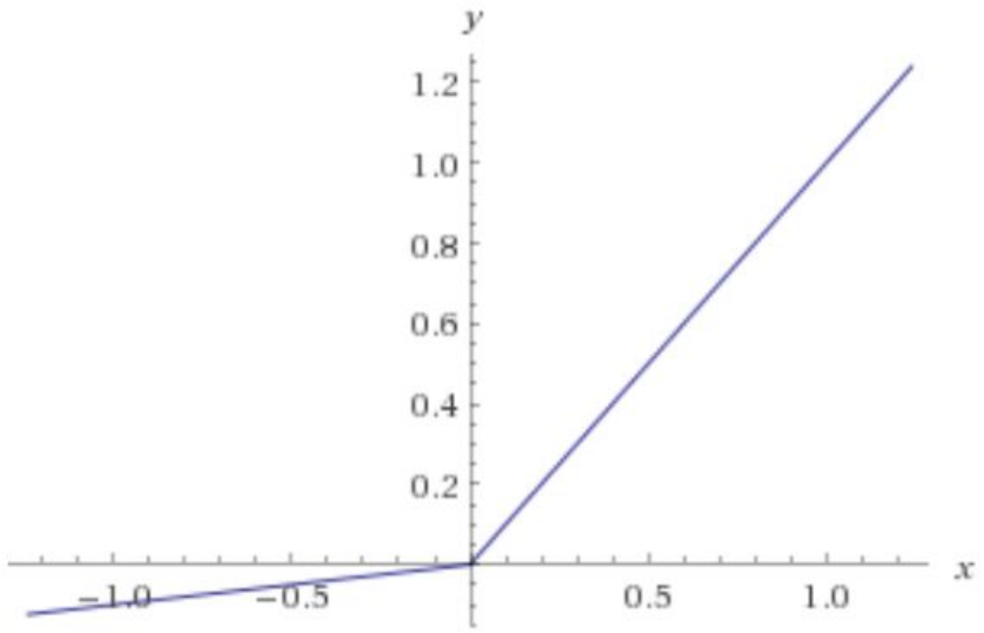


Range: $[0, \infty)$

- Problems:
  - Saturated on one side
    - Saturation “kills” the neurons
  - Outputs are not zero-centered
- Advantages
  - Saturated only on one side
  - Faster to compute

Leaky ReLU Function

$$f(x) = \max(0.01x, x)$$



Parametric Rectifier (PReLU):

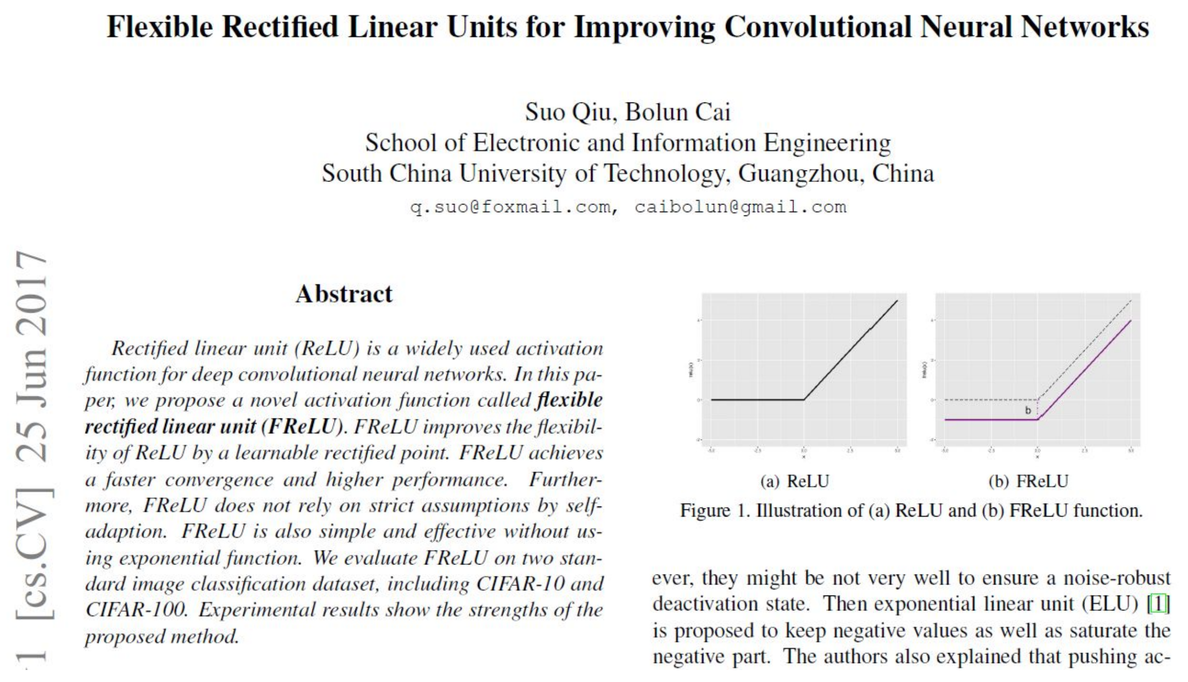
$$f(x) = \max(\alpha x, x)$$

- Problems:
  - Saturated on one side
    - Saturation “kills” the neurons
  - Outputs are not zero-centered, but a bit better
- Advantages
  - Saturated only on one side
  - Faster to compute
  - Converges faster

In Practice

- Use ReLU as the default activation function.
- Works well in most problems.
- Be careful with learning rates.
- Sigmoid is almost never used in practice.

Other Proposed Activations



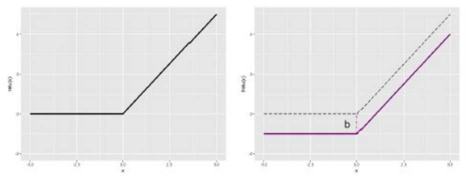
1 [cs.CV] 25 Jun 2017

Flexible Rectified Linear Units for Improving Convolutional Neural Networks

Suo Qiu, Bolun Cai
School of Electronic and Information Engineering
South China University of Technology, Guangzhou, China
q.suo@foxmail.com, caibolun@gmail.com

Abstract

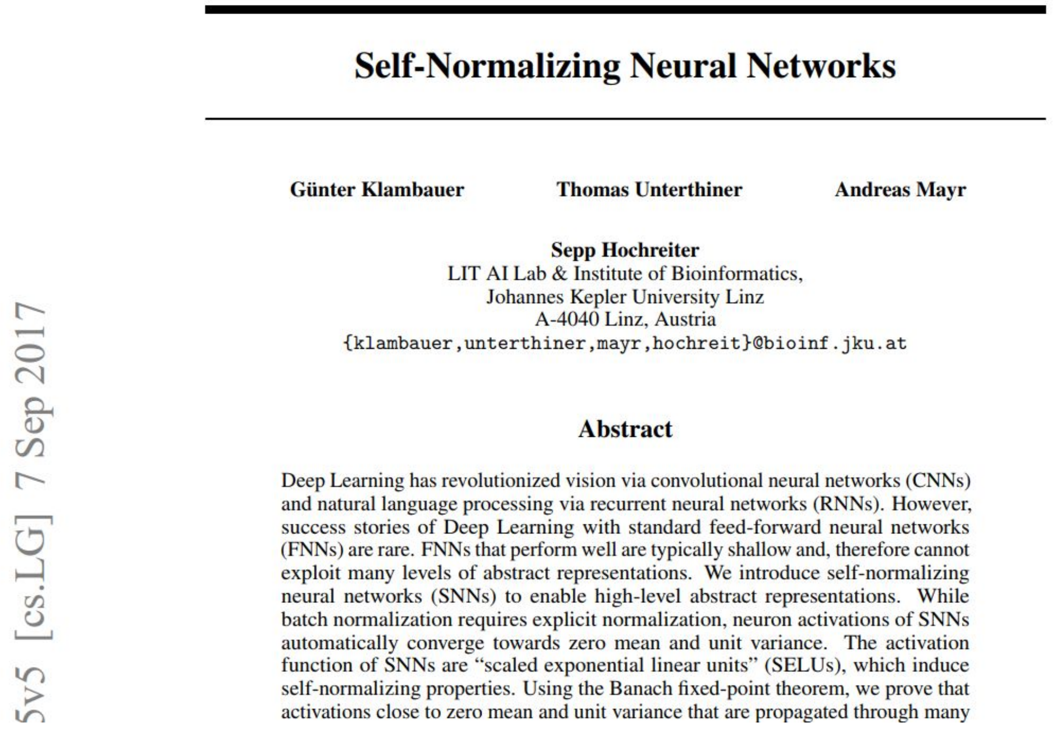
*Rectified linear unit (ReLU) is a widely used activation function for deep convolutional neural networks. In this paper, we propose a novel activation function called **flexible rectified linear unit (FReLU)**. FReLU improves the flexibility of ReLU by a learnable rectified point. FReLU achieves a faster convergence and higher performance. Furthermore, FReLU does not rely on strict assumptions by self-adaption. FReLU is also simple and effective without using exponential function. We evaluate FReLU on two standard image classification dataset, including CIFAR-10 and CIFAR-100. Experimental results show the strengths of the proposed method.*



(a) ReLU (b) FReLU

Figure 1. Illustration of (a) ReLU and (b) FReLU function.

ever, they might be not very well to ensure a noise-robust deactivation state. Then exponential linear unit (ELU) [1] is proposed to keep negative values as well as saturate the negative part. The authors also explained that pushing ac-



5v5 [cs.LG] 7 Sep 2017

Self-Normalizing Neural Networks

Günter Klambauer **Thomas Unterthiner** **Andreas Mayr**

Sepp Hochreiter
LIT AI Lab & Institute of Bioinformatics,
Johannes Kepler University Linz
A-4040 Linz, Austria
{klambauer,unterthiner,mayr,hochreit}@bioinf.jku.at

Abstract

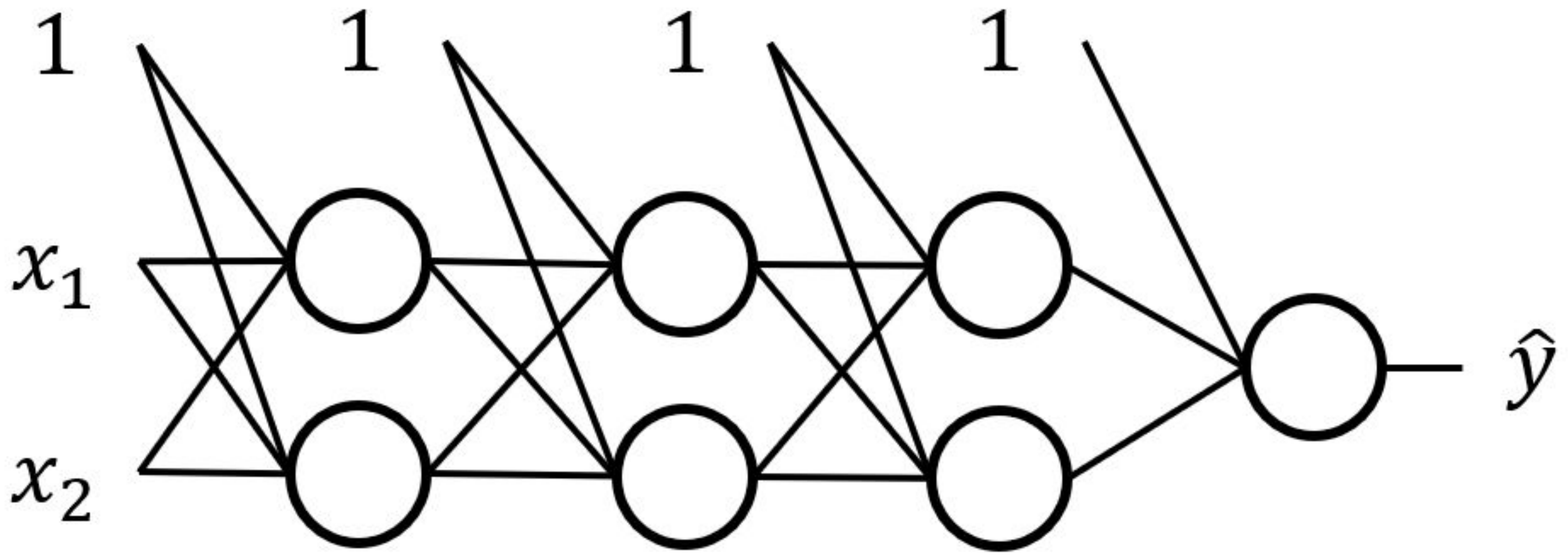
Deep Learning has revolutionized vision via convolutional neural networks (CNNs) and natural language processing via recurrent neural networks (RNNs). However, success stories of Deep Learning with standard feed-forward neural networks (FNNs) are rare. FNNs that perform well are typically shallow and, therefore cannot exploit many levels of abstract representations. We introduce self-normalizing neural networks (SNNs) to enable high-level abstract representations. While batch normalization requires explicit normalization, neuron activations of SNNs automatically converge towards zero mean and unit variance. The activation function of SNNs are "scaled exponential linear units" (SELUs), which induce self-normalizing properties. Using the Banach fixed-point theorem, we prove that activations close to zero mean and unit variance that are propagated through many

Other Considerations in Setting Up NN

- Weights Initialization
- Regularization
  - Batch Normalization
  - Dropout
- More Sophisticated Optimization / Learning Algorithms

Weights Initialization

- **Why does it matter?**
  - What happens if we initialize all weights to the same value (e.g., 0)?



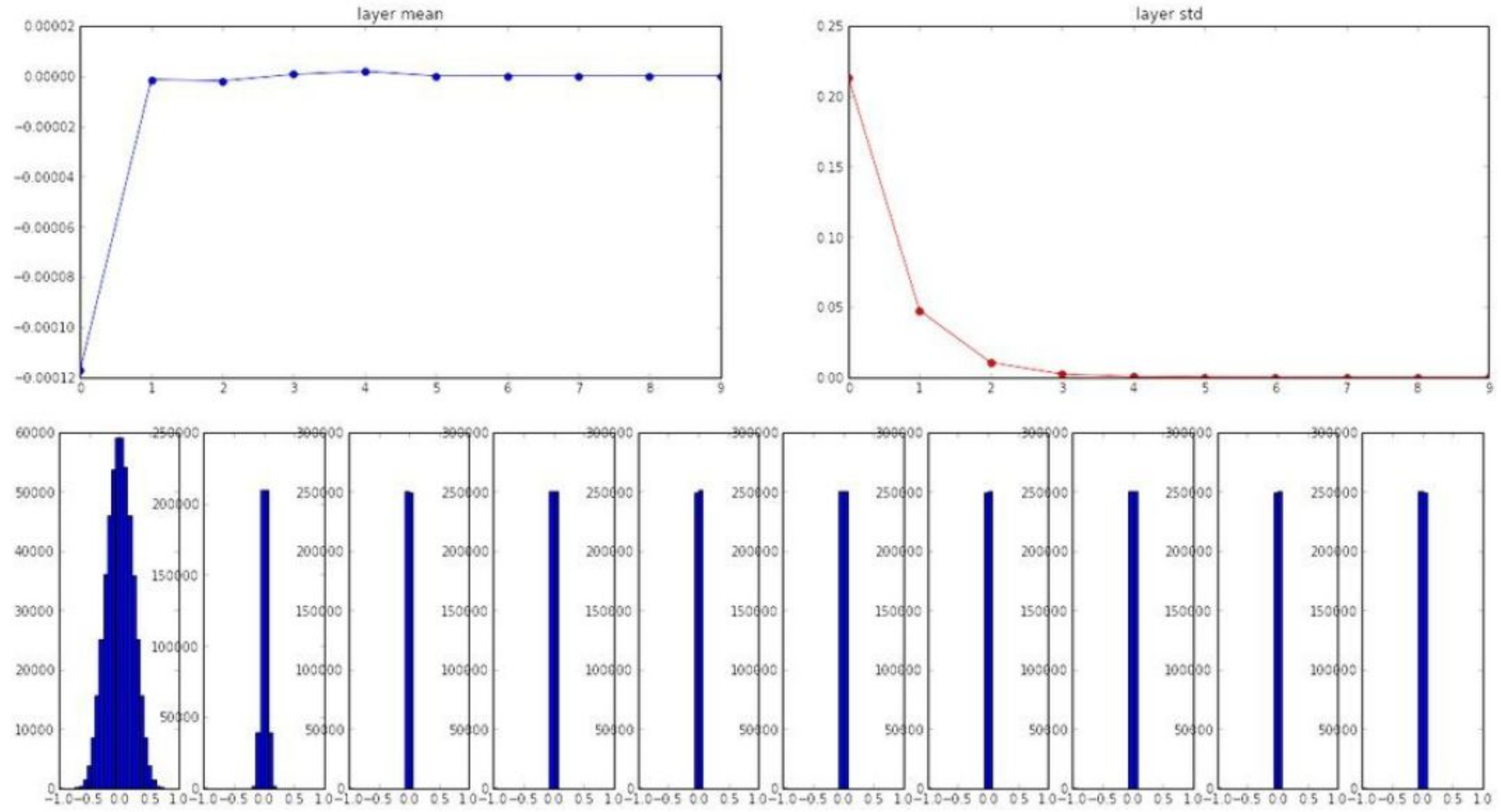
Weights Initialization

- Weights are normally initialized with small random numbers.
  - Gaussian with $\mu = 0$ and $\sigma = 0.01$

- This is okay for small networks but can lead to **non-homogeneous activations** for deeper networks.

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```

- Gaussian distribution
- $\mu = 0, \sigma = 0.01$
- Activation function: tanh

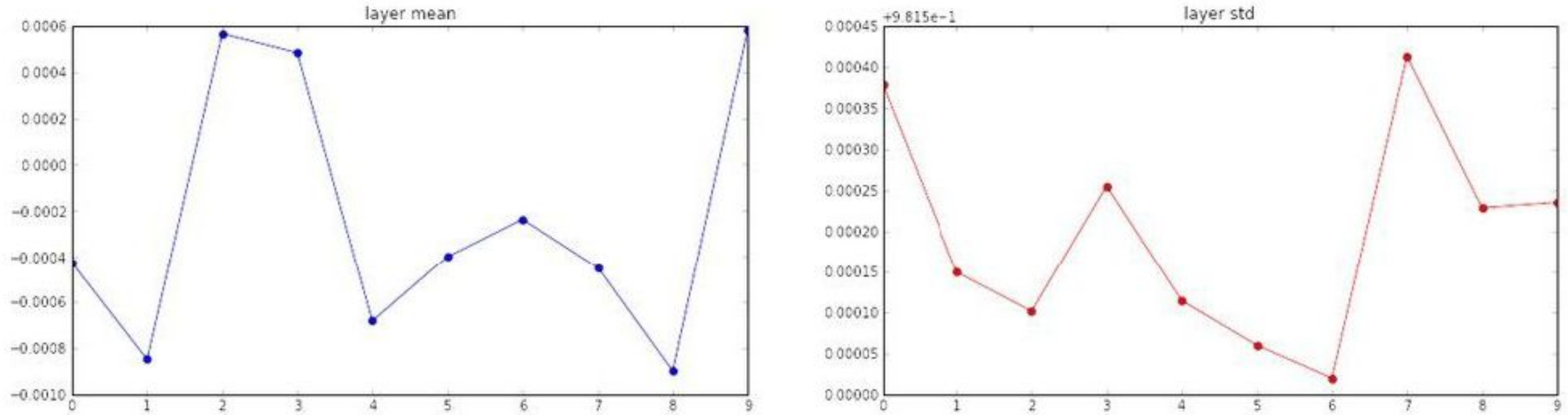


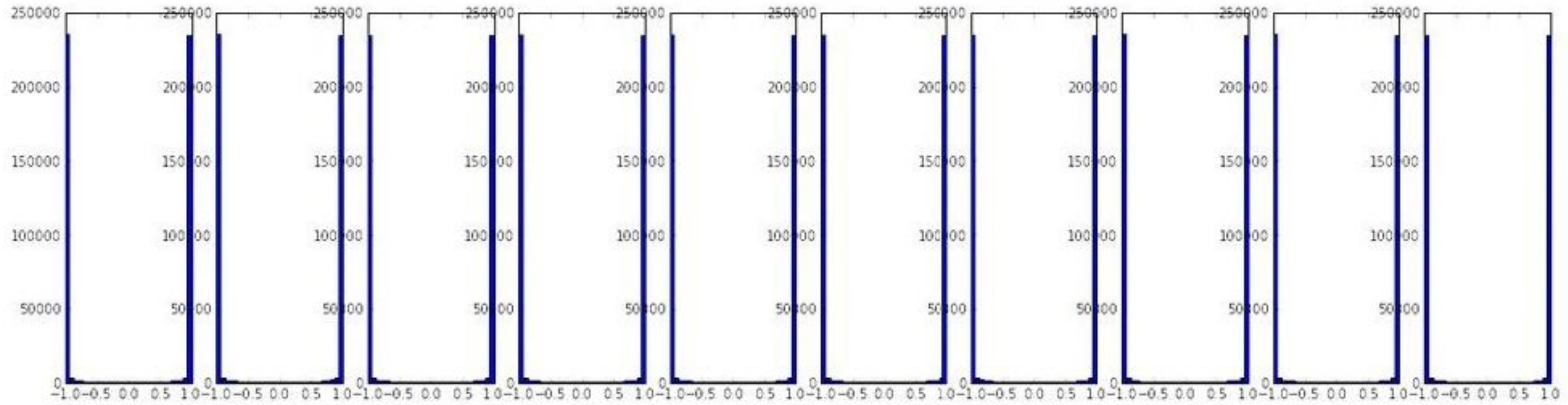
```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

*1.0 instead of *0.01

- Gaussian distribution
- $\mu = 0, \sigma = 1$
- Activation function: tanh





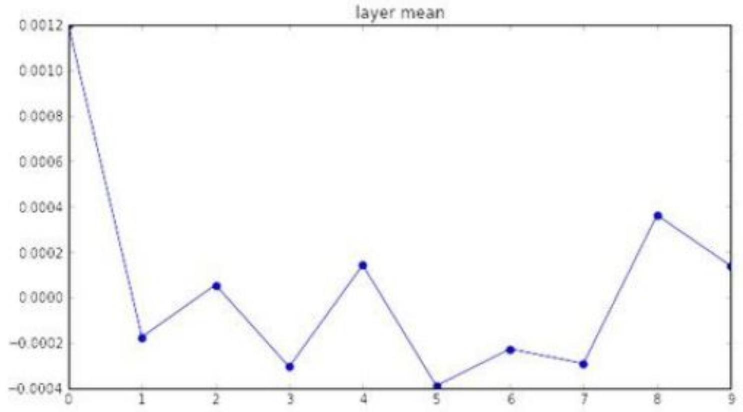
Weight initialization matters!

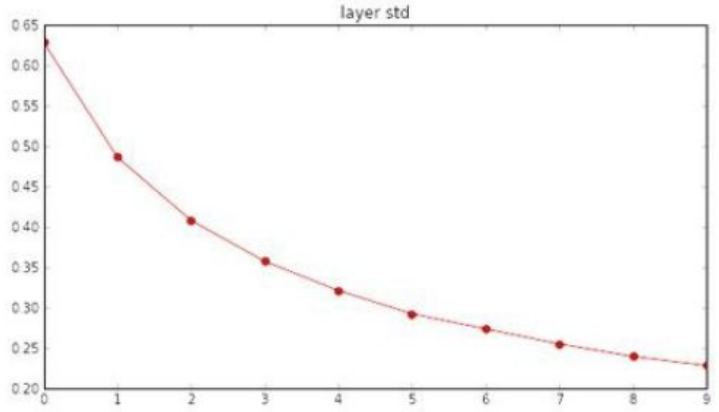
Xavier Initialization (Glorot et al.,

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

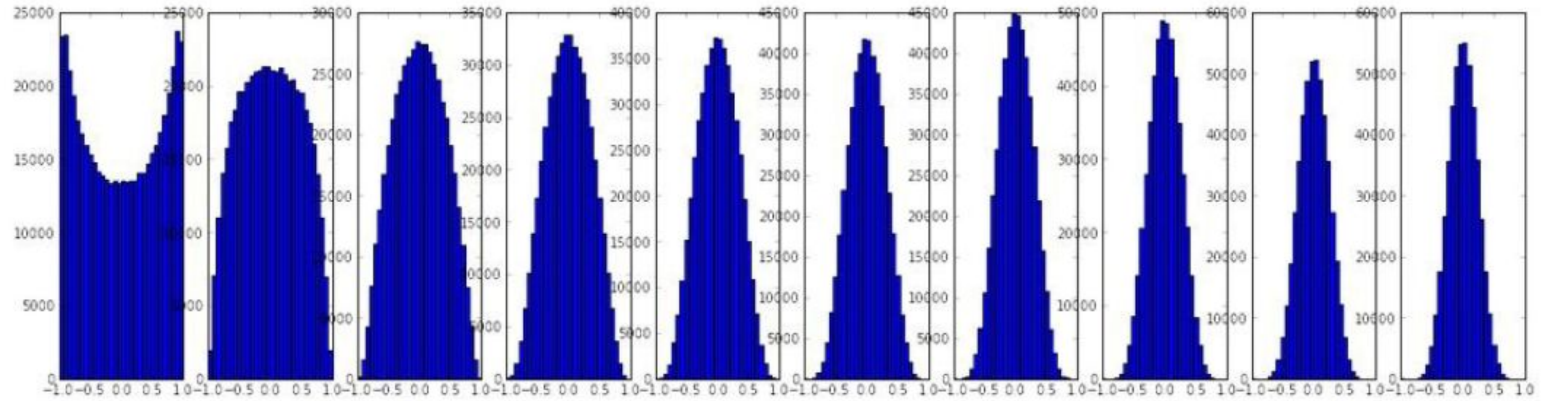
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

"Xavier initialization"
[Glorot et al., 2010]





Reasonable initialization.
(Mathematical derivation assumes linear activations)

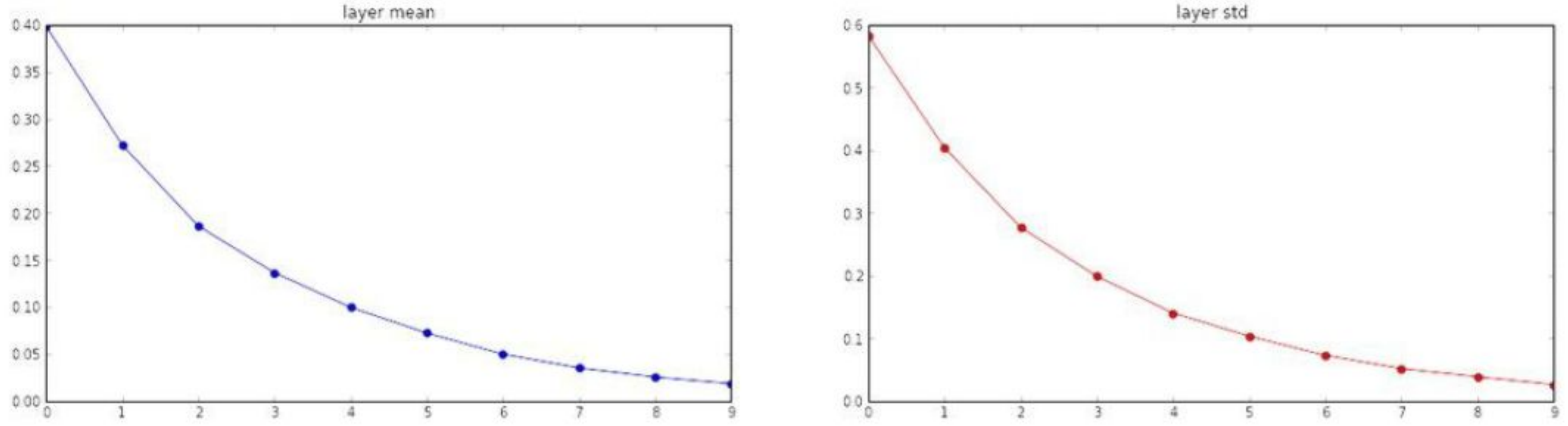


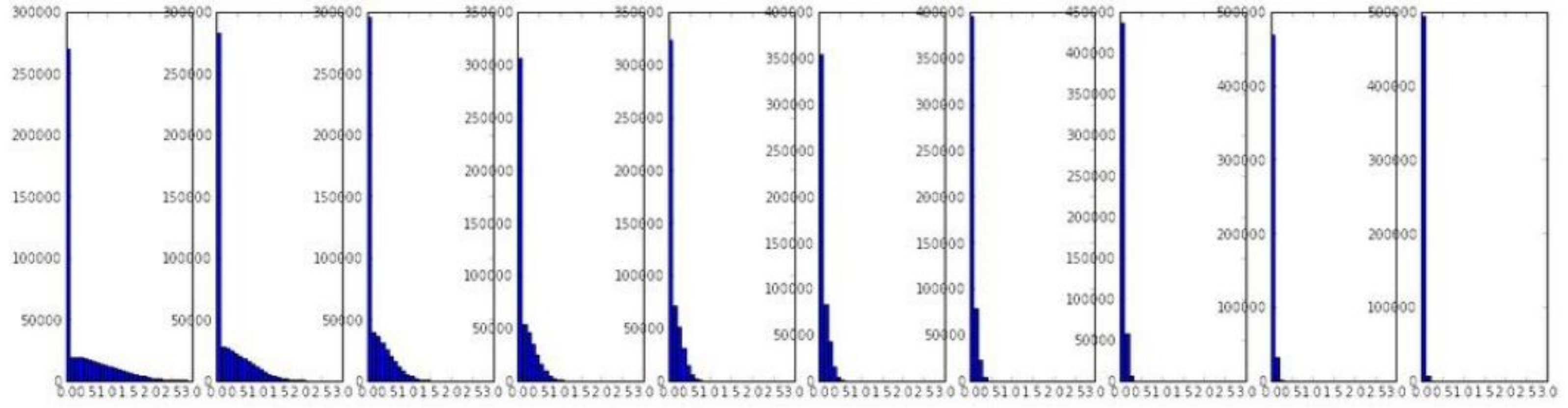
Xavier Initialization (Glorot et al.,

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

```python
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

but when using the ReLU nonlinearity it breaks.



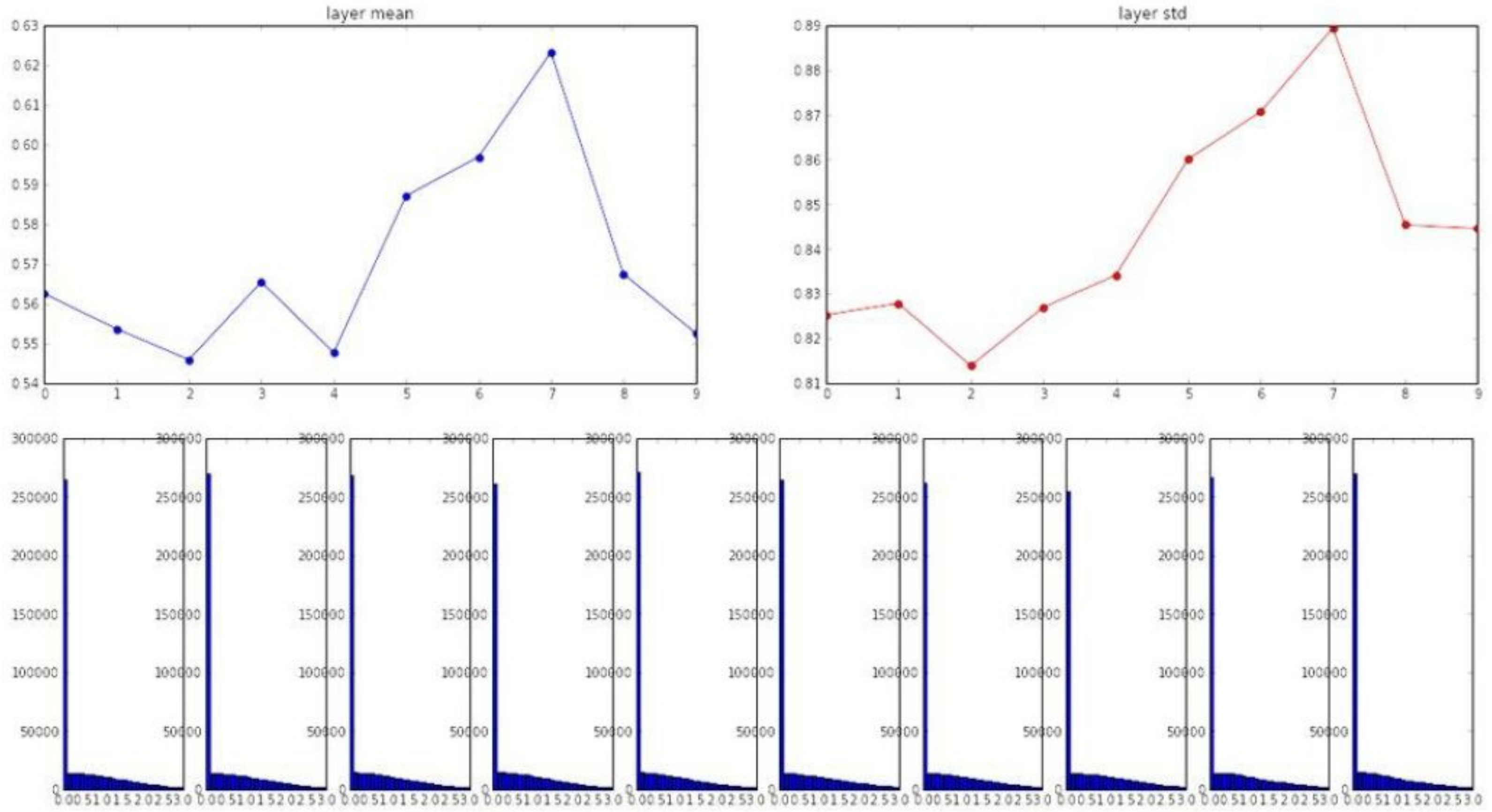


He Initialization (He. et al.. 2015)

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

He et al., 2015
(note additional /2)



Proper Initialization is Still An Active Area of Research

- Many studies on various schemes of initialize weights
- Bottom line: be aware that weight initialization can be factor to the ability of the network to “learn”.
- Ultimately, we want networks to learn and converge with few iterations

Regularization in Neural Networks

- Regularization: methods to **prevent a machine learning model from overfitting**

- Two common methods for neural networks
  - Batch normalization
  - Dropout

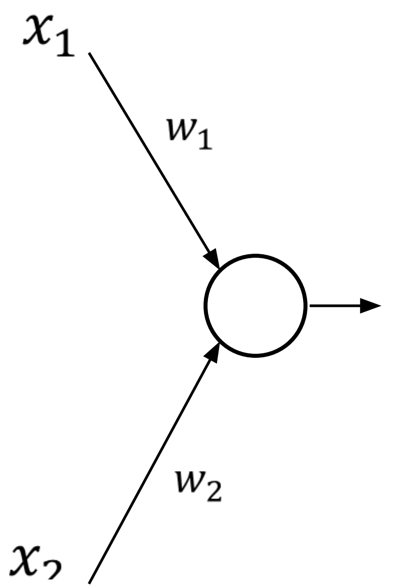
Batch Normalization

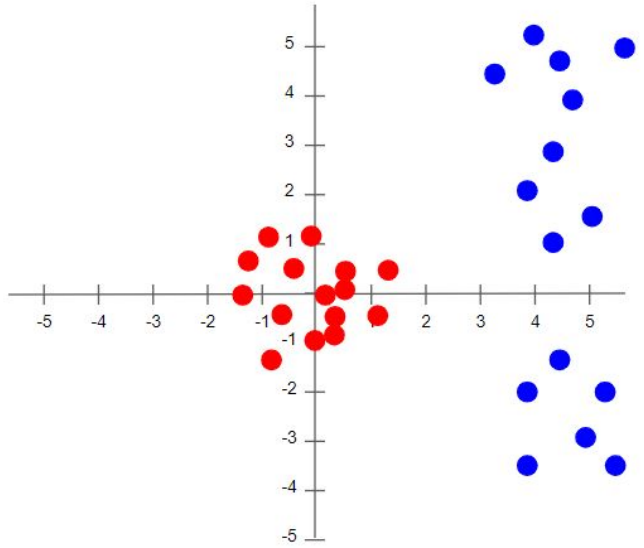
- Intuition: Normalize the outputs in each hidden layer to make the network converge faster.

- Why does normalization matter?

Batch Normalization

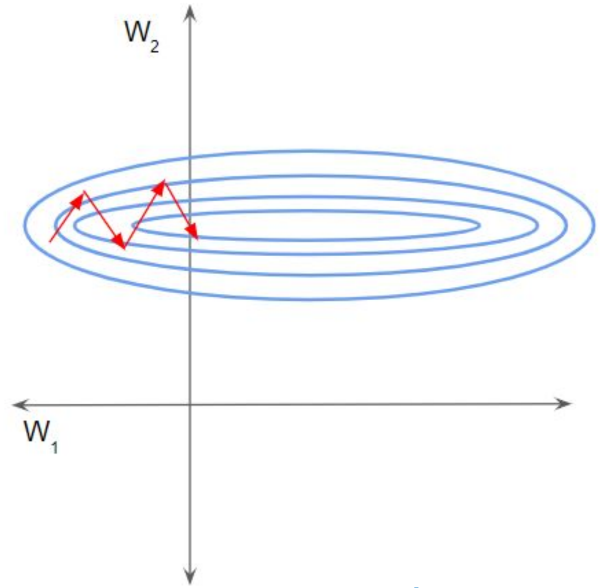
- If features are on the same scale, it helps the network converge faster



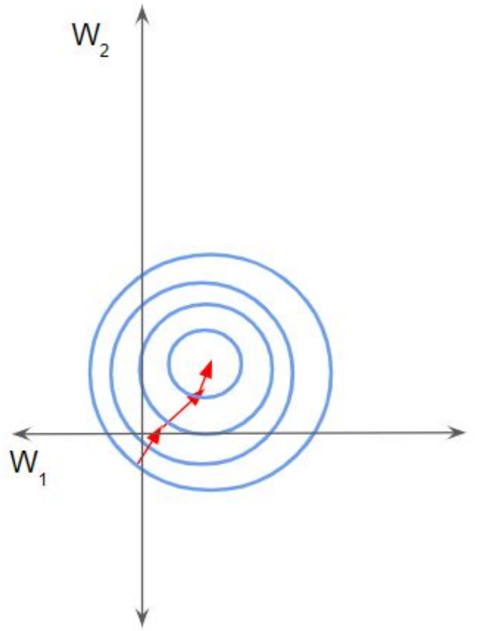


Blue: original data

Red: normalized data



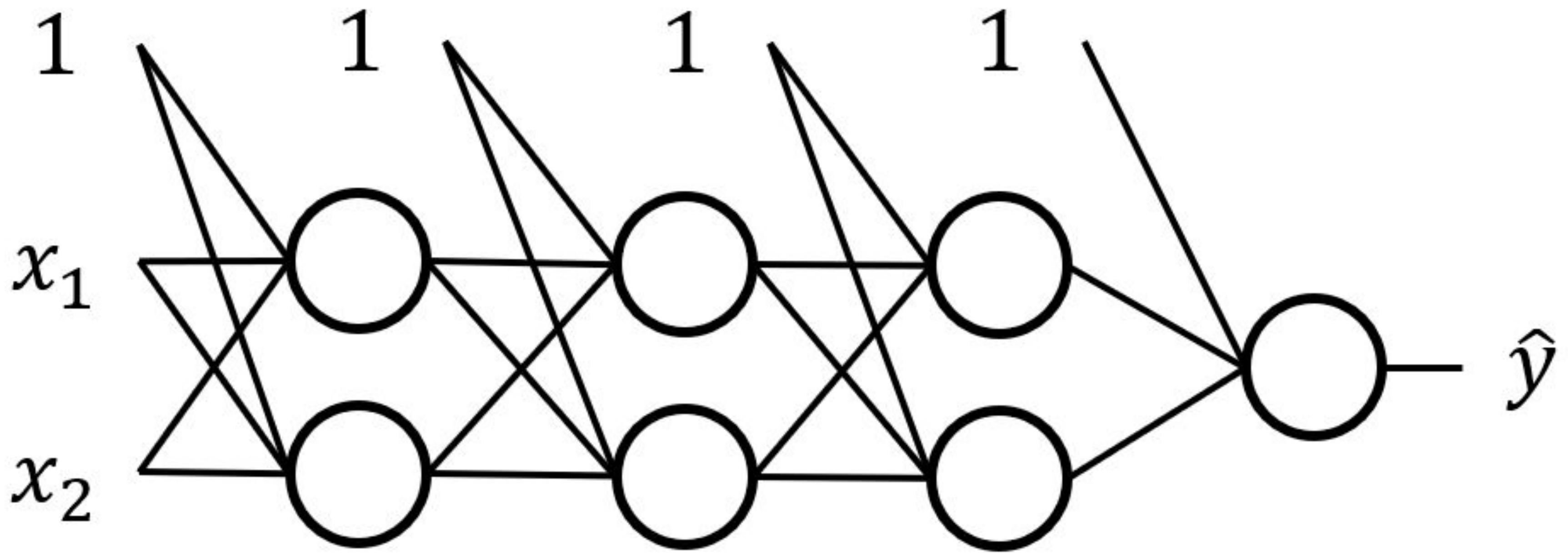
Loss landscape (original)



Loss landscape (normalized)

Batch Normalization

- Since hidden layers get their input from the previous layer, it makes sense to normalize those as well!



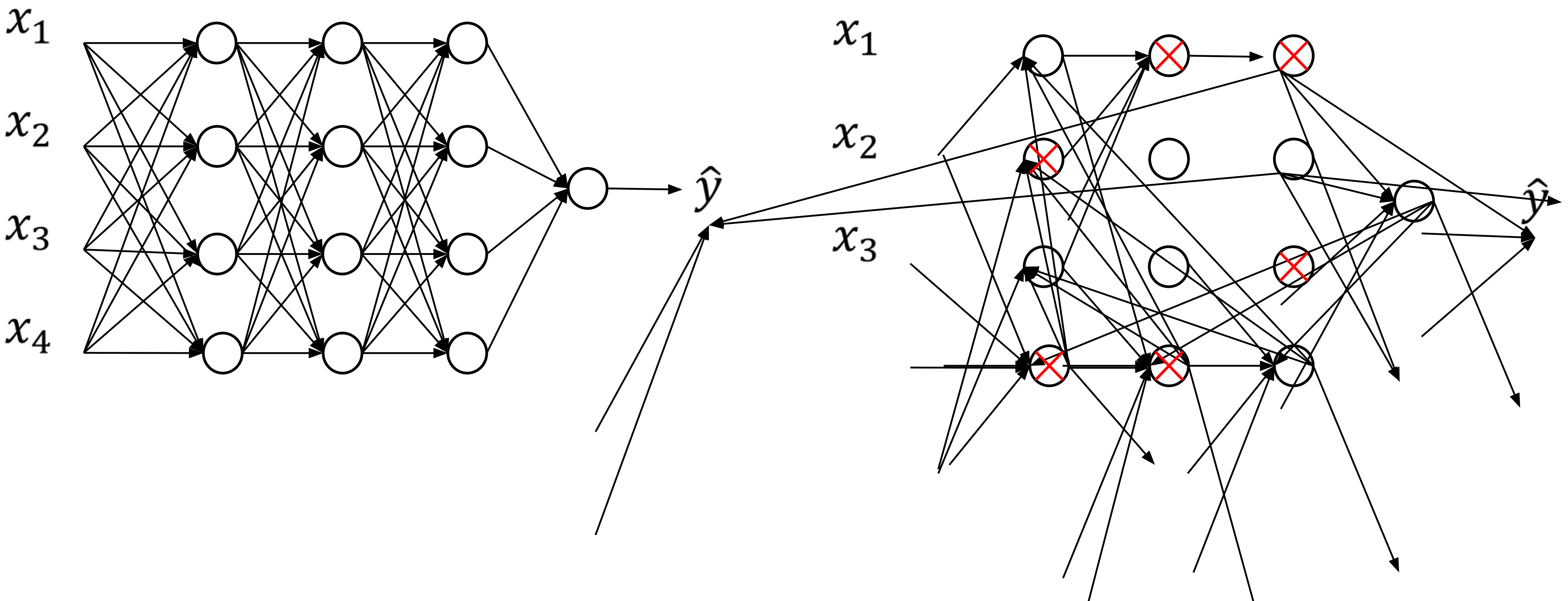
Batch Normalization

- What does the BN do?
  - For each batch,
    - Compute the mean and variance and normalize the values
      - $\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$
    - Allow the network to "squash" the range (i.e., change the mean and variance) to lower the loss
      - $y^{(k)} = \widehat{\gamma^{(k)}}\, \hat{x}^{(k)} + \beta^{(k)}$

    - Note: γ and β for each batch norm layer are also parameters of the network (they will also be adjusted during gradient descent).\
    - For more information on batch norm, refer to:
      - https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

Dropout

- Intuition: Can't rely on a single feature, so we must spread out the weights



Dropout

- Implementation:
  - Set a hyperparameter **keep-prob**, the number of nodes to keep for each layer.
    - **keep-prob=1.0** means no nodes will be dropped out.
    - **keep-prob=0.25** means ¾ of the nodes will be dropped out.
  - For each iteration, a random number of nodes will be dropped (i.e., set to 0).

Optimization Algorithms

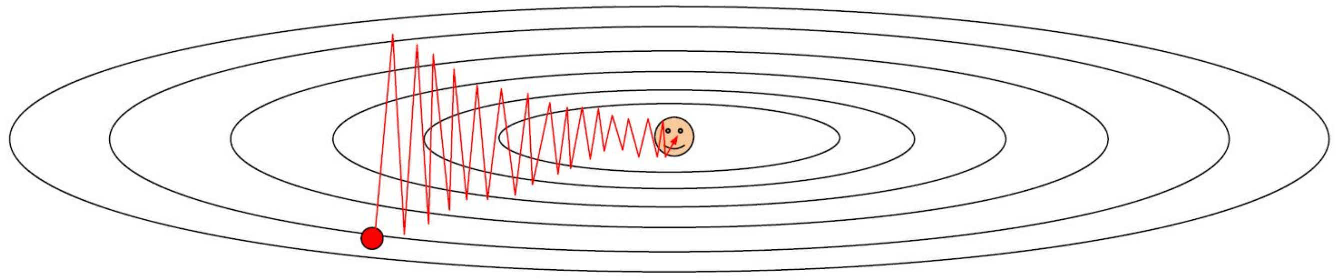
- Optimization algorithms (or learning algorithms) allow a machine learning model to minimize the cost function

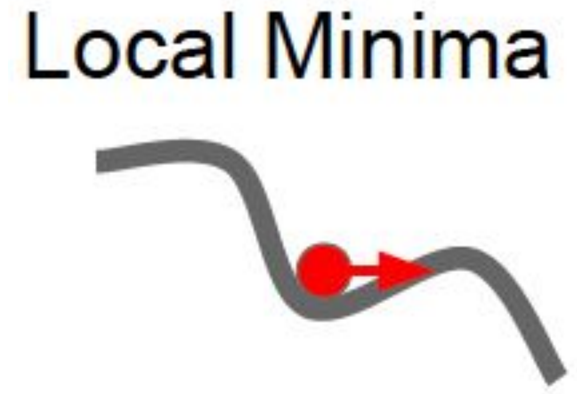
- One example that we have been using is **gradient descent**.

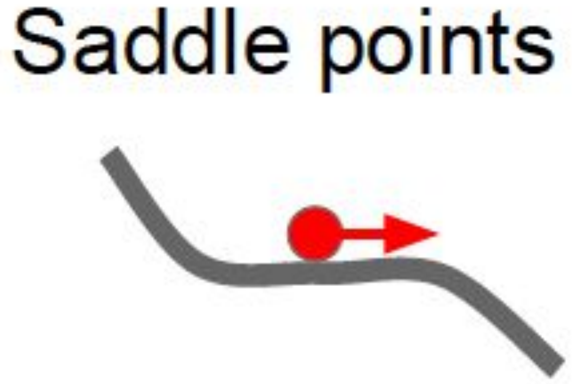
- A good optimization algorithm should:
  - Converge to an ideal minimum
  - Converge as quickly as possible

Problems with Vanilla Gradient Descent

Zigzagging on a certain parameter





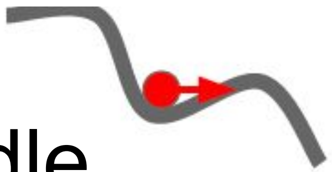


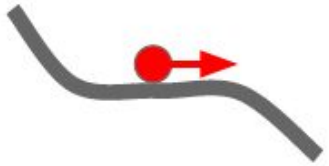
Gradient Descent Improvements

- Gradient Descent with Momentum
- Gradient Descent with RMSProp
- Adam Algorithm

Gradient Descent with Momentum

- How do we overcome saddle points and local minima? **Add momentum!**





$$V_{\partial\theta} = \beta V_{\partial\theta} + (1-\beta)\partial\theta$$

$$\theta = \theta - \alpha V_{\partial\theta}$$

	Without momentum	With momentum
1	100	100
2	100	100
3	1	90.1
4	100	91
5	1000	181.981

- More information about this in this video:
  - https://www.youtube.com/watch?v=k8fTYJPd3_I

In practice, $\beta = 0.9$ works well!

Gradient Descent with Momentum

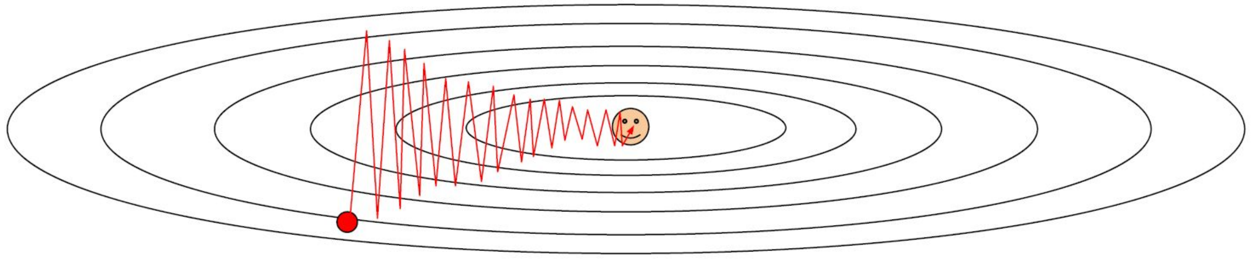
- A correction is sometimes added to account for the fact that the moving average is not very accurate at the beginning iterations.

$$\boxed{V_{\partial\theta} = \frac{\beta V_{\partial\theta} + (1-\beta)\partial\theta}{1-\beta^t}}$$

- More information about this in this video:
  - https://www.youtube.com/watch?v=lWzo8CajF5s

Gradient Descent with RMSProp

- How do we avoid zigzagging on certain parameters? **Put brakes when gradients are always large!**



$$\boxed{S_{\partial\theta} = \beta S_{\partial\theta} + (1-\beta)\partial\theta^2}$$

$$\boxed{\theta = \theta - \alpha \frac{\partial\theta}{\sqrt{S_{\partial\theta}}}}$$

If the squared gradients over the past iterations get too large, brake!

In practice, $\beta = 0.999$ works well!

- More information about this in this video:
  - www.youtube.com/watch?v=_e-LFe_igno

ADAM (Adaptive Moment Estimation)

$$V_{\partial\theta} = \frac{\beta_1 V_{\partial\theta} + (1 - \beta_1)\partial\theta}{1 - \beta_!^t}$$

momentum

first order optimization

$$S_{\partial\theta} = \frac{\beta_2 S_{\partial\theta} + (1 - \beta_2)\partial\theta^2}{1 - \beta_2^t}$$

rmsprop

second order optimization

$$\theta = \theta - \alpha \frac{V_{\partial\theta}}{\sqrt{S_{\partial\theta}} + \epsilon}$$

ADAM (Adaptive Moment Estimation)

$$V_{\partial\theta} = \frac{\beta_1 V_{\partial\theta} + (1-\beta_1)\partial\theta}{1-\beta_{!}^{t}}$$

momentum

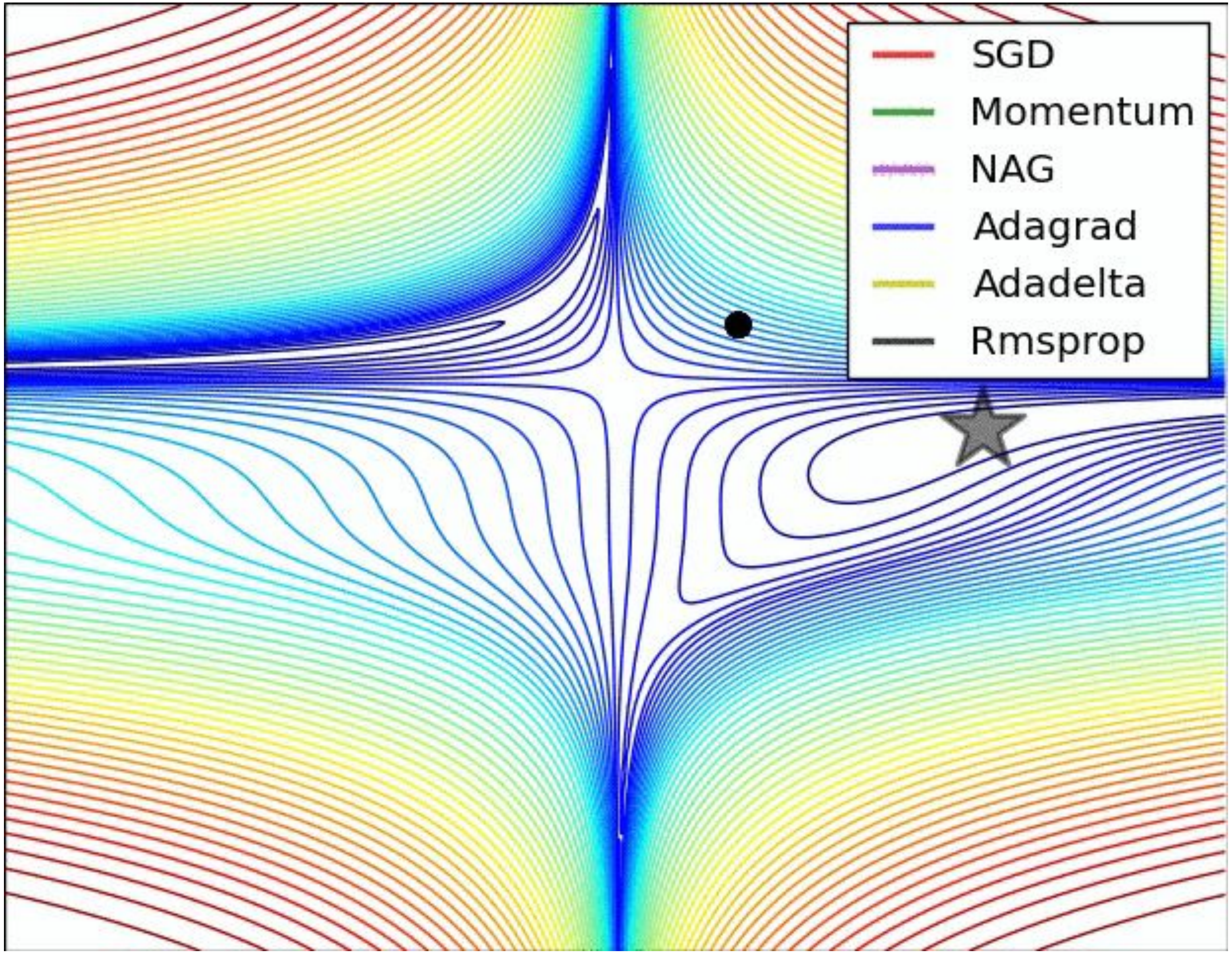
first order optimization

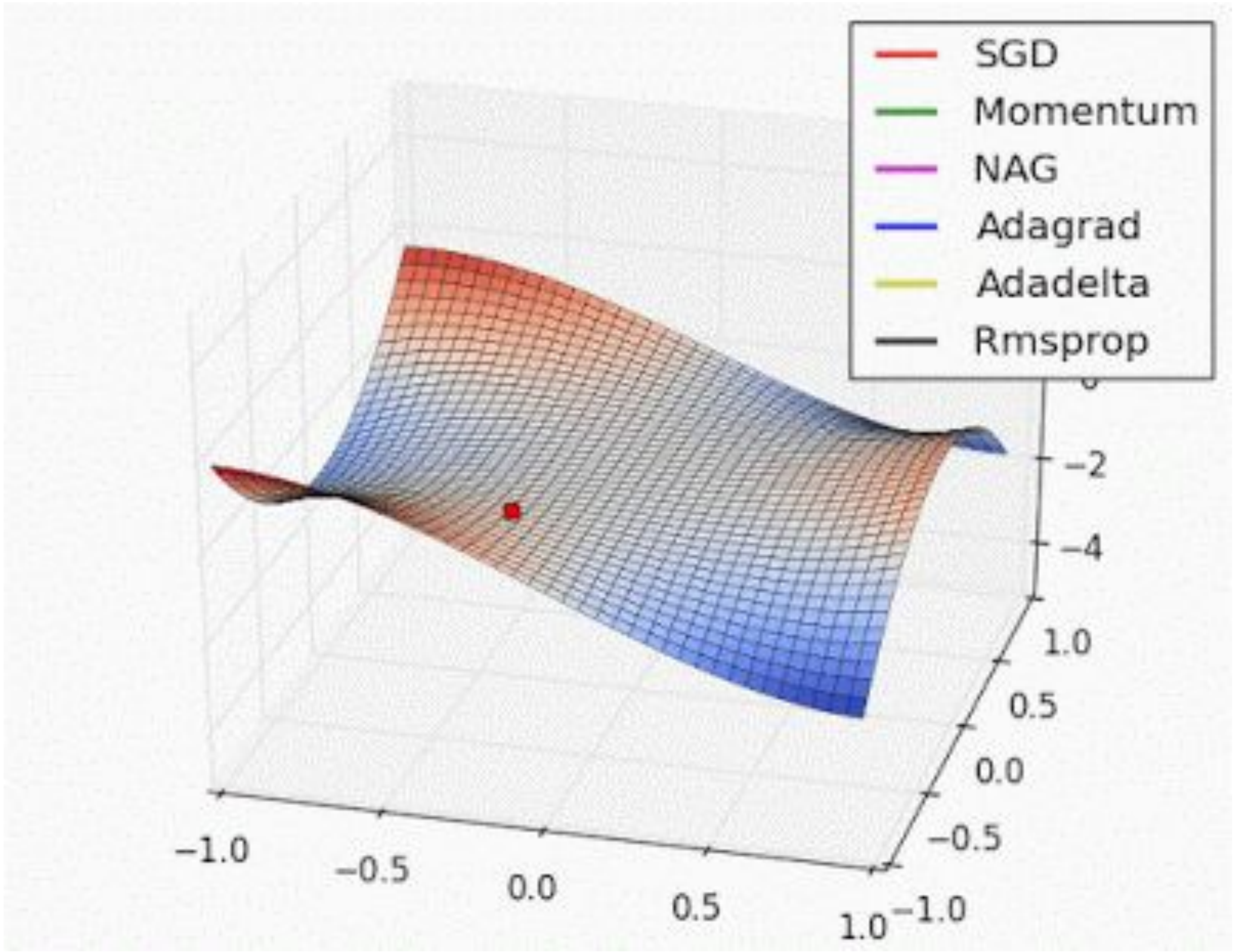
$$S_{\partial\theta} = \frac{\beta_2 S_{\partial\theta} + (1-\beta_2)\partial\theta^2}{1-\beta_2^t}$$

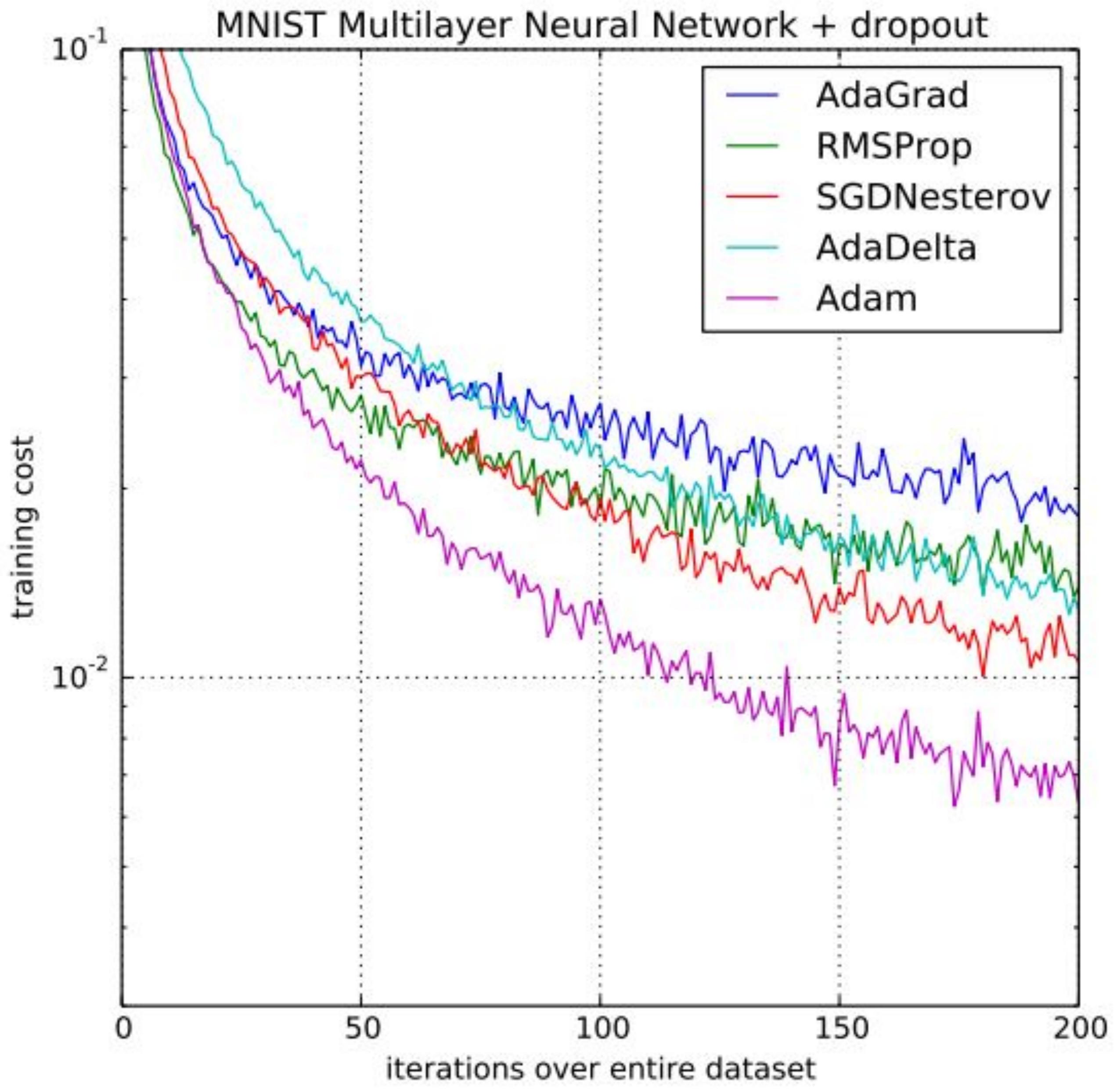
rmsprop

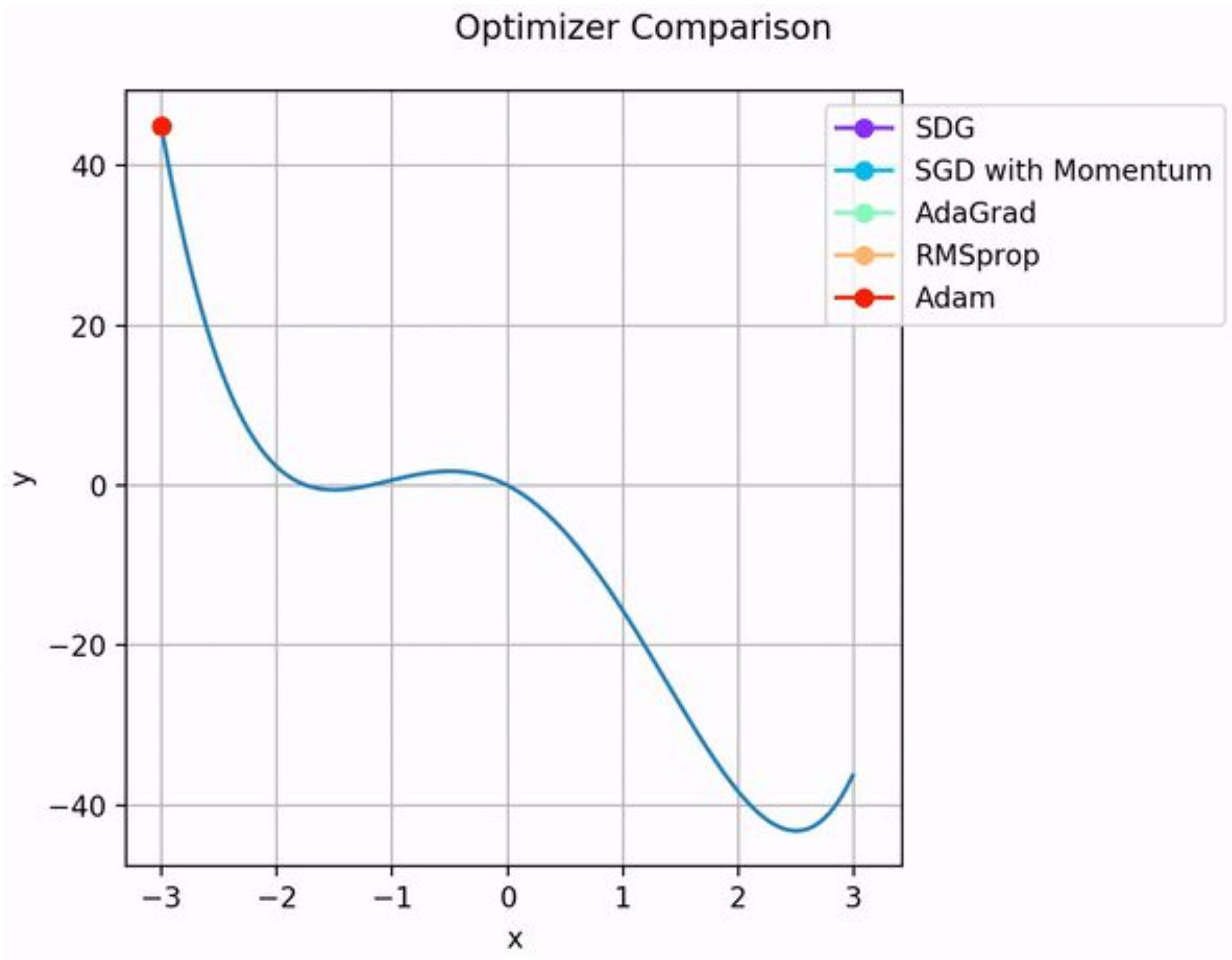
second order optimization

$$\theta = \theta - \alpha\frac{V_{\partial\theta}}{\sqrt{S_{\partial\theta}} + \epsilon}$$


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


MNIST Multilayer Neural Network + dropout
10^{-1}
10^{-2}
training cost
0
50
100
150
200
iterations over entire dataset
AdaGrad
RMSProp
SGDNesterov
AdaDelta
Adam


Optimizer Comparison
SDG
SGD with Momentum
AdaGrad
RMSprop
Adam
y
x
40
20
0
−20
−40
−3
−2
−1
0
1
2
3

In Practice

- Adam is a good default choice in many cases.
  - $\beta_1 = 0.9$ (good default choice)
  - $\beta_2 = 0.999$ (good default choice)
  - α still has to be tuned
  - $\epsilon = 1e - 7$ or $1e - 8$ (does not really affect performance much)
- Slowly decay learning rate over time.
  - Usually, decay every epoch
  - Some people manually decay the learning rate by observing the process