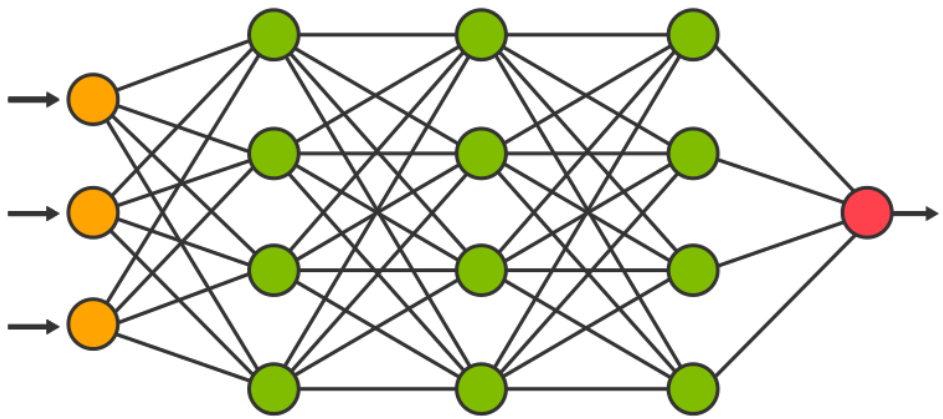
Introduction to Neural Networks

ML Instruction Team, Fall 2022

CE Department
Sharif University of Technology



Biological Analogy

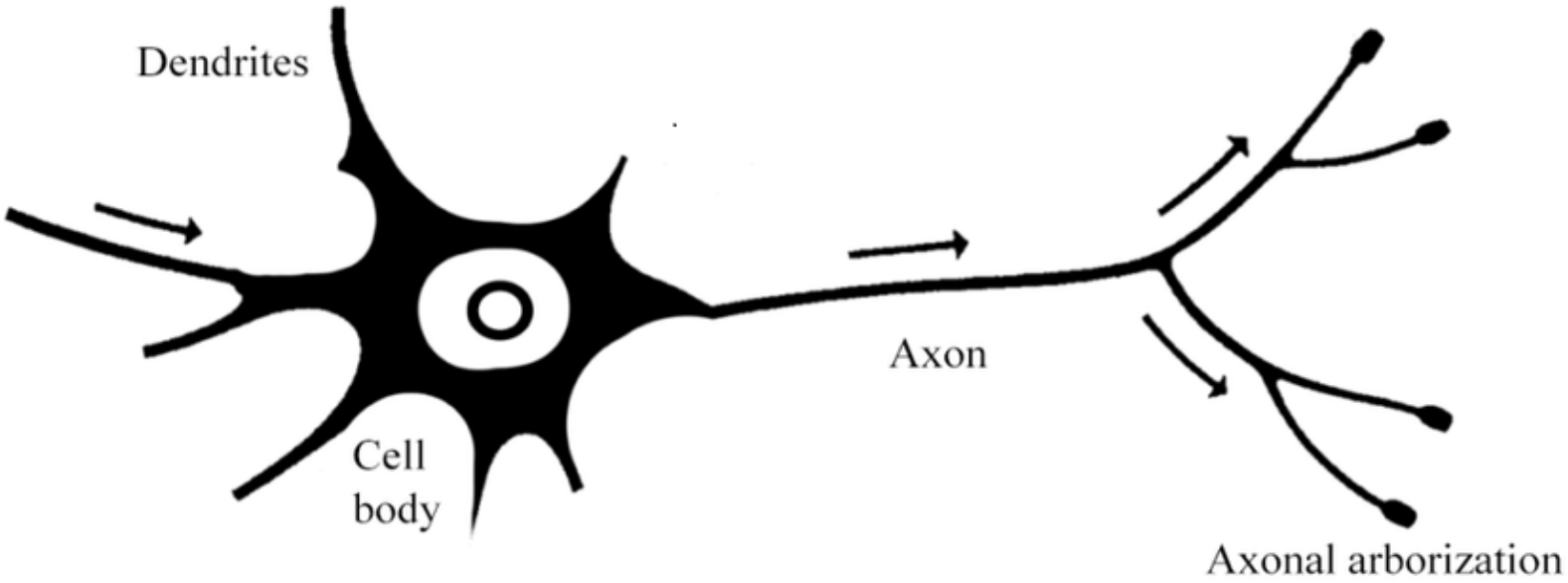


Figure: Anatomy of a biological neuron [1].

Biological analogy

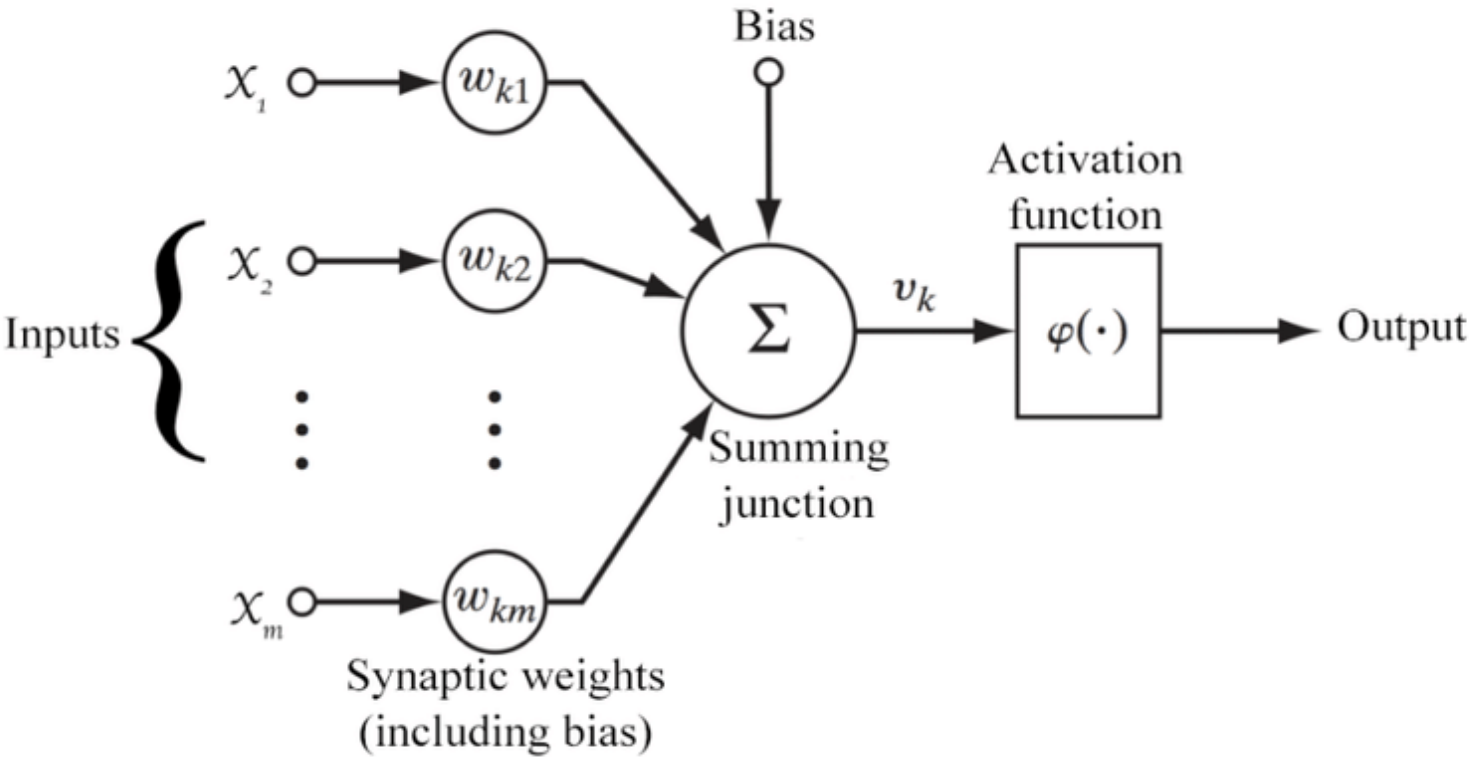


Figure: Neural network neuron [1].

Activation Functions

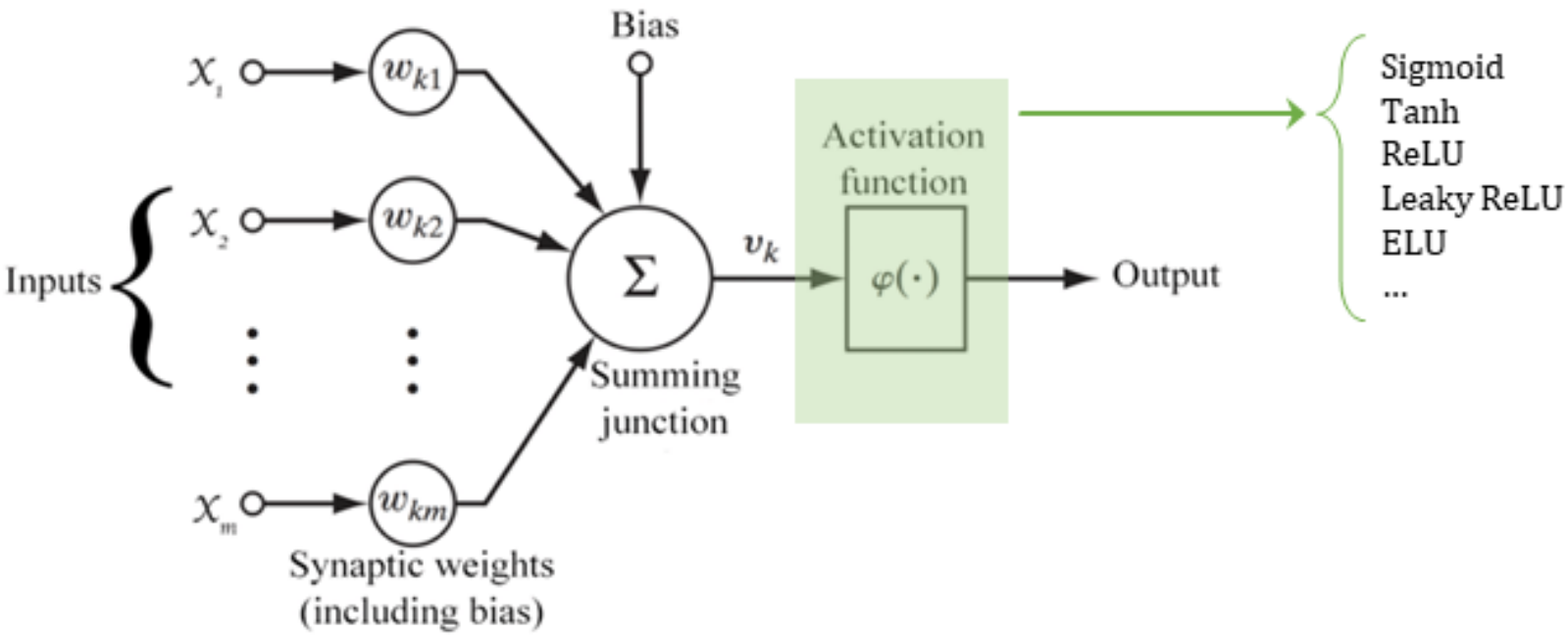


Figure: Activation function

Activation Functions

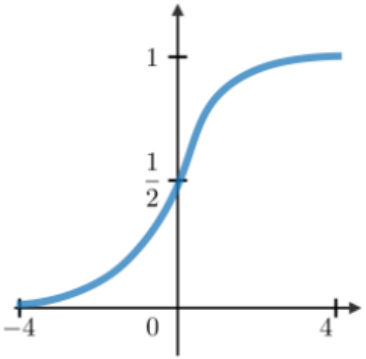
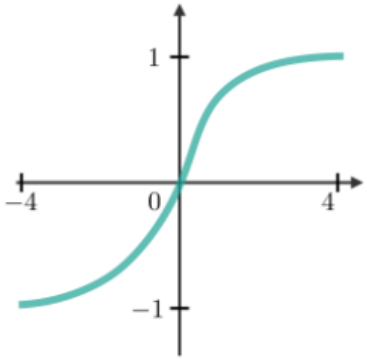
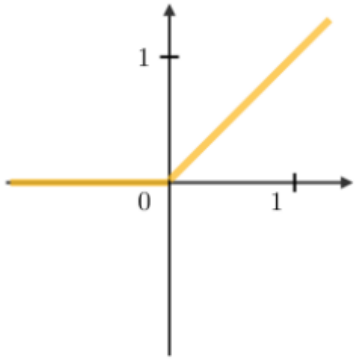
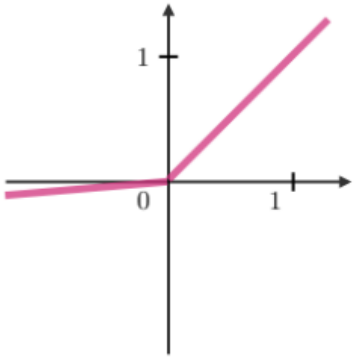
Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
			

Figure: Activation functions [2].

Gradient Descent

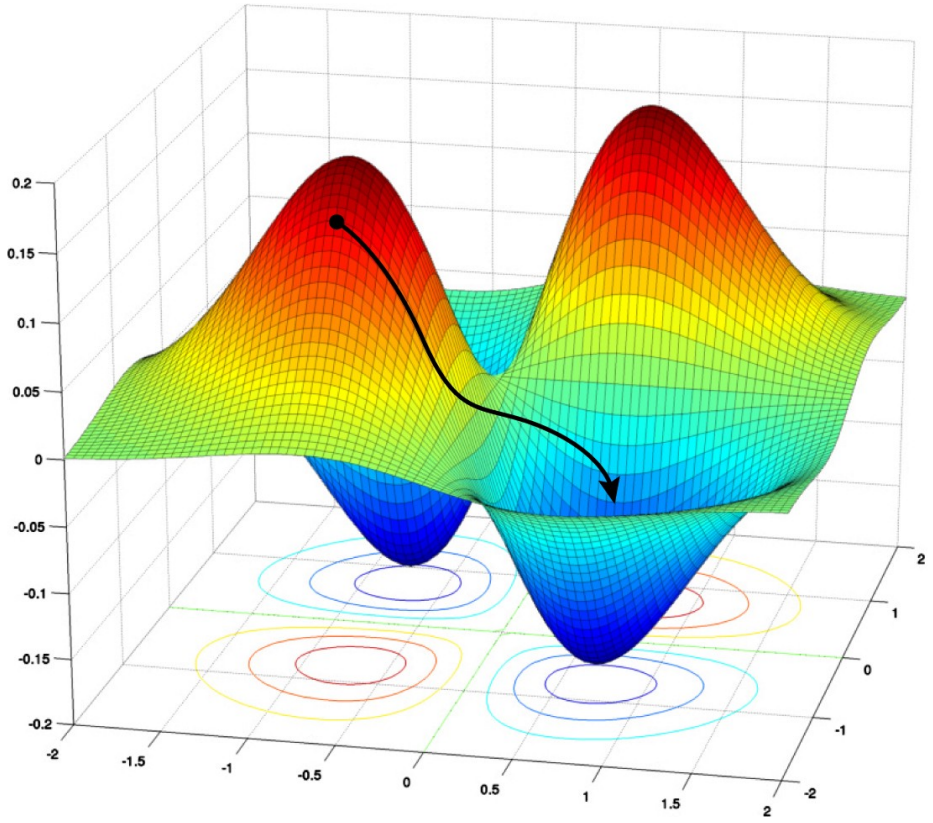


Figure: Gradient descent [3].

Limitations of Perceptrons

- If the data is not separable, the training algorithm may not terminate.
- Test accuracy rises at first, but then starts falling.
- We should have a sufficiently small learning rate η

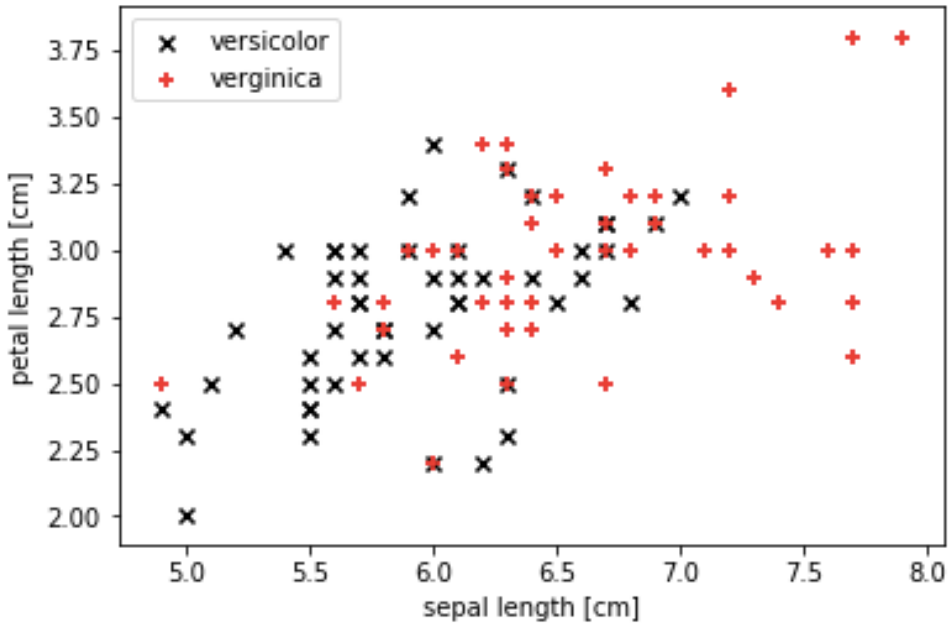


Figure: Not separable data [**?**].

Multi-Layered Perceptron

- Why Should we do?
  - One of the main weaknesses of linear models are that they are linear and they ran into problem when the data has non-linear relations.
  - So we take our biological inspiration further by chaining together perceptrons.

Figure: Inspiration [?].

Learning with hidden units

- Networks without hidden units are very limited in the input-output mappings they can model.
  - More layers of linear units do not help. Its still linear.
  - Fixed output non-linearities are not enough
- We need multiple layers of adaptive non-linear hidden units. This gives us a universal approximator.
  - We need an efficient way of adapting all the weights, not just the last layer. This is hard. Learning the weights going into hidden units is equivalent to learning features.
  - Nobody is telling us directly what hidden units should do.

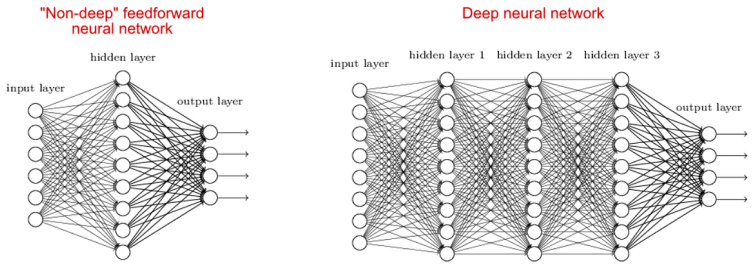


Figure: Multi layer perceptron [?].

XOR Example

- A Perceptron cannot learn the XOR function.
  - Positive cases: $(1,0) -> 1; (0,1) -> 1$
  - Negative cases: $(1,1) -> 0; (0,0) -> 0$
  - ?I need to draw a perceptron myself?

Multi-Layer perceptron neural architecture

- In a typical MLP network, the input units (X_i) are fully connected to all hidden layer units (Y_j) and the hidden layer units are fully connected to all output layer units (Z_k).
- Each of the connections between the input to hidden and hidden to output layer units has an associated weight attached to it (W_{ij} or W_{jk})
- The hidden and output layer units also derive their bias values (b_j or b_k) from weighted connections to units whose outputs are always 1 (true neurons)

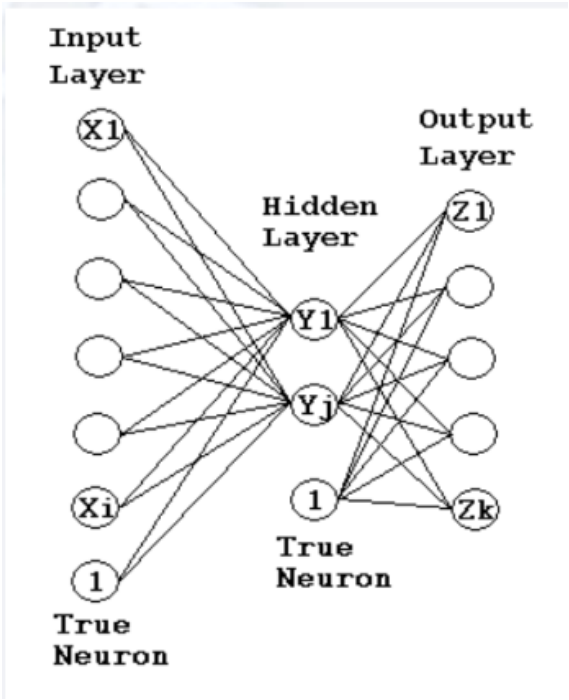


Figure: MLP Architecture [?].

Two-layer MLP

- Two-layer networks are Universal function approximators.
  - It can approximate any continuous function on a bounded subset of D-dimensional space.
- How many hidden units needed then?
  -

Two-layer MLP

- $X_i = input[i]$
- $Y_j = f(b_j^1 + \sum X_i W_{ij}^1)$
- $Z_k = g(b_k^2 + \sum Y_j W_{jk}^2)$

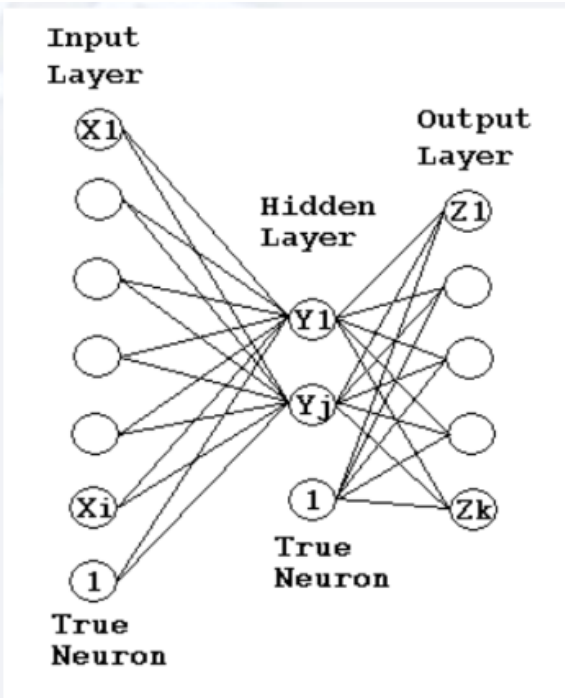


Figure: MLP Architecture [**?**].

Non-linearity

- 1-Layer MLP: $Z = g(Wx)$
- 2-Layer MLP: $Z = g(W_2 f(W_1 x))$
- 3-Layer MLP: $Z = g(W_3 f_2(W_2 f_1(W_1 x)))$
- What does deep mean?
  - In any directed acyclic graph (DAG) ”Depth” is the length of the longest path from a source to a sink

Learning by perturbing weights

- Randomly perturb one weight and see if it improves performance. If so, save the change.
  - Very inefficient. We need to do multiple forward passes on a representative set of training data just to change one weight.
  - Towards the end of learning, large weight perturbations will nearly always make things worse.
- We could randomly perturb all the weights in parallel and correlate the performance gain with the weight changes.
  - Not any better because we need lots of trials to "see" the effect of changing one weight through the noise created by all the others.

Training a MLP

- Now we need to find the Optimal weights for our network

Optimazation

$$\begin{cases} min_{W_N,\ldots,W_1} \sum_{n=1}^{N} \frac{1}{2}(t_n - Z_n)^2 \\ Z = g(W_N f_{N-1}(\ldots(W_2 f_1(W_1 x))\ldots)) \end{cases}$$

- Loss minimization: replace squared-loss with any other
- Regularization:
  - traditionally NN's are not regularized.
  - But you can add a regularization.
- Optimization by gradient descent (Not gurantees about optimality)

Training a 2-layered network

2-layered network optimazation term

$$\left\{ min_{W_2, W_1} \sum_{n=1}^{N} \frac{1}{2}(t_n - \sum iw_i^2 f(w_i^1 x_n))^2 \right.$$

Equiavantly

$$\begin{cases} min_{W_2, W_1} \sum_{n=1}^{N} \frac{1}{2}(t_n - \sum iw^2 h_h i, n \\ h_{i,n} = f(w_i^1 x_n) \end{cases}$$

Computing gradients: output layer

Back Propagation

- We don't know what the hidden units ought to do, but we can compute how fast the error changes as we change a hidden activity.
  - Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities.
  - Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined.
  - We can compute error derivatives for all the hidden units efficiently.
  - Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit.

Back propagation algorithm

- Initialize each W_i to some small random value
- Until the termination condition is met, Do
  - For each training example $< (x_1, ..., x_n), t >$ Do
    - Input the instance $(x_1, \ldots, x_n)$ to the network and compute the network outputs Z_k
    - for each output unit k
    - $\delta_{pk} = z_k(1 - z_k)(t_k - y_k)$
    - for each hidden unit h in the j layer
    - $\delta_{j,h} = y_{j,h}(1 - y_{j,h}) \sum_k W^j_{h,k} \delta_{j+1,k}$
    - for each network weight $W^j_{h,k}$ Do
    - $W^j_{h,k} = W^j_{h,k} + \Delta^j_{h,k}$ where $\Delta^j_{h,k} = \eta \delta_{j,k} x_{i,j}$

Back propagation

- Gradient descent over entire network weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - in practice often works well (can be invoked multiple times with different initial weights)
- Minimizes error training examples

Stochastic Gradient Descent Algorithm

- First, we need to know what does ”Gradient” means.
- Gradient, in plain terms means slope or slant of a surface.
- It means descending a slope to reach the lowest point on that surface.
- In other words, Gradient descent is an iterative algorithm, that starts from a random point on a function and travels down its slope in steps until it reaches the lowest point of that function

Stochastic Gradient Descent Algorithm Cont.

- Here are the steps of GD Algorithm
  - Find the slope of the objective function with respect to each parameter/feature. In other words, compute the gradient of the function.
  - Pick a random initial value for the parameters. (To clarify, in the parabola example, differentiate "y" with respect to "x". If we had more features like x1, x2 etc., we take the partial derivative of "y" with respect to each of the features.)
  - Update the gradient function by plugging in the parameter values.
  - Calculate the step sizes for each feature as : step size = gradient * learning rate.
  - Calculate the new parameters as : new params = old params -step size
  - Repeat steps 3 to 5 until gradient is almost 0.

Stochastic Gradient Descent Algorithm Cont.

- Why do we need Stochastic?
- Gradient descent is slow on huge data.
- "Stochastic", in plain terms means "random".
- Where can we potentially induce randomness in the gradient descent algorithm?
- While selecting data points at each step to calculate the derivatives, SGD randomly picks one data point from the whole data set at each iteration to reduce the computations enormously.

Learning Rate

- Learning rate is a hyper-parameter that controls how much we are adjusting the weights of our network with respect the loss gradient.

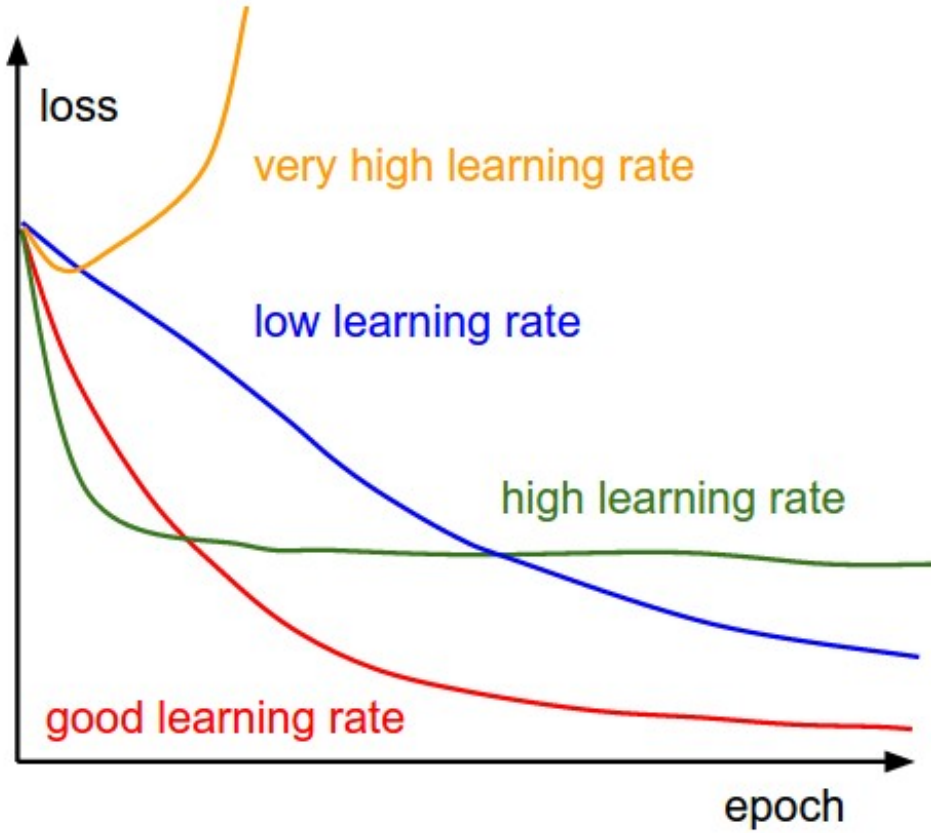


Figure: Learning Rate Effect

- The lower the value of learning rate, the slower we travel along the downward slope.
- While this might be a good idea (using a low learning rate) in terms of making sure that we do not miss any local minima, it could also mean that we'll be taking a long time to

Vanishing/Exploding Gradient

- Exploding
  - In a network of n hidden layers, n derivatives will be multiplied together.
  - If the derivatives are large then the gradient will increase exponentially as we propagate down the model until they eventually explode.
  - If the gradients keep on getting larger and larger as the backpropagation algorithm progresses.
  - This, in turn, causes very large weight updates and causes the gradient descent to diverge.
- Vanishing
  - The derivatives are smaller then the gradient
  - It will decrease exponentially as we propagate through the model until it eventually vanishes,
  - As the backpropagation algorithm advances downwards(or backward) from the output layer towards the input layer, the gradients often get smaller and approach zero which eventually leaves the weights of the initial or lower layers nearly unchanged.
  - As a result, the gradient descent never converges to the optimum.

Vanishing/Exploding Gradient Reasons

- Exploding
  - The model is not learning much on the training data therefore resulting in a poor loss.
  - The model will have large changes in loss on each update due to the models instability.
- Vanishing
  - The model will improve very slowly during the training phase and it is also possible that training stops very early, meaning that any further training does not improve the model.
  - The weights closer to the output layer of the model would witness more of a change whereas the layers that occur closer to the input layer would not change much (if at all).
  - Model weights shrink exponentially and become very small when training the model.

Vanishing/Exploding Gradient Solutions

- Proper Weight Initialization
  - The variance of outputs of each layer should be equal to the variance of its inputs.
  - The gradients should have equal variance before and after flowing through a layer in the reverse direction.
- Batch Normalization
  - It consists of adding an operation in the model just before or after the activation function of each hidden layer.
  - This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting.
  - In other words, the operation lets the model learn the optimal scale and mean of each of the layer's inputs.
  - To zero-center and normalize the inputs, the algorithm needs to estimate each input's mean and standard deviation.
  - It does so by evaluating the mean and standard deviation of the input over the current mini-batch (hence the name "Batch Normalization").

Saturating Functions

- Intuition
  - A saturating activation function squeezes the input.
- Definition
- Examples
  - The tanh (hyperbolic tangent) activation function is saturating as it squashes real numbers to range between (0, 1)
  - The sigmoid activation function is also saturating, because it squashes real numbers to range between (0, 1).
  - In contrast, The Rectified Linear Unit (ReLU) activation function is non-saturating.

Early Stopping

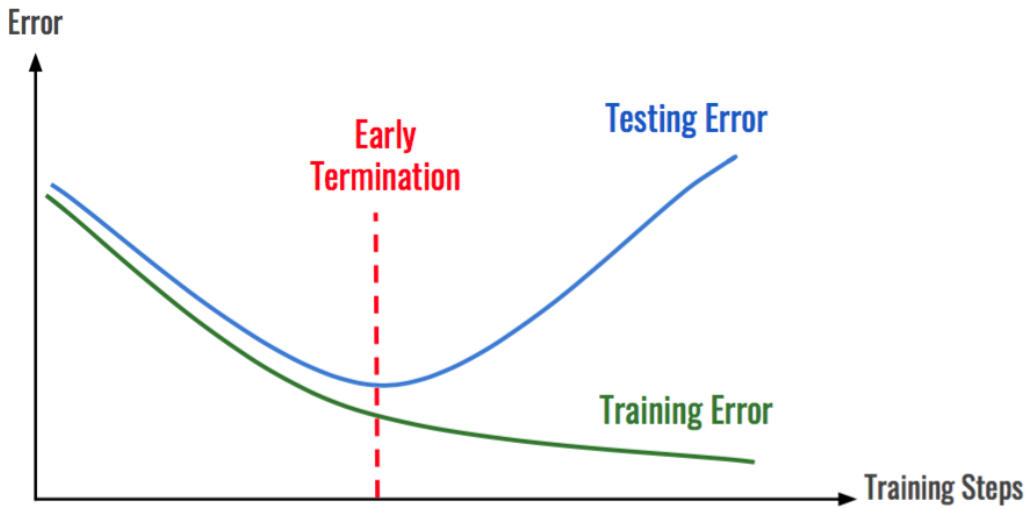


Figure: Early Stopping. Source

- Stop training when accuracy on the validation set decreases (or loss increases). Or keep track of the model parameters that worked best on validation set.

Regularization: Add term to loss

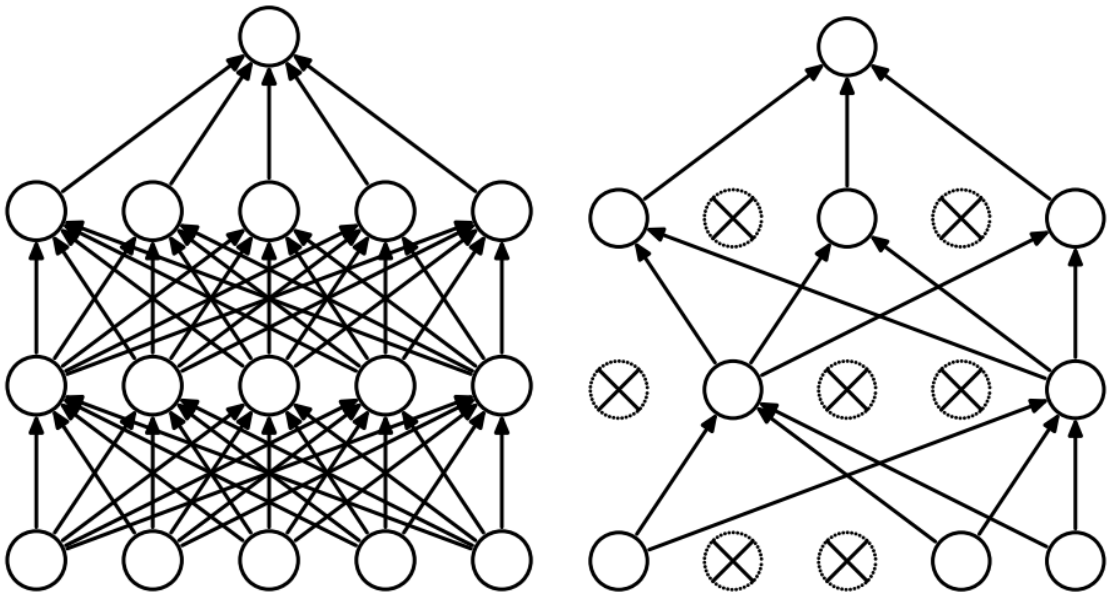
$$L = \frac{1}{N}\sum_{i=1}^{N} L(\phi(x_i), y_i) + \lambda R(W)$$

Common regularization terms:

L2 regularization	$R(W) = \sum_k \sum_l W_{k,l}^2$
L1 regularization	$R(W) = \sum_k \sum_l W_{k,l}^2$
Elastic net (L1 + L2)	$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + \lvert W_{k,l}^2 \rvert$

Regularization: Dropout

- Randomly set some of neurons to zero in forward pass.



(a) Without using dropout (b) After using dropout.

Figure: Behavior of dropout at training time. Source

Regularization: Dropout

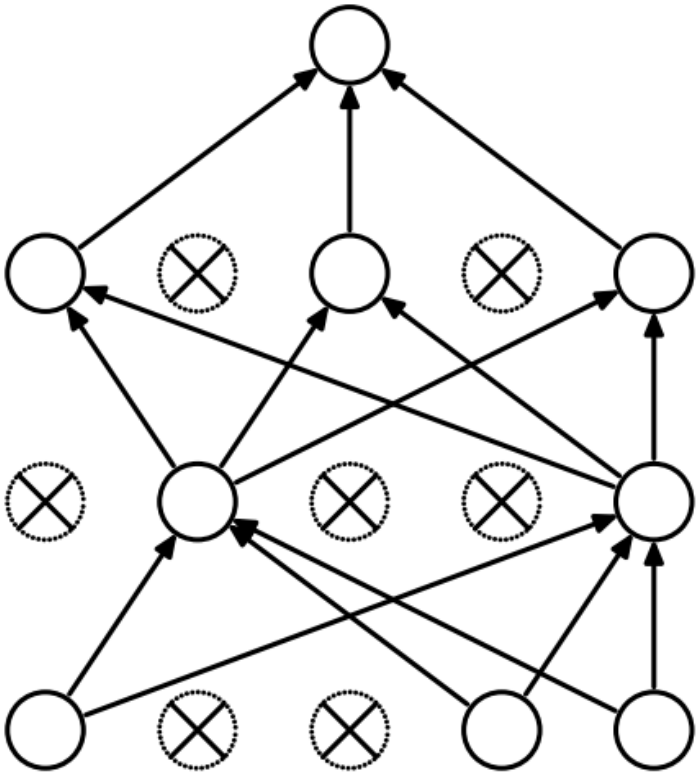


Figure: Source

- Dropout:
  - Prevents co-adaptation of features (forces network to have redundant representations).
  - Can be considered a large ensemble of models sharing parameters.

Dropout: Test Time

- Dropout makes output of network random!

$$y = f_W(x, z)$$

z: random mask
x: input of the layer
y: output of the layer

- We want to "average out" the randomness at test time:

$$y = f(x) = \mathbb{E}_z[f(x, z)] = \int p(z) f(x, z) dz$$

- Can we calculate the integral exactly?

Dropout: Test Time

- Dropout makes output of network random!

$$y = f_W(x, z)$$

z: random mask
x: input of the layer
y: output of the layer

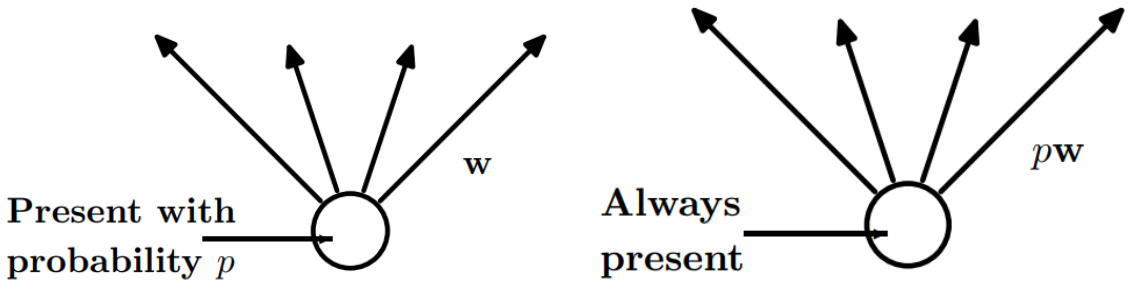
- We want to "average out" the randomness at test time:

$$y = f(x) = \mathbb{E}_z[f(x, z)] = \int p(z) f(x, z) dz$$

- Can we calculate the integral exactly?
- We need to approximate the integral.

Dropout: Test Time

- At test time neurons are always present and its output is multiplied by dropout probability:



(a) Without using dropout (b) After using dropout.

Figure: Behavior of dropout at test time. Source

Regularization: Adding Noise

- We've seen a common approach for regularization thus far:
  - **Training**: Add some kind of randomness (z) :

$$y = f_W(x, z)$$

  - **Testing**: Average out the randomness:

$$y = f(x) = \mathbb{E}_z[f(x, z)] = \int p(z) f(x, z) dz$$

Regularization: Adding Noise

- We've seen a common approach for regularization thus far:
  - **Training**: Add some kind of randomness (z) :

$$y = f_W(x, z)$$

  - **Testing**: Average out the randomness:

$$y = f(x) = \mathbb{E}_z[f(x, z)] = \int p(z) f(x, z) dz$$

- Adding noise is another way to prevent a neural network from overfitting on the training data. In every iteration, a random noise is added to the outputs of the layer, preventing consequent layers from co-adapting too much to the outputs of this layer.

Batch Normalization

Input: $x : N \times D$ Learnable Parameters: $\gamma, \beta : D$

Output: $y : N \times D$ Intermediates: $\mu, \sigma : D$, $\hat{x} : N \times D$

$$\mu_j = \text{(Running) average of values seen during training}$$

$$\sigma_j^2 = \text{(Running) average of values seen during training}$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Batch Normalization

- Batch normalization is done along with **C** axis in convolutional networks:

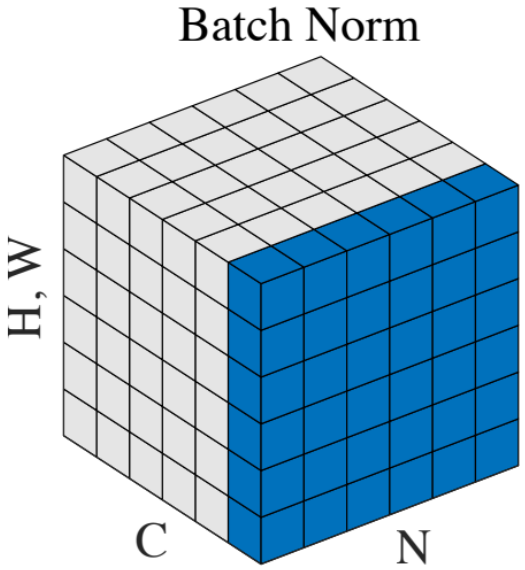


Figure: Batch normalization in CNNs Source.

- BN for FCNs: $x, y : N \times D \rightarrow \mu, \sigma, \gamma, \beta : 1 \times D$
- BN for CNNs: $x, y : N \times C \times H \times W \rightarrow \mu, \sigma, \gamma, \beta : 1 \times C \times 1 \times 1$
- In both cases: $y = \gamma(x - \mu)/\sigma + \beta$

Gradient Clipping

- In case of a large or small gradient, what will happen?

Gradient Clipping

- In case of a large or small gradient, what will happen?
- Gradient descent either won't change our position or will send us far away.

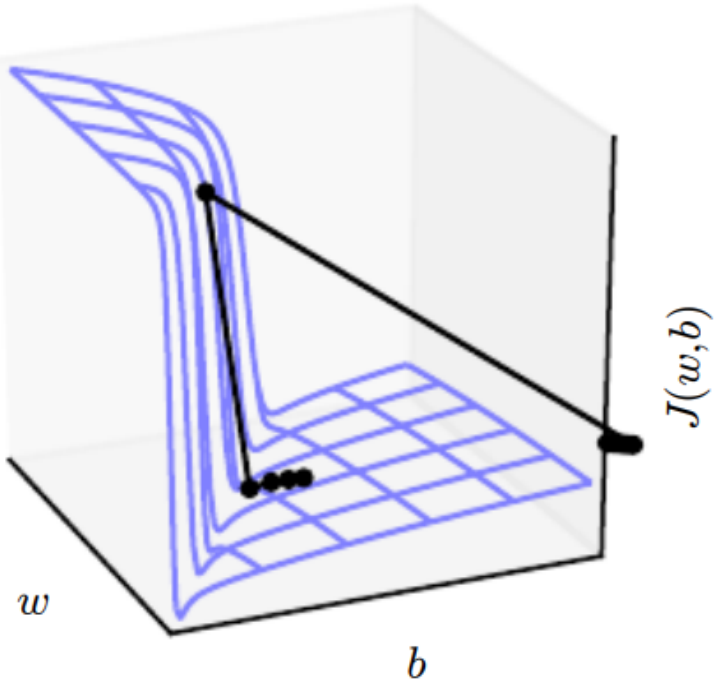


Figure: The problem of large gradient value [4].

Gradient Clipping

- Solve this problem simply by clipping gradient
- Two approaches to do so:
  - Clipping by value
  - Clipping by norm

Gradient Clipping by value

- Set a max (α) and min (β) threshold value
- For each index of gradient $\boldsymbol{g}_i$ if it is lower or greater than your threshold clip it:

$$\begin{aligned} &\text{if } \boldsymbol{g}_i > \alpha : \\ &\qquad \boldsymbol{g}_i \leftarrow \alpha \\ &\text{else if } \boldsymbol{g}_i < \beta : \\ &\qquad \boldsymbol{g}_i \leftarrow \beta \end{aligned}$$

- Clipping by value will not save gradient direction but still works well in practice.
- To preserve direction use clipping by norm.

Gradient Clipping by norm

- Clip the norm $\|\boldsymbol{g}\|$ of the gradient $\boldsymbol{g}$ before updating parameters:

$$\text{if } \|\boldsymbol{g}\| > v:$$
$$\boldsymbol{g} \leftarrow \frac{\boldsymbol{g}}{\|\boldsymbol{g}\|} v$$

  v is the threshold for clipping which is a hyperparameter.
- Gradient clipping saves the direction of gradient and controls its norm.

Gradient Clipping

- The effect of gradient clipping:

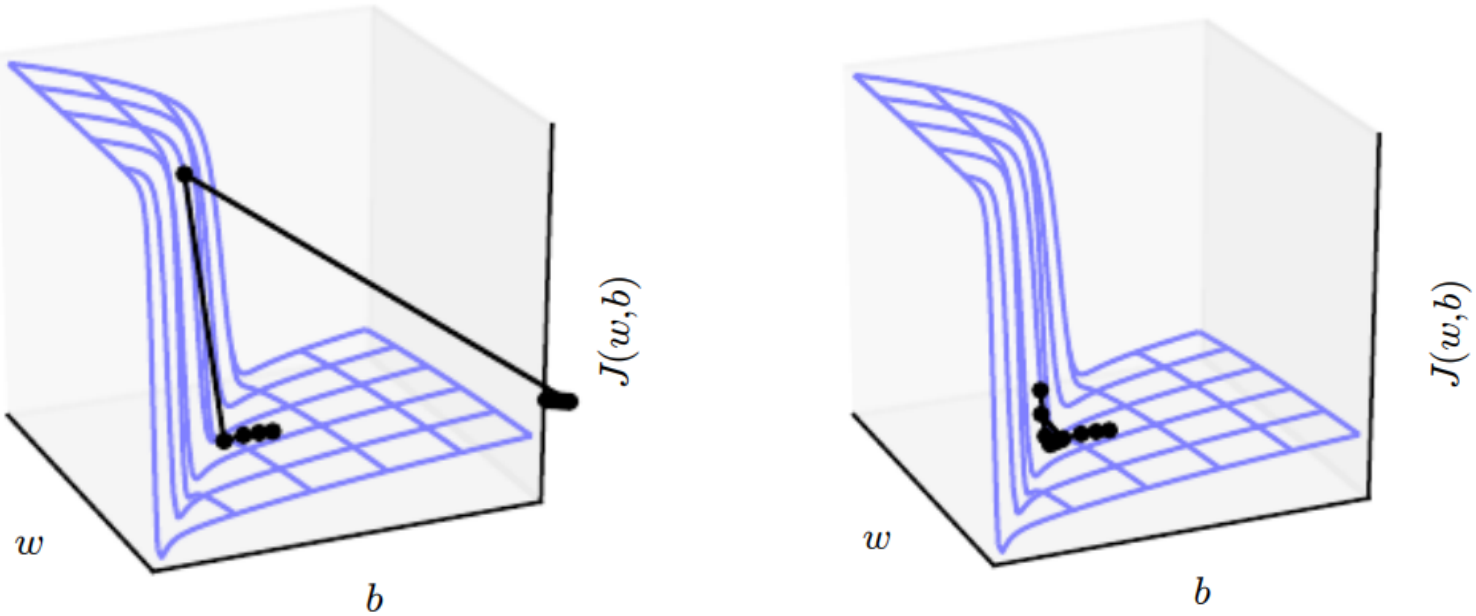


Figure: The "cliffs" landscape (left) without gradient clipping and (right) with gradient clipping [4].

Weight Initialization

- Is initialization really necessary?
- What are the impacts of initialization?

Weight Initialization

- Is initialization really necessary?
- What are the impacts of initialization?
- A bad initialization may increase convergence time or even make optimization diverge.
- How to initialize?
  - Zero initialization
  - Random initialization

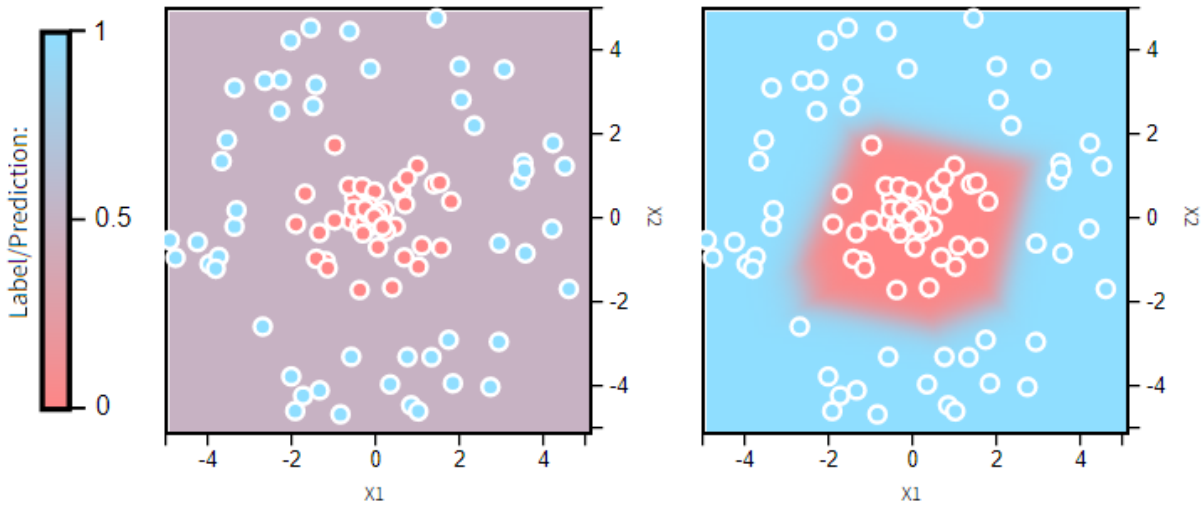
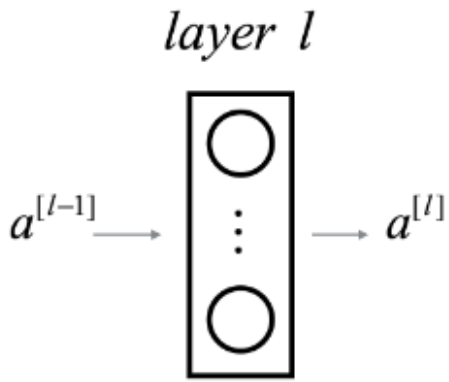


Figure: The output of a three layer network after about 600 epoch. (left) using a bad initialization method and (right) using an appropriate initialization [5].

Weight Initialization

Let's review some notations before we continue:

$$\begin{cases} n^{[l]} := \text{layer } l \text{ neurons number}, \\ W^{[l]} := \text{layer } l \text{ weights}, \\ b^{[l]} := \text{layer } l \text{ biases}, \\ a^{[l]} := \text{layer } l \text{ outputs} \end{cases}$$



Weight Initialization: Zero Initialization

Zero Initialization method:

$$\begin{cases} W^{[l]} = 0, \\ b^{[l]} = 0 \end{cases}$$

- Simple but perform very poorly. (why?)

Weight Initialization: Zero Initialization

Zero Initialization method:

$$\begin{cases} W^{[l]} = 0, \\ b^{[l]} = 0 \end{cases}$$

- Simple but perform very poorly. (why?)
- Zero initialization will lead each neuron to learn the same feature
- This problem is known as network failing to break symmetry
- In fact any constant initialization suffers from this problem.

Weight Initialization: Zero Initialization

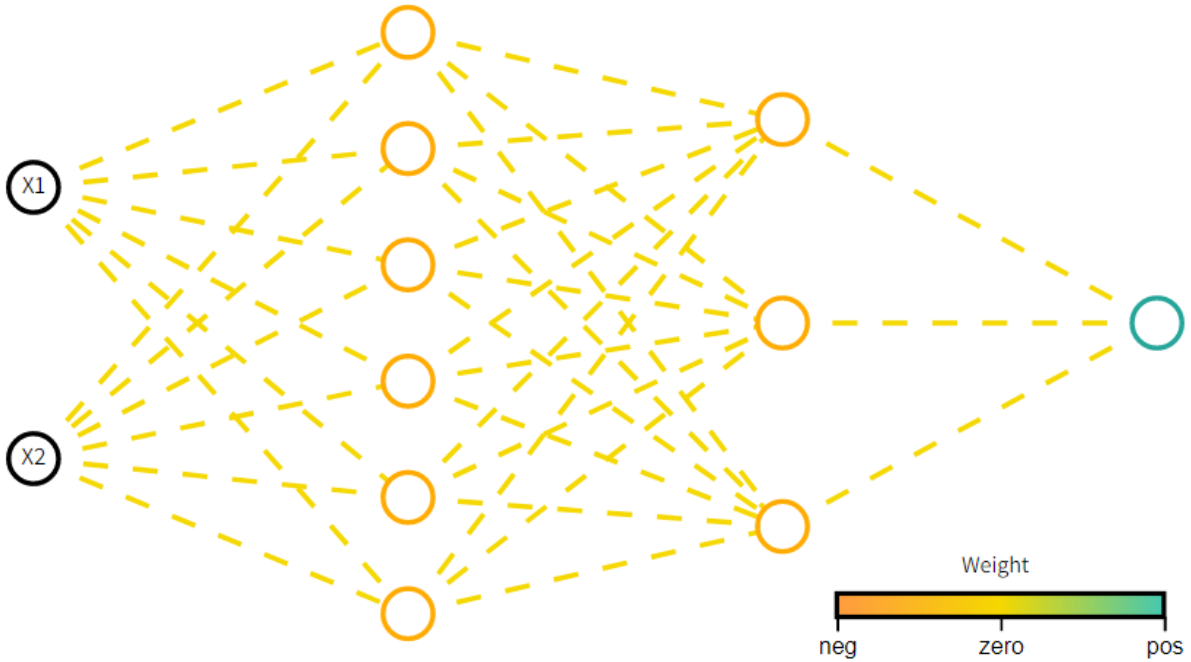


Figure: As we can see network has failed to break symmetry. There has been no improvement in weights after about 600 epochs of training [5].

- We need to break symmetry. How? using randomness.

Weight Initialization: Random Initialization

Simple Random Initialization:

$$\begin{cases} W^{[l]} \sim \mathcal{N}\left(\mu = 0, \sigma^2\right), \\ b^{[l]} = 0 \end{cases}$$

Weight Initialization: Random Initialization

Simple Random Initialization:

$$\begin{cases} W^{[l]} \sim \mathcal{N}\left(\mu = 0, \sigma^2\right), \\ b^{[l]} = 0 \end{cases}$$

- It depends on standard deviation (σ) value
- If it choose carefully, will perform well for small networks
- One can use $\sigma = 0.01$ as a best practice.
- But still has problems with deeper networks.
- Too small/large value for σ will lead to vanishing/exploding gradient problem.

Weight Initialization: Random Initialization

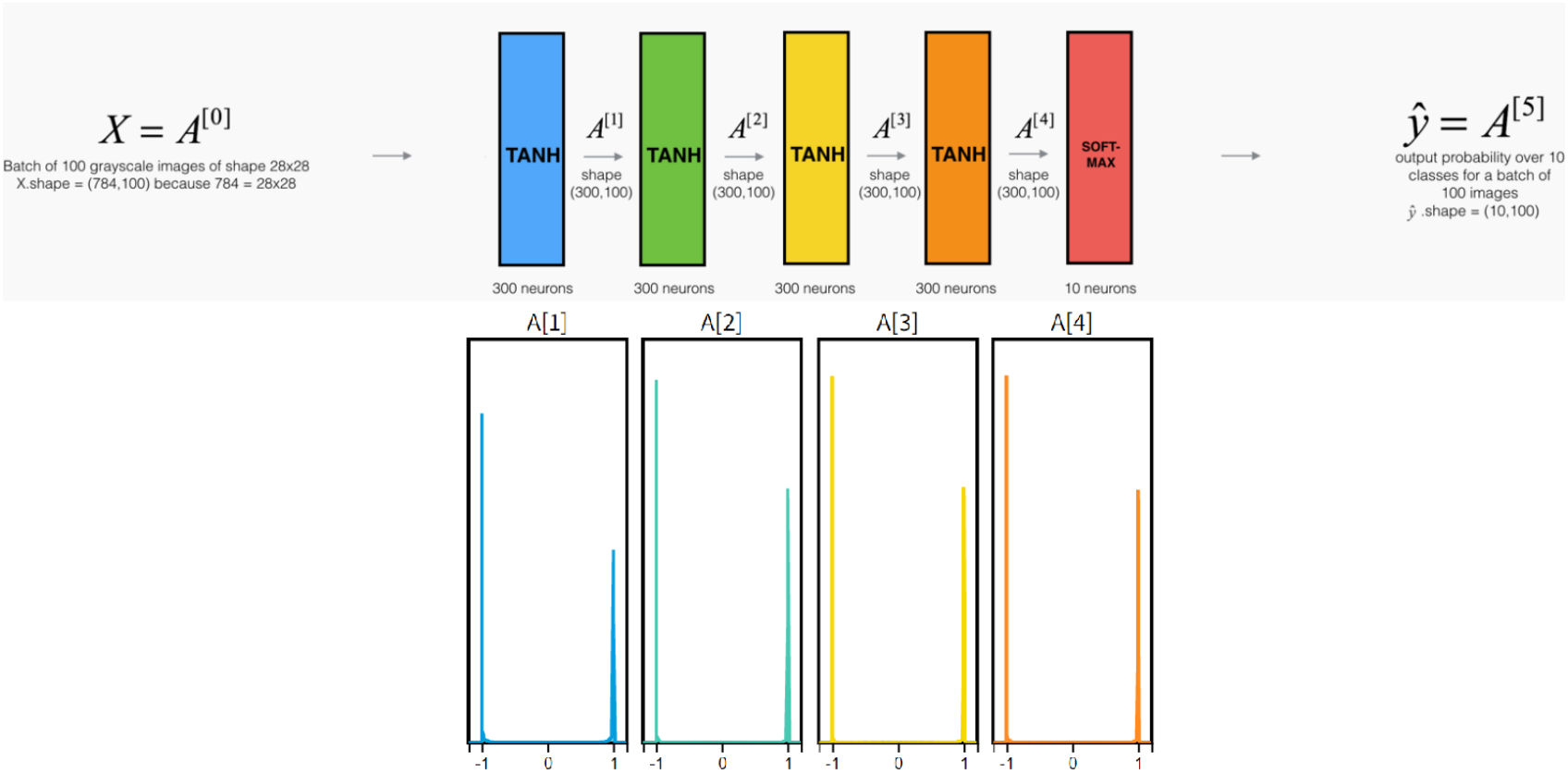


Figure: The problem of normal initialization. On the top, you can see the model architecture, and on the bottom, you can see the density of each layer's output. Model has trained on MNIST dataset for 4 epoch. Weights are initialized randomly from $\mathcal{N}(0, 1)$ [5].

Weight Initialization: Random Initialization

- How to have a better random initialization?
- We need to follow these rules:
  - keep the mean of the activations zero.
  - keep the variance of the activations same across every layer.
- How to do so?

Weight Initialization: Random Initialization

- How to have a better random initialization?
- We need to follow these rules:
  - keep the mean of the activations zero.
  - keep the variance of the activations same across every layer.
- How to do so?

Xavier Random Initialization:

$$\begin{cases} W^{[l]} \sim \mathcal{N}\left(\mu = 0, \sigma^2 = \frac{1}{n^{[l]}}\right), \\ b^{[l]} = 0 \end{cases}$$

(this method works fine for *tanh*, and you can read about why it works at [5].)

Weight Initialization: Random Initialization

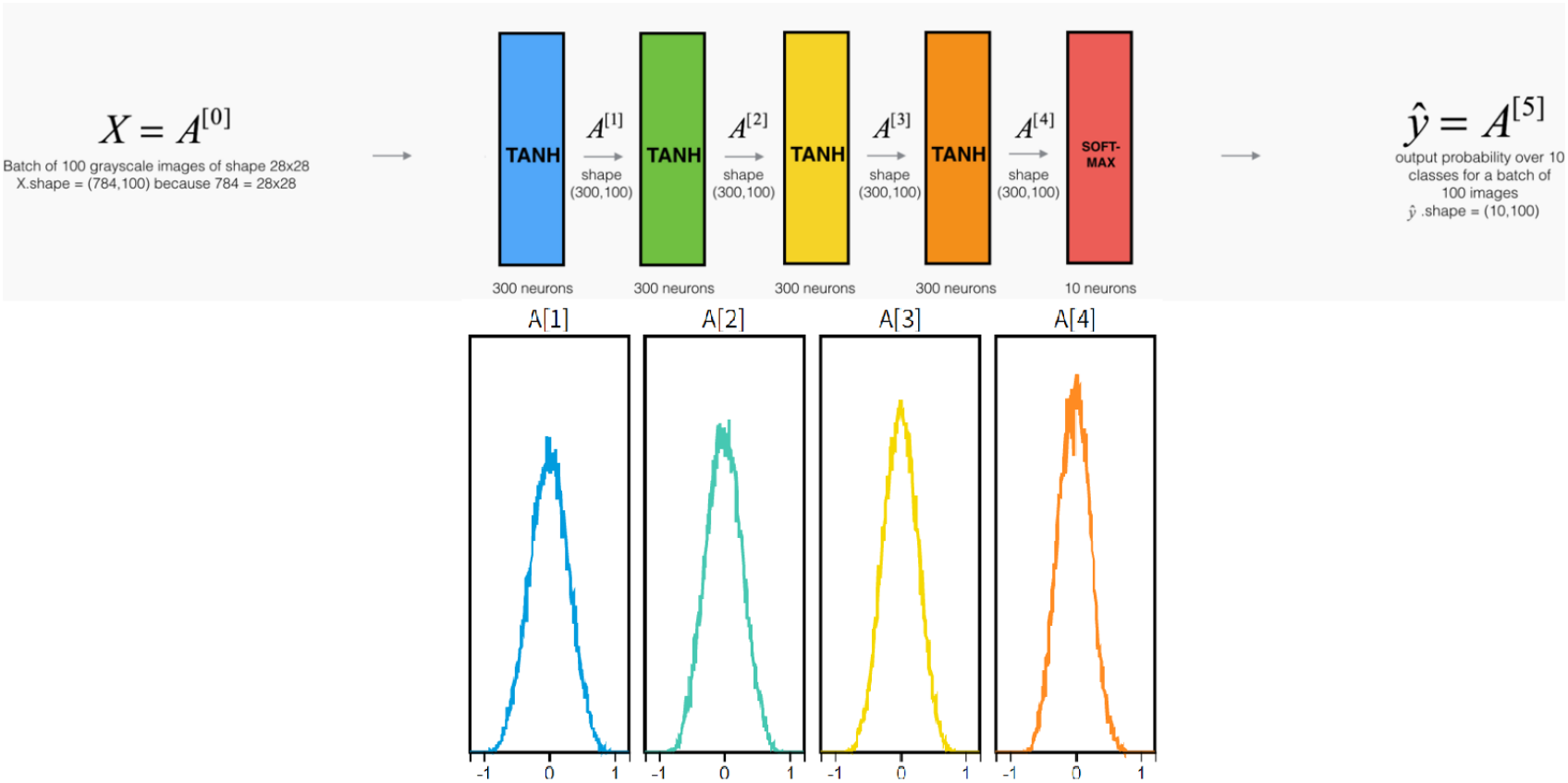


Figure: Vanishing gradient is no longer problem using Xavier initialization. Model has trained on MNIST dataset for 4 epoch. [5].

Weight Initialization

- We discussed weight initialization in previous slides.
- A good initialization will help the model with the vanishing/exploding gradient problem.
- Xavier method works well with `tanh` activation function.
  - If you use $ReLU$ activation use He initialization:

He Initialization:

$$\begin{cases} W^{[l]} \sim \mathcal{N}\left(\mu = 0, \sigma^2 = \frac{2}{n^{[l]}}\right), \\ b^{[l]} = 0 \end{cases}$$

Various GD types

- So far you got familiar with gradient-based optimization
- If $\boldsymbol{g}$ is the gradient of cost w.r.t parameters $\boldsymbol{\theta}$, then we will update parameters with this simple rule:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{g}$$

- But there is one question here, how to compute $\boldsymbol{g}$?
- Based on how we calculate $\boldsymbol{g}$ we will have different types of gradient descent:
  - Batch Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent

Various GD types

Review before continue:

Training cost function ($\mathcal{J}$) over a dataset usually is the average of loss function ($\mathcal{L}$) on entire training set, so for a dataset $\mathcal{D} = \{d_i\}_{i=1}^{n}$ we have:

$$\mathcal{J}(\mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(d_i; \boldsymbol{\theta})$$

Various GD types: Batch Gradient Descent

- In this type we use entire training set to calculate gradient

Batch Gradient Descent:

$$\boldsymbol{g} = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} \mathcal{L}(d_i, \boldsymbol{\theta})$$

- This really needs huge computation and so is slow for large training sets.

Various GD types: Batch Gradient Descent

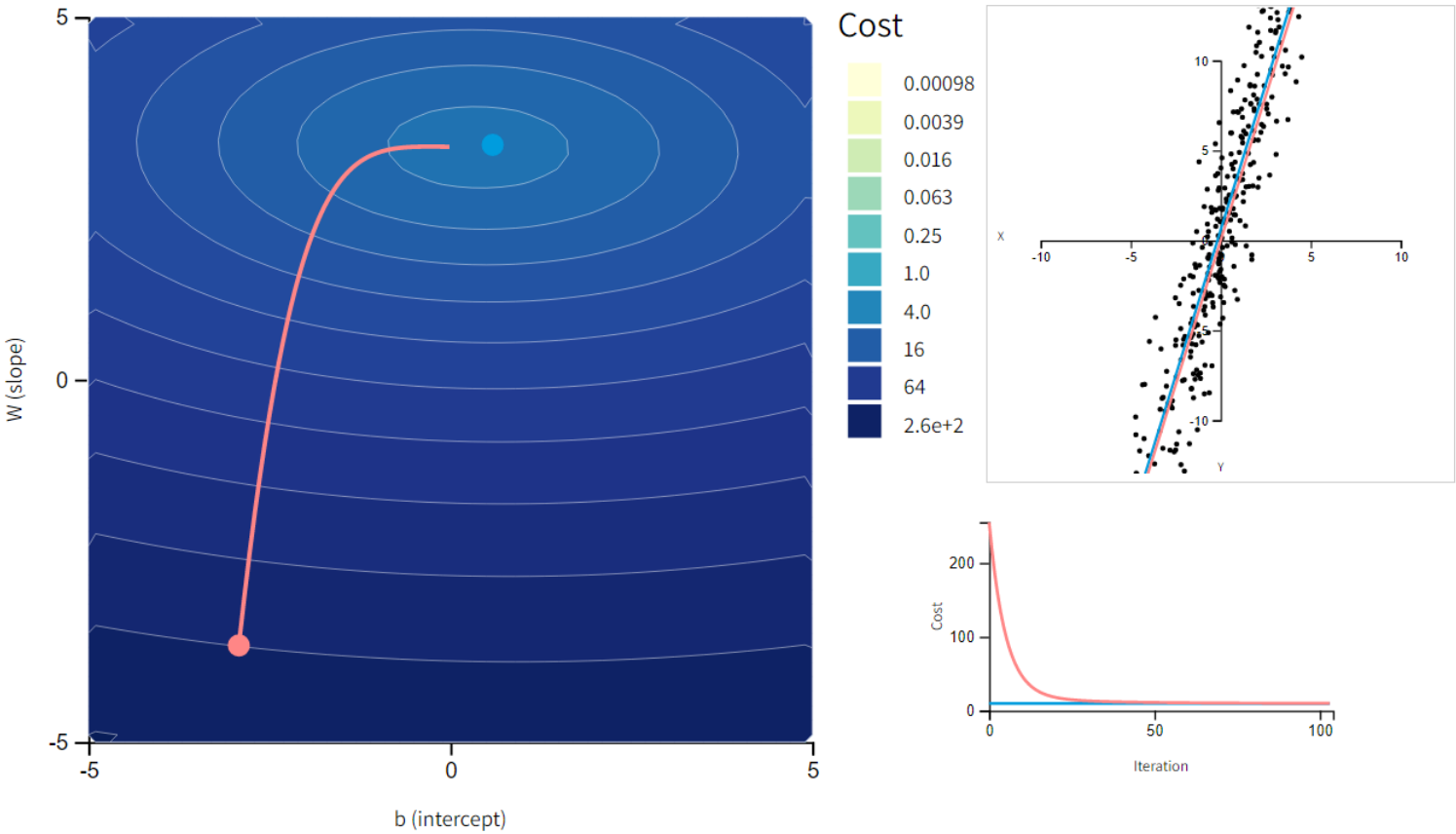


Figure: Optimization of parameters using BGD. Movement is very smooth [6].

Various GD types: Stochastic Gradient Descent

- Instead of calculating exact gradient, we can estimate it using our data
- This is exactly what SGD does, it estimates gradient using only single data point

Stochastic Gradient Descent:

$$\hat{\boldsymbol{g}} = \nabla_{\boldsymbol{\theta}} \mathcal{L}(d_i, \boldsymbol{\theta})$$

- As we use an approximation of gradient, instead of gently decreasing, the cost function will bounce up and down and decrease only on average.
- This method is really computationally efficient cause we only need to calculate gradient for one point per iteration.

Various GD types: Stochastic Gradient Descent

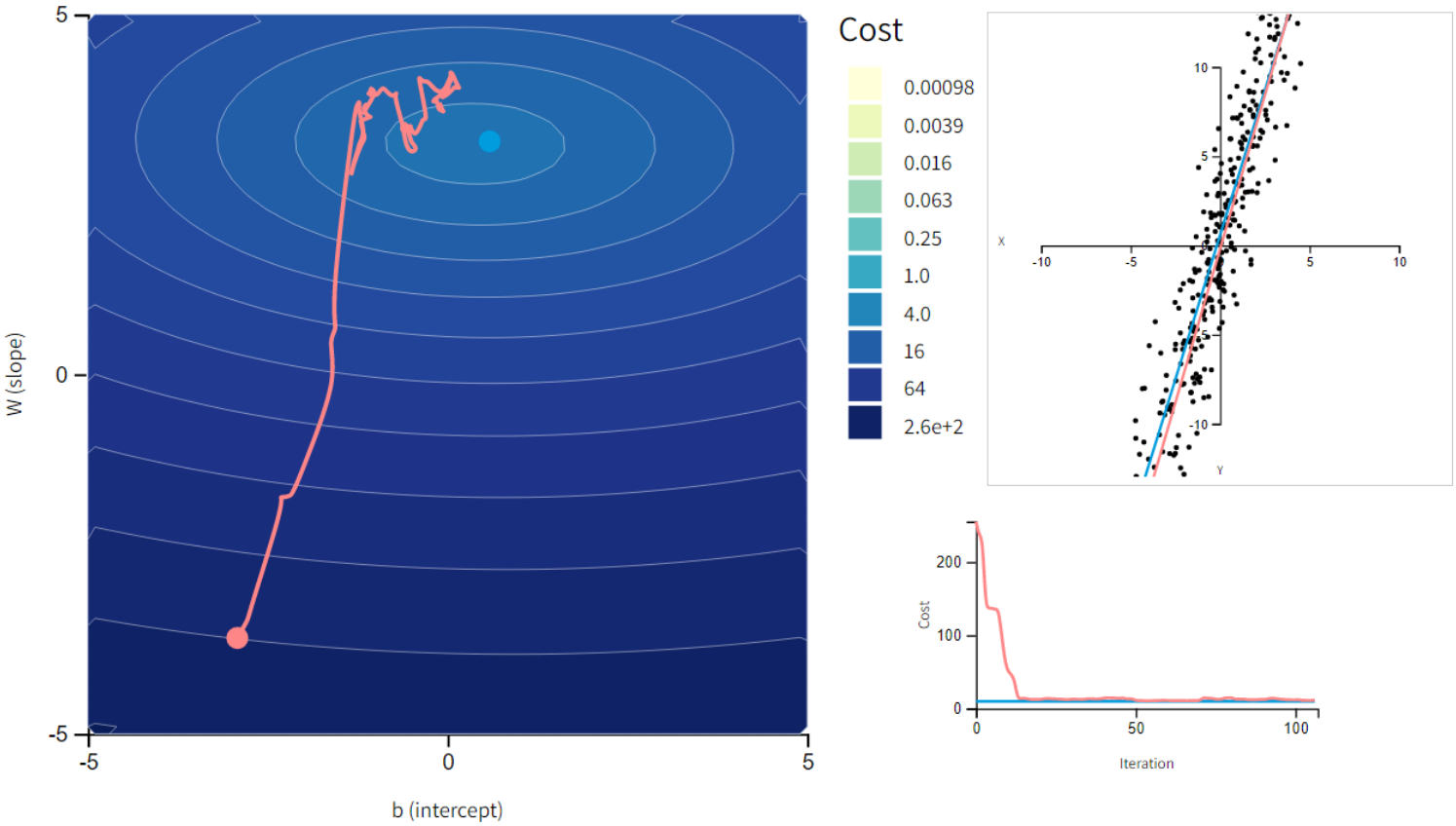


Figure: Optimization of parameters using SGD. As we expect, the movement is not that smooth [6].

Various GD types: Mini-Batch Gradient Descent

- In this method we still use estimation idea But use a batch of data instead of one point.

Mini-Batch Gradient Descent:

$$\hat{\boldsymbol{g}} = \frac{1}{|\mathcal{B}|} \sum_{d \in \mathcal{B}} \nabla_{\boldsymbol{\theta}} \mathcal{L}(d, \boldsymbol{\theta}), \quad \mathcal{B} \subset \mathcal{D}$$

- A better estimation than SGD
- With this way we can get a performance boost from hardware optimization, especially when using GPUs.
- Batch size ($|\mathcal{B}|$) is a hyperparameter you need to tune.

Various GD types: Mini-Batch Gradient Descent

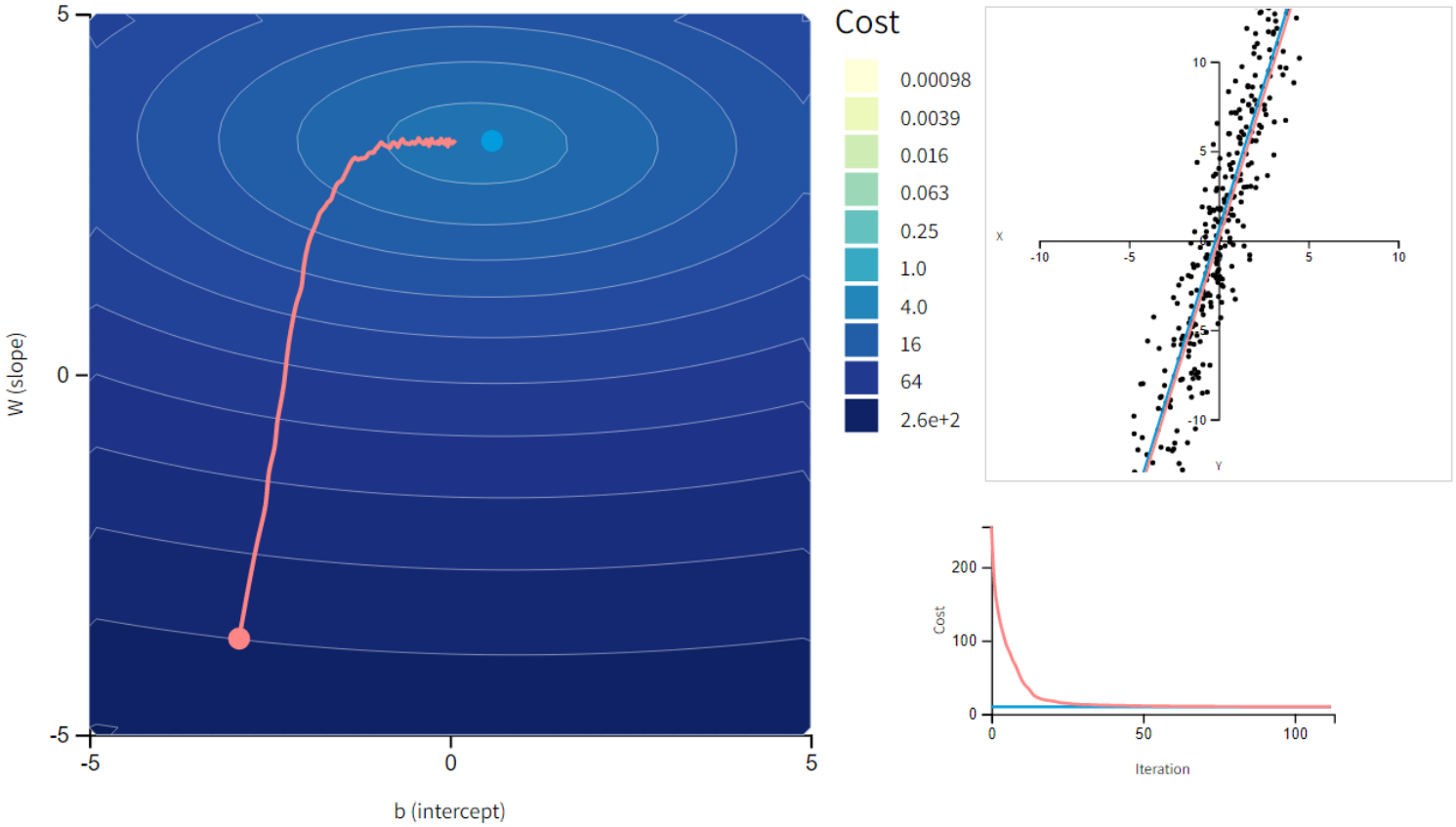


Figure: Optimization of parameters using MBGD. The movement is much smother than SGD and behave like BGD [6].

Various GD types

- So we got familiar with different types of GD.
  - ✓ Batch Gradient Descent (BGD)
  - ✓ Stochastic Gradient Descent (SGD)
  - ✓ Mini-Batch Gradient Descent (MBGD)

- The most recommended one is MBGD, because it is computational efficient.
- Choosing the right batch size is important to ensure convergence of the cost function and parameter values, and to the generalization of your model.

Thank You!

Any Question?

References


M. D. Ergün Akgün, "Biological and neural network neuron," 2018.
https://www.researchgate.net/publication/326417061_Modeling_Course_Achievements_of_Elementary_Education_Teacher_Candidates_with_Artificial_Neural_Networks.


L. Nalborczyk, "A gentle introduction to deep learning in r using keras," 2021.
https://www.barelysignificant.com/slides/vendredi_quanti_2021/vendredi_quantis#1.


N. Pandey, "What the hell is gradient descent?," 2022.
https://faun.pub/what-the-hell-is-gradient-descent-97a75cc5cc4e.


I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*.
MIT Press, 2016.
http://www.deeplearningbook.org.


K. Katanforoosh and D. Kunin, "Initializing neural networks," 2019.
https://www.deeplearning.ai/ai-notes/initialization/.


K. Katanforoosh and D. Kunin, "Parameter optimization in neural networks," 2019.
https://www.deeplearning.ai/ai-notes/optimization/.