Optimization

Minjong Lee

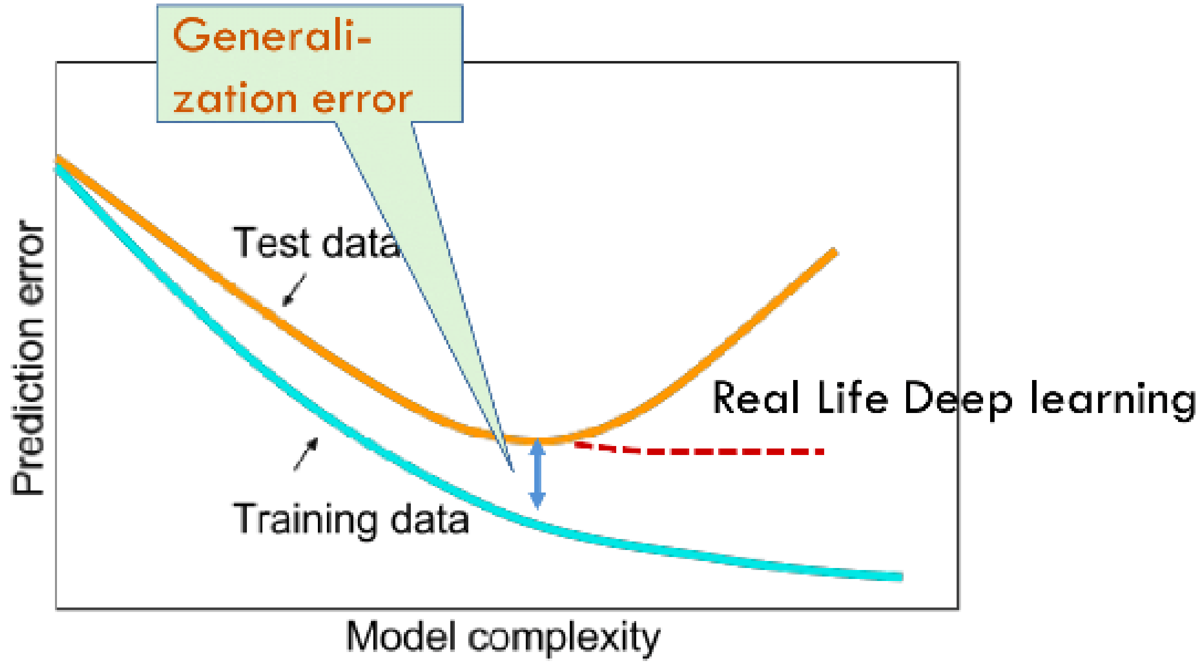
Pohang University of Science and Technology (Postech) CSE

Minjong.Lee@postech.ac.kr

Training/Test error and Generalization

- **Training error**
  - error of training dataset
- **Test error**
  - Error of test dataset (new and previously unseen data)
- **Generalization**
  - The ability to perform well on previously unobserved input

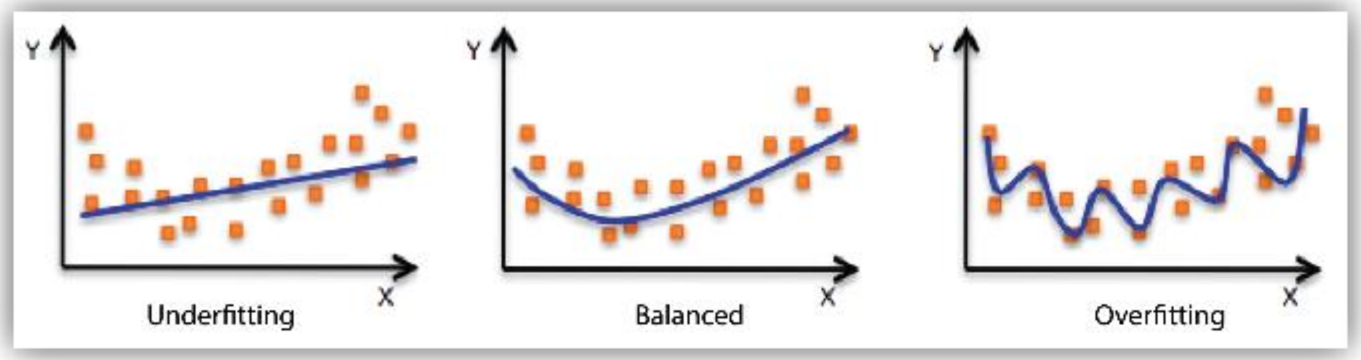
Training/Test error and Generalization



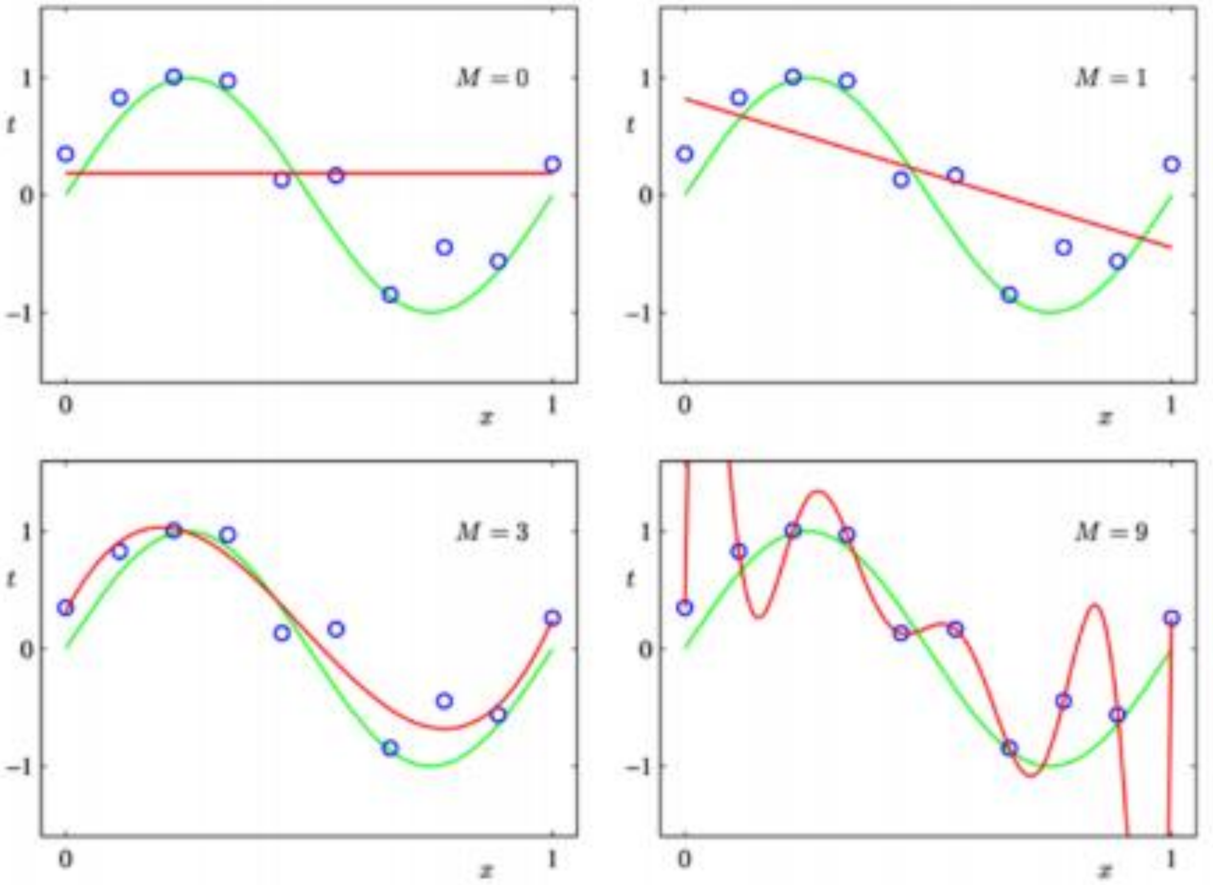
POSTECH

Overfitting and Underfitting

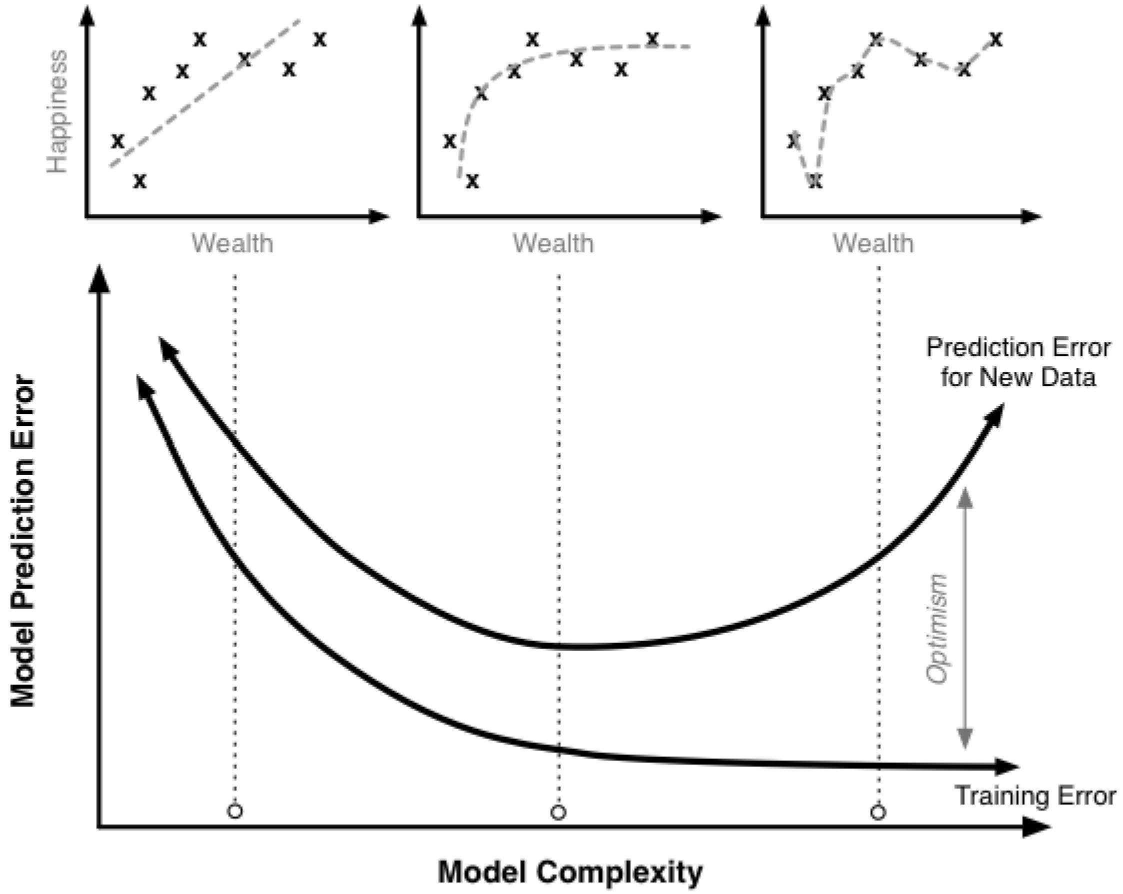
- Underfitting
  - The training data and test data have high error rates.
- Overfitting
  - The training data has a low error rate but the test data has a high error rate.



Overfitting and Underfitting



Overfitting and Underfitting



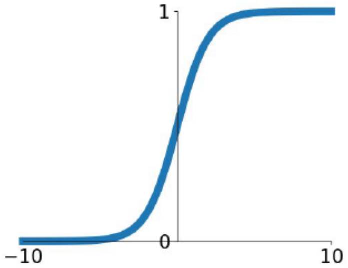
Contents

- Activation function
- Data preprocessing
- Batch Normalization
- Weight initialization
- Fancy optimizers

Activation function

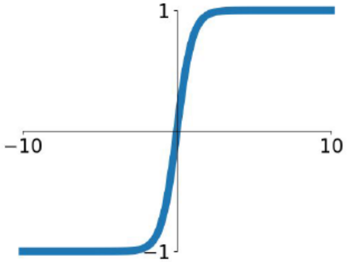
Sigmoid

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



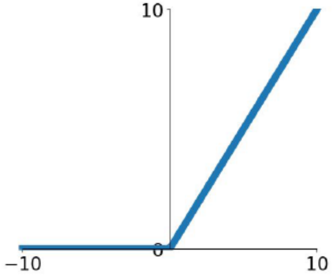
tanh

$\tanh(x)$



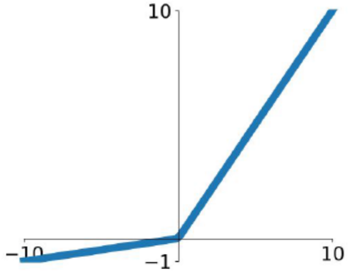
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

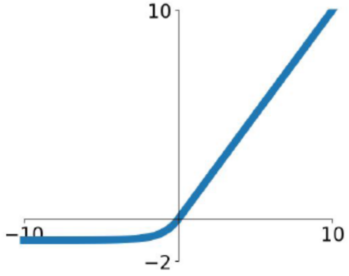


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

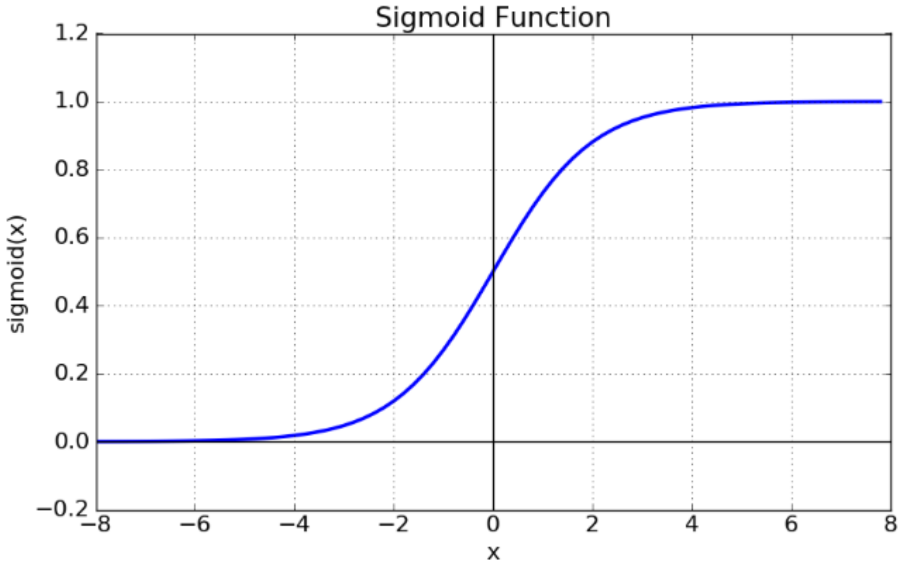
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

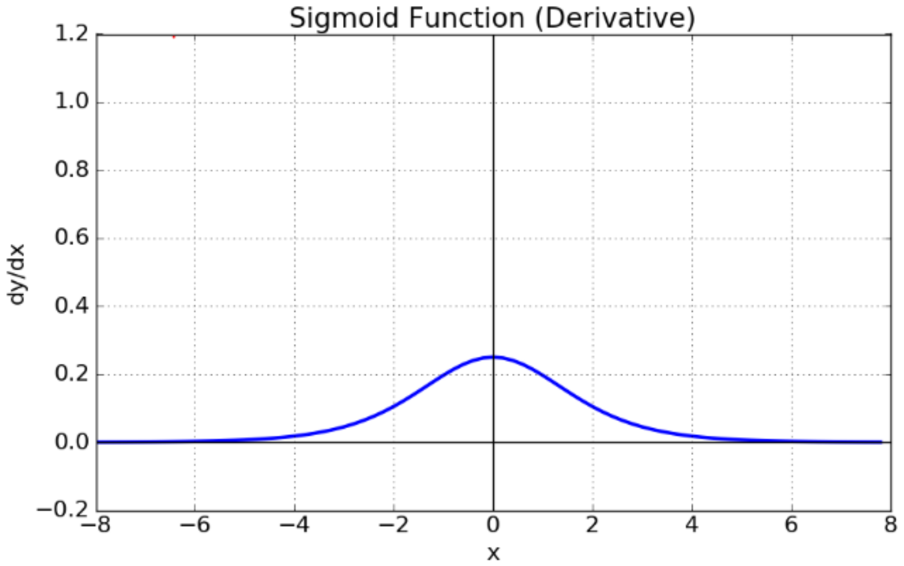


Activation function: Sigmoid

- $\sigma(x) = 1/(1 + e^{-x})$
- range [0, 1]

- Problem
  - Gradient Vanishing
  - Exp() is very expensive.



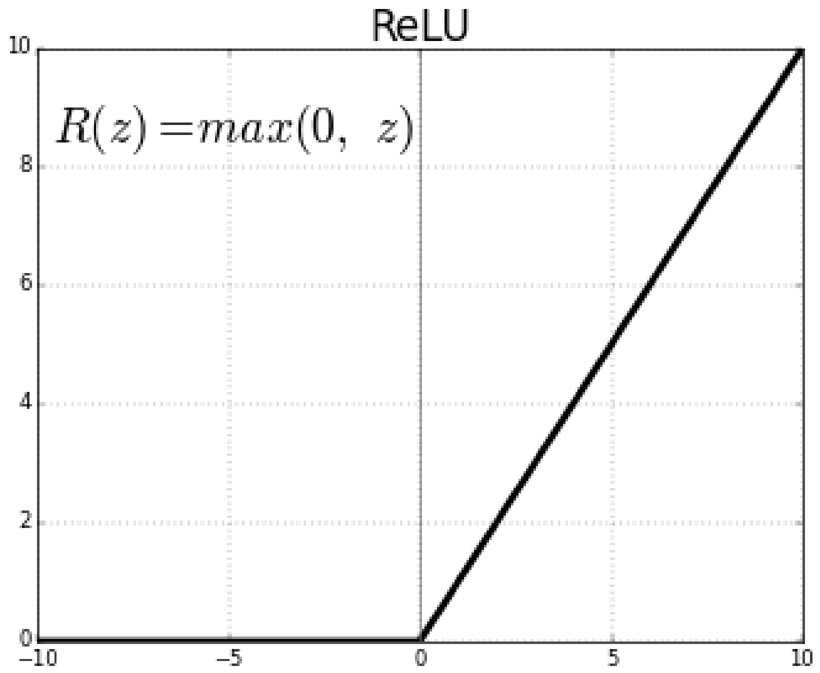


Activation function: ReLU

- f(x) = max(0, x)

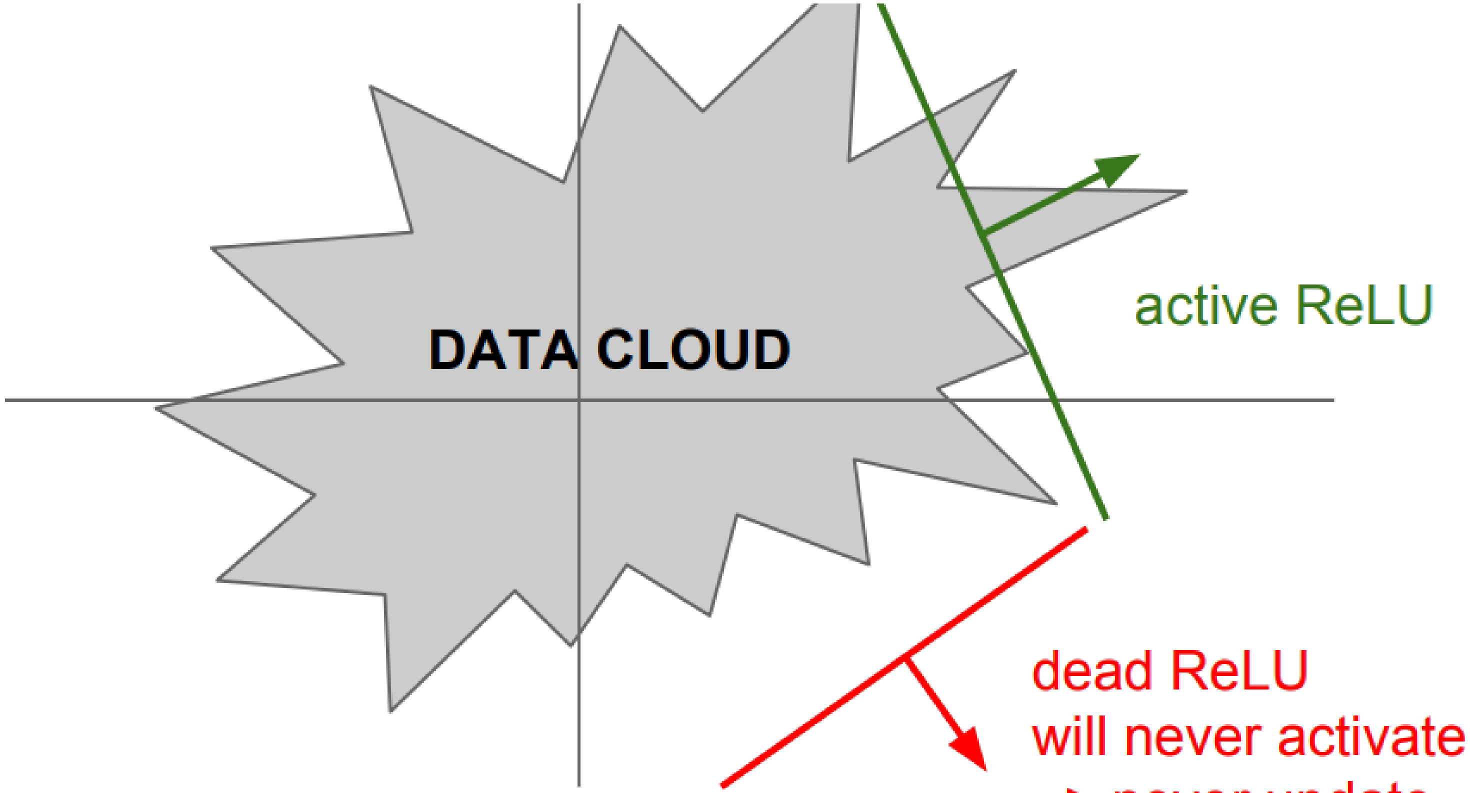
- Does not saturate (in + region)
- Very computationally efficient

- Problem
  - Dead ReLU

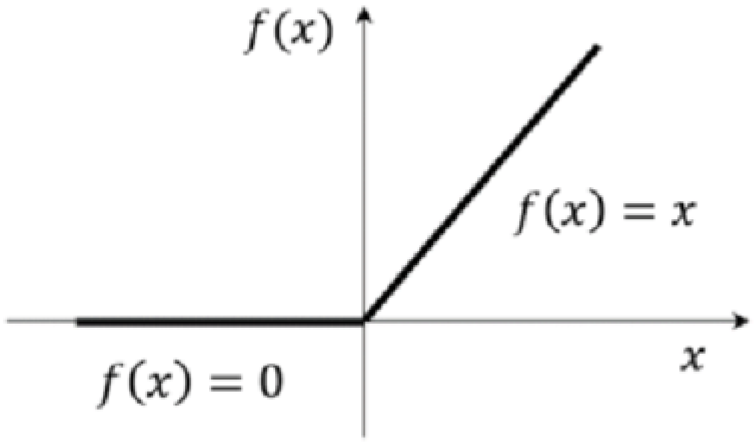


Activation function: ReLU

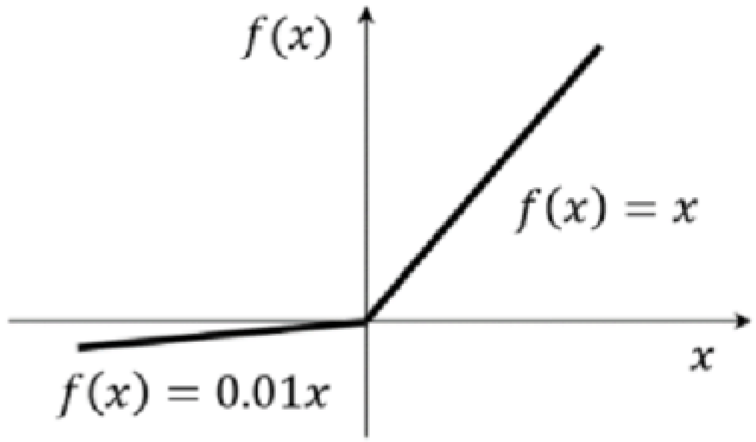
- Dead ReLU



Activation function: LeakyReLU

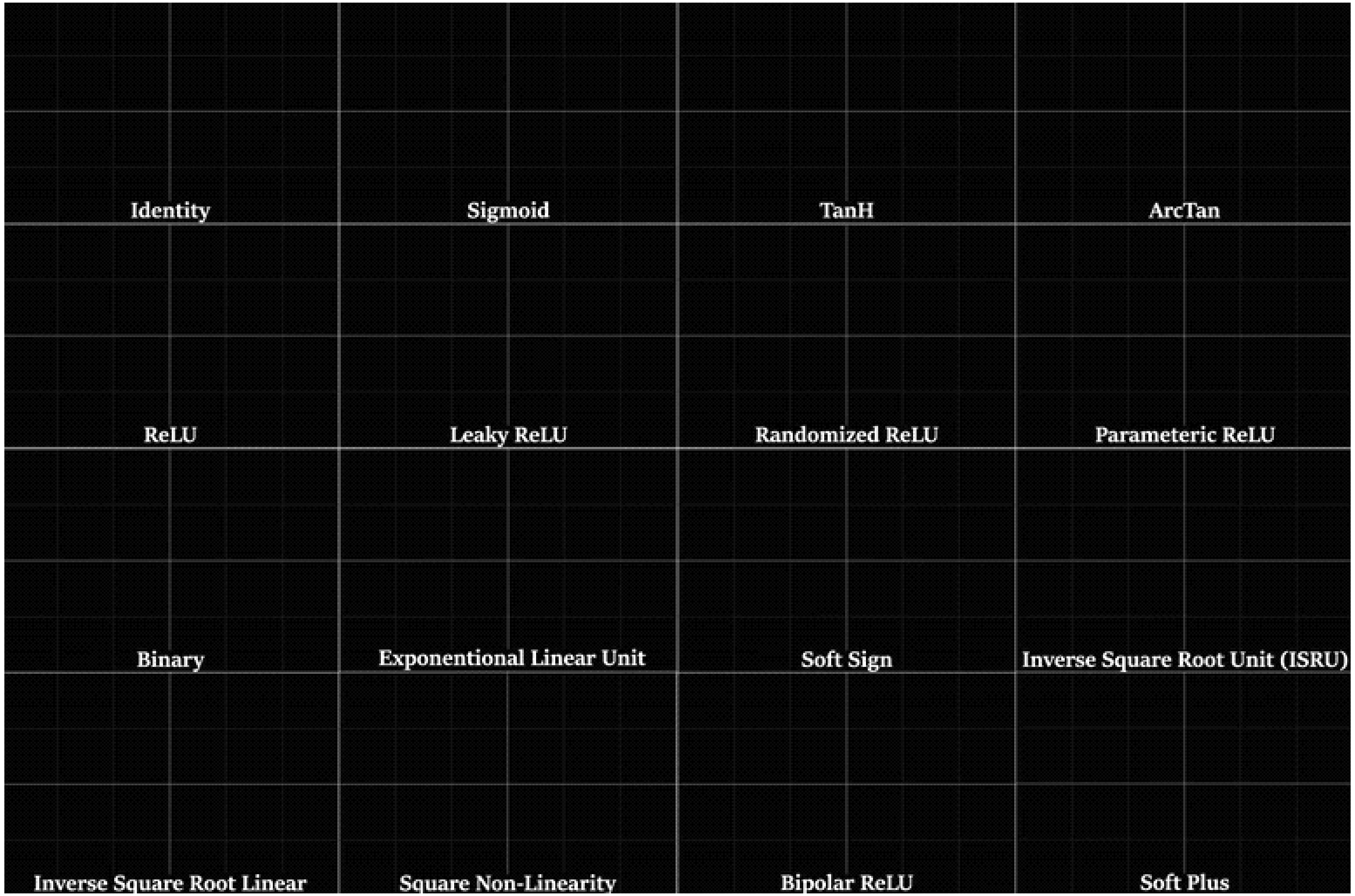


ReLU activation function



LeakyReLU activation function

Activation functions



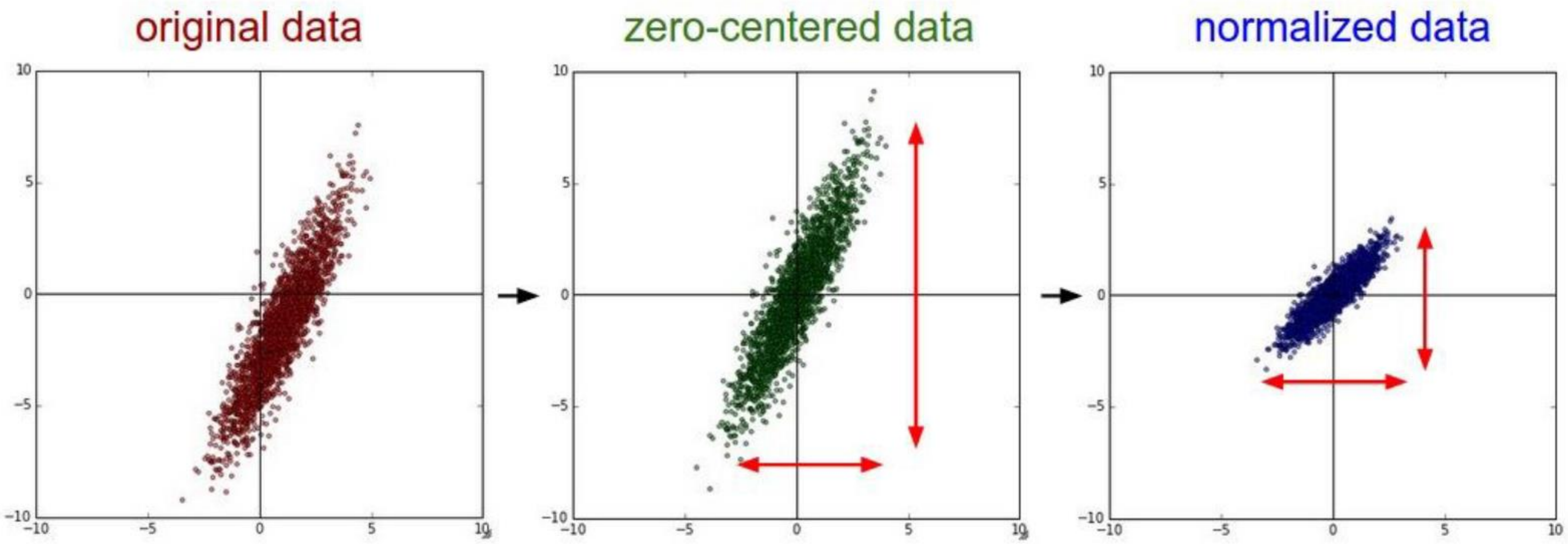
Contents

- Activation function
- Data preprocessing
- Batch Normalization
- Weight initialization
- Fancy optimizers

Data problem: Scale

- data
  - (height(m), weight(kg))
  - ex) (1.5m, 70kg)
- The network will be biased to the ‘weight’.
- To avoid this problem,
  - we must normalize the data.

Data preprocessing



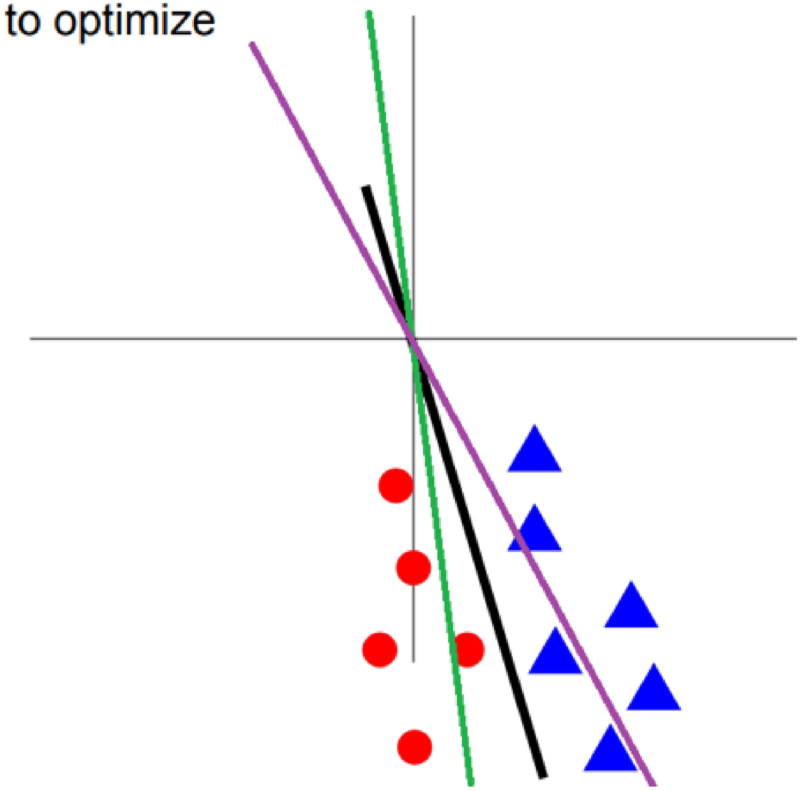

```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

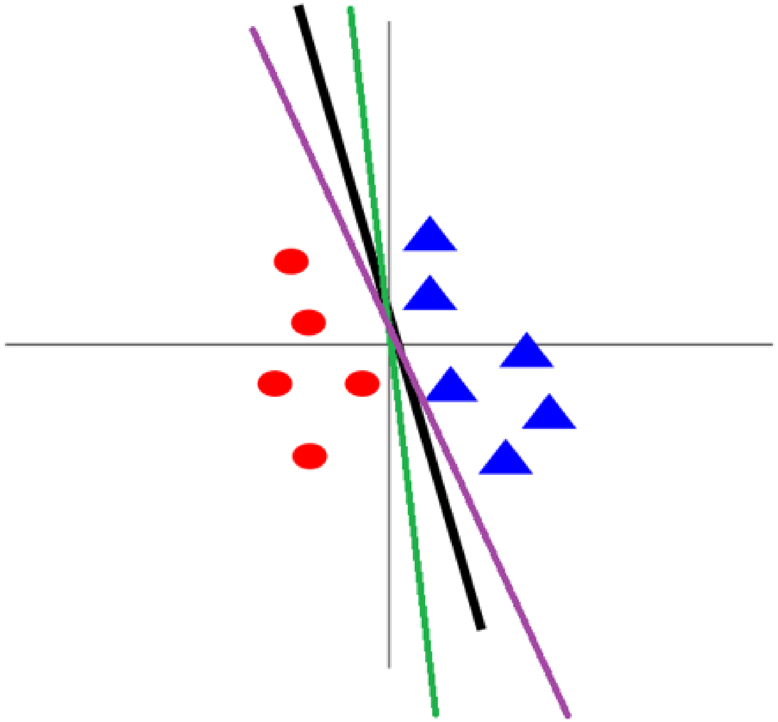
Assume X is data matrix, each sample in a row

Data preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize

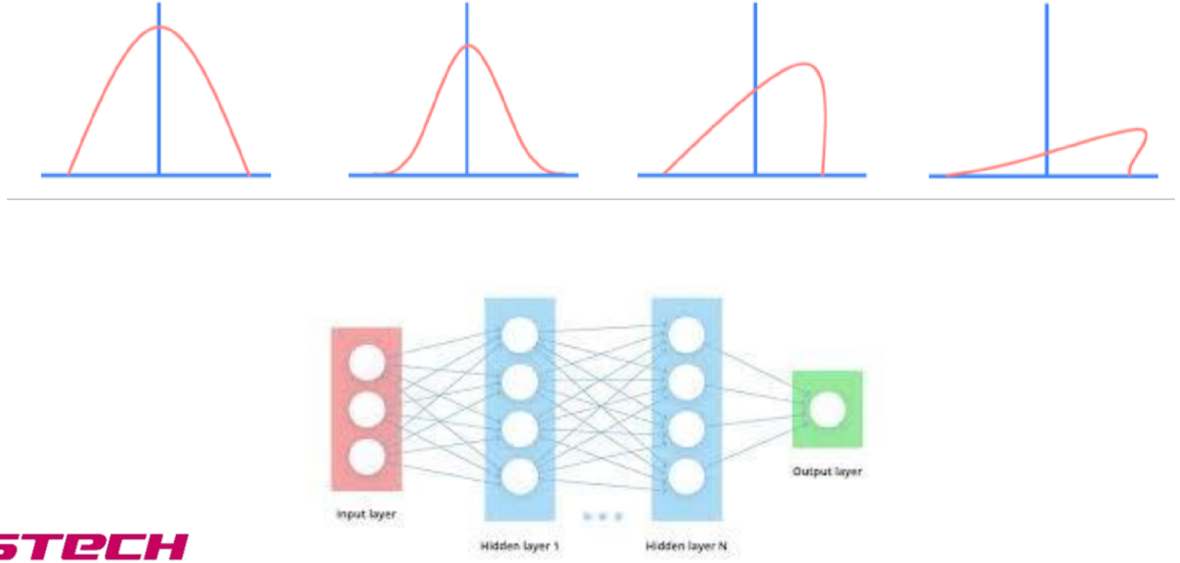


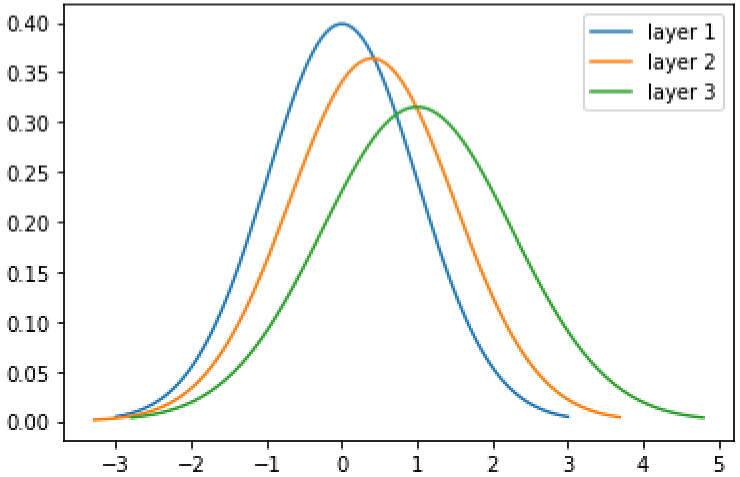
Contents

- Activation function
- Data preprocessing
- Batch Normalization
- Weight initialization
- Fancy optimizers

Covariate shift

- There are unexpected shifts in the distribution of layers' input.





Batch normalization

- Input: $X \in \mathbb{R}^{N \times D}$
- What if zero-mean, unit var is too hard of a constraints?
- Learnable scale and shift parameters:

$$\gamma, \beta \in \mathbb{R}^D$$

- Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{ij}$$

Per-feature mean. Shape is *D*

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{ij} - \mu_j)^2$$

Per-feature var. Shape is *D*

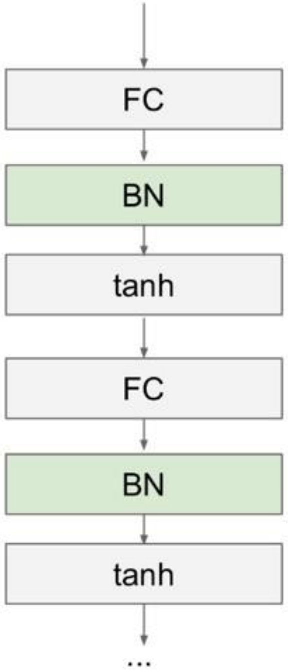
$$\hat{x}_{ij} = \frac{x_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalized x. Shape is *N*×*D*

$$y_{ij} = \gamma_j \hat{x}_{ij} + \beta_j$$

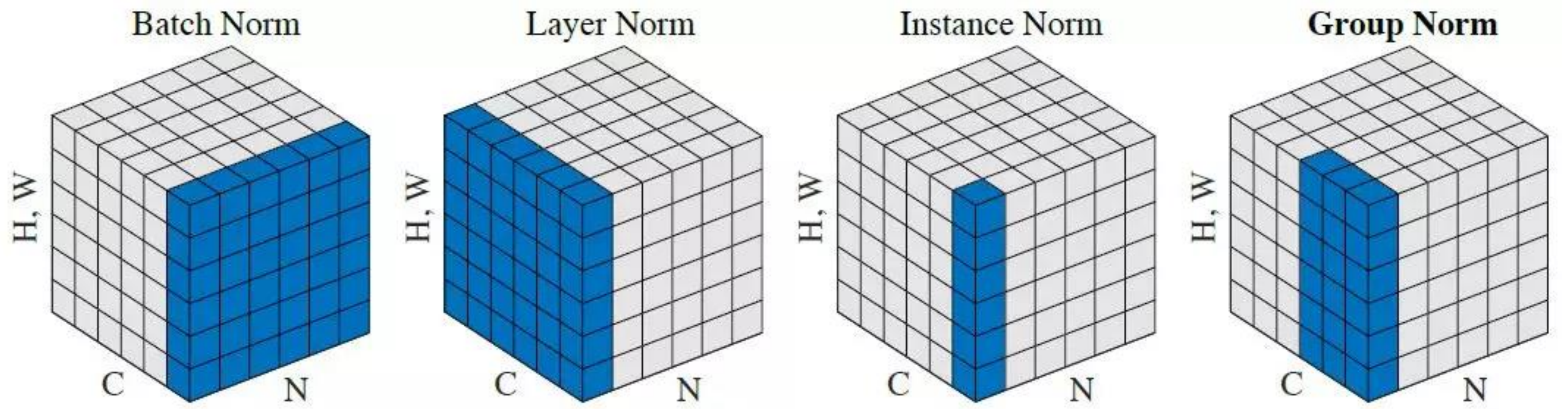
Output. Shape is *N*×*D*

Batch normalization



- Makes deep networks much easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this
  is a very common source of bugs!

Other normalization techniques



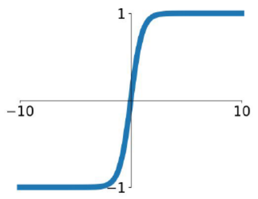
Contents

- Activation function
- Data preprocessing
- Batch Normalization
- Weight initialization
- Fancy optimizers

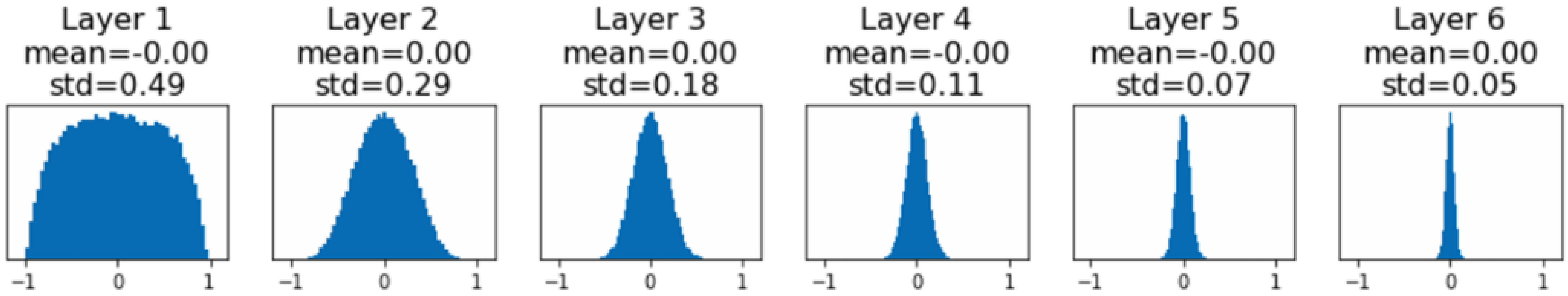
Weight initialization

- Init weights with $\mathcal{N}$**(0, 0.01)** with **tanh(x)**,

tanh
$\tanh(x)$



- All zero, no learning!



POSTECH

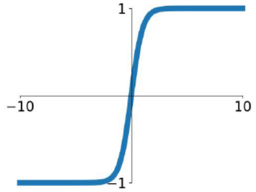
Weight initialization

- Init weights with $\mathcal{N}$**(0, 1)** with **tanh(x)**,

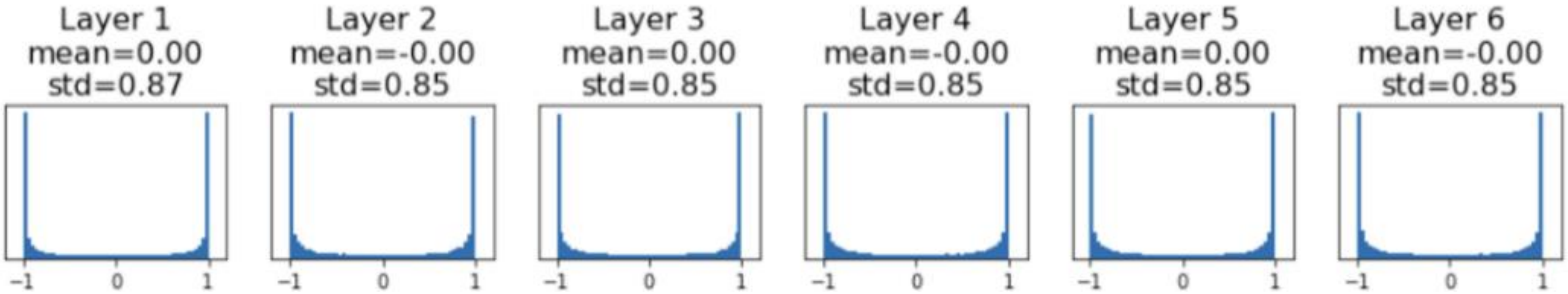
- Local gradient all zero, no learning!

tanh

$\tanh(x)$

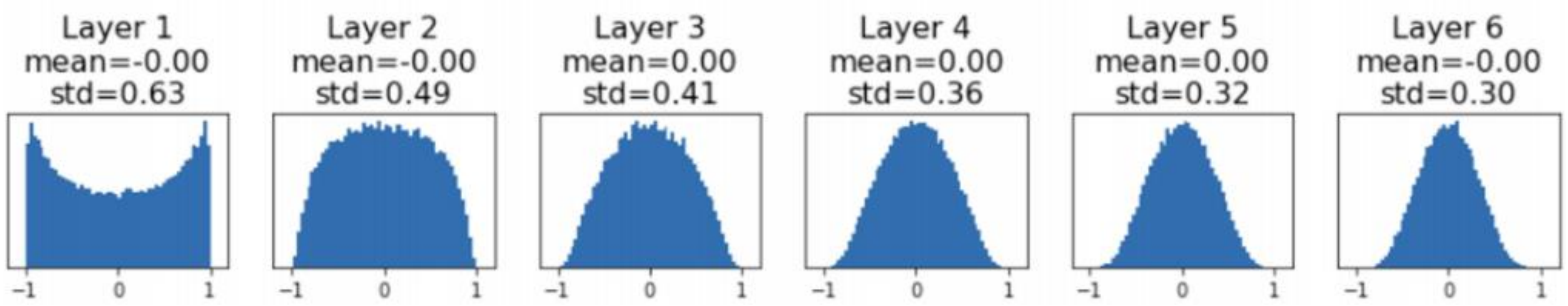


$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$



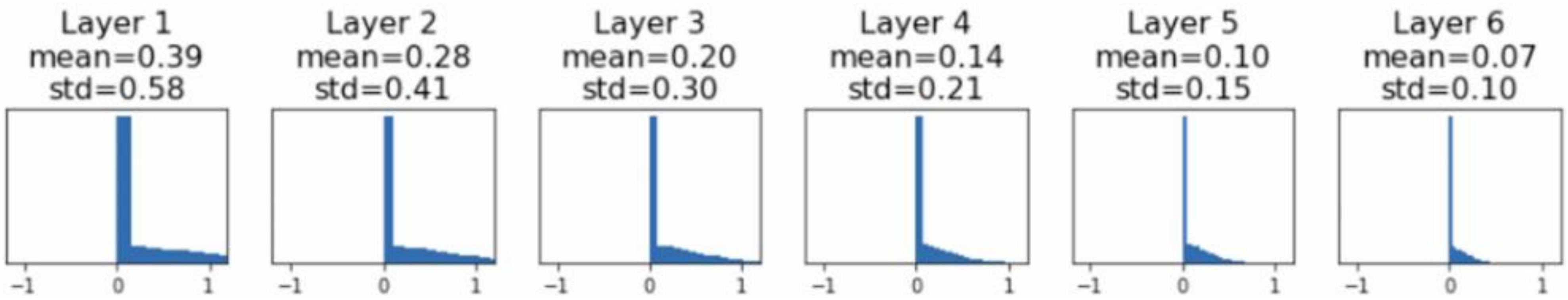
Xavier initialization

- Init weights with $\mathcal{N}$**(o, Var(W))** with **tanh(x)**,
  - Var(W) = $\frac{1}{\sqrt{D_{in}}}$
  - D_{in} : the size of the dimension of input



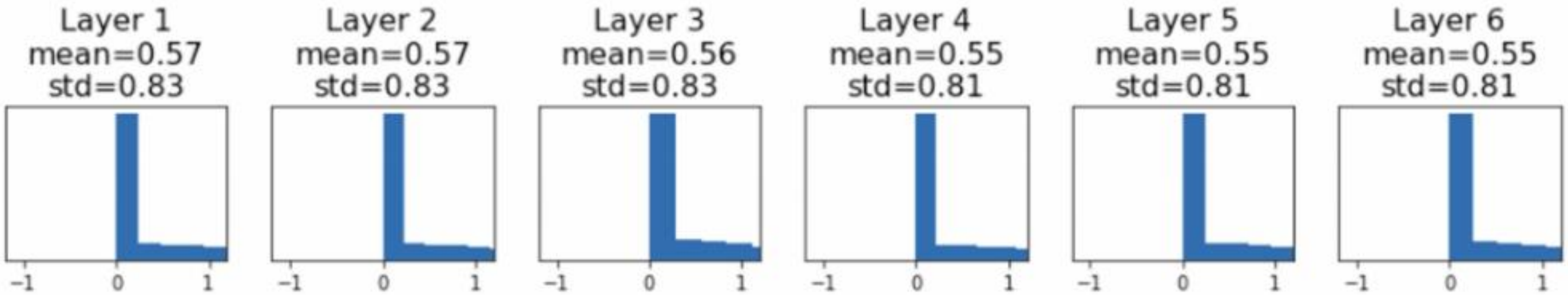
Xavier initialization

- Init weights with $\mathcal{N}$**(o, Var(W))** with **tanh(x)**,
  - Var(W) = $\frac{1}{\sqrt{D_{in}}}$
  - D_{in} : the size of the dimension of input



He initialization

- Init weights with $\mathcal{N}$**(0, Var(W))** with **tanh(x)**,
  - Var(W) = $\frac{\mathbf{2}}{\sqrt{D_{in}}}$
  - D_{in} : the size of the dimension of input

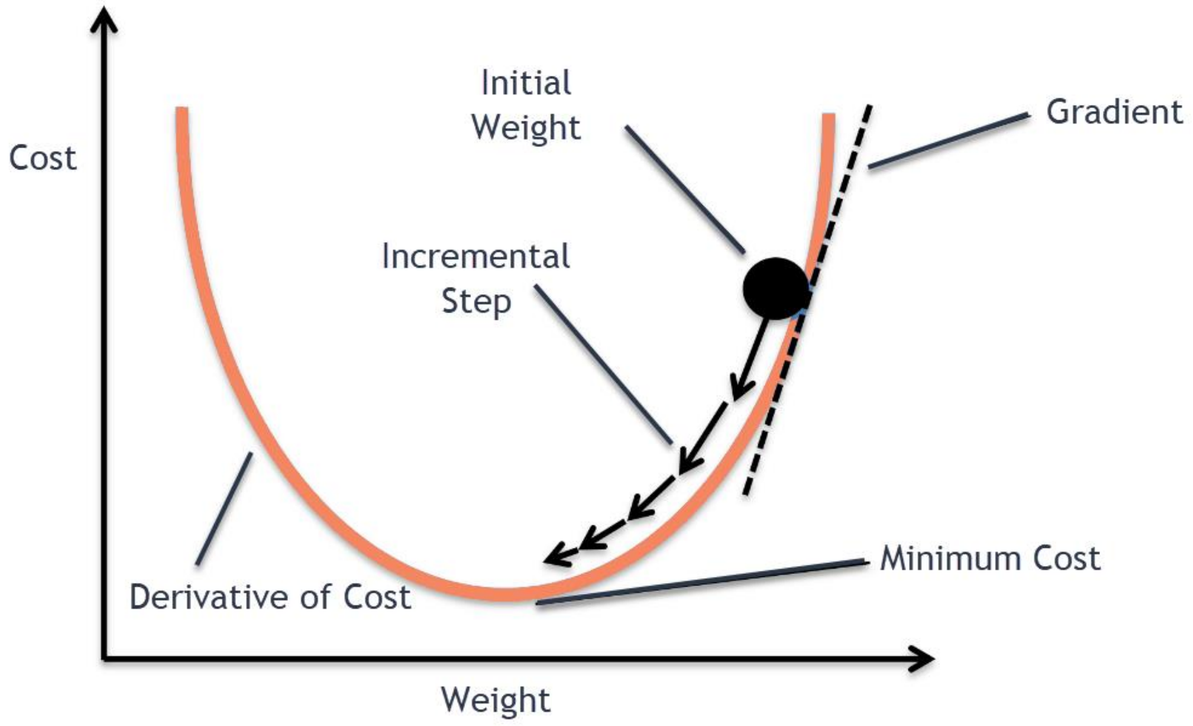


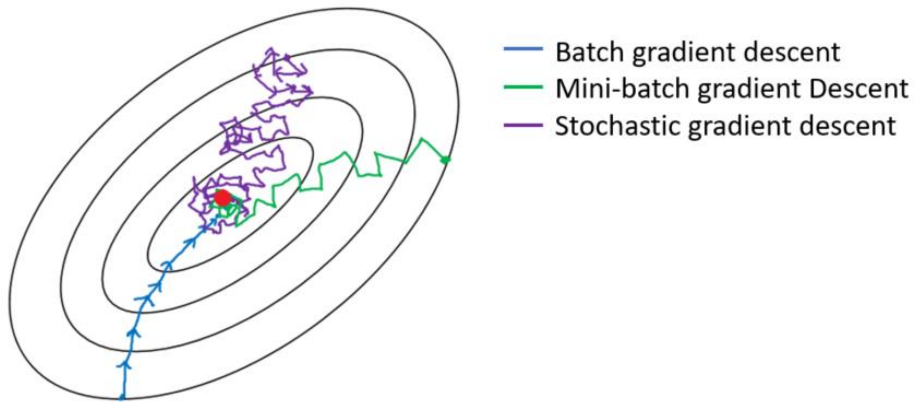
POSTECH

Contents

- Activation function
- Data preprocessing
- Batch Normalization
- Weight initialization
- Fancy optimizers

Stochastic Gradient Descent (SGD)

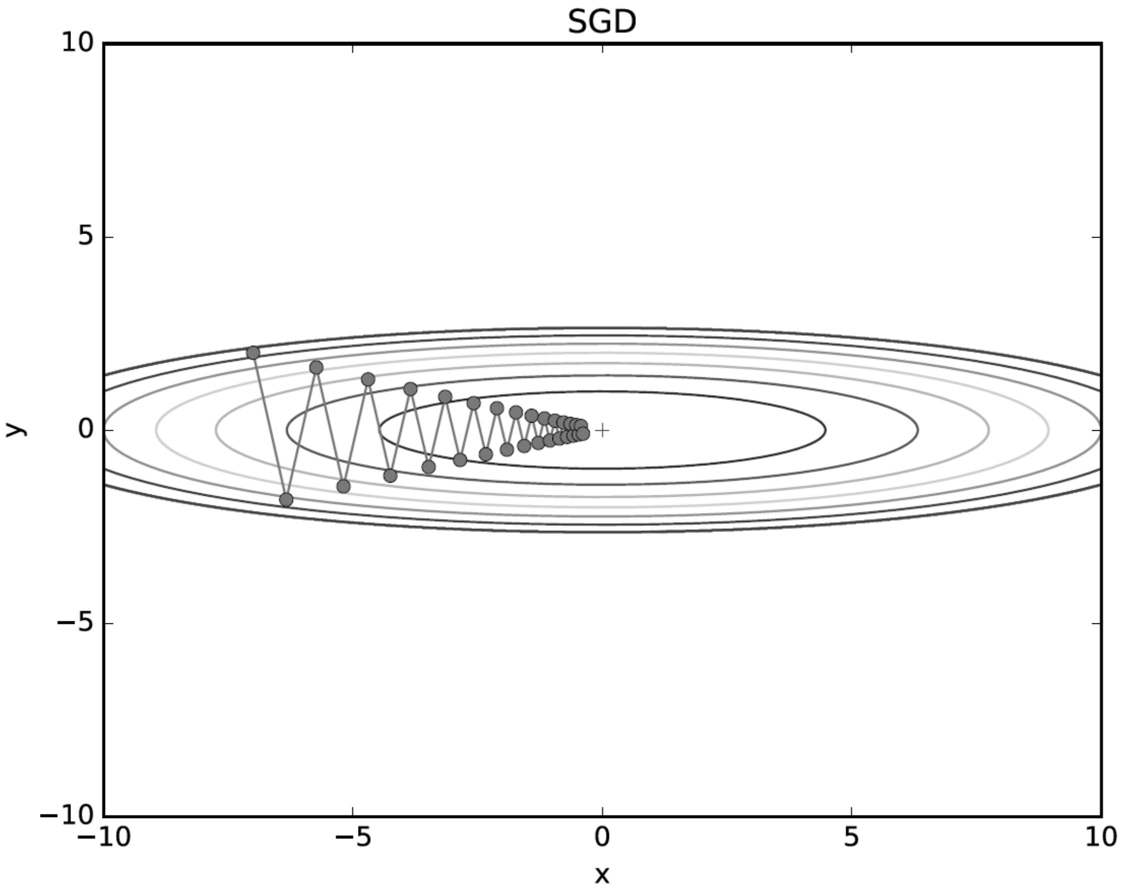




$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

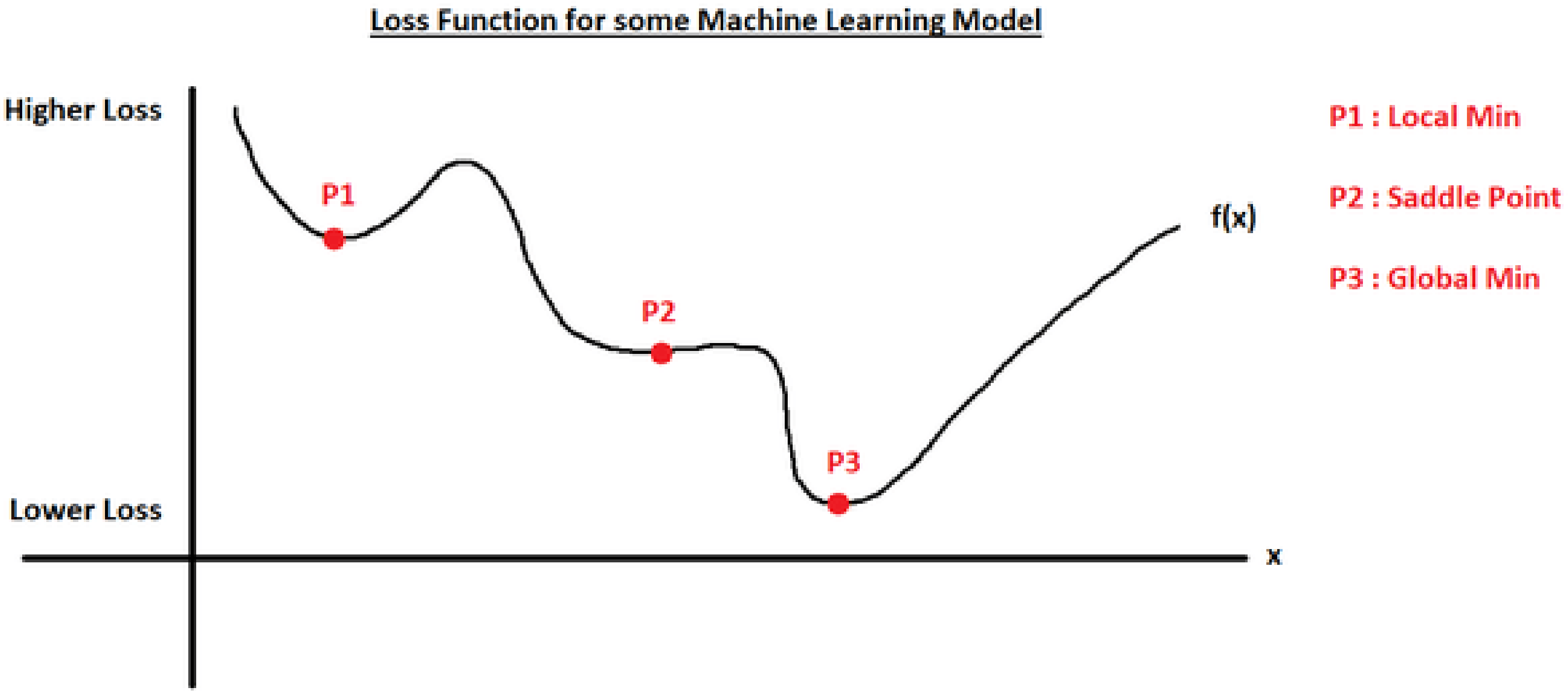
Stochastic Gradient Descent (SGD)

- Problem

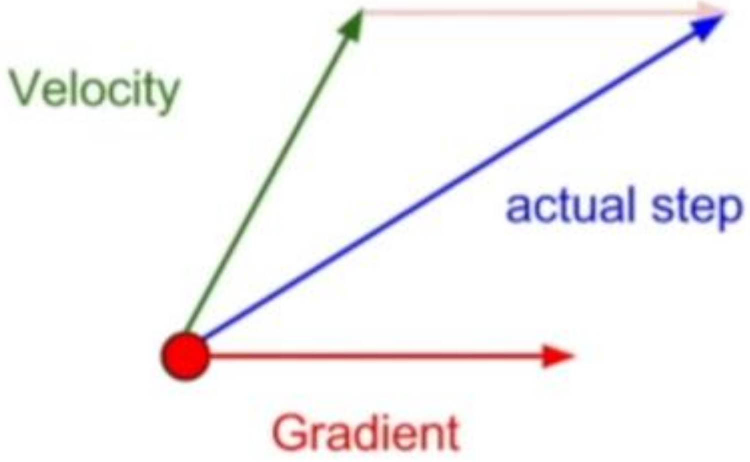


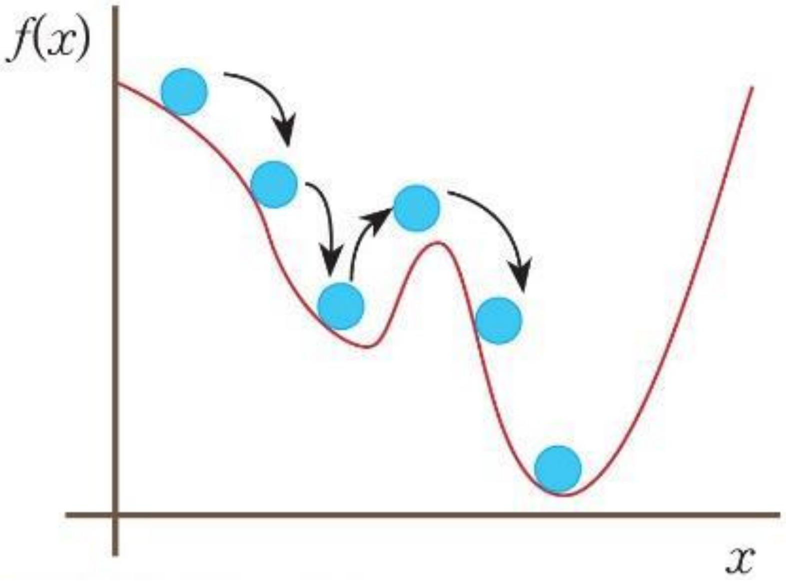
Problem of gradient descent

- Zero gradient, gradient descent gets stuck.



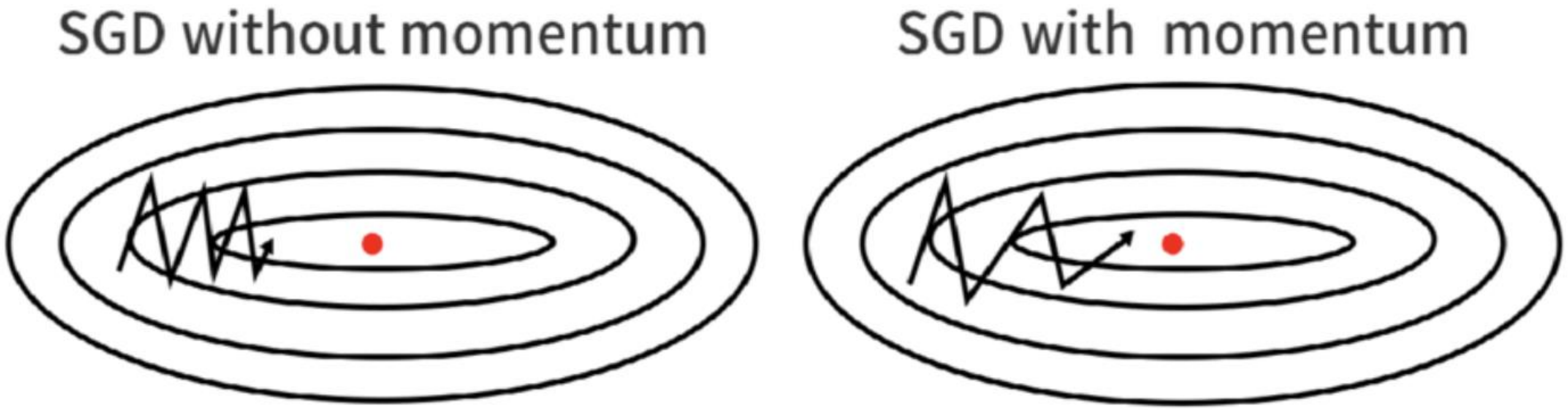
Momentum





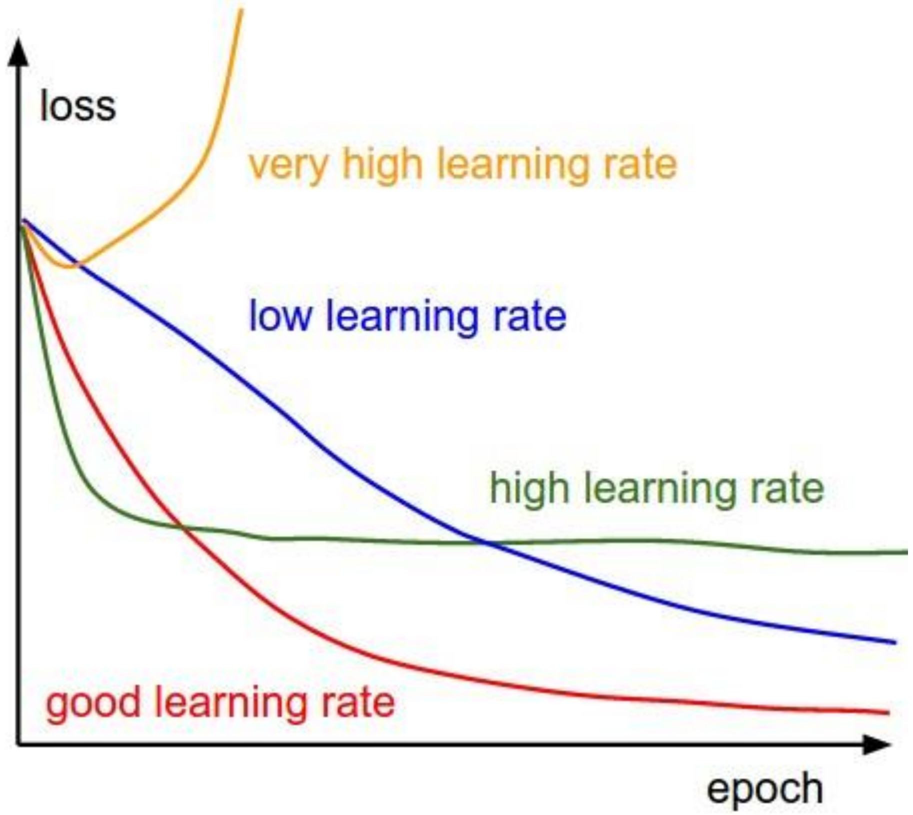
SGD + Momentum

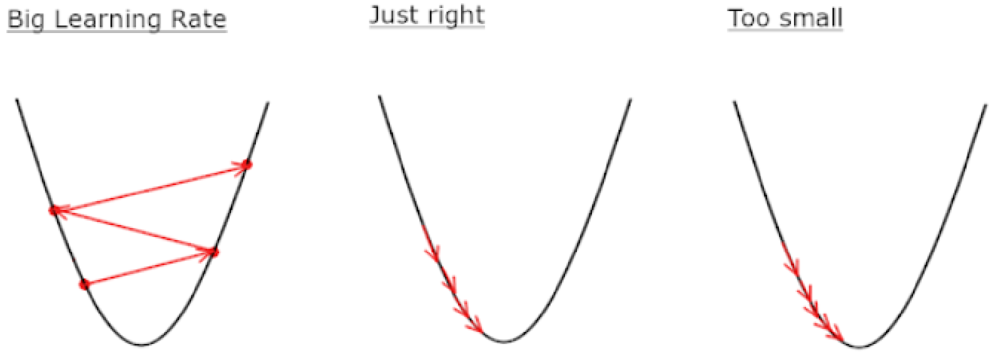
- SGD with momentum is better than only SGD.



$$V_t = \beta V_{t-1} + \alpha \nabla_w L(W, X, y)$$
$$W = W - V_t$$

Importance of learning rate





Fancy optimizer: AdaGrad (Adaptive Gradients)

- Learning rate decay
  - Decay the learning rate for parameters in proportion to their update history (more updates means more decay).

- When h is very big, the parameter will start receiving very small updates.

$$h \leftarrow h + \frac{\partial L}{\partial W} \odot \frac{\partial L}{\partial W}$$

$$W \leftarrow W - \eta \frac{1}{\sqrt{h}} \frac{\partial L}{\partial W}$$

Fancy optimizer: RMSProp

- Everything is very similar to AdaGrad, except now we decay the denominator as well.

$$h \leftarrow h + \frac{\partial L}{\partial W} \odot \frac{\partial L}{\partial W}$$
$$W \leftarrow W - \eta \frac{1}{\sqrt{h}} \frac{\partial L}{\partial W}$$

$$\Longrightarrow$$

$$h_i \leftarrow \rho h_{i-1} + (1-\rho) \frac{\partial L_i}{\partial W} \odot \frac{\partial L_i}{\partial W}$$
$$W \leftarrow W - \eta \frac{1}{\sqrt{h}} \frac{\partial L}{\partial W}$$

Fancy optimizer: Adam (Adaptive moment)

- RMSProp(learning rate decay) + Momentum

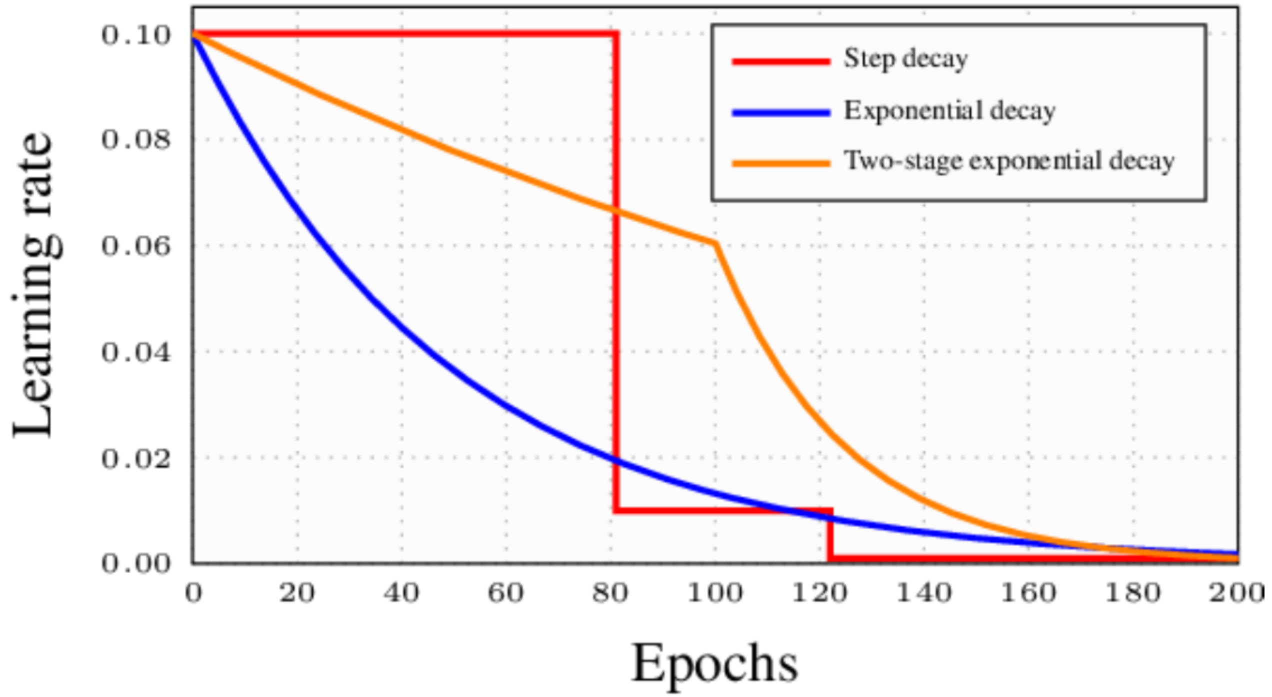
알고리즘 5-5 Adam

입력: 훈련집합 $\mathbb{X}$, $\mathbb{Y}$, 학습률 ρ, 모멘텀 계수 α_1, 가중 이동 평균 계수 α_2
출력: 최적의 매개변수 $\widehat{\boldsymbol{\Theta}}$

```
난수를 생성하여 초기해 Θ를 설정한다.
v = 0, r = 0
t=1
repeat
    그레이디언트 g = ∂J/∂Θ |_Θ 를 구한다.
    v = α₁v − (1 − α₁)g   // 속도 벡터
    v = 1/(1−(α₁)^t) v
    r = α₂r + (1 − α₂)g⊙g   // 그레이디언트 누적 벡터
    r = 1/(1−(α₂)^t) r
    ΔΘ = −ρ/(ε+√r) v
    Θ = Θ + ΔΘ
    t++
until (멈춤 조건)
Θ̂ = Θ
```

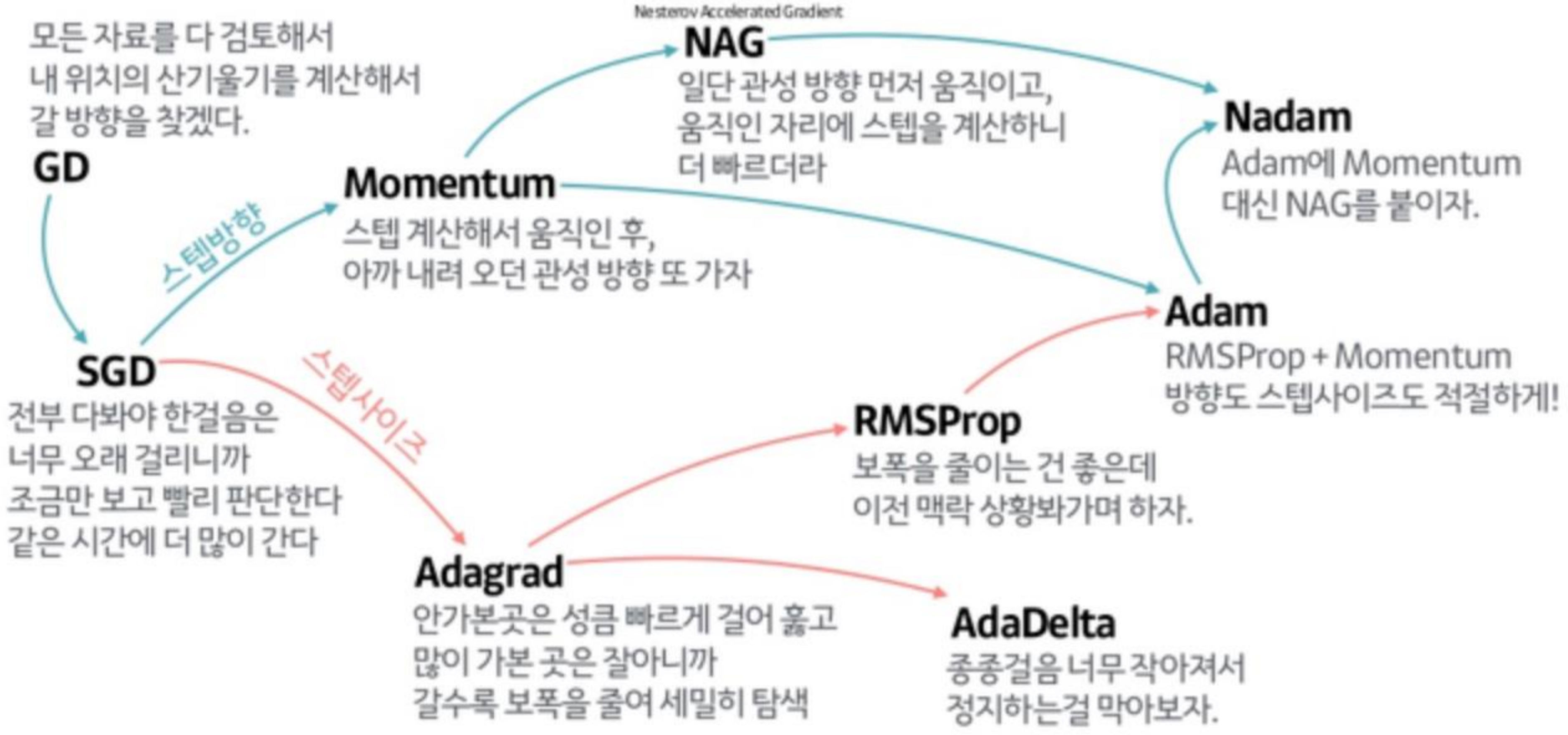
Scheduling learning rate

- SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter.

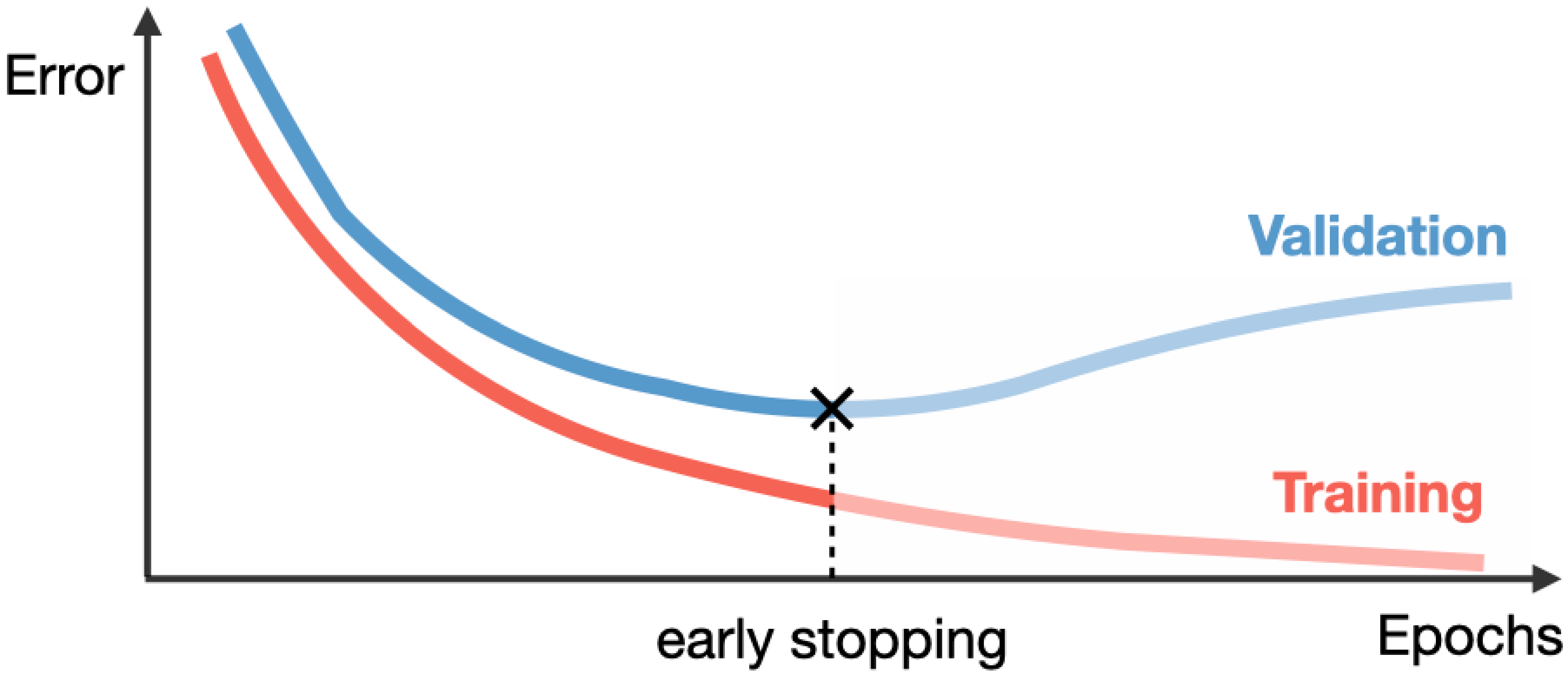


When we train NN, we typically start with large learning rate and decay over time.

Fancy optimizers



Early Stopping



Practice

- Activation

- Batch Normalization

- Weight Initialization

- Optimizers

Activation

Functions

`deserialize(...)` : Returns activation function given a string identifier.

`elu(...)` : Exponential Linear Unit.

`exponential(...)` : Exponential activation function.

`gelu(...)` : Applies the Gaussian error linear unit (GELU) activation function.

`get(...)` : Returns function.

`hard_sigmoid(...)` : Hard sigmoid activation function.

`linear(...)` : Linear activation function (pass-through).

`relu(...)` : Applies the rectified linear unit activation function.

`selu(...)` : Scaled Exponential Linear Unit (SELU).

`serialize(...)` : Returns the string identifier of an activation function.

`sigmoid(...)` : Sigmoid activation function, `sigmoid(x) = 1 / (1 + exp(-x))`.

`softmax(...)` : Softmax converts a vector of values to a probability distribution.

`softplus(...)` : Softplus activation function, `softplus(x) = log(exp(x) + 1)`.

`softsign(...)` : Softsign activation function, `softsign(x) = x / (abs(x) + 1)`.

`swish(...)` : Swish activation function, `swish(x) = x * sigmoid(x)`.

`tanh(...)` : Hyperbolic tangent activation function.

https://www.tensorflow.org/api_docs/python/tf/keras/activations

Batch Normalization

```
tf.keras.layers.BatchNormalization(
    axis=-1,
    momentum=0.99,
    epsilon=0.001,
    center=True,
    scale=True,
    beta_initializer='zeros',
    gamma_initializer='ones',
    moving_mean_initializer='zeros',
    moving_variance_initializer='ones',
    beta_regularizer=None,
    gamma_regularizer=None,
    beta_constraint=None,
    gamma_constraint=None,
    **kwargs
)
```

https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization

Weight Initialization

Classes

`class Constant` : Initializer that generates tensors with constant values.

`class GlorotNormal` : The Glorot normal initializer, also called Xavier normal initializer.

`class GlorotUniform` : The Glorot uniform initializer, also called Xavier uniform initializer.

`class HeNormal` : He normal initializer.

`class HeUniform` : He uniform variance scaling initializer.

`class Identity` : Initializer that generates the identity matrix.

`class Initializer` : Initializer base class: all Keras initializers inherit from this class.

`class LecunNormal` : Lecun normal initializer.

`class LecunUniform` : Lecun uniform initializer.

`class Ones` : Initializer that generates tensors initialized to 1.

`class Orthogonal` : Initializer that generates an orthogonal matrix.

`class RandomNormal` : Initializer that generates tensors with a normal distribution.

`class RandomUniform` : Initializer that generates tensors with a uniform distribution.

`class TruncatedNormal` : Initializer that generates a truncated normal distribution.

`class VarianceScaling` : Initializer capable of adapting its scale to the shape of weights tensors.

`class Zeros` : Initializer that generates tensors initialized to 0.

`class constant` : Initializer that generates tensors with constant values.

https://www.tensorflow.org/api_docs/python/tf/keras/initializers

Weight Initialization

- Xavier initialization

```
fc = layers.Dense(128, kernel_initializer=tf.keras.initializers.GlorotNormal)
```

⇩

```
fc = layers.Dense(128, kernel_initializer='glorot_normal')
```

Weight Initialization

- Normal distribution initialization
  - layers.Dense(128, kernel_initializer='normal')

- Xavier initialization
  - layers.Dense(128, kernel_initializer='glorot_normal')

- He initialization
  - layers.Dense(128, kernel_initializer='he_normal')

Optimizers

Classes

`class Adadelta` : Optimizer that implements the Adadelta algorithm.

`class Adagrad` : Optimizer that implements the Adagrad algorithm.

`class Adam` : Optimizer that implements the Adam algorithm.

`class Adamax` : Optimizer that implements the Adamax algorithm.

`class Ftrl` : Optimizer that implements the FTRL algorithm.

`class Nadam` : Optimizer that implements the NAdam algorithm.

`class Optimizer` : Base class for Keras optimizers.

`class RMSprop` : Optimizer that implements the RMSprop algorithm.

`class SGD` : Gradient descent (with momentum) optimizer.

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers

Practice

- With using skills that we learned, try to upgrade the network for better performance!