# Gradient Descent Optimization Methods

## Gradient Descent Optimization Methods

- Introduction of topics in gradient descent when training Deep Learning

- Contents
  - Loss Functions
  - Activation Functions
  - Variants of Gradient Method
  - Advanced Methods in Gradient Update
  - Weight Initialization
  - Other Tips

## Gradient Descent Method

- Deep learning methods are based on gradient method
- Therefore, many work has been done to improve gradient method

1) Improve learning rate: AdaGrad, RMSProp, AdaDelta, Adam, etc
2) Improve error function: MSE, cross-entropy, etc
3) Improve activation function: ReLU, Leaky ReLU, etc
4) Improve by adding additional terms: Regularization, Momentum, NAG, etc
5) etc

$$w \leftarrow w - \overset{1)}{\eta} \frac{\partial}{\partial w} \overset{2),\ 3)}{f(w)} \overset{4)}{+/-\ \alpha}$$

## Loss Functions

- Loss/error/objective function: a method of evaluating how well your algorithm performs in your dataset

($p_i$ : prediction, $y_i$: true/target value)

### Mean Absolute Error (MAE)

$$\frac{1}{n}\sum_i |p_i - y_i|$$

- Not differentiable
  - Use derivative=1 or -1
  - Use differentiable approximation function
- Robust to outliers
- Both for classification and regression

# Loss Functions

## Mean Squared Error (MSE)

$$\frac{1}{n}\sum_{i}(p_i - y_i)^2$$

- Very popular. Sometime use log form
- Usually better than MAE
- the MSE is great for learning outliers while the MAE is great for ignoring them
- Saturates when using with sigmoid activation function
- usually used for regression problem, but can be used both for classification and regression
- Without 1/n, it becomes L2 regularizer
- Instead of SSE(sum squared error), Keras uses the equivalent mean squared error (MSE), where instead of the sum we compute the average

# Loss Functions

## Cross entropy(CE)

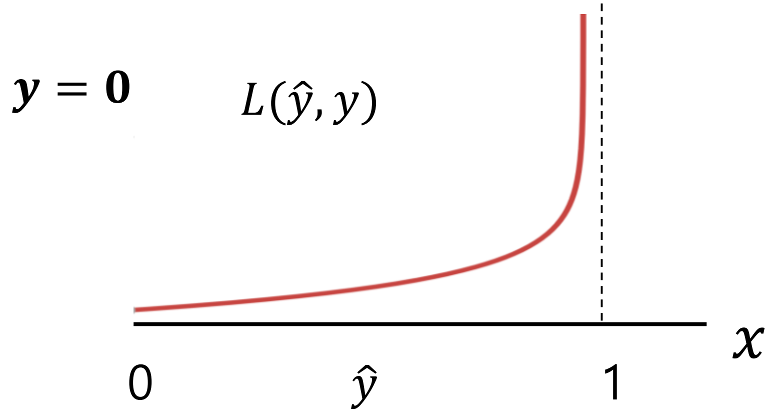
$$H(p, y) = \sum_i -(\boldsymbol{p_i} \log \boldsymbol{y_i})$$

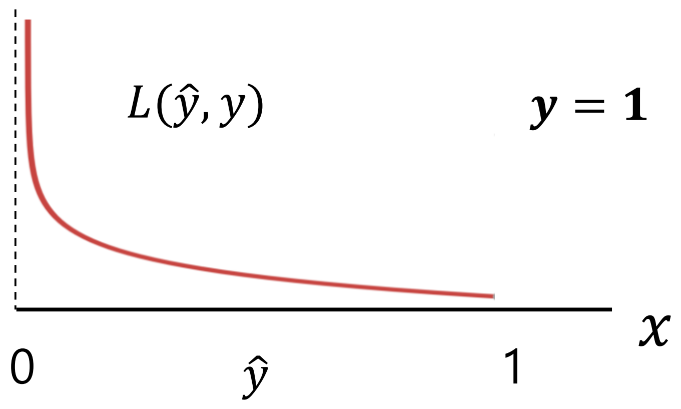
- Binary CE & multi-class(categorical) CE
- Average of self-information of y w.r.t. p
  - $-\log \boldsymbol{y_i}$ : self-information of y
- It penalizes heavily for being *very confident* and *very wrong*
- Default loss function to use for classification problems
- Good for classification of small number of class values
- Good regardless of activation function (most popular)
- Faster than MSE
- aka **log loss**

# Loss Functions

## Cross entropy(CE)

- Categorical CE is a good fit for multiclass problems, where the target output is a one-hot vector. (aka Logistic loss or Log loss)
- Mathematically, sparse categorical cross-entropy and categorical cross entropy are equivalent, they compute the same thing.
- In Keras (or other ML/DL library):
  - If your target outputs are one-hot vectors, use categorical cross-entropy.
  - If your target outputs are integer class labels, ranging from 0 to (number_of_classes – 1), then use sparse categorical cross-entropy.
- This way you do not have to write code to produce one-hot vectors.





## Loss Functions

### Kullback-Leibler(KL) measure/divergence

$$D_{kL}(y||p) = \sum_i \sum_k (y_{ik} \log(\frac{\boldsymbol{y_{ik}}}{\boldsymbol{p_{ik}}})) = H(p, y) - H(y)$$

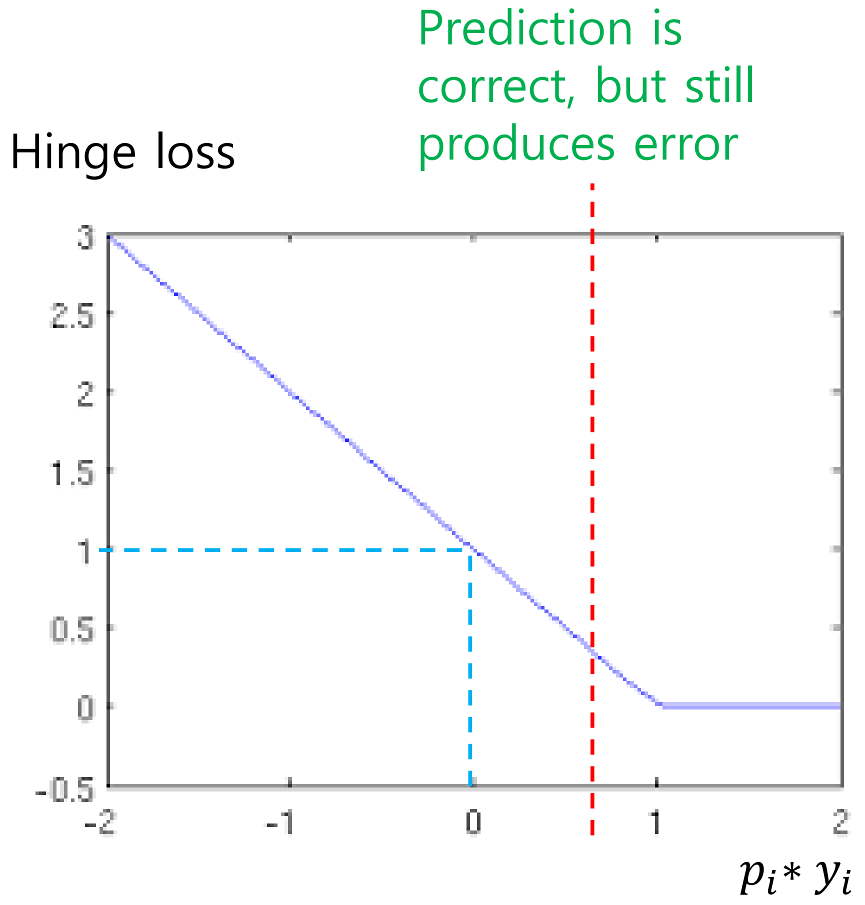
- Measure of how one probability distribution differs from a baseline distribution.
- Kullback-Leibler divergence is asymmetric, meaning that it depends on which distribution is considered the “baseline” distribution and which is considered the “comparison” distribution
- If $H(y)$ (entropy of true values y) is constant, KL is same as cross-entropy(CE)
  - $H(y) = \sum_i -\boldsymbol{y_i} \log \boldsymbol{y_i}$
- If a lot of datasets are only partially labelled or have noisy labels, we could probabilistically assign labels to the unlabelled portion of a dataset: KL !=CE

# Loss Functions

## Hinge loss

max{0, (1- $p_i * y_i$)}

- Intended for use with binary classification {-1, 1}
- attempting to ensure that each point is correctly and confidently classified
- The score of correct prediction should be greater than the score of incorrect predictions **by some margin (e.g.: 1)**
- When $p_i$ and $y_i$ have **opposite** signs, the Hinge loss increases linearly with $p_i$
- When $p_i$ and $y_i$ have the **same** sign ($p_i$ predict is correct)
  - If $|p_i| \geq 1$, the Hinge loss=0.
  - If $|p_i| < 1$, the Hinge loss increases linearly with $p_i$ (correct prediction, but not confident enough),
- Used for maximum-margin classification(e.g.: SVM)
- Gives high penalty for wrong answers
- Not differentiable, but convex
- Squared hinge loss is also available



# Loss Functions

## Hinge loss

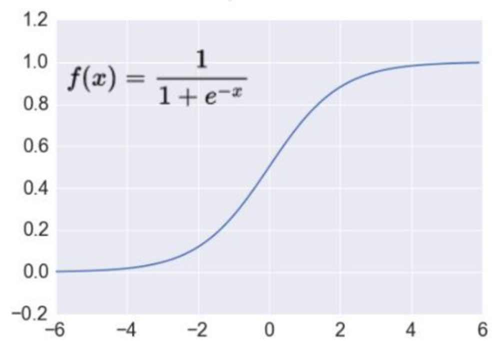
Hinge loss = max{0, (1- $p_i * y_i$)}

| ID | actual($y_i$) | predicted($p_i$) | Hinge loss |
|---|---|---|---|
| 0 | 1 | 0.97 | 0.03 |
| 1 | 1 | 1.20 | 0 |
| 2 | 1 | 0.00 | 1 |
| 3 | 1 | -0.25 | 1.25 |
| 4 | -1 | -0.88 | 0.12 |
| 5 | -1 | -1.01 | 0 |
| 6 | -1 | 0.00 | 1 |
| 7 | -1 | 0.40 | 1.4 |

**Total Hinge loss = 4.8**

# Activation Functions

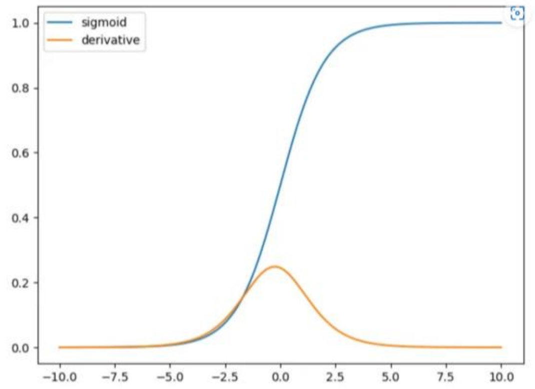
## Sigmoid



Takes a real-valued number and "squashes" it into range between 0 and 1.

$$R^n \rightarrow [0,1]$$

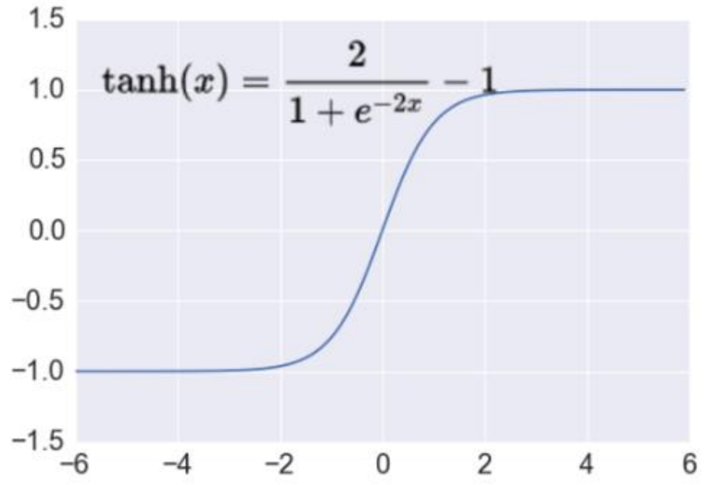
$$f'(x) = \sigma(x)(1 - \sigma(x))$$



- When activation value is near 0 or 1, the gradient is almost zero, causing vanishing gradient problem
- If the initial weights are too large then most neurons would become saturated and the network will barely learn.
- Slow in convergence (computationally expensive)
- If the data coming into a neuron is always positive, then the gradient on the weights, during backpropagation, will become either all be positive, or all negative (Not zero-centered)
- Can be used in output layer of classification (ranges between 0 and 1)
- It is especially used for models where we have to predict the probability as an output
- The function is monotonic but function's derivative is not.

# Activation Functions

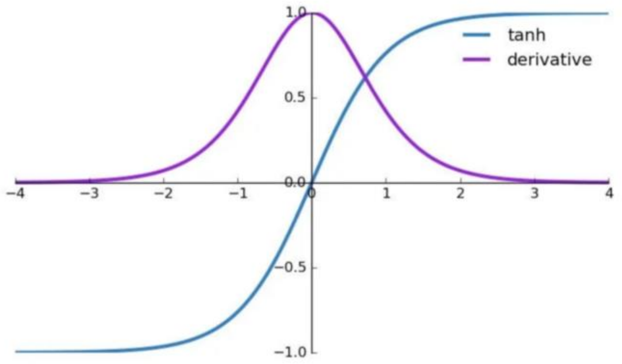
## Tanh



Takes a real-valued number and "squashes" it into range between -1 and 1.

$$R^n \rightarrow [-1,1]$$

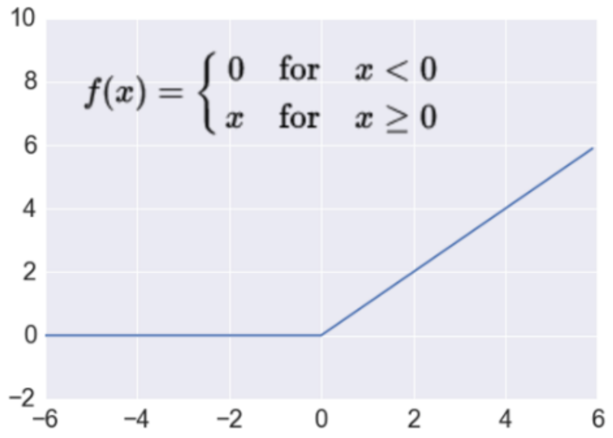
$$f'(x) = 1 - (tanh\,(x))^2$$



- Very similar to sigmoid
- Can be used in output layer of classification (ranges between -1 and 1)
- Like sigmoid, tanh neurons can saturate
- Unlike sigmoid, output is zero-centered
- Tanh is a scaled sigmoid: $tanh(x) = 2sigm(2x) - 1$
- In practice, the tanh non-linearity is preferred to the sigmoid nonlinearity
- Gradient is stronger than sigmoid which makes the learning faster

# Activation Functions

## ReLU (Rectified Linear Unit)



Takes a real-valued number and thresholds it at zero

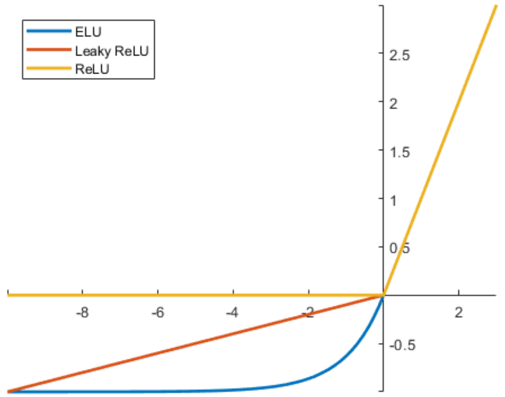
$$f(x) = max(0, x)$$

$$R^n \rightarrow R_+^n$$

- It was found to greatly accelerate (e.g. a factor of 6) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions
- Most popular and easy to implement
- Can use in hidden layer, not in output layer
- In output layer, use sigmoid/tanh (binary) or softmax (multi-class) for classification and linear function for regression problem
- Computationally efficient
- Some ReLU units simply die during training and never reactivate
- Sometimes, ReLU blows up the activation

# Activation Functions

## Leaky ReLU



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

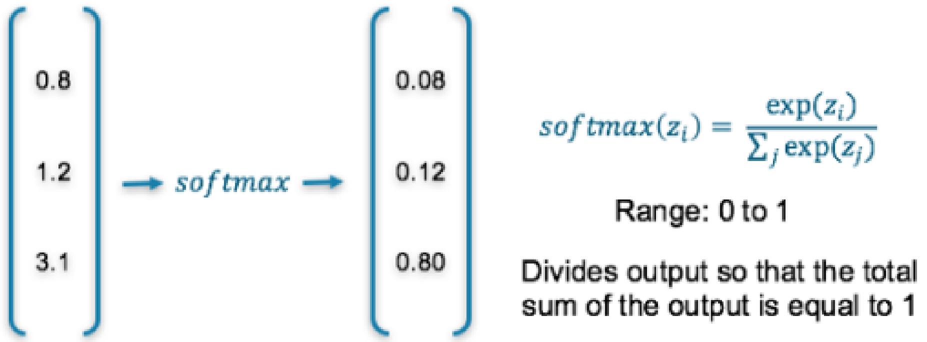
- Solve the problem of dying ReLU
- Advantages to ReLU are unclear
- If you encounter a lot of dead neurons, may use Leaky ReLU
- Some variants
- **1) PReLU**:
  - f(x)=max(αx,x) The slope(α) in the negative region is a parameter
- **2) ELU (Exponential Linear Units)**

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

- Negative part is a curve instead of line

# Activation Functions

## Softmax



$$softmax(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Range: 0 to 1

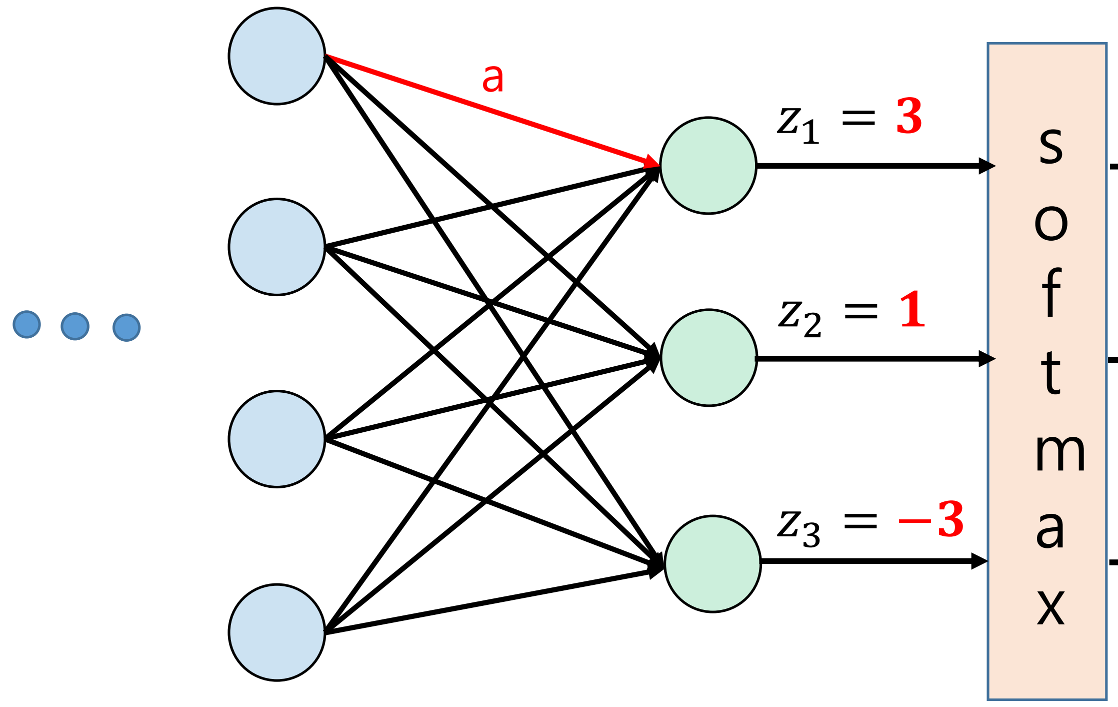
Divides output so that the total sum of the output is equal to 1

- Special function on last layer
- Mostly widely used in multi-class classification problem
- # of input = # of output
- Squashes a *C*-dimensional vector of arbitrary real values to a *C*-dimensional vector of real values in the range (0, 1) that add up to 1.
- Each value ranges between 0 and 1 and the sum of all values is 1 so can be used to model probability distributions. Turns the output into a probability distribution on classes.
- Mimics one-hot-encoding
- Only used in the output layer rather than throughout the network
- For multi-label classification, never use softmax. Use sigmoid instead

# Activation Functions

## Softmax

- The softmax layer applies softmax activations to output a probability value in the range [0, 1]
- Changes in weight a changes value of $z_1$ and thus changes every $y_i$ value.



$y_1 = e^{z_1} \Big/ \sum_{j=1}^{3} e^{z_j} = \mathbf{0.88}$

$y_2 = e^{z_2} \Big/ \sum_{j=1}^{3} e^{z_j} = \mathbf{0.12}$

$y_3 = e^{z_3} \Big/ \sum_{j=1}^{3} e^{z_j} \approx \mathbf{0}$

$$y_i = \frac{\exp(z_i)}{\sum_j \exp(z_i)}$$

***Probability:***
- $0 < y_i < 1$
- $\sum_i y_i = 1$

## Identity(Linear) Activation Function

- Sometimes you may want the equivalent of “no activation function”.
- Mathematically, we want the activation function to leave its input unchanged, so we want the identity function: $f(x) = x$.
- In Keras, this is called the “linear” activation.
- This is also the default activation, which is used if you don’t specify an activation function.
- It is rarely useful to use the “linear” activation in a hidden layer.
- In a sequential model, a sequence of layers with linear activation can be replaced with a single layer that is mathematically equivalent.
- This activation function is used in regression problems
    - E.g., the last layer can have linear activation function, in order to output a real number (and not a class membership)

## Which Activation Function to Choose?

- Begin using the ReLU activation function and proceed to other functions if optimum results are not achieved
- The ReLU activation function should be used only in the hidden layers.
- Sigmoid or Tanh functions should not be used in the hidden layers as training gets affected due to the problem of Vanishing Gradients.

- Choosing functions for the output layer
  - Regression problems: Linear Function
  - Binary Classification: Sigmoid Function
  - Multi-class Classification: Softmax Function
  - Multi-label Classification: Sigmoid Function

- Choosing functions for the hidden layer
  - CNN: ReLU Function
  - RNN: Tanh (or sometimes Sigmoid) Function

## Variants of Gradient Descent

- Three variants of gradient descent, which differ in how much data we use to compute the gradient of the loss(objective) function.

- Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

**1. Batch gradient descent**

- Computes the gradient of the cost function w.r.t. to the parameters for the entire training dataset
- Very slow and is intractable for datasets that don't fit in memory.

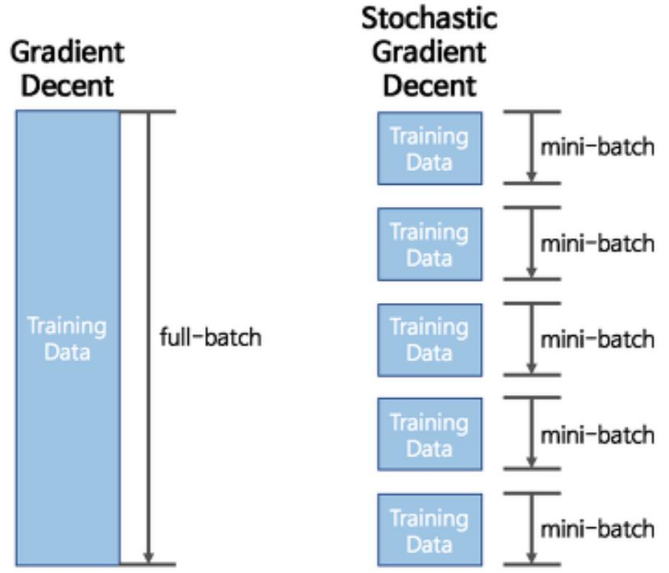
**2. Stochastic gradient descent (SGD)**

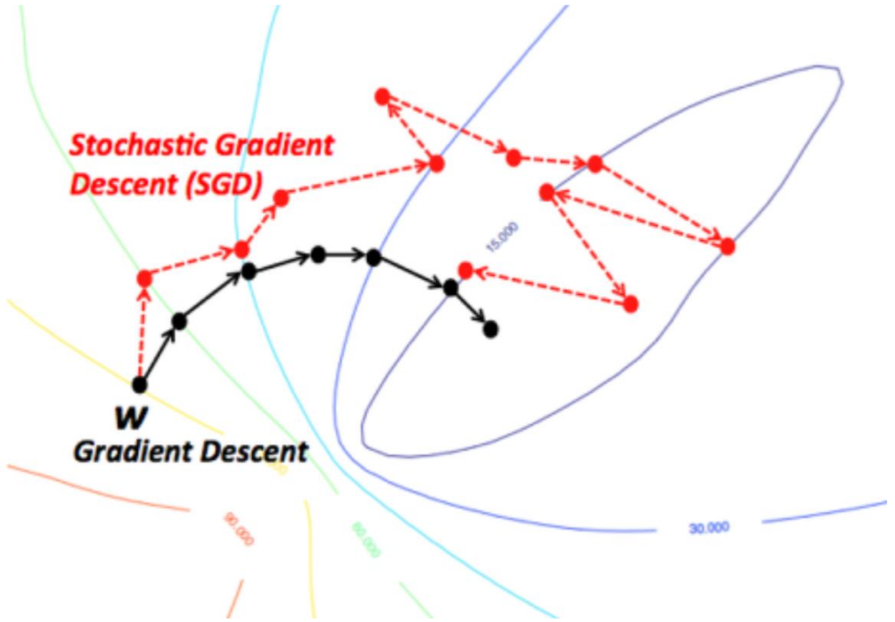
- Performs a parameter update for *each* training example
- Much faster and can also be used to learn online
- Performs frequent updates with a high variance that cause the objective function to fluctuate heavily

# Variants of Gradient Descent

## 3. Mini-batch gradient descent

- Takes the best of both worlds and performs an update for every mini-batch of training examples
- Reduces the variance of the parameter updates, which can lead to more stable convergence
- Can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient.
- Common mini-batch sizes range between 50 and 256, but can vary for different applications.





## Learning Rates in Batch Gradient

- **Batch gradient**
  - Prone to local minimum
  - In (1/N)*(summation of gradient), when N is large, gradient becomes small (close to 0)
  - Therefore, in batch gradient increase learning rate depending on batch size.
  - Because of that, we'll consider that we're talking about the classic mini-batch gradient descent method.
  - When the batch size is multiplied by k, the learning rate should also be multiplied by k, (or sqrt(k)) while other hyperparameters stay unchanged.

- **Stochastic gradient**
  - avoids local minimum

- If we use any adaptive gradient descent optimizer, such as Adam, Adagrad, or any other, there's no need to change the learning rate after changing batch size.

## Methods in Momentum

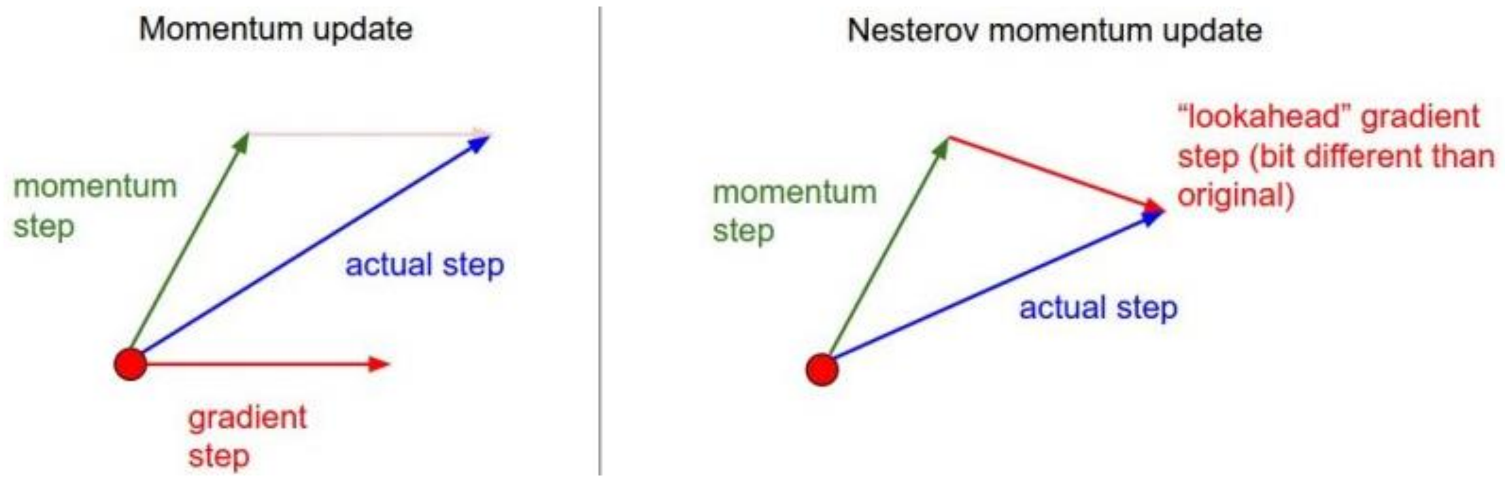
Traditional momentum

$$w_{t+1} = w_t + M_t$$

$$M_t = \alpha M_{t-1} - \eta \nabla_w J(w_t)$$

### Nesterov Accelerated Gradient (NAG)

- Have a smarter ball that knows to slow down before the hill slopes up again
- NAG first makes a big jump in the direction of the previous gradient (green vector), measures the gradient (red vector), which results in the final NAG (blue vector).
- This anticipatory update prevents us from going too fast and results in increased responsiveness

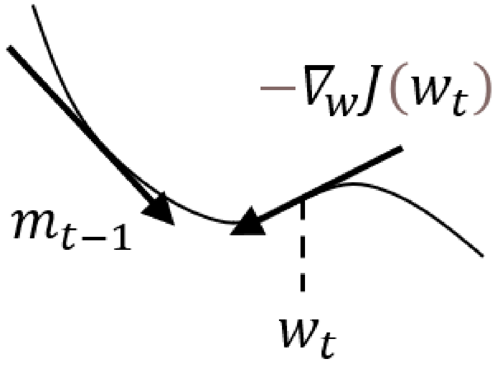


$$w_{t+1} = w_t + M_t$$

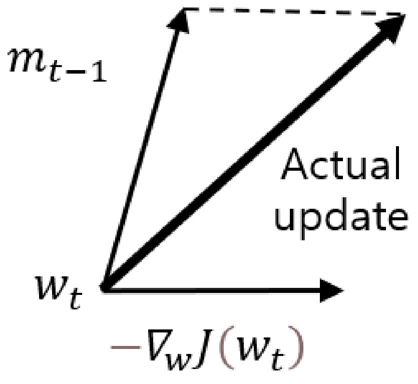
$$M_t = \alpha M_{t-1} - \eta \nabla_w J(w_t + \alpha M_{t-1})$$

# Methods in Momentum

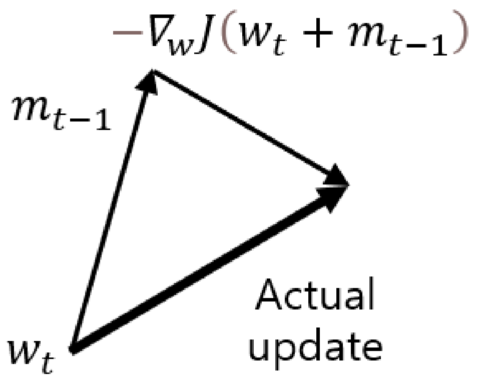
## Nesterov Accelerated Gradient (NAG)



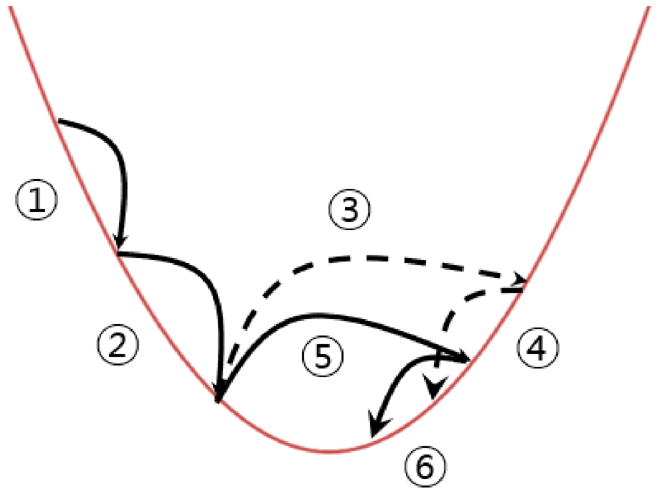
Momentum



Momentum



NAG



$$w_{t+1} = w_t + M_t$$

$$M_t = \alpha M_{t-1} - \eta \nabla_w J(w_t + \alpha M_{t-1})$$

# Methods in Learning Rate

$$w_{t+1} = w_t - \eta \nabla_w J(w_t)$$

## AdaGrad

- Maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. natural language).
- Low learning rates for parameters with frequently occurring features, and high learning rates for parameters with infrequent features
- Well-suited for dealing with sparse data
- $G_t$ : a diagonal matrix where each diagonal element is the sum of the squares of the gradients w.r.t. $w$ up to time step t. (ϵ: a smoothing term)
- Without the square root operation, the algorithm performs much worse.
- The accumulated sum keeps growing, which in turn causes the learning rate to shrink and eventually become infinitesimally small

$$G_t = G_{t-1} + \left(\nabla_w J(w_t)\right)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla_w J(w_t)$$

# Methods in Learning Rate

## RMSProp

- Adaptive learning rate method proposed by Geoff Hinton in his Lecture
- Maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing).
- The algorithm does well on online and non-stationary problems (e.g. noisy)
- RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients.
- $G_t$ is approximately averaging over $1/(1-\gamma)$ previous values.
- Hinton suggests $\gamma$ to be set to 0.9, while a good default value for the learning rate $\eta$ is 0.001

$$G_t = \gamma G_{t-1} + (1-\gamma)\left(\nabla_w J(w_t)\right)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla_w J(w_t)$$

## Methods in Learning Rate

### AdaDelta

- An extension of Adagrad that seeks to reduce its monotonically decreasing learning rate
- Replace the diagonal matrix $G_t$ with the decaying average over past squared gradients (same as RMSProp)
- In addition, learning rate is replaced by a term of exponentially decaying average squared parameter updates
- No need to define learning rate

$$G_t = \gamma G_{t-1} + (1-\gamma)\left(\nabla_w J(w_t)\right)^2 \qquad S_t = \gamma S_{t-1} + (1-\gamma)(\Delta w_t)^2$$

$$\Delta w_t = \frac{\sqrt{S_{t-1} + \epsilon}}{\sqrt{G_t + \epsilon}} \nabla_w J(w_t) \qquad w_{t+1} = w_t - \Delta w_t$$

## Methods in Learning Rate

### Adam(Adaptive Moment Estimation)

- Moment: n-th moment of a random variable is defined as the expected value of that variable to the power of n $m_n = E[X^n]$

- First moment (E[X]) is mean : E[X]
- (Since $Var(X) = E[X^2] - E[X]^2$) Second moment ($E[X^2]$) is uncentered variance (meaning we don't subtract the mean during variance calculation).

- Computes the decaying averages of past and past squared gradients $m_t$ and $v_t$ respectively
- $m_t$ and $v_t$ are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively

# Methods in Learning Rate

## Adam(Adaptive Moment Estimation)

- Combines RMSProp and Momentum

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_w J(w_t)$$ : Momentum method

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla_w J(w_t)\right)^2$$ : RMSProp method

the algorithm calculates an exponential moving average of the gradient and the squared gradient.
The parameters $\beta_1$ and $\beta_2$ control the decay rates of these moving averages.

- Use the following unbiased estimate

$$\widehat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad \widehat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\widehat{v}_t + \epsilon}}\widehat{m}_t$$

In the beginning, $m_0$=$v_0$=0. If we use the normal formula, then the first several values will be too small.
Solve this problem by dividing $m_t$ and $v_t$ by $1-\beta_1^t$ and $1-\beta_2^t$, respectively
Instead of slowly accumulating the first several values, they are now divided by a small number scaling them into larger values.

- The authors propose default values of 0.9 for $\beta_1$, 0.999 for $\beta_2$, and $10^{-8}$ for ϵ.
- Suggested as the default optimization method for deep learning applications.

## Weight Initialization

- Proper initialization of parameters is important
- Don't initialize all weight to 0
- Gaussian Initialization with mean=0, sd=1

### 1) LeCun Initialization

($n_{in}$: number of neurons feeding into it, $n_{out}$: number of neurons the result is fed to)
- Suppose $n = n_{in}$, we want the following $z$ not to be extreme.

$$z = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

- Therefore, the larger n in $z$, the smaller w to be

* LeCun Normal Initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{1}{n_{in}}}$$

* LeCun Uniform Initialization

$$W \sim U(-\sqrt{\frac{1}{n_{in}}},\ +\sqrt{\frac{1}{n_{in}}})$$

## Weight Initialization

### 2) Xavier Initialization

* Xavier Normal Initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

* Xavier Uniform Initialization

$$W \sim U(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \ +\sqrt{\frac{6}{n_{in} + n_{out}}})$$

### 3) He Initialization

* He Normal initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in}}}$$

* He Uniform initialization

$$W \sim U(-\sqrt{\frac{6}{n_{in}}}, \ +\sqrt{\frac{6}{n_{in}}})$$

- Xavier is effective when using sigmoid & tanh and He is effective in ReLU

## Batch Normalization

- In general, Gradient descent converges much faster with feature scaling than without it.

$$h_1 = \sigma(w_1 X), h_2 = \sigma(w_2 h_1) = \sigma\big(w_2 \sigma(w_1 X)\big), h_3 = \cdots$$

- Internal covariate shift

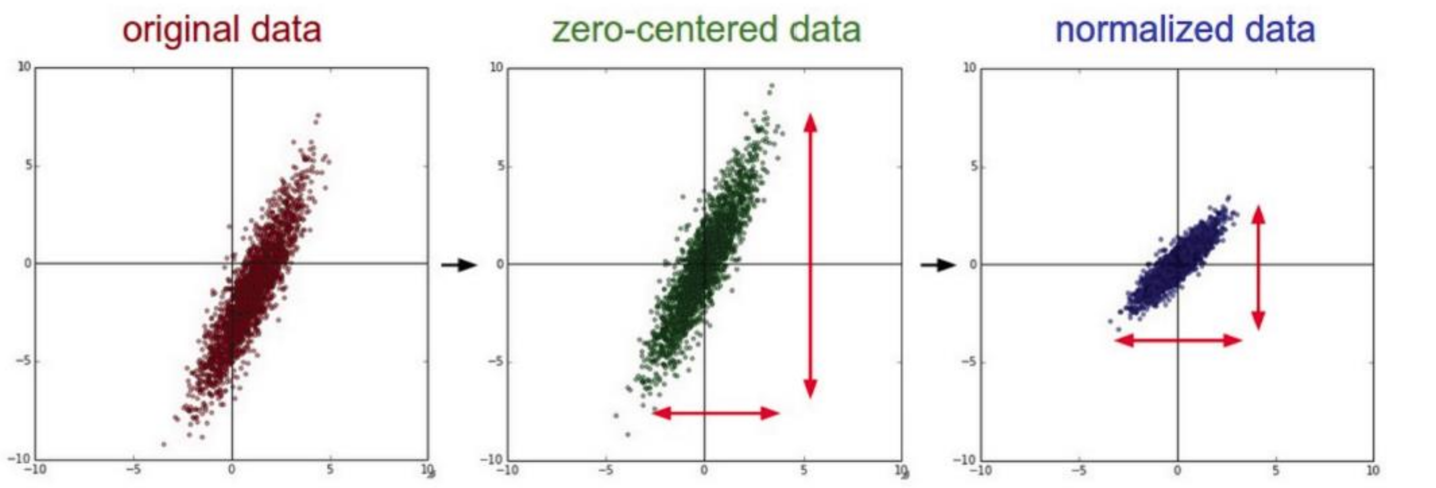
- Batch Normalization (BN) is a normalization method/layer for neural networks

- It consists of **normalizing activation vectors from hidden layers** using the first and the second statistical moments (mean and variance) of the current batch.

- This normalization step is applied right before (or right after) the nonlinear function.

# Normalizing Input Data

- **Data preprocessing** - helps convergence during training
  - Mean subtraction, to obtain zero-centered data
    - Subtract the mean for each individual data dimension (feature)
  - Normalization
    - Divide each feature by its standard deviation
      - To obtain standard deviation of 1 for each data dimension (feature)
    - Or, scale the data within the range [0,1] or [-1, 1]
      - E.g., image pixel intensities are divided by 255 to be scaled in the [0,1] range



Picture from: https://cs231n.github.io/neural-networks-2/

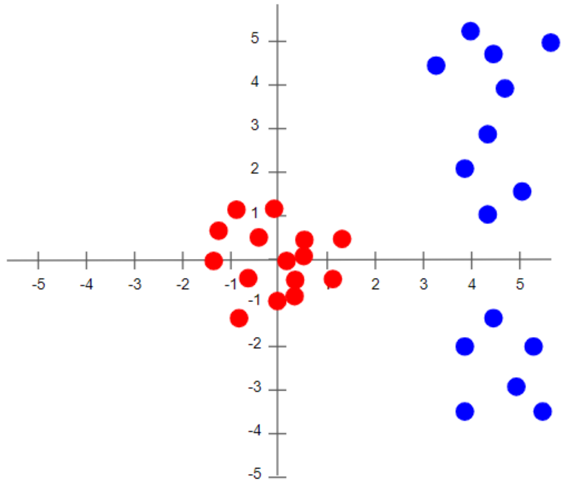
# Normalizing Input Data

- Normalizing data (Z-score normalization)

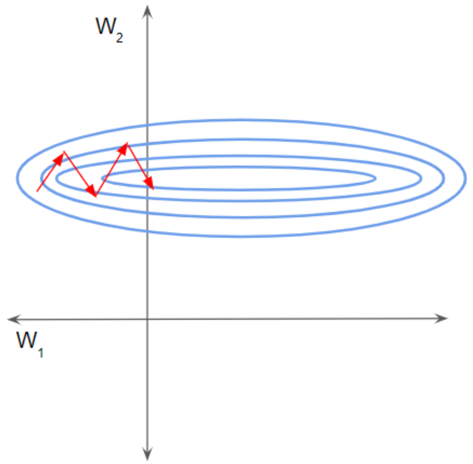
$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i, \qquad \sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

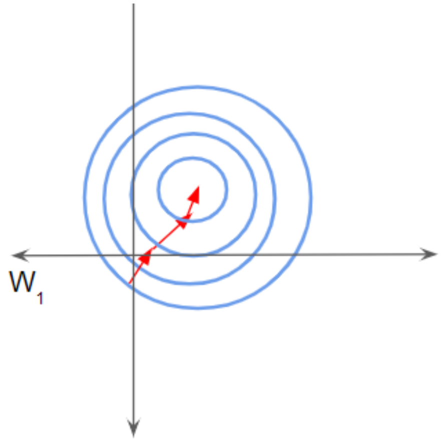
What normalized data looks like



Different scales take longer to reach the minimum



Normalized data helps the network converge faster

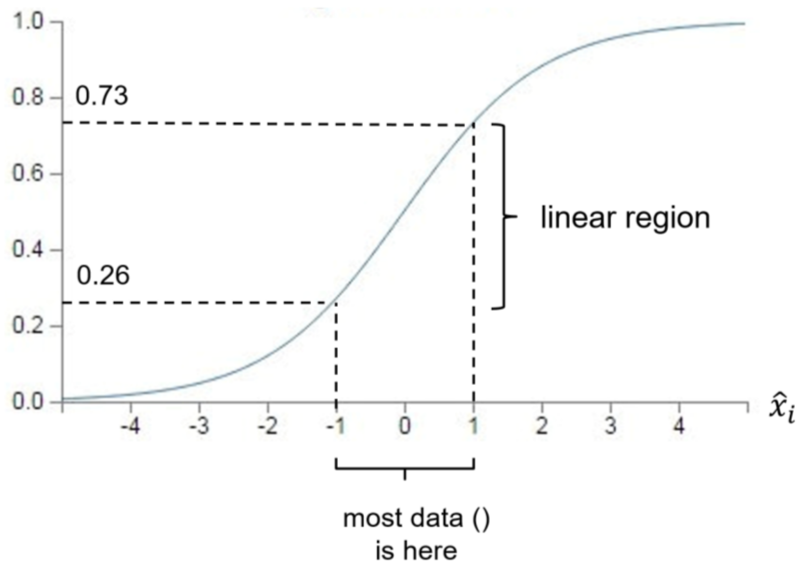


# Batch Normalization

- Z normalizing or whitening method
  1) *bias* parameter is gone.

$$x' = weight * x + bias$$

  2) loss of non-linearity with sigmoid or tanh



- Batch Normalization adds another layer of computation

$$y_i = \gamma \hat{x}_i + \beta$$

- Now $y_i$ values are spread in different region (not only in linear region) depending on the values of $\beta$ and $\gamma$

- The optimal values of $\beta$ and $\gamma$ are automatically determined by training.