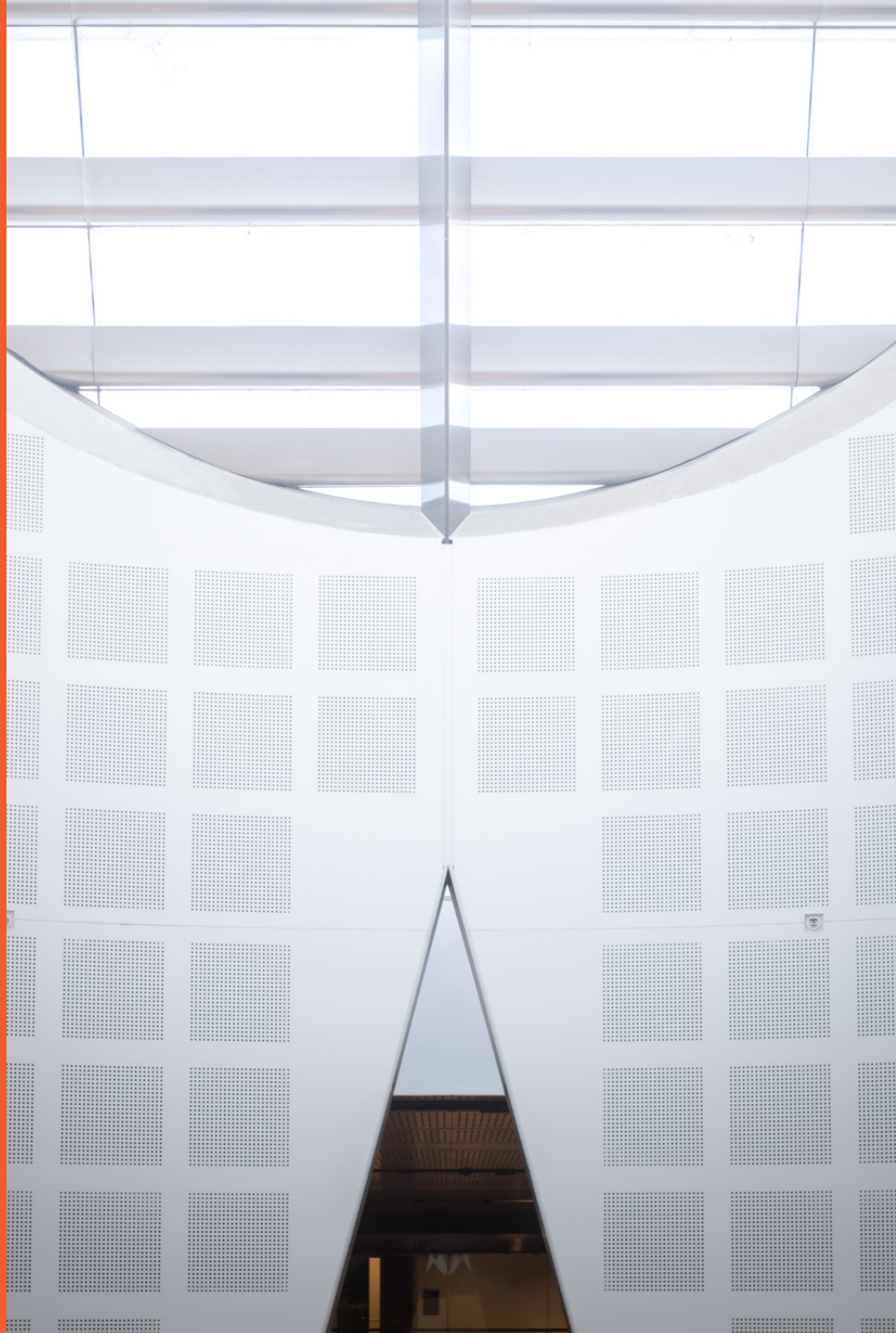
# Optimization for Training Deep Models

Dr Chang Xu
UBTECH Sydney AI Centre

Acknowledgements:
Erkun Yang and Zeyu Feng

THE UNIVERSITY OF SYDNEY



# Quick Review

# Outline

1. Gradient Descent Optimization
   - Gradient descent
   - Momentum
   - Nesterov
   - Adagrad
   - Adadelta
   - Rmsprop
   - Adam
   - Adamax
2. Activation Function
3. Initialization
4. Dropout
5. Batch Normalization
6. Weight normalization

# Outline

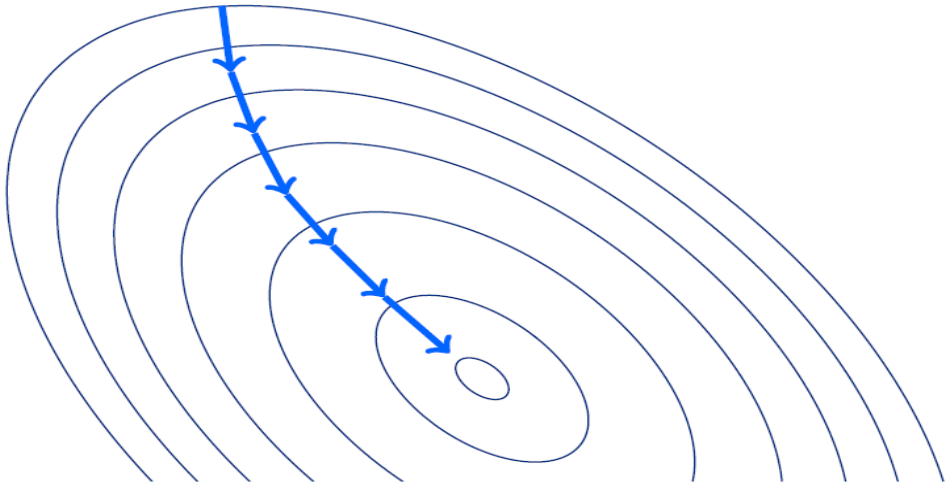
1. Gradient Descent Optimization
   - Gradient descent
   - Momentum
   - Nesterov
   - Adagrad
   - Adadelta
   - Rmsprop
   - Adam
2. Activation Function
3. Initialization
4. Dropout
5. Batch Normalization
6. Weight Normalization

# Gradient descent

- $J(\theta, \mathcal{X})$is the objective function,$\theta$ are the parameters, $\mathcal{X}$is the training dataset. We want to optimize $J(\theta, \mathcal{X})$by:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta, \mathcal{X})$$

Where $\nabla_\theta J(\theta, \mathcal{X})$is the gradient of objective function w.r.t. the parameters. $\eta$is the learning rate.



# Gradient descent

- **Batch gradient descent:**

  Compute the gradient for the **entire** training dataset.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta, \mathcal{X}^{(1:end)})$$

- **Stochastic gradient descent:**

  Compute the gradient for **each** training example.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta, \mathcal{X}^{(i)})$$

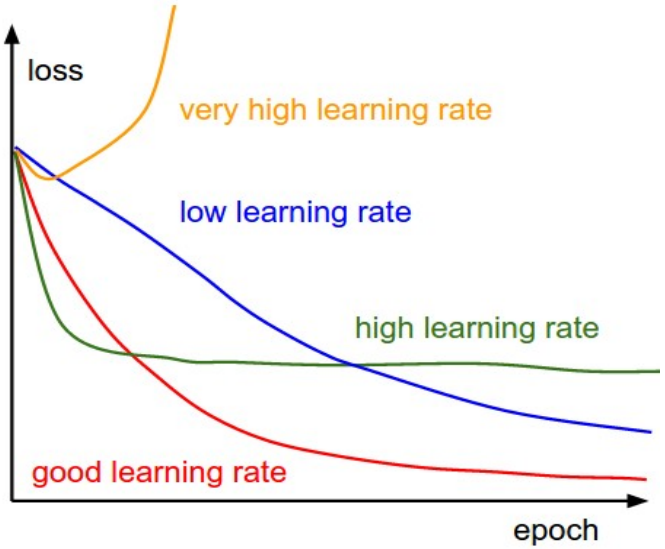
- **Mini-batch gradient descent:**

  Compute the gradient for every **mini-batch** training examples.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta, \mathcal{X}^{(i:i+n)})$$

# Some Challenges For Gradient descent

- **Choosing a proper learning rate can be difficult.**
  - A learning rate that is too small leads to painfully slow convergence.
  - A learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.




http://srdas.github.io/DLBook/GradientDescentTechniques.html

# Some Challenges For Gradient descent

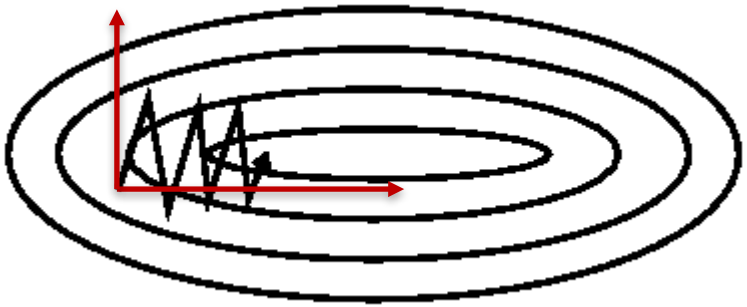
- **The same learning rate applies to all parameter updates.**
  - If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.

# Some Challenges For Gradient descent

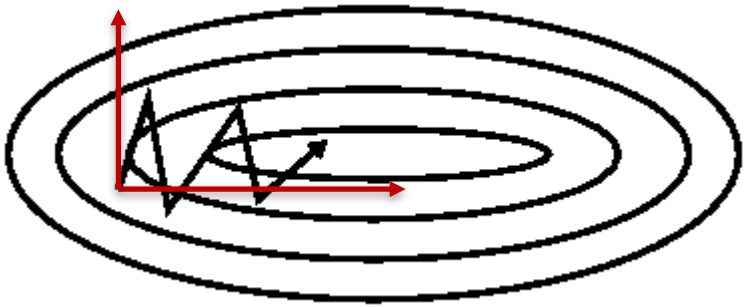
- **Easily get trapped in numerous saddle points.**
  - Saddle points are points where one dimension slopes up and another slopes down. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

# Momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima.



Without Momentum



With Momentum

https://www.willamette.edu/~gorr/classes/cs449/momrate.html

Momentum tries to accelerate SGD in the relevant direction and dampens oscillations.

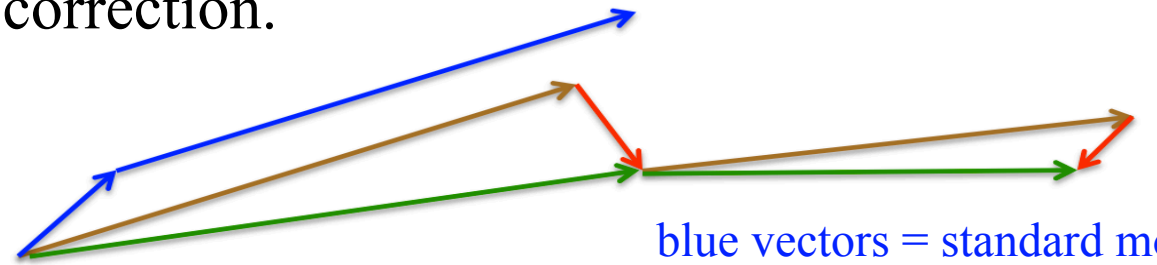
# Momentum

The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta_t = \theta_{t-1} - v_t$$

# Nesterov accelerated gradient (NAG)

- The **standard momentum** method:
  - First computes the gradient at the current location
  - Then takes a big jump in the direction of the updated accumulated gradient.
- **Nesterov accelerated gradient**（often works better）.
  - First make a big jump in the direction of the previous accumulated gradient.
  - Then measure the gradient where you end up and make a correction.



https://isaacchanghau.github.io/2017/05/29/parameter-update-methods/

blue vectors = standard momentum
brown vector = jump, red vector = correction,
green vector = accumulated gradient ,

# Nesterov accelerated gradient (NAG)

First make a big jump in the direction of the previous accumulated gradient.
Then measure the gradient where you end up.
Finally accumulate the gradient and make a correction.

First make a big jump

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Then measure the gradient where you end up

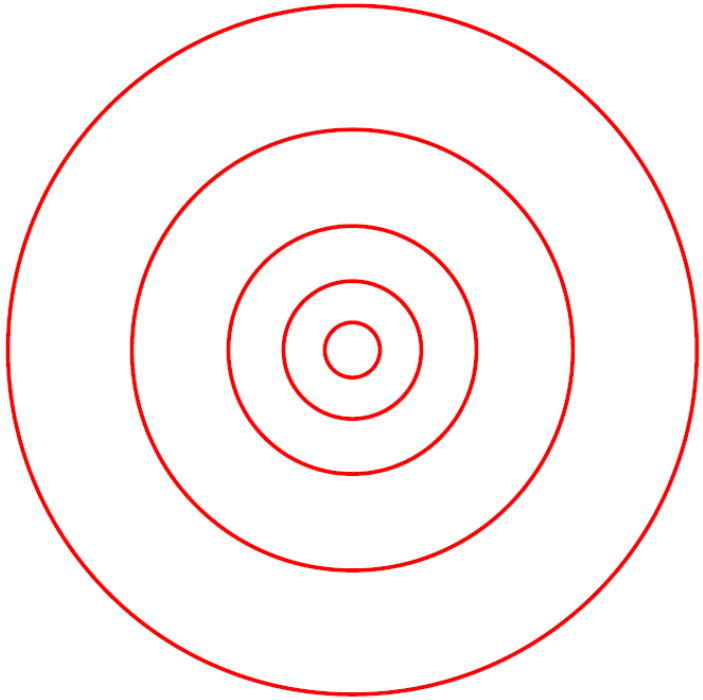
Finally accumulate the gradient and make a correction

# Adaptive Learning Rate Methods

# Motivation

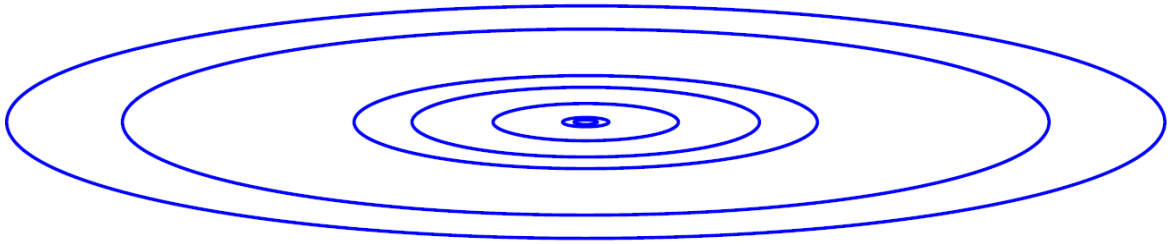
- Till now we assign the same learning rate to all features.
- If the feature vary in importance and frequency, why is this a good idea?
- It's probably not!

# Motivation



Nice (all features are equally important)

# Motivation



Harder

# • Adagrad

- **Original gradient descent:**

The same for all parameters

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot \nabla_\theta J(\theta_i)$$

Perform an update for all parameters using the same learning rate.

- **Adagrad:**

Adaptive learning for different parameters

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot \nabla_\theta J(\theta_i)$$

Adapt the learning rate to the parameters.

# Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$G_{t,ii} = \sum_{j=1}^{t} \sqrt{g_{t,i}^2}$$

$$g_{t,i} = \nabla_\theta J(\theta_i)$$

- **Adapt the learning rate to the parameters.**
  - Perform larger updates for infrequent updated parameters, and smaller updates for frequent updated parameters.
- **Well-suited for dealing with sparse data.**

# Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$G_{t,ii} = \sum_{j=1}^{t} \sqrt{g_{t,i}^2}$$

$$g_{t,i} = \nabla_\theta J(\theta_i)$$

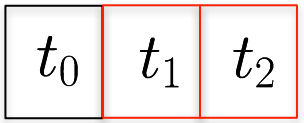
- **The learning rate eventually become infinitesimally small.**
  - $G_{t,ii} = \sum_{j=1}^{t} \sqrt{g_{t,i}^2}$ increase monotonically, $\frac{\eta}{\sqrt{G_{t,ii} + \epsilon}}$ will eventually be infinitesimally small.
- **Need to manually select a global learning rate** $\eta$ .

# Adadelta

- In Adagrad, the denominator accumulates the squared gradients from each iteration.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$
$$G_{t,ii} = \sum_{j=1}^{t} \sqrt{g_{t,i}^2}$$
$$g_{t,i} = \nabla_\theta J(\theta_i)$$

➢ **Accumulate over window.**




If we set window size to 2, we can get that

$$t = \frac{1}{2}(t_1 + t_2)$$

# Adadelta - Accumulate over window

- Storing fixed previous squared gradients cannot accumulate to infinity and instead becomes a local estimate using recent gradients.
- Adadelta implements it as an exponentially decaying average of the squared gradients.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

$$RMS[g]_t = -\sqrt{E[g^2]_t + \epsilon}$$

# Adadelta

- Need for a manually selected global learning rate.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$G_{t,ii} = \sum_{j=1}^{t} \sqrt{g_{t,i}^2}$$

$$g_{t,i} = \nabla_\theta J(\theta_i)$$

➢ **Correct units with Hessian approximation.**

# Adadelta - Correct units with Hessian approximation

- The units in SGD and Momentum relate to the gradient, not the parameter.

$$\theta = \theta - \eta \cdot g$$

$$units\ of\ \Delta\theta \propto units\ of\ g \propto \frac{\partial J}{\partial \theta} \propto \frac{1}{units\ of\ \theta}$$

- Second order methods such as Newton's method do have the correct units.

$$\theta = \theta - H^{-1} g$$

$$units\ of\ \Delta\theta \propto H^{-1} g \propto \frac{\frac{\partial J}{\partial \theta}}{\frac{\partial^2 J}{\partial \theta^2}} \propto units\ of\ \theta$$

# Adadelta - Correct units with Hessian approximation

Since:

$$\Delta\theta = \frac{\frac{\partial J}{\partial \theta}}{\frac{\partial^2 J}{\partial \theta^2}} \Rightarrow \frac{1}{\frac{\partial^2 J}{\partial \theta^2}} = \frac{\Delta\theta}{\frac{\partial J}{\partial \theta}}$$

We assume the curvature is locally smooth and approximate $\Delta\theta_t$ by computing the exponentially decaying RMS of $\Delta\theta$ .

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

# RMSprop

- RMSprop and Adadelta have both been developed independently around the same time.
- RMSprop in fact is identical to the first update vector of Adadelta.

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

# Adam

Combine the advantages of **Adgrad** and **RMSprop**.

- Adgrad: adaptive learning rate for different parameters.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} m_t$$

- RMSprop: exponentially decaying average.
  - Store an exponentially decaying average of past squared gradients $m_t$.
  - Keep an exponentially decaying average of past gradients $v_t$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

# Adam - bias-correct

Since $m_t$ and $v_t$ are initialized as vectors of 0's, they are biased towards zero, especially during the initial time steps or when the decay rates are small.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

# Adam

The overall update rules:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

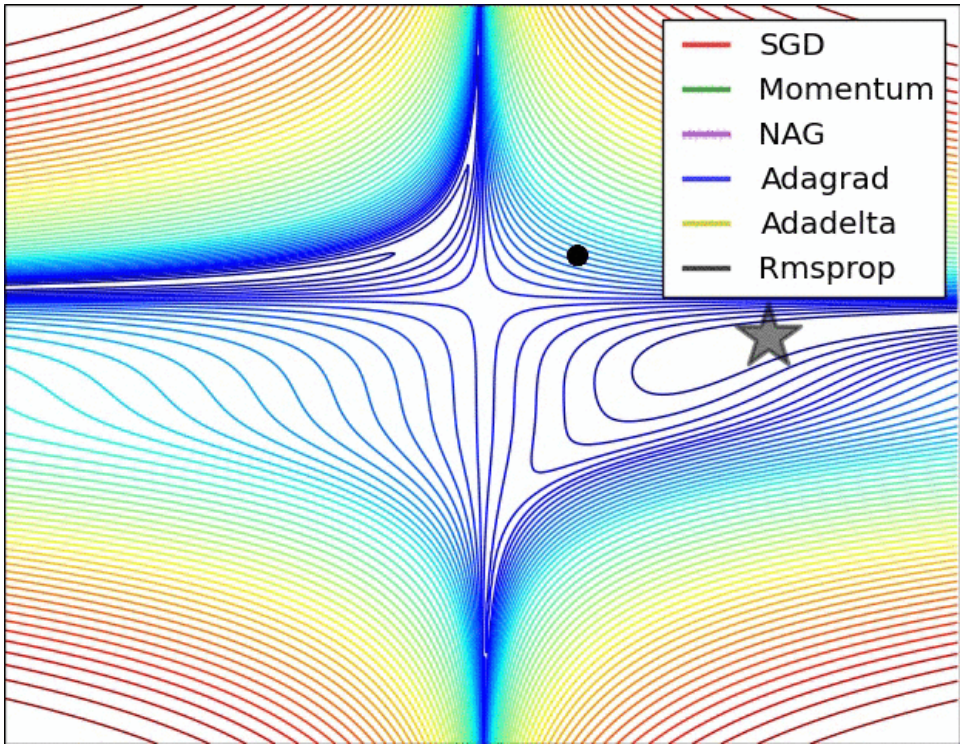
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \ \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

# Visualization

- Both run towards the negative gradient direction
- Momentum and NAG run faster in the in the relevant direction



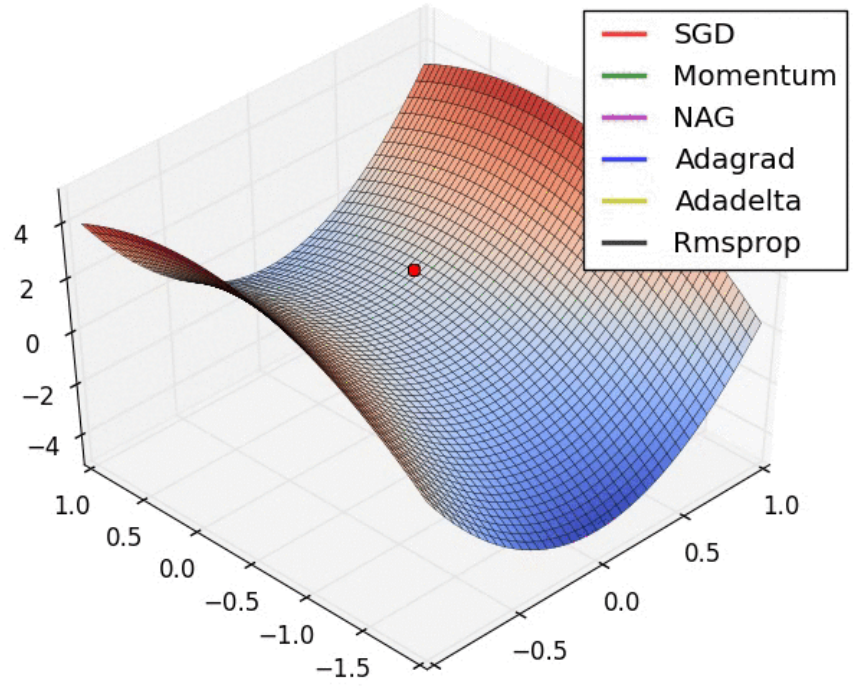

Contours of a loss face

http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

# Visualization

**Saddle point**: a point where one dimension has a positive slope, while the other dimensions has negative slopes.

- It's easier for **Rmsprop**, **Adadelta**, and **Adagrad** to escape the saddle point.




Behaviour around a saddle point

http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

# 1. Gradient Descent Optimization

| Methods | Adaptive lnearning rate | Consistent units | Easily hand Saddle points |
|---|---|---|---|
| SGD | No | No | No |
| Momentum | No | No | No |
| Nesterov | No | No | No |
| Adagrad | Yes | No | Yes |
| Adadelta | Yes | Yes | Yes |
| Rmsprop | Yes | No | Yes |
| Adam | Yes | No | Yes |

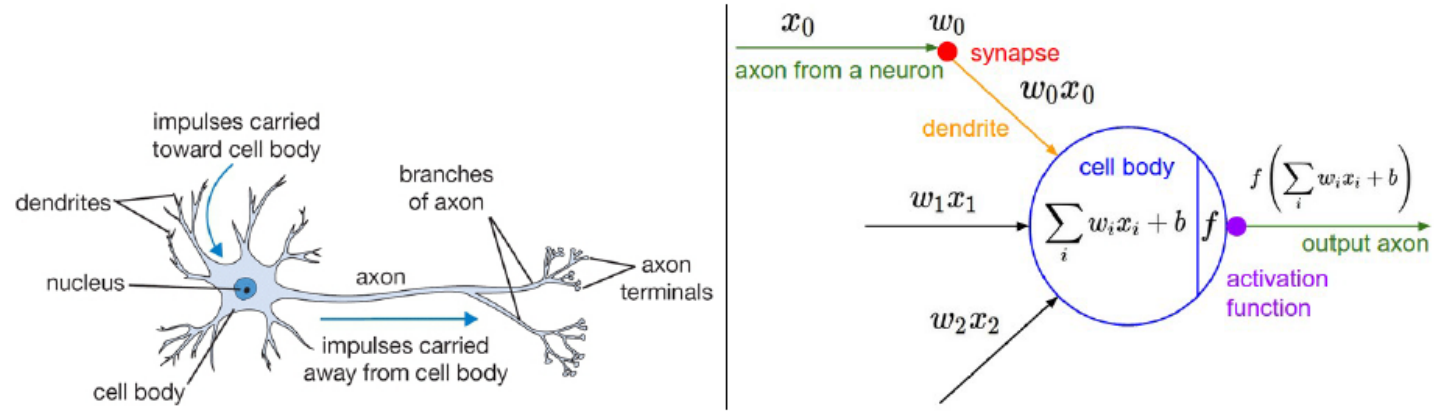
# Outline

1. Gradient Descent Optimization
   - Gradient descent
   - Momentum
   - Nesterov
   - Adagrad
   - Adadelta
   - Rmsprop
   - Adam
2. Activation Function
3. Initialization
4. Dropout
5. Batch Normalization
6. Weight Normalization

# Activation Function

## Why activation function?

- **From the perspective of biology:**

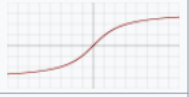
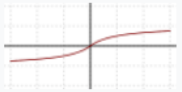
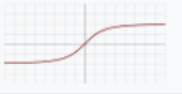
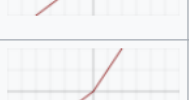



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

https://isaacchanghau.github.io/2017/05/22/Activation-Functions-in-Artificial-Neural-Networks/

- **From the perspective of mathematics:**
  Make neural networks non-linear.

# Activation Function

| Name | Plot | Equation | Derivative (with respect to $x$) |
|---|---|---|---|
| Identity |  | $f(x) = x$ | $f'(x) = 1$ |
| Binary step |  | $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$ |
| Logistic (a.k.a. Sigmoid or Soft step) |  | $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$ [1] | $f'(x) = f(x)(1 - f(x))$ |
| TanH |  | $f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$ | $f'(x) = 1 - f(x)^2$ |
| ArcTan |  | $f(x) = \tan^{-1}(x)$ | $f'(x) = \frac{1}{x^2+1}$ |
| Softsign [7][8] |  | $f(x) = \frac{x}{1+\lvert x\rvert}$ | $f'(x) = \frac{1}{(1+\lvert x\rvert)^2}$ |
| Inverse square root unit (ISRU)[9] |  | $f(x) = \frac{x}{\sqrt{1+\alpha x^2}}$ | $f'(x) = \left(\frac{1}{\sqrt{1+\alpha x^2}}\right)^3$ |
| Rectified linear unit (ReLU)[10] |  | $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$ |
| Leaky rectified linear unit (Leaky ReLU)[11] |  | $f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$ |
| Parameteric rectified linear unit (PReLU)[12] |  | $f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$ | $f'(\alpha, x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$ |

https://en.wikipedia.org/wiki/Activation_function

# Outline

1. Gradient Descent Optimization
   - Gradient descent
   - Momentum
   - Nesterov
   - Adagrad
   - Adadelta
   - Rmsprop
   - Adam
2. Activation Function
3. Initialization
4. Dropout
5. Batch Normalization
6. Weight Normalization

# Initialization

- All zero initialization
- Small random numbers
- Calibrating the variances

# Initialization

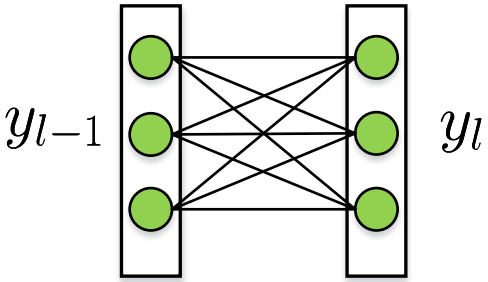
**All zero initialization:**

Every neuron computes the same output.

They will also compute the same gradients.

They will undergo the exact same parameter updates.

There will be no difference between all the neurons.




$$y_l = W y_{l-1} + b, \quad \frac{\partial \mathcal{L}}{\partial y_{l-1}} = W^\top \frac{\partial \mathcal{L}}{\partial y_l}$$
$$if \quad W = 0, \quad then \quad \frac{\partial \mathcal{L}}{\partial y_{l-1}} = 0$$
$$then, \quad \frac{\partial \mathcal{L}}{\partial y_m} = 0, \quad m = 0, ...l-1.$$

# Initialization

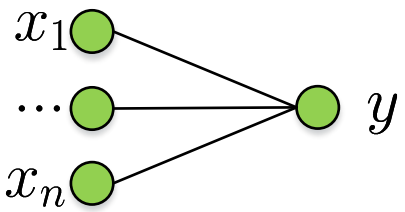
**Small random numbers:**

- With proper data normalization, we can assume that approximately half of the weights will be positive and half of them will be negative.
- We want the initial weights to be very close to zero.

$$W \sim 0.01 \cdot \mathcal{N}(0, 1)$$

# Initialization

Suppose a neuron with n inputs $y = \sum_i^n w_i x_i$, the variance is:




$$
\begin{aligned}
\mathrm{Var}(y) &= \mathrm{Var}(\sum_i^n w_i x_i) \\
&= \sum_i^n \mathrm{Var}(w_i x_i) \\
&= \sum_i^n [E(w_i)]^2 \mathrm{Var}(x_i) + E[(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= \sum_i^n \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= (n\mathrm{Var}(w))\,\mathrm{Var}(x)
\end{aligned}
$$

# Initialization

## Calibrating the variances

$$\mathrm{Var}(y) = (n\mathrm{Var}(w))\,\mathrm{Var}(x)$$

To keep information flow, we want the outputs to have the same variance as the inputs, we need $n\mathrm{Var}(w) = 1$

From a forward-propagation point of view:

$$n_{in}\mathrm{Var}(w) = 1$$

From a back-propagation point of view:

$$n_{out}\mathrm{Var}(w) = 1$$

Compromise between these two constraints:

$$\mathrm{Var}(w) = \frac{2}{n_{in}+n_{out}}$$

# Initialization (Xavier) -- [Xavier Glorot, Yoshua.Bengio 2010]

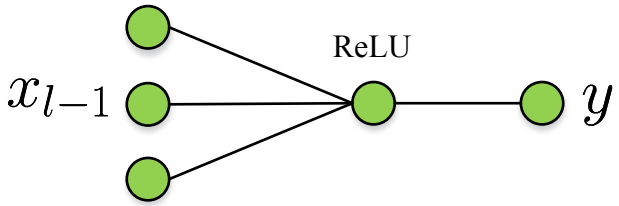
The variance of the inputs and outputs are the same for all layers. But the variance of the gradients might still vanish or explode as we consider deeper networks.

$$s^i = Z^i W^i + b^i$$
$$z^{i+1} = f(s^i)$$
$$\mathrm{Var}[\frac{\partial Cost}{\partial s^i}] = [n\mathrm{Var}[W]]^{d-i}\mathrm{Var}[x]$$
$$\mathrm{Var}[\frac{\partial Cost}{\partial w^i}] = [n\mathrm{Var}[W]]^{d}\mathrm{Var}[x]\mathrm{Var}[\frac{\partial Cost}{\partial s^i}]$$

$$W \sim U[-\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}]$$

# Initialization（He） -- [Kaiming He, Xiangyu Zhang, et al. 2015]

If we use ReLU as the activation function, n is the number of inputs, then we have:

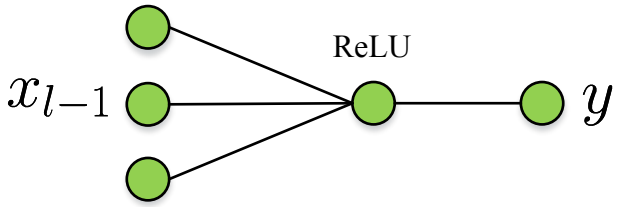



$$
\begin{aligned}
\mathrm{Var}(y) &= \mathrm{Var}(\sum_i^n w_i x_i) \\
&= \sum_i^n \mathrm{Var}(w_i x_i) \\
&= \sum_i^n [E(w_i)]^2 \mathrm{Var}(x_i) + [E(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= \sum_i^n [E(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= \sum_i^n [E(x_i)]^2 \mathrm{Var}(w_i) + (E[x_i^2] - [E(x_i)]^2)\mathrm{Var}(w_i) \\
&= \sum_i^n E[x_i^2]\mathrm{Var}(w_i) = (n\mathrm{Var}(w))E[x_i^2]
\end{aligned}
$$

# Initialization（He） -- [Kaiming He, Xiangyu Zhang, et al. 2015]

If we use ReLU as the activation function, n is the number of inputs, then for layer $l$ we have:

$$\mathrm{Var}(y_l) = (n\mathrm{Var}(w))\, E[x_{l-1}^2]$$
$$x_{l-1} = max(0, y_{l-1})$$




If $y_{l-1}$ has zero mean, and has a symmetric distribution around zero:

$$E[x_{l-1}^2] = E[(max(0, y_{l-1}))^2] = \frac{1}{2}\mathrm{Var}[y_{l-1}]$$

So we have:

$$\boxed{\mathrm{Var}(w) = \frac{2}{n}}$$

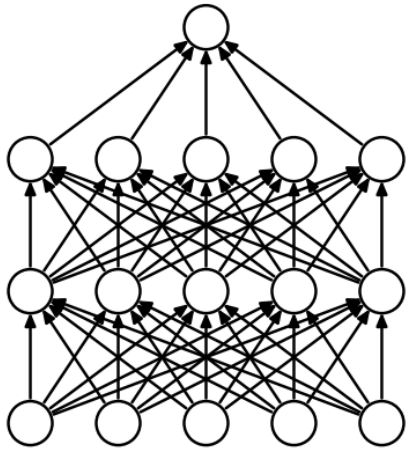
# Outline

1. Gradient Descent Optimization
   - Gradient descent
   - Momentum
   - Nesterov
   - Adagrad
   - Adadelta
   - Rmsprop
   - Adam
2. Activation Function
3. Initialization
4. Dropout
5. Batch Normalization
6. Weight Normalization

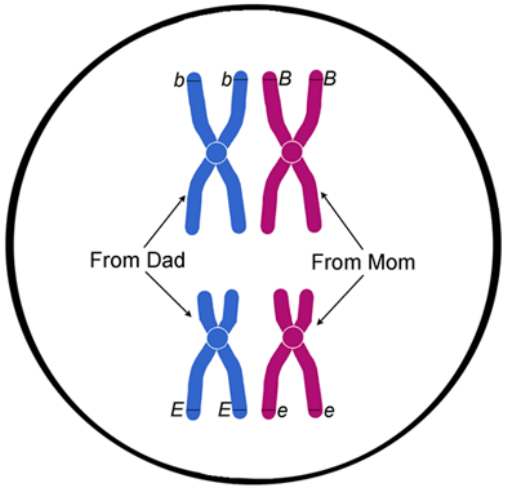
# Dropout

It is a simple but very effective technique that could alleviate overfitting in training phase.

The inspiration for Dropout (Hinton et al., 2013), came from the role of sex in evolution.



- Genes work well with another small random set of genes.
- Similarly, dropout suggests that each unit should work with a random sample of other units.

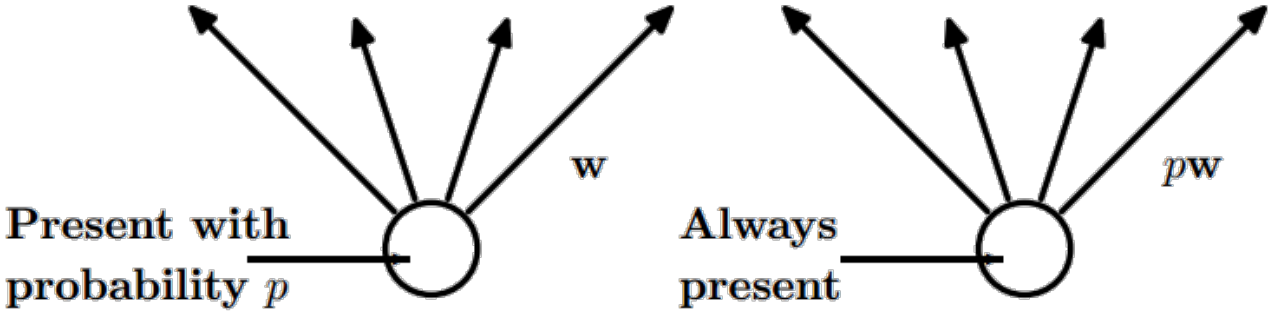



pinterest.com

# Dropout

- At training (each iteration):

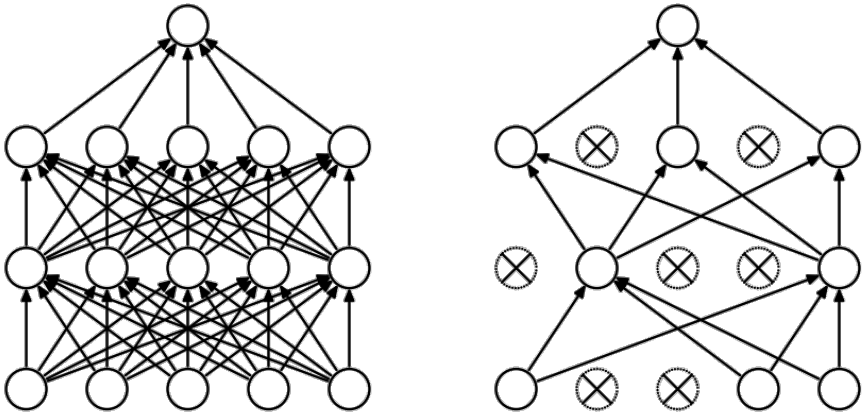
  Each unit is retained with a probability $p$.




- At test:

  The network is used as a whole.

  The weights are scaled-down by a factor of $p$ (e.g. 0.5).

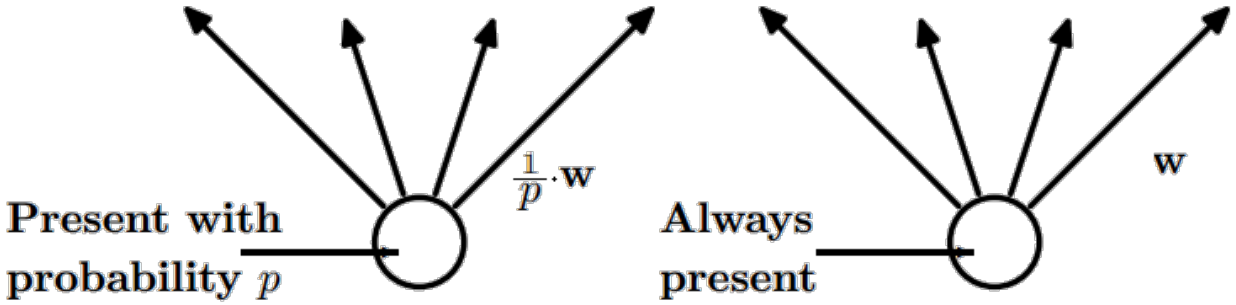


# Dropout

- At training (each iteration):

  Each unit is retained with a probability $p$.

  ✓ Weights are scaled-up by a factor of $1/p$.



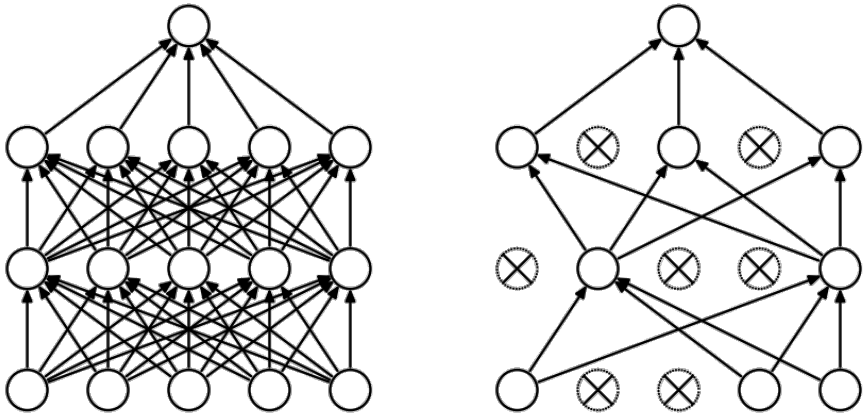

- At test:

  The network is used as a whole.

  ✓ No scaling is applied.

# Dropout

In practice, dropout trains $2^n$ networks ($n$ – number of units).



Drop is a technique which deal with overfitting by combining the predictions of many different large neural nets at test time.

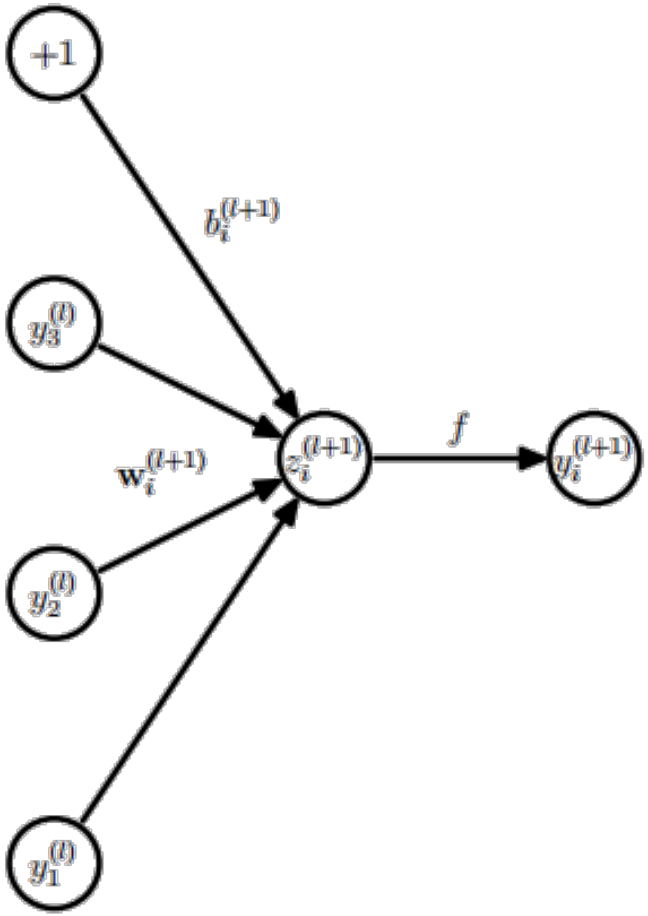
$$y^{l+1} = \sum_M p(M) f\left((M * W) y^l\right) \approx f\left(\sum_M p(M)(M * W) y^l\right) = f\left(pW y^l\right)$$

# Dropout

The feed-forward operation of a standard neural network is

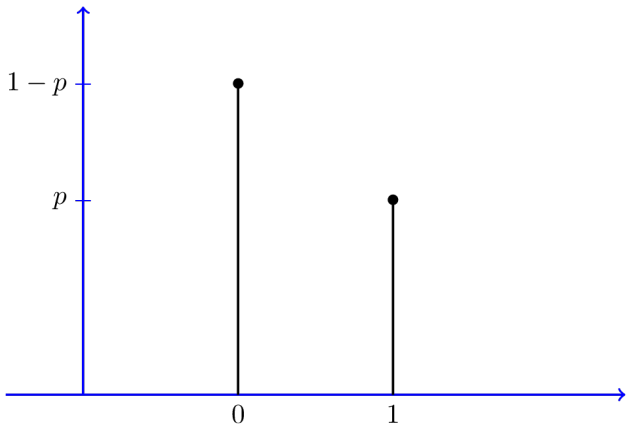
$$z_i^{(l+1)} = \boldsymbol{w}_i^{(l+1)} \boldsymbol{y}^{(l)} + b_i^{(l+1)}$$

$$y_i^{(l+1)} = f\left(z_i^{(l+1)}\right)$$



# Dropout

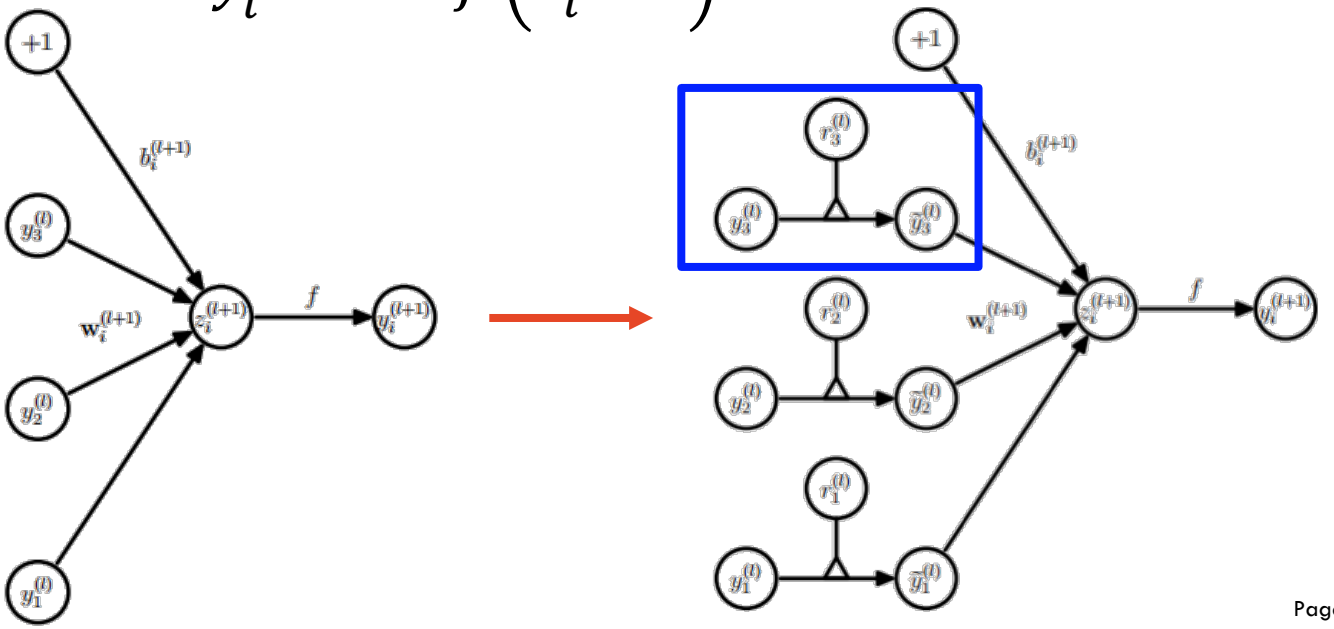
With dropout: $r_i^{(l)} \sim \text{Bernoulli}(p)$




$$\tilde{\boldsymbol{y}}^{(l)} = \boldsymbol{r}^{(l)} * \boldsymbol{y}^{(l)}$$

$$z_i^{(l+1)} = \boldsymbol{w}_i^{(l+1)} \tilde{\boldsymbol{y}}^{(l)} + b_i^{(l+1)}$$

$$y_i^{(l+1)} = f\left(z_i^{(l+1)}\right)$$



# Weight Decay

- Limiting the growth of the weights in the network.
- A term is added to the original loss function, penalizing large weights:

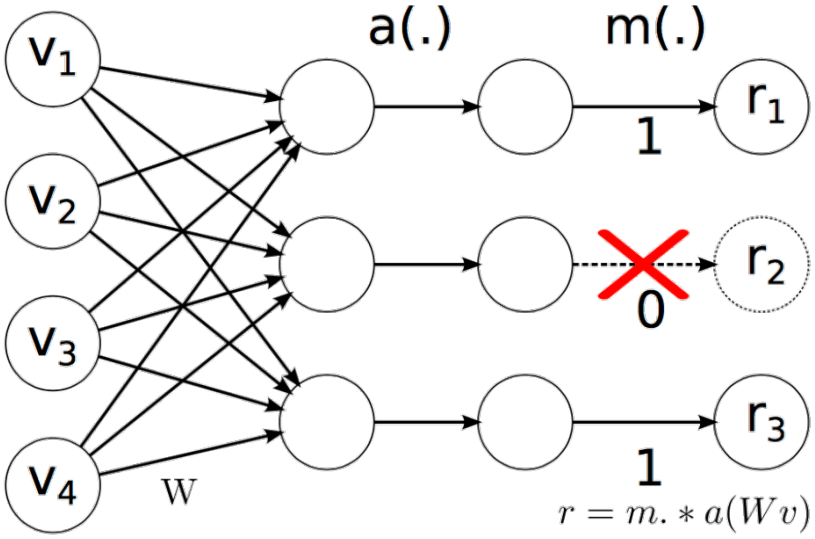
$$\mathcal{L}_{new}(\boldsymbol{w}) = \mathcal{L}_{old}(\boldsymbol{w}) + \frac{1}{2}\lambda\|\boldsymbol{w}\|_2^2$$

Dropout has more advantages over weight decay (Helmblod et al., 2016):

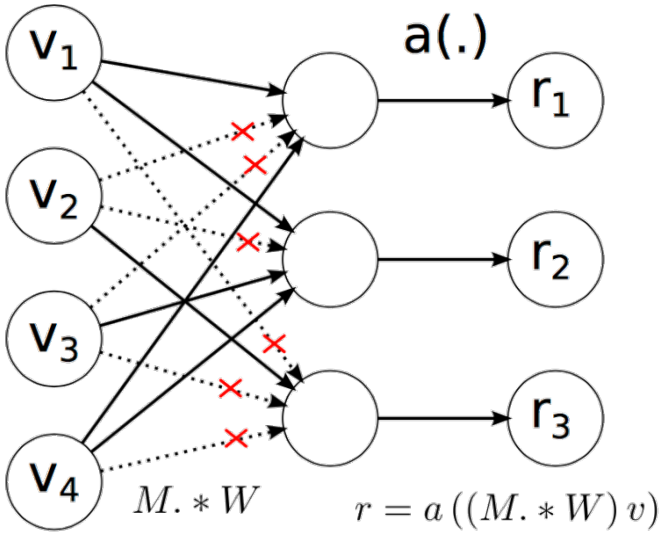
- Dropout is scale-free: dropout does not penalize the use of large weights when needed.
- Dropout is invariant to parameter scaling: dropout is unaffected if weights in a certain layer are scaled up by a constant $c$, and the weights in another layer are scaled down by a constant $c$.

# DropConnect

- DropConnect (Yann LeCun et al., 2013) generalizes dropout.
- Randomly drop connections in network with probability 1- $p$.
- As in Dropout, $p = 0.5$ usually gives the best results.




DropOut Network




DropConnect Network

# DropConnect

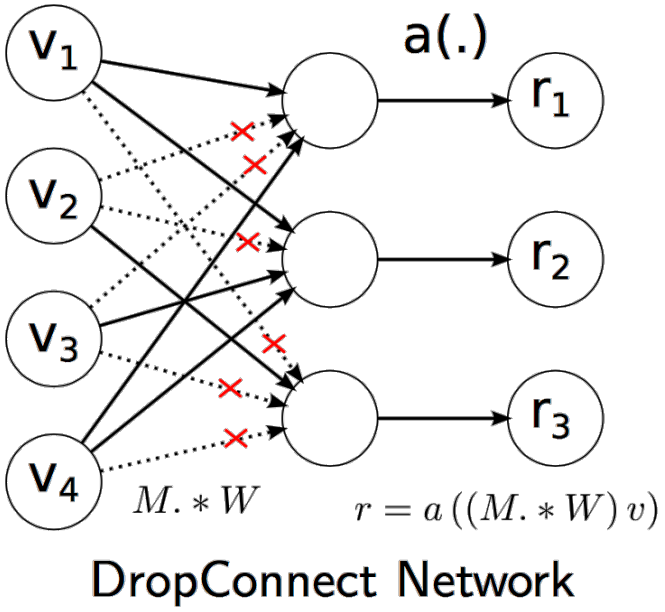
A single neuron before activation function:

$$z = (M.* W)v$$

Approximate $z$ by a Gaussian distribution:

$$E_M[z] = pWv$$

$$V_M[z] = p(1-p)(W.* W)(v.* v)$$




DropConnect Network

At test:

Draw samples of z according to the Gaussian distribution.

Feed the samples into the activation function ($f(z)$).

Average.

# Outline

1. Gradient Descent Optimization
   - Gradient descent
   - Momentum
   - Nesterov
   - Adagrad
   - Adadelta
   - Rmsprop
   - Adam
   - Adamax
2. Activation Function
3. Initialization
4. Dropout
5. Batch Normalization
6. Weight Normalization

# Feature scaling

- In machine learning algorithms, the functions involved in the optimization process are sensitive to normalization
  - For example: Distance between two points by the Euclidean distance. If one of the features has a broad range of values, the distance will be governed by this particular feature.
  - After, normalization, each feature contributes approximately proportionately to the final distance.

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

- Good practice for numerical stability for numerical calculations, and to avoid ill-conditioning when solving systems of equations.

# Classical normalizations

Two methods are usually used for rescaling or normalizing data:

- Scaling data all numeric variables to the range [0,1]. One possible formula is given below:

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

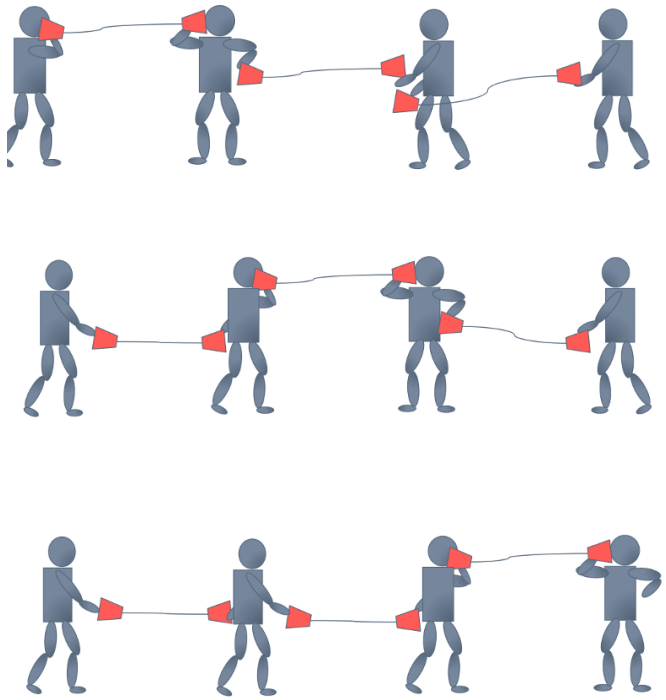
- To have zero mean and unit variance:

$$x_{new} = \frac{x - \mu}{\sigma}$$

- In the NN community this is call Whitening

# Internal covariate shift:

- The cup game example
- The problems are entirely systemic and due entirely to faulty red cups.
- The situation is analogous to forward propagation
- First layer parameters change and so the distribution of the input to your second layer changes.

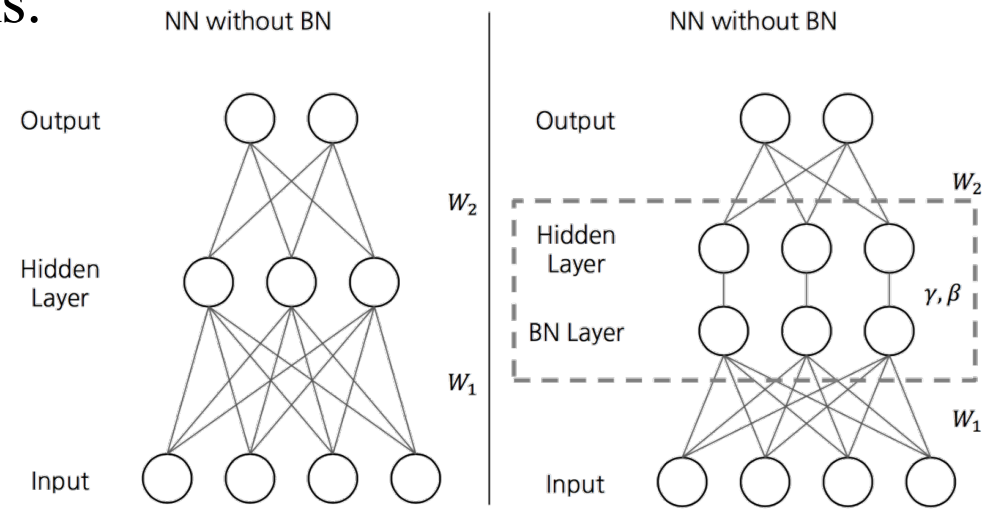


# Batch Normalization (BN)

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

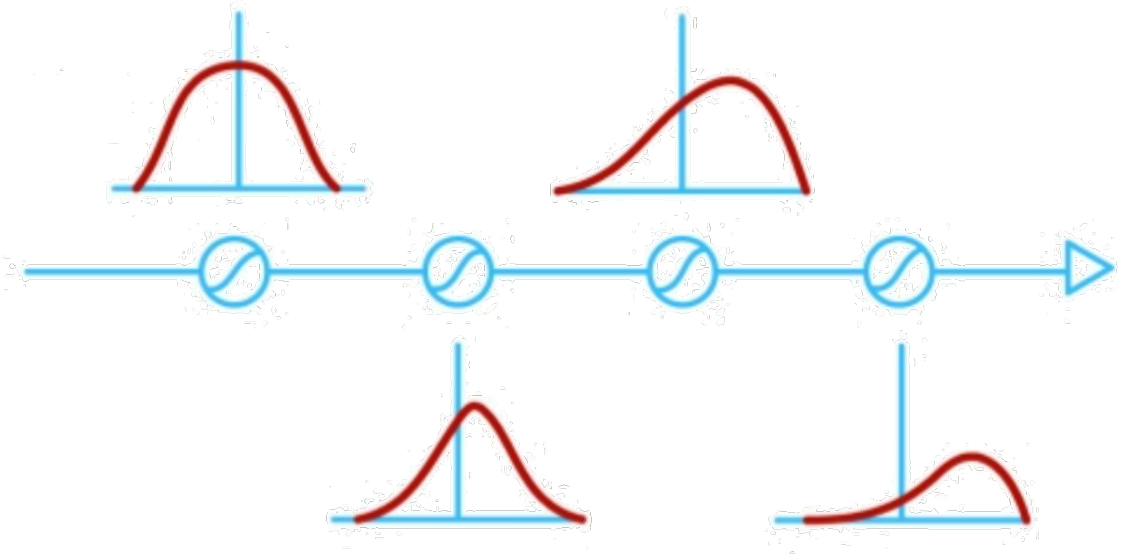
- Usually inputs to neural networks are normalized to either the range of [0, 1] or [-1, 1] or to mean=0 and variance=1

- BN essentially performs Whitening to the intermediate layers of the networks.



# Batch normalization
# Why it is good?

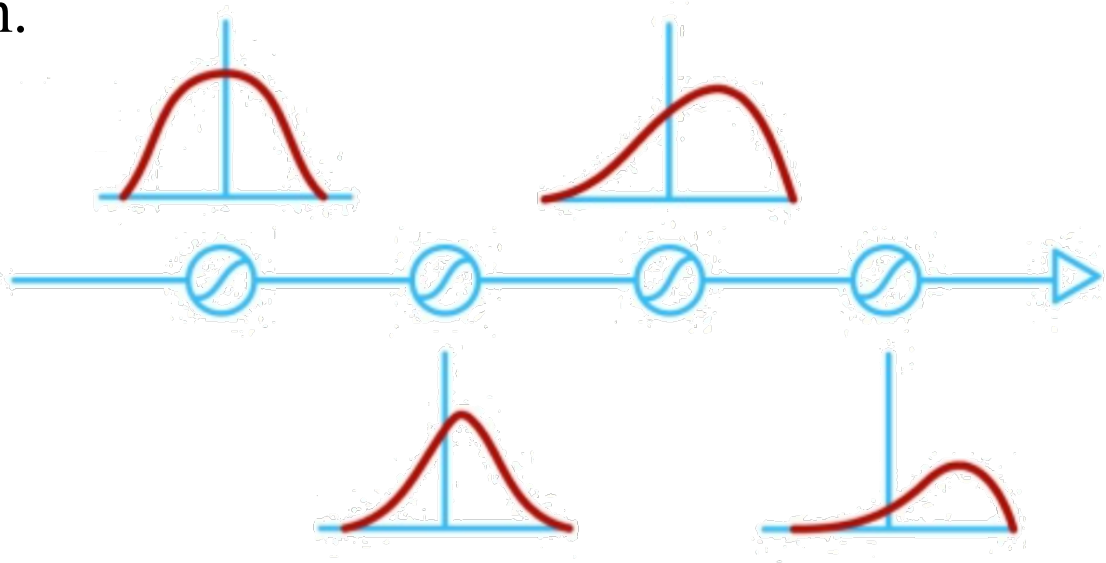
- BN reduces *Covariate Shift*. That is the change in distribution of activation of a component. By using BN, each neuron's activation (sigmoid) becomes (more or less) a Gaussian distribution, i.e. its usually not active, sometimes a bit active, rare very active.



# Batch normalization
# Why it is good?

- Covariate Shift is undesirable, because the later layers have to keep adapting to the change of the type of distribution.

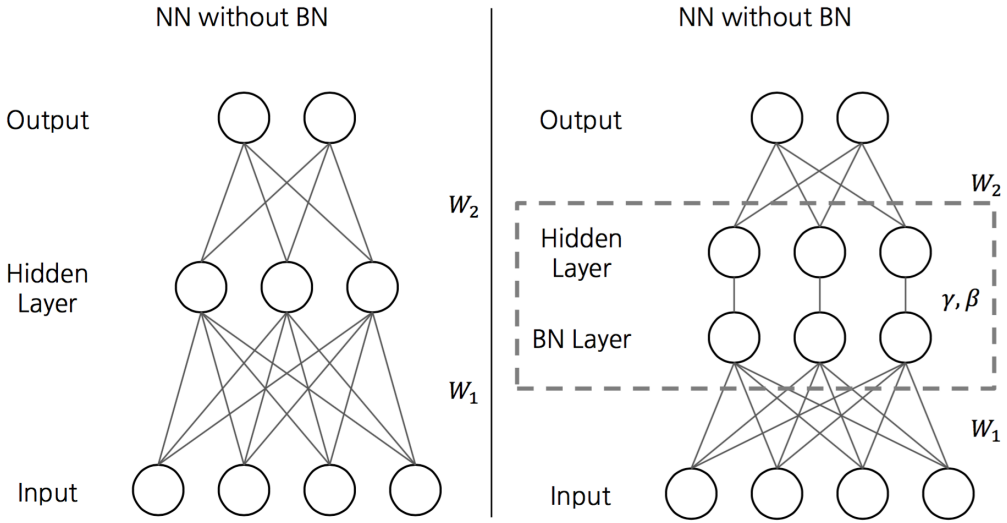
- BN reduces effects of exploding and vanishing gradients, because every becomes roughly normal distributed. Without BN, low activations of one layer can lead to lower activations in the next layer, and then even lower ones in the next layer and so on.



# Batch normalization layer

We introduce, for each activation $x^{(k)}$, a pair of parameters $\gamma^{(k)}$, $\beta^{(k)}$, which scale and shift the normalized value:

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{k}$$




A new layer is added so the gradient can "see" the normalization and make adjustments if needed.

# The Batch Transformation: formally from the paper.

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma$, $\beta$

**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

$\varepsilon$ is a constant added to the mini-batch variance for numerical stability.

$\gamma$ and $\beta$ are learned to keep what the layer can represent.

# The full algorithm as proposed in the paper

**Input:** Network $N$ with trainable parameters $\Theta$;
subset of activations $\{x^{(k)}\}_{k=1}^{K}$

**Output:** Batch-normalized network for inference, $N_{\text{BN}}^{\text{inf}}$

1: $N_{\text{BN}}^{\text{tr}} \leftarrow N$ // Training BN network
2: **for** $k = 1 \ldots K$ **do**
3: Add transformation $y^{(k)} = \text{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to $N_{\text{BN}}^{\text{tr}}$ (Alg. 1)
4: Modify each layer in $N_{\text{BN}}^{\text{tr}}$ with input $x^{(k)}$ to take $y^{(k)}$ instead
5: **end for**
6: Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
7: $N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$ // Inference BN network with frozen // parameters
8: **for** $k = 1 \ldots K$ **do**
9: // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
10: Process multiple training mini-batches $\mathcal{B}$, each of size $m$, and average over them:

$$\text{E}[x] \leftarrow \text{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$
$$\text{Var}[x] \leftarrow \frac{m}{m-1}\text{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

11: In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma,\beta}(x)$ with

$$y = \frac{\gamma}{\sqrt{\text{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\text{E}[x]}{\sqrt{\text{Var}[x]+\epsilon}}\right)$$

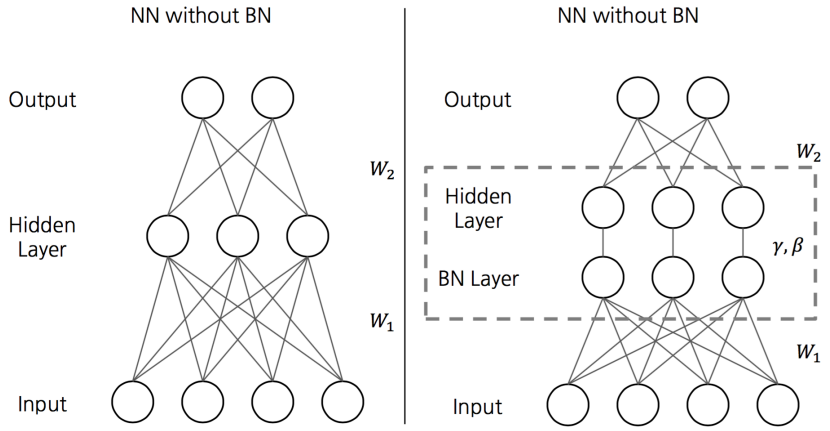
12: **end for**

**Algorithm 2:** Training a Batch-Normalized Network

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: $\gamma$, $\beta$

**Output:** $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$



# Batch normalization: Benefits in practice

- BN reduces training times. (less Covariate Shift, less exploding/vanishing gradients.)

- BN reduces demand for regularization, e.g. dropout or L2 norm.
  - Because the means and variances are calculated over batches and therefore every normalized value depends on the current batch. I.e. the network can no longer just memorize values and their correct answers.)

- BN allows higher learning rates. (Because of less danger of exploding/vanishing gradients.)

- BN enables training with saturating nonlinearities in deep networks, e.g. sigmoid. (Because the normalization prevents them from getting stuck in saturating ranges, e.g. very high/low values for sigmoid.)

# The BN transformation is scalar invariant

- Batch Normalization also makes training more resilient to the parameter scale.

$$
\begin{aligned}
\mu_{\mathcal{B}} &\leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i && \text{//mini-batch mean} \\
\sigma_{\mathcal{B}}^2 &\leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 && \text{//mini-batch variance} \\
\hat{x}_i &\leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} && \text{//normalize} \\
y_i &\leftarrow \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) && \text{//scale and shift}
\end{aligned}
$$

$$\mathbf{BN}(W\mathrm{u}) = \mathbf{BN}((aW)\mathrm{u})$$

# The BN transformation is scalar invariant

- Normally, large learning rates may increase the scale of layer parameters, which then amplify the gradient during backpropagation and lead to the model explosion

- However, with Batch Normalization, backpropagation through a layer is unaffected by the scale of its parameters.

$$\frac{\partial \mathrm{BN}((aW)\mathrm{u})}{\partial \mathrm{u}} = \frac{\partial \mathrm{BN}(W\mathrm{u})}{\partial \mathrm{u}}$$

$$\frac{\partial \mathrm{BN}((aW)\mathrm{u})}{\partial (aW)} = \frac{1}{a} \cdot \frac{\partial \mathrm{BN}(W\mathrm{u})}{\partial W}$$

Batch Normalization will stabilize the parameter growth.

# Outline

1. Gradient Descent Optimization
   - Gradient descent
   - Momentum
   - Nesterov
   - Adagrad
   - Adadelta
   - Rmsprop
   - Adam
   - Adamax
2. Activation Function
3. Initialization
4. Dropout
5. Batch Normalization
6. Weight Normalization

# Weight Normalization

- Weight Normalization is normalization technique, similar to batch normalization
- It normalizes each layer's weights

$$y = \phi(\mathbf{w} \cdot \mathbf{x} + b)$$

$$\mathbf{w} = \frac{g}{\|\mathbf{v}\|}\mathbf{v}$$

where $g$ is a scalar, and $\mathbf{v}$ is a vector having the same dimension with $\mathbf{w}$. $\|\mathbf{v}\|$ denotes the Euclidean norm of $\mathbf{v}$.

By decoupling the norm of the weight vector (g) from the direction of the weight vector (v/||v||), we speed up convergence of our stochastic gradient descent optimization.

# Weight Normalization

**Relationship with Batch normalization**

**Computational efficient:** Convolutional neural networks usually have much fewer weights than pre-activations, so normalizing the weights is often much cheaper computationally.

**Deterministic:** the norm of $\mathbf{v}$ is non-stochastic, while in batch normalization, the mean and variance can have high variance for small mini-batch.

# Summarization

1. Gradient Descent Optimization
   - Gradient descent
   - Momentum
   - Nesterov
   - Adagrad
   - Adadelta
   - Rmsprop
   - Adam
   - Adamax
2. Activation Function
3. Initialization
4. Dropout
5. Batch Normalization
6. Weight normalization

# Thank you!