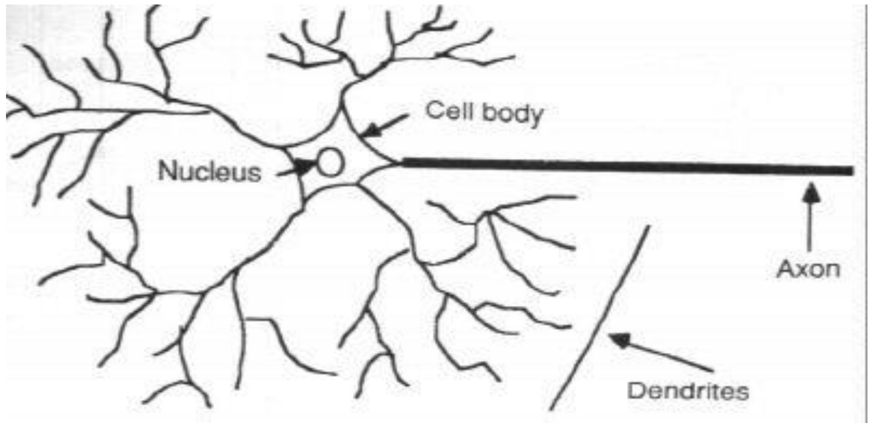
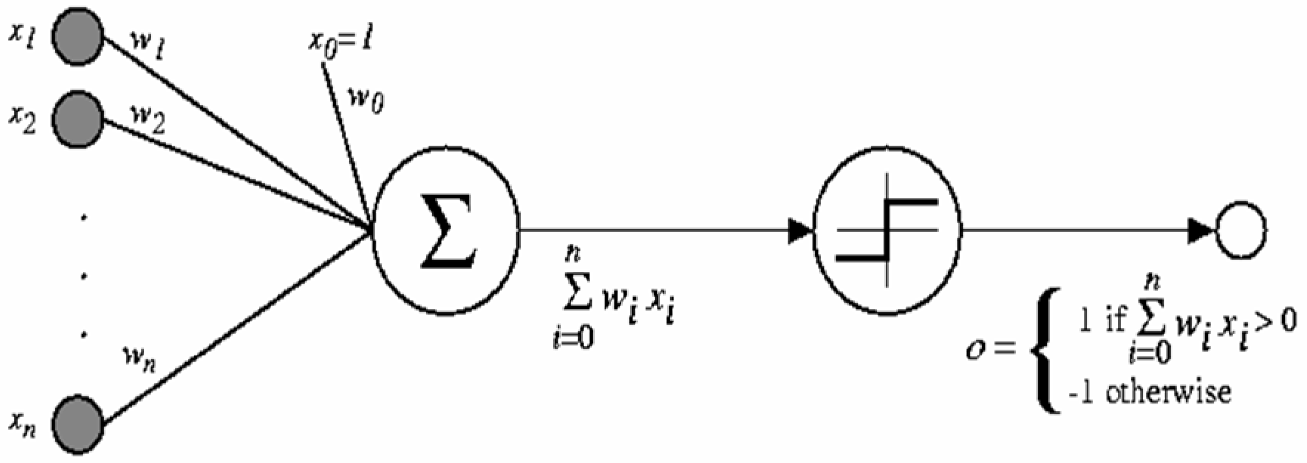
# Deep Neural Network & Gradient Methods

## Neural Network Representation

- An ANN is composed of processing elements, organized in different ways to form the network’s structure.
- Each element receives inputs, processes inputs and delivers a single output.
- The input can be raw input data or the output of other elements. The output can be the final result (e.g. 1 means yes, 0 means no) or inputs to other elements.




$x_1$
$w_1$
$x_2$
$w_2$
$x_0=1$
$w_0$
$w_n$
$x_n$
$\Sigma$
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

## 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

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

## Loss Functions

### 1. Mean Squared Error (MSE)

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

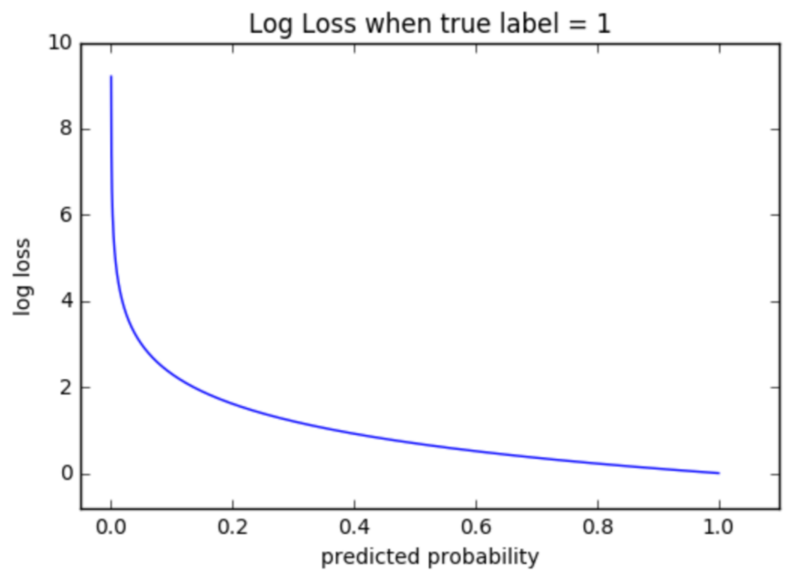
- Very popular. Sometime use log form
- Usually better than MAE
- Saturates when using with sigmoid activation function
- Both for classification and regression
- Without 1/n, it becomes L2 regularizer

## Loss Functions

### 2. Cross entropy(CE)

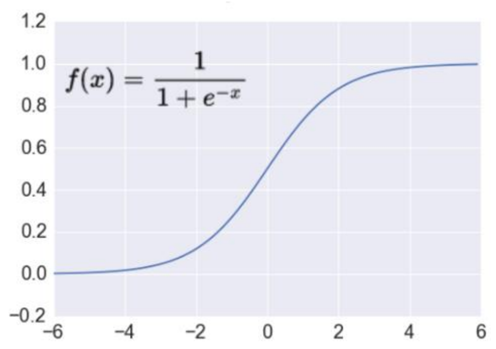
$$H(p,y) = \sum_i -(\boldsymbol{p_i} \log \boldsymbol{y_i})$$

- Binary CE & multi-class CE
- Average of information of p w.r.t. 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



# Activation Functions

## 1. Sigmoid



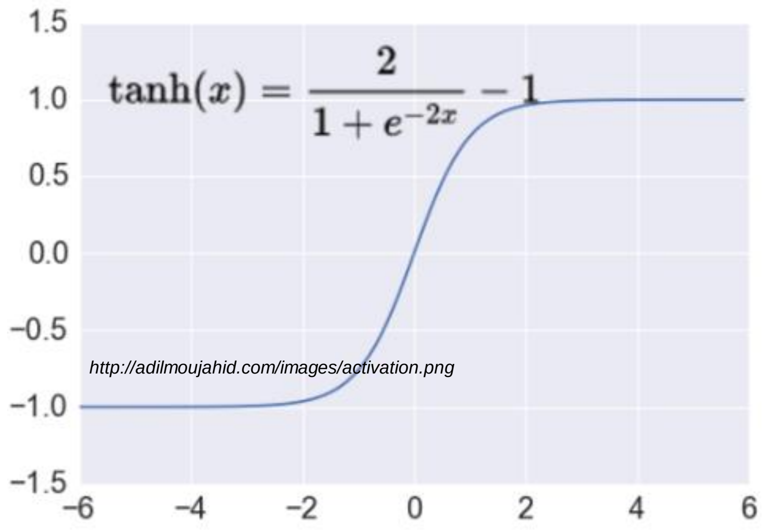
Takes a real-valued number and "squashes" it into range between 0 and 1.

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

- 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
- 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)
- Computationally expensive
- 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

## 2. Tanh



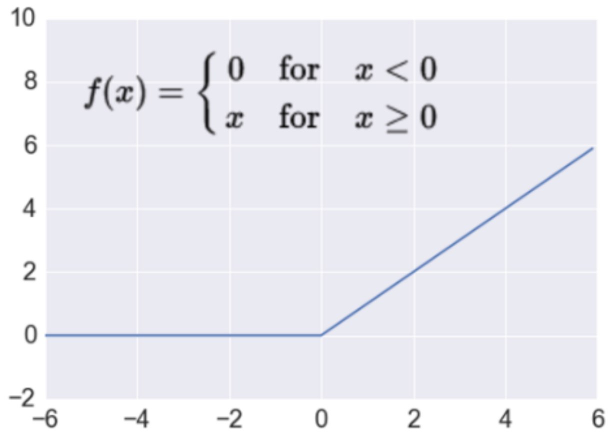
Takes a real-valued number and "squashes" it into range between -1 and 1.

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

- 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 Function

## 3. ReLU (Rectified Linear Unit)



Takes a real-valued number and thresholds it at zero

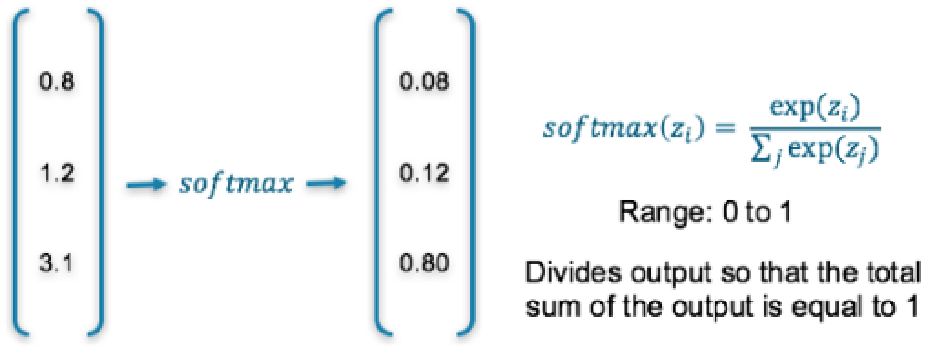
$$f(x) = \max(0, x)$$

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

- 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. (e.g., large negative weights, $\eta$ is high)
- Sometimes, ReLU blows up the activation

# Activation Functions

## 4. Softmax


0.8
1.2
3.1
softmax
0.08
0.12
0.80
$softmax(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$
Range: 0 to 1
Divides output so that the total sum of the output is equal to 1

- Special function on last layer
- Use in multi-class classification problem
- # of input = # of output
- Squashes a *C*-dimensional vector ***O*** 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

## 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.

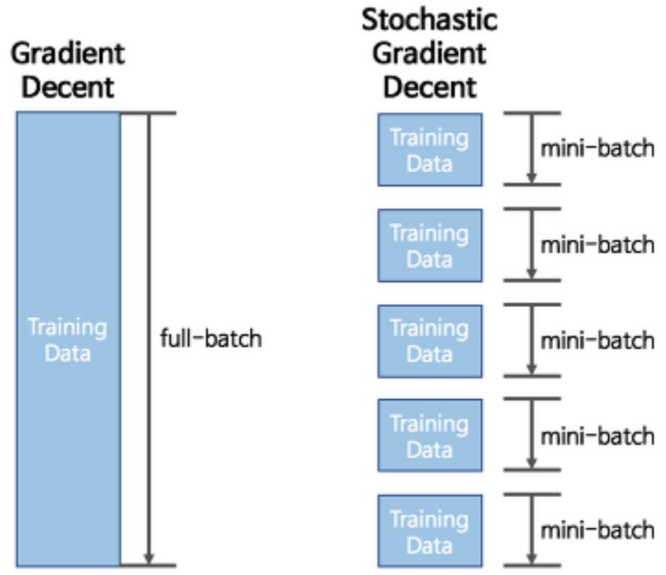
**2. Stochastic gradient descent (SGD)**

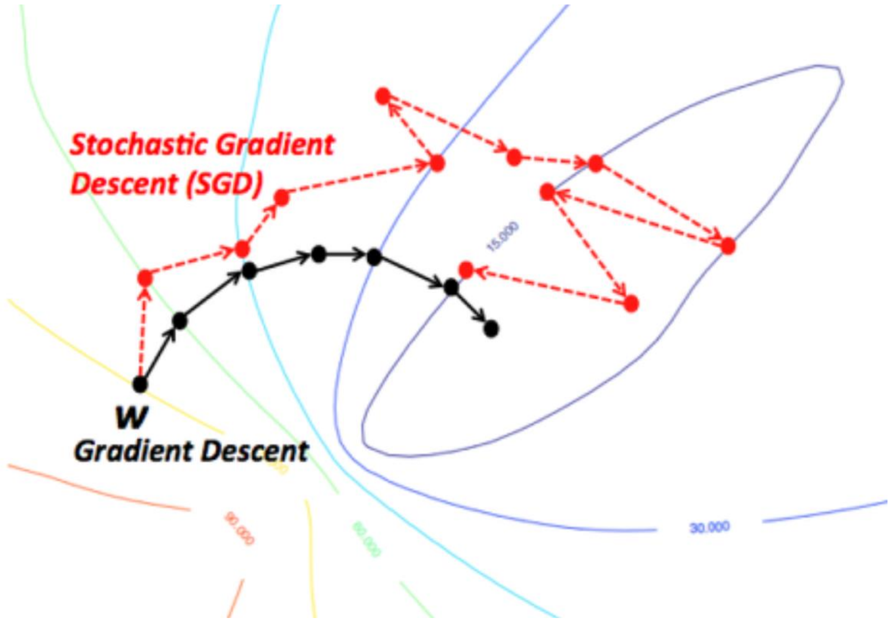
- Performs a parameter update for *each* training example
- Much faster and can also be used to learn online
- Performs frequent updates with a high variance that cause the objective function to fluctuate heavily

# 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.





## Methods in Learning Rate

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

### 1. 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. ($\epsilon$: 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

## 2. 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.
- 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

## 3. 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

## 4. 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 $Var(X) = E\left[X^2\right] - E[X]^2$) Second moment ($E\left[X^2\right]$) 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, and the parameters beta1 and beta2 control the decay rates of these moving averages.

- Use the following unbiased estimate

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

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

moment estimates are biased towards zero. This bias is overcome by first calculating the biased estimates before then calculating bias-corrected estimates.

- 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.

## Methods in Momentum

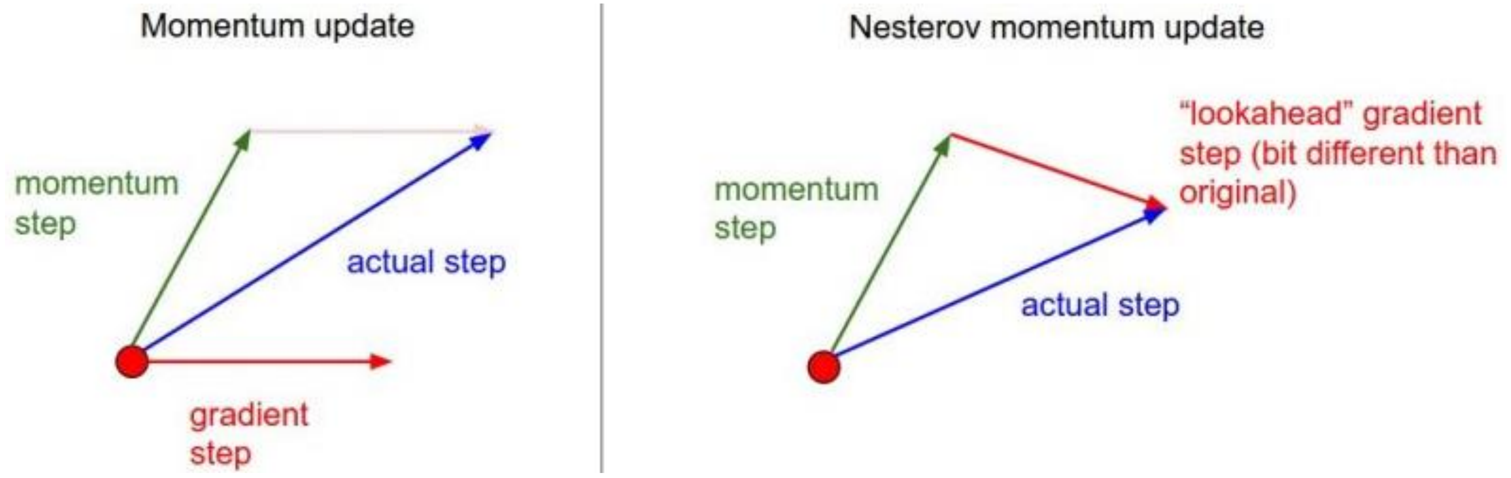
Traditional momentum

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

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

## Nesterov Accelerated Gradient (NAG)

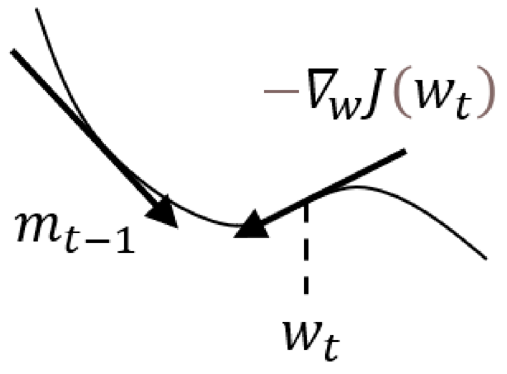
- Have a smarter ball that knows to slow down before the hill slopes up again
- NAG first makes a big jump in the direction of the previous gradient (green vector), measures the gradient (red vector), which results in the final NAG (blue vector).
- This anticipatory update prevents us from going too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



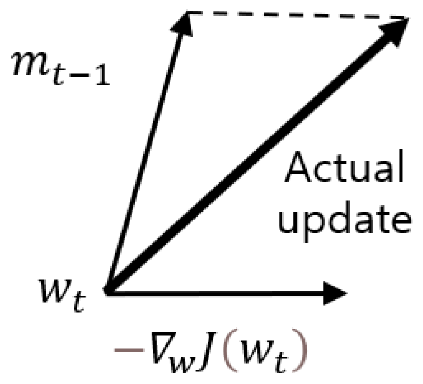
$$w_{t+1} = w_t + M_t$$

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

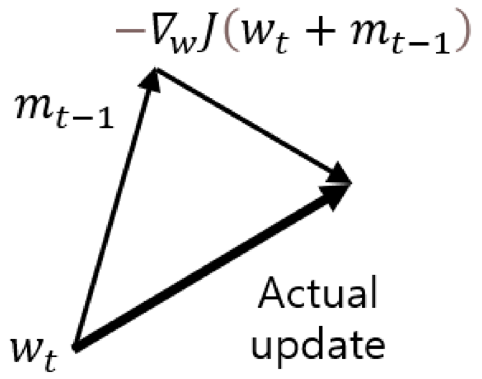
# Nesterov Accelerated Gradient (NAG)



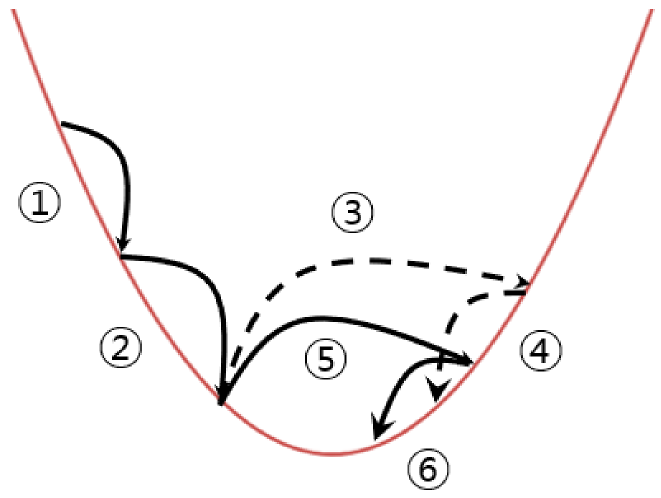
**Momentum**



**Momentum**



**NAG**



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

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

## Weight Initialization

- Proper initialization of parameters is important
- Never 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)

* 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.

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

- 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.

# Batch Normalization

- Whitening method

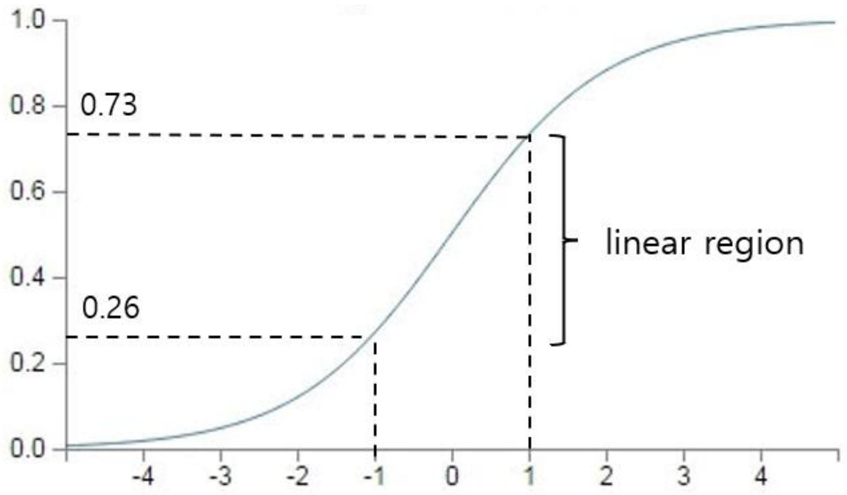
$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i, \qquad \sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

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

1) Using whitening method, *bias* parameter is gone.

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

2) loss of non-linearity $\longrightarrow$



- Batch Normalization adds another layer of computation

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

## Regularization

## Regularization

- The most widely used method against overfitting in machine learning
- Regularization: Machine learning technique that constrains our optimization problem to discourage complex models
- Occam's razor
- Improve generalization of our model on unseen data
- Reduce the complexity of model
  - VC dimension (why not use it?), Number of parameters,
- Very important topic in machine learning

# Regularization

- Idea: A new regularization term (regularizer) is added to loss (objective) function

- Loss function with regularization term is modified as follows
  - $J(w)$ : loss function, λ : regularization constant
  - $R(w)$ : regularization term (aka regularizer). Can be regarded as the complexity of model

$$J_{reg}(w) = J(w) + \lambda * R(w)$$

- Gradient descent method minimizes $J_{reg}(w)$ instead of $J(w)$

- As the value of λ rises, it reduces the value of w's and thus reducing the variance

- Goal of regularization term ($R(w)$) is to make model parameters simpler, specifically
  - reduces the number of parameters (L1/Lasso regularization) or
  - makes the values of parameters smaller (L2/Ridge regularization)

## Regularizer

- A regularizer is an additional criteria to the loss function to make sure that we don’t overfit
- It’s called a regularizer since it tries to keep the parameters more normal/regular.
  - significantly reduces the variance of the model, without substantial increase in its bias
- Generally, we don’t want huge weights
- If weights are large, a small change in a feature can result in a large change in the prediction. Also gives too much weight to any one feature
- Might also prefer weights of 0 for features that aren’t useful
- How do we encourage small weights? or penalize large weights?

# Common Regularizers

- From $J_{reg}(w) = J(w) + \lambda * R(w)$

## L1 (Lasso) Regularization

- Regularizer, $R(w)$, is given as the sum of weights

$$\boxed{R(w) = \sum_i |w_i|}$$

$$J_{reg}(w) = J(w) + \lambda \sum_i |w_i|$$

- Forces less important parameters be zero
- Reduces the number of features automatically (feature selection)
- Regularization constant (λ, weight decay value) determines how dominant regularization is during gradient computation
- Regularizer is not differentiable
- Generate multiple solutions
- Also known as Lasso regularization

# Common Regularizers

## L2 (Ridge) Regularization

- L2 (Ridge) regularization is given as the sum of weight squares

$$\boxed{R(w) = \sum_i w_i^2}$$

$$J_{reg}(w) = J(w) + \lambda \sum_i w_i^2$$

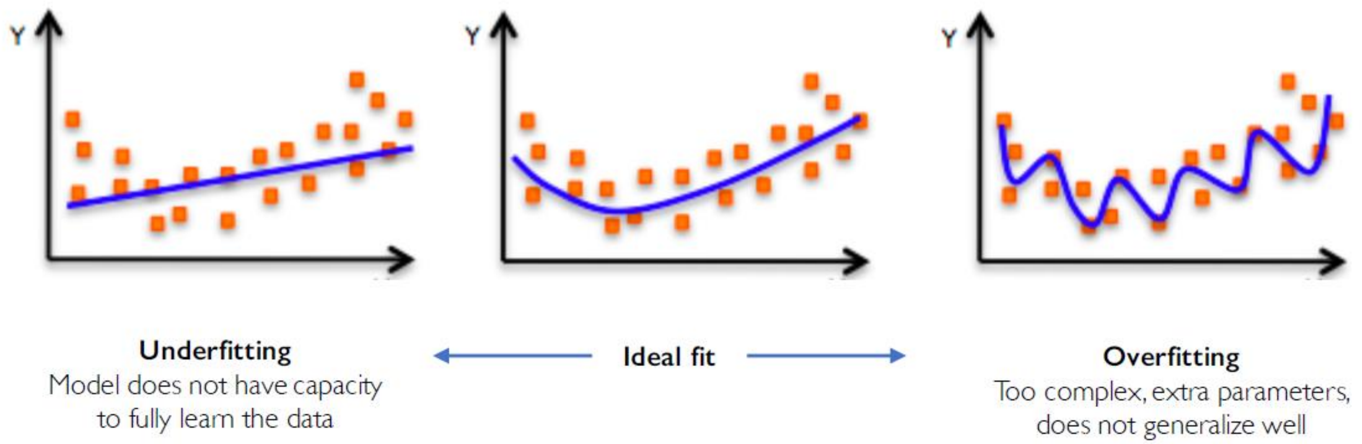
- Regularization term penalizes big weights
- Big weight decay coefficient $\rightarrow$ big penalty to big weights
- Squared weights penalizes large values more
- Generate a unique solution

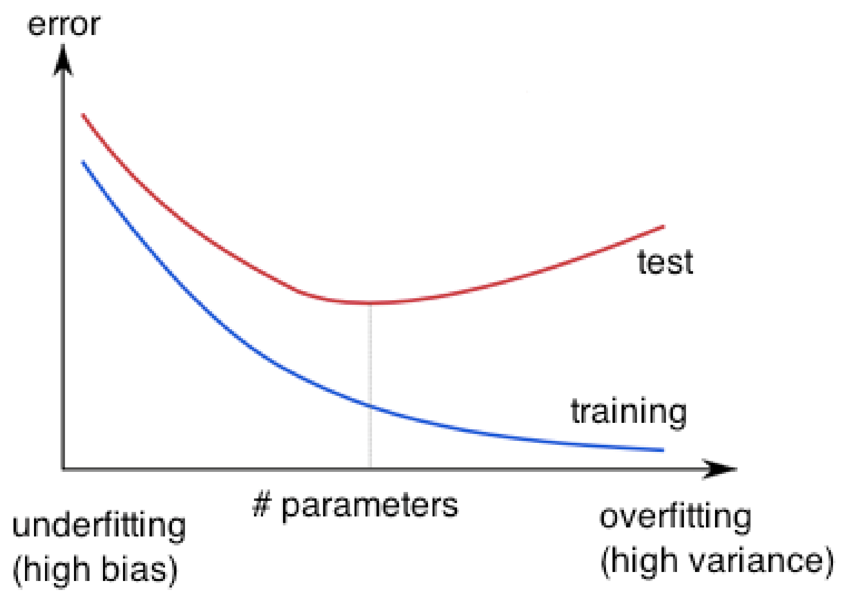
## Overfitting

## Overfitting

- Every machine learning has the issue of overfitting problem
- Occam’s razor: prefer simpler model.
- In decision tree, we use pruning technique to avoid overfitting problem
  - Makes the decision tree simpler
- Overfitting is a serious problem in deep learning since it has many layers with too many parameters
- Methods used in Deep Learning (or neural network) to avoid overfitting
  1) Regularization
  2) Dropout
  3) Data Augmentation
  4) Early Stop
  5) etc

# Overfitting

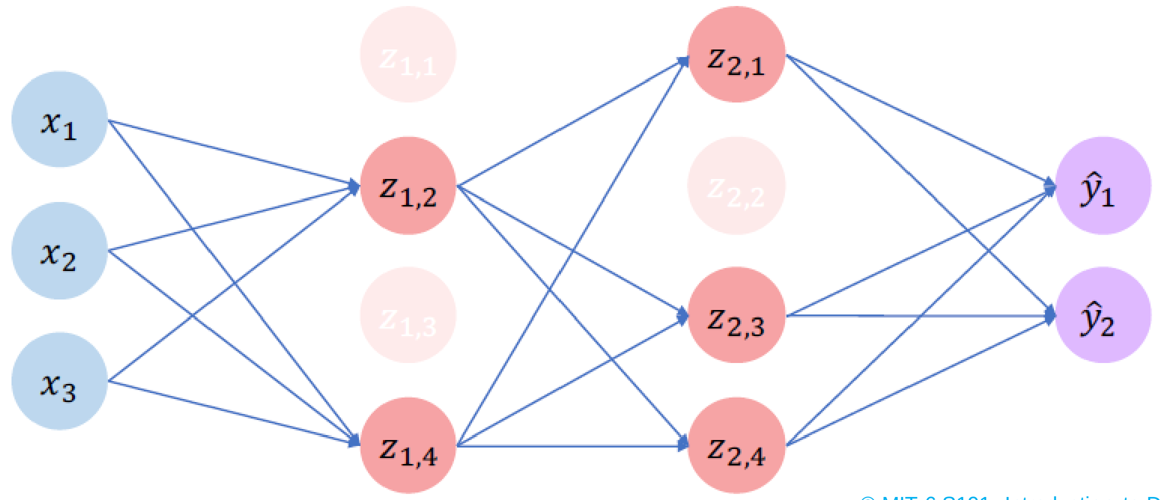

Y
Y
Y
Underfitting
Model does not have capacity to fully learn the data
Ideal fit
Overfitting
Too complex, extra parameters, does not generalize well



Overfitting: Learned hypothesis may **fit** the training data very well, even outliers (**noise**) but fail to **generalize** to new examples (test data)

# Dropout

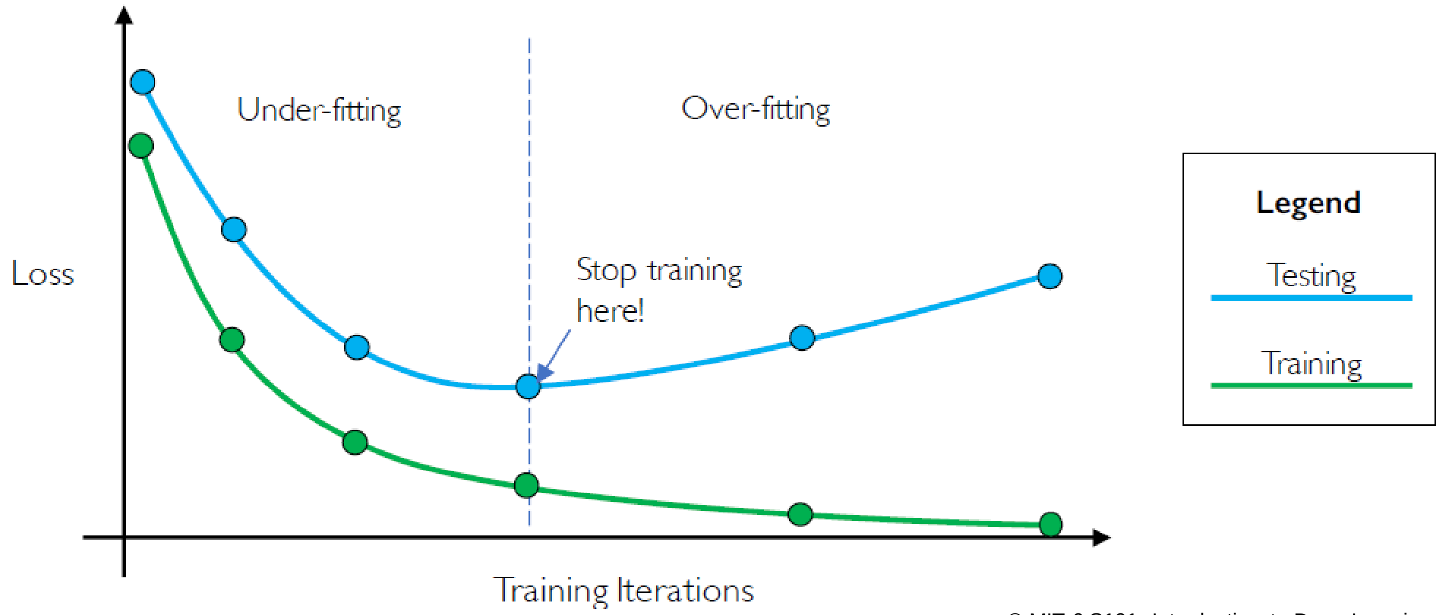
- During training, randomly set some activations to 0  during training
- Each unit retained with fixed probability p, independent of other units
- Hyper-parameter p to be chosen (tuned)
- Typically drop 50% of activations in layer
- Forces network to not rely on any 1 node
- Select different dropout nodes at each mini-batch
- May use different p value in layerwise



© MIT 6.S191: Introduction to Deep Learning
introtodeeplearning.com

## Early Stopping

- Use validation error to decide when to stop training
- Stop training before we have a chance to overfit
- Stop when monitored quantity has not improved after n subsequent epochs
- n is called patience
- generally this is not recommended by some people



© MIT 6.S191: Introduction to Deep Learning
introtodeeplearning.com

## Data Augmentation

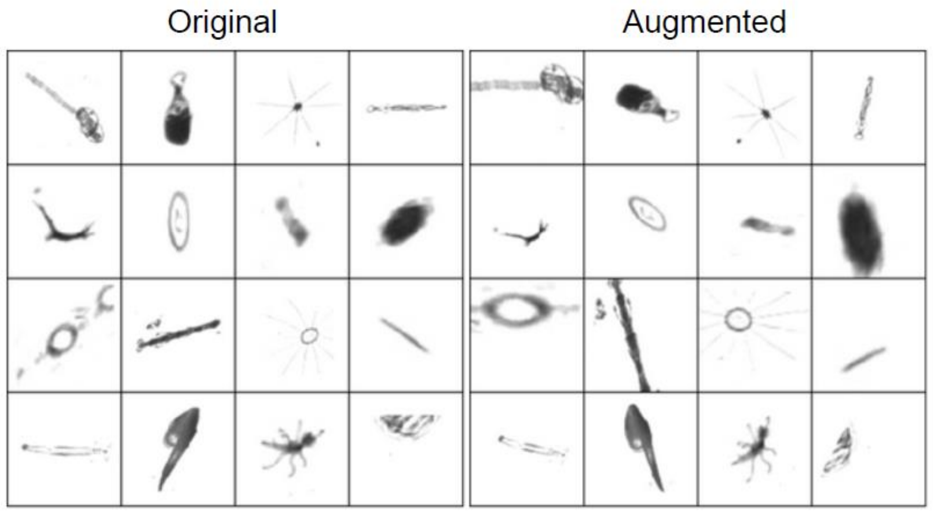
- In practice, the amount of data we have is limited.
- One way to get around this problem is to create fake data and add it to the training set.
- Dataset augmentation has been a particularly effective technique for a specific classification problem: object recognition.
- Images are high dimensional and include an enormous variety of factors of variation, many of which can be easily simulated.
- Operations like translating the training images a few pixels in each direction can often greatly improve generalization

## Data Augmentation

- Techniques for Data Augmentation
  - Flip: flip images horizontally and vertically
  - Rotation: image dimensions may not be preserved after rotation
  - Scale: image can be scaled outward or inward
  - Crop: randomly sample a section from the original image. Then resize this section to the original image size
  - Translation: move the image along the X or Y direction (or both).

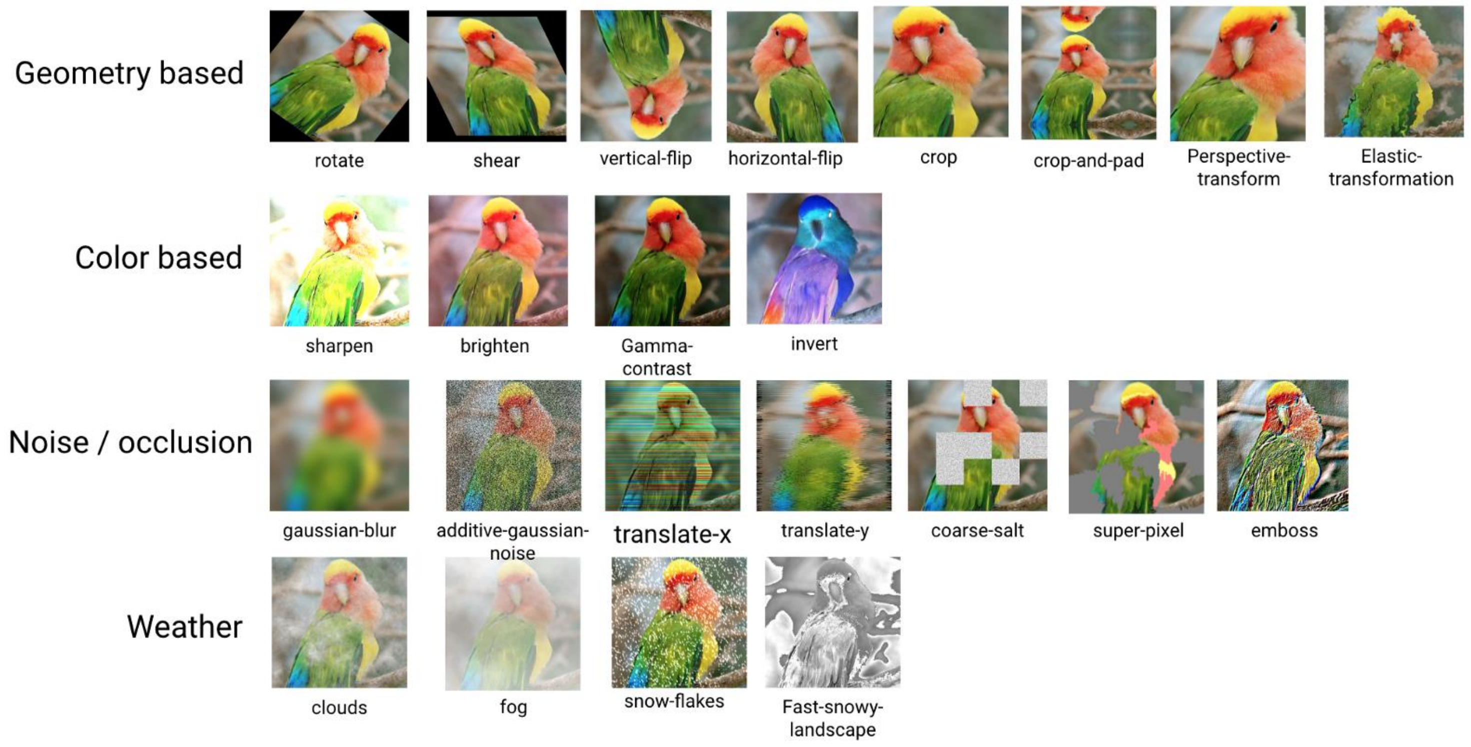
- (*) Advanced Augmentation Techniques
  - E.g.: Landscapes in different seasons.
  - Use GAN (generative Adversarial Network) to generate new images for data augmentation

# Data Augmentation



Pre-processed images (left) and augmented versions of the same images (right).

Source: Ian Goodfellow et al. Deep Learning, MIT press, 2016

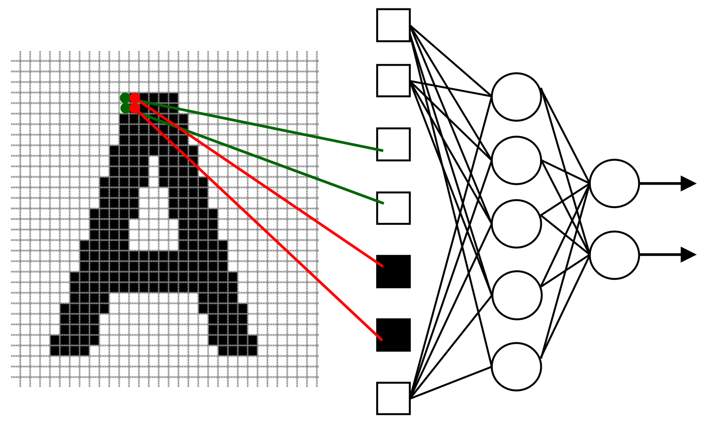


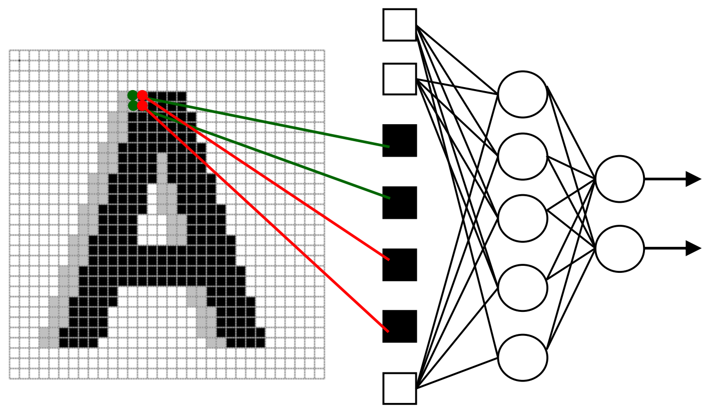
https://blog.insightdatascience.com/automl-for-data-augmentation-e87cf692c366?gi=bc0b8b5a353a

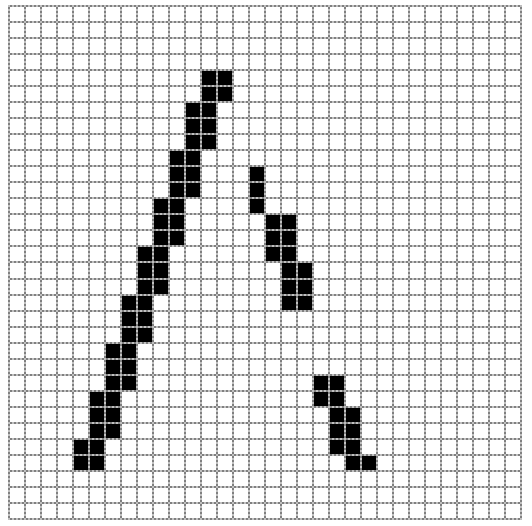
## CNN

## Drawbacks of Neural Networks

- Little or no invariance to shifting, scaling, and other forms of distortion







154 input change from 2 shift left
77 : black to white
77 : white to black

## Basic Ideas of Convolutional Neural Networks

- In traditional neural networks, every neuron in the network is connected to every neuron in adjacent layers (fully-connected)

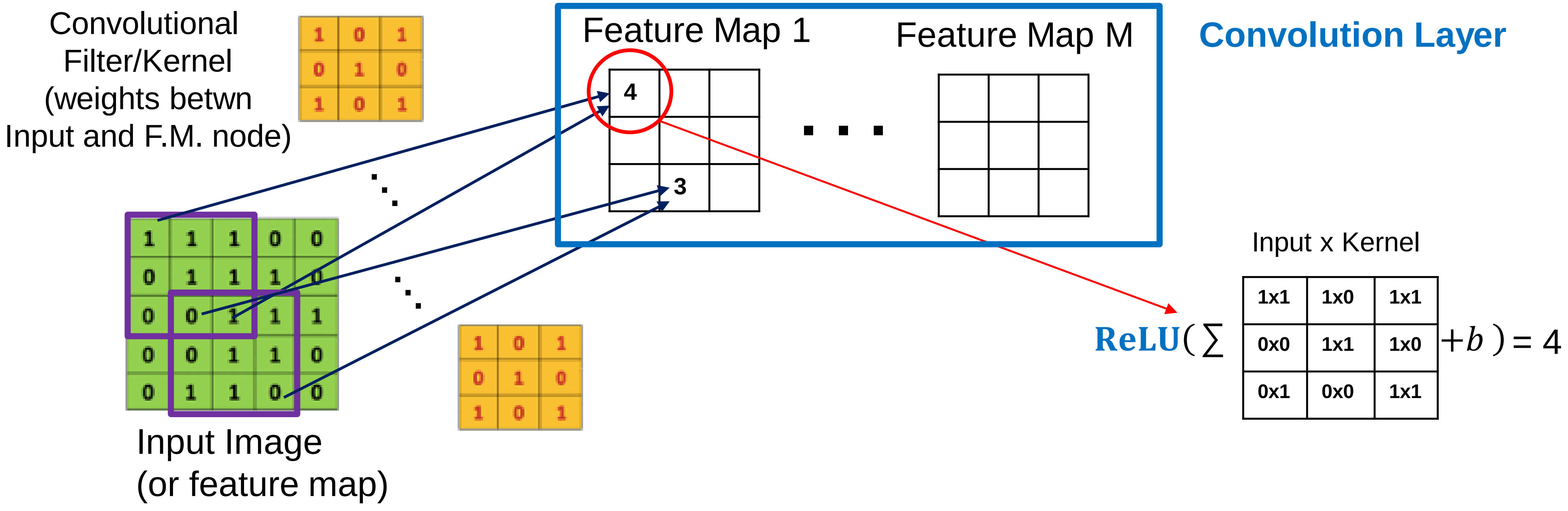
- As we increase the number of hidden layers (deep neural network), the number of parameters (weights) increases exponentially, causing overfitting

- How do we significantly reduce the number of parameters with hidden layers

- Convolutional neural networks(CNN) use a special architecture which is particularly well-adapted to classify images.

  1) Every node in hidden layer has the same weight vector with lower level layer nodes.
  2) Every node in hidden layer is connected with a small portion of lower level layer
  3) The number of nodes in hidden layer is reduced even further by using pooling layer

# Convolutional Layer

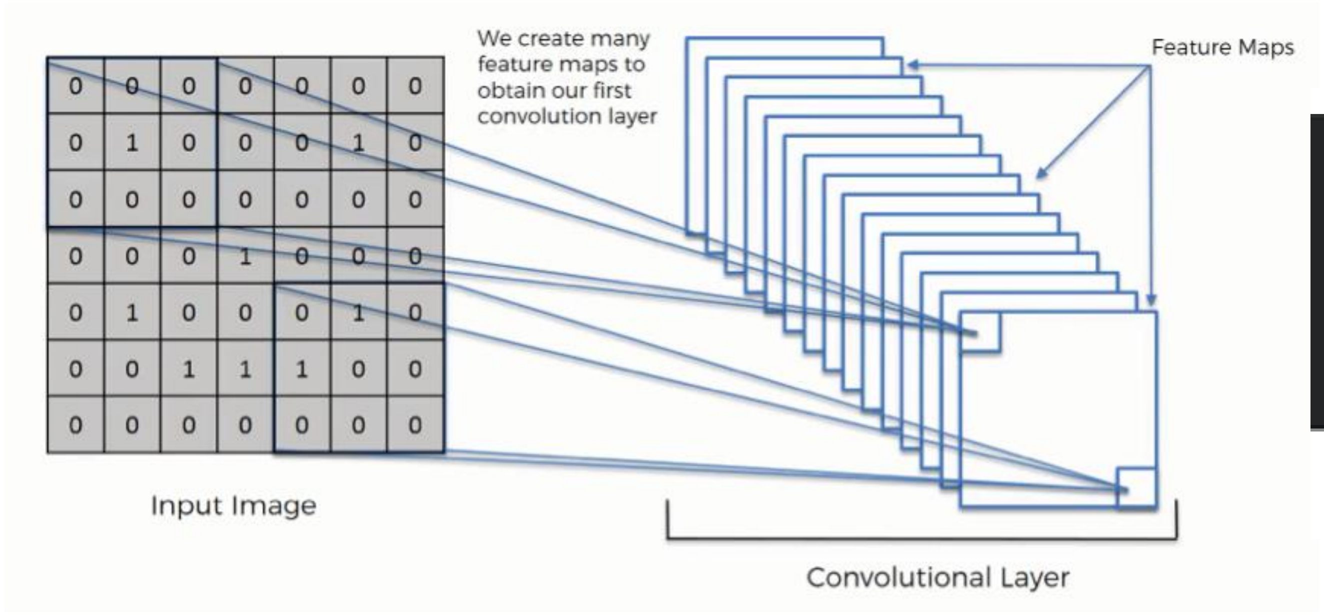

Convolutional Filter/Kernel (weights betwn Input and F.M. node)
Feature Map 1
Feature Map M
Convolution Layer
Input x Kernel
Input Image (or feature map)
4
3

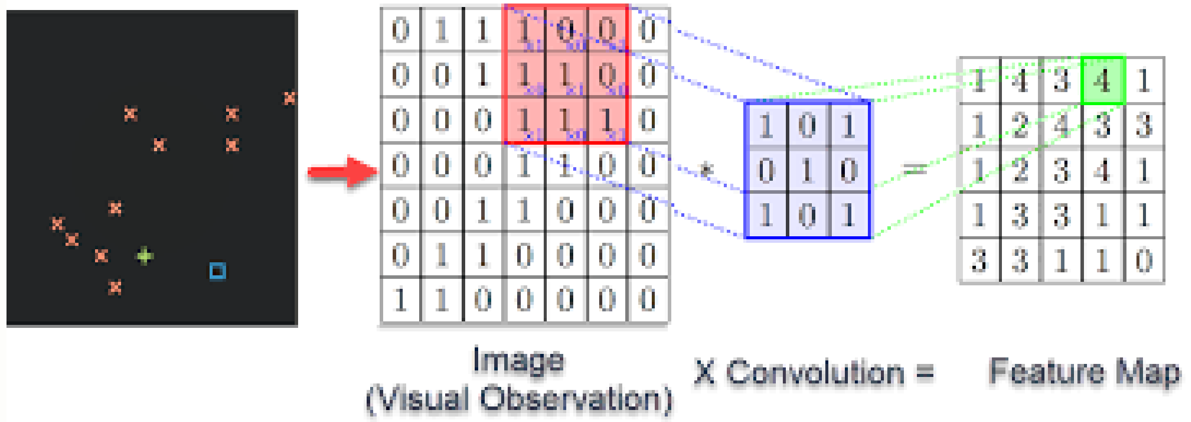
$$\mathrm{ReLU}\left(\sum \begin{array}{|c|c|c|} \hline 1\text{x}1 & 1\text{x}0 & 1\text{x}1 \\ \hline 0\text{x}0 & 1\text{x}1 & 1\text{x}0 \\ \hline 0\text{x}1 & 0\text{x}0 & 1\text{x}1 \\ \hline \end{array} + b\right) = 4$$

- Convolution filter represents the weight values between Input and feature map
- Every node in the same feature map shares the same convolution filter (weight sharing)
- Every node in feature map looks at a small portion of input (sparse connectivity)
  - Each node in feature map looks at a different region of input
- Convolution Filters are NOT given by human. It is learned from algorithm

# Convolutional Layer

- Convolutional Layer consists of multiple feature maps
- Each feature map has different Convolution Filters
- Each feature map is to detect a certain part of an image (e.g. nose)

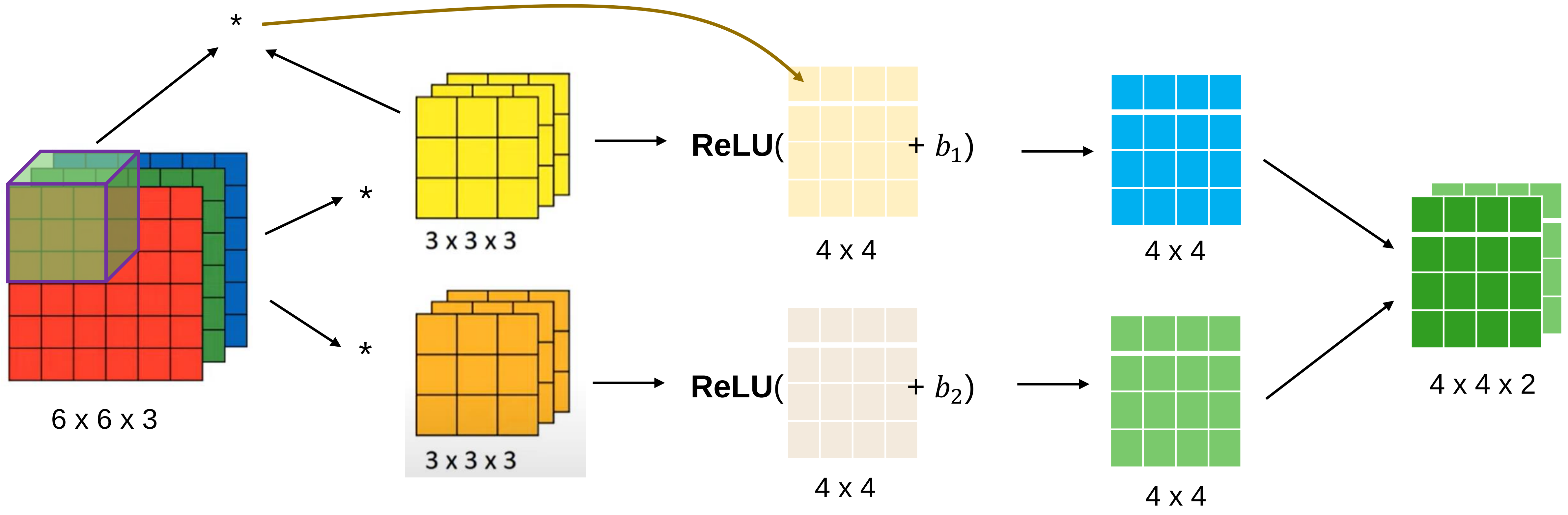

We create many feature maps to obtain our first convolution layer
Feature Maps
Input Image
Convolutional Layer



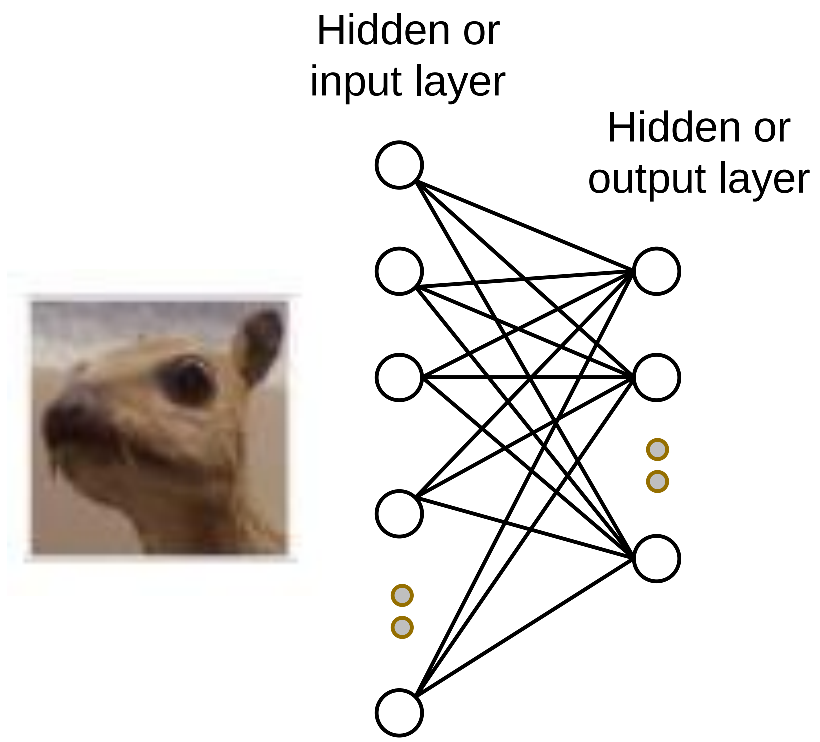
https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-1-convolution-operation

https://www.oreilly.com/library/view/learn-unity-ml-agents/9781789138139/16671cc8-0aff-433a-878c-7430be8b9aa1.xhtml

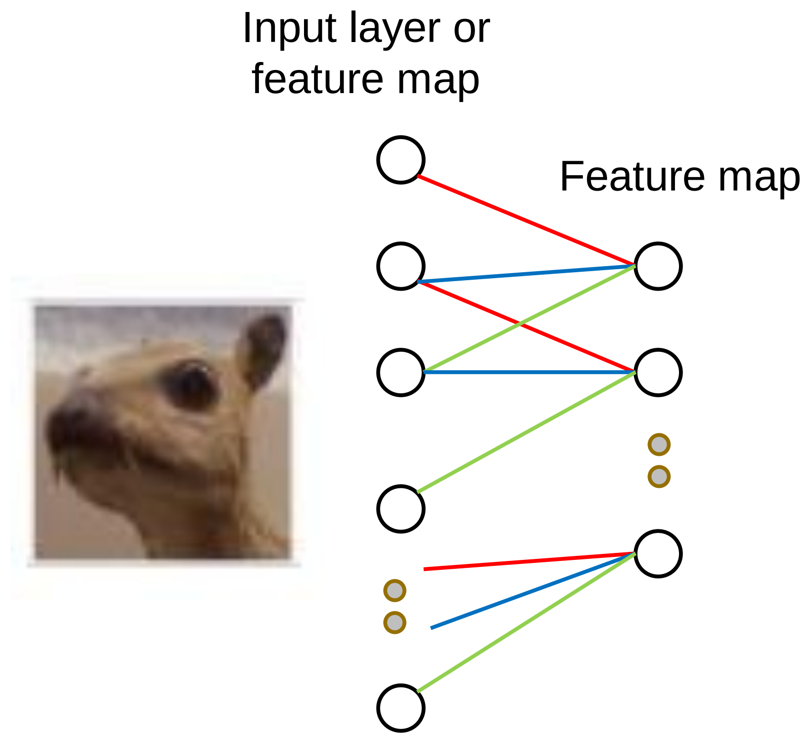
# Convolution Layer (multiple channels)



## Sparse Connectivity & Weight Sharing



Fully Connected



Sparse Connectivity/
Weight sharing

## Role of Feature Maps

- All the neurons in the first hidden layer detect exactly the same feature. Informally, think of the feature detected by a hidden neuron as the kind of input pattern: it might be an edge in the image, for instance, or maybe some other type of shape, just at different locations in the input image.

- To do image recognition we'll need more than one feature map. And so a complete convolutional layer consists of several different feature maps

- Convolutional networks are well adapted to the translation invariance of images

- A big advantage of sharing weights and biases is that it greatly reduces the number of parameters involved in a convolutional network

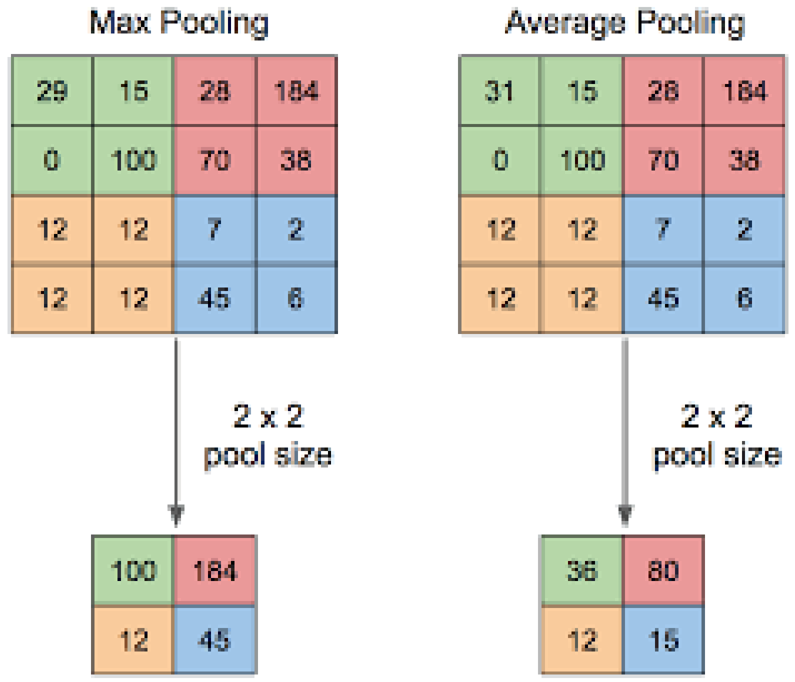
## Pooling Layer

- Pooling layers are usually used immediately after convolutional layers. What the pooling layers do is simplify the information in the output from the convolutional layer
- A pooling layer takes each feature map output from the convolutional layer and prepares a condensed feature map
- A big benefit is that there are many fewer pooled features, and so this helps reduce the number of parameters needed in later layers.

# Pooling Layer

- For instance, each unit in the pooling layer may summarize 2X2 neurons in the previous layer.
- One common procedure for pooling is known as *max-pooling*. In max-pooling, a pooling unit simply outputs the maximum activation in the 2X2 region
- We can think of max-pooling as a way for the network to ask whether a given feature is found anywhere in a region of the image. It then throws away the exact positional information.

Feature Map

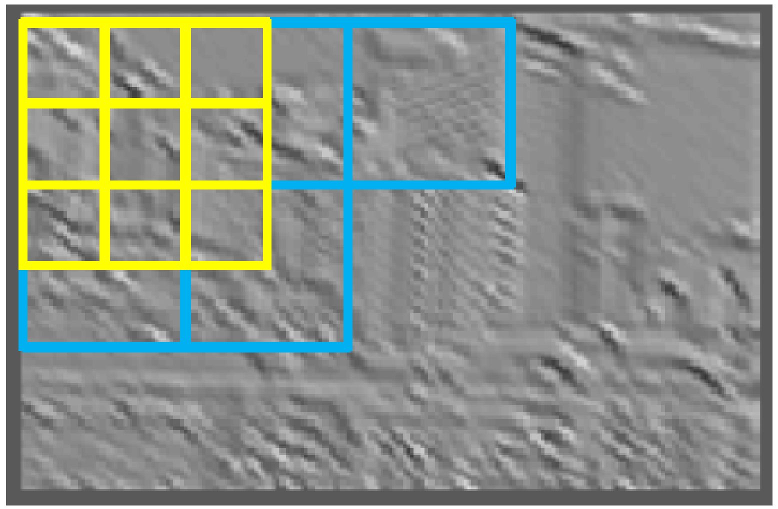


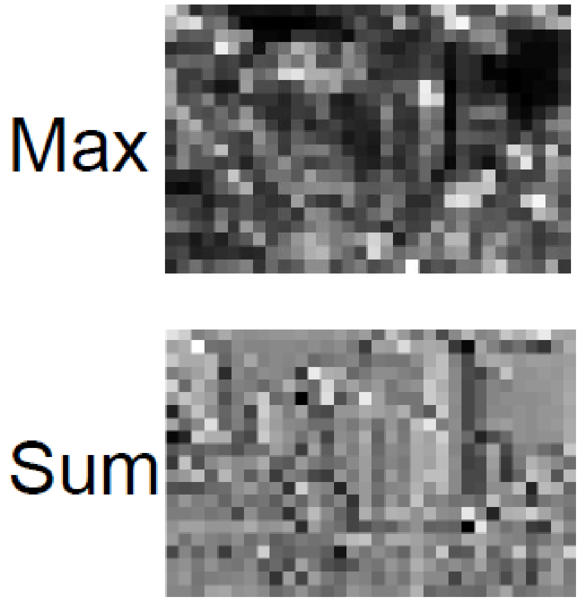
Pooling Layer

# Pooling Layer

- The intuition is that once a feature has been found, its exact location isn't as important as its rough location relative to other features.

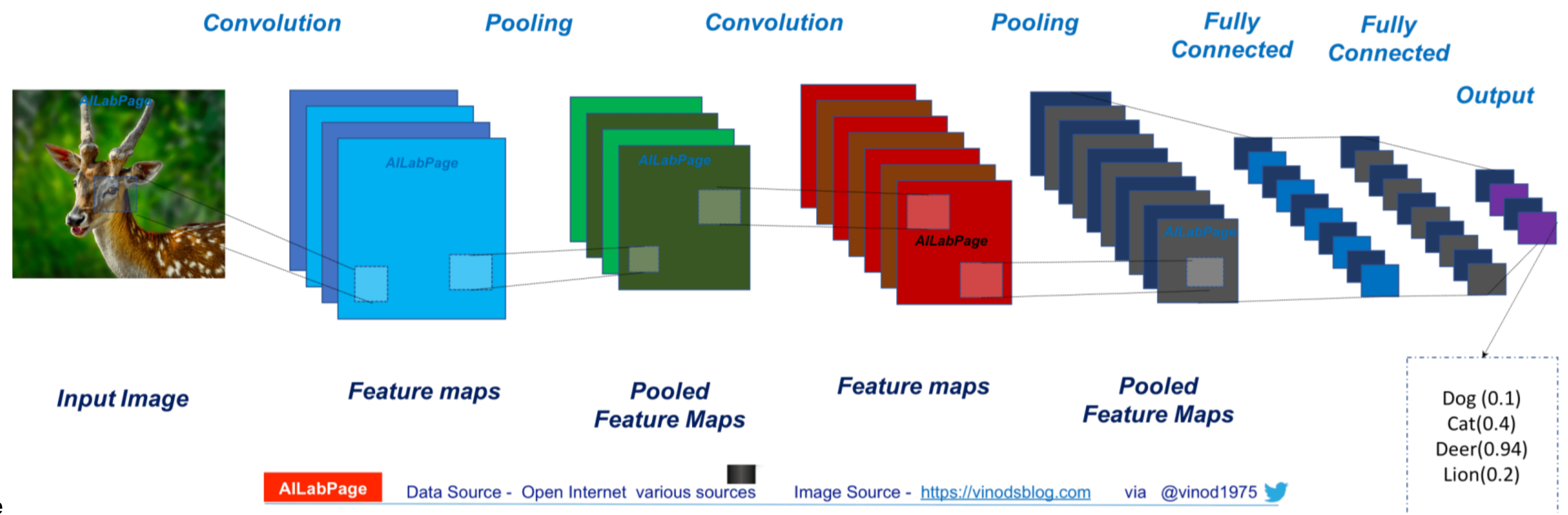
- Max-pooling isn't the only technique used for pooling. Other common approach are known as *Sum pooling & Average Pooling*. Here, instead of taking the maximum activation, we take the sum (or average) of the activations in the region.



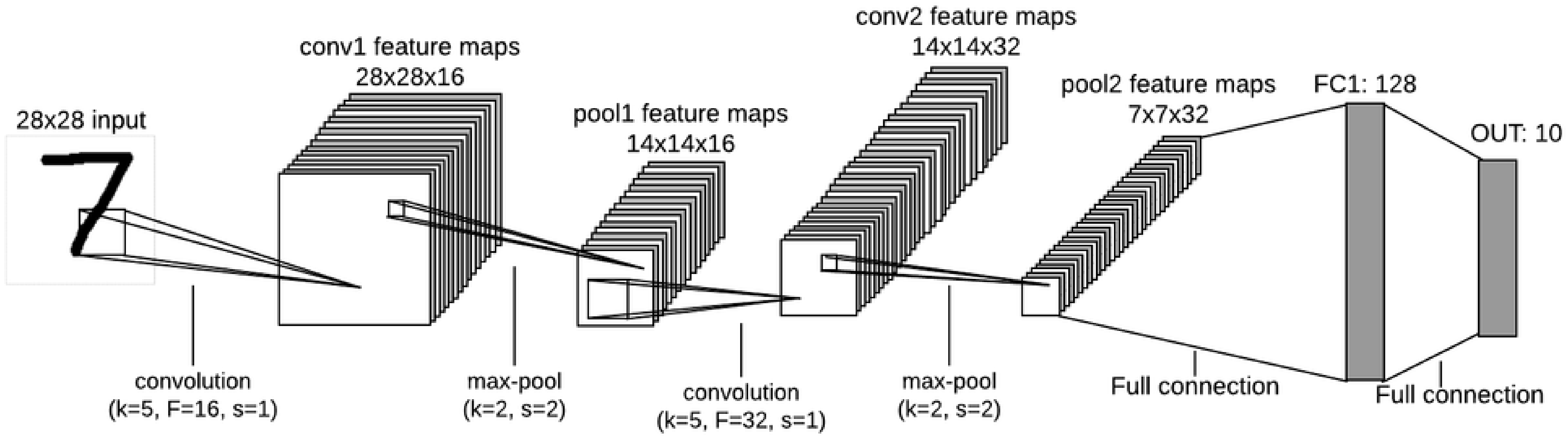


# Putting It All Together

- The final layer of connections in the network is a fully-connected layer. That is, this layer connects *every* neuron from the max-pooled layer to every one of the output neurons.
- Train the network using stochastic gradient descent and backpropagation.
- Need to make few modifications to the backpropagation procedure due to *weight sharing, sparse connectivity, etc*



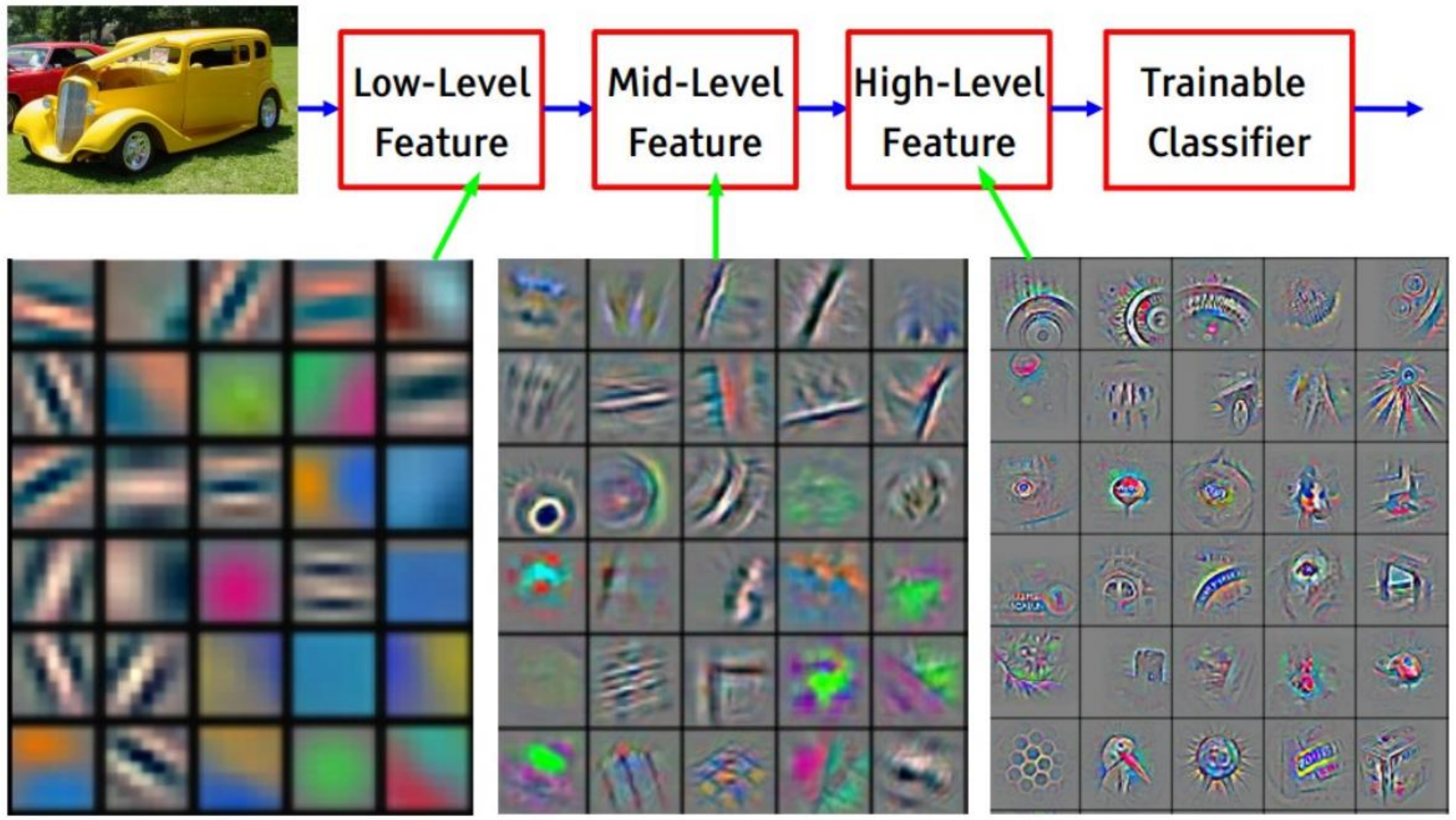
## CNN's Topology



https://www.easy-tensorflow.com/tf-tutorials/convolutional-neural-nets-cnns

- Convolution and/or pooling can be repeated many times
- Full connection
  - Regular neural network connection
  - No weight sharing

# Hierarchical Feature Extraction



## RNN

## Why do we need RNNs?

- The data we have seen so far is assumed to be i.i.d. (**i**ndependent and **i**dentical **d**istributions)
- Many real data is in temporal order.
  - Frames from video, Words in sentence, Snippets of audio,
- Characteristics of sequential data
  - 1) Current data depends on past data.
    - E.g.: Current word depends on previous words
  - Current frame depends on previous frame
  - 2) Temporal data is variable length
- The limitations of the Neural network (including CNNs)
  - Rely on the assumption of independence among the (training and test) examples.
    - After each data point is processed, the entire state of the network is lost
  - Rely on examples being vectors of fixed length
- We need to model the data with temporal/sequential structures and varying length of inputs and outputs

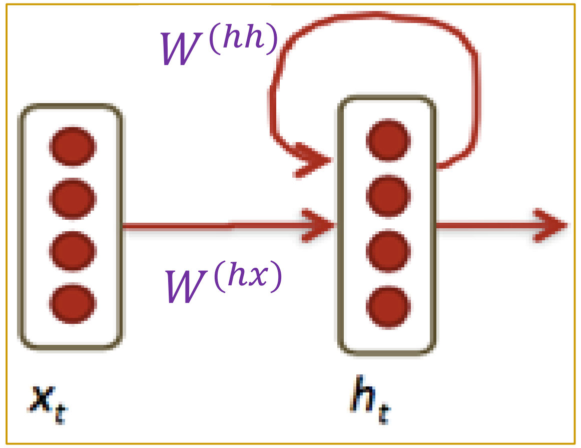
# Recurrent Neural Network

- RNN: Neural network where the current value of node is used in next time step
- Can produce an output at each time step
- Can naturally process variable length temporal data
- Train using back-propagation through time(BPTT)
- Use same set of weights at all time steps

NN: $h_t = \sigma(W^{(hx)} x_t)$

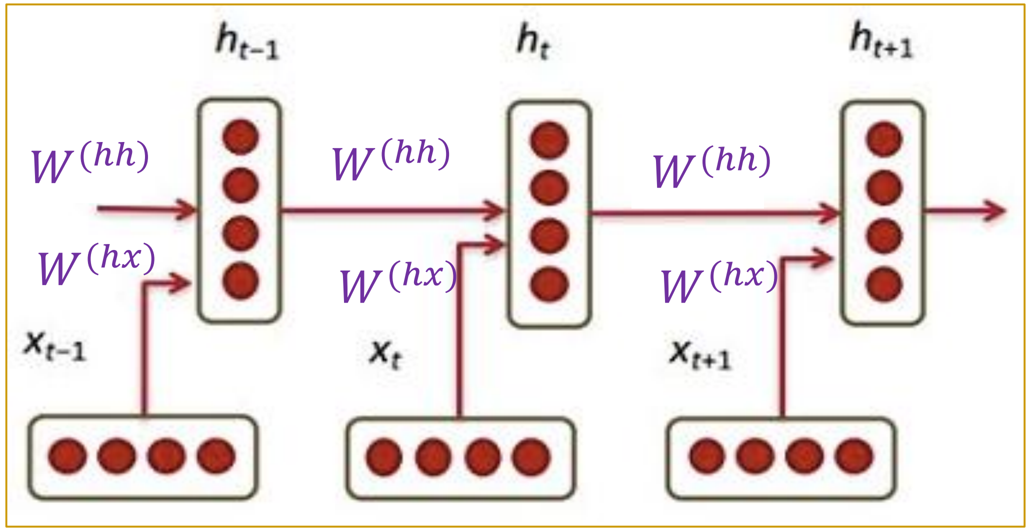
RNN: $h_t = \sigma(W^{(hh)} h_{t-1} + W^{(hx)} x_t)$

Previous state



https://pbs.twimg.com/media/C2j-8j5UsAACgEK.jpg

$\triangleq$

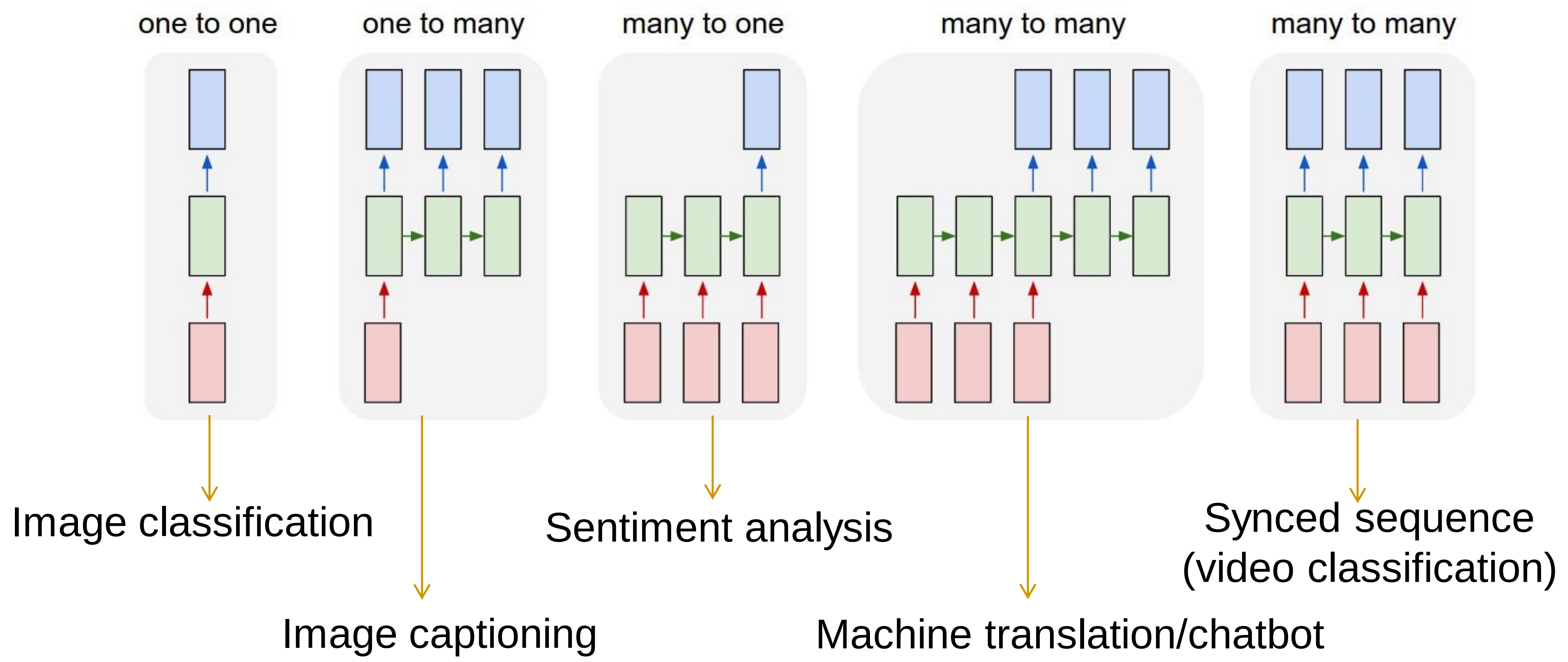


# Sharing Weights in RNN

- In CNN, we share the weights across different regions.
- Sharing weights could reduce the number of parameters & improve the performance
- Let's share the weights AGAIN in RNN
  - Share weights across regions -> CNN
  - Share weights across time step -> RNN
  - Share weights across tree structure -> Recursive NN (not covered here)
- Sparse connectivity (in CNN) is not used in RNN. RNN is fully connected network.

# Sequence to Sequence(Seq2Seq) Learning

- Since RNN can produce an output at each time step, it enables Seq2Seq learning
- Seq2Seq learning: given input, algorithm generates a sequences of output



http://cs231n.stanford.edu

# Seq2Sequence Learning Example

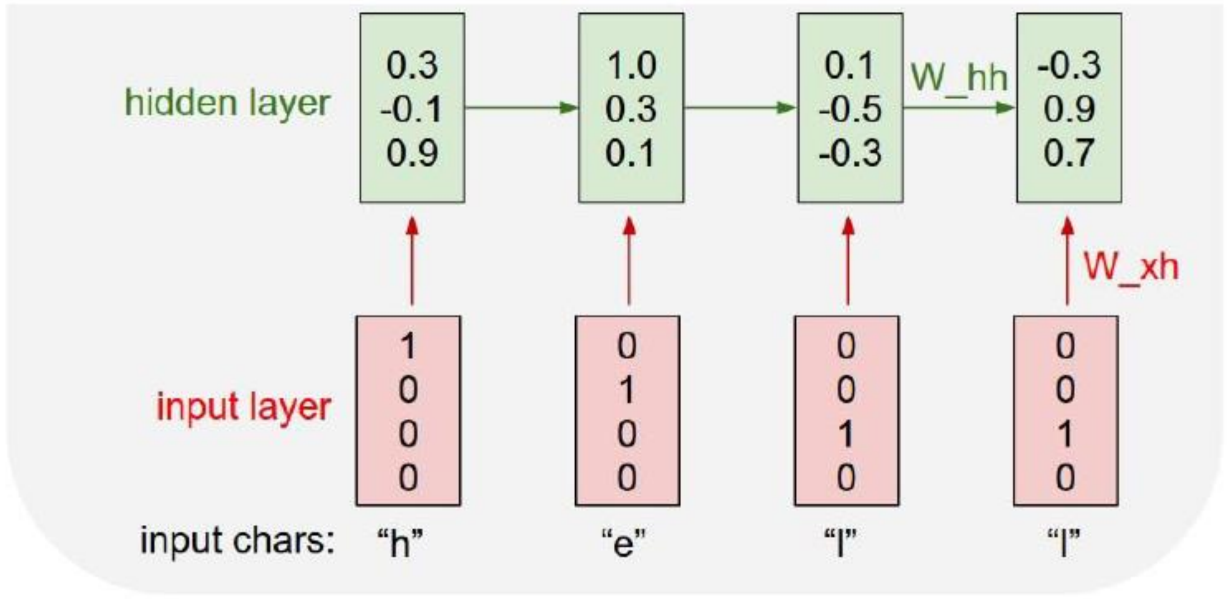
- e.g.: train RNN to generate “hello” in sequence

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

**Character-level language model example**

Vocabulary: [h,e,l,o]

Example training sequence: **“hello”**



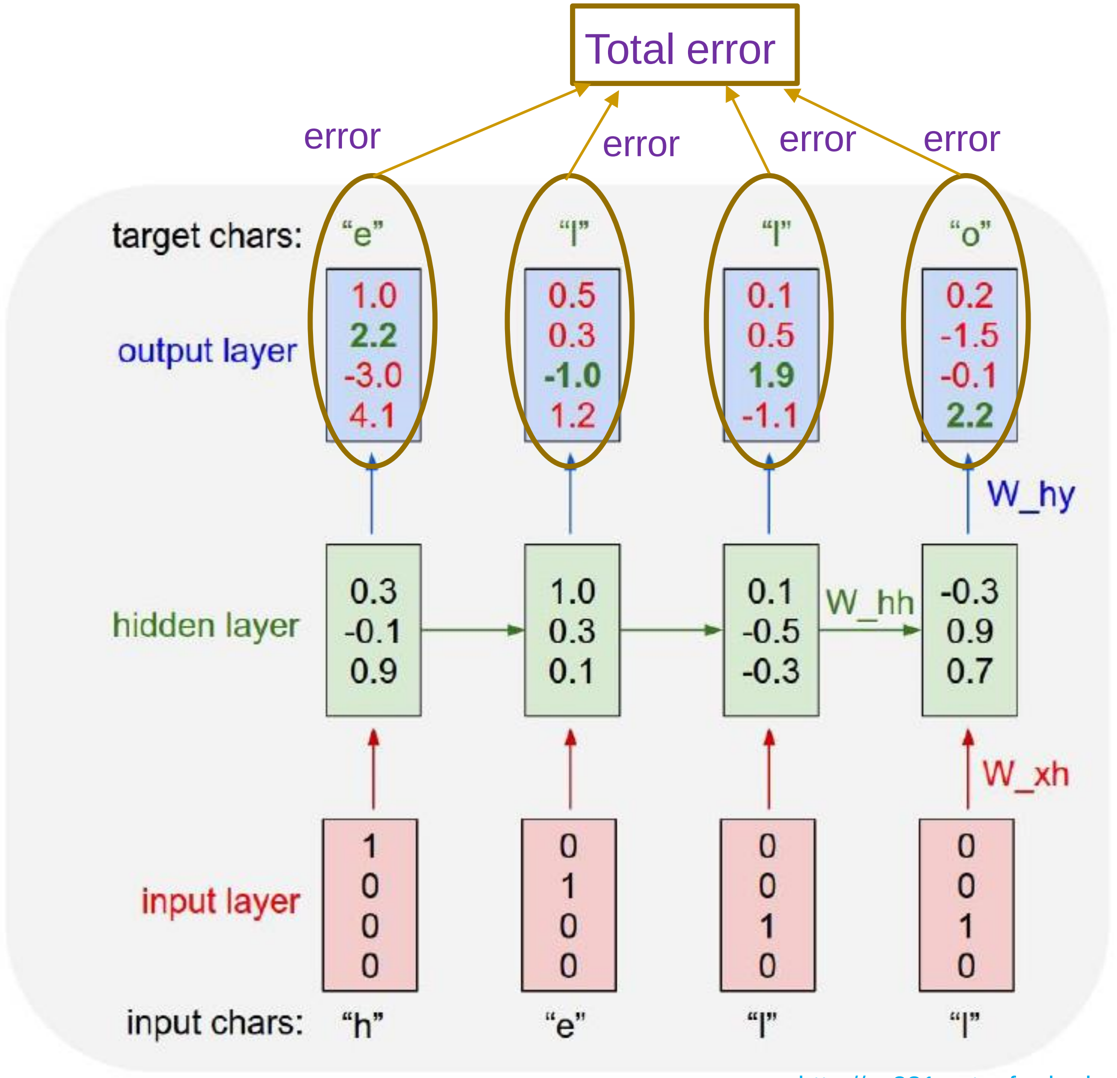
http://cs231n.stanford.edu

## Seq2Sequence Learning Example

**Character-level language model example**

Vocabulary: [h,e,l,o]

Example training sequence: **“hello”**



http://cs231n.stanford.edu

**Training time:**

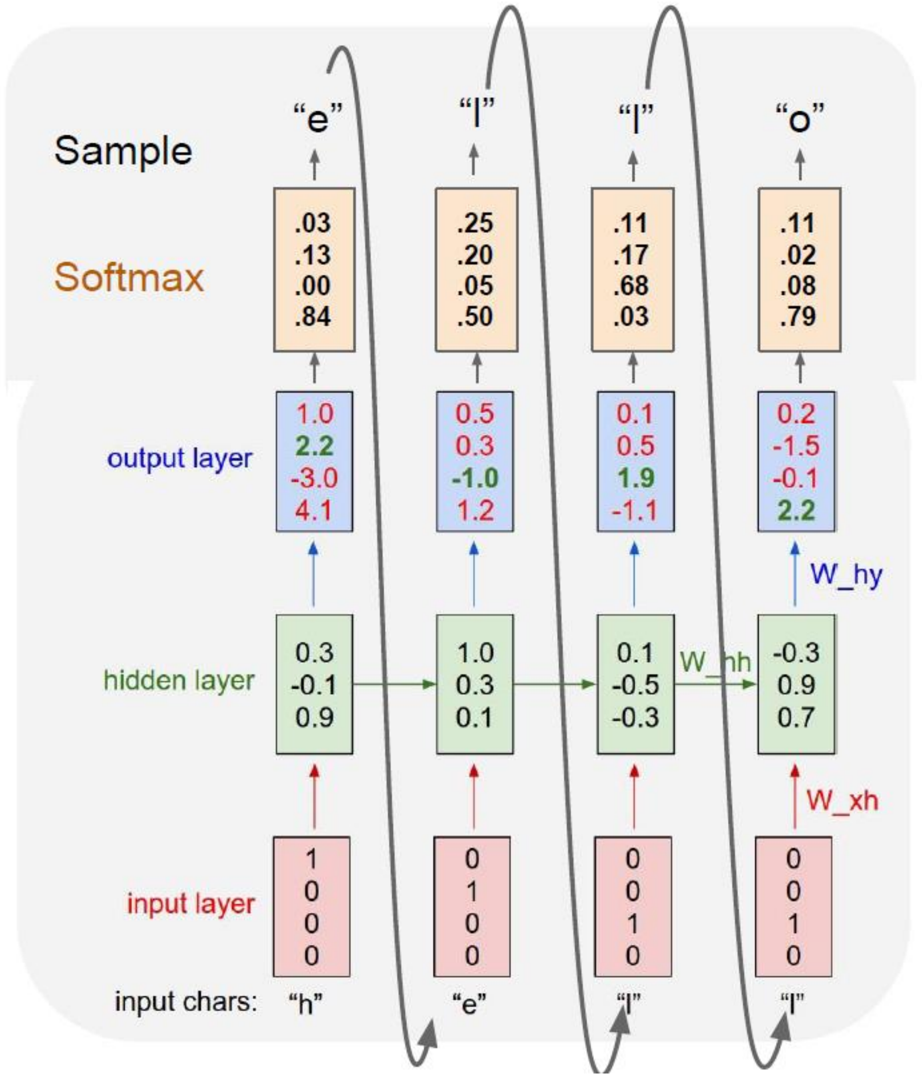
- Use BPTT to train entire RNN network
- (W_hy, W_hh, W_xh) is same in each time step

## Seq2Sequence Learning Example

**Character-level language model example**

Vocabulary: [h,e,l,o]

Example training sequence: **"hello"**



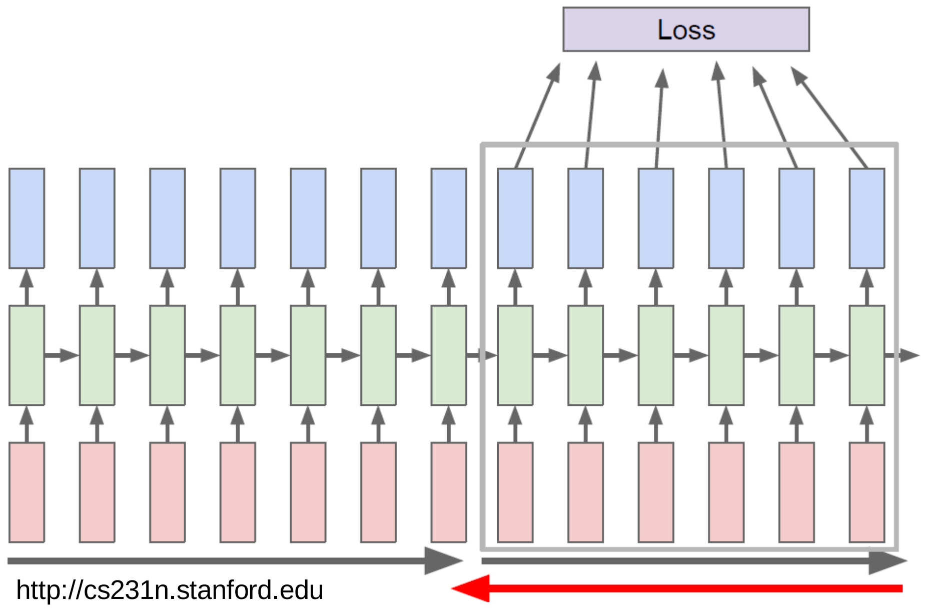
http://cs231n.stanford.edu

**Test time:**

- Generate output at each time step
  - pick a seed character sequence
  - generate the next character
  - feed back into the model
  - then the next …
- This process is basic concept of generative learning
- When generating next sample, use probabilistic sampling from softmax (Not deterministic sampling)
- We assumed to pick 'e' at the first step even though 'o' has the highest prob.

# (Truncated) Backpropagation Through Time

- Back Propagation Through Time (BPTT) is used to learn the RNN
- BPTT is an extension of the back-propagation (BP)
- Vanishing Gradients in RNN are Different from the Case in NNs
- If it was just a case of vanishing gradients in NNs, we could just rescale the per-layer learning rate, but that does not really fix the training difficulties
- Can’t do that with RNNs because the weights are shared, & total true gradient = sum over different “depths”
- We can’t look back forever, and use truncated BPTT

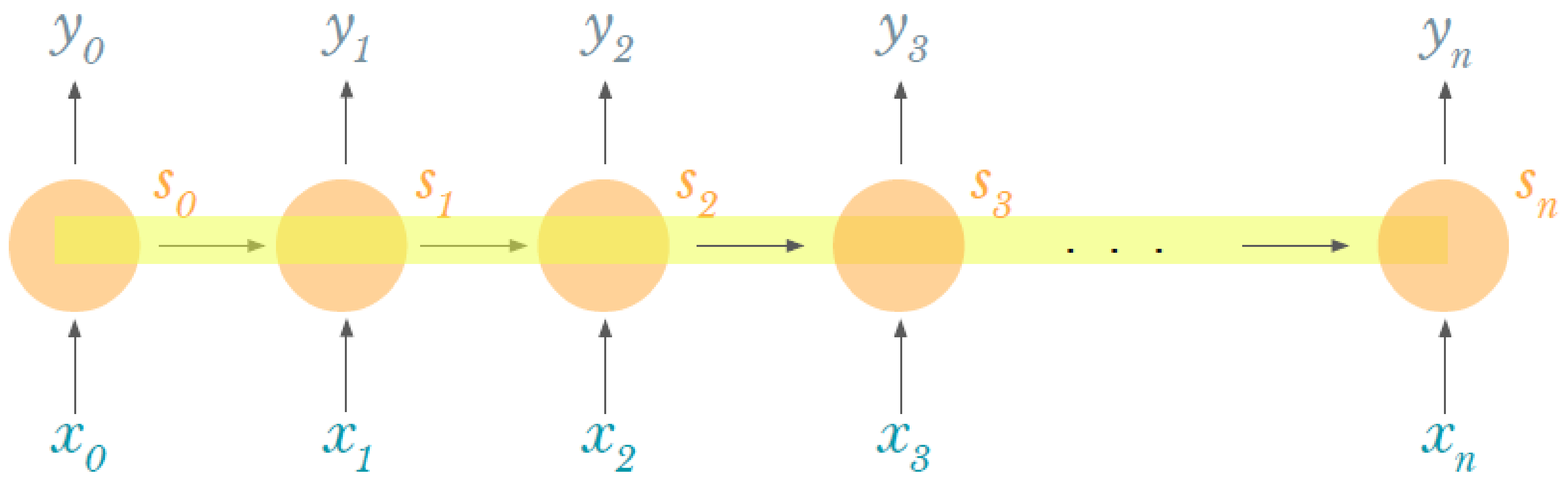


Summation of error in each step

http://cs231n.stanford.edu

Truncated

## (Truncated) Backpropagation Through Time



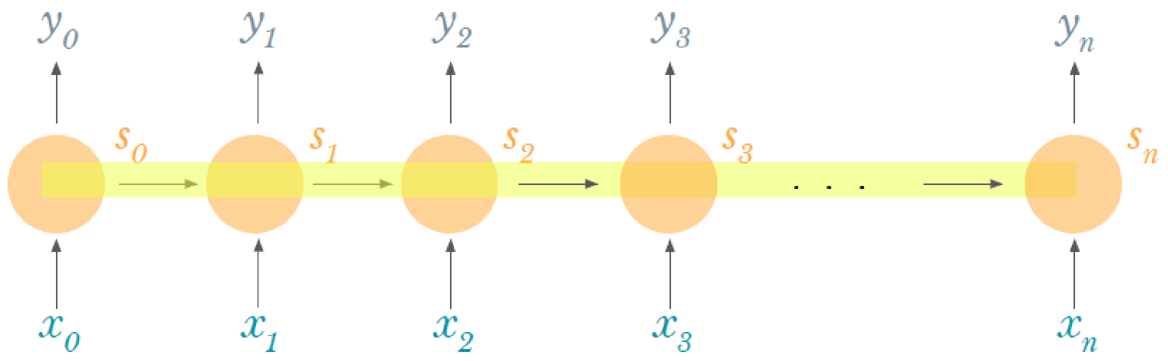
- Error function: $\boxed{J_n = -\sum_k L(y_k, \widehat{y_k})}$
- BPTT also uses gradient descent
  - Update $w = w - \eta \frac{\partial Jn}{\partial w}$

$\widehat{y_k}$ : true value
$y_k$ : prediction
$L$ : Loss function

# Vanishing Gradient in RNN

$$\frac{\partial J_n}{\partial W} = \sum_{k=0}^{n} \frac{\partial J_n}{\partial y_n} \frac{\partial y_n}{\partial s_n} \boxed{\frac{\partial s_n}{\partial s_k}} \frac{\partial s_k}{\partial W}$$

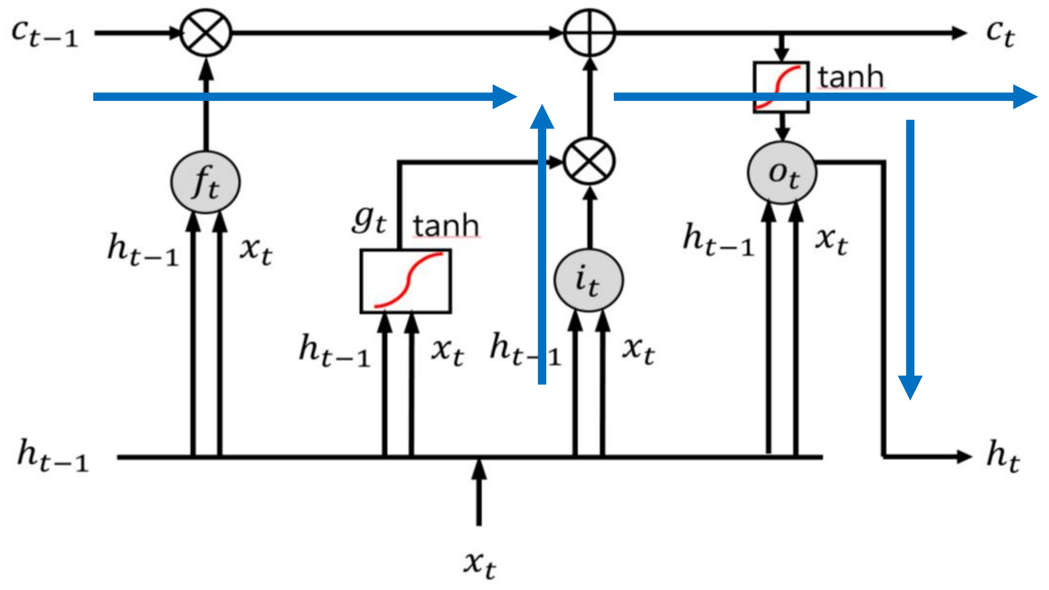
$$\frac{\partial s_n}{\partial s_{n-1}} \frac{\partial s_{n-1}}{\partial s_{n-2}} \cdots \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}$$



Gradient contributions from "far away" steps become zero, and the state at those steps doesn't contribute to what you are learning: You end up not learning long-range dependencies.

- Vanishing gradient in RNN means it forgets the past
- It doesn't remember what has happened in the past

# Long Short-Term Memory(LSTM)



1) $f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right)$ (forget gate)
2) $i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$ (input gate)
3) $o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$ (output gate)
4) $g_t = \tanh\left(W_{xg}x_t + W_{hg}h_{t-1} + b_g\right)$
5) $c_t = c_{t-1} \otimes f_t + g_t \otimes i_t$
6) $h_t = \tanh(c_t) \otimes o_t$

Traditional RNNs are a special case of LSTMs: Set the input gate, the forget gate, the output gate to all ones.

- Use memory cell to store information at each time step.

- Use “gates” to control the flow of information through the network.
  - Forget gate: limit information passed from one cell to the next
  - Input gate: how much of the new information will be let through the memory cell.
  - Output gate: how much of the information will be passed to expose to the next time step.

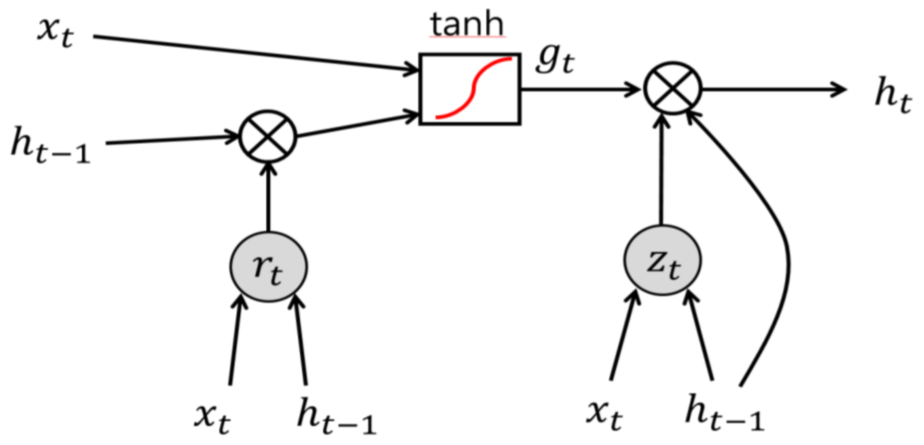
## Advantages of LSTM

- Non-decaying error backpropagation.
- For long time lag problems, LSTM can handle noise and continuous values.
- No parameter fine tuning.
- Memory for long time periods
- LSTM solves the vanishing gradient and the long memory limitation problem
- LSTM can learn sequences with more than 1000 time steps.

## Gated Recurrent Unit (GRU)

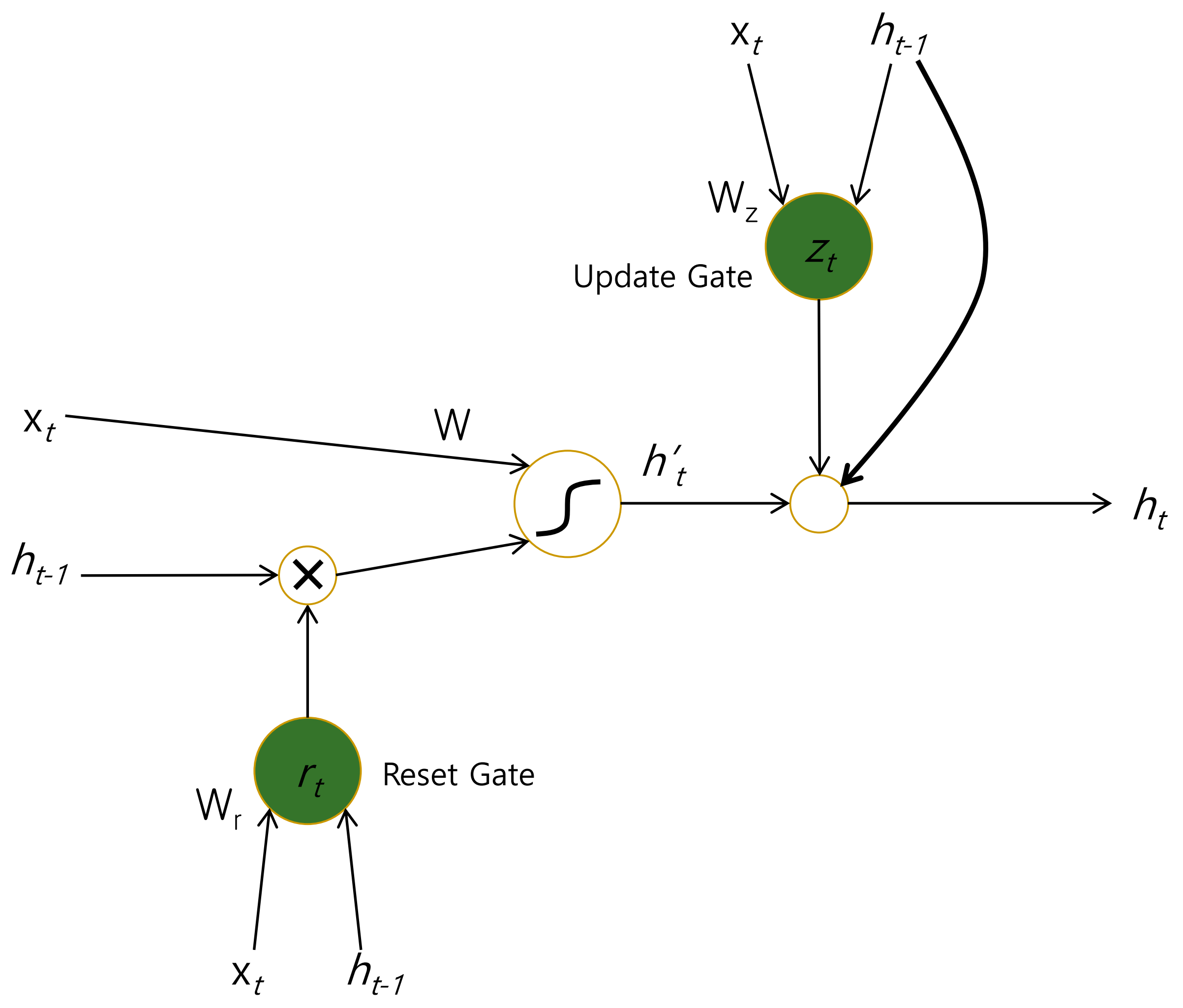
- A very simplified version of the LSTM
  - Only two gates: ‘update’ gate and ‘reset’ gate
  - Merges forget and input gate into a single ‘update’ gate
  - Merges cell and hidden state
  - No non-linearity in output value

- Has fewer parameters than an LSTM and has been shown to outperform LSTM on some tasks



1) $r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r)$ (reset gate)
2) $z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z)$ (update gate)
3) $g_t = \tanh\left(W_{hg}(r_t \otimes h_{t-1}) + W_{xg}x_t + b_g\right)$
4) $h_t = h_{t-1} \otimes z_t + g_t \otimes (1 - z_t)$

# GRU



- Compute a reset gate based on current input vector and hidden stat

$$r_t = s\left(W_r\begin{pmatrix}x_t\\ h_{t-1}\end{pmatrix}+b_f\right)$$

- Compute an update gate based on current input vector and hidden stat

$$z_t = s\left(W_z\begin{pmatrix}x_t\\ h_{t-1}\end{pmatrix}+b_f\right)$$

- New memory

$$h'_t = \tanh W\begin{pmatrix}x_t\\ r_t \otimes h_{t-1}\end{pmatrix}$$

- Final memory

$$h_t = (1- z_t)\text{Ä}\, h_{t-1} + z_t\,\text{Ä}\, h'_t$$