

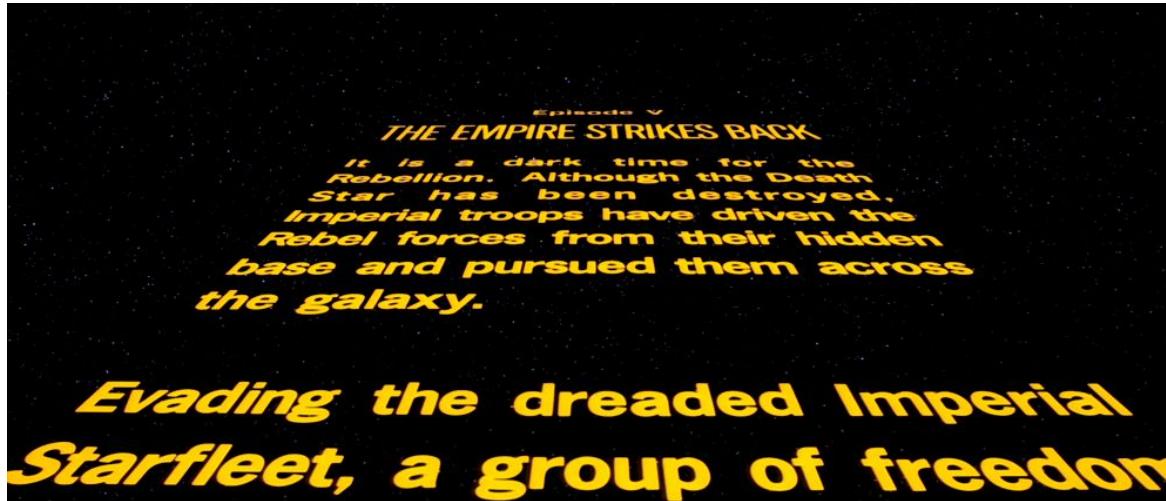
# TOKENIZATION

---

Intro to neural networks

# The need for segmentation

- Text as a stream of characters



- We need a way to understand the meaning of text
  - Break into words (assign meaning to word) ← Tokenization
  - Break into sentences (put word meanings back to sentence meaning)

# Tokenization

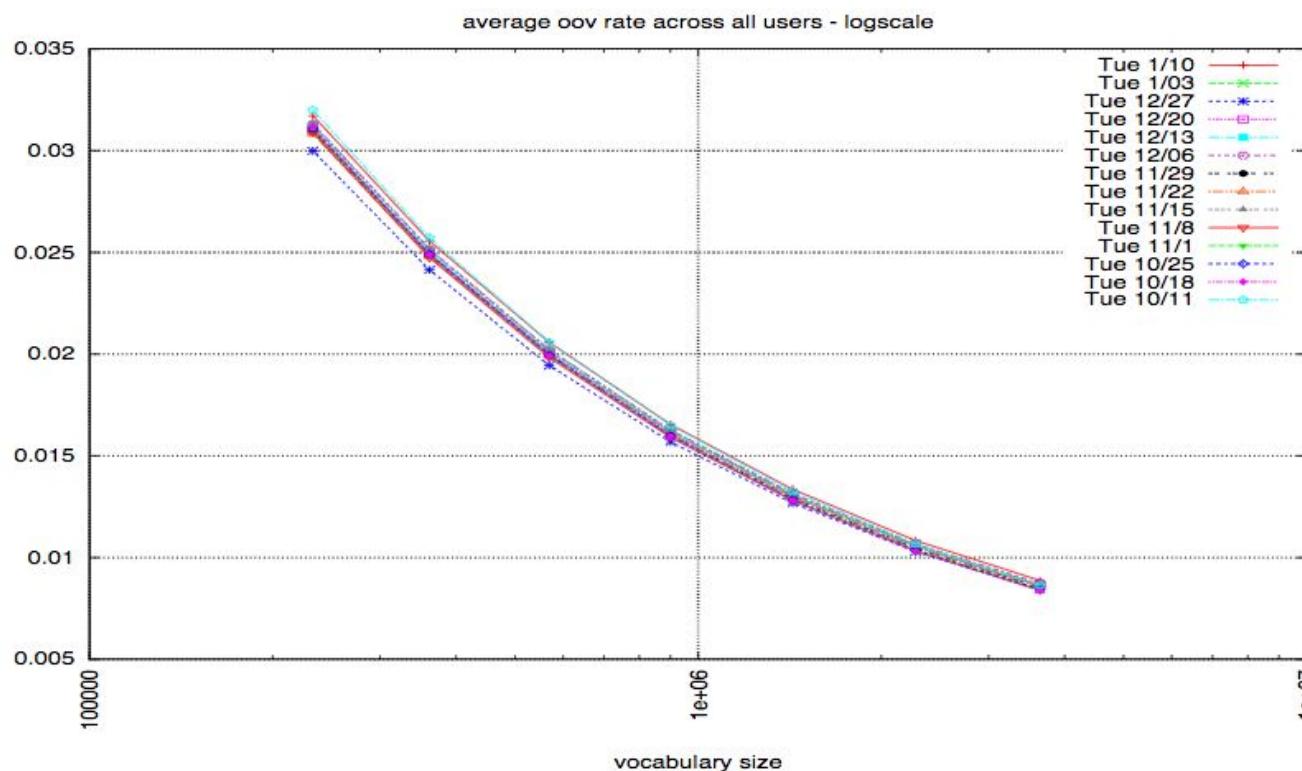
- A token should
  - 1. Linguistically significant
  - 2. Methodologically useful

# Dictionary-based vs Machine-learning-based

- Dictionary-based
  - Longest matching
  - Maximal matching
- Machine-learning-based

# Dictionary-based drawbacks

- Cannot handle words outside of the dictionary  
**(Out-of-Vocabulary, OOV words)**
- Performs worse than machine-learning-based approach



# NEURAL NETWORKS

---

Deep learning = Deep neural networks =  
neural networks

# DNNs (Deep Neural Networks)

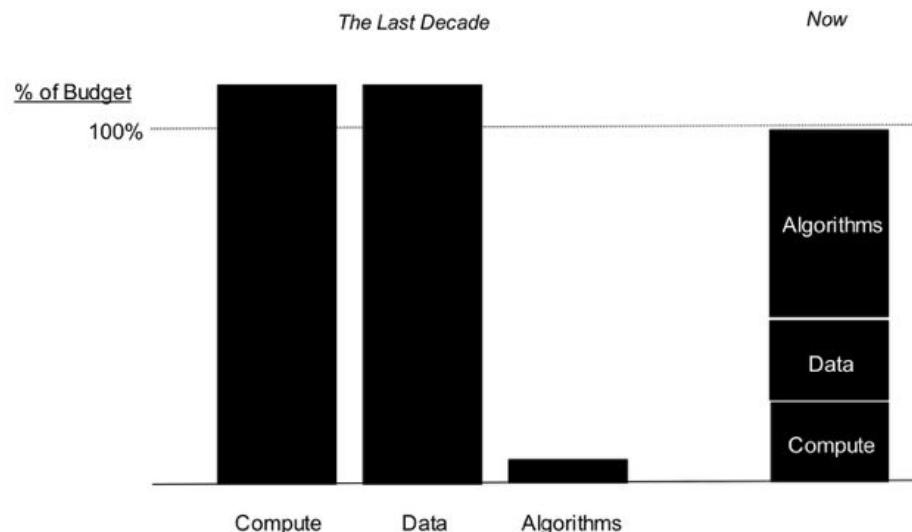
- Why deep learning?
- Greatly improved performance in ASR and other tasks (Computer Vision, Robotics, Machine Translation, NLP, etc.)
- Surpassed human performance in many tasks

Deep learning in NLP	Without Deep Learning	With Deep Learning
<b>Parsing</b>	92.89	<b>94.61</b>
<b>Machine Translation</b>	37.0	<b>41.16</b>
<b>Sentiment Analysis</b>	81.3	<b>88.6</b>
<b>Natural Language Inference</b>	78.2	<b>89.9</b>

Source: Regina Barzilay NLP@MIT

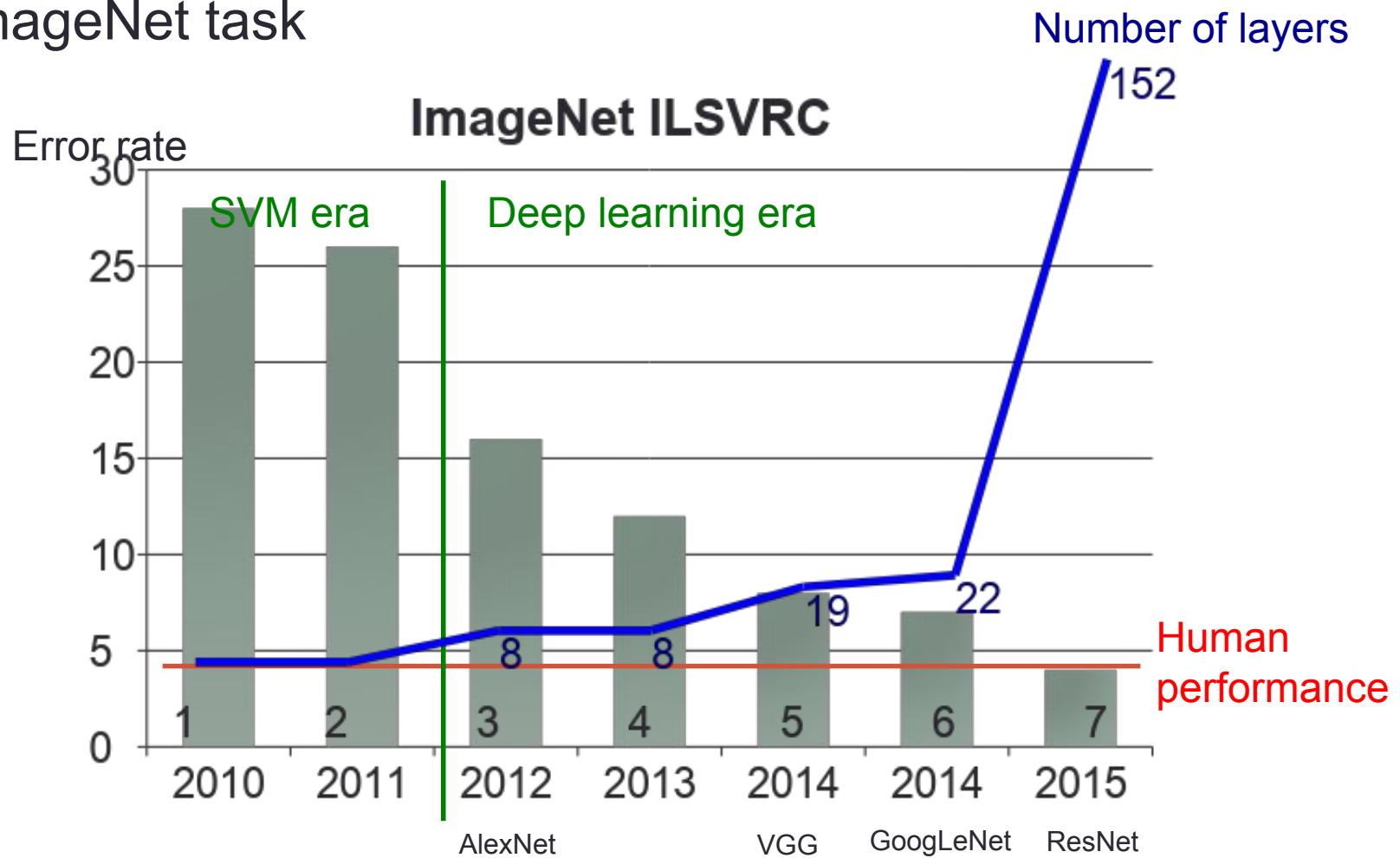
# Why now

- Neural Networks has been around since 1990s
- Big data – DNN can take advantage of large amounts of data better than other models
- GPU – Enable training bigger models possible
- Deep – Easier to avoid bad local minima when the model is large

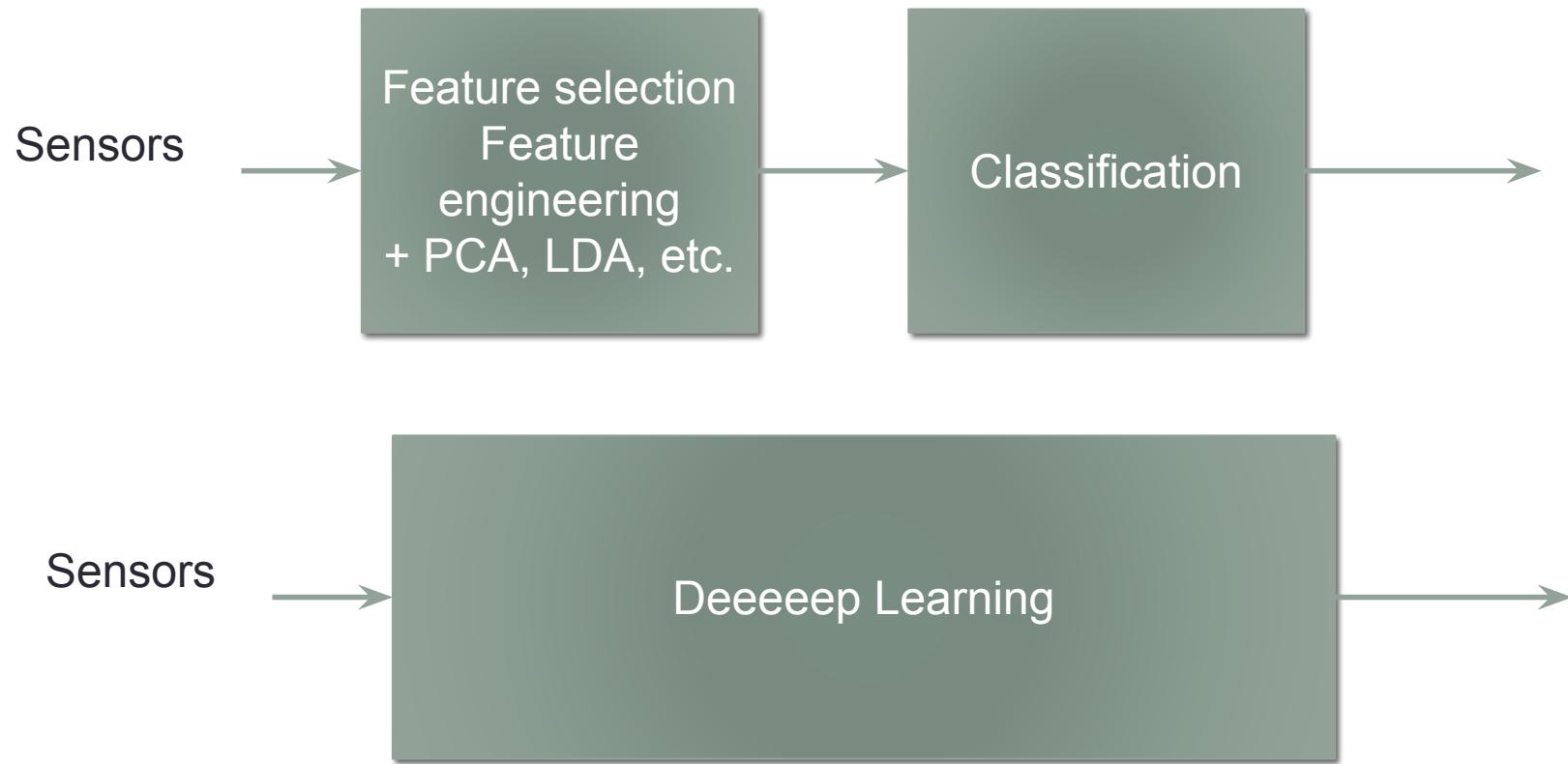


# Wider and deeper networks

- ImageNet task



# Traditional VS Deep learning

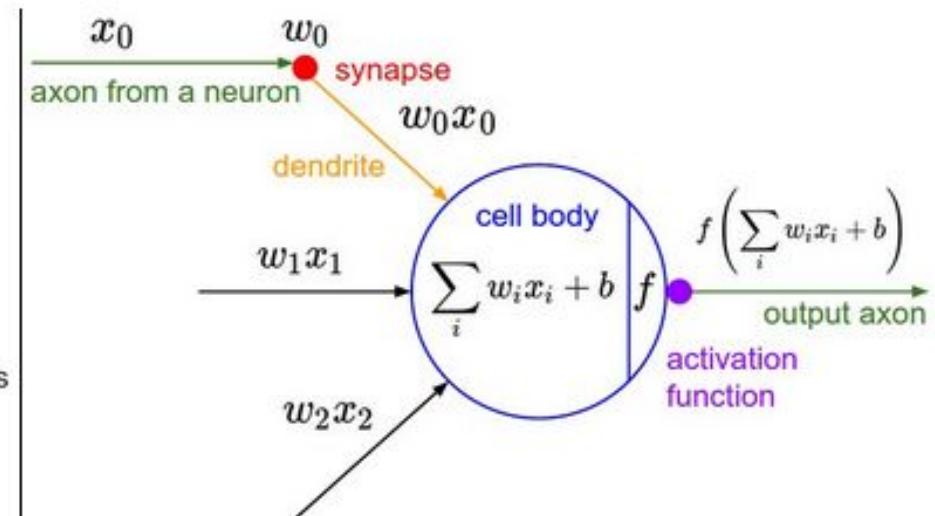
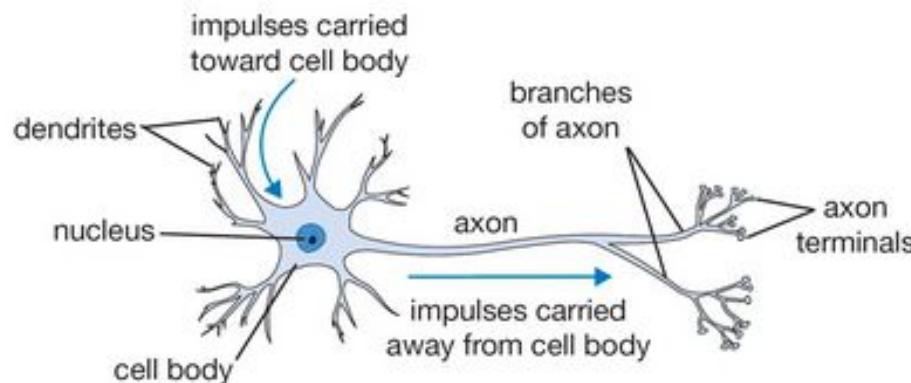


# Neural networks

- Fully connected networks
  - Neuron
  - Non-linearity
  - Softmax layer
- DNN training
  - Loss function
  - SGD and backprop
  - Learning rate
  - Overfitting
- CNN, RNN, LSTM, GRU

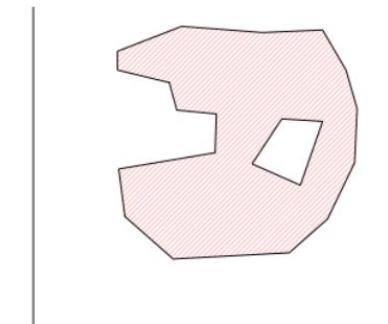
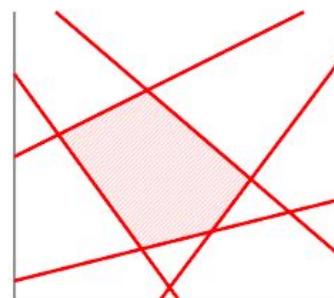
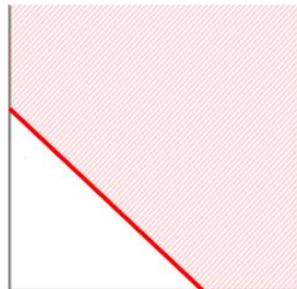
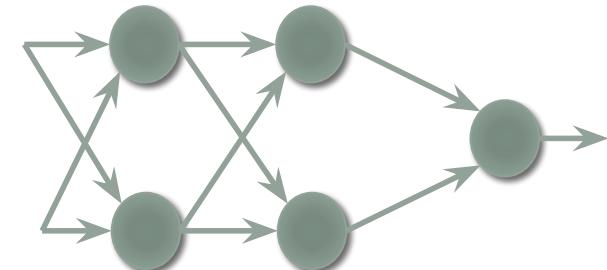
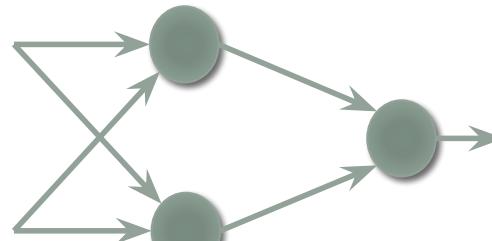
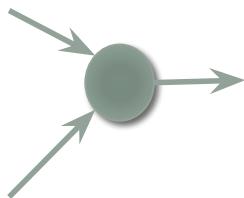
# Fully connected networks

- Many names: feed forward networks or deep neural networks or multilayer perceptron or artificial neural networks
- Composed of multiple neurons



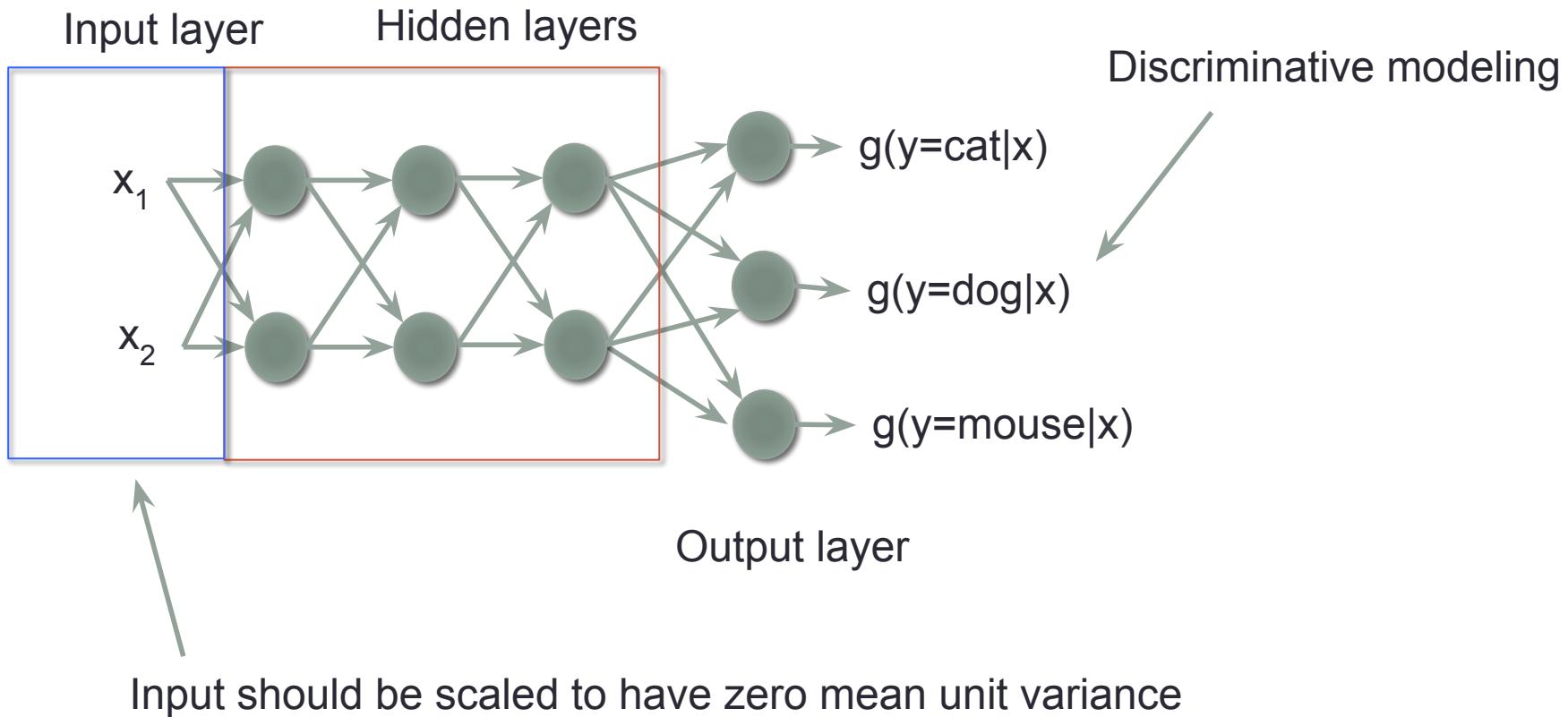
# Combining neurons

- Each neuron splits the feature space with a hyperplane
- Stacking neuron creates more complicated decision boundaries
- More powerful but prone to overfitting



# Terminology

Deep in Deep neural networks means many hidden layers



# Matrices

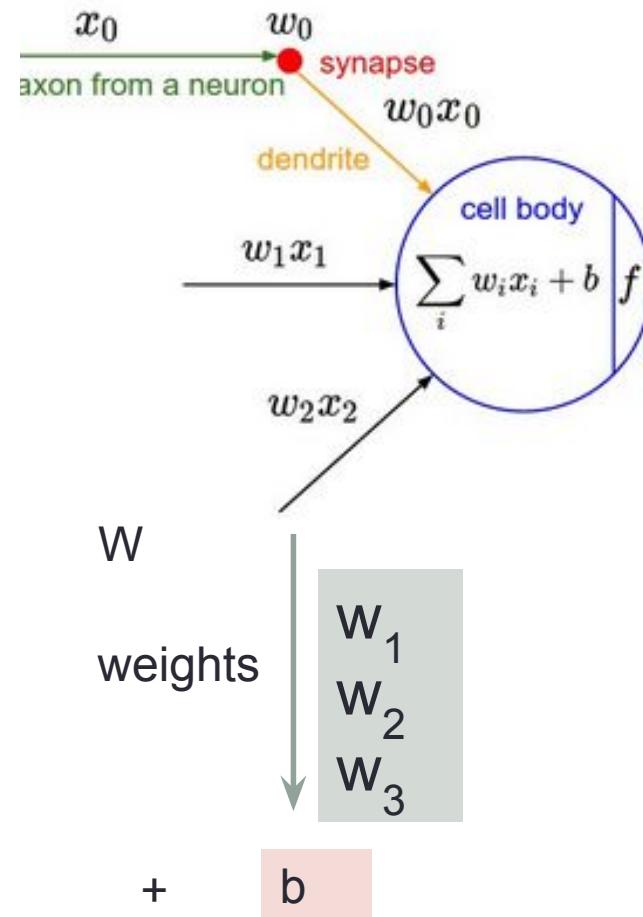
- Inputs

features ↓

$$\begin{matrix} x_1 \\ x_2 \\ x_3 \end{matrix} \quad X$$

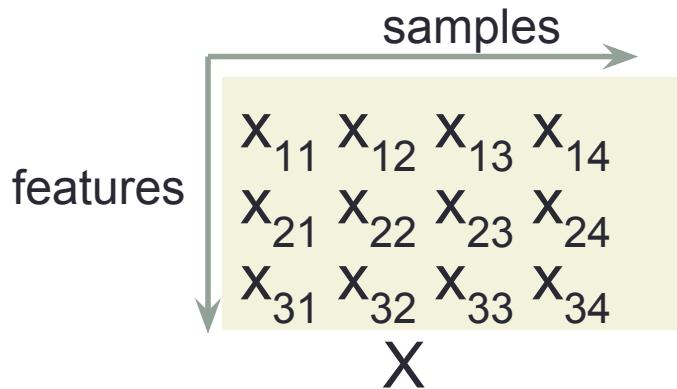
$$W^T X + b$$

$$\begin{matrix} W_1 & W_2 & W_3 \\ x_1 \\ x_2 \\ x_3 \end{matrix}$$



# Matrices

- Inputs



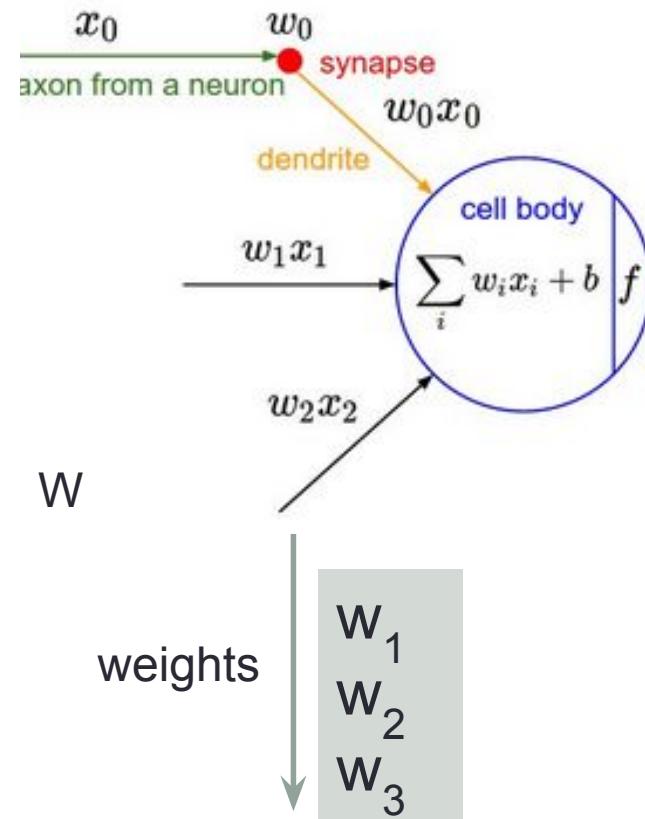
$$W^T X + b$$

$$\begin{matrix} W_1 & W_2 & W_3 \end{matrix}$$

$$\begin{matrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \end{matrix}$$

+

$$b$$



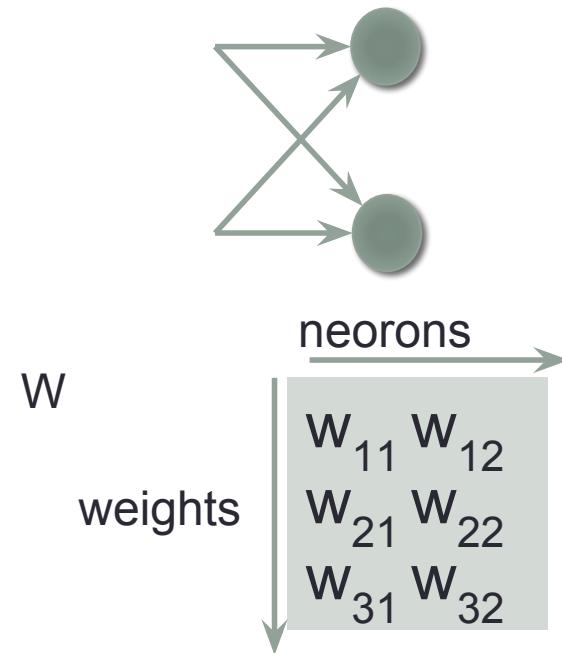
# Matrices

- Inputs

samples →

features ↓	$x_{11}$	$x_{12}$	$x_{13}$	$x_{14}$
	$x_{21}$	$x_{22}$	$x_{23}$	$x_{24}$
	$x_{31}$	$x_{32}$	$x_{33}$	$x_{34}$

$\times$



$$W^T X + b$$

$$\begin{matrix} W_{11} & W_{21} & W_{31} \\ W_{21} & W_{22} & W_{23} \end{matrix}$$

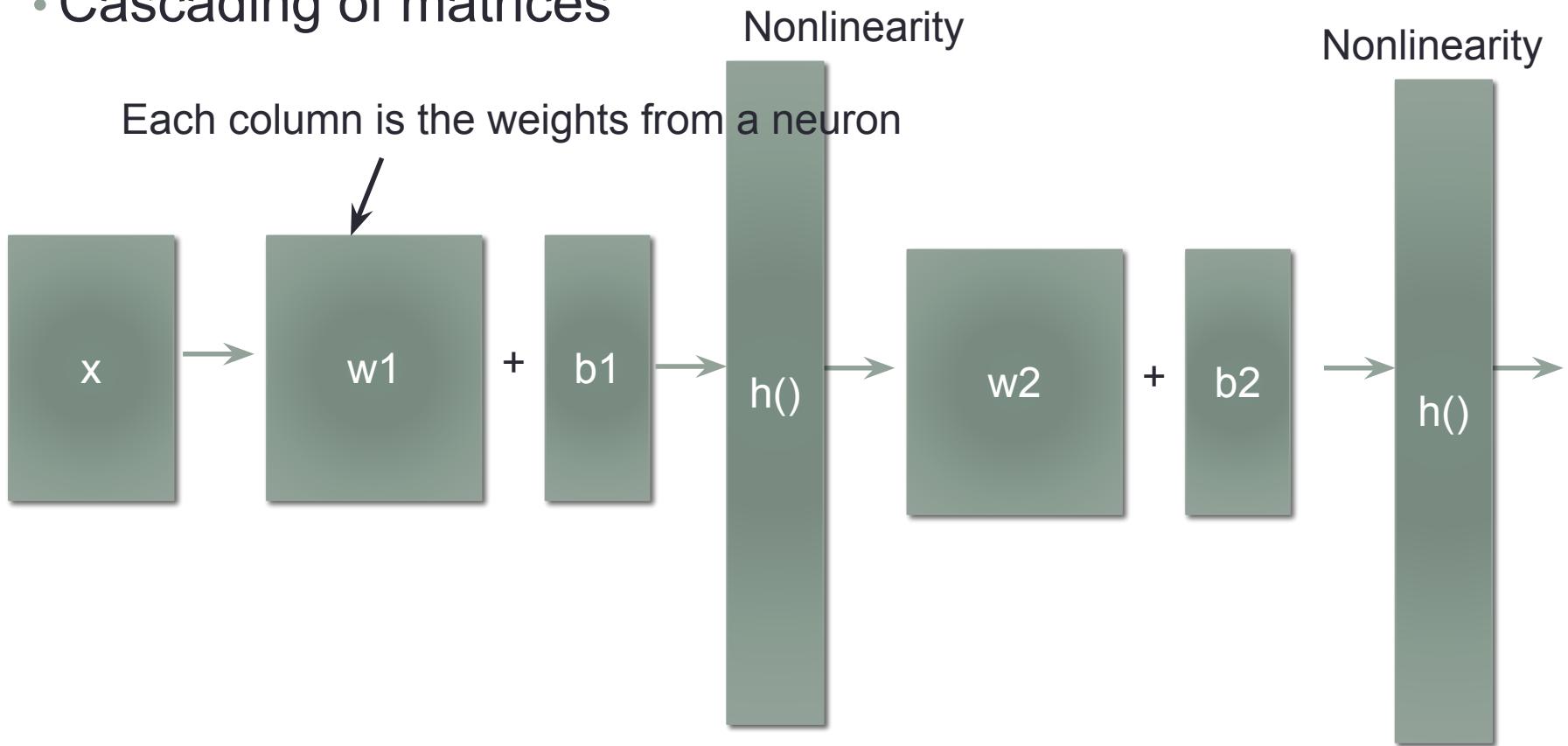
$$\begin{matrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \end{matrix}$$

+

$$\begin{matrix} b_1 \\ b_2 \end{matrix}$$

# More linear algebra

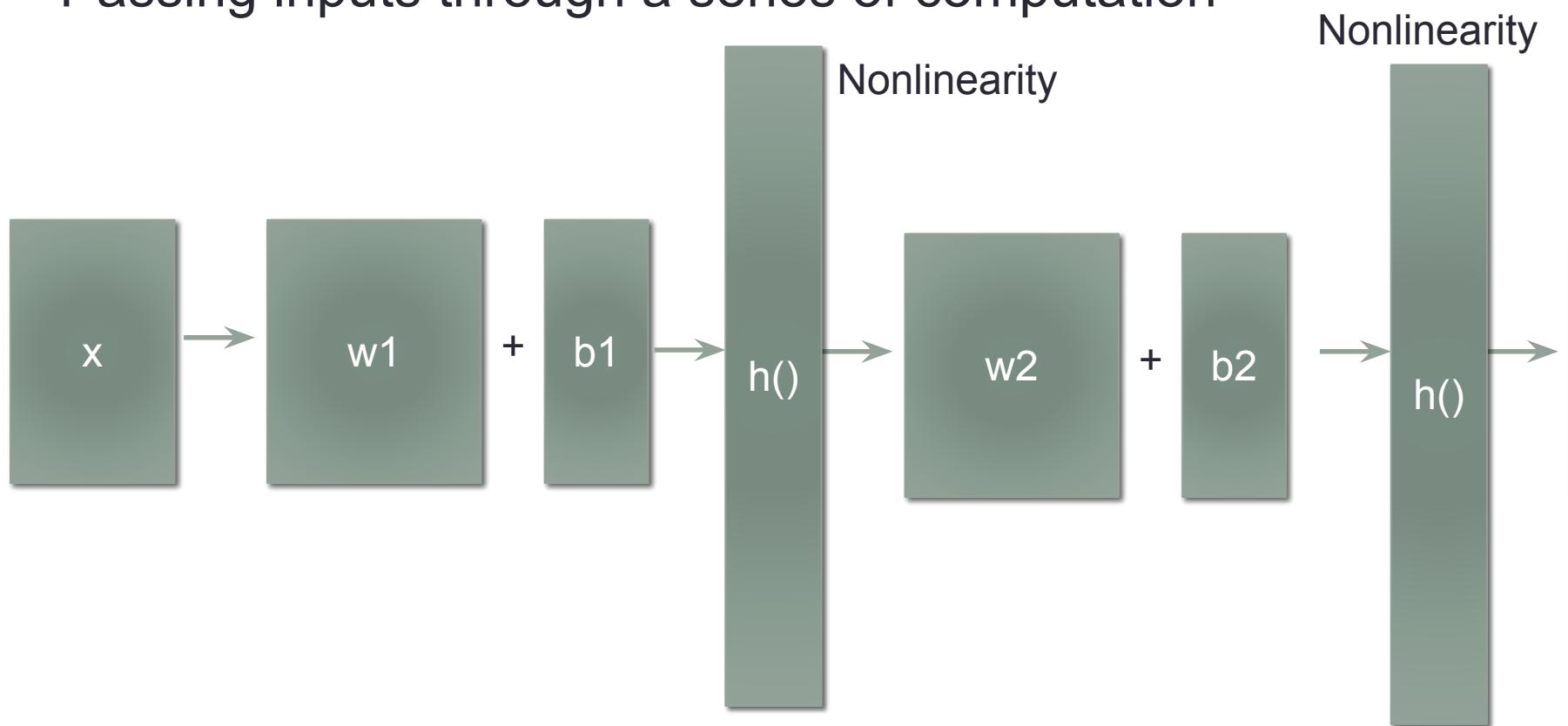
- Cascading of matrices



$$h(W_2^T h(W_1^T X + b_1) + b_2)$$

# Computation graph

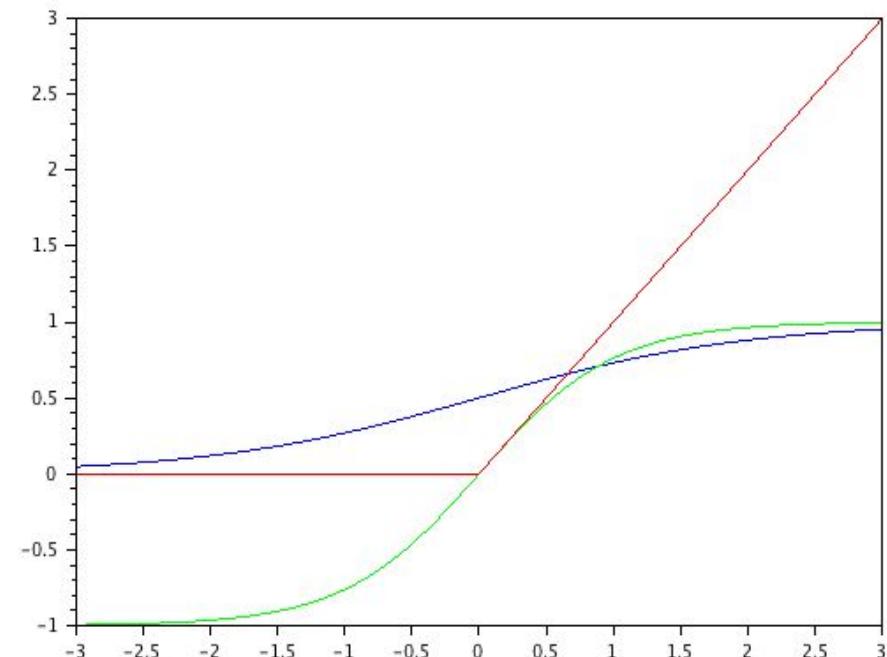
- Passing inputs through a series of computation



$$h(W_2^T h(W_1^T X + b_1) + b_2)$$

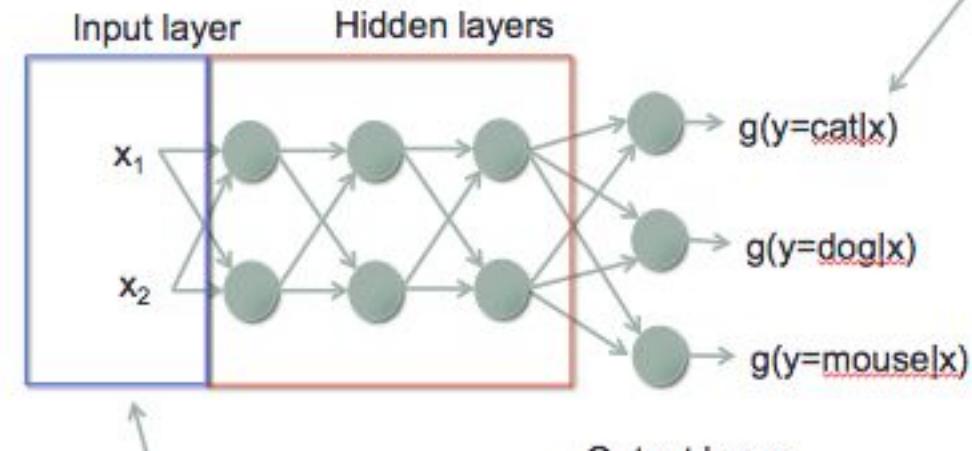
# Non-linearity

- The Non-linearity is important in order to stack neurons
  - If  $F$  is linear, a multi layered network can be collapsed as a single layer (by just multiplying weights together)
- Sigmoid or logistic function
- $\tanh$
- Rectified Linear Unit (ReLU)
- Most popular is ReLU and its variants (Fast to train, and more stable)



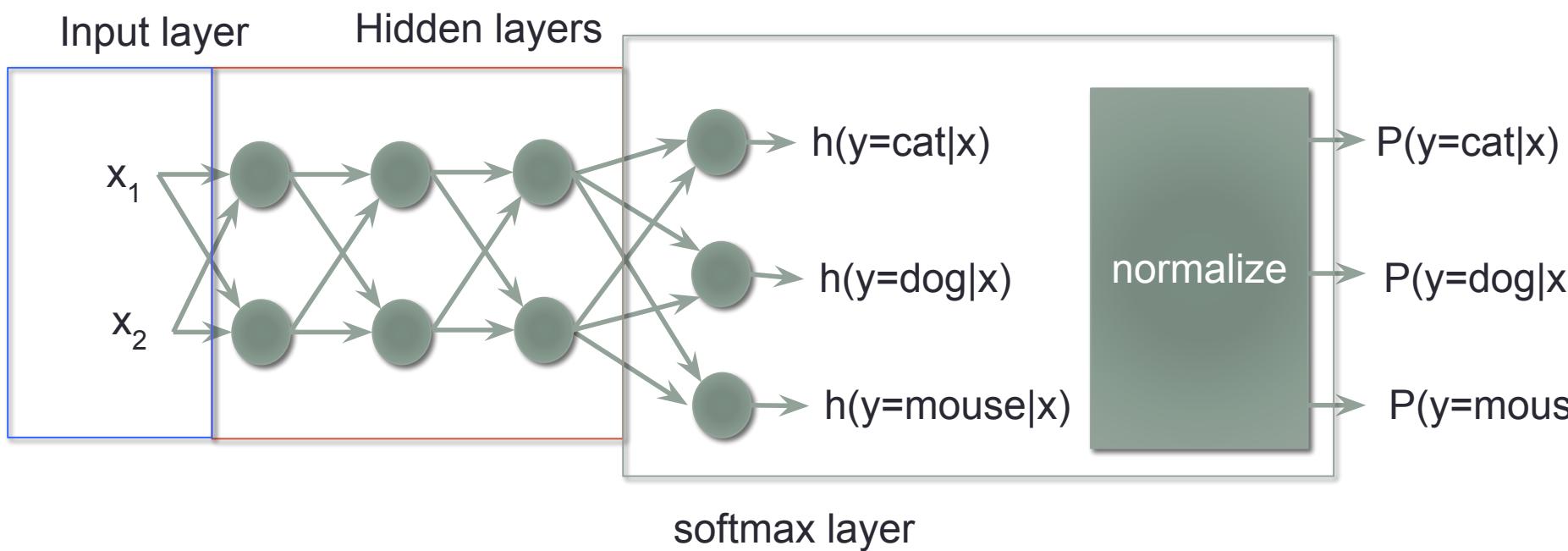
# Output layer – Softmax layer

- We usually want the output to mimic a probability function ( $0 \leq P \leq 1$ , sums to 1)
- Current setup has no such constraint
- The current output should have highest value for the correct class.
  - Value can be positive or negative number
- Takes the exponent
- Add a normalization



# Softmax layer

$$P(y = j|x) = \frac{e^{h(y=j|x)}}{\sum_y e^{h(y|x)}}$$

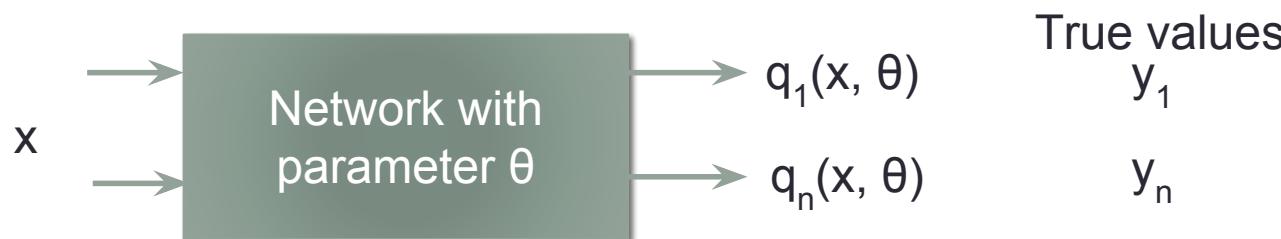


# Neural networks

- Fully connected networks
  - Neuron
  - Non-linearity
  - Softmax layer
- DNN training
  - Loss function
  - SGD and backprop
  - Learning rate
  - Overfitting
- CNN, RNN, LSTM, GRU

# Objective function (Loss function)

- Can be any function that summarizes the performance into a single number
- Cross entropy
- Sum of squared errors



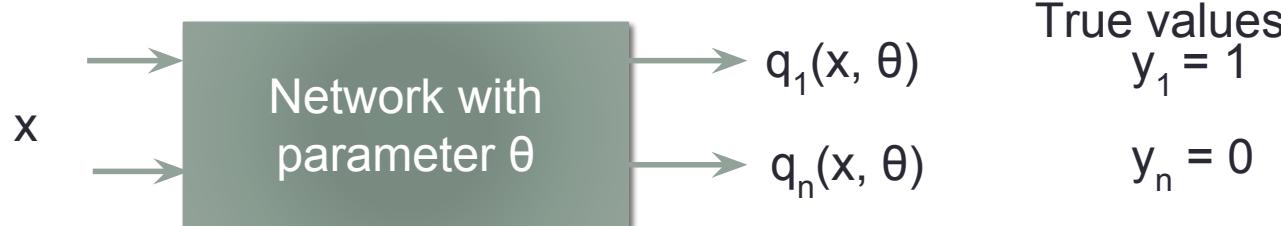
# Cross entropy loss

- Used for softmax outputs (probabilities), or classification tasks

$$L = -\sum_n y_n \log q_n(x, \theta)$$

- Where  $y_n$  is 1 if data  $x$  comes from class  $n$   
0 otherwise

- $L$  only has the term from the correct class
- $L$  is non negative with highest value when the output matches the true values, a “loss” function

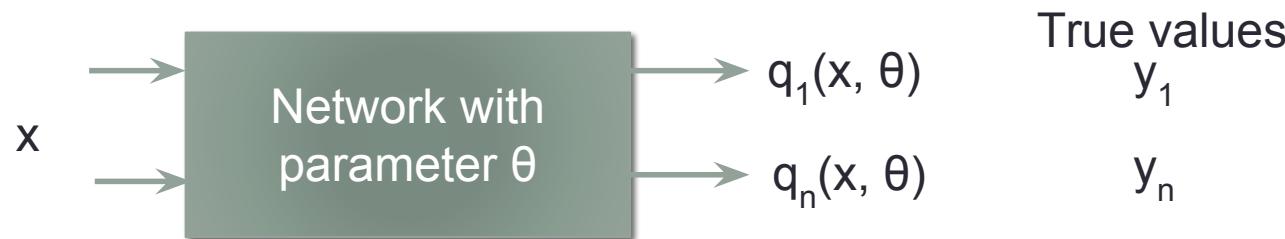


# Sum of squared errors

- Used for any real valued outputs such as regression

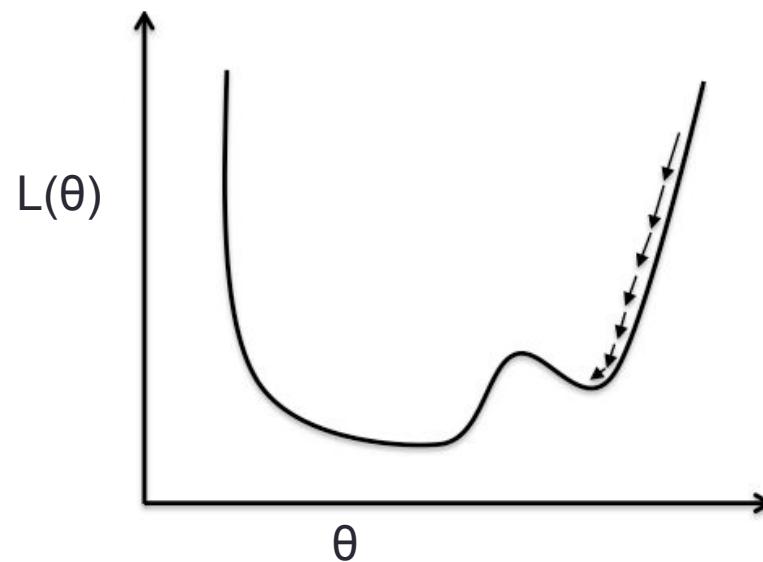
$$L = \frac{1}{2} \sum_n (y_n - q_n(x, \theta))^2$$

- Non negative, the better the lower the loss



# Minimization using gradient descent

- We want to minimize  $L$  with respect to  $\theta$  (weights and biases)
  - Differentiate with respect to  $\theta$
  - Gradients passes through the network by Back Propagation



# Differentiating a neural network model

- We want to minimize loss by gradient descent
- A model is very complex and have many layers! How do we differentiate this!!?



# Back propagation

- Forward pass
  - Pass the value of the input until the end of the network
- Backward pass
  - Compute the gradient starting from the end and passing down gradients using chain rule

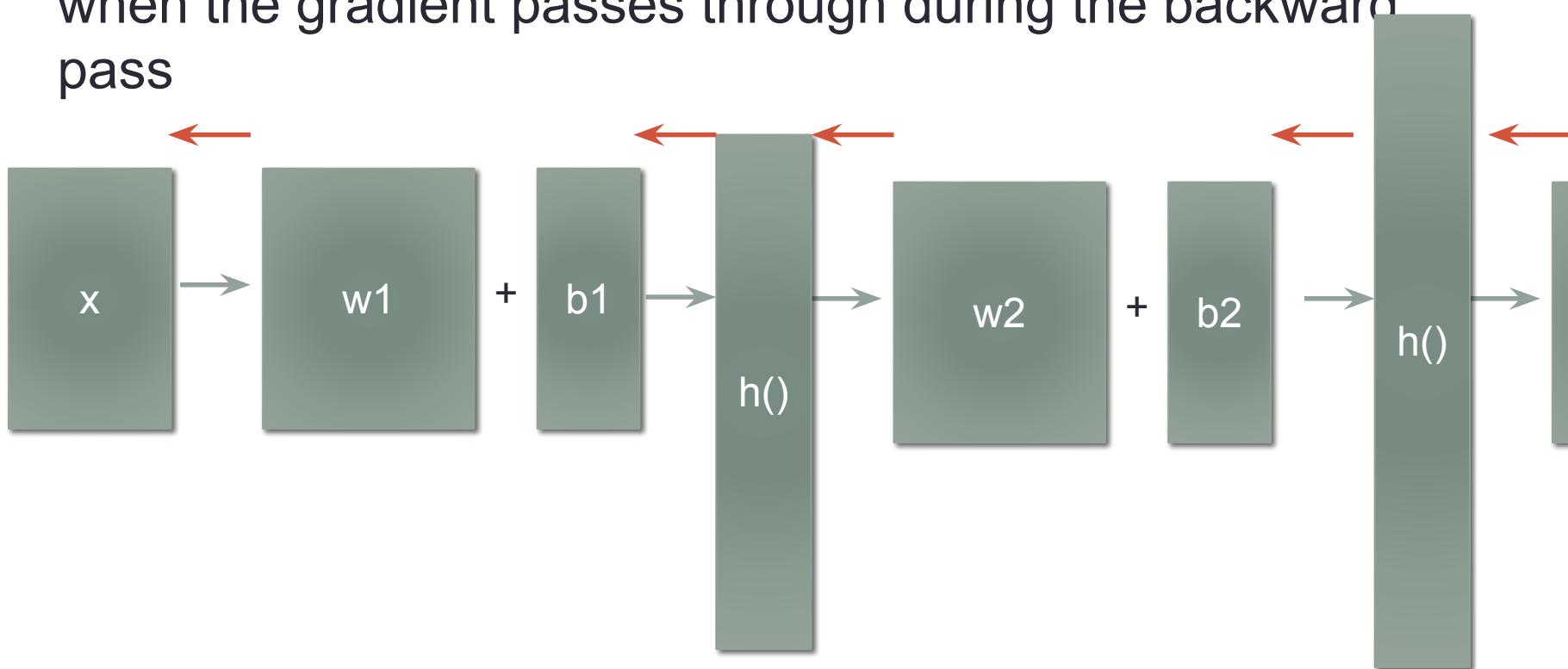
Examples to read

<https://alonalj.github.io/2016/12/10/What-is-Backpropagation/>

<https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>

# Backprop and computation graph

- We can also define what happens to a computing graph when the gradient passes through during the backward pass



This lets us to build any neural networks without having to redo all the derivation as long as we define a forward and backward computation for the block.

# Initialization

- The starting point of your descent
- Important due to local minimas
- Not as important with large networks AND big data (>100 hours for ASR)
- Now usually initialized randomly
  - One strategy

$$W \sim \text{Uniform}(0, \frac{1}{\sqrt{\text{FanIn} + \text{FanOut}}})$$

- For ReLUs
$$w = np.random.randn(n) * \sqrt{2.0/n}$$
- Or use a pre-trained network as initialization

# Stochastic gradient descent (SGD)

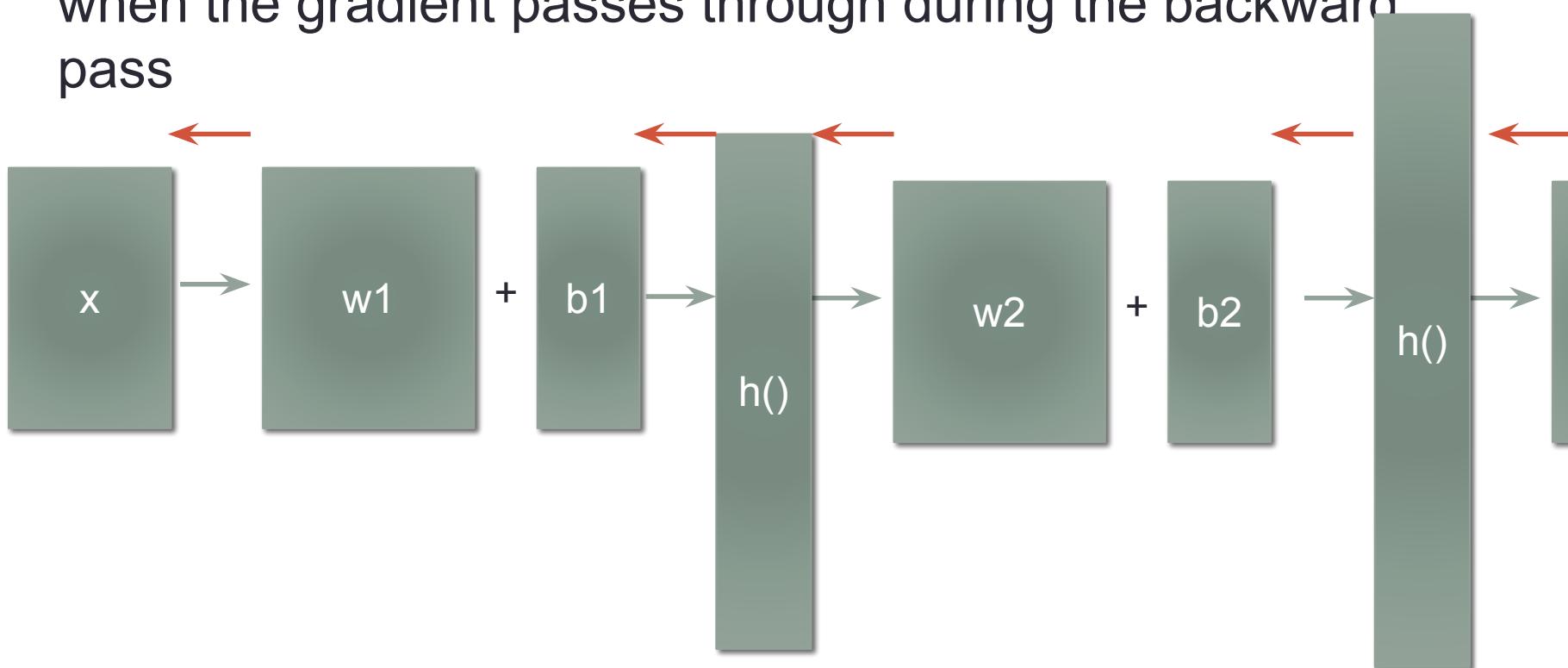
- Consider you have one million training examples
  - Gradient descent computes the objective function of **all** samples, then decide direction of descent
    - Takes too long
  - SGD computes the objective function on **subsets** of samples
    - The subset should not be biased and properly randomized to ensure no correlation between samples
- The subset is called a mini-batch
- Size of the mini-batch determines the training speed and accuracy
  - Usually somewhere between 32-1024 samples per mini-batch
- Definition: 1 batch vs 1 epoch

# Back propagation

- Regularization terms only appears at the particular weights when doing the derivative
- What about cross entropy?

# Backprop and computation graph

- We can also define what happens to a computing graph when the gradient passes through during the backward pass



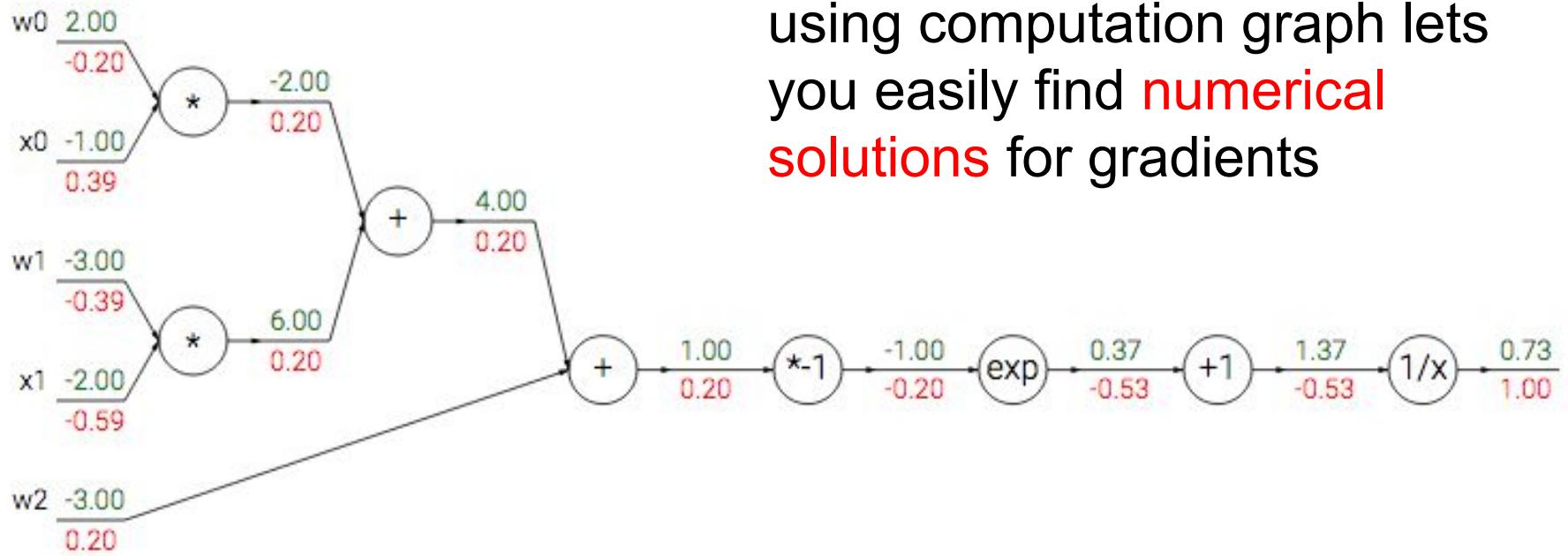
This lets us to build any neural networks without having to redo all the derivation as long as we define a forward and backward computation for the block.

# Numerical gradient flow

- Let's find the gradient of

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

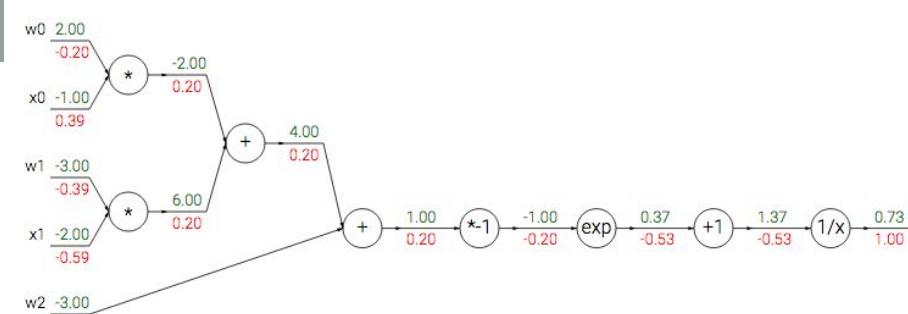
Computation graph



Doing backprop (chain rule) by using computation graph lets you easily find **numerical solutions** for gradients

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

- $w = [0, -3, -3]$
- $x = [-1, -2]$
- $t_0 = w[0]*x[0]$
- $t_1 = w[1]*x[1]$
- $t_{01} = t_0 + t_1$
- $t_{012} = t_{01} + w[2]$
- $n_t = -t_{012}$
- $e = \exp(n_t)$
- $\text{denom} = e + 1$
- $f = 1/\text{denom}$

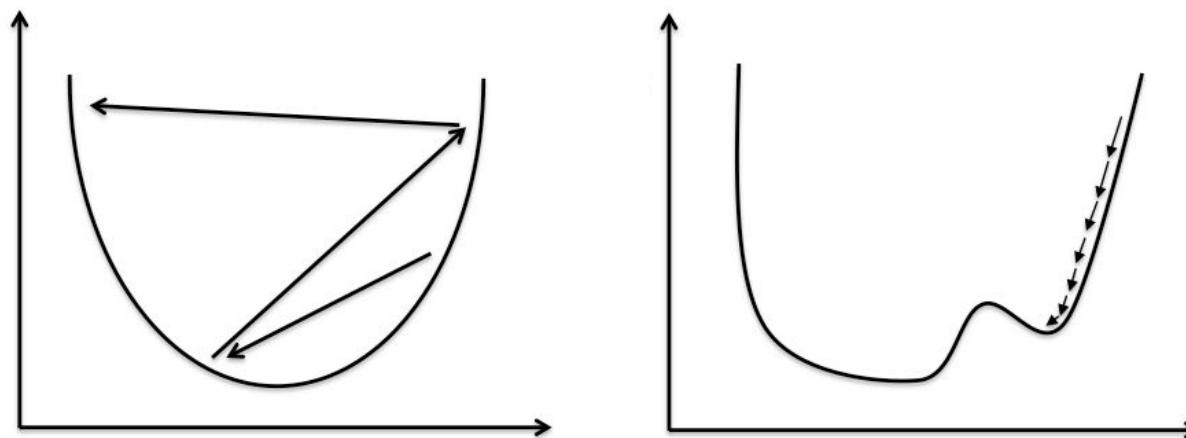


- $\text{ddenom} = -1/\text{denom}/\text{denom}$
- $de = 1 * \text{ddenom}$
- $dn_t = \exp(n_t) * de$
- $dt_{012} = -dn_t$
- $dw_2 = 1 * dt_{012}$
- $dt_{01} = 1 * dt_{012}$
- $dt_0 = 1 * dt_{01}$
- $dt_1 = 1 * dt_{01}$
- $dw_1 = x[1]dt_1$
- $dx_1 = w[1]dt_1$
- $dw_0 = x[0]dt_0$ ;  $dx_0 = w[0]dt_0$

Perform backward pass in reverse order. No need to explicitly find overall derivative

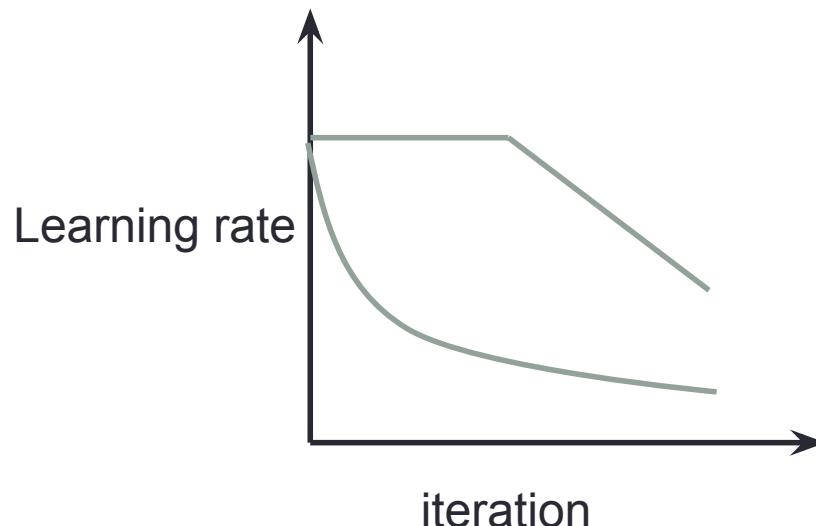
# Learning rate

- How fast to go along the gradient direction is controlled by the learning rate
- Too large models diverge
- Too small the model get stuck in local minimas and takes too long to train



# Learning rate scheduling

- Usually starts with a large learning rate then gets smaller later
- Depends on your task
- Automatic ways to adjust the learning rate : Adagrad, Adam, etc. (still need scheduling still)

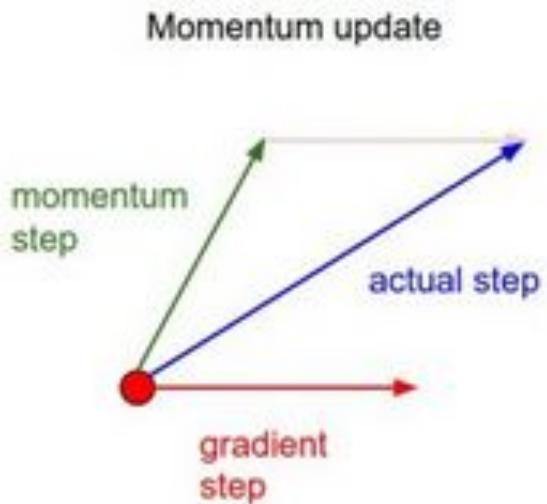


# Learning rate strategies (annealing)

- Step decay: reduce learning rate by  $x$  after  $y$  epochs
- New bob method: half learning rate every time the validation error goes up. Only plausible in larger tasks
- Exponential decay: multiplies the learning rate by  $\exp(-\text{rate} * \text{epoch number})$
- ADAM + variants
  - Most popular for its ease of use. Still need some decay for best performance
  - People find simple SGD with momentum and decay to perform better (with proper tuning)
    - Several research trying to fix this

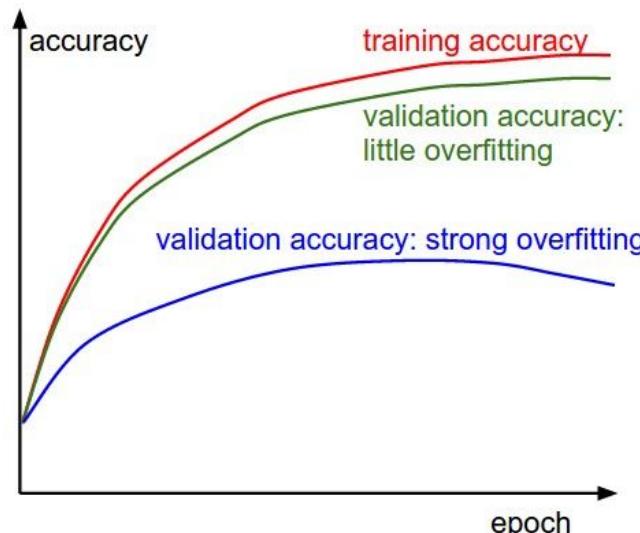
# Momentum

- SGD updates can be noisy because of mini-batch sampling
- Momentum lets the optimization remembers the general direction
- Simple GD update is of the form
  - $W_{\text{new}} = W_{\text{old}} - \text{learning rate} * \text{gradient}$
- Momentum adds “**velocity**”
  - $V_{\text{new}} = \text{momentum rate} * V_{\text{old}} + \text{learning rate} * \text{gradient}$
  - $W_{\text{new}} = W_{\text{old}} - V_{\text{new}}$
- Momentum helps for faster convergence



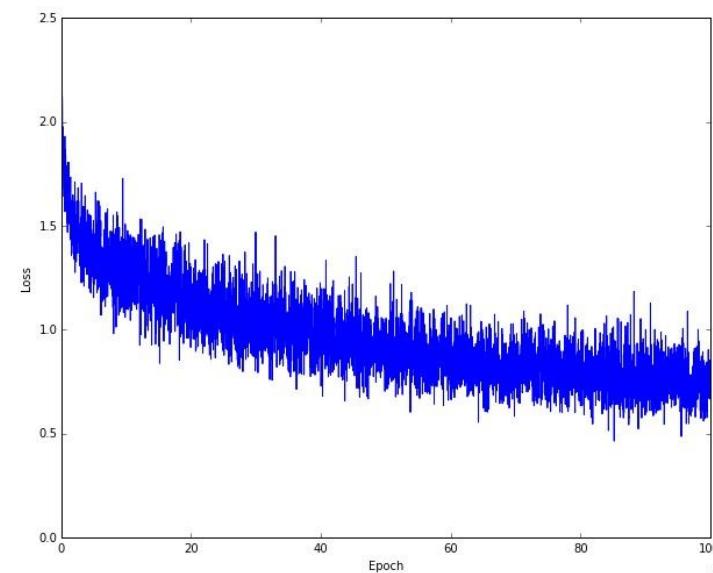
# Overfitting

- You can keep doing back propagation forever!
- The training loss will always go down
- But it overfits
- Need to monitor performance on a held out set
- Stop or decrease learning rate when overfit happens



# Monitoring performance

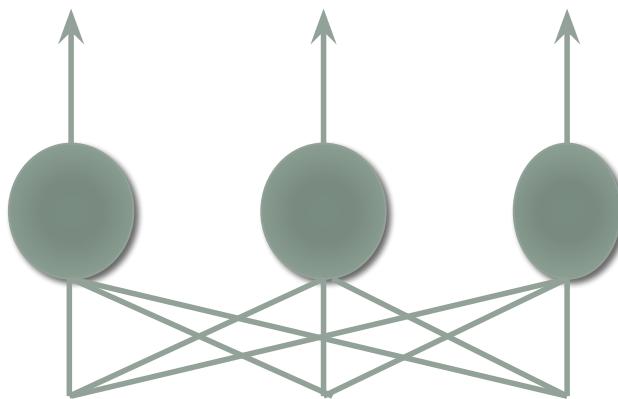
- Monitor performance on a dev/validation set
  - This is NOT the test set
- Can monitor many criterions
  - Loss function
  - Classification accuracy
- Sometimes these disagree
- Actual performance can be noisy, need to see the trend



# Reducing overfitting - dropout

- A regularization technique for reducing overfitting
- Randomly turn off different subset of neurons during training
  - Network no longer depend on any particular neuron
  - Force the model to have redundancy – robust to any corruption in input data
  - A form of performing model averaging (ensemble of experts)
- Now a standard technique

# Dropout visualized

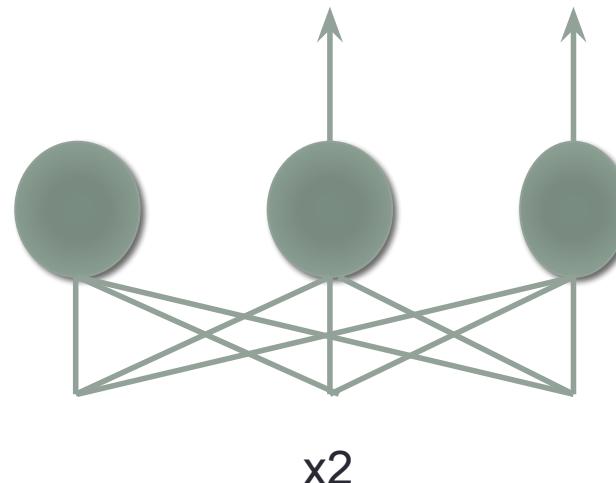
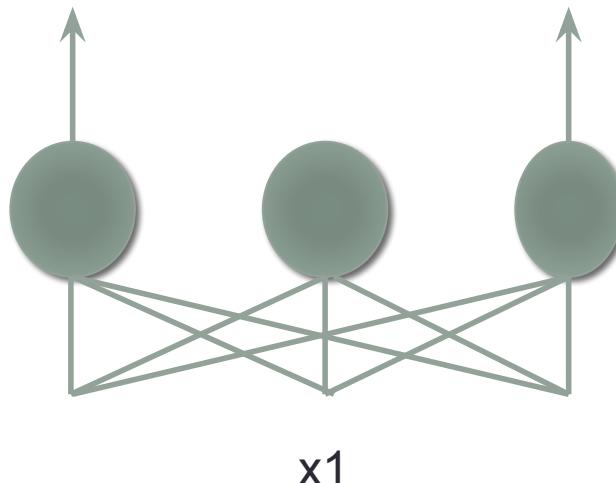


Model

# Dropout training time

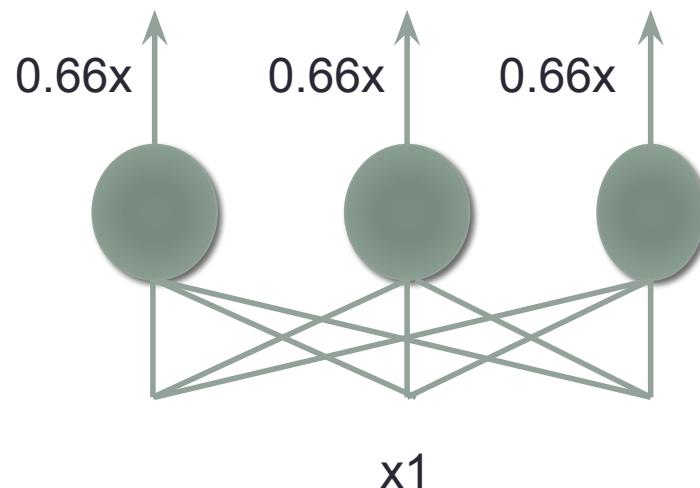
Drop out rate 0.33

Randomly removed outputs  
for each training sample

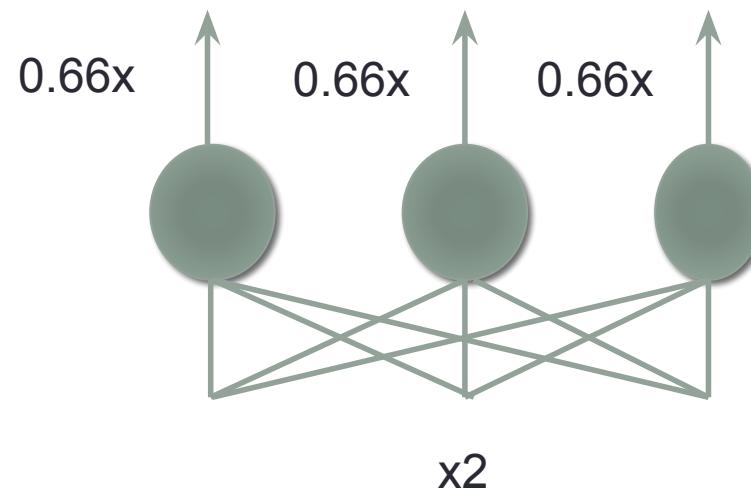


# Dropout test time

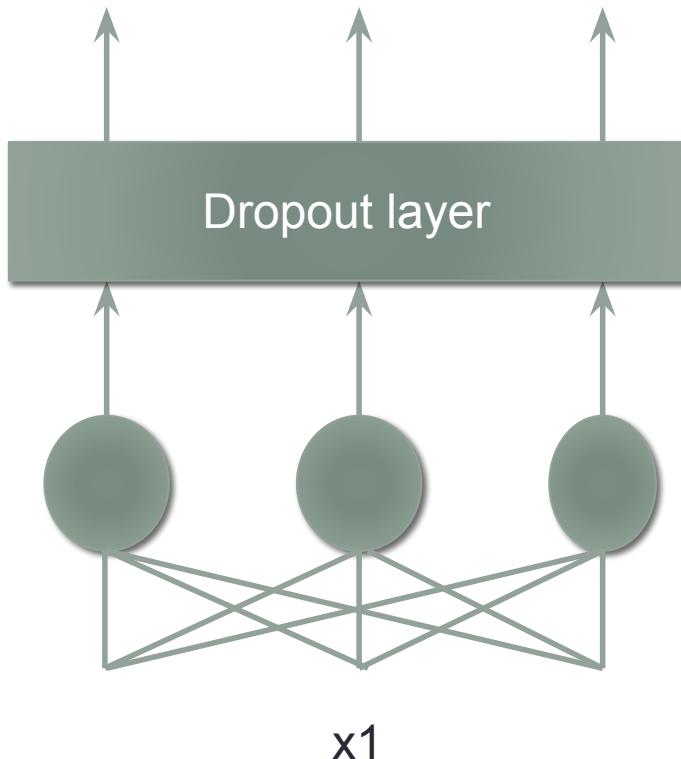
Dropout rate 0.33



Scale outputs so the output contribution is around the same



# Dropout implementation



## Dropout layer

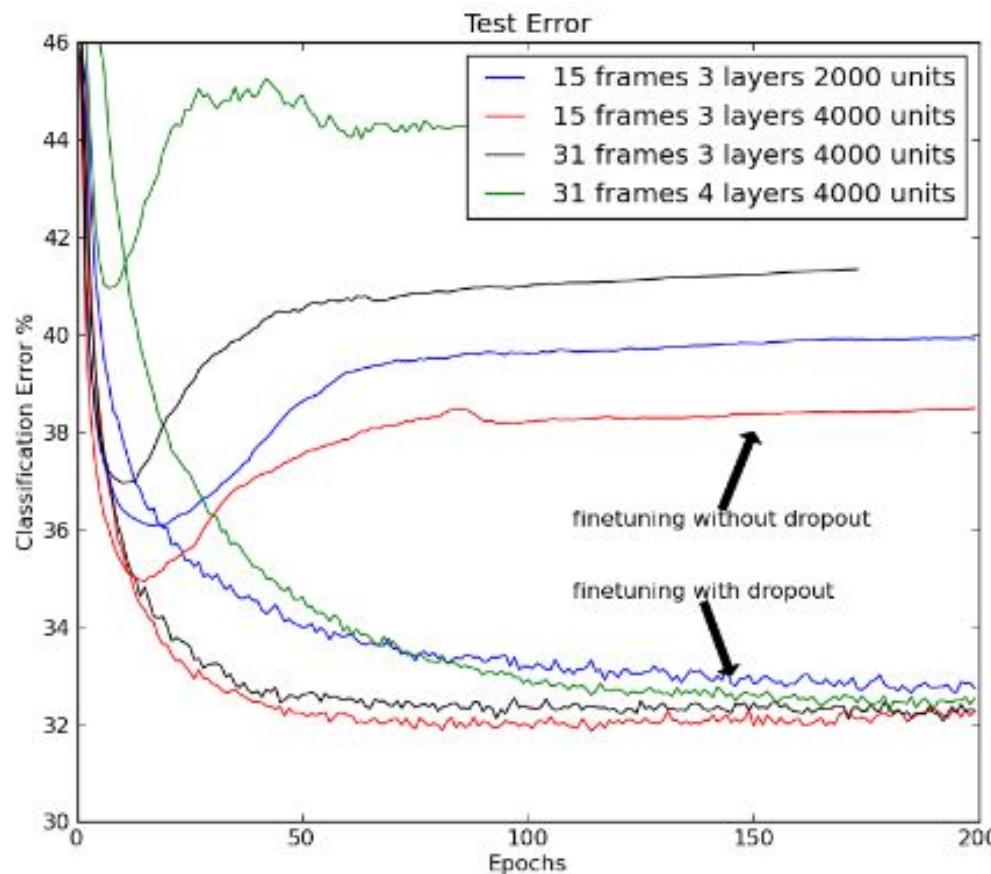
Just another layer that drops inputs

## Inverted dropout

A variant of dropout that scales the dropout at training time, so that you don't have to scale at test time.

# Dropout on TIMIT

- A phoneme recognition task



# Where to dropout

- You can add dropout in the hidden layers (0-0.5)
- Or input layers (0-0.2 is typical)
  - “Noising” the inputs, data augmentation
- Dropout in NLP
  - Usually works better than batchnorms for NLP
  - Drop full words at the embedding
  - Recurrent dropout <http://arxiv.org/abs/1512.05287>

# Vanishing/Exploding gradient

- Backprop introduces many multiplications down chain
- The gradient value gets smaller and smaller
  - The deeper the network the smaller the gradient in the lower layers
  - Lower layers changes too slowly (or not at all)
  - Hard to train very deep networks (>6 layers)
- The opposite can also be true. The gradient explodes from repeated multiplication
  - Put a maximum value for the gradient (Gradient clipping)

# Neural networks

- Fully connected networks
  - Neuron
  - Non-linearity
  - Softmax layer
- DNN training
  - Loss function
  - SGD and backprop
  - Learning rate
  - Overfitting
- CNN, RNN, LSTM, GRU

# Convolutional Neural Networks (CNNs)

- Consider an image of a cat. DNNs need different neurons to learn every possible location a cat can be

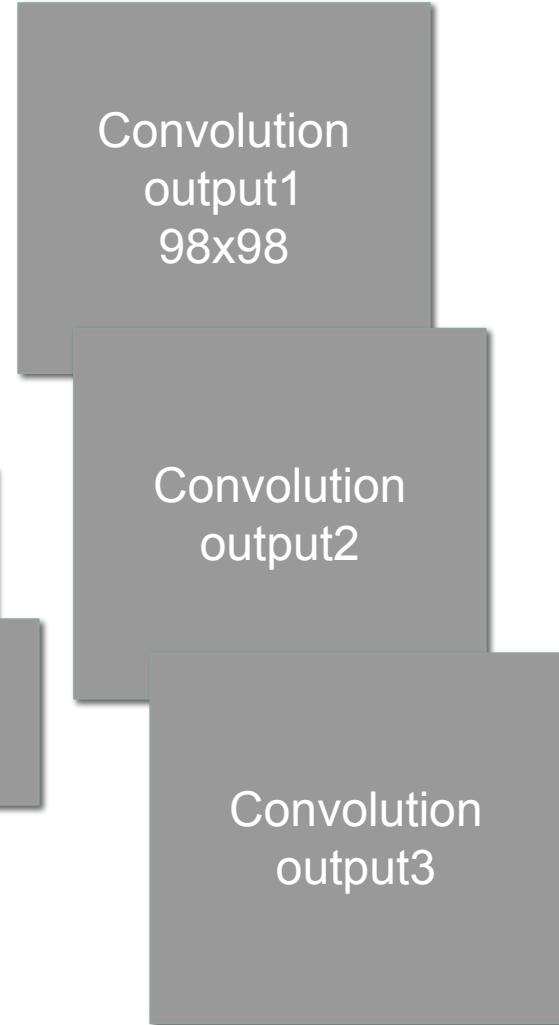
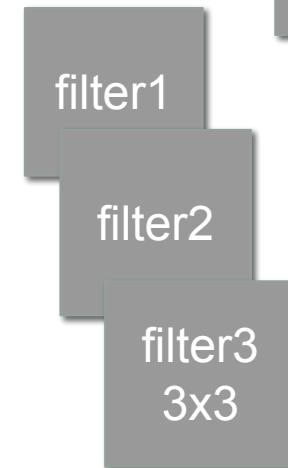
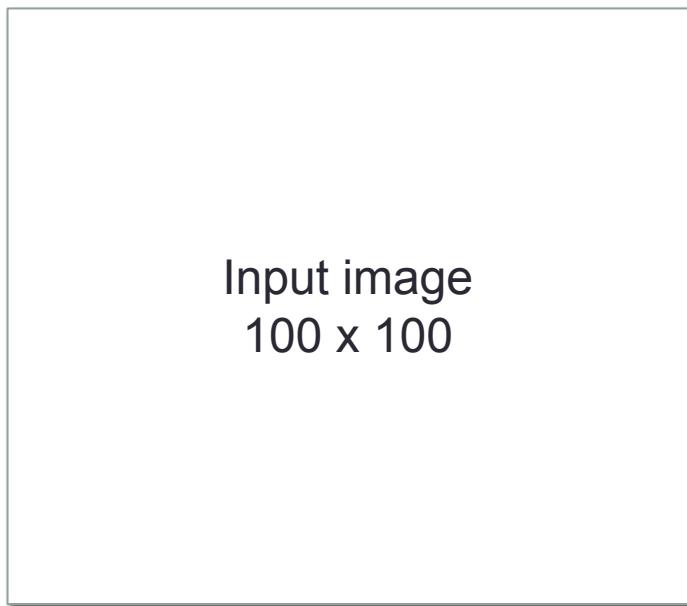


- Can we use the same parameters to learn that a cat exists regardless of location?
- 2 parts: convolutional layer and pooling layer

# Convolutional filters

Multiply inputs with filter values

Output one feature map per filter



# Convolutional filters

0	1	-1
1	0	1
1	2	0
1	2	3
4	5	6
7	8	9

$$1*2 + -1*3 + 1*4 + 1*6 + 1*7 + 2*8 = 32$$

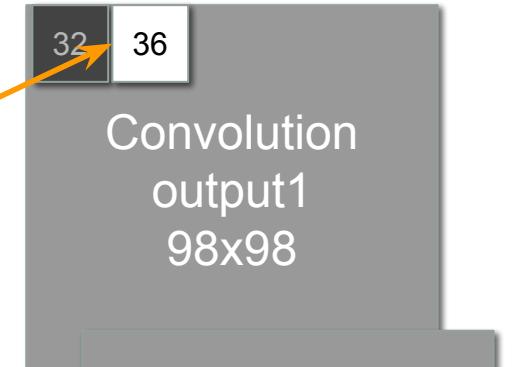
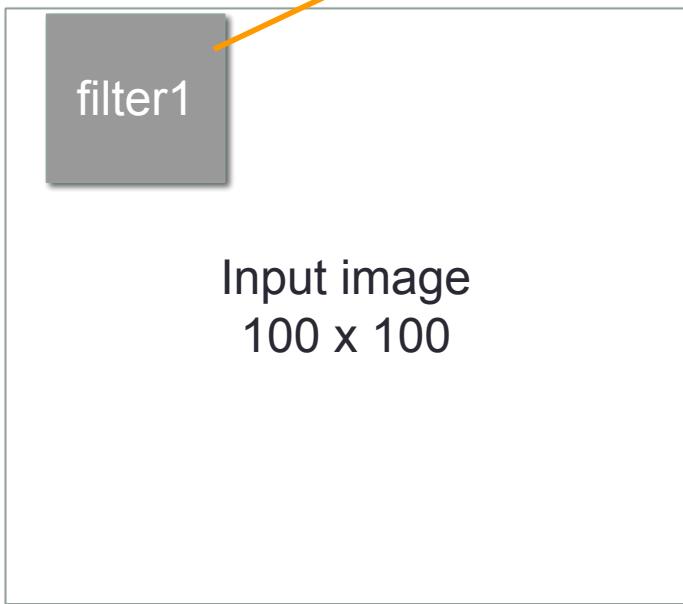


# Convolutional filters

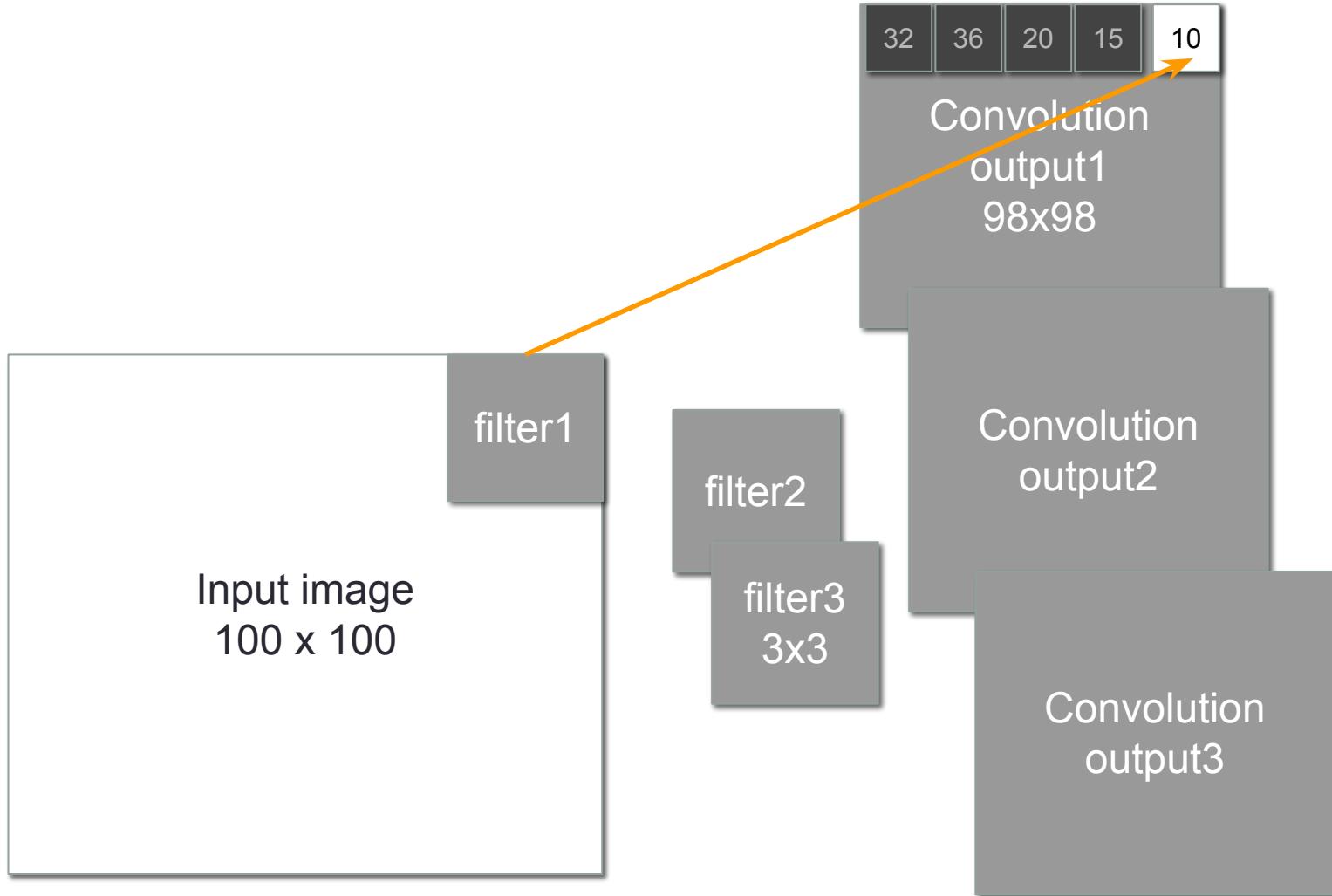
Stride of 1

0	1	-1
1	0	1
1	2	0
2	3	1
5	6	3
8	9	8

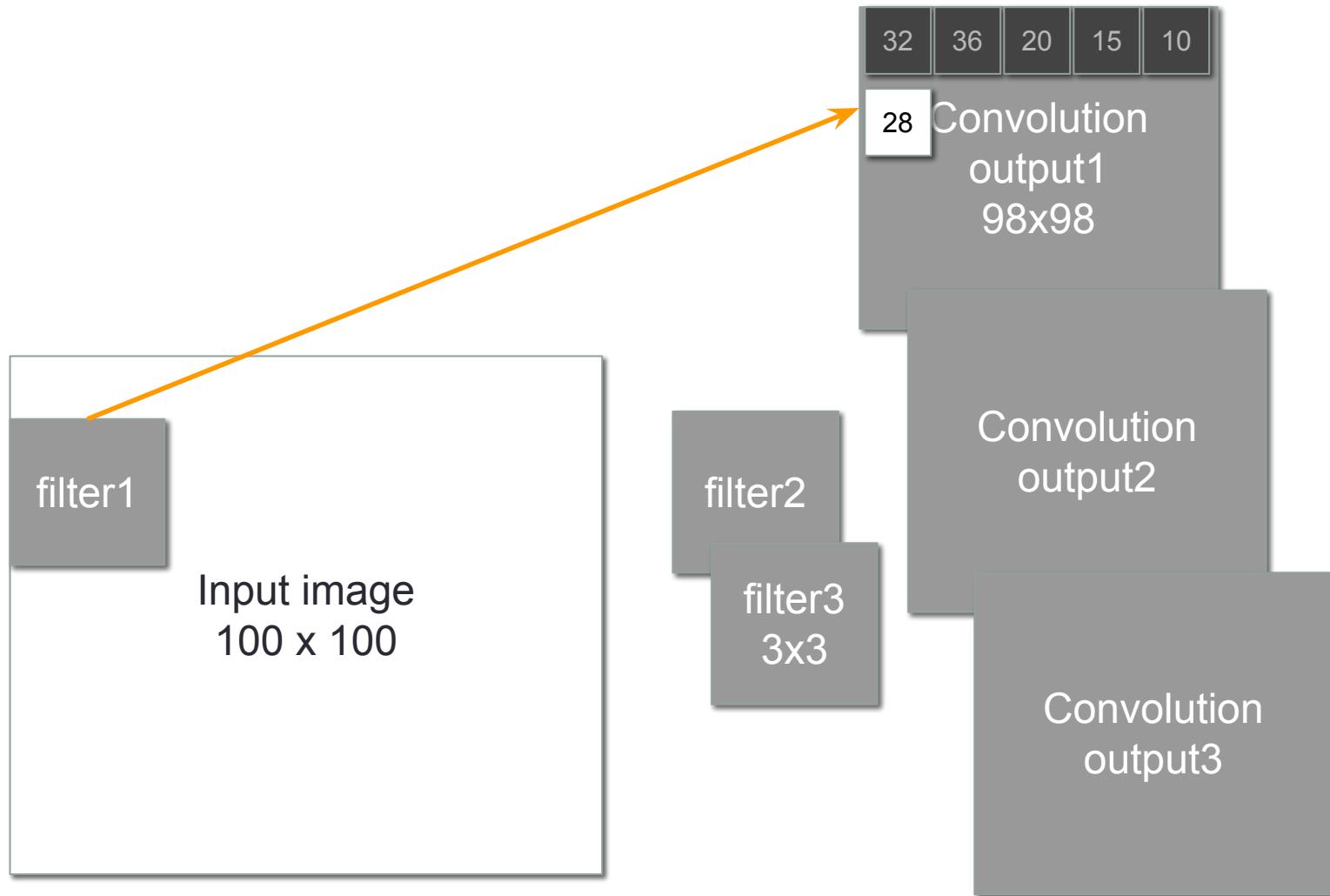
$$1*3 + -1*1 + 1*5 + 1*3 + 1*8 + 2*9 = 36$$



# Convolutional filters

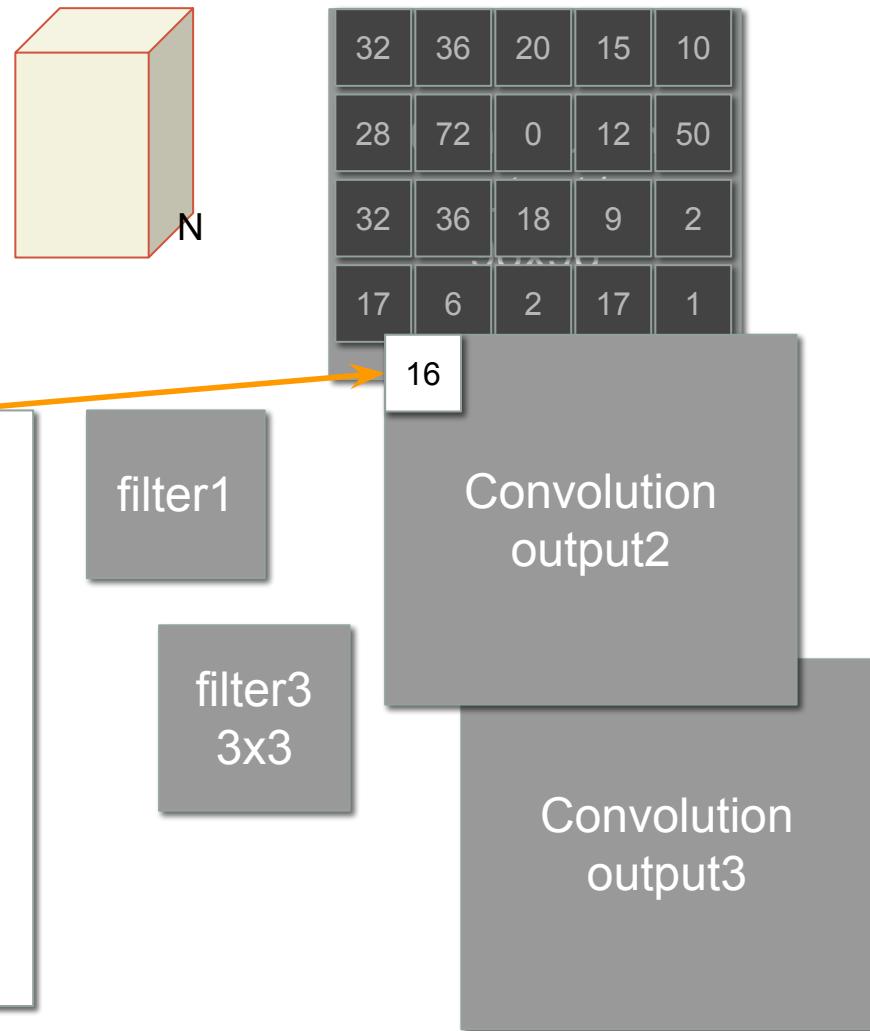


# Convolutional filters



# Convolutional filters

N filters means N feature maps  
You get a 3 dimensional output

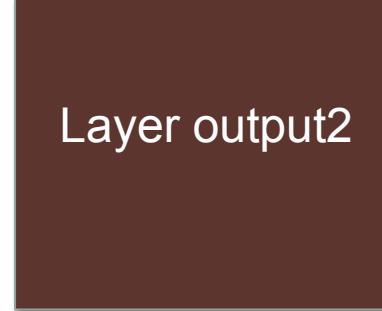


# Pooling/subsampling

Reduce dimension of the feature maps



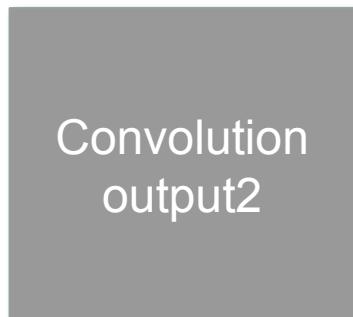
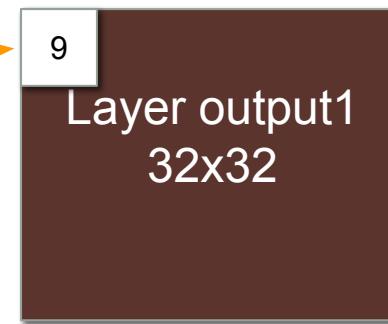
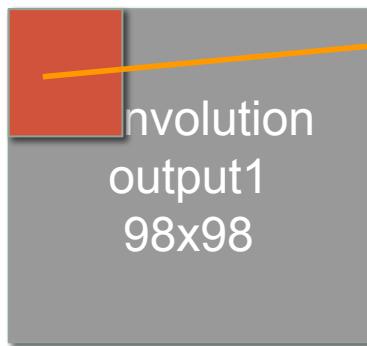
3x3 Max filter  
with no overlap



# Pooling/subsampling

1	2	3
4	5	6
7	8	9

Max = 9

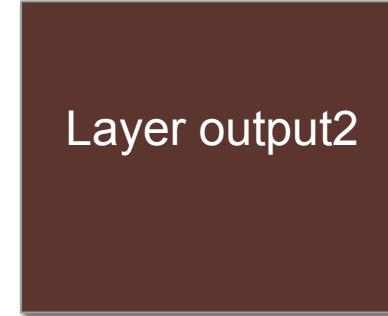
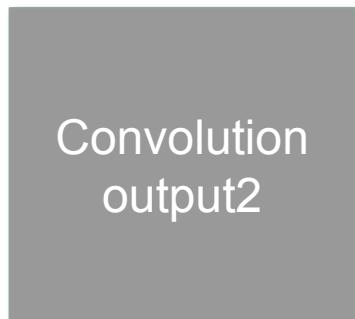
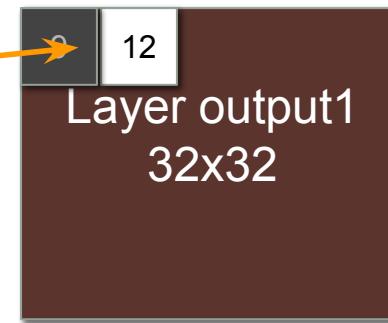
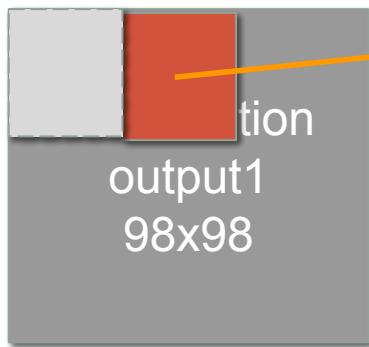


# Pooling/subsampling

5	2	1
5	7	1
9	5	12

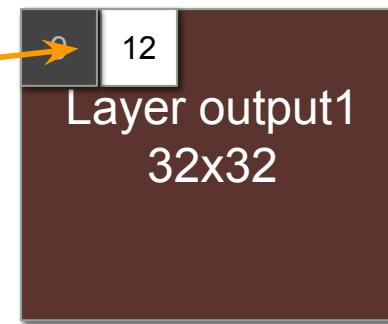
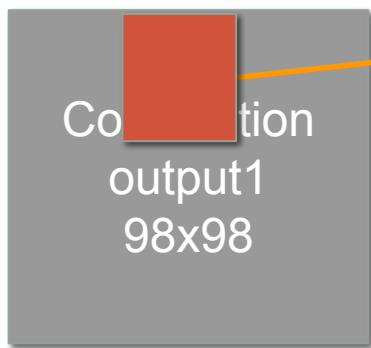
Max = 12

Stride = 3



# Pooling/subsampling

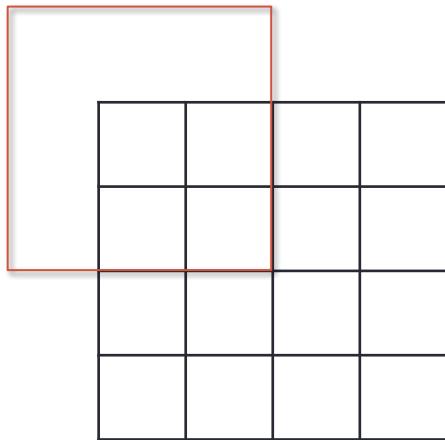
Can use other functions besides max  
Example, average



# Convolution puzzle

5 filters 3x3 filter pad, stride 1, pad 1

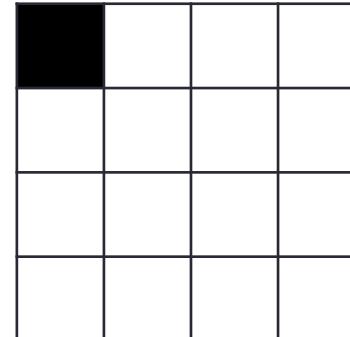
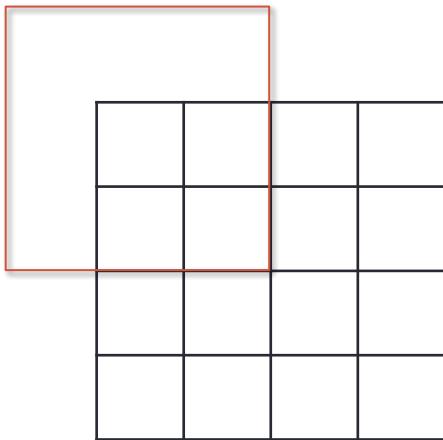
What is the output size?



# Convolution puzzle

5 filters 3x3 filter pad, stride 1, pad 1

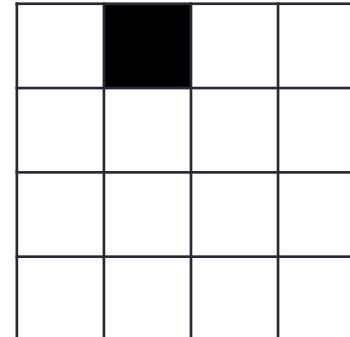
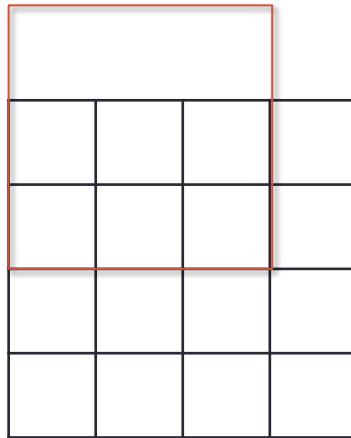
What is the output size?



# Convolution puzzle

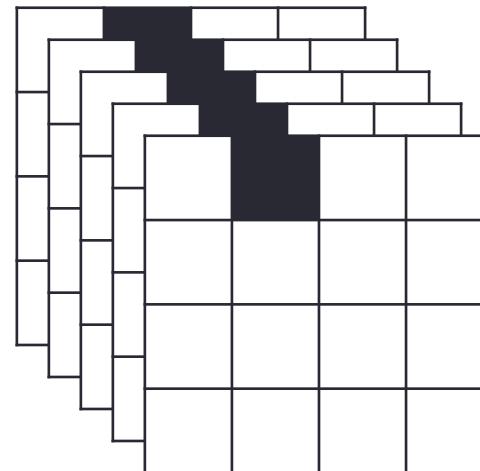
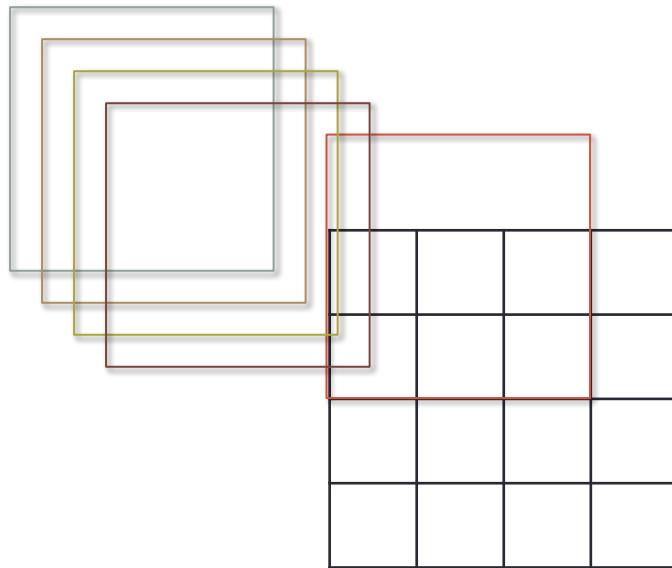
5 filters 3x3 filter pad, stride 1, pad 1

What is the output size?



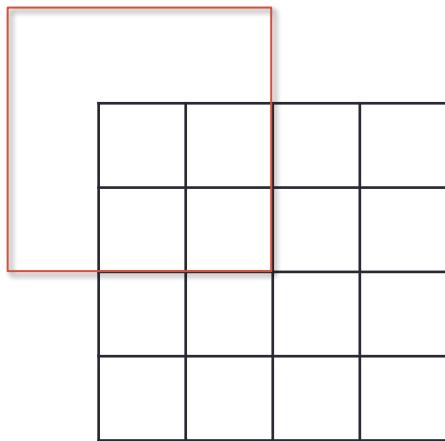
# Convolution puzzle

5 filters 3x3 filter pad, stride 1, pad 1



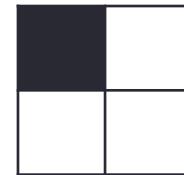
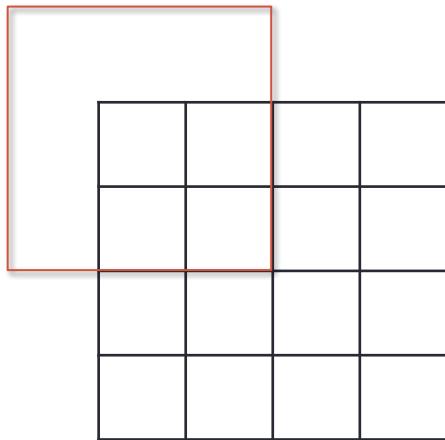
# Convolution puzzle

3x3 filter pad, stride 2, pad 1



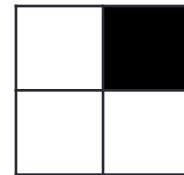
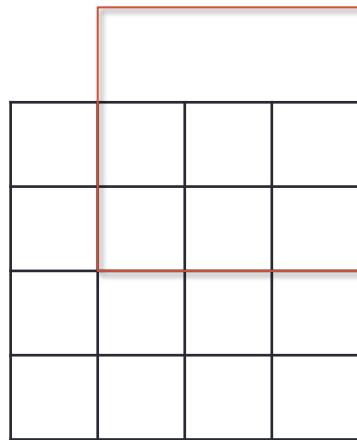
# Convolution puzzle

3x3 filter pad, stride 2, pad 1



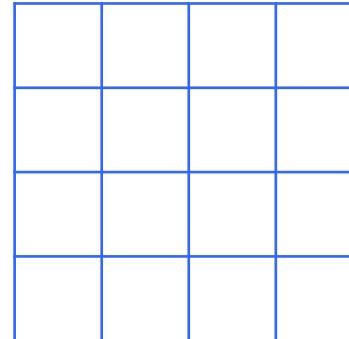
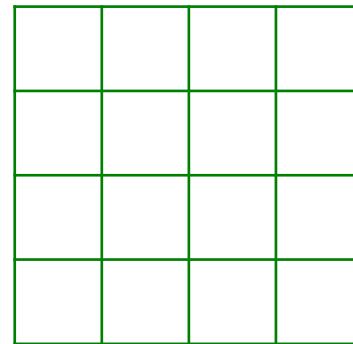
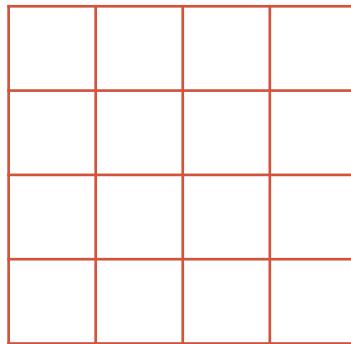
# Convolution puzzle

3x3 filter pad, stride 2, pad 1



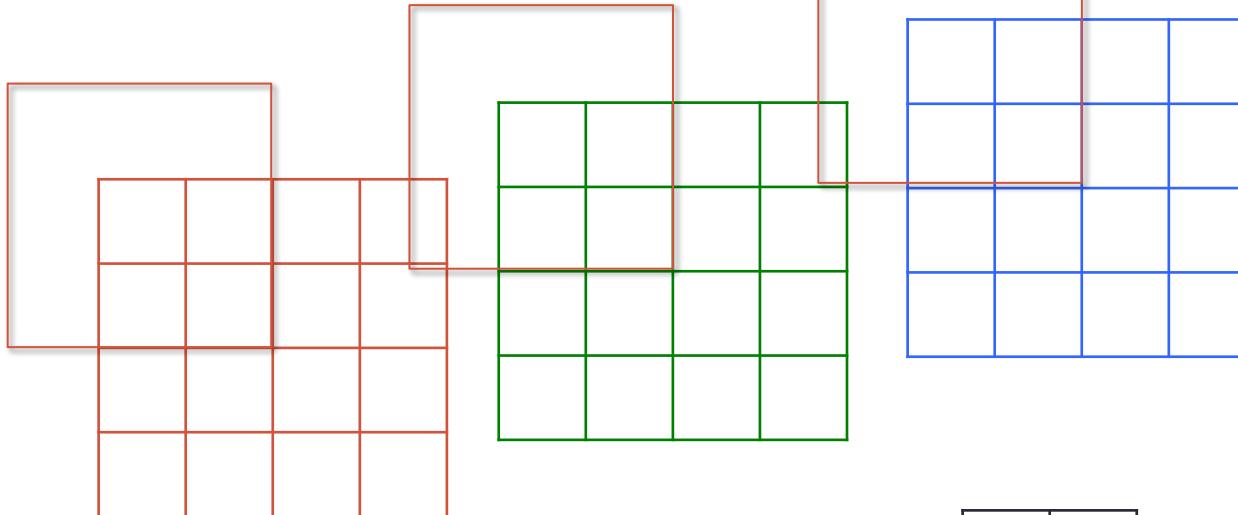
# Convolution puzzle

RGB input (3 channels) 5 filters 3x3 filter pad, stride 2, pad 1

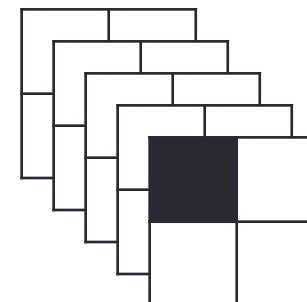


# Convolution puzzle

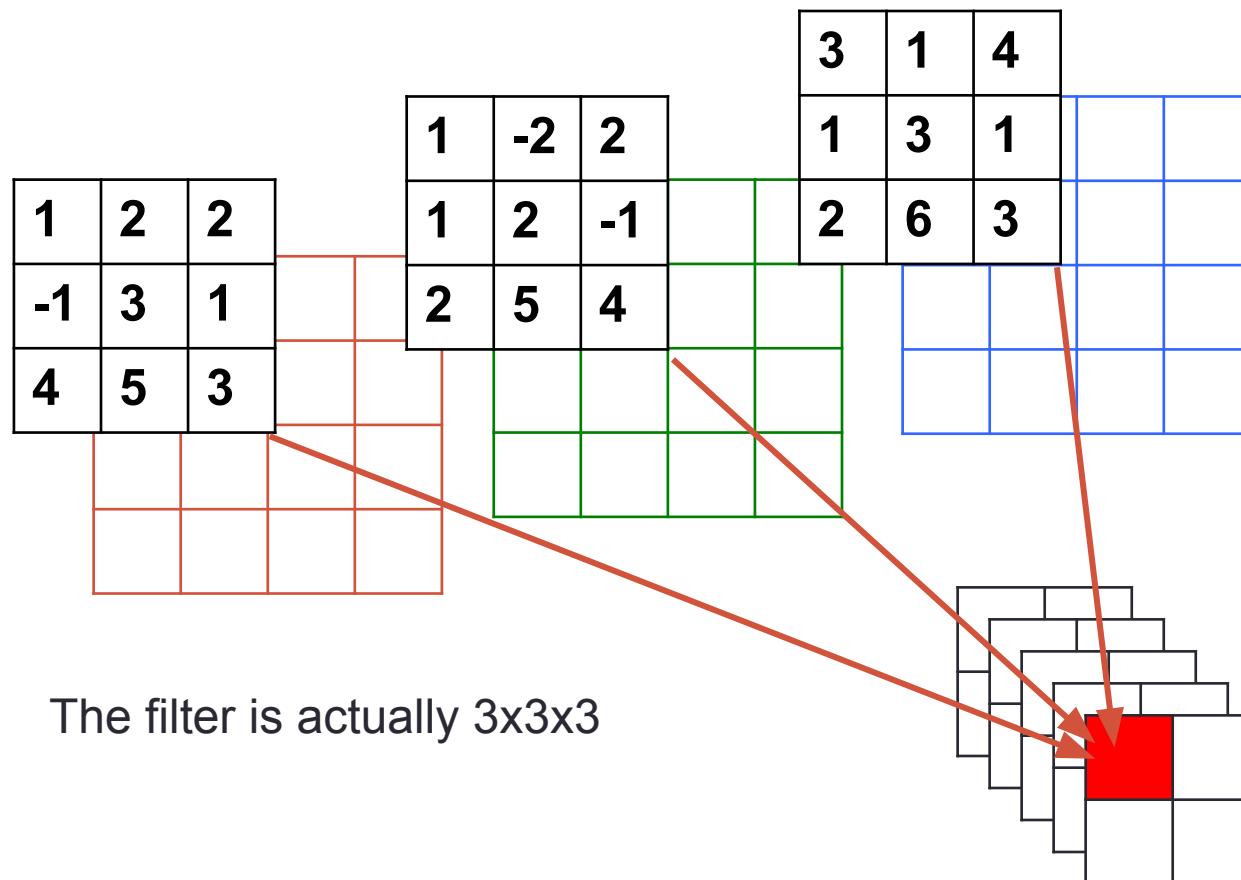
RGB input (3 channels) 5 filters 3x3 filter pad, stride 2, pad 1



The filter is actually 3x3x3

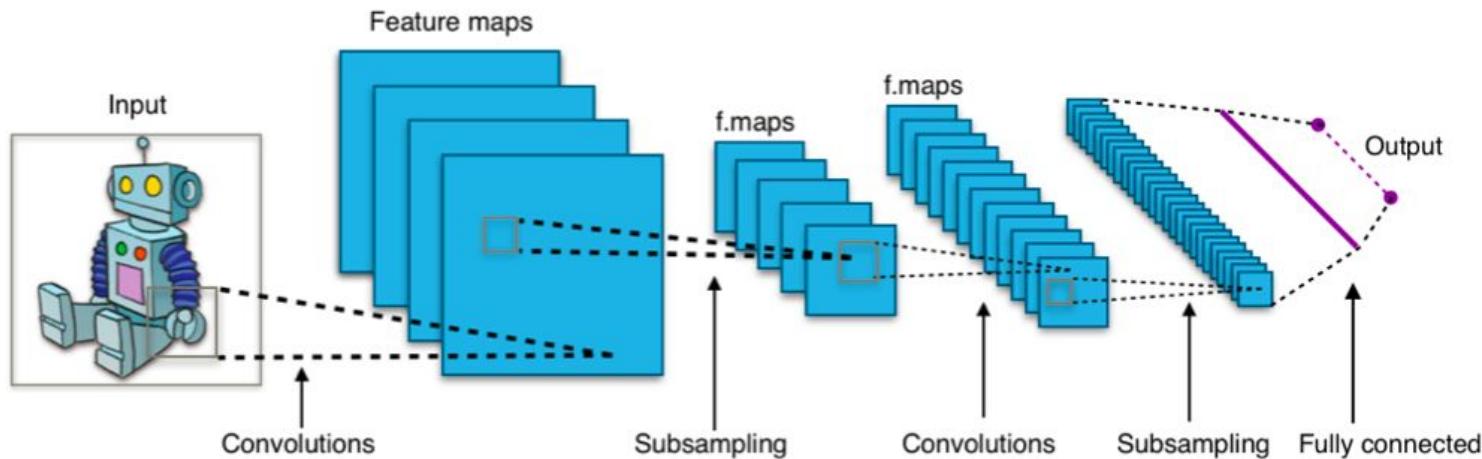


# Convolution puzzle



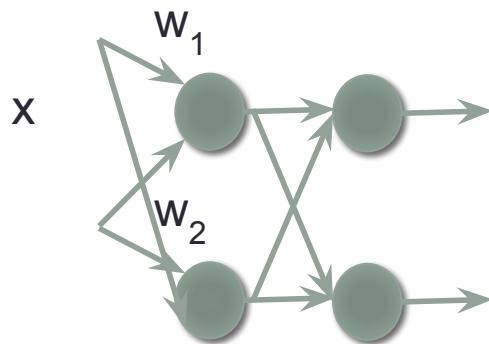
# CNN overview

- Filter size, number of filters, filter shifts, and pooling rate are all parameters
- Usually followed by a fully connected network at the end
  - CNN is good at learning low level features
  - DNN combines the features into high level features and classify

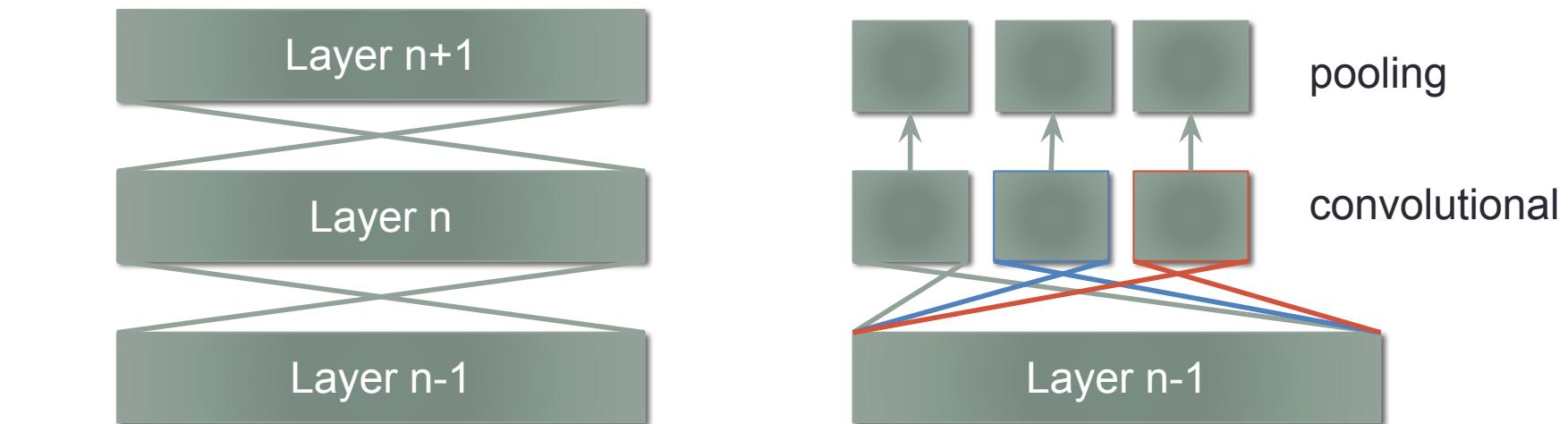


# Parameter sharing in convolution neural networks

- $W^T x$

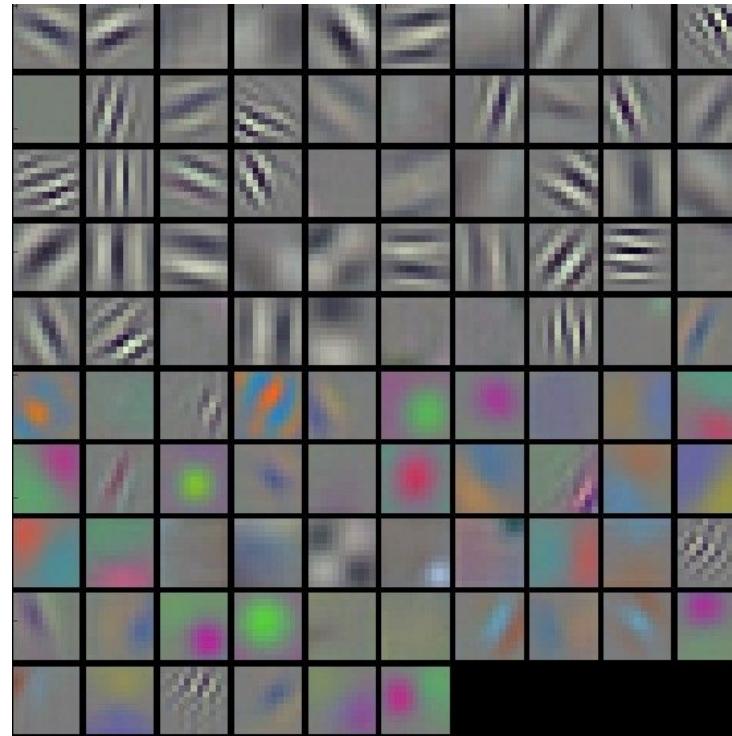


- Cats at different location might need two neurons for different locations in fully connect NNs.
- CNN shares the parameters in 1 filter
- The network is no longer fully connected



# Visualizing convolutional layers

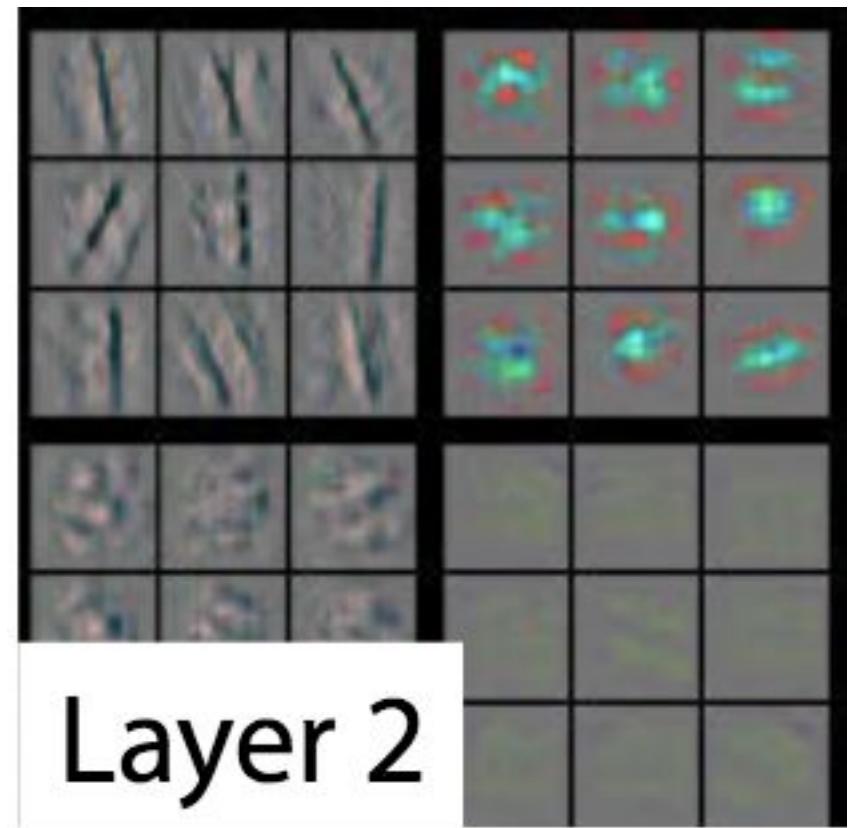
- We can visualize the weights of the filter
- “Matched filters”



# Higher layer captures higher-level concepts

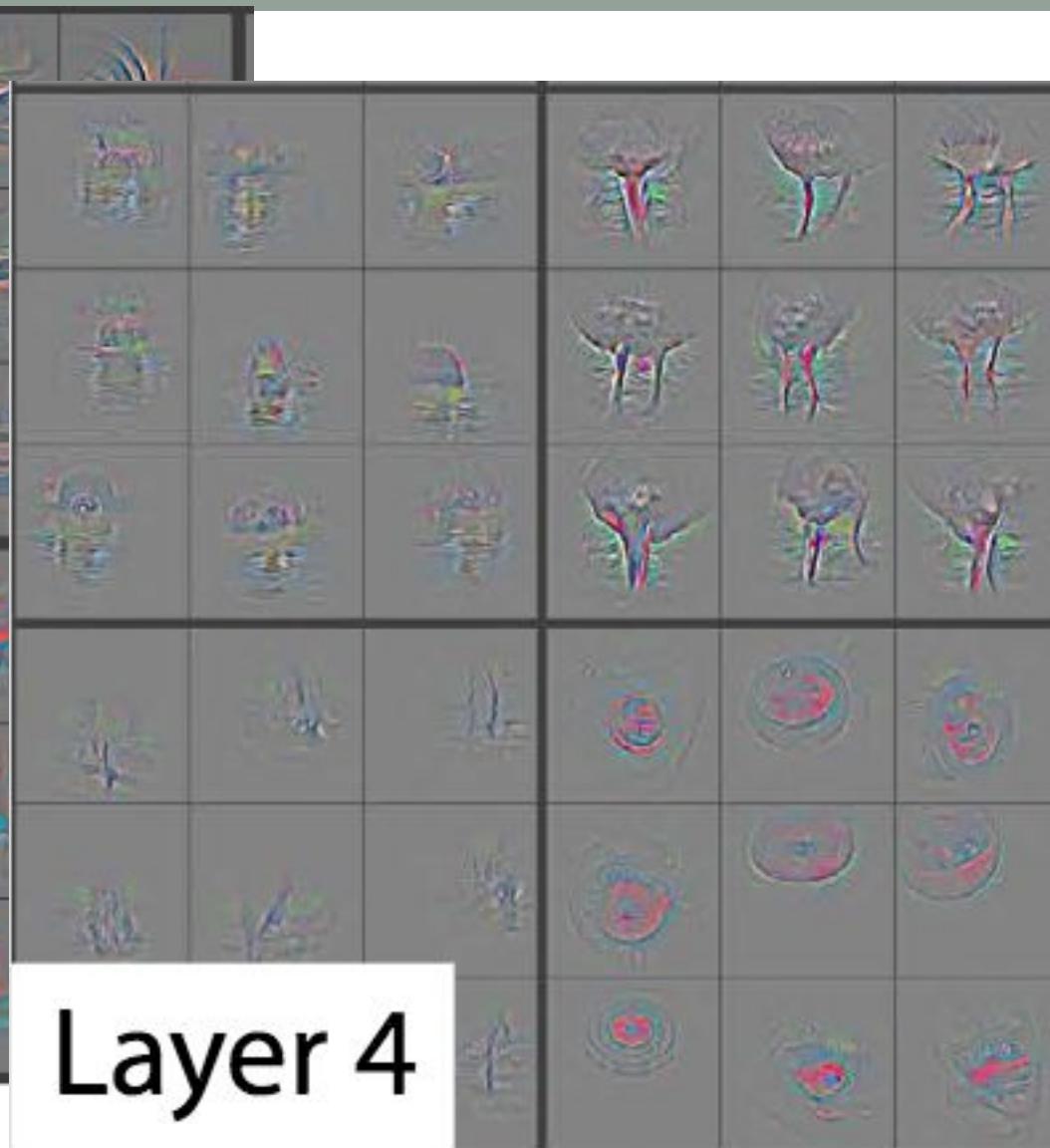


Layer 1





Layer 3



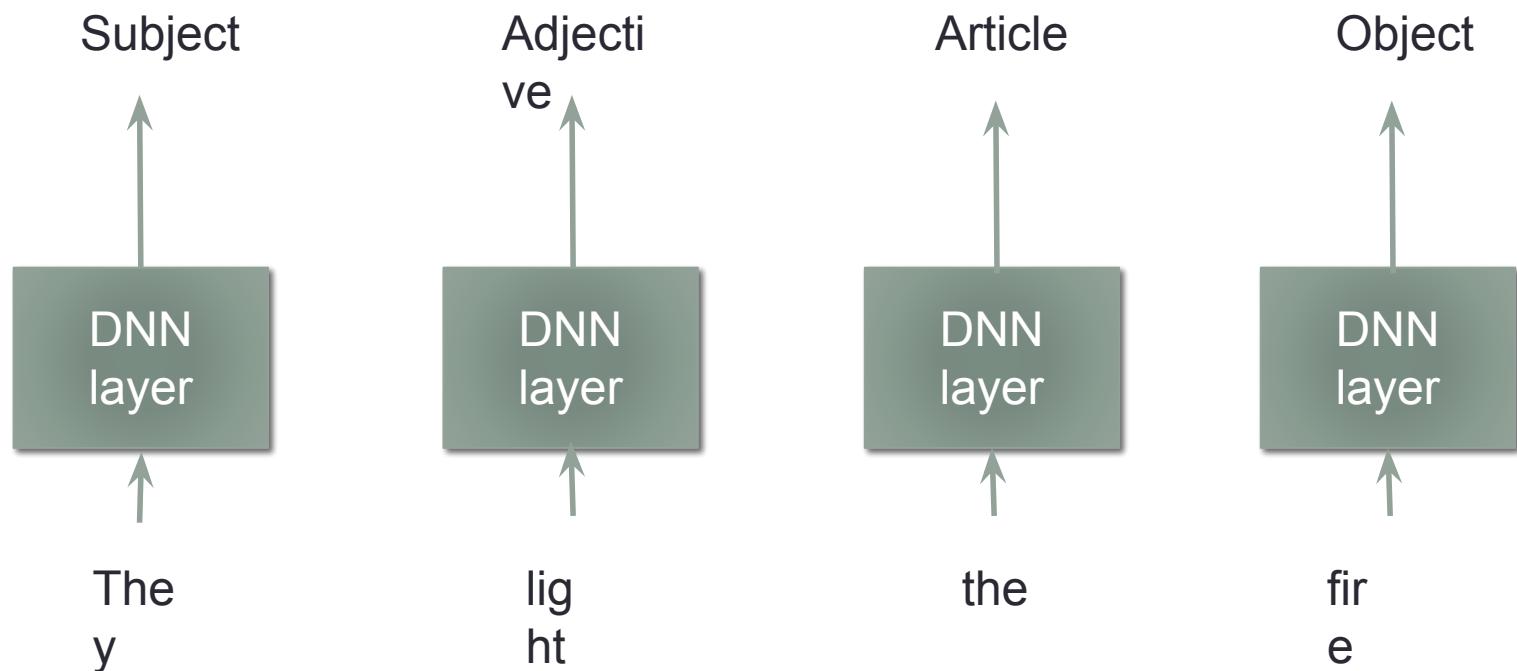
Layer 4

# CNN

- Convolutional layer
- Subsampling
- Sharing of parameters in space
- Sharing of parameters in time?

# Recurrent neural network (RNN)

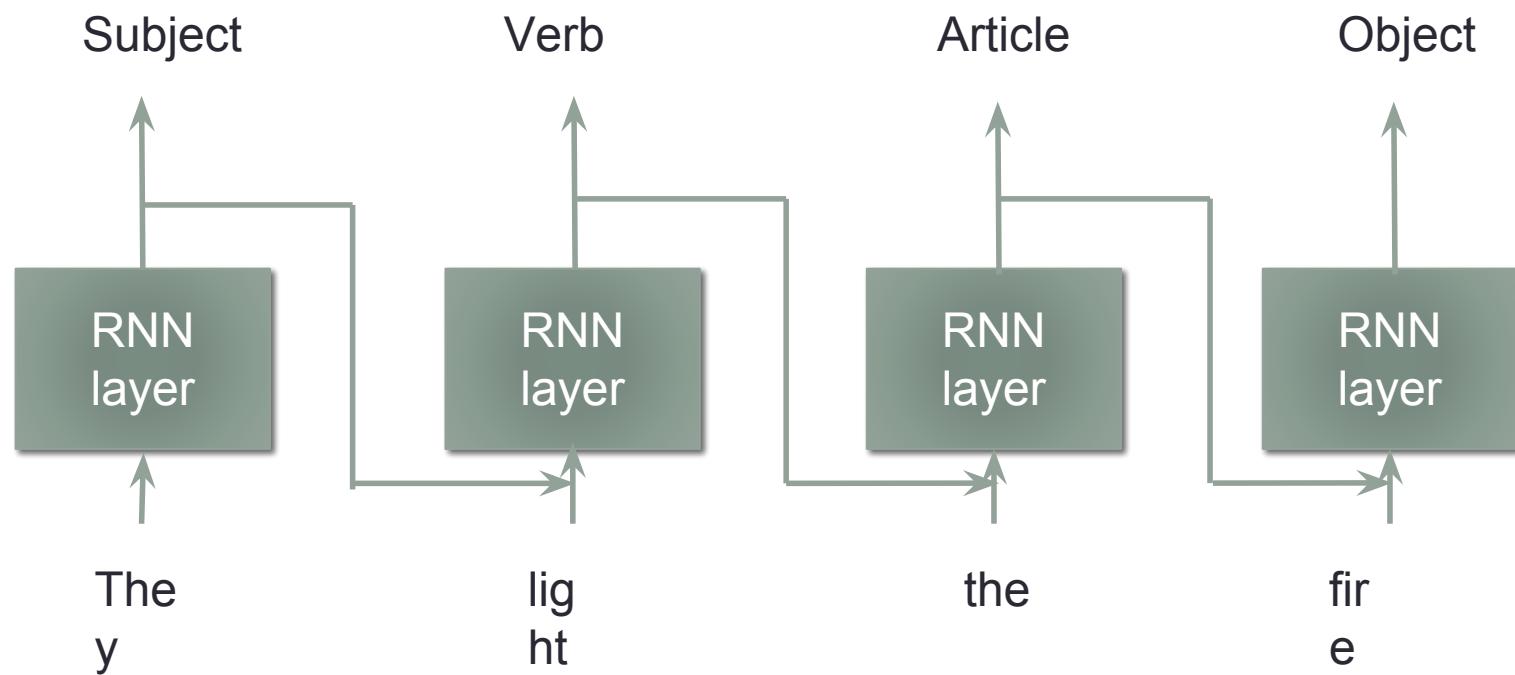
- DNN framework



Problem1: need a way to remember the past

# Recurrent neural network (RNN)

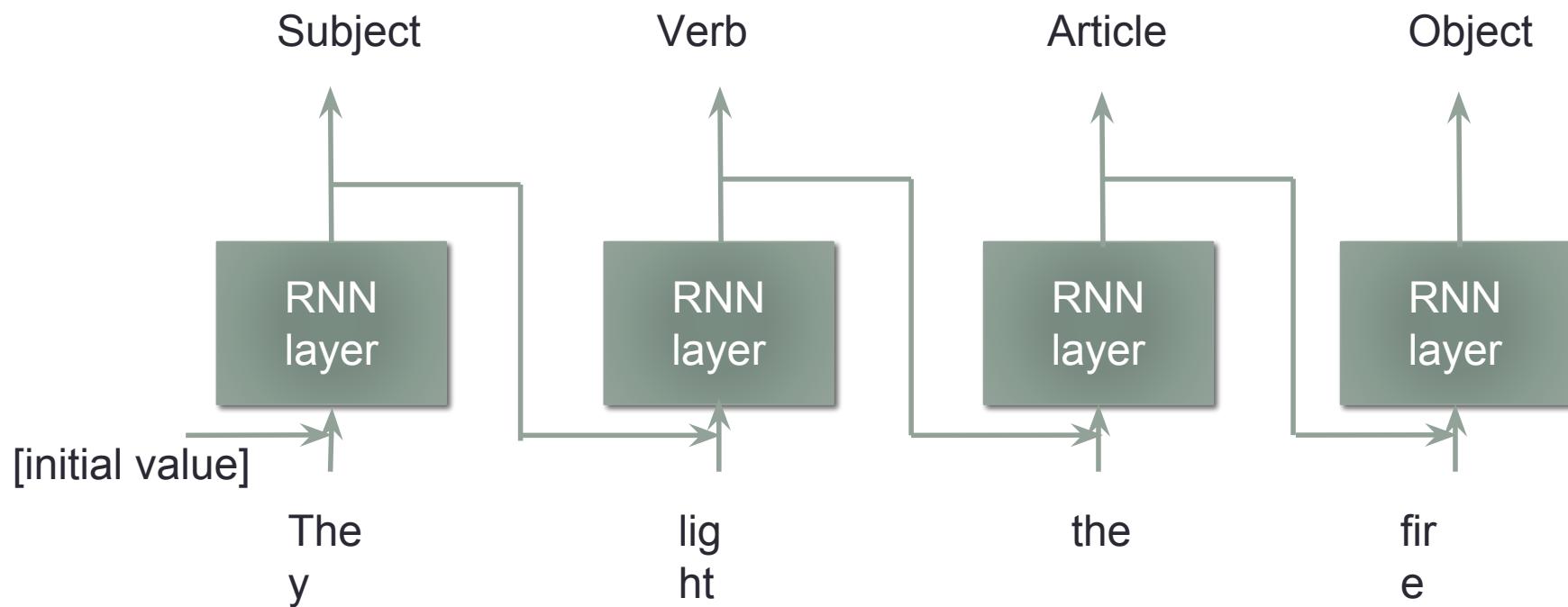
- RNN framework



Output of the layer encodes something meaningful about the past

# Recurrent neural network (RNN)

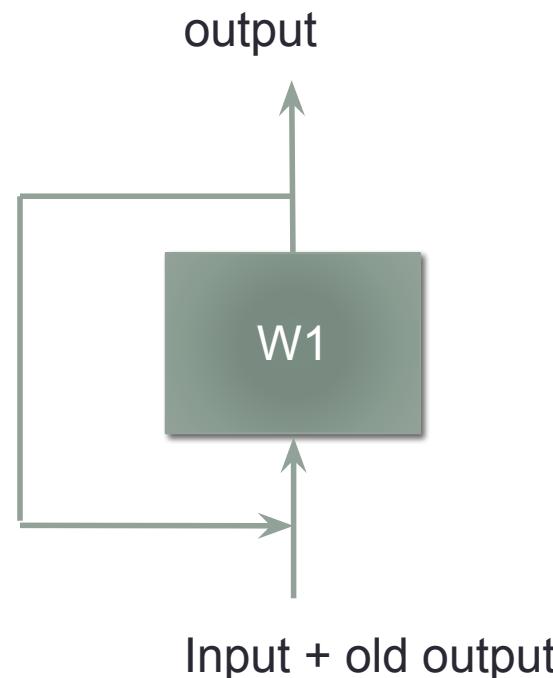
- RNN framework



New input feature = [original input feature, output of the layer at previous time step]

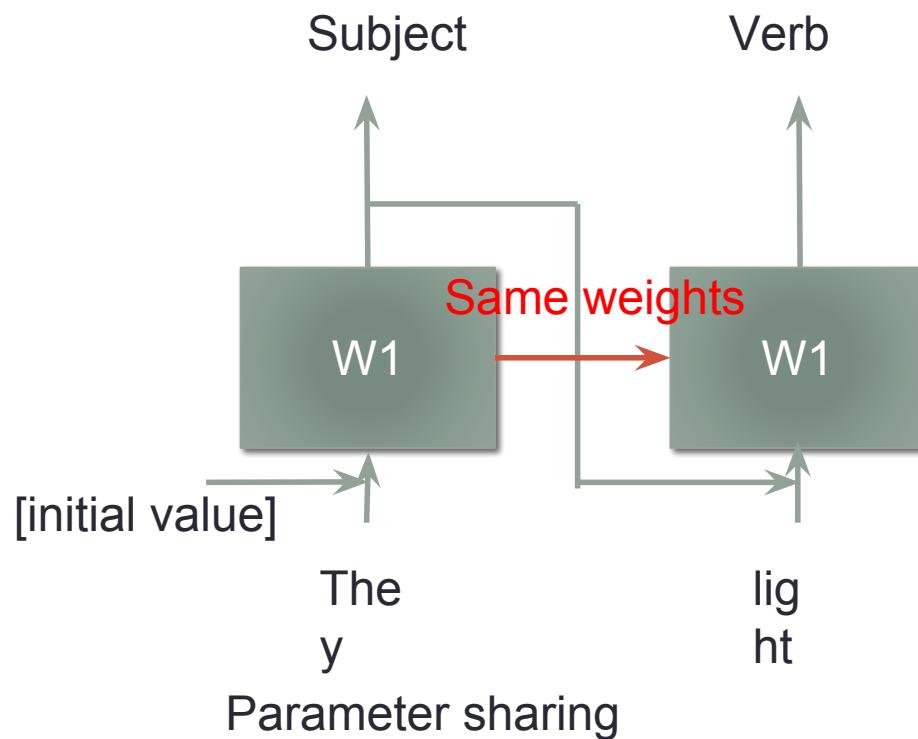
# Recurrent neural network (RNN)

- Unrolling of a recurrent layer.



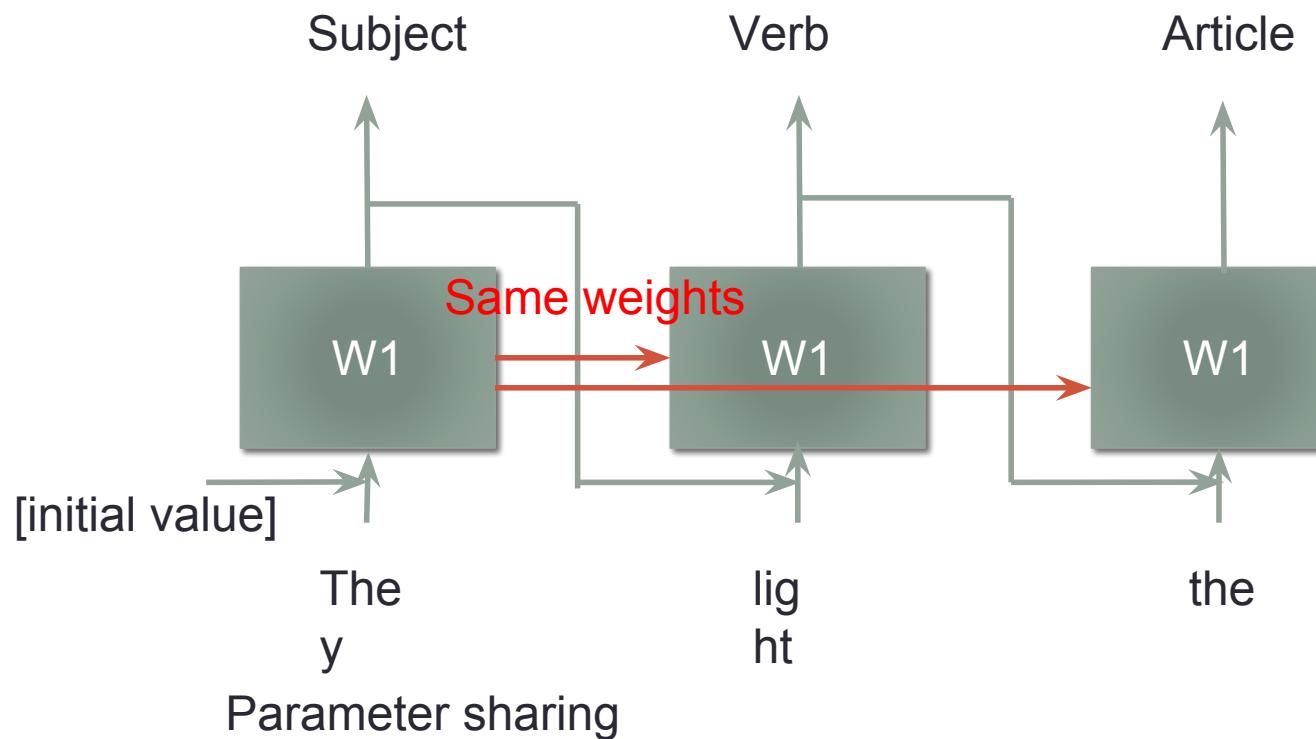
# Recurrent neural network (RNN)

- Unrolling of a recurrent layer.



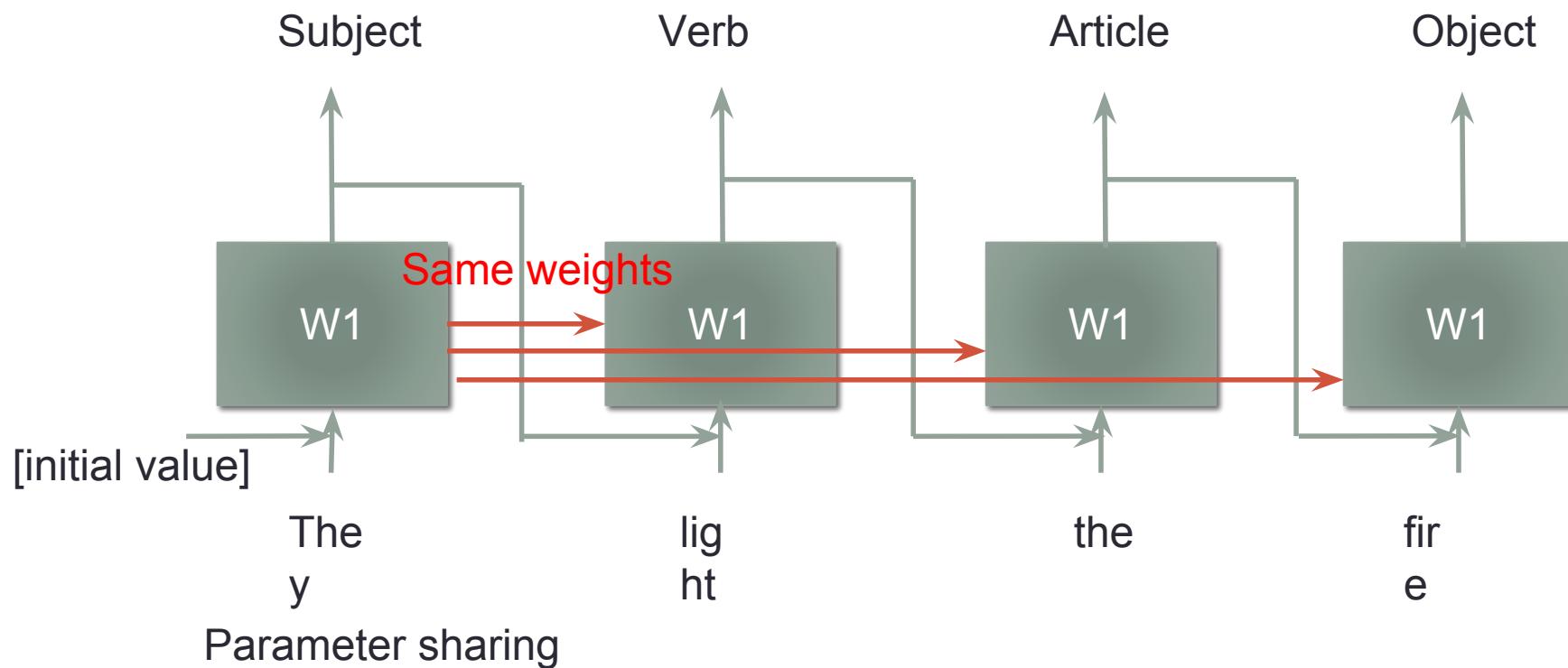
# Recurrent neural network (RNN)

- Unrolling of a recurrent layer.



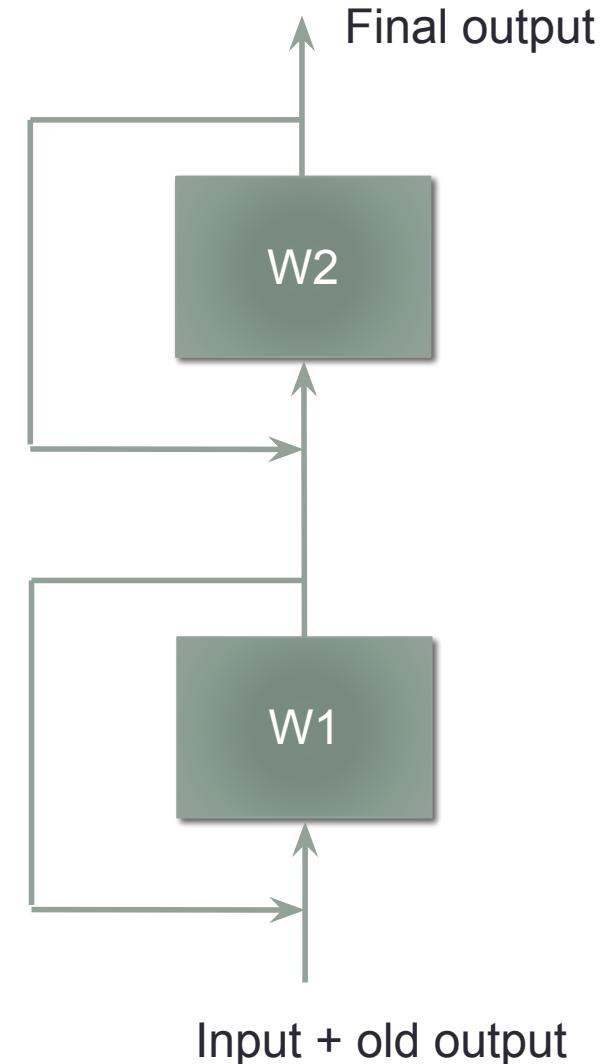
# Recurrent neural network (RNN)

- Unrolling of a recurrent layer.



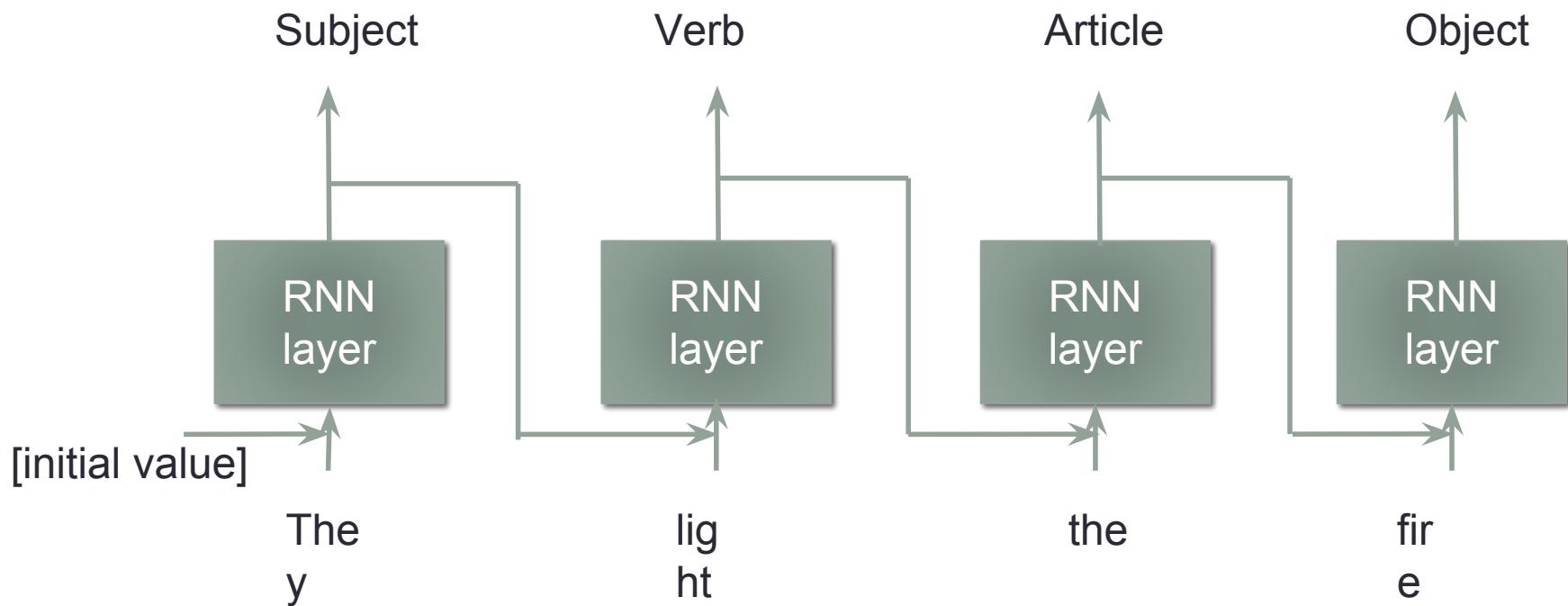
# Recurrent neural network (RNN)

- Stacks of recurrent layer



# Training a recurrent neural network

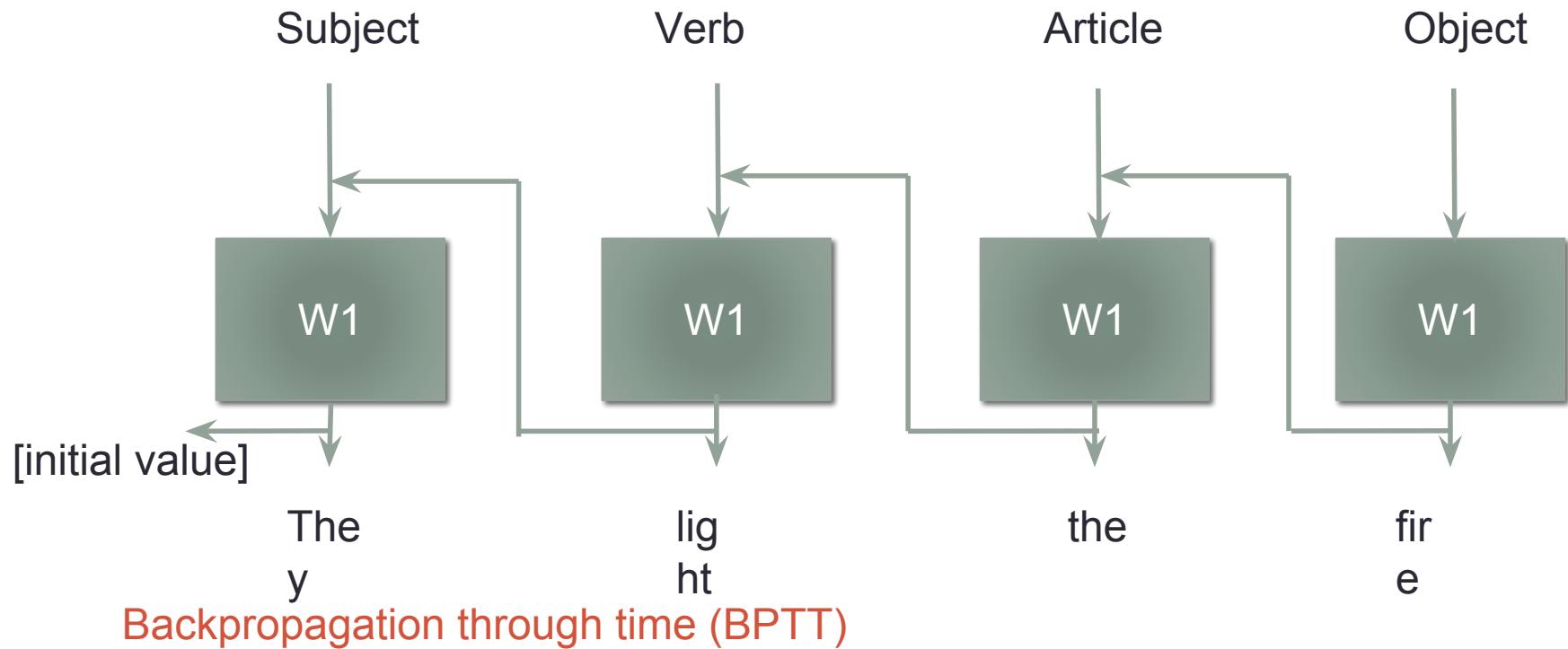
- RNN framework



New input feature = [original input feature, output of the layer at previous time step]

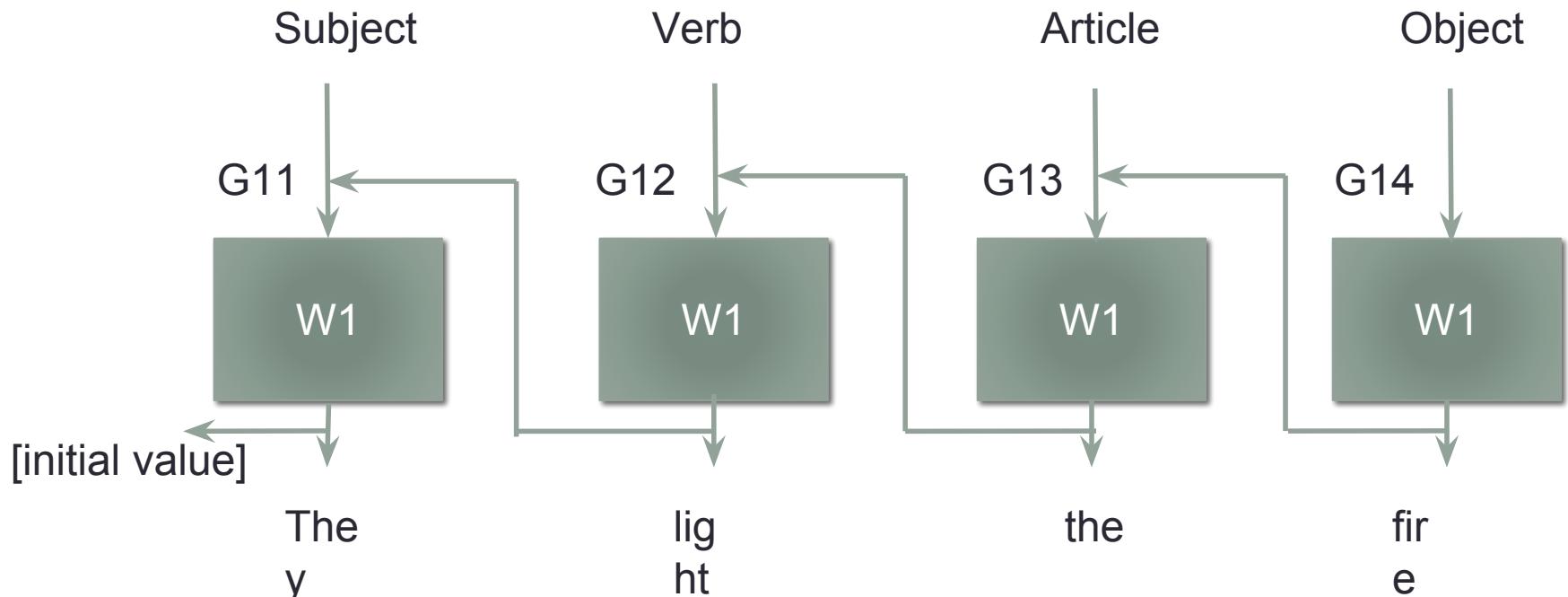
# Training a recurrent neural network

- Backward Computation graph



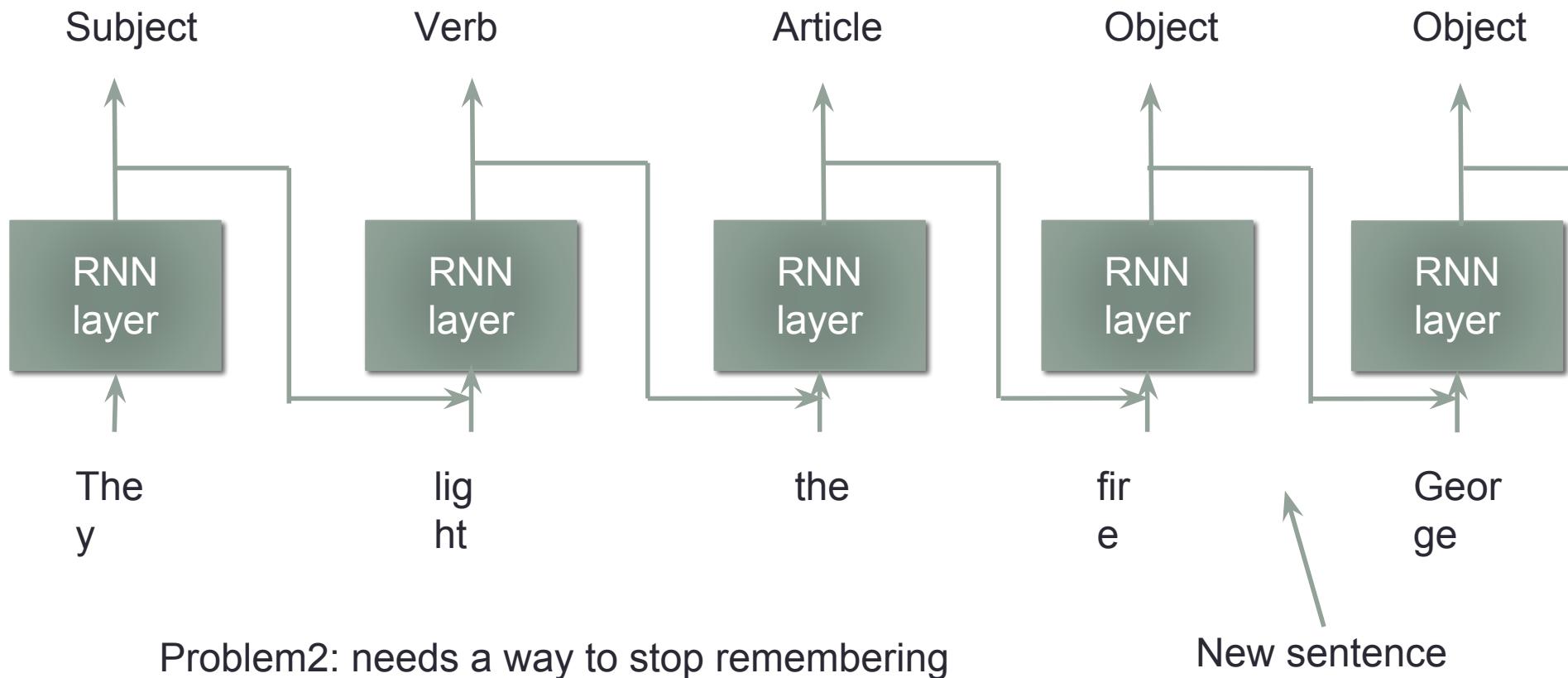
# BPTT

- Backward Computation graph



Problem1: cannot deal with infinitely long recurrent  
Gradient explosion, vanishing

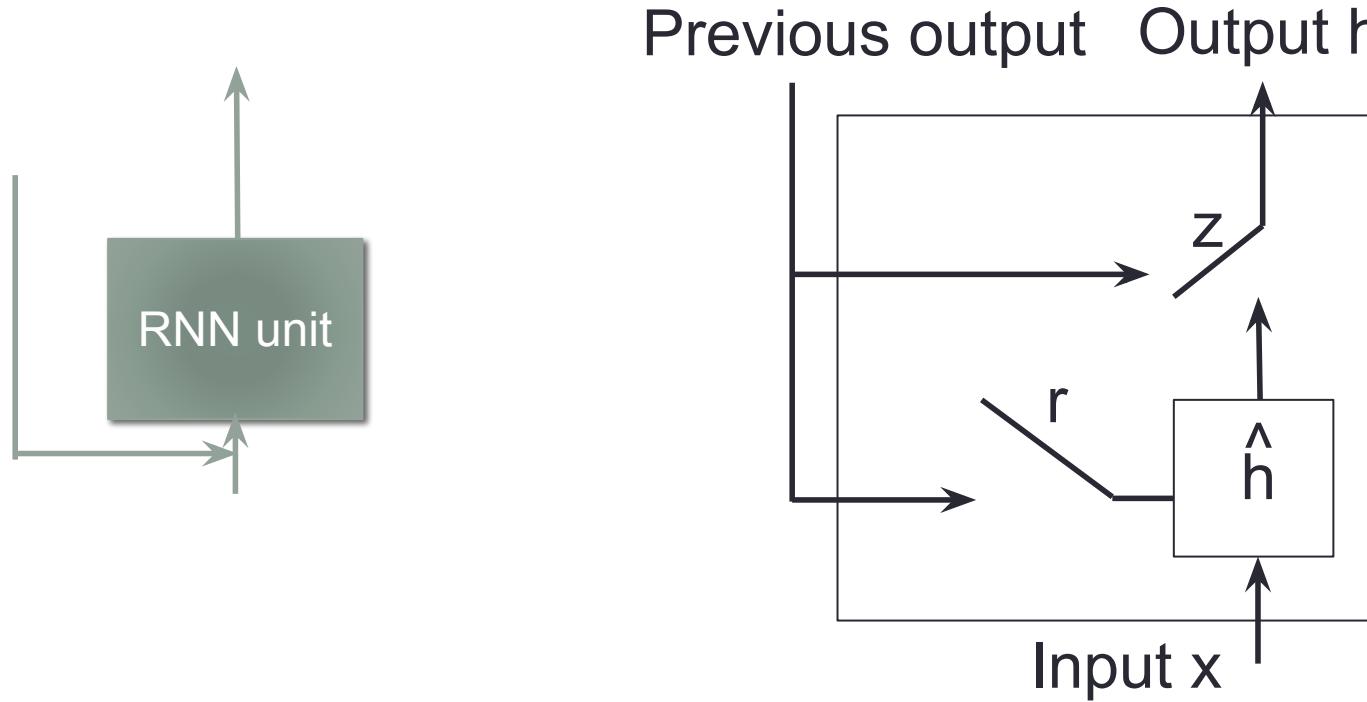
# Recurrent neural network (RNN)



Can the network learn when to start and stop remembering things?

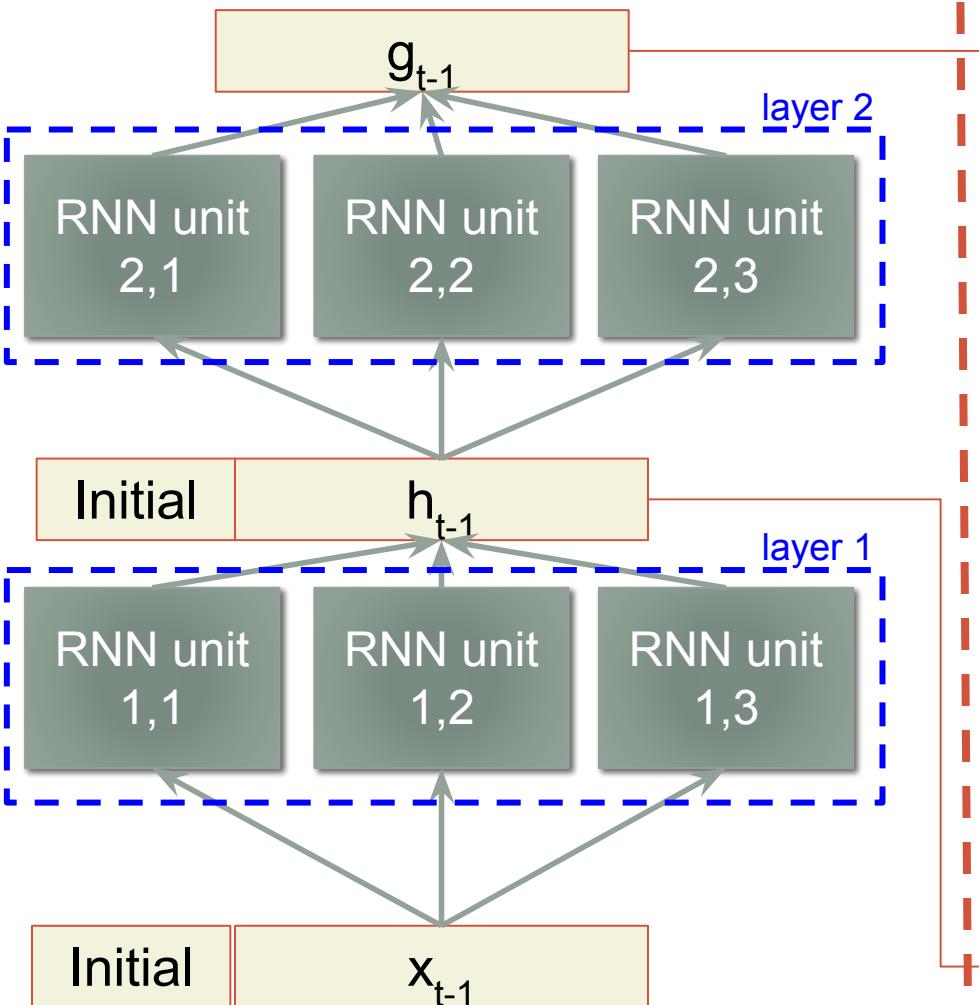
# Gated Recurrent Unit (GRU)

- Forms a Gated Recurrent Neural Networks (GRNN)
- Add gates that can choose to reset ( $r$ ) or update ( $z$ )

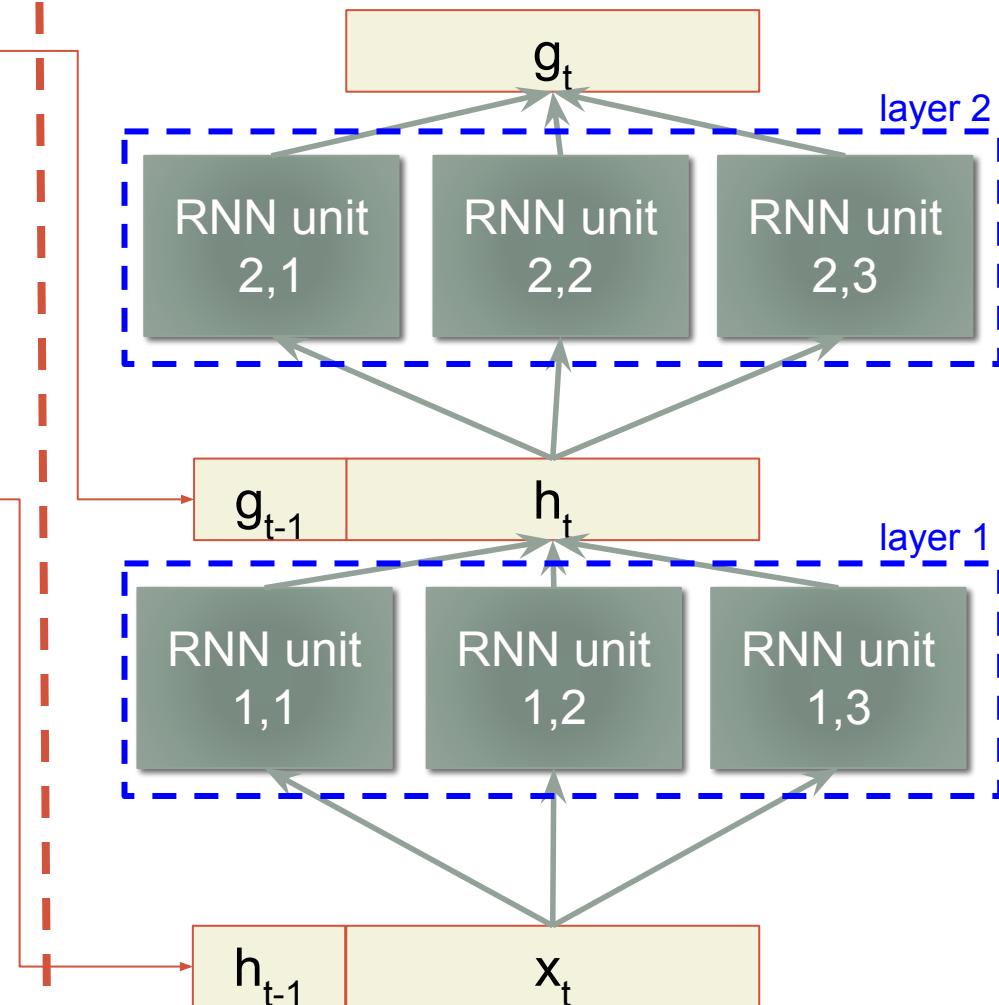


# RNN layers (expanded in time)

Time step t-1

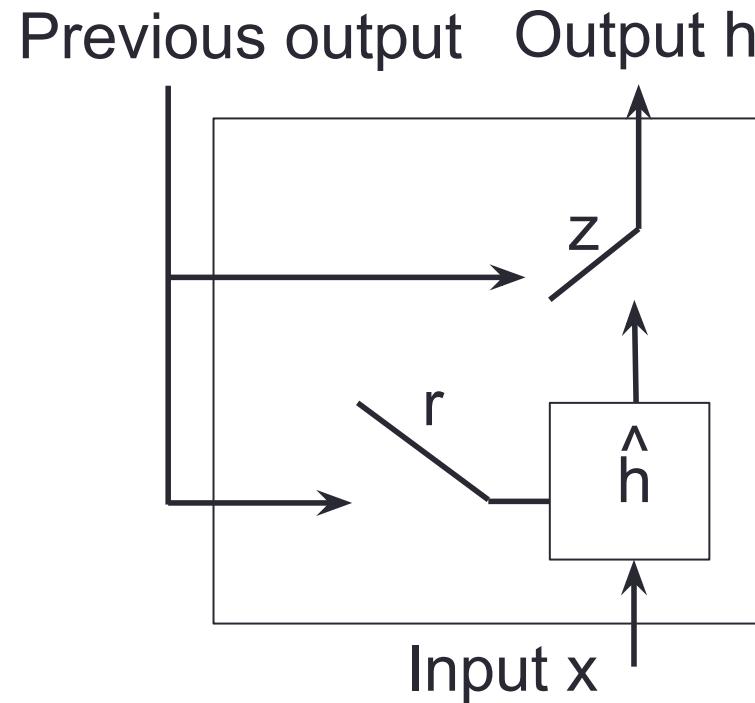
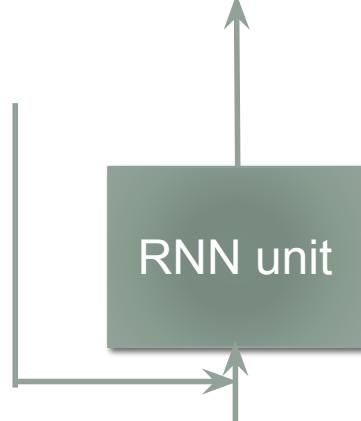


Time step t



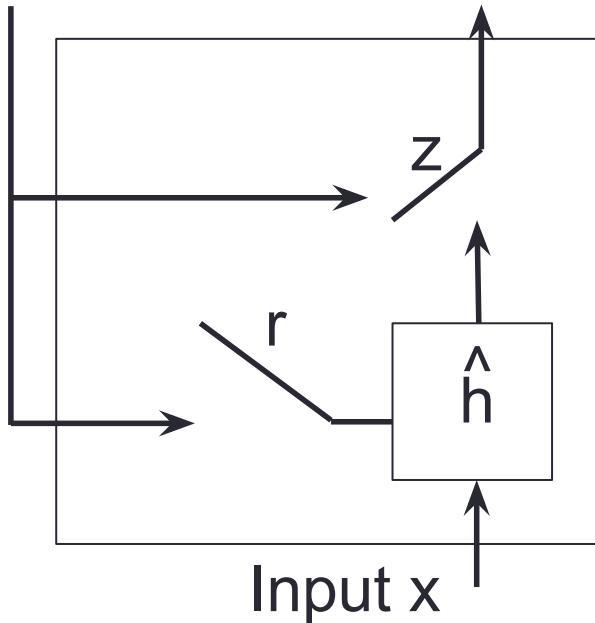
# Gated Recurrent Unit (GRU)

- Forms a Gated Recurrent Neural Networks (GRNN)
- Add gates that can choose to reset ( $r$ ) or update ( $z$ )



# Gated Recurrent Unit (GRU)

Previous output    Output h

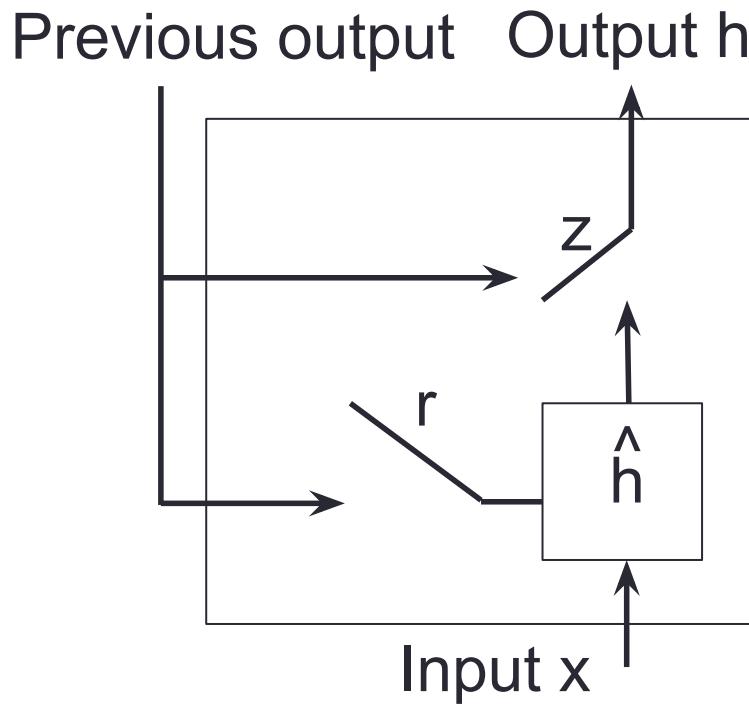


Neuron index

$$h_t^j = (1 - z_t^j)h_{t-1}^j + z_t^j \hat{h}_t^j$$

time index

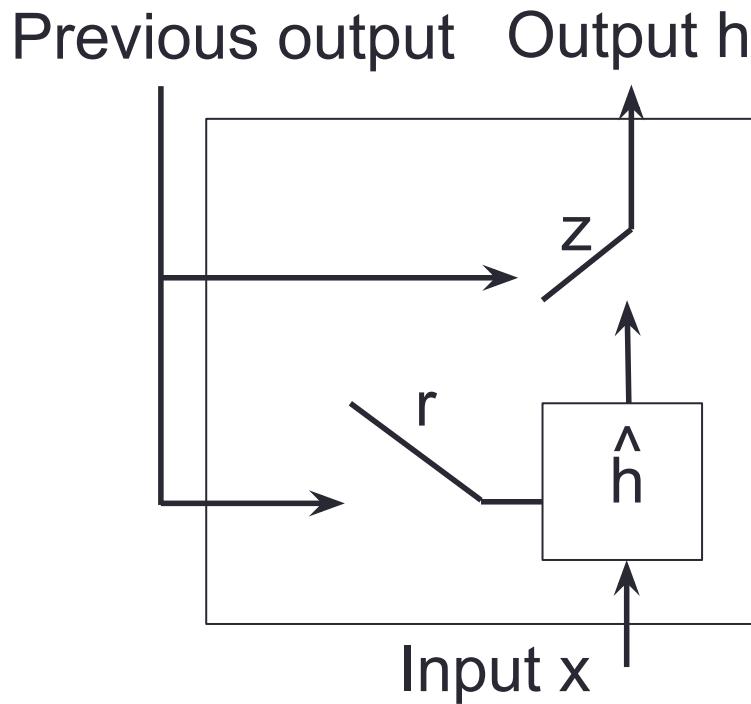
# Gated Recurrent Unit (GRU)



$$h_t^j = (1 - z_t^j)h_{t-1}^j + z_t^j \hat{h}_t^j$$

One GRU neuron output (scalar)

# Gated Recurrent Unit (GRU)



$$h_t^j = (1 - z_t^j)h_{t-1}^j + z_t^j \hat{h}_t^j$$

$$\hat{h}_t^j = \tanh^j (W \mathbf{x}_t + U (\mathbf{r}_t \odot \mathbf{h}_{t-1}))$$

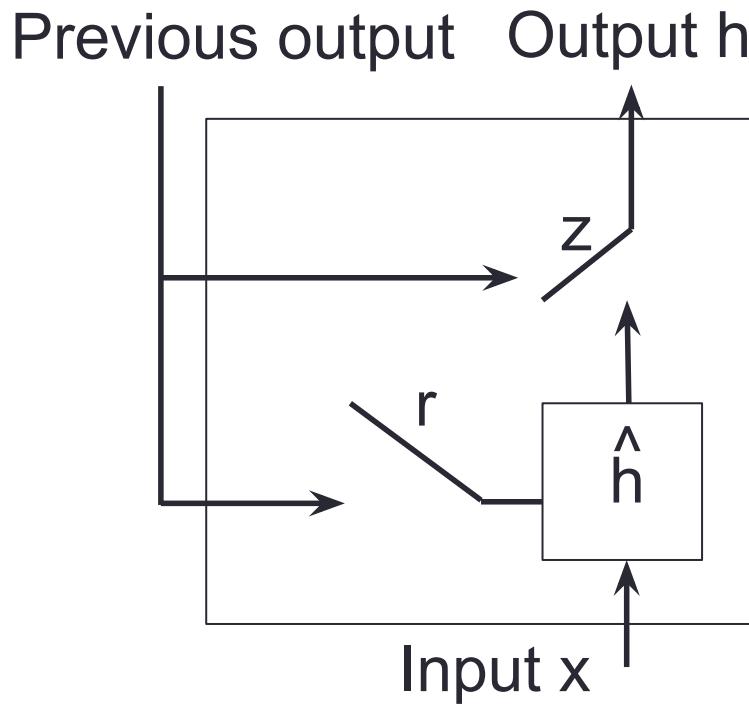
Linear transform with matrix multiply

Vector (each value from each GRU unit in the previous layer)

$$\mathbf{x}_t^j = \mathbf{h}_t^j$$

Element-wise product

# Gated Recurrent Unit (GRU)



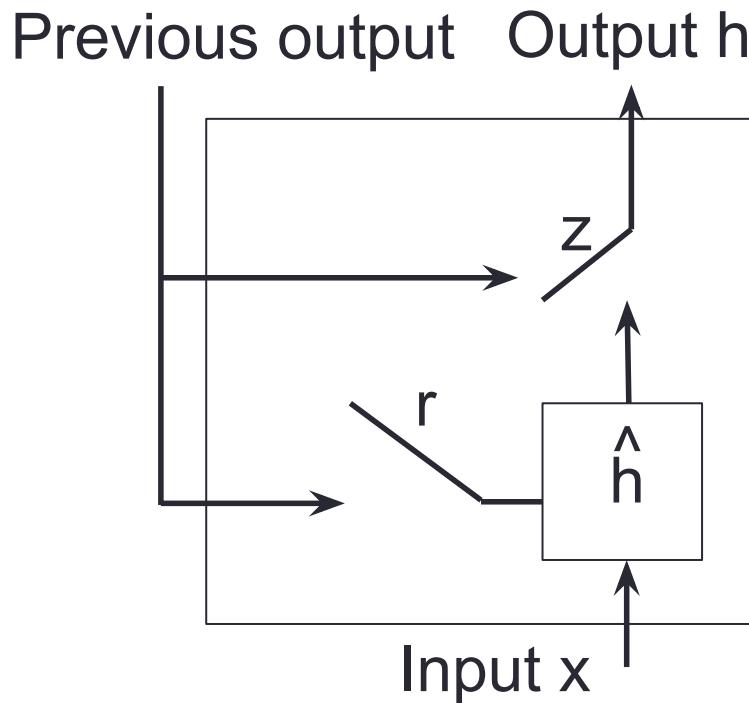
$$h_t^j = (1 - z_t^j)h_{t-1}^j + z_t^j \hat{h}_t^j$$

$$\hat{h}_t^j = \underline{\tanh^j}(W\mathbf{x}_t + U(\mathbf{r}_t \odot \mathbf{h}_{t-1}))$$

Takes the j-th element

Bounds the output

# Gated Recurrent Unit (GRU)



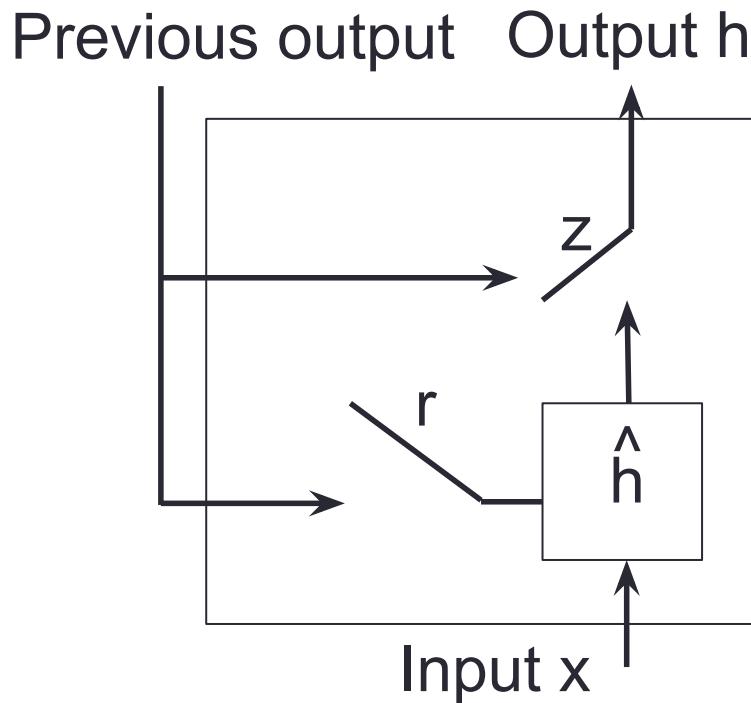
$$h_t^j = (1 - z_t^j)h_{t-1}^j + z_t^j \hat{h}_t^j$$

$$\hat{h}_t^j = \tanh^j(W\mathbf{x}_t + U(\mathbf{r}_t \odot \mathbf{h}_{t-1}))$$

$$z_t^j = \text{sigmoid}^j(W_z\mathbf{x}_t + U_z\mathbf{h}_{t-1})$$

Indicates a different set of weights

# Gated Recurrent Unit (GRU)



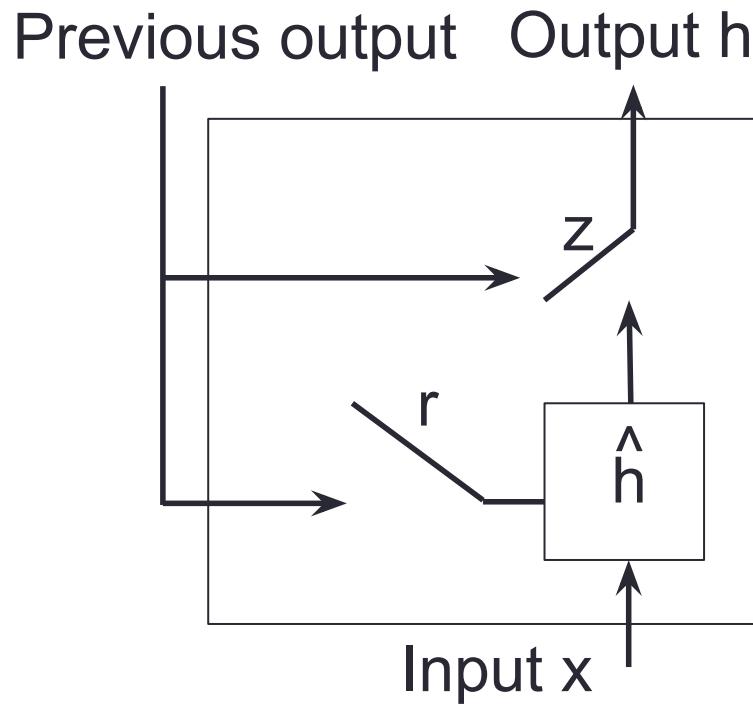
$$h_t^j = (1 - z_t^j)h_{t-1}^j + z_t^j \hat{h}_t^j$$

$$\hat{h}_t^j = \tanh^j(W\mathbf{x}_t + U(\mathbf{r}_t \odot \mathbf{h}_{t-1}))$$

$$z_t^j = \text{sigmoid}^j(W_z \mathbf{x}_t + U_z \mathbf{h}_{t-1})$$

Bounds the output to 0 to 1 for interpolation

# Gated Recurrent Unit (GRU)



$$h_t^j = (1 - z_t^j)h_{t-1}^j + z_t^j \hat{h}_t^j$$

$$\hat{h}_t^j = \tanh^j(W\mathbf{x}_t + U(\mathbf{r}_t \odot \mathbf{h}_{t-1}))$$

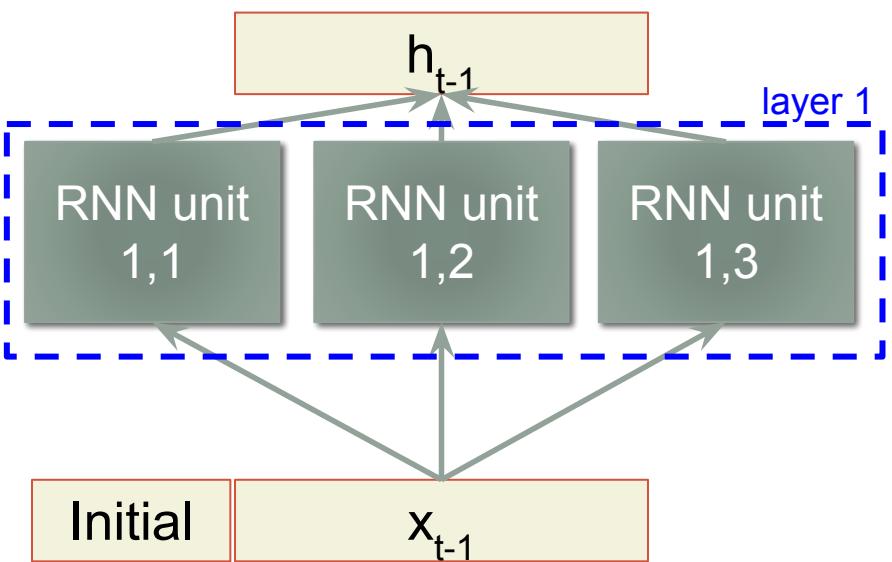
$$z_t^j = \text{sigmoid}^j(W_z \mathbf{x}_t + U_z \mathbf{h}_{t-1})$$

$$r_t^j = \text{sigmoid}^j(W_r \mathbf{x}_t + U_r \mathbf{h}_{t-1})$$

# Gated Recurrent Unit (GRU) layer

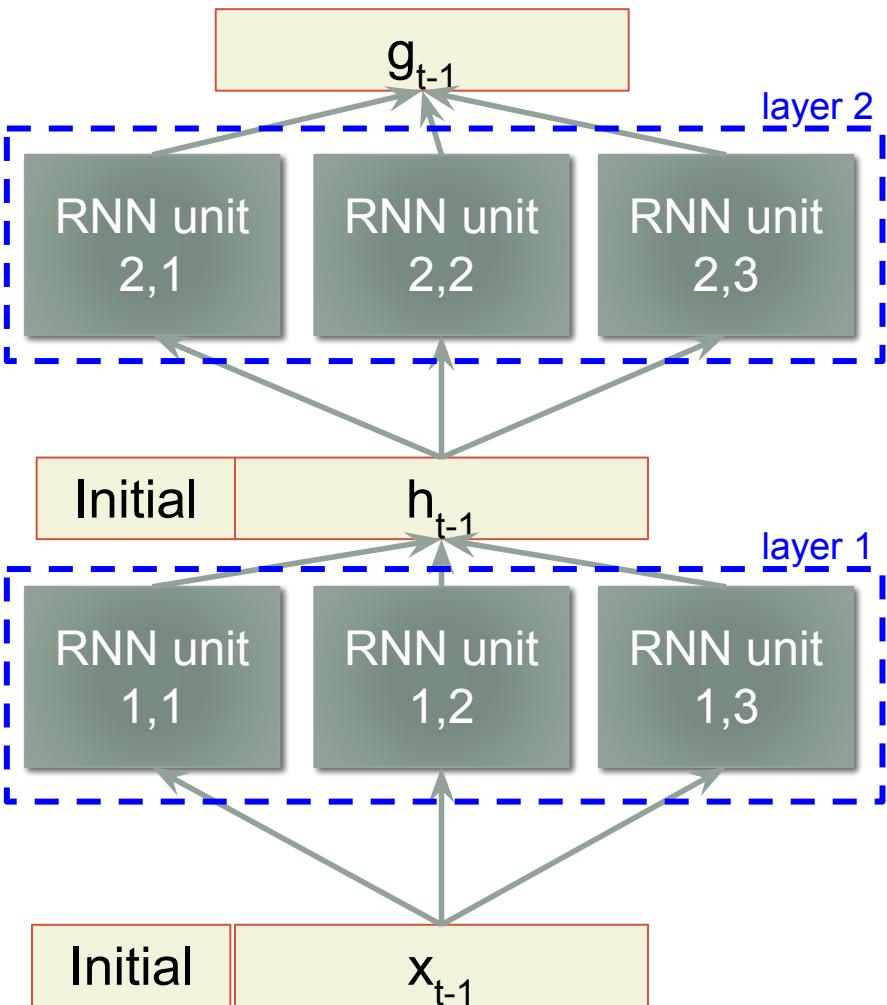
Time step 1

Time step 2



# Gated Recurrent Unit (GRU) layer

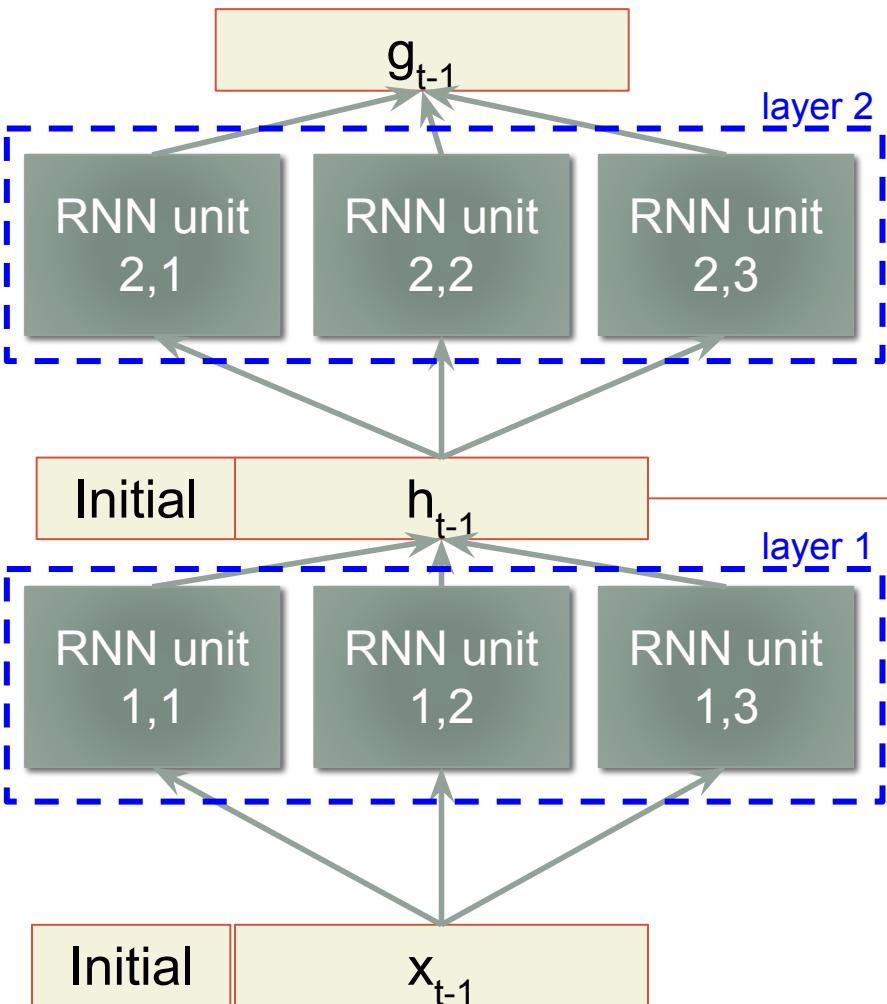
Time step 1



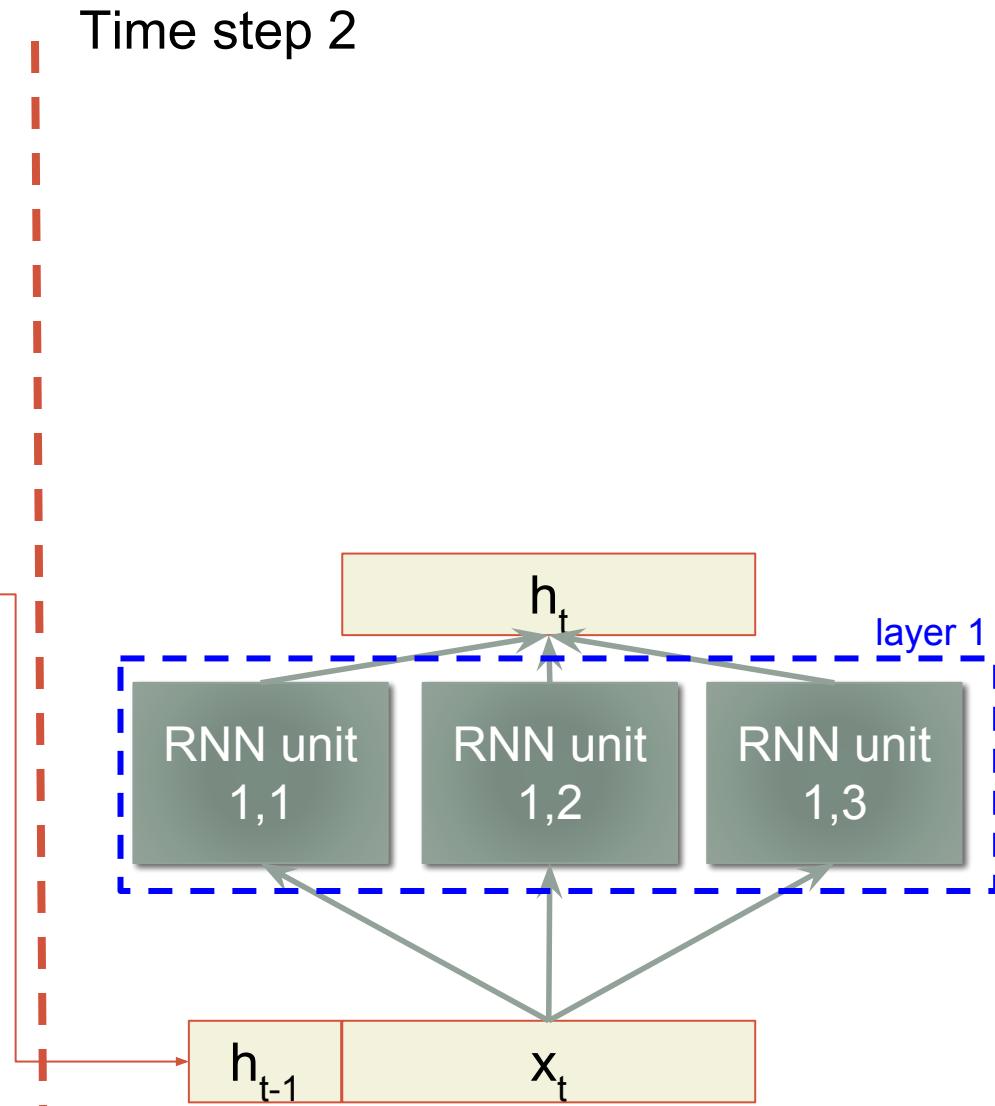
Time step 2

# Gated Recurrent Unit (GRU) layer

Time step 1

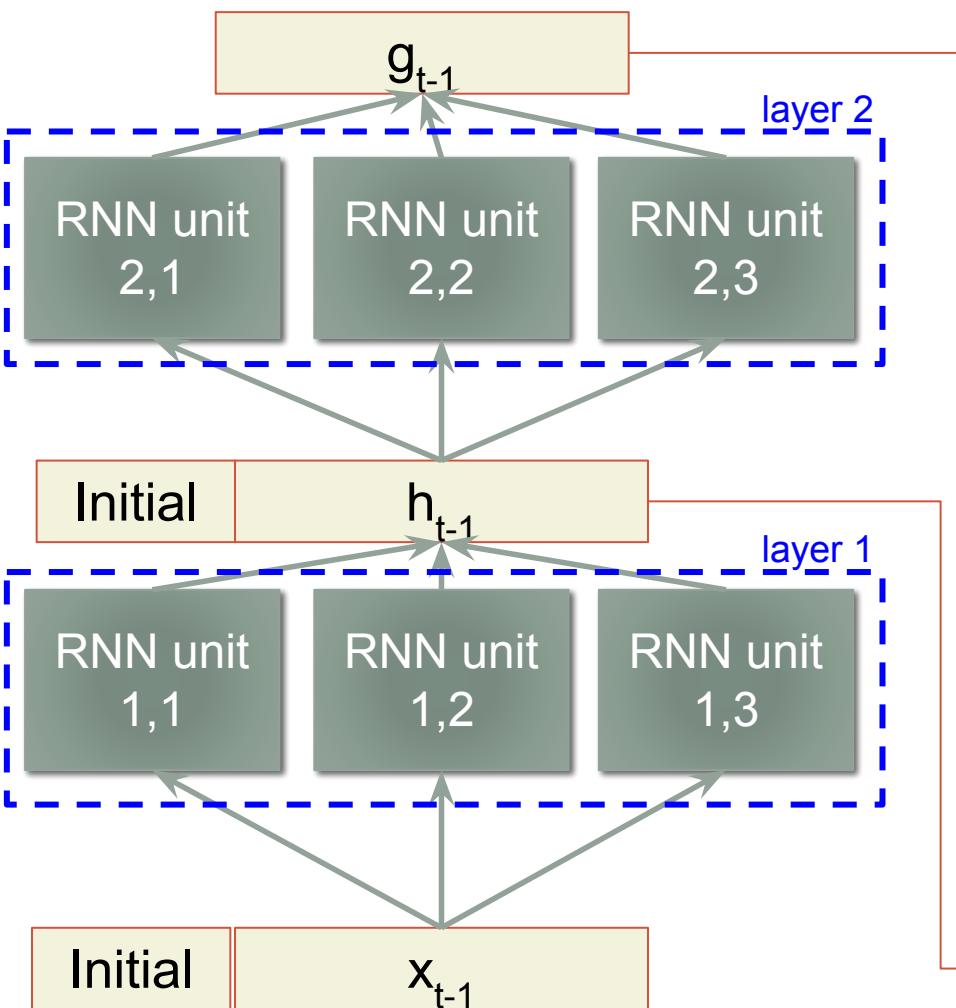


Time step 2

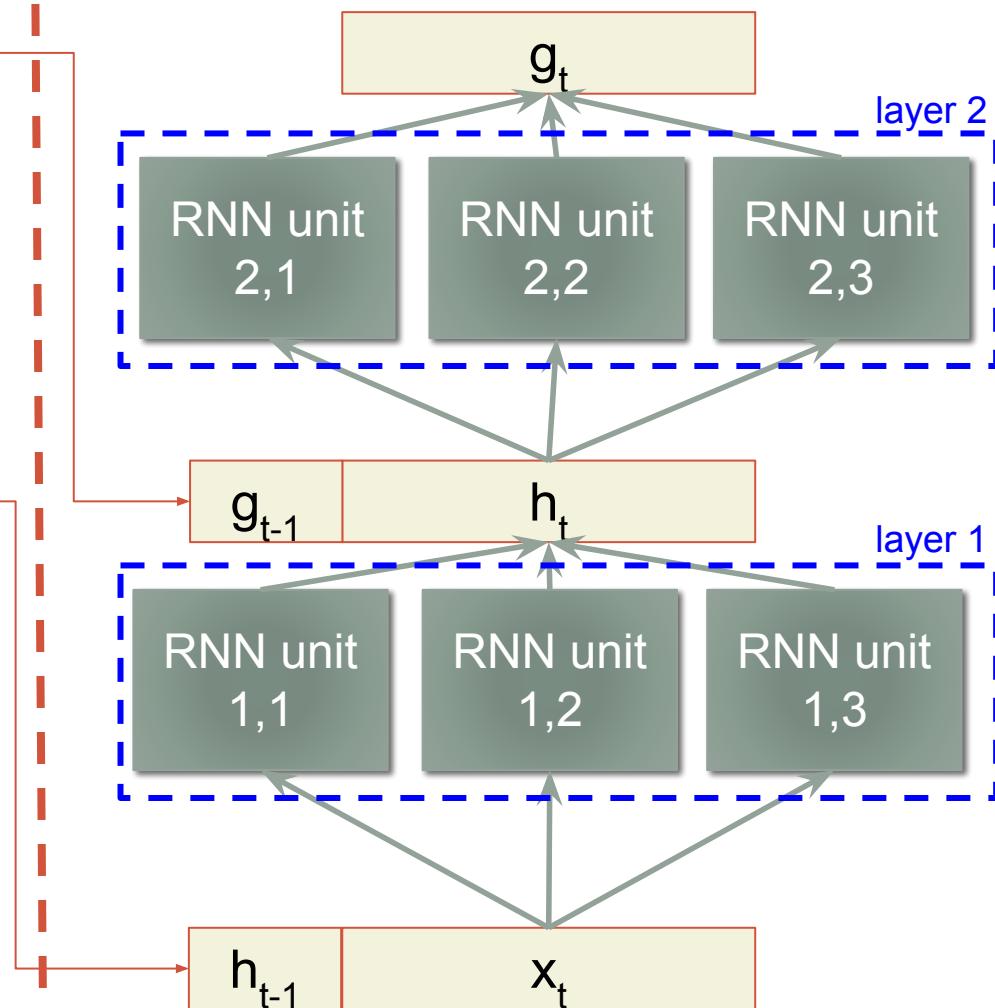


# Gated Recurrent Unit (GRU) layer

Time step 1

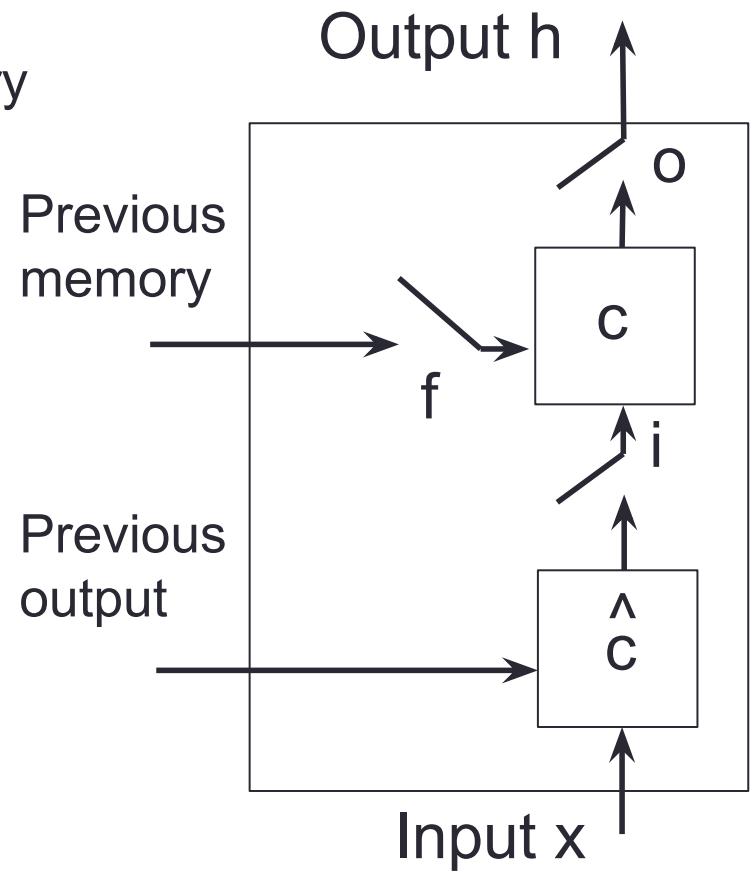
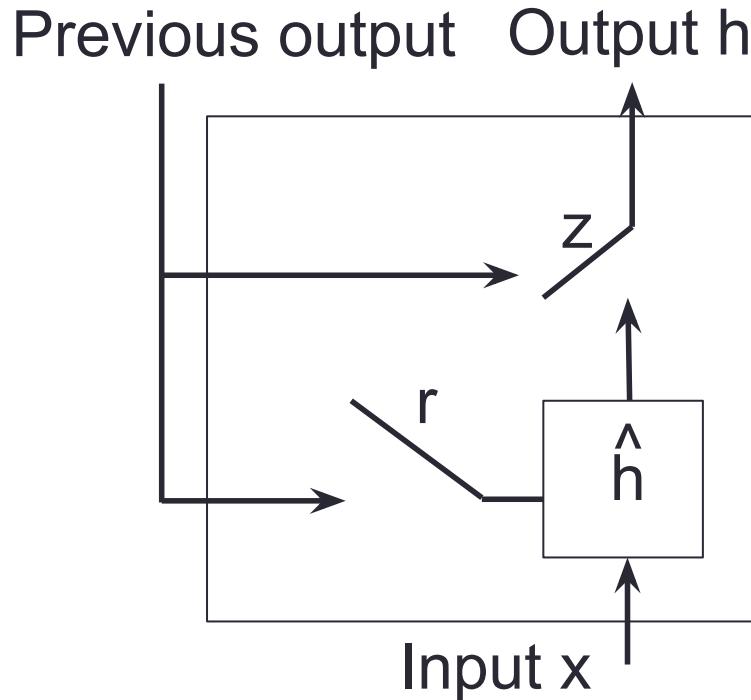


Time step 2

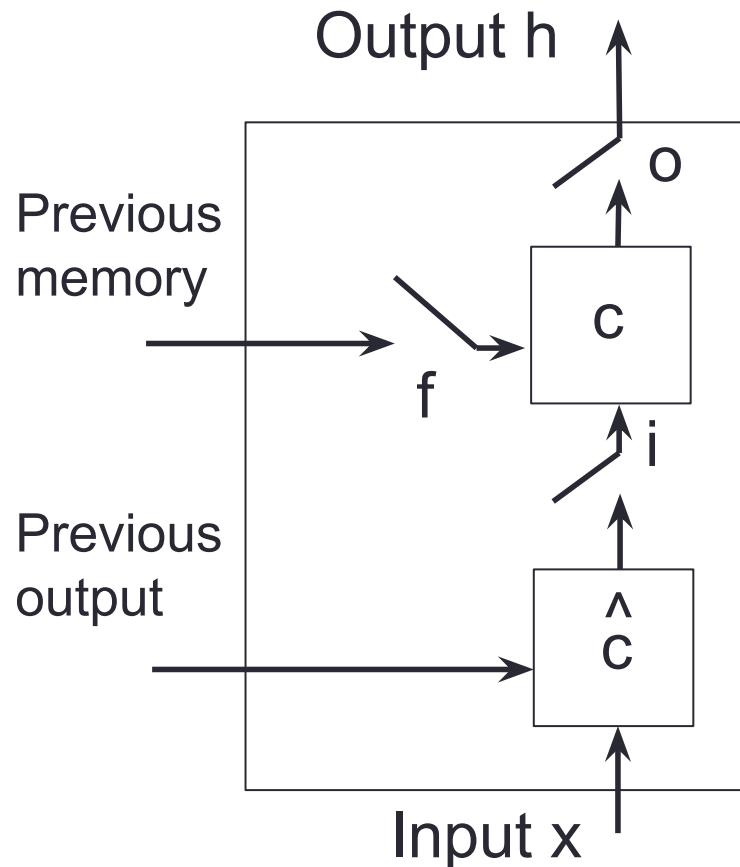


# Long Short-Term Memory (LSTM)

- Have 3 gates, forget ( $f$ ), input ( $i$ ), output ( $o$ )
- Has an **explicit memory cell** ( $c$ )
  - Does not have to output the memory



# Long Short-Term Memory (LSTM)

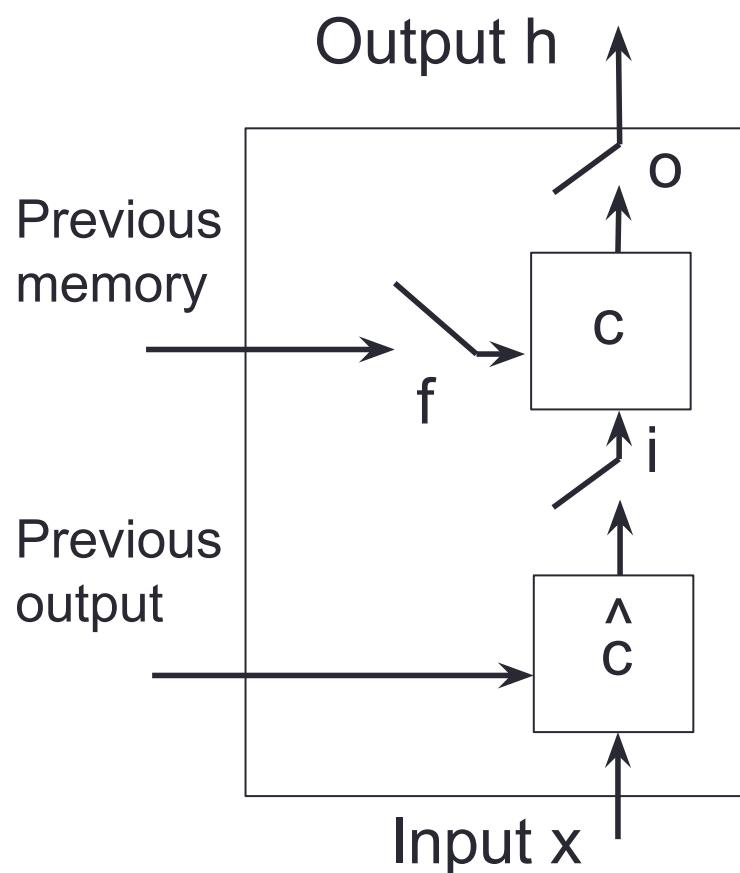


$$i_t^j = F^j(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + V_i \mathbf{c}_{t-1})$$
$$o_t^j = F^j(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + V_o \mathbf{c}_t)$$
$$f_t^j = F^j(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + V_j \mathbf{c}_{t-1})$$

Contribution from memory “Peephole connection”

Vs are diagonal matrices(Each cell can only see its own memory)

# Long Short-Term Memory (LSTM)



$$i_t^j = F^j(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + V_i \mathbf{c}_{t-1})$$

$$o_t^j = F^j(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + V_o \mathbf{c}_t)$$

$$f_t^j = F^j(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + V_j \mathbf{c}_{t-1})$$

$$h_t^j = o_t^j \tanh(c_t^j)$$

$$c_t^j = f_t^j c_{t-1}^j + i_t^j \hat{c}_t^j$$

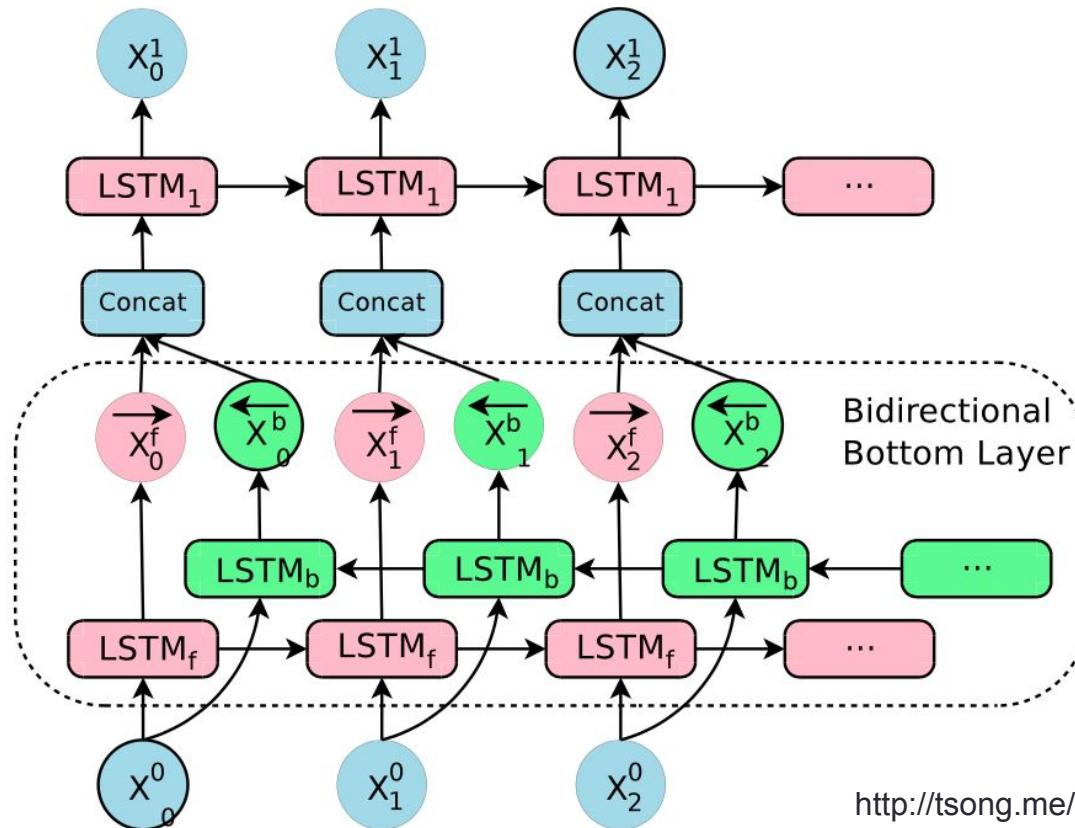
$$\hat{c}_t^j = \tanh^j(W_c \mathbf{x}_t + U_c \mathbf{h}_{t-1})$$

# GRU vs LSTM

- GRU and LSTM offers the same performance with large dataset
  - GRU better for smaller dataset (less parameters)
  - GRU faster to train and faster runtime (smaller model)
- Use GRUs!

# Bi-directional LSTM

- The previous GRU/LSTM only goes backward in time (uni-directional)
- Most of the time information from the future is useful for predicting the current output



# LSTM remembers meaningful things

Cell sensitive to position in line:

```
The sole importance of the crossing of the Berezina lies in the fact  
that it plainly and indubitably proved the fallacy of all the plans for  
cutting off the enemy's retreat and the soundness of the only possible  
line of action--the one Kutuzov and the general mass of the army  
demanded--namely, simply to follow the enemy up. The French crowd fled  
at a continually increasing speed and all its energy was directed to  
reaching its goal. It fled like a wounded animal and it was impossible  
to block its path. This was shown not so much by the arrangements it  
made for crossing as by what took place at the bridges. When the bridges  
broke down, unarmed soldiers, people from Moscow and women with children  
who were with the French transport, all--carried on by vis inertiae--  
pressed forward into boats and into the ice-covered water and did not,  
surrender.
```

Cell that turns on inside quotes:

```
"You mean to imply that I have nothing to eat out of.... On the  
contrary, I can supply you with everything even if you want to give  
dinner parties," warmly replied Chichagov, who tried by every word he  
spoke to prove his own rectitude and therefore imagined Kutuzov to be  
animated by the same desire.
```

```
Kutuzov, shrugging his shoulders, replied with his subtle penetrating  
smile: "I meant merely to say what I said."
```

# Embeddings

- A way to encode information to a lower dimensional space
  - PCA
  - We learn about this lower dimensional space through data

# One hot encoding

- Categorical representation is usually represented by **one hot encoding**
- Categorical representations examples:
  - Words in a vocabulary, characters in Thai language

Apple -> 1 -> [1, 0, 0, 0, ...]

Bird -> 2 -> [0, 1, 0, 0, ...]

Cat -> 3 -> [0, 0, 1, 0, ...]

- **Sparse** representation
  - Spare means most dimension are zero

# One hot encoding

- Sparse – but lots of dimension
  - Curse of dimensionality
- Does not represent meaning.

Apple -> 1 -> [1, 0, 0, 0, ...]

Bird -> 2 -> [0, 1, 0, 0, ...]

Cat -> 3 -> [0, 0, 1, 0, ...]

$$|\text{Apple} - \text{Bird}| = |\text{Bird} - \text{Cat}|$$

# Getting meaning into the feature vectors

- You can add back meanings by hand-crafted rules
- Old-school NLP is all about feature engineering
- Word segmentation example:
  - Cluster Numbers
  - Cluster letters
- Concatenate them
- 𠂇 = [0 0 0 0 1 0 0 0, 1, 0]
- 𠮩 = [0 0 0 1 0 0 0 0, 0, 1]
- 𠮾 = [1 0 0 0 0 0 0 0, 0, 2]
- Which rules to use?
  - Try as many as you can think of, and do feature selection or use models that can do feature selection

# Dense representation

- We can encode sparse representation into a lower dimensional space
  - $F: \mathbb{R}^N \rightarrow \mathbb{R}^M$ , where  $N > M$

Apple -> 1 -> [1, 0, 0, 0, ...] -> [2.3, 1.2]

Bird -> 2 -> [0, 1, 0, 0, ...] -> [-1.0, 2.4]

Cat -> 3 -> [0, 0, 1, 0, ...] -> [-3.0, 4.0]

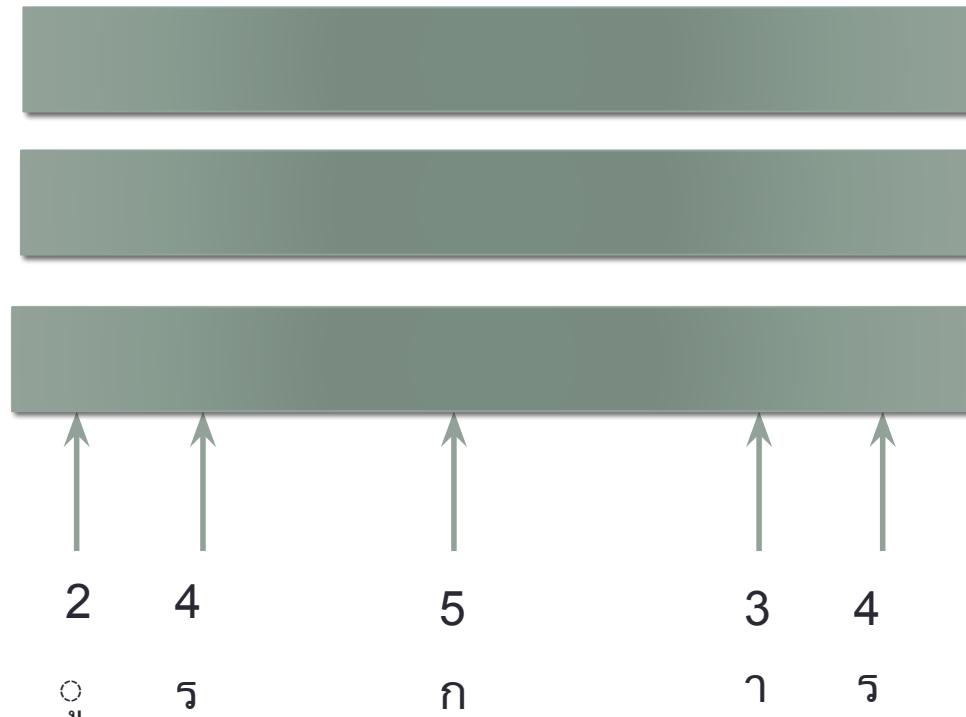
- We can do this by using an embedding layer

# Word segmentation with fully connected networks

1 = word beginning, 0 = word middle



Logistic function



# Adding embedding layer



Embedding layer  
shares the same  
weights



Parameter sharing!



[1, -1] [3, -2] [5.3, -2.1] [5.3, -3.1] [7.1, -2.1]

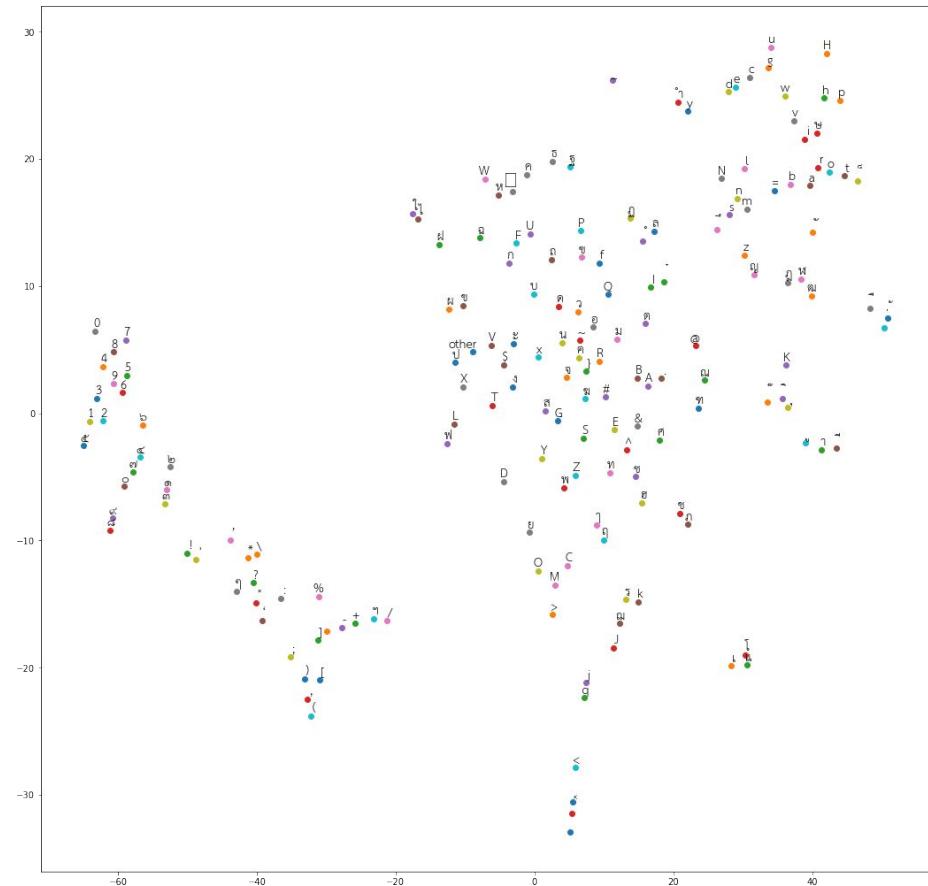


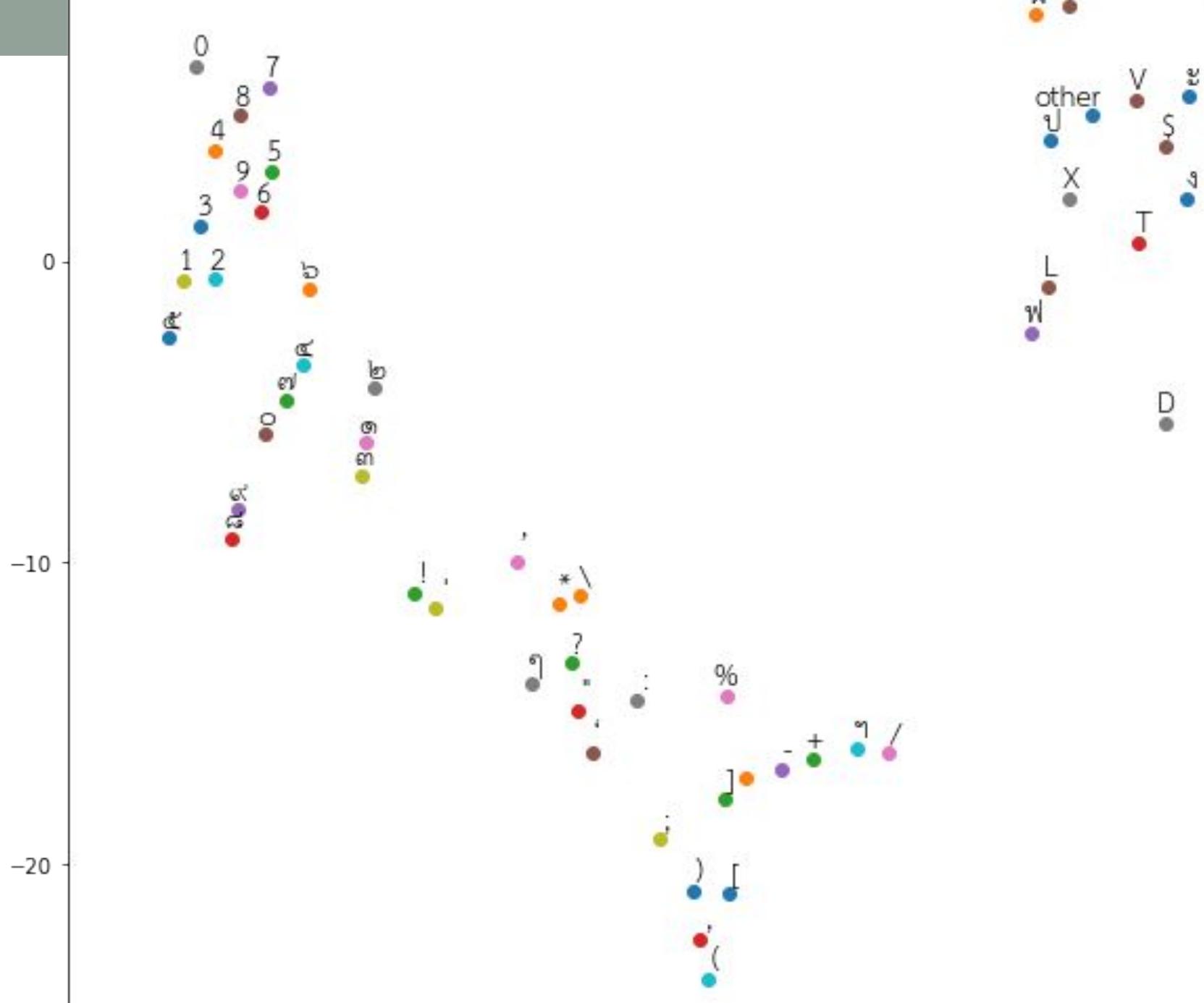
More on embeddings  
in the next two  
lectures!

2 4 5 3 4

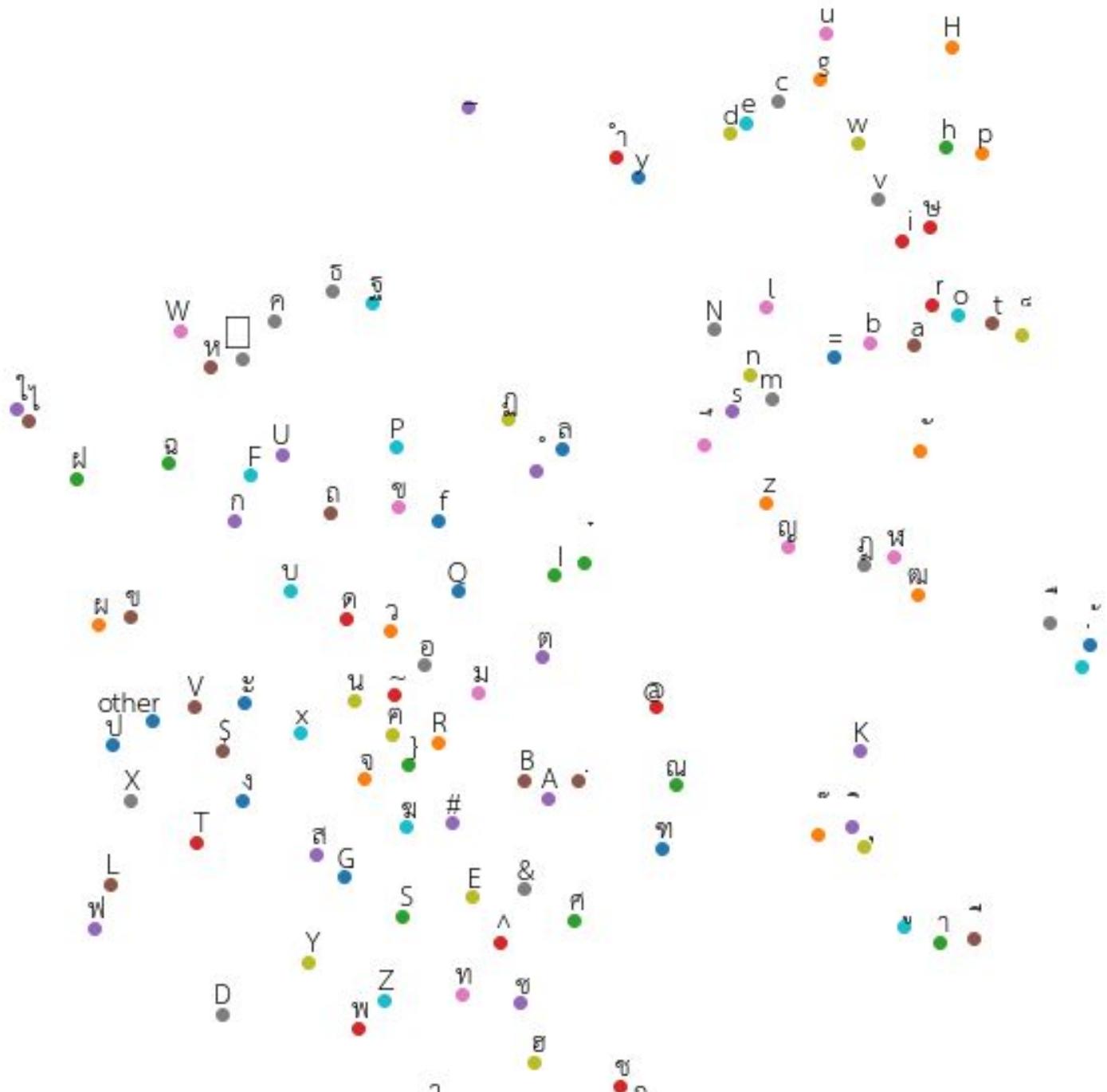
# Embedding and meaning (semantics)

- Meaning is inferred from the task
- Embedding of 32 dimensions -> t-SNE into 2 dimension for visualization
- Automatically!



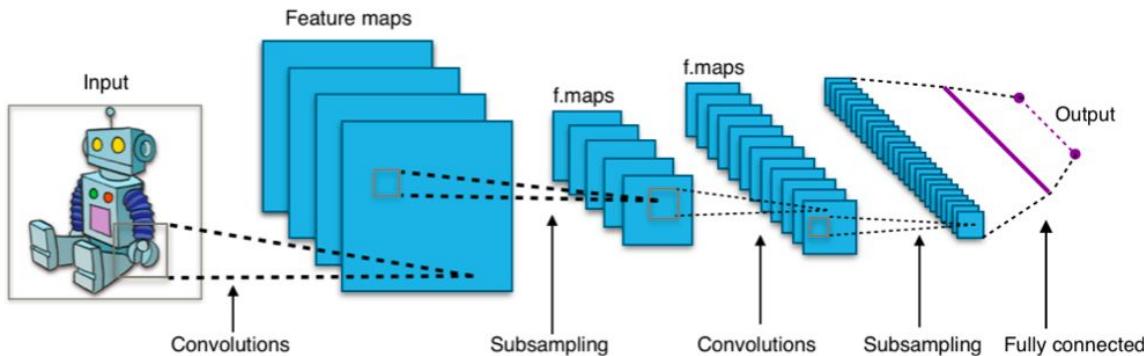
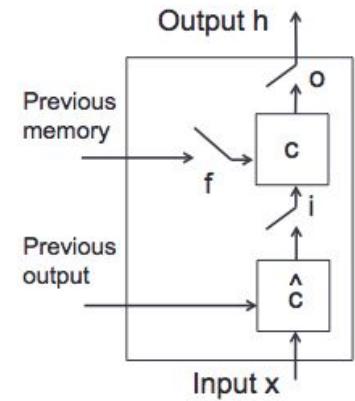
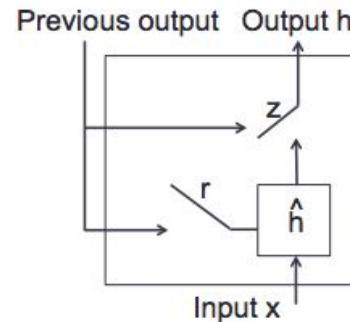






# Neural networks

- Fully connected networks
  - SGD, backprop
- CNN
- RNN, LSTM, GRU

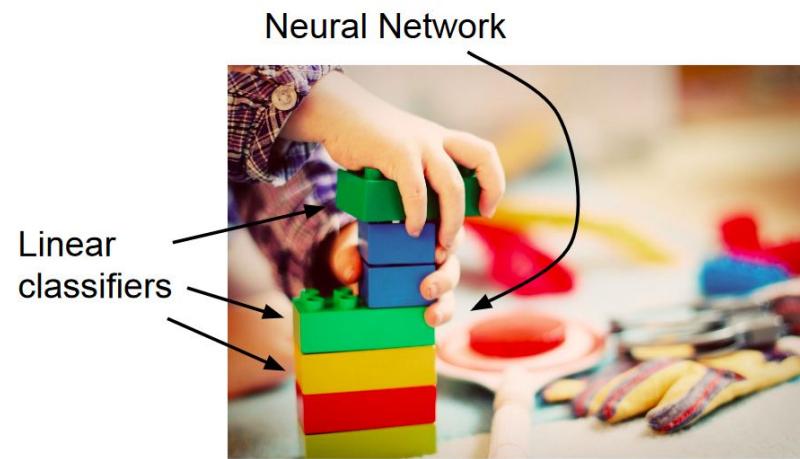
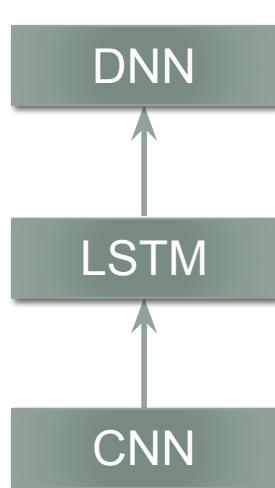


Attention modeling  
Recursive neural networks

← Future lectures

# DNN Legos

- Typical models now consists of all 3 types
  - CNN: local structure in the feature. Used for feature learning.
  - LSTM: remembering longer term structure or across time
  - DNN: Good for mapping features for classification. Usually used in final layers



# Back to tokenization...

TABLE II  
RESULTS OF THE SIX BEST TEAMS

Type of participants	F-Measure (%)	Time (mm:ss)
<i>Non-Students<sup>a</sup></i>	97.94937	00:47
<i>Non-Students</i>	97.84097	02:46
<i>Non-Students</i>	97.18822	00:26
<i>Bachelor Students<sup>b</sup></i>	95.78162	01:08
<i>Master Students</i>	95.56670	12:14
<i>PhD+Master Students</i>	92.02067	02:28

<sup>a</sup>Best of the BEST 2009 Award Winner

<sup>b</sup>BEST Student 2009 Award Winner

Pattern recognition course homework best F-score: 97.91

Sertis' model: 99.18

**BEST 2009 : Thai word segmentation software contest**

<http://ieeexplore.ieee.org/document/5340941/>

[https://sertiscorp.com/thai-word-segmentation-with-bi-directional\\_rnn/](https://sertiscorp.com/thai-word-segmentation-with-bi-directional_rnn/)

# Tokenization - English

- Even English has tokenization issues!
  - Space is usually not enough
  - aren't
    - are + n't
    - aren't
    - arent
    - aren t
    - are + not
  - San Francisco
- Usually includes the text normalization step
- Again depends on application
  - “aren't” might be different from “are not” for sentiment analysis

# Other English issues - hyphens

- “*the New York-based co-operative was fine-tuning forty-two K-9-like models.*”
- Lexical vs Sentential hyphens

# Tokenization of non-standard text

- Twitter

@SentimentSymp: can't wait for the Nov 9 #Sentiment talks! YAAAAAAAY!!! >:-D [http://sentimentsymposium.com/.](http://sentimentsymposium.com/)

Needs to correctly tokenize

Emoticons

Twitter markup (# and @)

Capitalization (and html tags for bold, etc.)

Lengthening

Dates

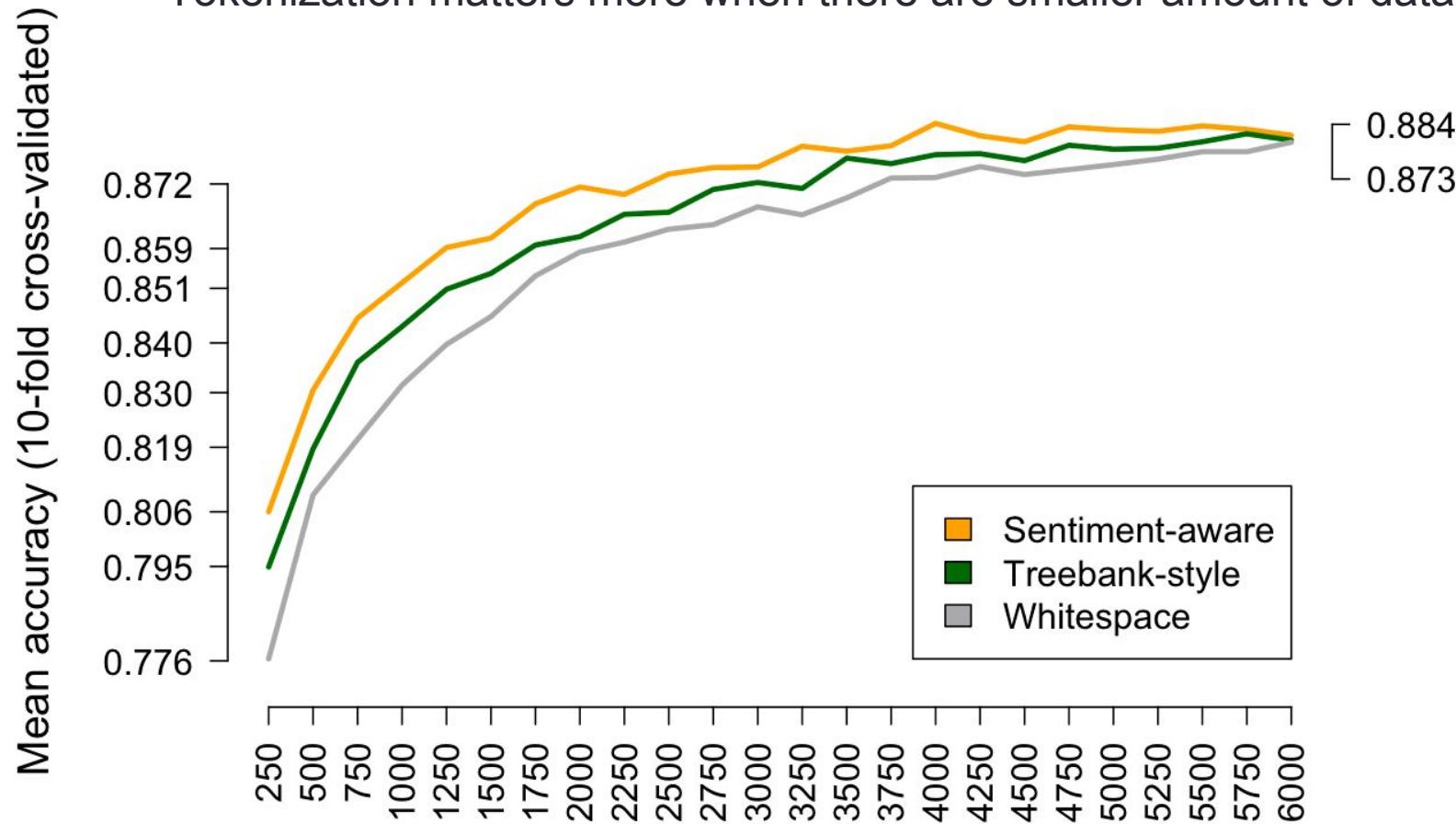
@  
SentimentSymp  
:  
ca  
n't Standard tokenizer  
wait (Stanford tokenizer)  
for  
the  
Nov  
9  
#  
Sentiment  
talks  
!  
YAAAAAAAY  
!  
!  
!  
&gt;  
;  
:  
-D  
http  
:  
//sentimentsymposium.com/  
.

## Hand-crafted for tweets

```
@sentimentsymp
:
can't
wait
for
the
Nov_09
#sentiment
talks
!
YAAAAY
!
!
!
>:-D
http://sentimentsymposium.com/
.
```

# Sentiment analysis (in-domain)

Tokenization matters more when there are smaller amount of data



# Sentiment analysis (out-of-domain)

Better tokenizer gives better portability



# End-to-end models

- Classical machine learning systems usually break the problem into smaller subtasks
  - Self-driving:
    - Image -> objects detection -> path finding -> steering
  - Speech2speech translation:
    - Speech A -> text A -> text B -> Speech B
- End-to-end models use one large neural networks process the input and generate the desired output
  - Image -> steering
  - Speech A -> Speech B

# End-to-end NLP?

Discourse

Semantics

CommunicationEvent(e)  
 Agent(e, Alice)  
 Recipient(e, Bob)  
 SpeakerContext(s)  
 TemporalBefore(e, s)

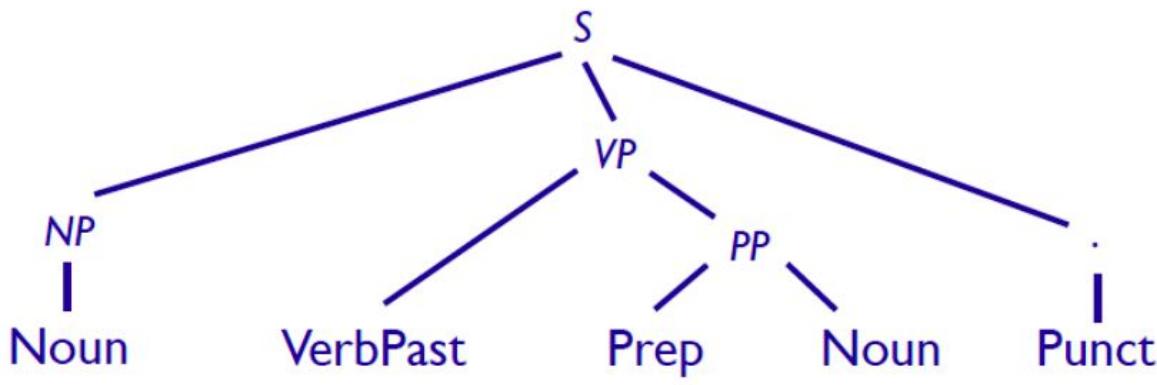
Syntax: Constituents

Syntax: Part of Speech

Words

Morphology

Characters



Alice talked to Bob.

talk -ed [VerbPast]

Alice talked to Bob.

# Towards no tokenization? :Character-aware neural language models

- Input: previous characters
- Output: next word

CNN over characters capture character sequence patterns

Fully connected

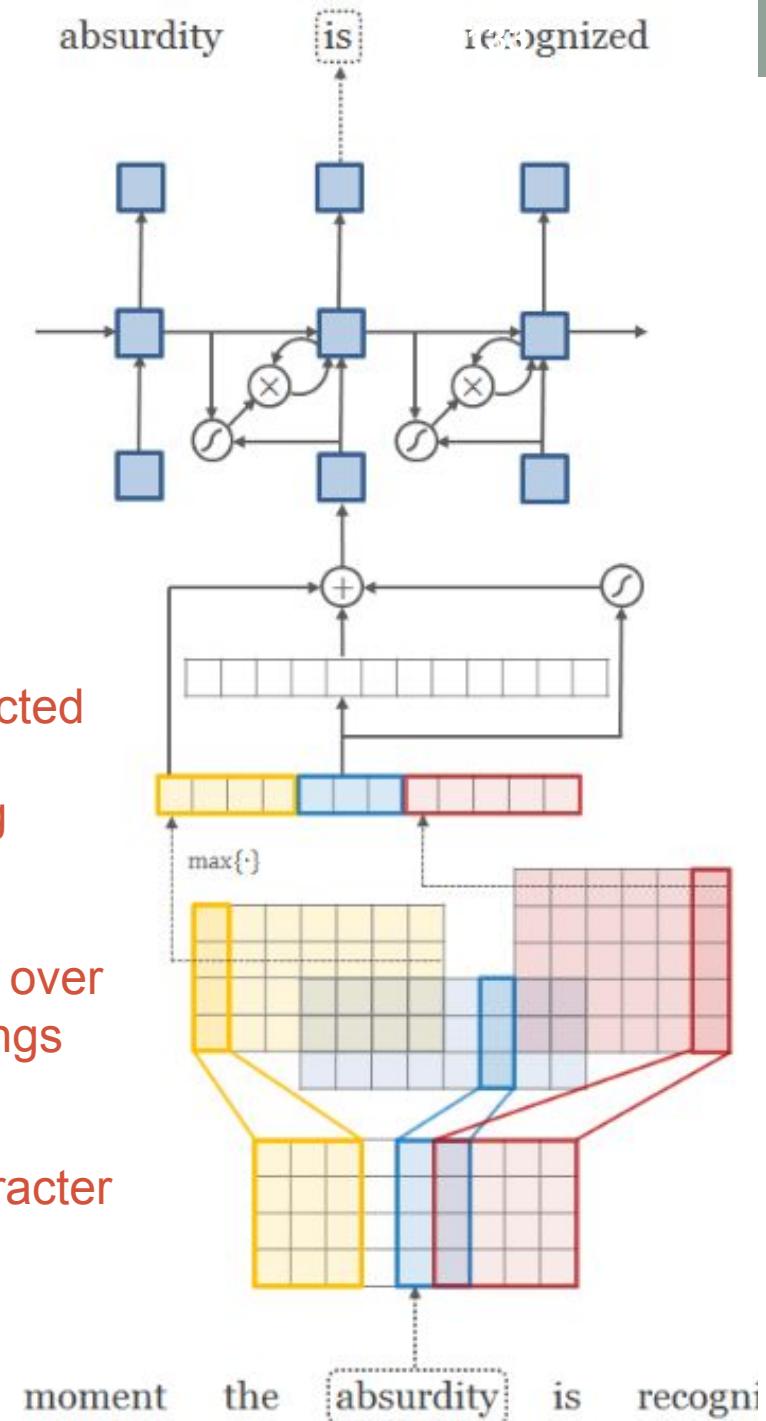
LSTM

Fully connected

Max pooling

Convolutional layer over character embeddings

Character to character embeddings

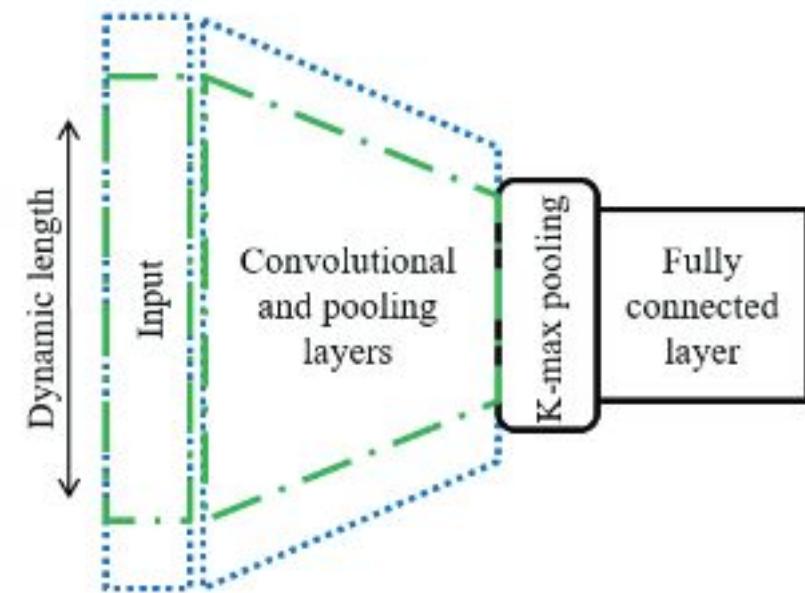


# Towards no tokenization

- Text classification using charCNN on Thai

Method	Accuracy (%)	$F_1$ (%)
Naïve Bayes, BoW	87.2	87.1
Naïve Bayes, TF-IDF	89.0	88.9
Logistic Regression, BoW	94.8	94.8
Logistic Regression, TF-IDF	94.7	94.7
SVM, BoW	93.7	93.7
SVM, TF-IDF	95.2	95.2
DCNN (Kalchbrenner et al., 2014)	<b>95.9</b>	<b>95.9</b>
Proposed Char-CNN	95.4	95.4

Word-based methods



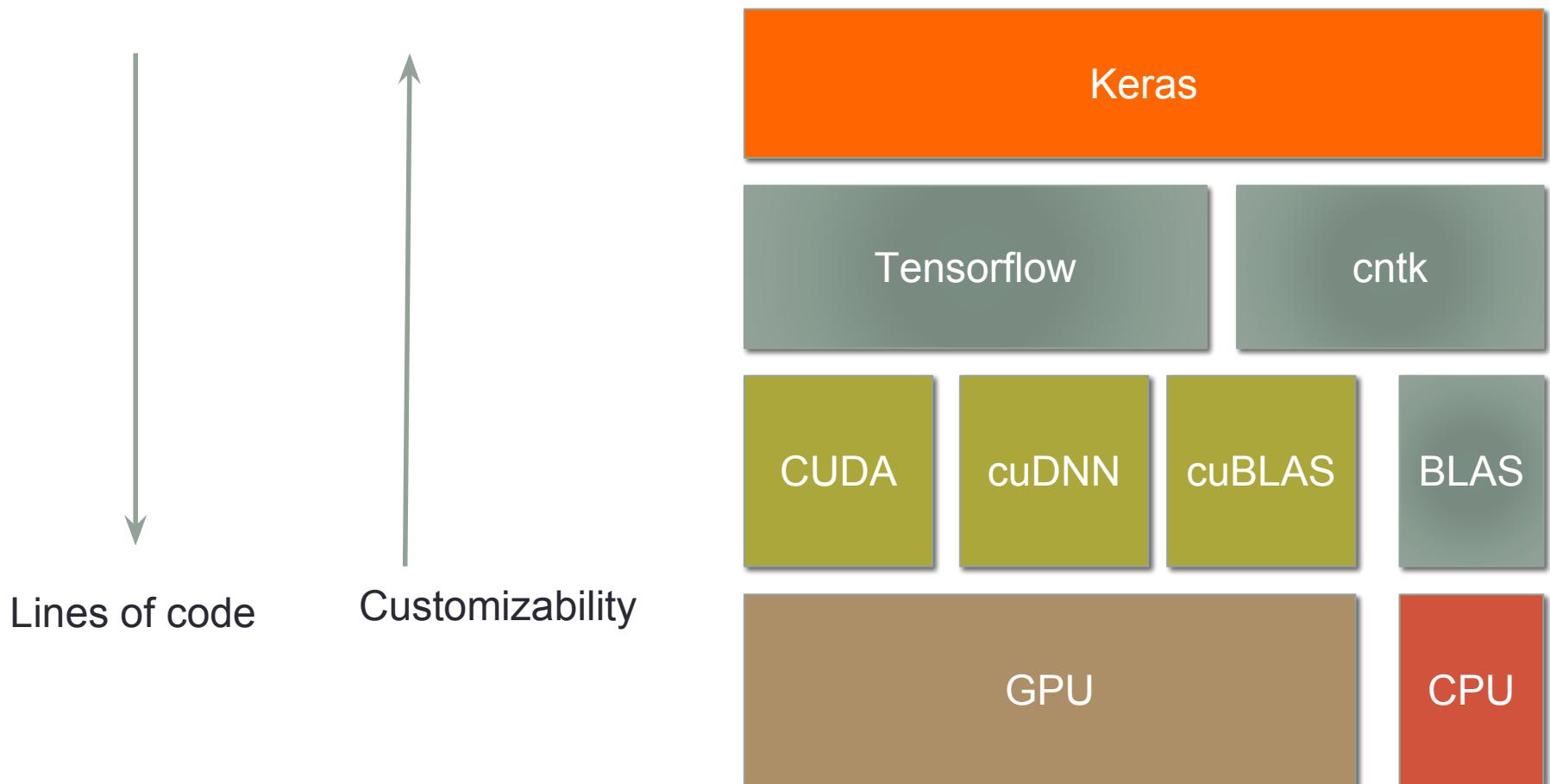
A character-level convolutional neural network with dynamic input length for Thai text categorization  
<http://ieeexplore.ieee.org/document/7886102/>

# Caveats of end-to-end models

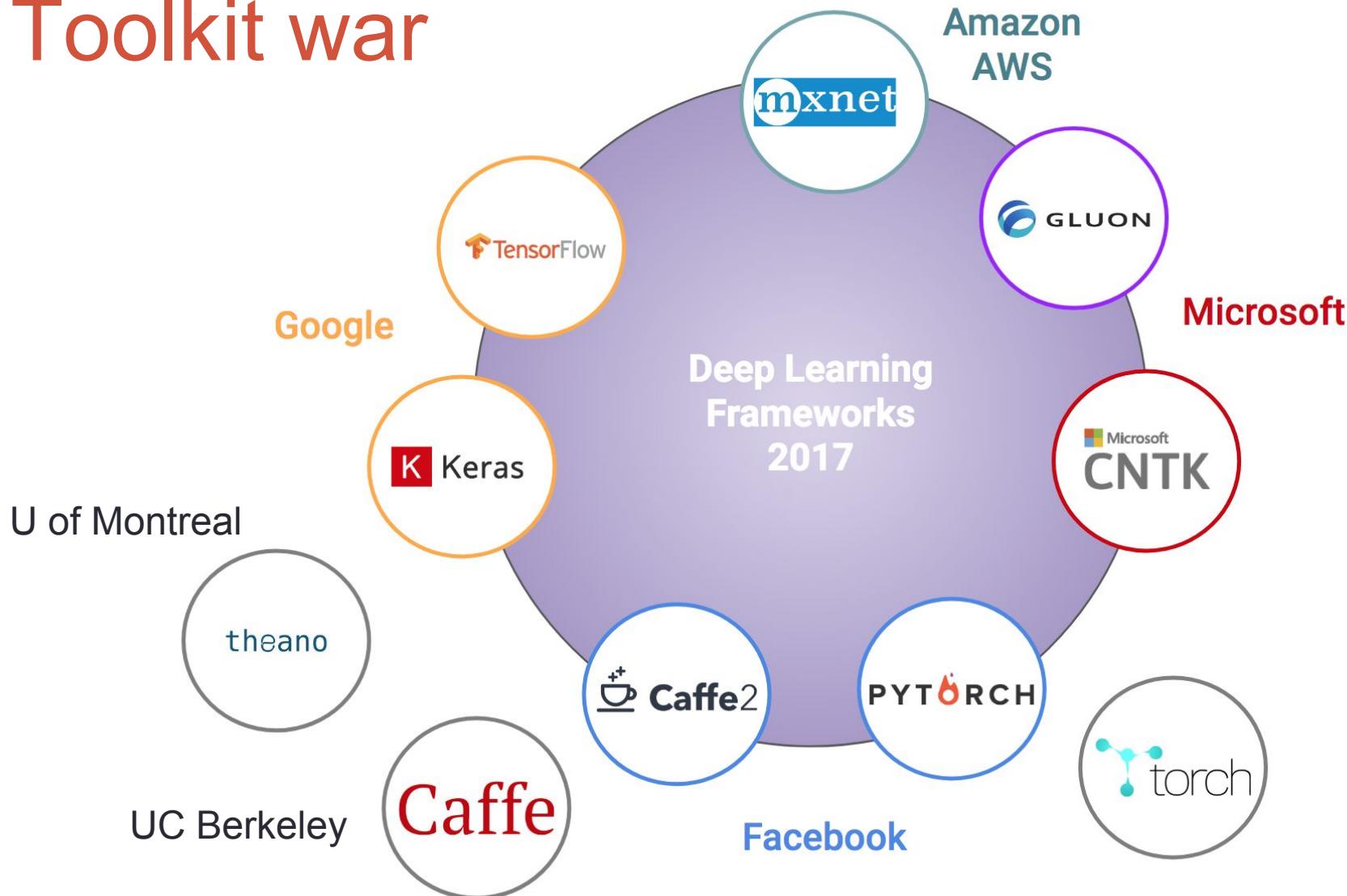
- Requires lots of data for the specific task
- Traditional model can train each sub-task on old data
  - Speech A -> text A -> text B -> Speech B
  - Speech A -> Speech B
- Speech A -> Speech C?

# What toolkit

- Tensorflow – lower level
- Keras – higher level



# Toolkit war



# Which?

- Easiest to use and play with deep learning: Keras
- Easiest to use and tweak: pytorch

A graph is created on the fly

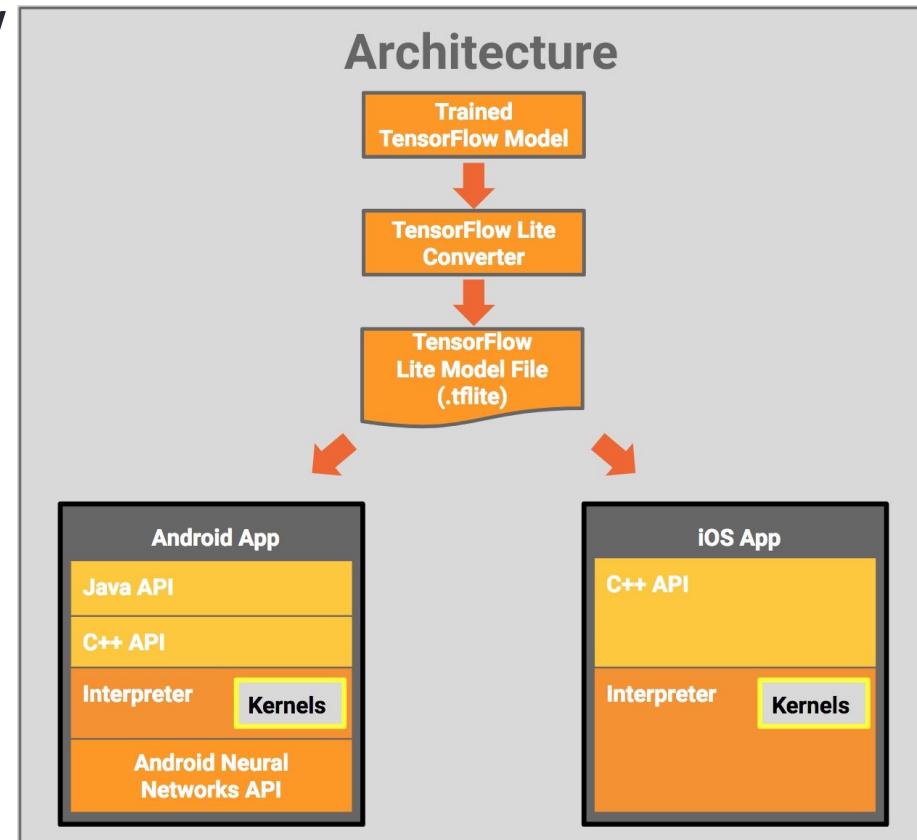
```
from torch.autograd import Variable  
  
x = Variable(torch.randn(1, 10))  
prev_h = Variable(torch.randn(1, 20))  
W_h = Variable(torch.randn(20, 20))  
W_x = Variable(torch.randn(20, 10))
```



Dynamic computation graph in pytorch. Good for when size of and structure of input data varies. Example parse tree.

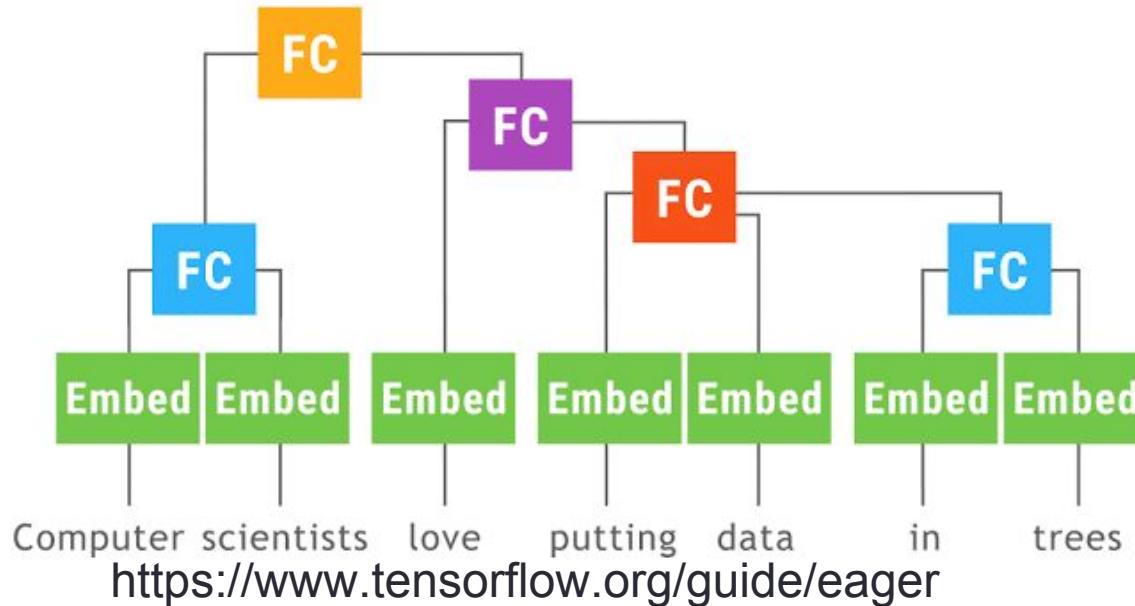
# Which?

- Easiest to use and play with deep learning: Keras
- Easiest to use and tweak: pytorch
- Easiest to deploy: tensorflow
  - Tensorflow lite for mobile



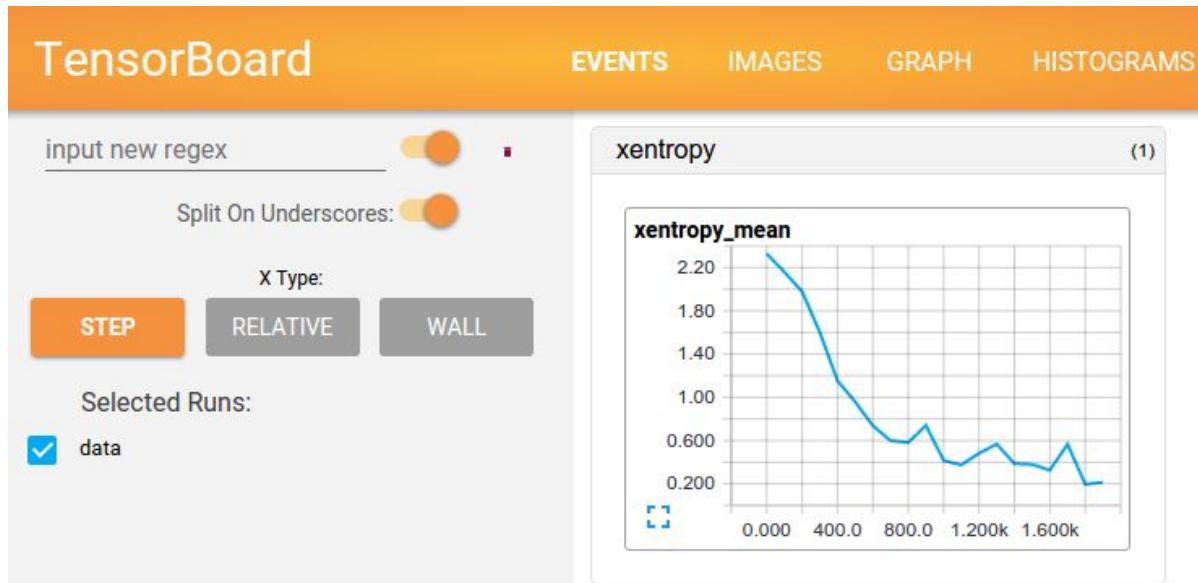
# Which?

- Easiest to use and play with deep learning: Keras
- Easiest to use and tweak: PyTorch
- Easiest to deploy: TensorFlow
  - TensorFlow Lite for mobile
  - Now support dynamic graph: eager execution mode



# Which?

- Easiest to use and play with deep learning: Keras
- Easiest to use and tweak: PyTorch
- Easiest to deploy: TensorFlow
- Best tools: TensorFlow Tensorboard visualization



[https://www.tensorflow.org/get\\_started/summaries\\_and\\_tensorboard](https://www.tensorflow.org/get_started/summaries_and_tensorboard)

# Which?

- Easiest to use and play with deep learning: Keras
- Easiest to use and tweak: PyTorch
- Easiest to deploy: TensorFlow
  - TensorFlow Lite for mobile
  - Now support dynamic graph: eager execution mode
- Biggest community: TensorFlow
  - Github repositories & pre-trained models
  - Pytorch community growing fast
- TPUs: Google

README.md



# ONNX

Linux	Windows
<a href="#">build</a> <span>passing</span>	<a href="#">build</a> <span>passing</span>

[Open Neural Network Exchange \(ONNX\)](#) is the first step toward an open ecosystem that empowers AI developers to choose the right tools as their project evolves. ONNX provides an open source format for AI models. It defines an extensible computation graph model, as well as definitions of built-in operators and standard data types. Initially we focus on the capabilities needed for inferencing (evaluation).

[Caffe2](#), [PyTorch](#), [Microsoft Cognitive Toolkit](#), [Apache MXNet](#) and other tools are developing ONNX support. Enabling interoperability between different frameworks and streamlining the path from research to production will increase the speed of innovation in the AI community. We are an early stage and we invite the community to submit feedback and help us further evolve ONNX.

<https://github.com/onnx/onnx>

# Sidenote: Tensor?

- 0<sup>th</sup> order tensor – scalar
- 1<sup>st</sup> order tensor – vector
- 2<sup>nd</sup> order tensor – matrix
- Think of tensors as multi-dimensional arrays

# Demos

- Keras
- You should read before doing the homework
  - <https://keras.io/getting-started/functional-api-guide/>

# Keras steps

- Define the network
  - Compile the network
  - Fit the network

```
def get_feedforward_nn():
    input1 = Input(shape=(21,))
    x = Dense(100, activation='relu')(input1)
    x = Dense(100, activation='relu')(x)
    x = Dense(100, activation='relu')(x)
    out = Dense(1, activation='sigmoid')(x)

    model = Model(inputs=input1, outputs=out)
    model.compile(optimizer=Adam(),
                  loss='binary_crossentropy',
                  metrics=['acc'])

    return model
```

# Keras is easy!

## Dense

[\[source\]](#)

```
keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros')
```

Just your regular densely-connected NN layer.

`Dense` implements the operation: `output = activation(dot(input, kernel) + bias)` where `activation` is the element-wise activation function passed as the `activation` argument, `kernel` is a weights matrix created by the layer, and `bias` is a bias vector created by the layer (only applicable if `use_bias` is `True` ).

- **Note:** if the input to the layer has a rank greater than 2, then it is flattened prior to the initial dot product with `kernel`.

## Example

```
# as first layer in a sequential model:
model = Sequential()
model.add(Dense(32, input_shape=(16,)))
# now the model will take as input arrays of shape (*, 16)
# and output arrays of shape (*, 32)

# after the first layer, you don't need to specify
# the size of the input anymore:
model.add(Dense(32))
```

## Dropout

[\[source\]](#)

```
keras.layers.Dropout(rate, noise_shape=None, seed=None)
```

Applies Dropout to the input.

Dropout consists in randomly setting a fraction `rate` of input units to 0 at each update during training time, which helps prevent overfitting.

### Arguments

- `rate`: float between 0 and 1. Fraction of the input units to drop.
- `noise_shape`: 1D integer tensor representing the shape of the binary dropout mask that will be multiplied with the input.  
For instance, if your inputs have shape `(batch_size, timesteps, features)` and you want the dropout mask to be the same for all timesteps, you can use `noise_shape=(batch_size, 1, features)`.
- `seed`: A Python integer to use as random seed.

**Conv1D**

## Number of filters

[\[source\]](#)

```
keras.layers.Conv1D(filters, kernel_size, strides=1, padding='valid', dilation_rate=1, activation=None, use_b:
```

### Size of filter

1D convolution layer (e.g. temporal convolution).

This layer creates a convolution kernel that is convolved with the layer input over a single spatial (or temporal) dimension to produce a tensor of outputs. If `use_bias` is True, a bias vector is created and added to the outputs. Finally, if `activation` is not `None`, it is applied to the outputs as well.

When using this layer as the first layer in a model, provide an `input_shape` argument (tuple of integers or `None`), e.g. `(10, 128)` for sequences of 10 vectors of 128-dimensional vectors, or `(None, 128)` for variable-length sequences of 128-dimensional vectors.

## Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel\_size**: An integer or tuple/list of a single integer, specifying the length of the 1D convolution window.
- **strides**: An integer or tuple/list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: One of `"valid"`, `"causal"` or `"same"` (case-insensitive). `"valid"` means "no padding". `"same"` results in padding the input such that the output has the same length as the original input. `"causal"` results in causal (dilated) convolutions, e.g. `output[t]` does not depend on `input[t+1:]`. Useful when modeling temporal data where the model should not violate the temporal order. See [WaveNet: A Generative Model for Raw Audio](#), section 2.1.

# Deep learning demo

Red boundaries Indicate output and input tensors

