Language Understanding

02 - Introduction to Neural Networks

Hossein Zeinali

Agenda

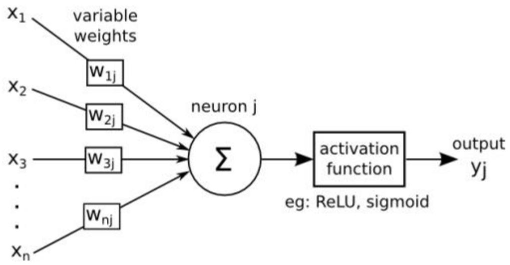
- Perceptron
- Feedforward Neural Network
- Backpropagation
- Recurrent Neural Network
  - Long Short-Term Memory
  - Gated Recurrent Units
- Convolutional Neural Network

Neural Networks

- Fundamental computational tool
- Called neural because their origins lie in the human neuron
- Neural network is a network of small computing units
- **Universal approximation theorem**:
  - Hornik (1991) showed that any bounded and regular function $\mathbb{R}^d \rightarrow \mathbb{R}$ can be approximated at any given precision by a neural network with one hidden layer containing a finite number of neurons, having the same activation function, and one linear output neuron.
    - This result was earlier proved by Cybenko (1989) in the particular case of the sigmoid activation function.
- This theorem is interesting from a theoretical point of view.
  - From a practical point of view, this is not really useful since the number of neurons in the hidden layer may be very large.
  - The strength of deep learning lies in the deep (number of hidden layers) of the networks.

Neuron (Computational Unit)

- The building block of a neural network
  - Takes a set of real valued numbers as input
  - Performs some computation on them
  - Finally produces an output



- Given a set of inputs $x_1, \dots, x_n$, corresponding weights $w_1, \dots, w_n$ and a bias b, so the weighted sum z can be represented as:

$$z = b + \sum_i w_i x_i$$

- And in vector notation:

$$z = \boldsymbol{w} \cdot \boldsymbol{x} + b$$

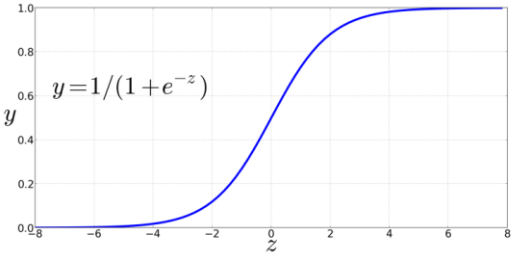
- The output of the neuron is calculated by applying a non-linear function f to z

$$y = f(z)$$

Non-linear (activation) Functions

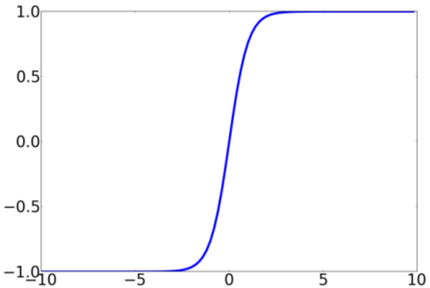
- **Sigmoid:**

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



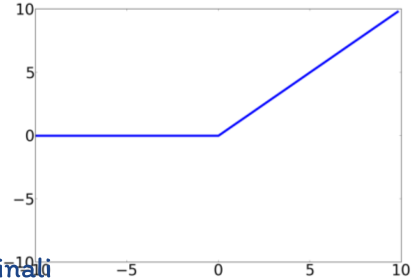
- **Tanh:**

$$y = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$



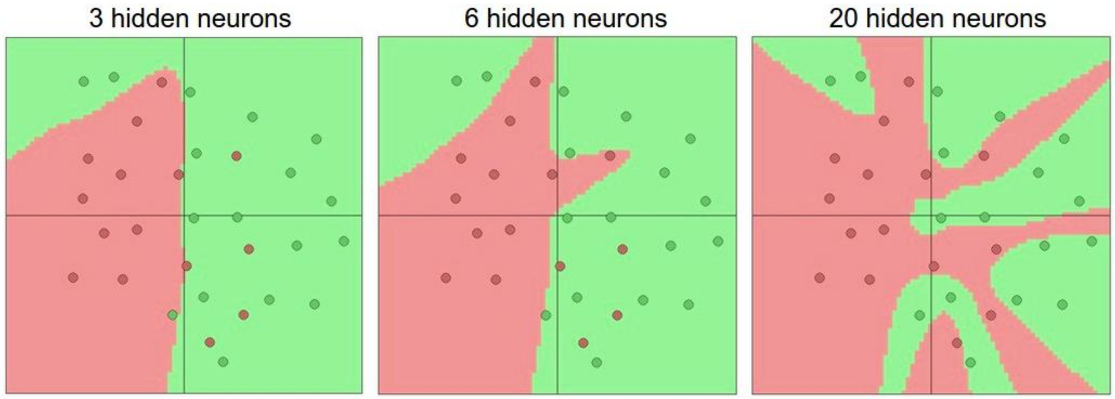
- Rectified linear unit **(ReLU)**:

$$y = \max(z, 0)$$



Non-linear (activation) Functions

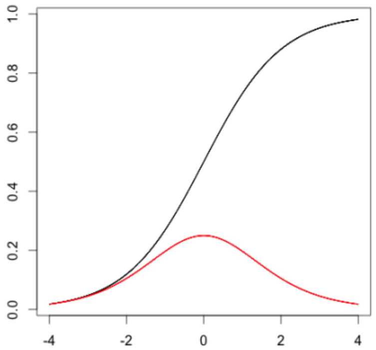
- Non-linearities needed to learn complex (non-linear) representations of data, otherwise the NN would be just a linear function $W_1 W_2 x = Wx$
- More layers and neurons can approximate more complex functions



http://cs231n.github.io/assets/nn1/layer_sizes.jpeg

Non-linear (activation) Functions

- Historically, the sigmoid was the mostly used activation function
  - It is differentiable and allows to keep values in the interval $[0, 1]$.
  - Nevertheless, it is problematic since its gradient is very close to 0 when $|x|$ is not close to 0.
  - With neural networks with a high number of layers, this causes troubles for the backpropagation.
    - This is why the sigmoid function was supplanted by the rectified linear function.



- The ReLU function also has a sparsification effect.
  - The ReLU function and its derivative are equal to 0 for negative values, and no information can be obtain in this case for such a unit.
- Leaky ReLU: $$\varphi(x) = \max(x, 0) + \alpha \min(x, 0)$$

Perceptron, AND, OR, and XOR

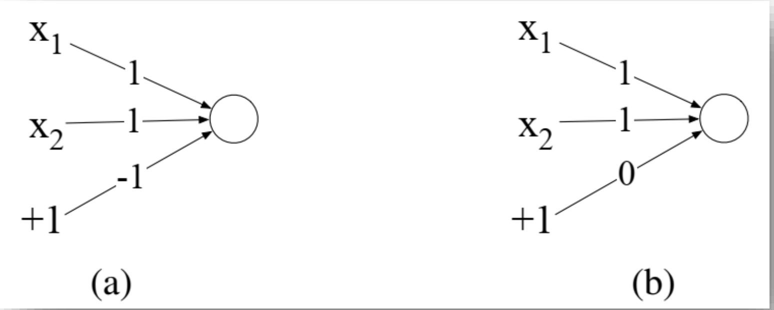
- **Perceptron**: a very simple neural unit that has a binary output and does **not** have a non-linear activation function.

$$y = \begin{cases} 1, & if\ \boldsymbol{w} \cdot \boldsymbol{x} + b > 0 \\ 0, & if\ \boldsymbol{w} \cdot \boldsymbol{x} + b \leq 0 \end{cases}$$

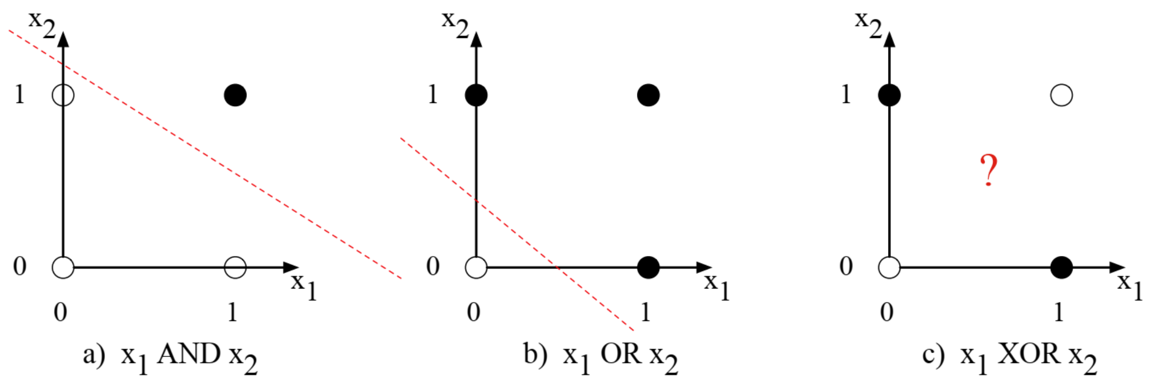
AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

Perceptron, AND, OR, and XOR

- AND/OR

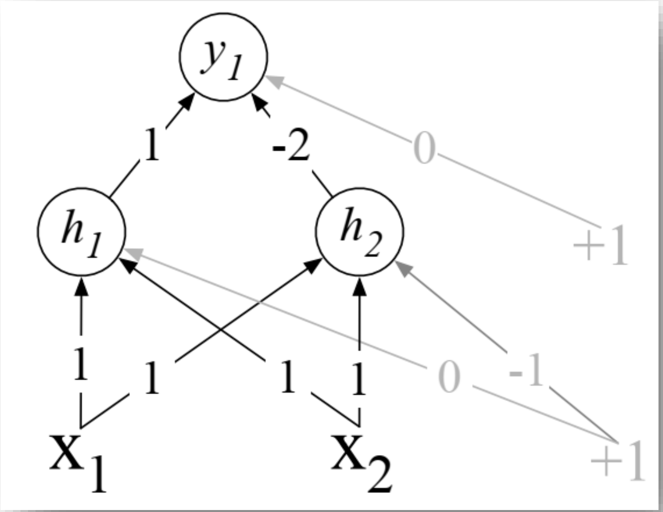


- It is not possible to build a perceptron to compute logical XOR!



Perceptron, AND, OR, and XOR

- The XOR function can be calculated by a layered network of units.
- Solution with three ReLU units (note: linear units cannot solve the problem):



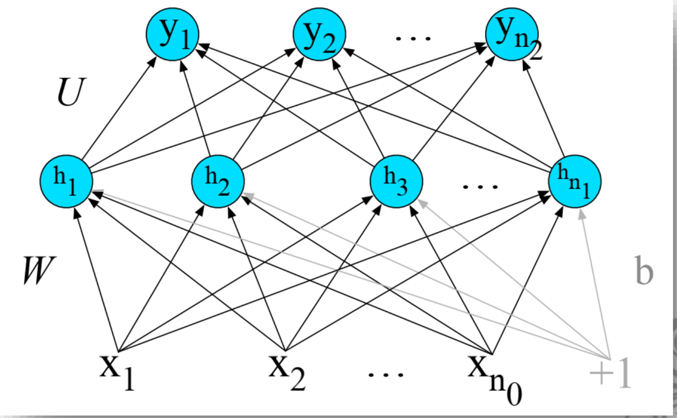
- A network formed by many layers of purely linear units can always be reduced to a single layer of linear units with appropriate weights.

Feed-Forward Neural Networks

- Is a multilayer network in which the units are connected with no cycles
  - Outputs from units in each layer are passed to the next higher layer.
  - No outputs are passed back to lower layers.
- Also called multi-layer perceptrons (or MLPs)
- A simple **fully-connected** example:

$$h = g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\boldsymbol{W} \in \mathbb{R}^{n_1 \times n_0}\text{, and } \boldsymbol{b} \in \mathbb{R}^{n_1}$$



Feed-Forward Neural Networks

- For the output layer, the activation function is generally different from the one used on the hidden layers.
- The number of outputs is depend on the application
  - In the case of regression, we apply no activation function on the output layer.
  - One output for binary classification: **sigmoid** can be used
  - Multimodal classification: **softmax** can be used
- The **softmax** function:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}} \qquad 1 \leq i \leq d$$

- Softmax converts real-valued numbers (logits) to a probability distribution.
- A neural network classifier can be divided to:
  - A representation network to convert input to h (last hidden layer)
  - Running standard logistic regression on top of hidden representation h

Feed-Forward Neural Networks

- Notations:
  - Use superscripts in square brackets to mean layer numbers, starting at 0 for the input layer. We have: $\boldsymbol{W}^{[i]}, \boldsymbol{b}^{[i]}$
  - n_i means the number of units at layer i
  - Use $g(\cdot)$ to stand for the activation function
  - Use $\boldsymbol{a}^{[i]}$ to mean the output from layer i, and $z^{[i]}$ to mean the combination of weights and biases $\boldsymbol{W}^{[i]}\boldsymbol{a}^{[i-1]} + \boldsymbol{b}^{[i]}$.
  - The 0th layer is for inputs, so the inputs x we'll referred as $a^{[0]}$.
- The algorithm for computing the forward step:

for i **in** 1..n

$$z^{[i]} = W^{[i]} a^{[i-1]} + b^{[i]}$$

$$a^{[i]} = g^{[i]}(z^{[i]})$$

$$\hat{y} = a^{[n]}$$

Training Neural Nets

Introduction

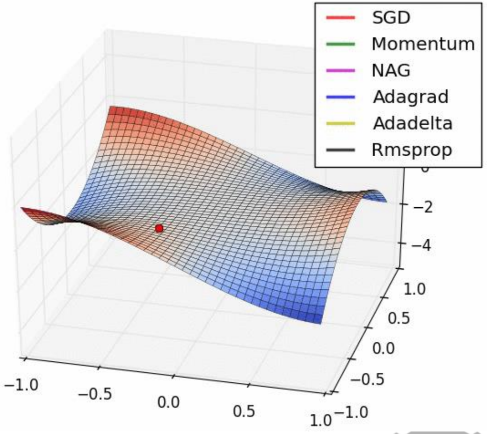
- The goal of the training procedure:
  - Learn parameters $\boldsymbol{W}^{[i]}$ and $\boldsymbol{b}^{[i]}$ for each layer i
  - Make $\hat{y}$ for each training observation as close as possible to the true y.

- Requirements:
  - Loss function: i.e. cross-entropy loss
  - Optimization algorithm: i.e. gradient descent
  - Error backpropagation

Training

Sample labeled data (**batch**) → **Forward** it through the network, get predictions → **Back-propagate** the errors → **Update** the network weights

- Optimize (min. or max.) **objective/cost function**
- Generate **error signal** that measures difference between predictions and target values
- Use error signal to change the **weights** and get more accurate predictions
- Subtracting a fraction of the **gradient** moves you towards the **(local) minimum of the cost function**



Loss Function

- Models the distance between the system output and the gold output

$$L(\hat{y}, y) = \text{How much } \hat{y} \text{ differs from the true } y$$

- It is classical to estimate the parameters by maximizing the likelihood (or equivalently the logarithm of the likelihood).
  - This corresponds to the minimization of the loss function which is the opposite of the log likelihood.
- We are looking for a function that prefers the correct class labels of the training examples to be *more likely*.
  - This is called **conditional maximum likelihood estimation.**
- The resulting loss function is the negative log likelihood loss, generally called the **cross-entropy loss.**

Cross-Entropy Loss

- For a binary classifier, with the sigmoid at the final layer:

$$L_{CE}\,(\hat{y}, y) \;=\; -\log p(y|x) = -[\mathrm{y} \log \hat{y} + \;(1-y)\log(1-\hat{y})]$$

- In a multinomial classifier with C classes:

$$L_{CE}\,(\hat{y}, y) \;=\; -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

- In case of **hard classification** task, y is a **one-hot vector.**

$$L_{CE}\,(\hat{y}, y) \;=\; -\log \hat{y}_i$$

Loss Functions and Output

Classification

Training examples

R^n x {class_1, …, class_n}
(one-hot encoding)

Output Layer

Soft-max
[map R^n to a probability distribution]

$$P(y=j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^{\mathsf{T}}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^{\mathsf{T}}\mathbf{w}_k}}$$

Cost (loss) function

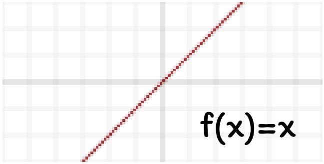
Cross-entropy

$$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}\left[y_k^{(i)} \log \hat{y}_k^{(i)} + \left(1 - y_k^{(i)}\right) \log\left(1 - \hat{y}_k^{(i)}\right)\right]$$

Regression

R^n x R^m

Linear (Identity) or Sigmoid



Mean Squared Error

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Mean Absolute Error

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left|y^{(i)} - \hat{y}^{(i)}\right|$$

Gradient Descent

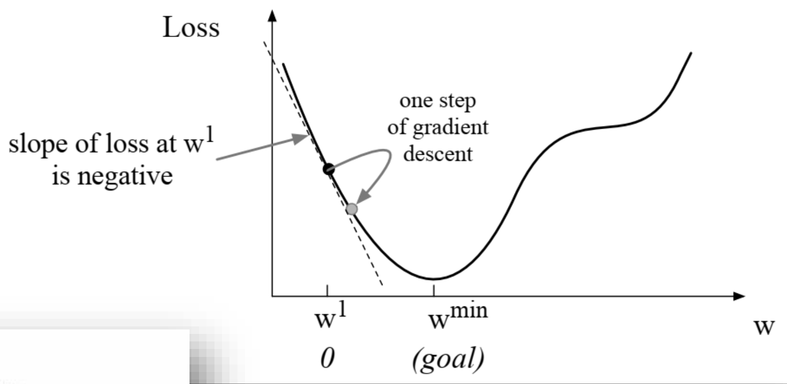
- Find the optimal weights by minimizing the defined loss function

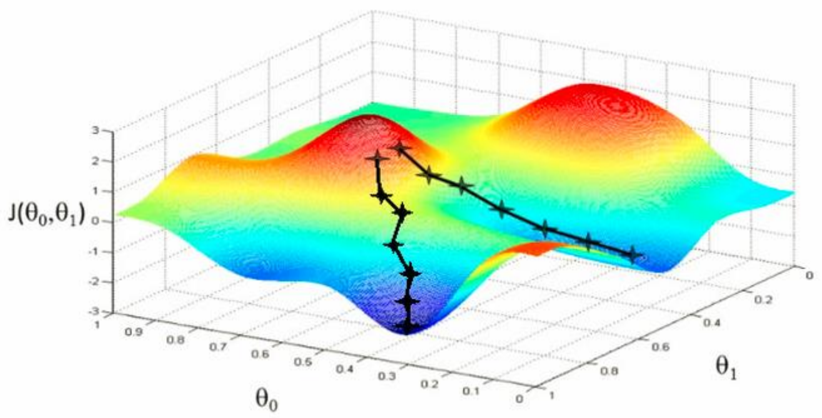
$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i}^{m} L_{CE}(y^{(i)}, x^{(i)}; \theta)$$

- Gradient descent
  - A method that finds a minimum of a function by figuring out in which direction the function's slope is rising the most steeply, and moving in the opposite direction.

Gradient Descent

- Convex vs Non-convex





The gradient of a function of many variables is a vector pointing in the direction of the greatest increase in a function.

Gradient Descent

- Update formula for single variable:

$$w^{t+1} = w^{t} - \eta \frac{d}{dw} L(f(x; w), y)$$

where η is **learning rate.**

- In multivariable form:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^{t} - \eta \nabla L(f(x; \boldsymbol{\theta}), y)$$

- For each variable in $\boldsymbol{\theta}$, the gradient will have a component that tells us the slope with respect to that variable.

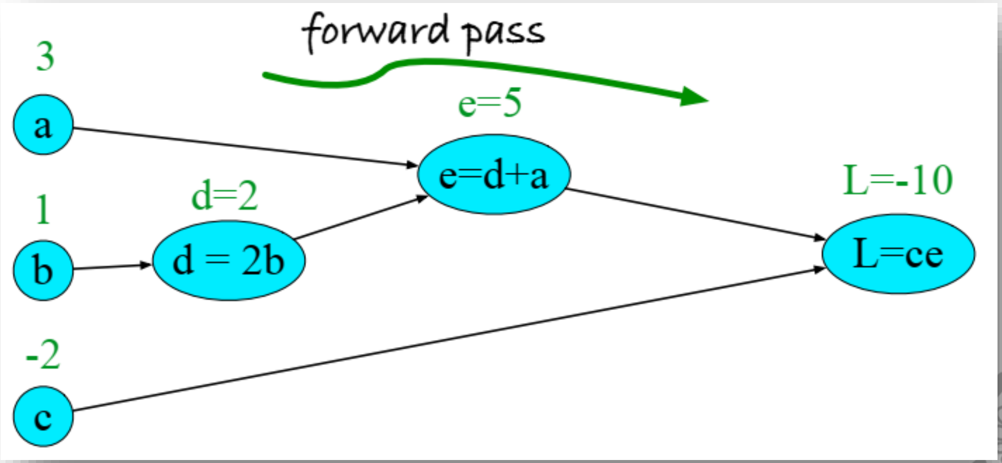
$$\nabla L(f(x; \boldsymbol{\theta}), y) = \left[\frac{\partial}{\partial w_1} L(f(x; \boldsymbol{\theta}), y), \dots, \frac{\partial}{\partial w_n} L(f(x; \boldsymbol{\theta}), y)\right]^{t}$$

Computing the Gradient

- Requires the partial derivative of the loss function with respect to each parameter.
- For a network with one weight layer, we could simply use the derivative of the loss.
    - These derivatives only give correct updates for the last weight layer
- The solution is **error backpropagation** algorithm

Computation Graphs

- A computation graph is a representation of the process of computing a mathematical expression:
- Example: $L(a;b;c) = c(a + 2b)$
- Using explicit operations:



Backward Differentiation

- How we can compute the derivative of the output function L with respect to each of the input variables?
- Backwards differentiation makes use of the **chain rule**.

$$f(x) = u(v(x)) \Rightarrow \frac{df}{dx} = \frac{du}{dv}\frac{dv}{dx}$$

$$f(x) = u\left(v(w(x))\right) \Rightarrow \frac{df}{dx} = \frac{du}{dv}\frac{dv}{dw}\frac{dw}{dx}$$

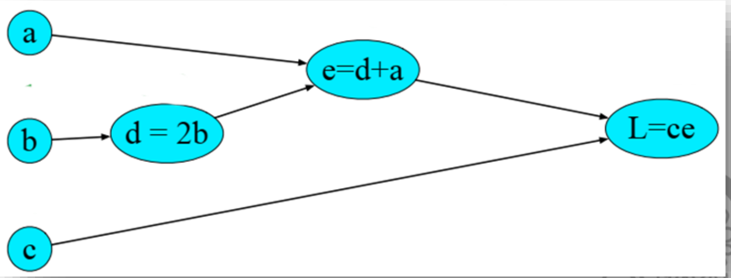
Backward Differentiation

- For the previous example we have:

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

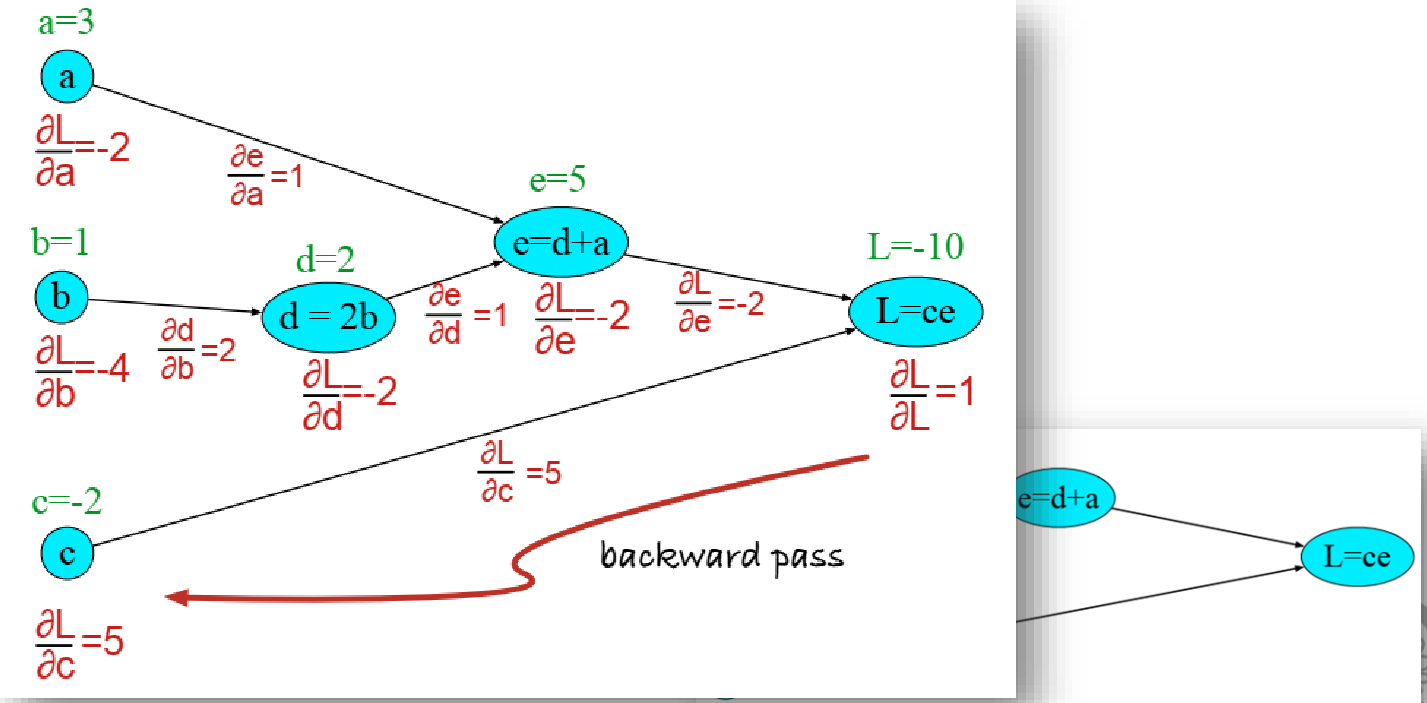
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

$$\frac{\partial L}{\partial c} = e$$

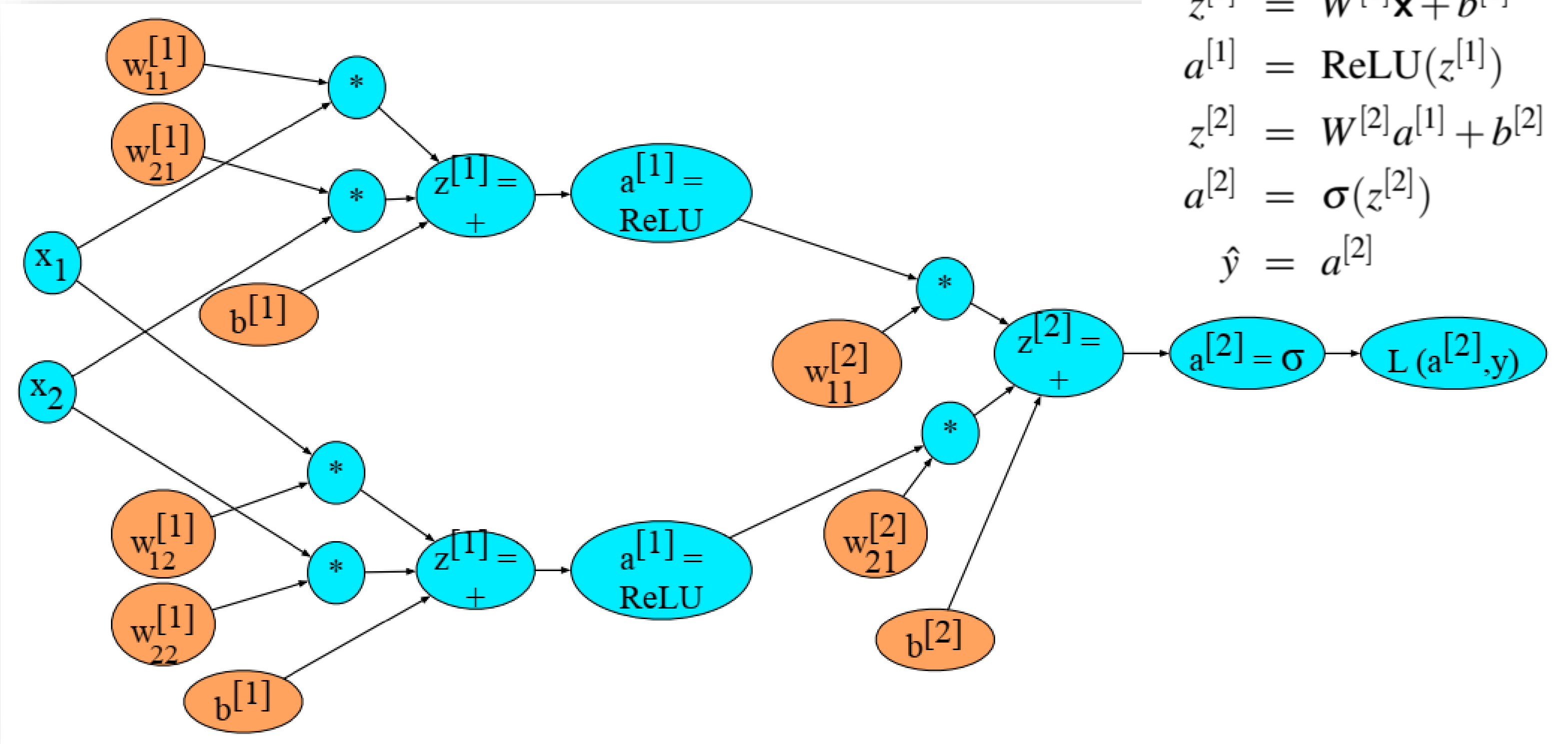


Backward Differentiation

- For the previous example we have:



Computation Graphs for NN



$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \mathrm{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

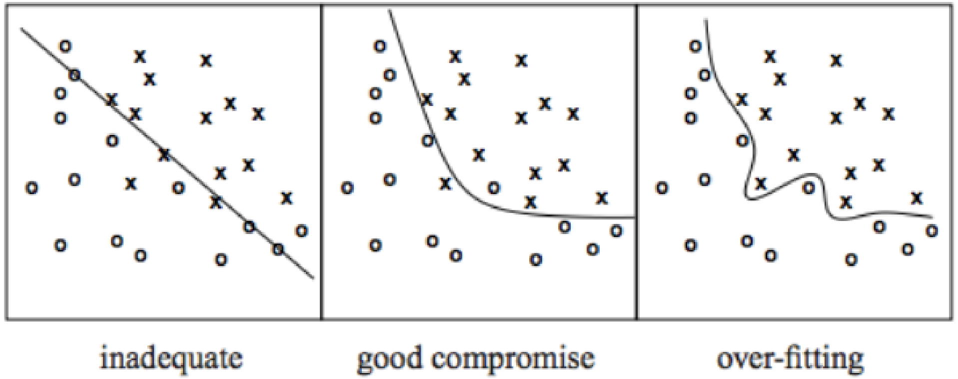
Different Types of Training

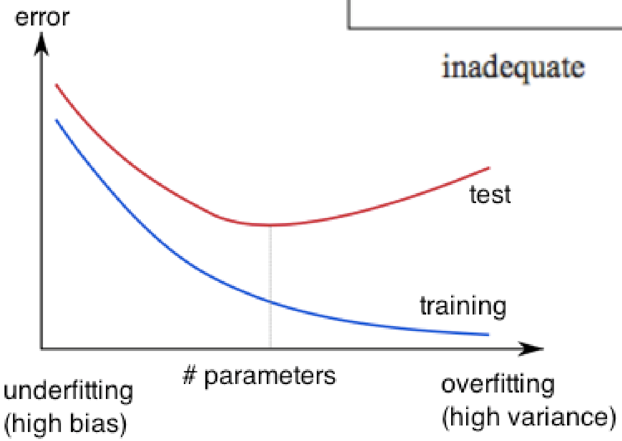
- **Gradient Decent:**
  - Also called Batch gradient descent
  - Computes the gradient using the whole dataset.

- **Stochastic Gradient Decent:**
  - An online algorithm that minimizes the loss function by computing its gradient after each training example.

- **Mini-batch training**:
  - Train on a group of m examples that is less than the whole dataset.
  - The mini-batch gradient is the average of the individual gradients.

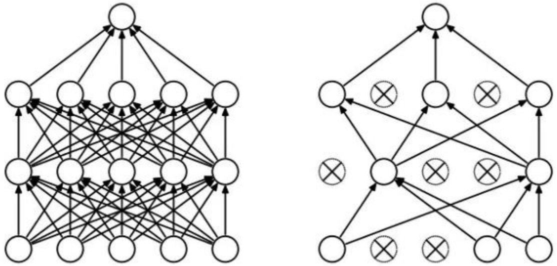
Overfitting





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

Regularization



Dropout

- Randomly drop units (along with their connections) during training
- Each unit retained with fixed probability p, independent of other units
- Hyper-parameter p to be chosen (tuned)

L2 = weight decay

- Regularization term that penalizes big weights, added to the objective
- Weight decay value determines how dominant regularization is during gradient computation
- Big weight decay coefficient => big penalty for big weights

$$J_{reg}(\theta) = J(\theta) + \lambda \sum_k \theta_k^2$$

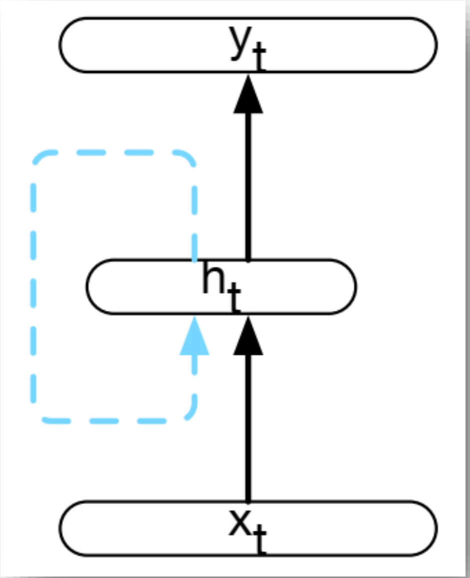
Early-stopping

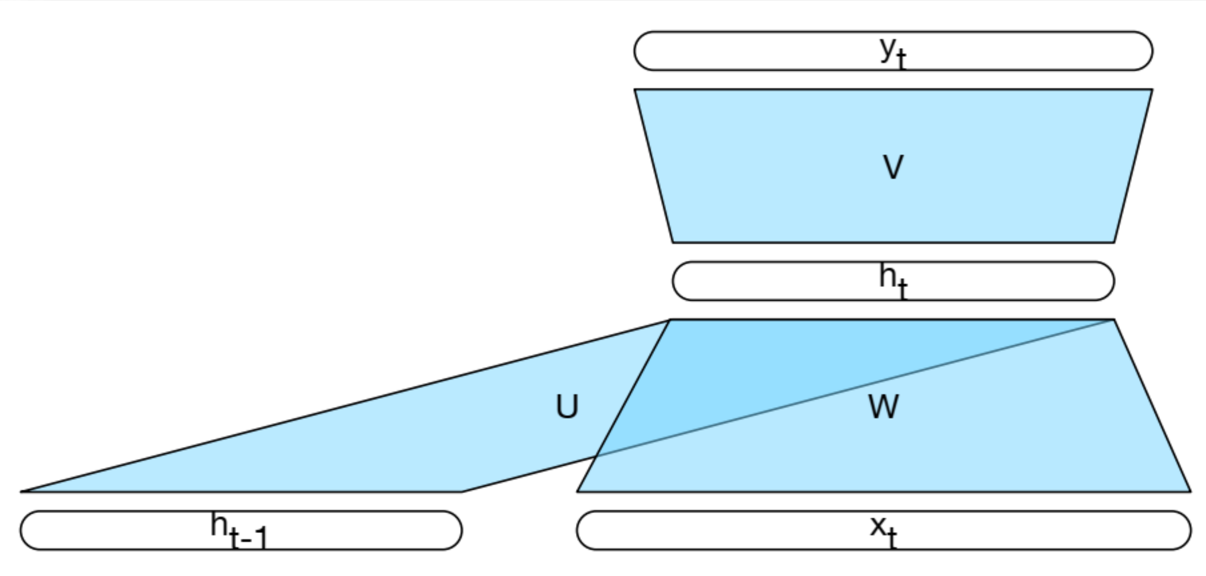
- Use validation error to decide when to stop training
- Stop when monitored quantity has not improved after n subsequent epochs

Recurrent Neural Networks

Simple Recurrent Neural Networks

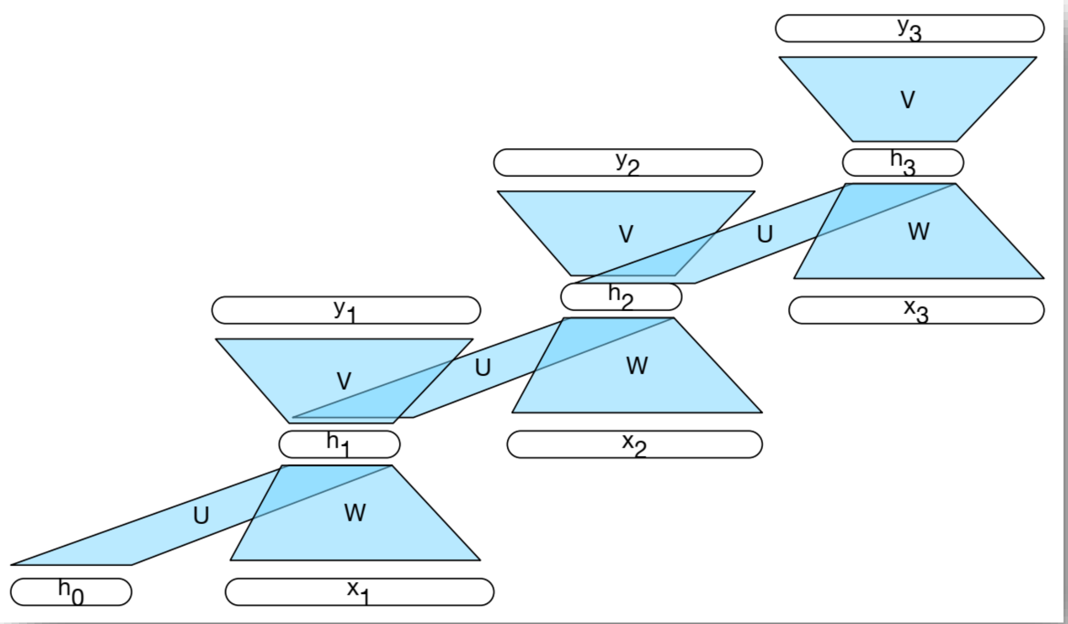
- Any network that contains a cycle within its network connections.
  - The value of a unit is directly, or indirectly, dependent on earlier outputs as an input.



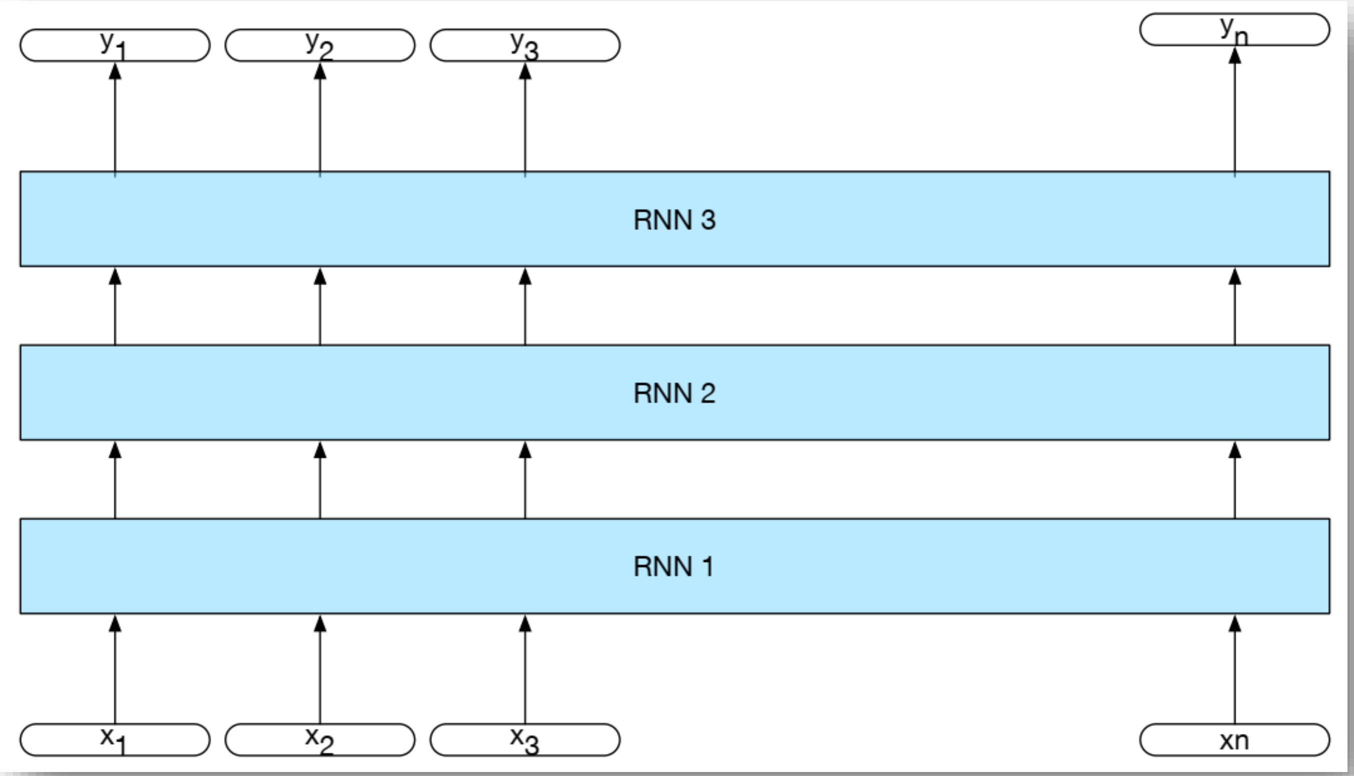


Unfolded RNN

- **Backpropagation through time** (BPTT) is a gradient-based technique for training of simple recurrent neural networks.

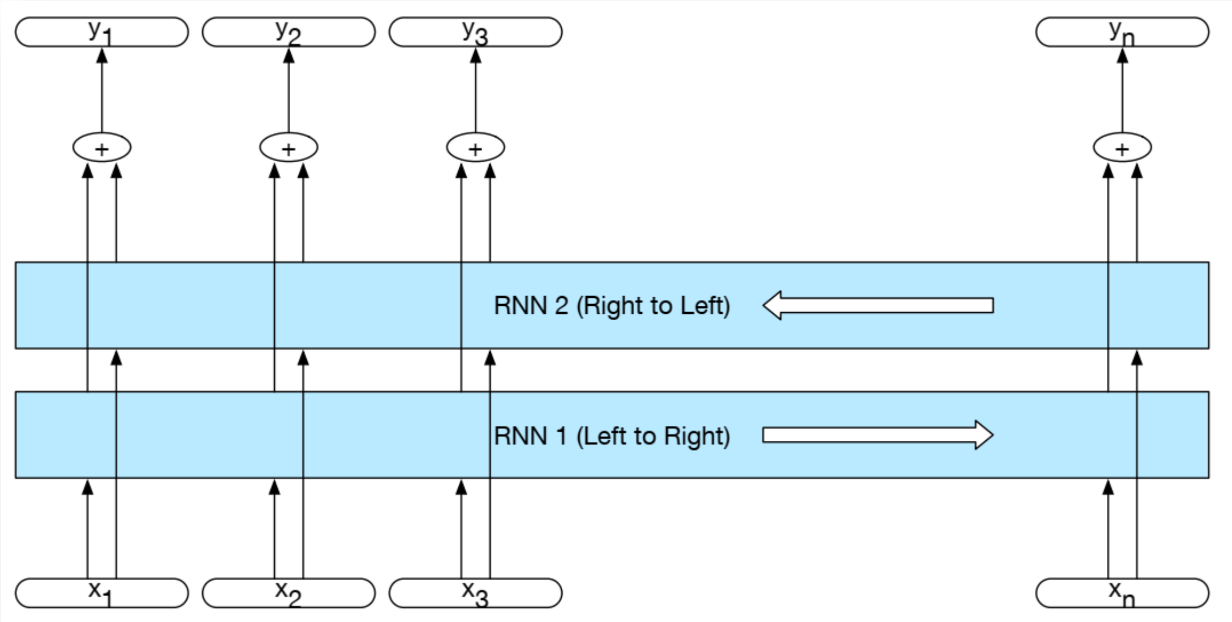


Stacked RNNs

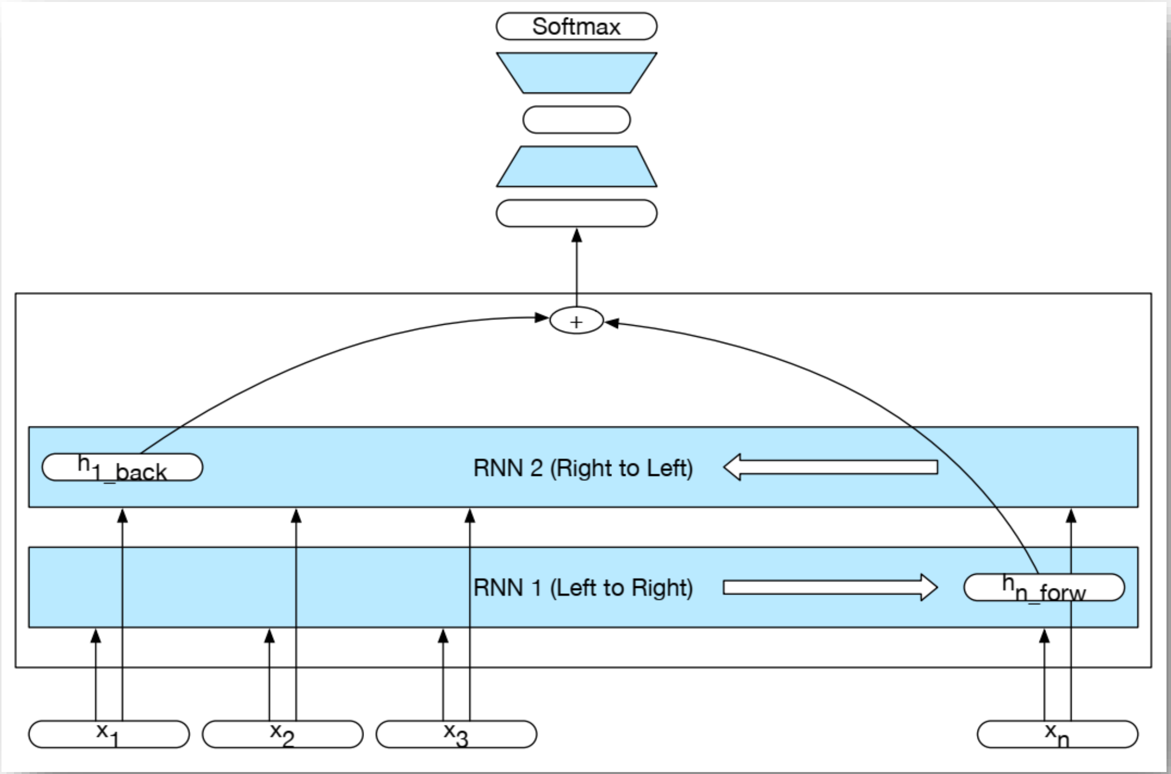


Bidirectional RNNs

- A Bi-RNN consists of two independent RNNs
  - Combine the outputs of the two networks into a single representation



Bi-RNNs for Classification



LSTMs and GRUs

- It is quite difficult to train RNNs for tasks that require information distant from the current point of processing.
  - The information encoded in hidden states tends to be fairly local, more relevant to the most recent parts of the input sequence.
- Inability of RNNs:
  - Hidden layer should perform two tasks simultaneously
  - Vanishing gradients subject to repeated multiplications
- Solutions:
  - The network needs to learn to forget information that is no longer needed and to remember information required for decisions still to come.

Long Short-Term Memory

- Divide the context management problem into two sub-problems:
  - Removing information no longer needed from the context.
  - Adding information likely to be needed for later decision making.
- Main changes:
  - Adding an explicit context layer to the architecture
  - Using *gates* to control the flow of information
- Gates design pattern; each consists of:
  - A feedforward layer
  - Followed by a sigmoid activation function
  - Followed by a pointwise multiplication with the layer being gated.

Long Short-Term Memory

- Forget gate: delete information from the context that is no longer needed.

$$f_t = \sigma\left(U_f h_{t-1} + W_f x_t\right)$$
$$k_t = c_{t-1} \odot f_t$$

- Extract actual information (same as RNN):

$$g_t = \tanh\left(U_g h_{t-1} + W_g x_t\right)$$

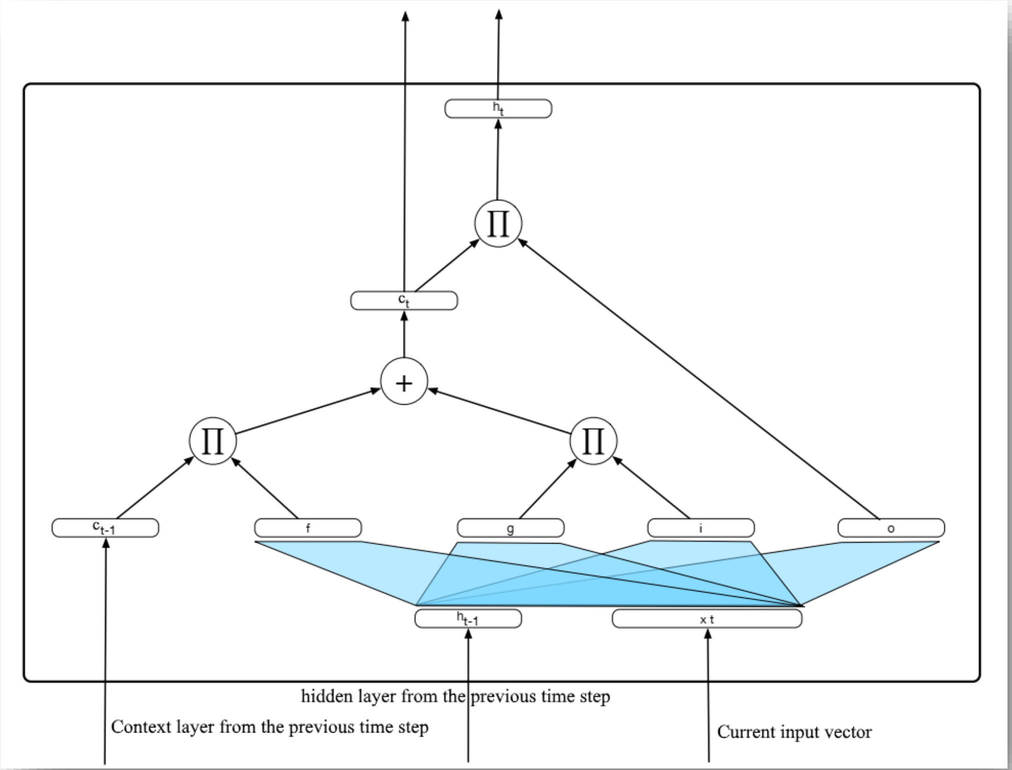
- Add gate: select the information to add to the current context.

$$i_t = \sigma(U_i h_{t-1} + W_i x_t)$$
$$c_t = g_t \odot i_t + k_t$$

- Output gate:

$$o_t = \sigma(U_o h_{t-1} + W_o x_t)$$
$$k_t = o_t \odot \tanh(c_t)$$

Long Short-Term Memory



Gated Recurrent Units (GRU)

- LSTMs has 4 times more parameters than RNN
- GRU reduces the number of gates to 2
  - **Reset gate**: decide which aspects of the previous hidden state are relevant

$$r_t = \sigma(U_r h_{t-1} + W_r x_t)$$

  - **Update gate:**

$$z_t = \sigma(U_z h_{t-1} + W_z x_t)$$

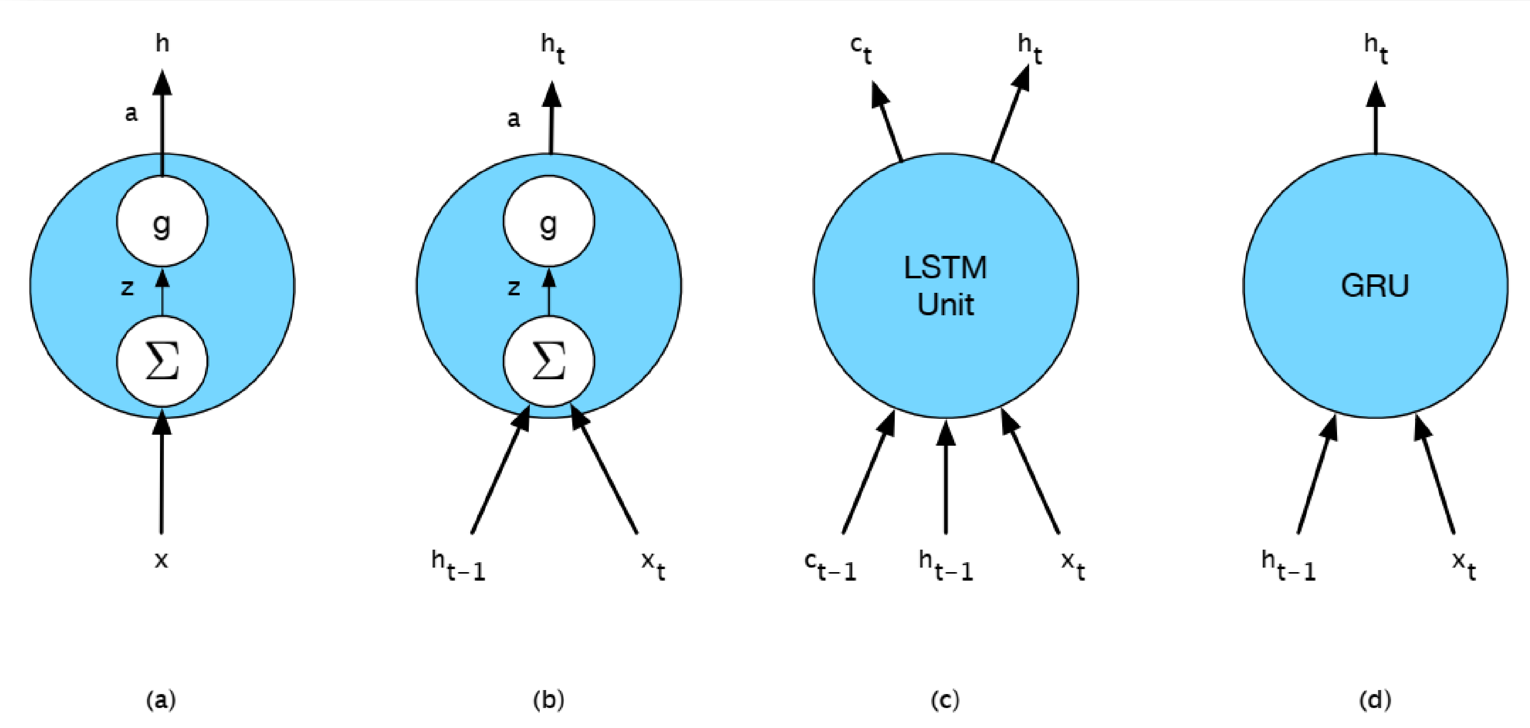
- Intermediate representation:

$$\hat{h}_t = \tanh(U(r_t \odot h_{t-1}) + W x_t)$$

- Calculate the output:

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

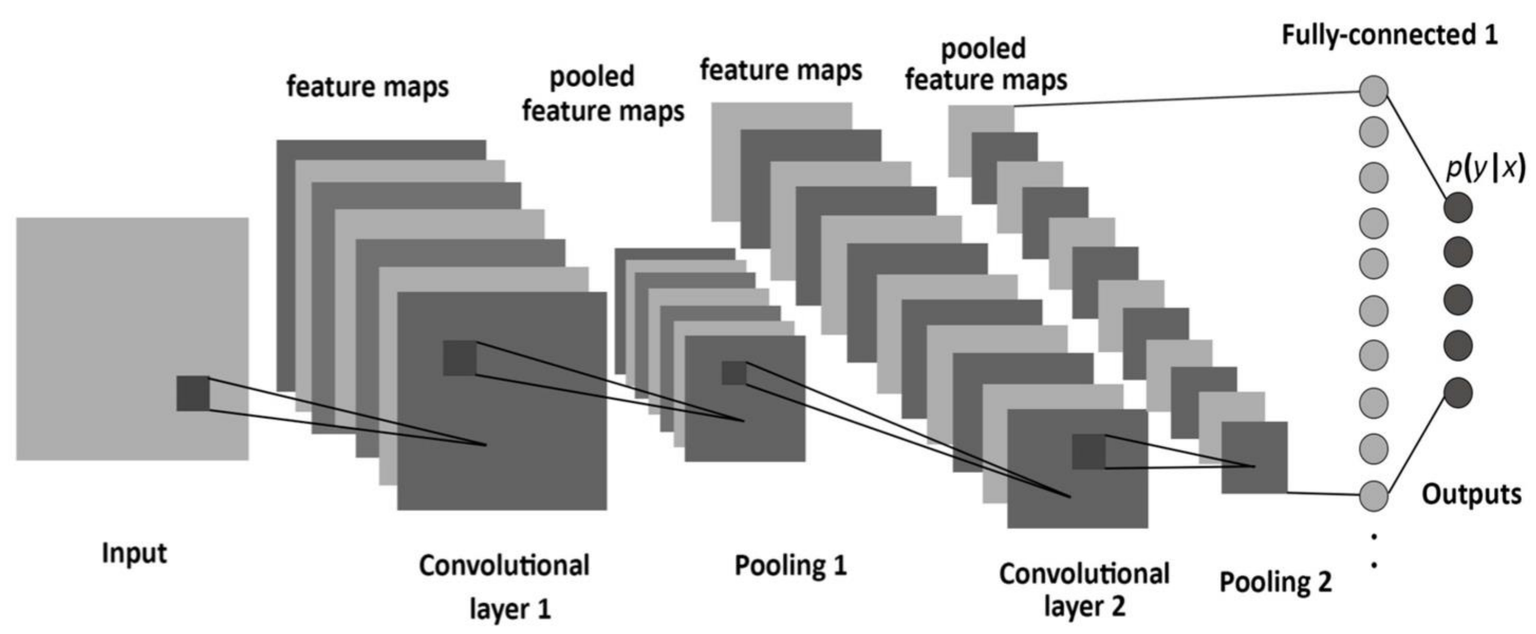
Comparison of Different Units



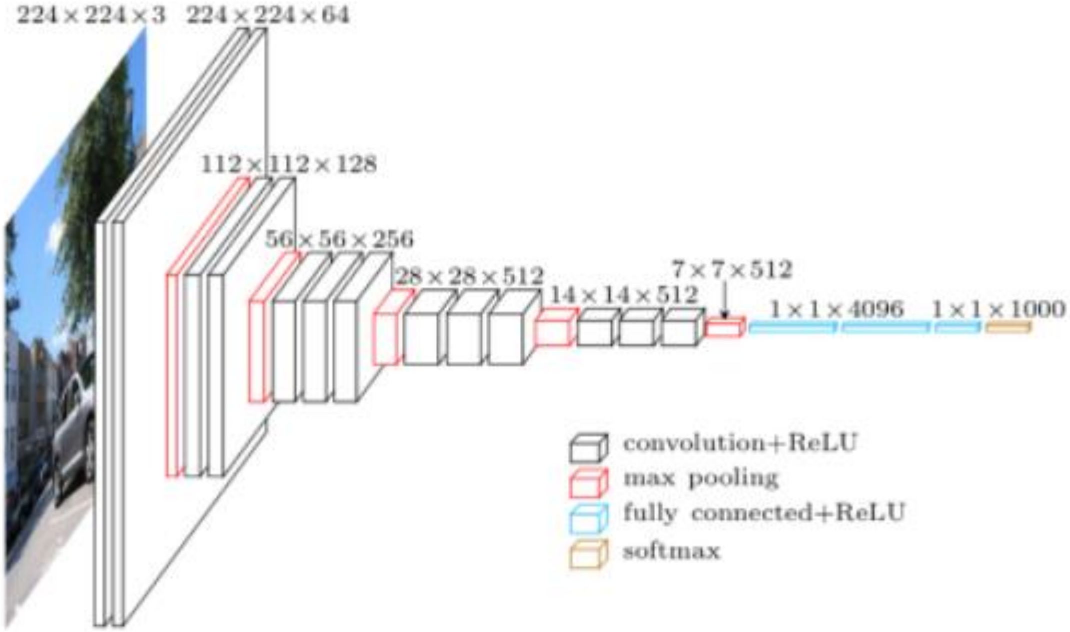
Convolutional Neural Network

Convolutional Neural Network

- Below we see a typical 2D convolutional neural network (CNN).



Convolutional Neural Network



Convolutional Neural Network

- A CNN has two main components:
  - Convolutions
  - Pooling
- Convolutions serve to partition the image and look at local regions instead of the entire image as given
- Pooling reduces dimensionality which helps in the optimization of parameters
- In Feedforward NNs, each input neuron is connected to each output neuron in next layer.
  - While CNNs use convolutions over the input layer to compute the output.
  - This results in local connections: each region of the input is connected to a neuron in the output.

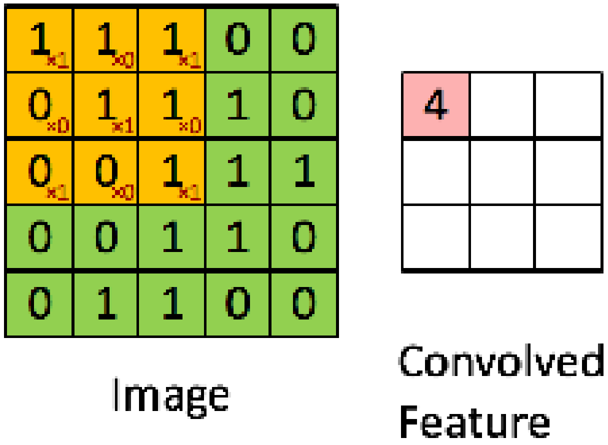
2D convolution

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input matrix

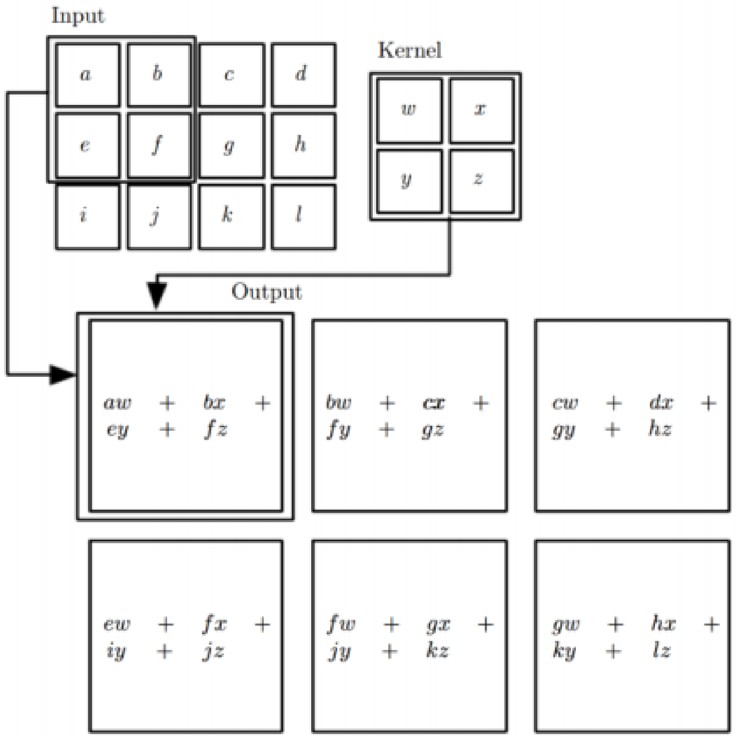
1	0	1
0	1	0
1	0	1

Convolutional
3x3 filter



http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

2D convolution



Max Pooling

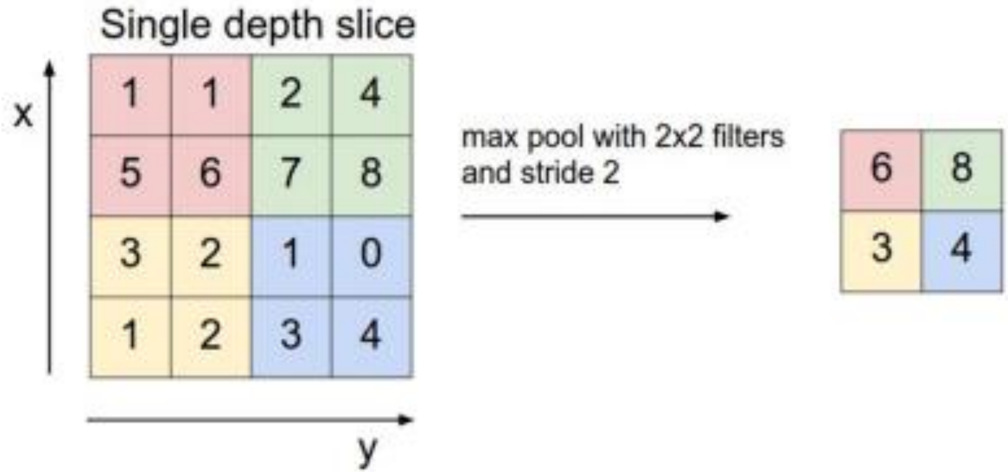
Feature Map

6	4	8	5
5	4	5	8
3	6	7	7
7	9	7	2

Max-Pooling

https://shafeentejani.github.io/assets/images/pooling.gif

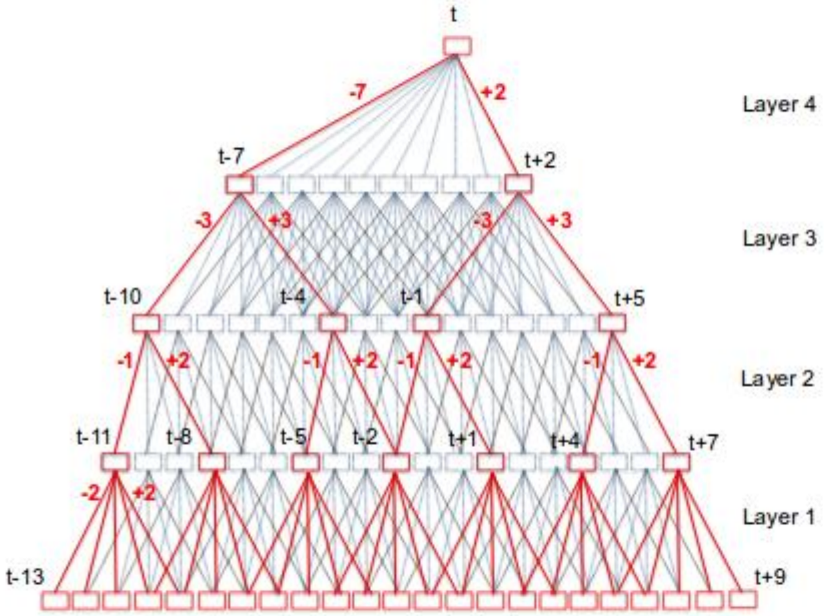
Max Pooling



Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

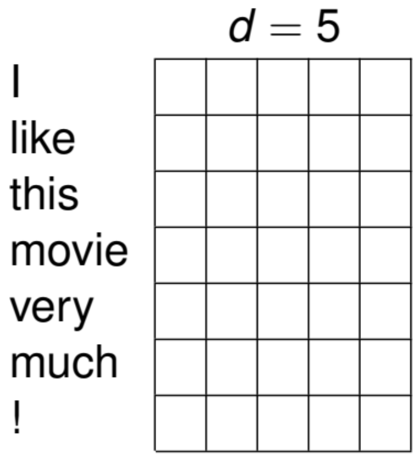
1-Dimensinal CNN

- Also called Time Delay Neural Network (TDNN)



CNN in Text Processing

- Begin with a tokenized sentence and convert it into a matrix.
  - Rows are *d*-dimensional word vectors for each token
  - Let *s* denote sentence length, then matrix is $s \times d$
  - Sentence looks like an image now, we can apply convolutions.



CNN in Text Processing

- In vision filters slide over local patches of an image. While in NLP, filters slide over full rows of the matrix (words).
  - The **width** of the filter is same as d width of input matrix.
  - The **height** h or region size of the filter is number of adjacent rows.
  - Sliding windows over 2-5 words at a time is typical.

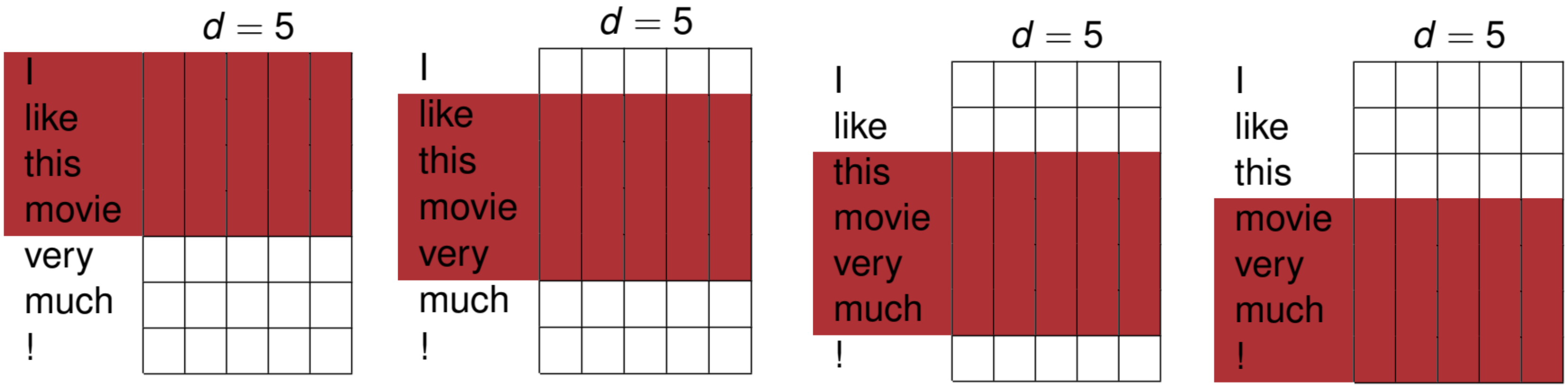
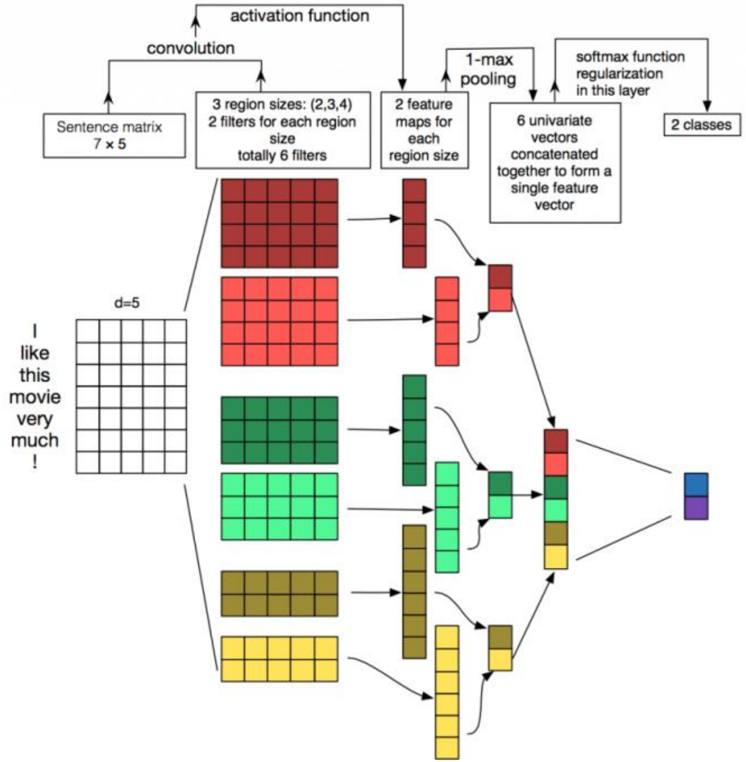


Illustration of CNN Model



Thanks for your attention

References and IP Notice

- Daniel Jurafsky and James H. Martin, “Speech and Language Processing”, 3rd ed., 2019
- Some slides on CNN were selected from Mirella Lapata’s slides.
- Some graphics were selected Slidesgo template