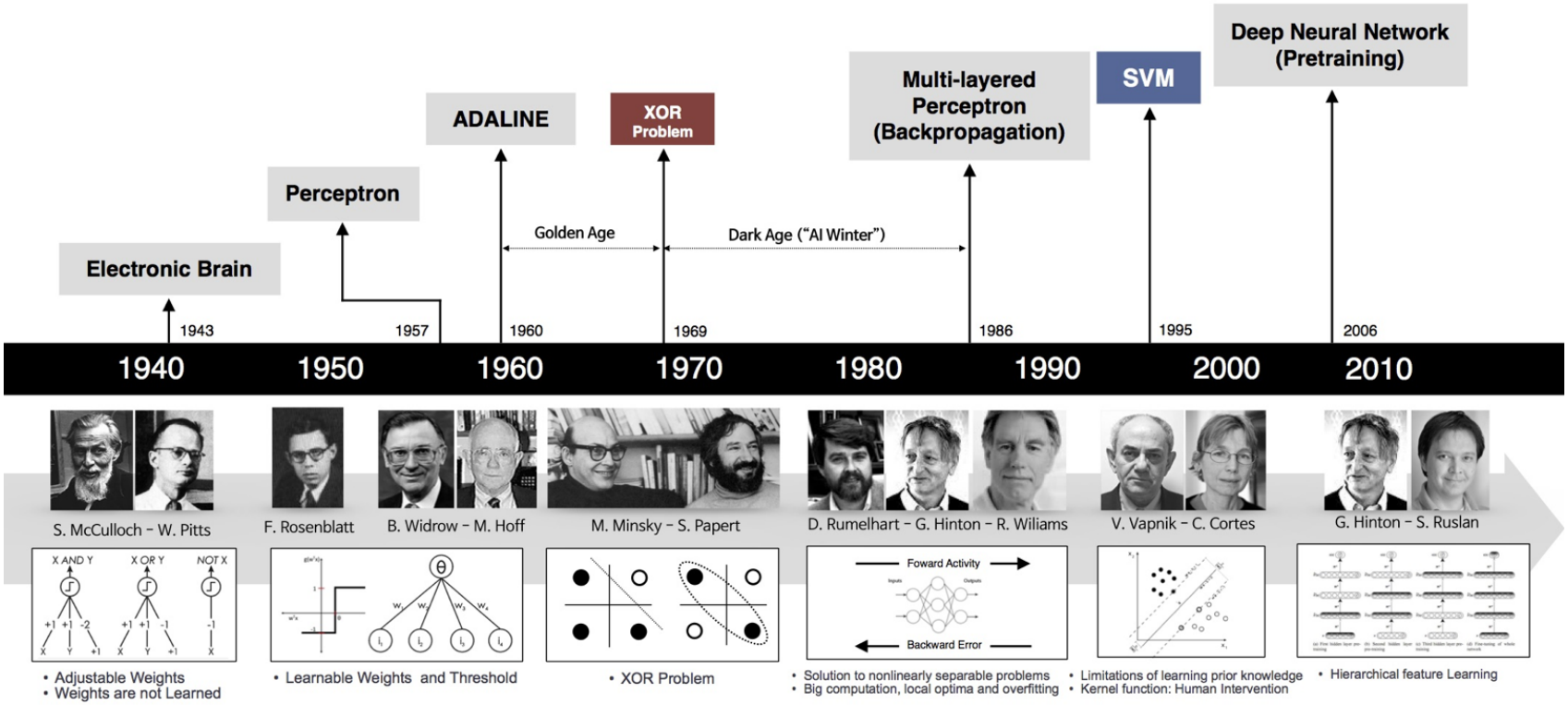
Deep Feedforward Network

Jie Tang

Tsinghua University

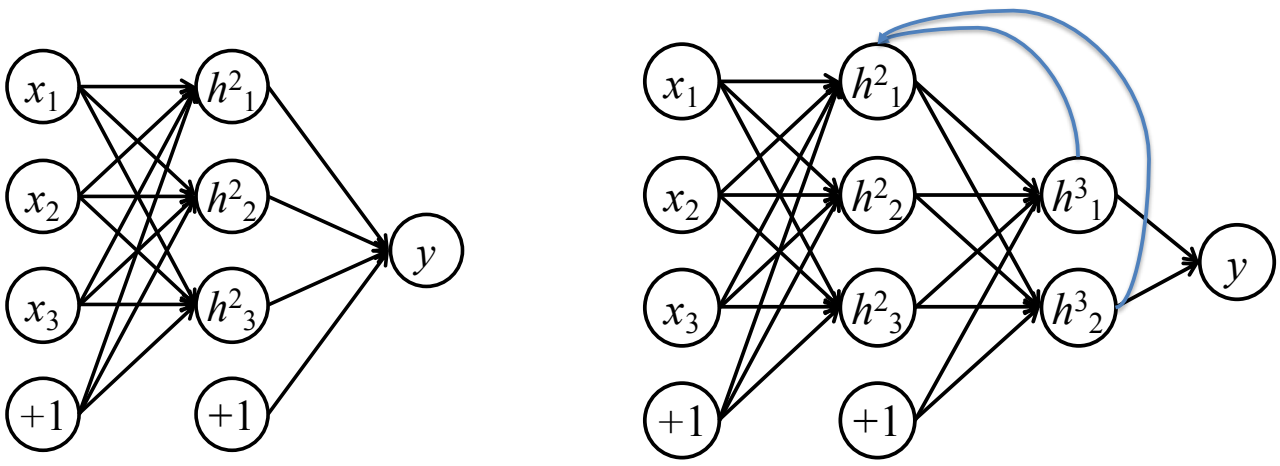
September 17, 2024

Overview


Deep Neural Network (Pretraining)
SVM
Multi-layered Perceptron (Backpropagation)
XOR Problem
ADALINE
Perceptron
Electronic Brain
Golden Age
Dark Age ("AI Winter")
1943
1957
1960
1969
1986
1995
2006
1940
1950
1960
1970
1980
1990
2000
2010
S. McCulloch - W. Pitts
F. Rosenblatt
B. Widrow - M. Hoff
M. Minsky - S. Papert
D. Rumelhart - G. Hinton - R. Wiliams
V. Vapnik - C. Cortes
G. Hinton - S. Ruslan
X AND Y
X OR Y
NOT X
Foward Activity
Backward Error
• Adjustable Weights
• Weights are not Learned
• Learnable Weights and Threshold
• XOR Problem
• Solution to nonlinearly separable problems
• Big computation, local optima and overfitting
• Limitations of learning prior knowledge
• Kernel function: Human Intervention
• Hierarchical feature Learning

Deep Feedforward Networks

- A **deep feedforward network** (DFN), also referred to as **Multi-layer Perceptron** and **Feedforward Neural Network**, is an artificial neural network wherein connections between the units do not form a **cycle**. It is the first and also the simplest type of ANN.
- When extended to include feedback connections, they are called **Recurrent Neural Networks** (RNN).

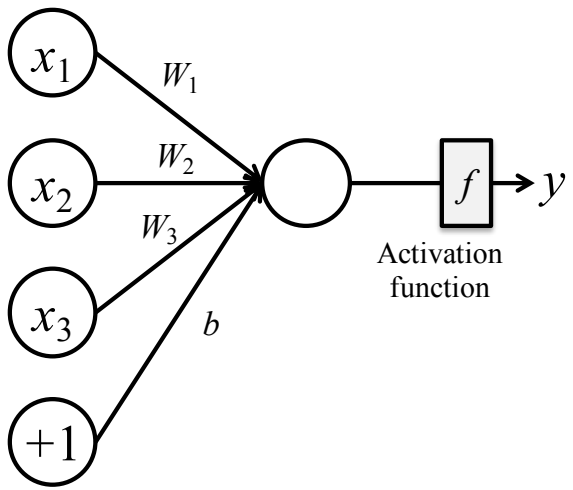


(a) DFN (b) RNN

Deep Forward Network (DFN) v.s. Recurrent Neural Network (RNN).

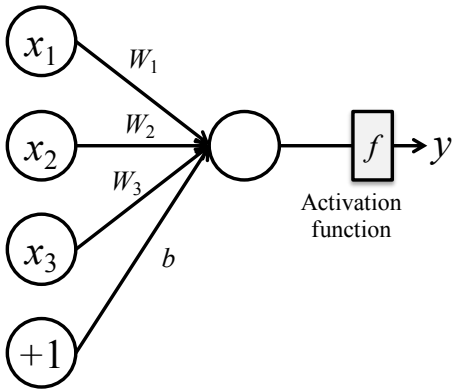
Perceptron

- in 1957, perceptron was first invented at the Cornell Aeronautical Laboratory by **Frank Rosenblatt**.
- in 1940s, a similar neuron was actually described by McCulloch and Pitts;
- in 1958, Rosenblatt made statements that Perceptron will grow up to be a computer that is **"able to walk, talk, see, write, reproduce itself and be conscious of its existence."**;
- in 1960s, people recognized that multilayer perceptron had far greater processing power than perceptrons with one layer (**Minsky and Papert**; **Grossberg**);
- in 1964, kernel perceptron (Aizerman);
- in 1998, Margin bounds guarantees were given in the general non-separable case first by **Freund and Schapire**;
- in 1999, voted percepton — giving comparable generalization bounds to the kernel SVM.



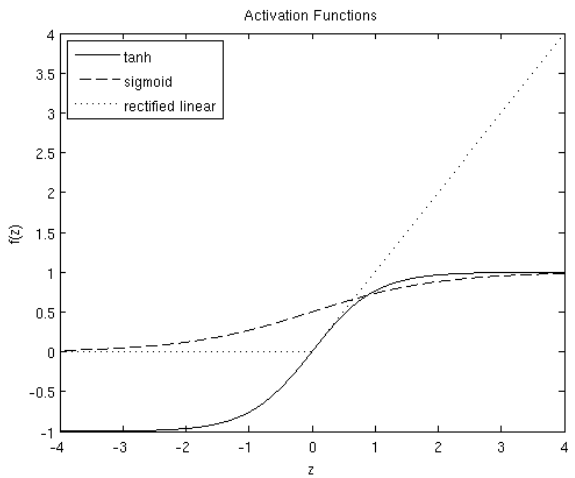
Single-layer Perceptron

$$y = f(W^T x) = f(\sum_{i=1}^{3} W_i x_i + 1 \times b)$$



where f is the activation function:

- **sigmoid function** $f(z) = \frac{1}{1+\exp(-z)}$
- **hyperbolic tangent** $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- **rectified linear function**
  $f(z) = \max(0, x)$



remember, it is useful

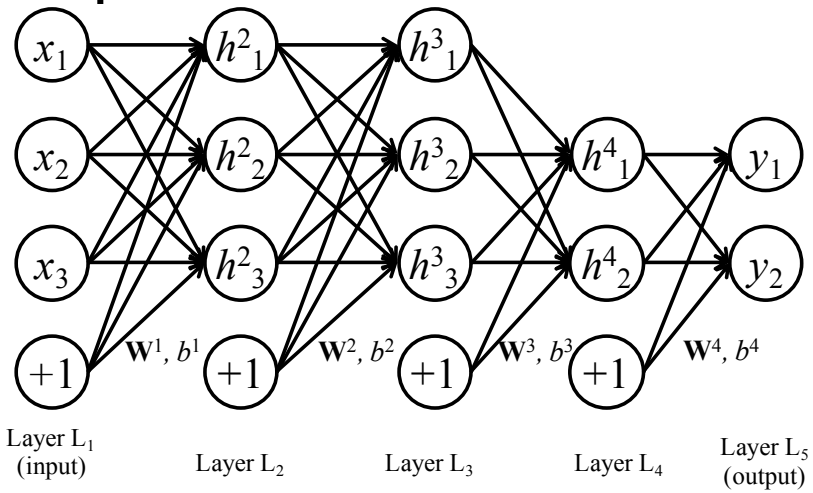
- **sigmoid function** $f'(z) = f(z)(1 - f(z))$
- **hyperbolic tangent** $f'(z) = 1 - (f(z))^2$
- **rectified linear function**
  $f'(z) = 0$, if $z \leq 0$; and 1 otherwise

Softmax Units for Multinomial Output Distributions

A **softmax** output unit is defined by:

- $softmax(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$
- Softmax units naturally represent a probability distribution over a discrete variable with k possible values.
- A linear layer predicts unnormalized log probabilities $z = W^\top h + b$. where $z_i = \log \tilde{P}(y = i|x)$
- This can be seen as a generalization of the sigmoid function which was used to represent a probability distribution over a binary variable.
- Softmax functions are most often used as the output of a classifier.
- The log in the log-likelihood can undo the exp of the softmax: $\log softmax(z)_i = z_i - \log \sum_j \exp(z_j)$.

Deep Feedforward Networks

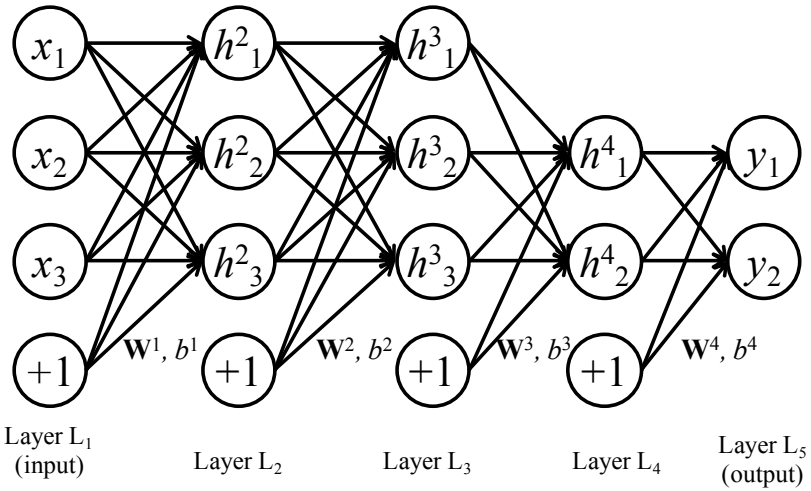


- bias units $\{+1\}$, input layer $\{x_i\}$, hidden layer(s) h^l_i, output layer y_i;
- **Goal:** approximate some map function f^*
  – e.g., a classifier $y = f^*(x)$ to map an input x onto a category y
  – Feedforward Network defines a mapping

$$y = f^*(x; \theta)$$

  – and learns parameters $\theta = (\mathbf{W}^l, \mathbf{b}^l)$ that result in the best approximation.

Deep Feedforward Networks

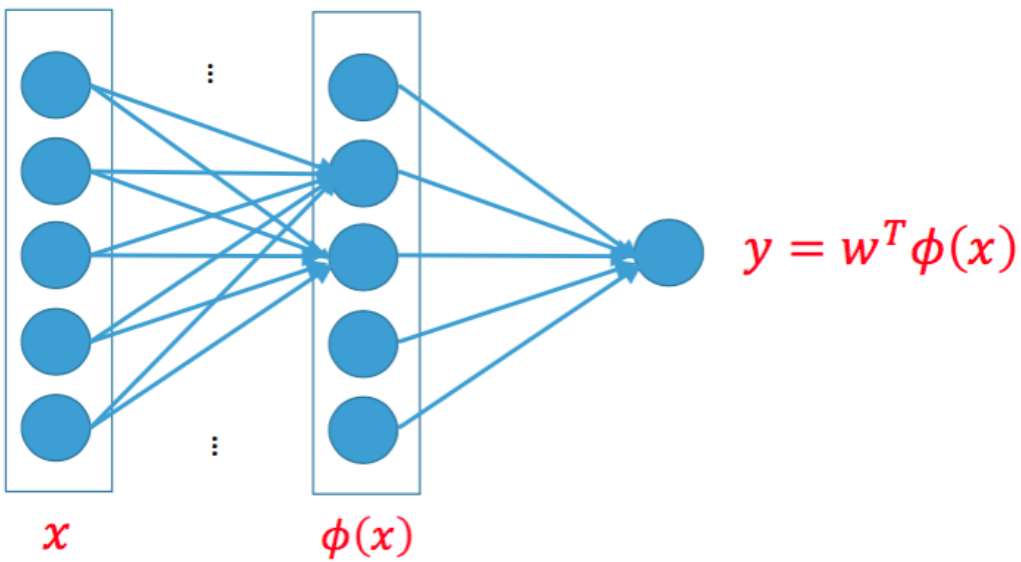


$$
\begin{aligned}
\mathbf{z}^2 &= \mathbf{W}^1 \times \mathbf{x} + b^1 \\
\mathbf{h}^2 &= f(\mathbf{z}^2) \\
\mathbf{z}^{l+1} &= \mathbf{W}^l \times \mathbf{h}^l + b^l \\
\mathbf{y} &= f(\mathbf{z}^L)
\end{aligned}
\tag{1}
$$

Outline

Extending Linear Models

- Linear functions $\phi(x) = W^T x$ do not work in some cases: need some nonlinearity $\phi(x)$.



Extending Linear Models

- To represent non-linear functions of x, by transforming x using a non-linear function $\phi(x)$
  - Similar "kernel" trick obtains a nonlinearity
  - SVM: $f(x) = w^T x + b \rightarrow f(x) = b + \sum_i \alpha_i \phi(x)\phi(x^i)$
  - where $\phi(x)\phi(x^i) = K(x, x^i)$

Then how to **choose** the mapping ϕ.

- Use a very **generic** ϕ, such as infinite-dimensional ϕ, but the generalization is poor.
- **Manually engineer** ϕ, but with little transfer between domains.
- The strategy of deep learning is to learn ϕ. We parametrize the **representation as** $\phi(x; \theta)$ and use the optimization algorithm to find the θ.
  - capture the benefit of the first approach by being highly generic —— by using a very broad family $\phi(x; \theta)$.
  - the human designer only needs to find the right general function family rather than precisely the right function.

Example: Learning XOR

- We will not be concerned with statistical generalization.
  - Perform correctly on the four points
    - $\mathbb{X} = \{[0,0]^T, [0,1]^T, [1,0]^T, [1,1]^T\}$
  - The only challenge is to fit the training set.
- The XOR function provides the target function $y = f^*(x)$ that we want to learn. Our model provides a function $y = f(x;\theta)$ and our learning algorithm will adapt the parameters θ to make f as similar as possible to f^*.

Linear model doesn't fit

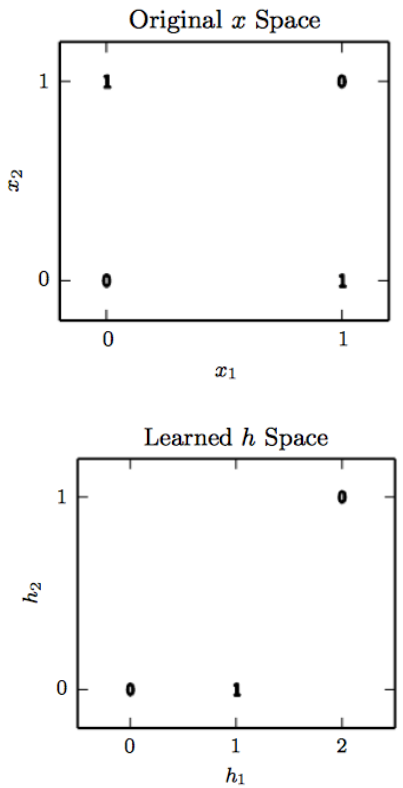
- Treat this problem as a regression problem and use a mean squared error
  - The MSE loss function is

$$J(\theta) = \frac{1}{4} \sum_{x \in \mathbb{X}} (f^*(x) - f(x; \theta))^2$$

  - Usually not used for binary data, but math is simple.
- We now choose the form of model, consider a linear model.
  - $f(x; \mathbf{w}, b) = x^T \mathbf{w} + b$
  - Minimizing the $J(\theta)$, we obtain $\mathbf{w} = 0$ and $b = \frac{1}{2}$
  - Then the linear model simply outputs $\frac{1}{2}$ everywhere.

Linear model doesn't fit

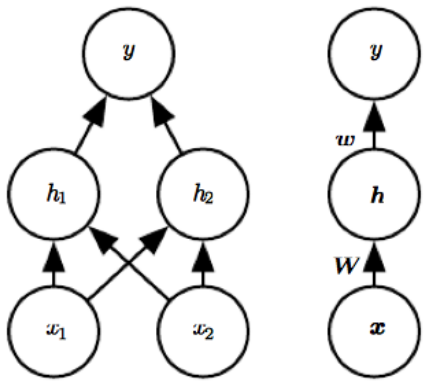
- Solving the XOR problem by learning a representation.
  - When $x_1 = 0$, output has to increase with x_2
  - When $x_1 = 1$, output has to decrease with x_2
  - Linear model $f(x, w, b) = x_1 w_1 + x_2 w_2 + b$ has to assign a single weight to x_2 so it **cannot solve the problem**.
- A better solution.
  - We user a simple feedforward network.
    - **one hidden layer** containing two hidden units.
  - the nonlinear features have mapped both $x = [1, 0]^T$ and $x = [0, 1]^T$ to a single point in feature space, $h = [1, 0]^T$.
    - A linear model can now solve the problem.



Feedforward Network for XOR

- **Layer 1 (hidden layer)**:
  $h_i = f(\mathbf{x}^T \mathbf{W}_{:,i} + c_i)$
  - c are bias variables.
  - g is typically chosen to be an element-wise activation function.
  - the default activation function is the *rectified linear unit* or ReLU defined by $g(z) = max\{0, z\}$
- **The complete network**:

$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^T f(\mathbf{W}^T \mathbf{x} + \mathbf{c}) + b$$



Feedforward Network for XOR

- Let $\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, $\mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$, $\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$, and $b = 0$. We get correct answers on all points.
- Now walk through how models processes.
  - Design matrix $\mathbf{X}$ of all four points.
  - $\mathbf{XW}$.
  - Adding $\mathbf{c}$.
  - Applying ReLU.
  - Multipying by $\mathbf{w}$.

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix},$$

$$\mathbf{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

$$\mathbf{XW+c} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix},$$

$$\mathbf{g(XW+c)} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\mathbf{w}g(\mathbf{XW+c}) + b = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Feedforward Network for XOR

- We simply specified the solution by defining
  - It achieves zero error.

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \text{ and } b = 0$$

- In real situations there might be billions of parameters and billions of training examples.
  - So one cannot simply guess the solution.
- Instead gradient descent optimization can find parameters that produce very little error.

Outline

Loss Function

- Support we are given a set of **training data** $\{(x^1, y^1), \cdots, (x^n, y^n)\}$ of n training examples. To learning our **parameters** $\theta = (\mathbf{W}, \mathbf{b})$, we can define the following **loss function**:

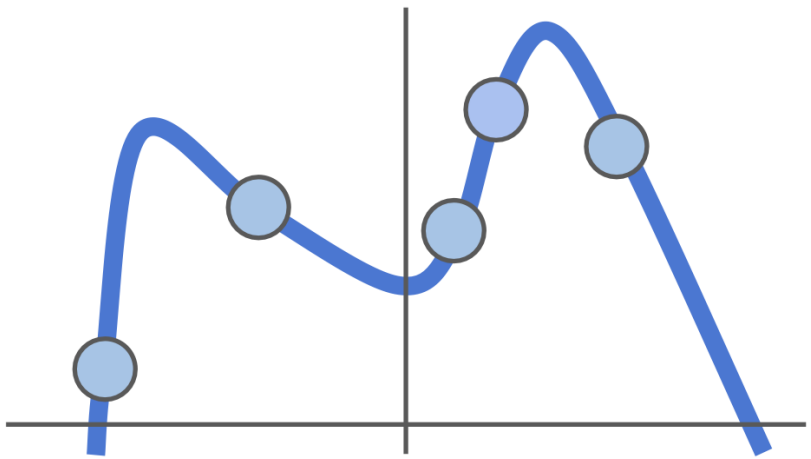
$$\mathcal{J}(W, b; x, y) = \frac{1}{2}\|\hat{y} - y\|^2$$

  where $\hat{y}$ is the value predicted by the learned neural network.
- Loss function measures the difference between the predicted output and the ground truth.

Loss Function

$$\mathcal{J}(W, b; x_{train}, y_{train}) = \frac{1}{2}\|\hat{y} - y_{train}\|^2$$

Training Loss: Model predictions should match training data. It can guide the model optimization's direction.

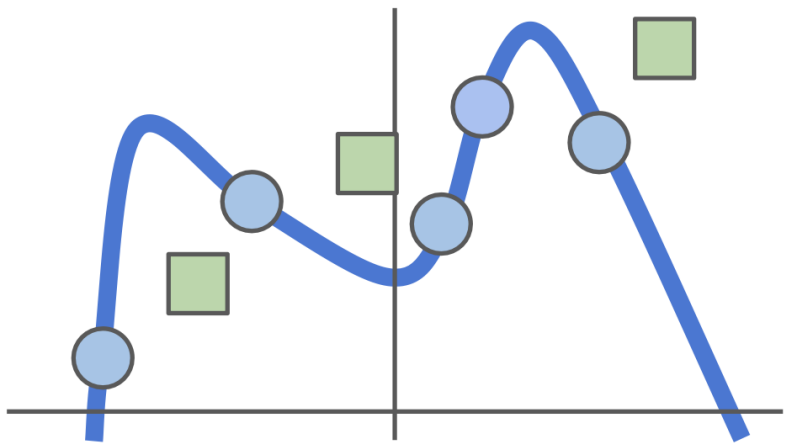


Blue: training data. Training loss = 0.

Loss Function

$$\mathcal{J}(W, b; x_{valid}, y_{valid}) = \frac{1}{2}\|\hat{y} - y_{valid}\|^2$$

Validation Loss: Test the generalization ability of the model on unseen data.



Green: validation data. Validation loss is large. **Over-fitting!**

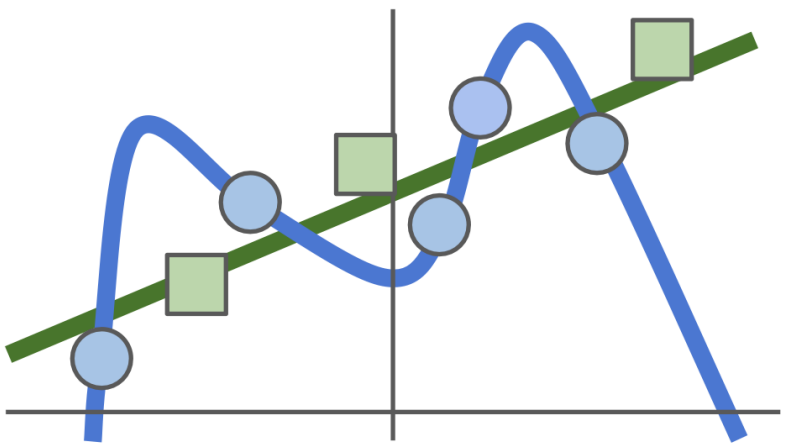
Regularization Term

To avoid overfitting, we could add a **regularization term**

$$\mathcal{J}(W,b) = \frac{1}{n}\left[\sum_{i=1}^{n}\frac{1}{2}\|\hat{y}-y\|^2\right] + \frac{\lambda}{2}\sum_{l=1}^{L}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l=1}}(W_{ji}^{l})^2$$

Model should be "simple", so it works on test data

* please note that we do not regularize b, as it makes little difference.



Green line: after regularization.

Types of Loss Functions

- Regression Tasks: Mean Squared Error (L2 loss), Mean Absolute Error (L1 loss)...
- Classification Tasks: Cross-Entropy ...
- Other: Hinge Loss (SVM) ...

Loss Function	**Formula**
Mean Squared Error (MSE)	$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
Mean Absolute Error (MAE)	$\frac{1}{n}\sum_{i=1}^{n}\lvert y_i - \hat{y}_i\rvert$
Categorical Cross-Entropy	$-\sum_{i=1}^{n} y_i \log(\hat{y}_i)$
Hinge Loss (for SVM)	$\max(0, 1 - y \cdot \hat{y})$

Common Loss Functions and Their Formulas

Outline

Model Learning

- Our goal is to minimize $\mathcal{J}(W, b)$ as a loss function of W and b.
- We first initialize each parameter $W_{ij}^{(l)}$ and each $b_i^{(l)}$ to a **small random value** near zero (say according to a $Normal(0, \epsilon^2)$ distribution for some small ϵ).
  - all-zero initialization will be very bad;
  - W_{ij}^{l} will be the same for all h and z, —- $h_1^{l+1} = h_2^{l+1} = \cdots$ for any input.
- We then apply an optimization algorithm such as batch gradient descent.
- Since $\mathcal{J}(W, b)$ is a non-convex function, gradient descent is susceptible to local optima; however, in practice gradient descent usually works fairly well.

Backpropagation

- One iteration of gradient descent updates the parameters W, b as follows:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) \quad (2)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b) \quad (3)$$

- where α is the learning rate.
- The key step is computing the partial derivatives above.
- We next describe the **backpropagation** algorithm, which gives an efficient way to compute these partial derivatives.

Backpropagation

Forward propagation

The inputs x provide the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{y}$.

Backward propagation

simply called **backprop**, allows the information from the cost to then flow backwards through the network, in order to compute the gradient.

Training

- One iteration of gradient descent updates the parameters W, b as follows:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} \mathcal{J}(W, b) \tag{4}$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} \mathcal{J}(W, b) \tag{5}$$

- where α is the learning rate.

$$\begin{aligned} \frac{\partial}{\partial W_{ij}^{(l)}} \mathcal{J}(W, b) &= \left[\frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial W_{ij}^{l}} \mathcal{J}(W, b; x, y) \right] + \lambda W_{ij}^{l} \\ \frac{\partial}{\partial b_i^{(l)}} \mathcal{J}(W, b) &= \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial b_i^{l}} \mathcal{J}(W, b; x, y) \end{aligned} \tag{6}$$

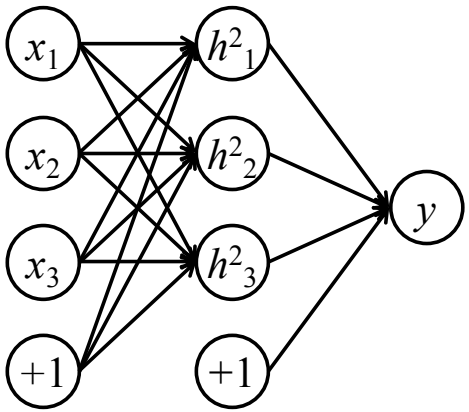
- Now, our problem becomes how to compute $\frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial W_{ij}^{l}} \mathcal{J}(W, b; x, y)$ with **backpropagation**.

Backpropagation: Intuition

- Given a training example (x, y), we will first run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $\hat{y}$.
- Then, for each node i in layer l, we would like to compute an "error term" $\delta_i^{(l)}$ that measures how much that node was "responsible" for any errors in our output.
- For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{(L)}$.
- For hidden units, we will compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $z_i^{(l)}$ as an input.

Backpropagation: forward

- Given a training example (x, y), we will first run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $\hat{y}$.



$$z_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \quad (7)$$

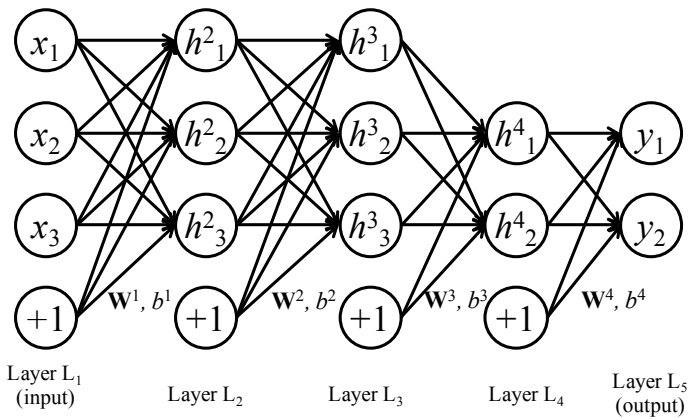
$$z_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \quad (8)$$

$$z_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \quad (9)$$

$$\hat{y} = z_1^{(3)} = f(W_{11}^{(2)}z_1^{(2)} + W_{12}^{(2)}z_2^{(2)} + W_{13}^{(2)}z_3^{(2)} + b^{(2)}) \quad (10)$$

Backpropagation: error in the output layer

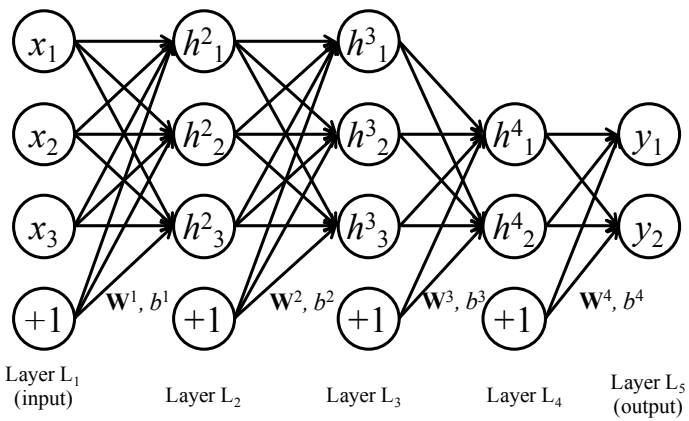
- Then, for each node i in layer l, we would like to compute an "error term" $\delta_i^{(l)}$ that measures how much that node was "responsible" for any errors in our output.
- For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{(L)}$.



$$\delta_i^L = \frac{\partial}{\partial z_i^L} \frac{1}{2} \|y - \hat{y}\|^2 = -(y_i - f(z_i^L)) \cdot f'(z_i^L) \tag{11}$$

Backpropagation: error backpropagation

- For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{(L)}$.
- **For hidden units, we will compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $z_i^{(l)}$ as an input.**

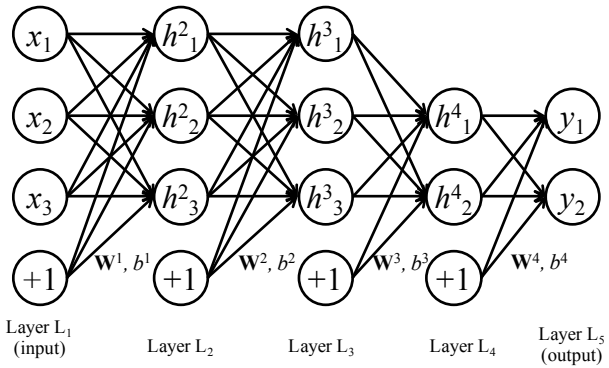


For $l = L-1, L-2, \cdots, 2$ and **each node** i in layer l, set

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)}\right) f'(z_i^{(l)}) \tag{12}$$

Backpropagation: error backpropagation (2)

- For hidden units, we will compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $z_i^{(l)}$ as an input.



For $l = L-1, L-2, \cdots, 2$ and **each node** i in layer l, set

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)}\right) f'(z_i^{(l)})$$

Recall **chain rule**: if $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

Backpropagation: derivations

- Compute the desired partial derivatives, which are given as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = z_j^{(l)} \delta_i^{(l+1)} \tag{13}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)} \tag{14}$$

- Plug it back into:

$$\begin{aligned} \frac{\partial}{\partial W_{ij}^{(l)}} \mathcal{J}(W, b) &= \left[\frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial W_{ij}^{l}} \mathcal{J}(W, b; x, y) \right] + \lambda W_{ij}^{l} \\ \frac{\partial}{\partial b_i^{(l)}} \mathcal{J}(W, b) &= \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial b_i^{l}} \mathcal{J}(W, b; x, y) \end{aligned} \tag{15}$$

Backpropagation: summary (pseudo-code)

- repeat
  - $\Delta \mathbf{W}^l := 0$, $\Delta \mathbf{b}^l := 0$ for all l;
  - for $i = 1$ to n,
    - perform forward propagation;
    - use backpropagation to computer errors at each layer;
    - compute partial derivatives $\nabla_{\mathbf{W}^l} \mathcal{J}(W, b; x, y)$ and $\nabla_{\mathbf{b}^l} \mathcal{J}(W, b; x, y)$
    - $\Delta \mathbf{W}^l := \Delta \mathbf{W}^l + \nabla_{\mathbf{W}^l} \mathcal{J}(W, b; x, y)$;
    - $\Delta \mathbf{b}^l := \Delta \mathbf{b}^l + \nabla_{\mathbf{b}^l} \mathcal{J}(W, b; x, y)$;
  - update the parameters

$$\mathbf{W}^l = \mathbf{W}^l - \alpha \left[(\frac{1}{n} \Delta \mathbf{W}^l) + \lambda \mathbf{W}^l \right]$$

$$\mathbf{b}^l = \mathbf{b}^l - \alpha \left[(\frac{1}{n} \Delta \mathbf{b}^l) \right]$$

- until convergence.

Question: What are general limitations of back propagation rule?

A. Local minima problem

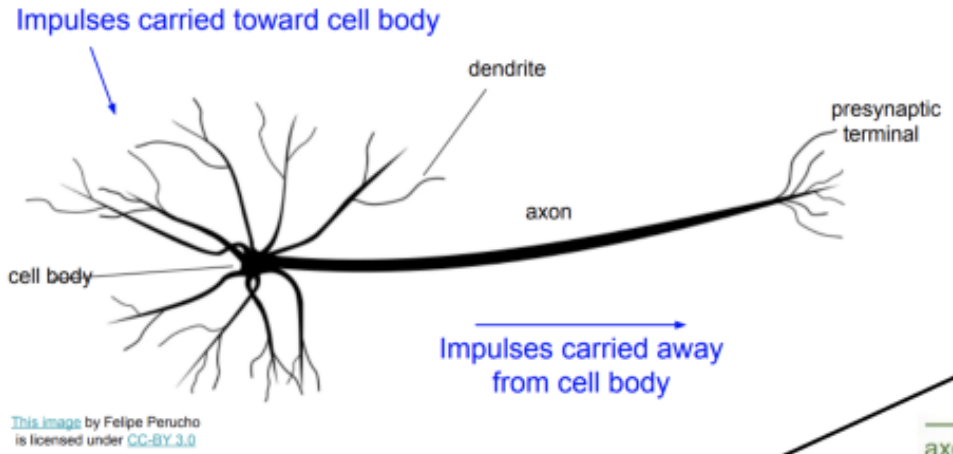
B. Slow convergence

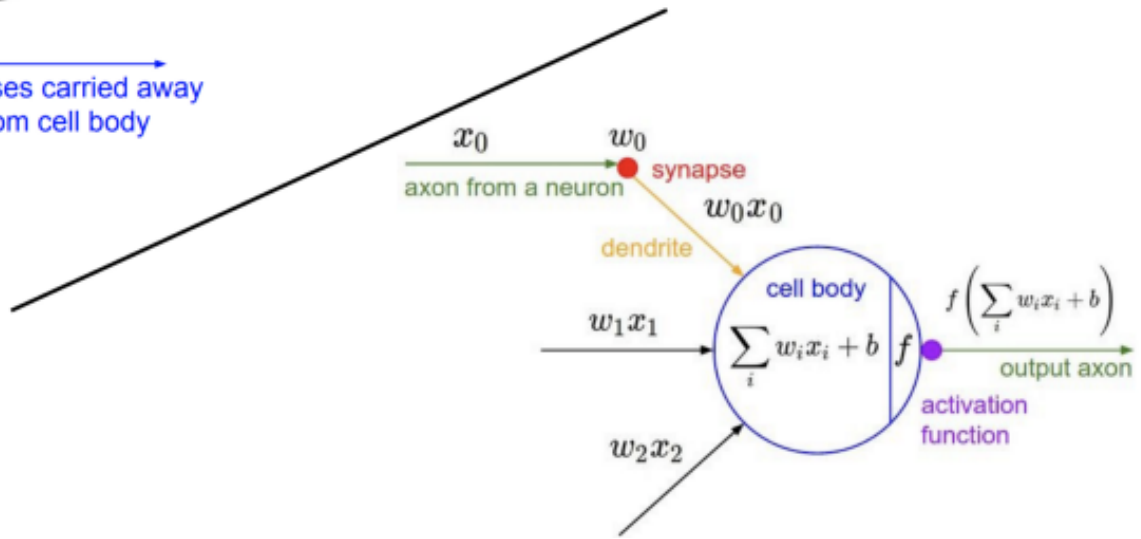
C. Sensitive to noisy data

D. All of the mentioned

Outline

Why do we need activation functions?





We use activation functions to simulate the two states of activation and non-activation of animal neurons.
On the other hand, we also need nonlinear layers to enhance the expressiveness of the network.

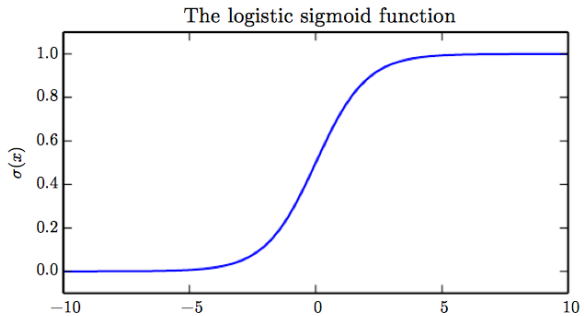
Activation functions

The design of activation functions for hidden units is an extremely active area of research.

- Logistic Sigmoid
- Hyperbolic Tangent
- Rectified Linear Units
- Generalized Rectified Linear Units
- Gated Linear Units
- Maxout Units

Logistic Sigmoid

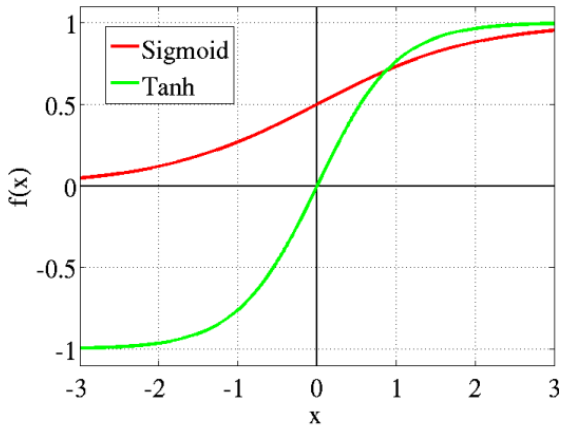
- $g(z) = \sigma(z)$. It's also used in output units.
- Logistic sigmoid activation function, sigmoidal units saturate across most of their domain and can make gradient-based learning very difficult.



- Recurrent networks, many probabilistic models, and some autoencoders have additional requirements that rule out the use of piecewise linear activation functions and make sigmoidal units more appealing despite the drawbacks of saturation.

Hyperbolic Tangent

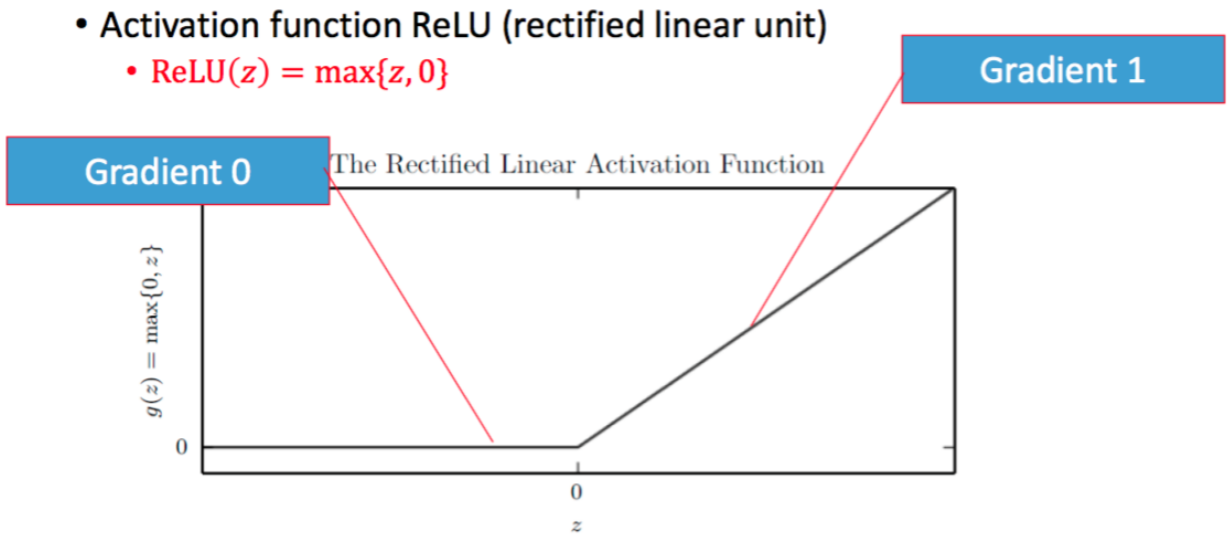
- $g(z) = tanh(z) = 2\sigma(2z)$-1.
- The hyperbolic tangent activation function typically performs better than the logistic sigmoid.



- Because tanh is similar to the identity function near 0, $\hat{y} = w^T \tanh(U^T \tanh(V^T x))$ resembles training a linear model $\hat{y} = w^T U^T V^T x$ so long as the activations of the network can be kept small. This makes training the tanh network easier.

Rectified Linear Units

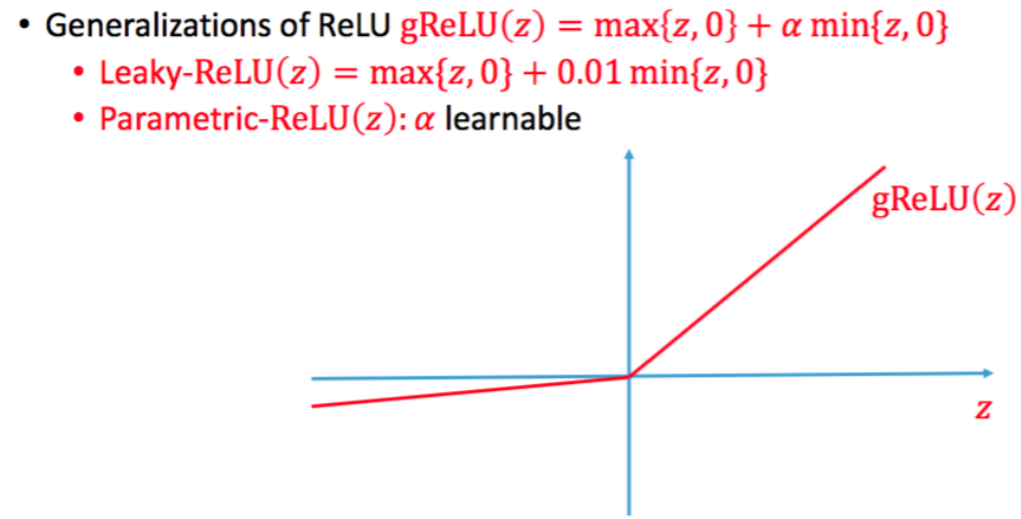
Rectified linear units are an **excellent** default choice of hidden unit (Nair&Hinton, 2010).



- The second derivative of the rectifying is 0 almost everywhere and the derivative is 1.
- So the gradient direction is far more useful for learning than it would be with activation functions that introduce second-order effects. Rectified linear units and its generalizations are based on the principle that models are easier to optimize if their behavior is closer to linear.

Generalized Rectified Linear Units

- One drawback to RLU is that they cannot learn via gradient-based methods on examples for which their activation is zero.



- It is used for object recognition from images, where it makes sense to seek features that are invariant under a polarity reversal of the input illumination.

Gated Linear Units(GLU)

- Input is split into two parts: one goes through a linear transformation, and the other is passed through a gating mechanism.

$$GLU(x, W, V, b, c) = \sigma(xW + b) \otimes (xV + c)$$

allowing the model to regulate the flow of information dynamically.

- Sigmoid can be replaced by GELU or Swish function

$$GeGLU(x, W, V, b, c) = GELU(xW + b) \otimes (xV + c)$$

$$SwiGLU(x, W, V, b, c) = Swish(xW + b) \otimes (xV + c)$$

↑

LLM

Maxout Units

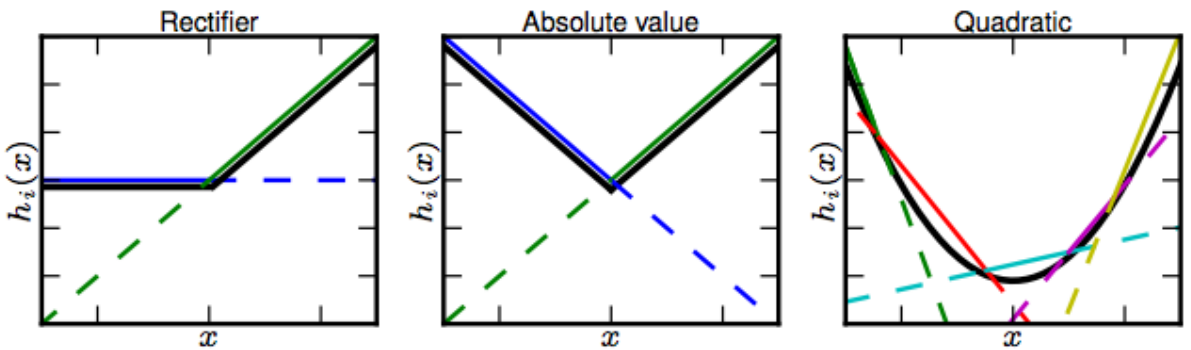
- **Maxout units** generalizes rectified linear units further. Instead of applying an element-wise function $g(z)$, maxout units divide z into groups of k values. One of these groups:

$$g(z)_i = \max_{j \in G^{(i)}} z_j \tag{16}$$

  where $G^{(i)}$ is the set of indices into the inputs for group i, $\{(i-1)k+1, ..., ik\}$.
- A maxout unit can learn a piecewise linear, convex function with up to k pieces. Maxout units can thus be seen as **learning the activation function** itself rather than just the relationship between units.

Maxout Units

- In a convolutional network, a maxout feature map can be constructed by taking the maximum across k affine feature maps (i.e., pool across channels).
- A single maxout unit can approximate arbitrary convex functions. The figure shows how maxout behaves with a 1D input.



Question: What is the relationship between DNN and Logistic Regression?

- Hint: consider **Sigmoid** activation function.

Question: What is the relationship between DNN and Logistic Regression?

- Logistic Regression is a special case of a Neural Network. A DNN is equivalent to Logistic Regression when the following conditions satisfy:
  - Feed-forward only
  - No hidden layer
  - Sigmoid/Softmax activation
  - Cross entropy loss

Architectural Considerations

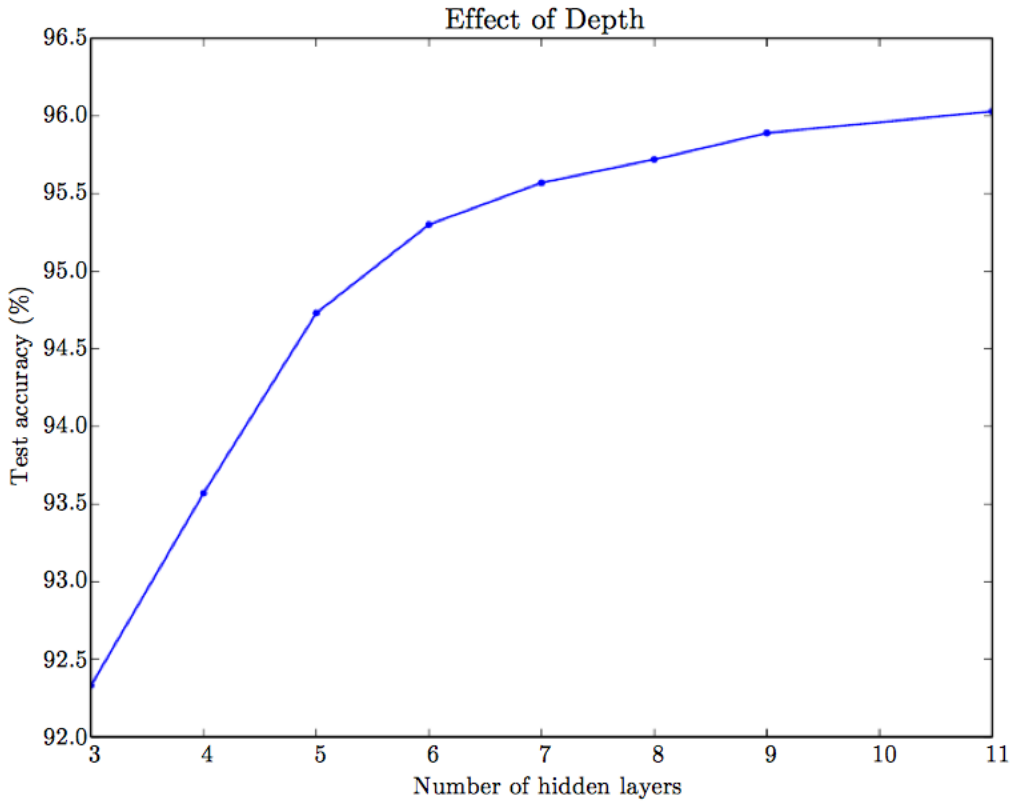
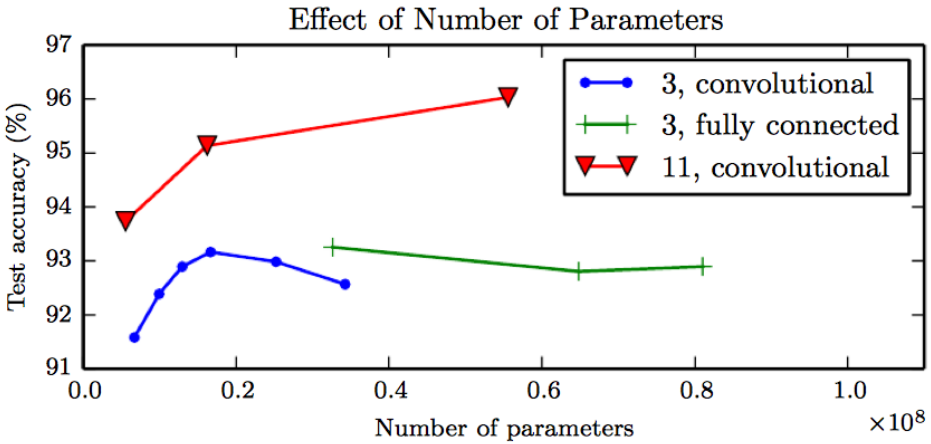


Figure 6.6: Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d). The test set accuracy consistently increases with increasing depth. See Fig. 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.

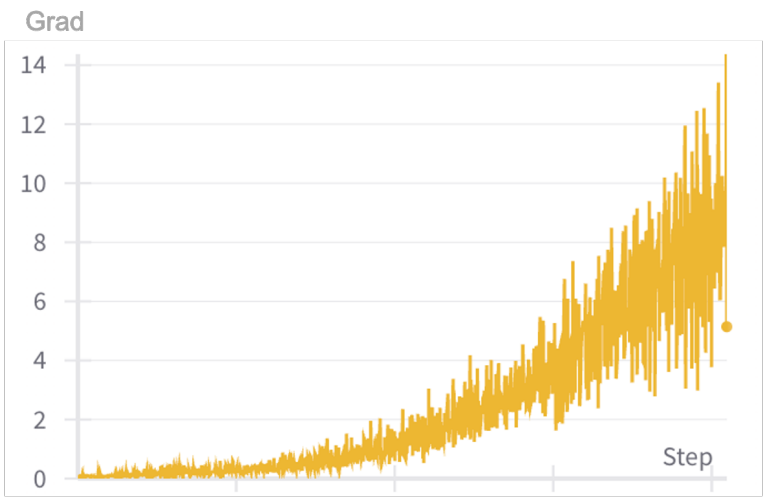
Architectural Considerations



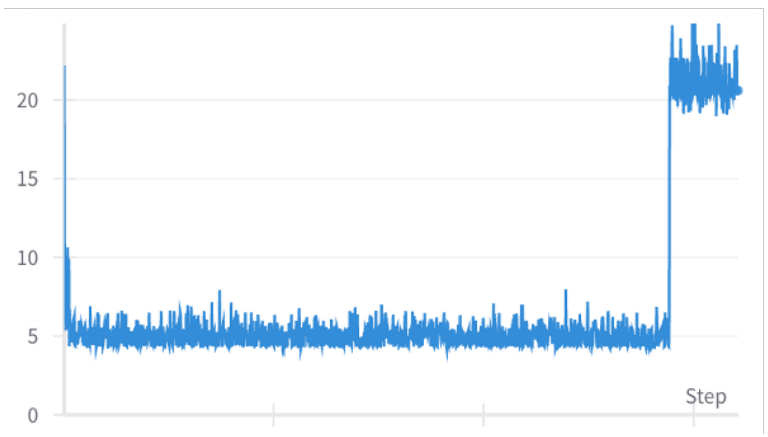
- Shallow models overfit at around 20 million parameters while deep ones can benefit from having over 60 million.
- It expresses a belief that the function should consist of many simpler functions composed together.
  - It could results in learning a representation that is composed in turn of simpler representations (e.g., corners defined in terms of edges)
  - or learning a program with sequentially dependent steps (e.g., first locate a set of objects, then segment them from each other, then recognize them).

Training Instability

As the model's layers deepen, the gradients may grow exponentially during back-propagation, which can lead to instability in training and cause the model to crash.



Grad Norm

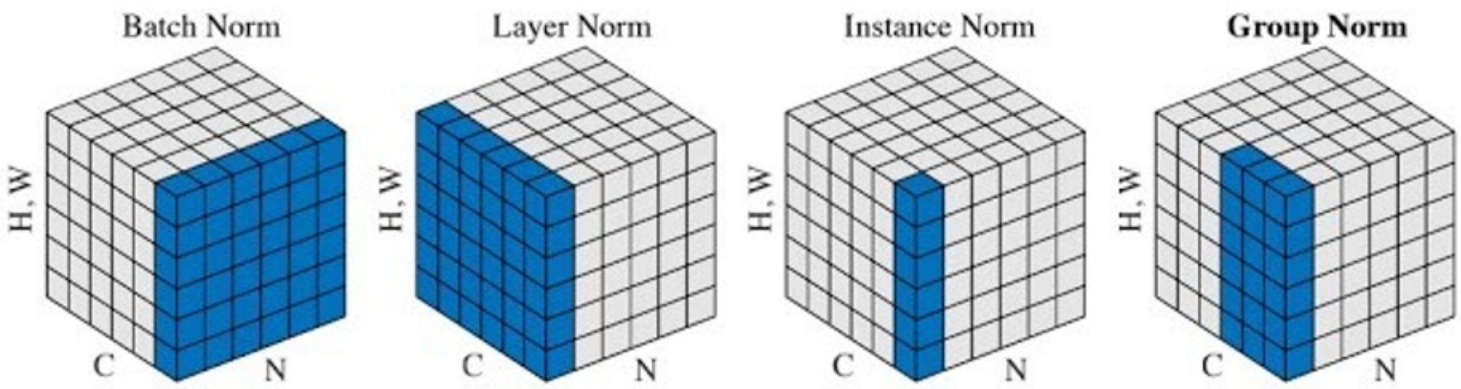


Loss

Stable hidden state: Normalization

- Normalization is the process of adjusting the values of input data or the intermediate activations in a neural network to a standard range.
- It helps stabilize and accelerate the training process by reducing internal covariate shift.
- Formula:

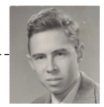
$$\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_b^2 + \epsilon}}$$



Normalization Methods

Outline

Advanced Architecture

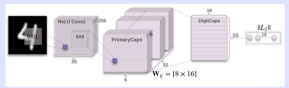


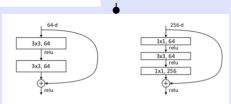
Frank Rosenblatt
Cornell University
psychologist

Geoffery Hinton
University of Toronto
deep learning

Yann Lecun
New York University
deep learning

Capsule Nets(2017)

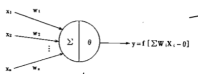


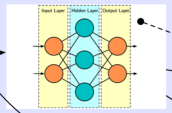


Kaiming He (何恺明)
MSRA => FAIR
computer vision

BPNN/MLP(1986)

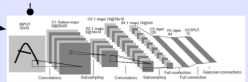
Perceptron(1958)

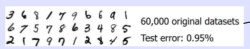




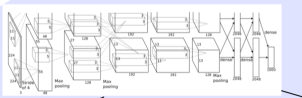
Neocognitron(1980)
[convolution & pooling]

LeNet/CNN(1998)





AlexNet(2012)



Relu, dropout & bigger

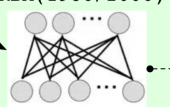
ResNet(2016)

VGG(2014)
GoogLeNet(2015)

DenseNet(2017)

Hopfield Network(1982)
[recurrent & feedback]

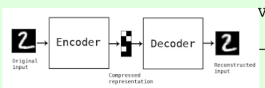
RBM(1986/2006)



stack

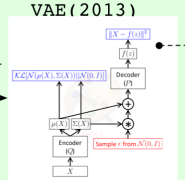
Deep Belief
Nets(2006)

AutoEncoder(1989/2006)
Denosing Autoencoder(2008)



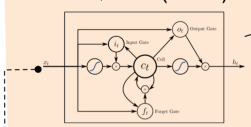
Variational
Inference

VAE(2013)



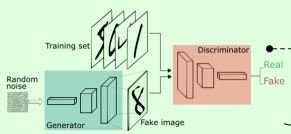
Max Welling
University of Amsterdam
statistical learning

RNN/LSTM(1997)



RNN in Speech
Recognition(2013)

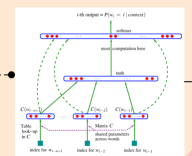
GAN(2014)



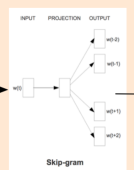
DCGAN(2014)
WGAN(2017)
PGGAN(2017)

Ian Goodfellow
Google Brain
deep adversarial learning

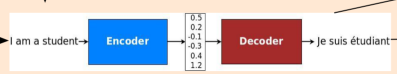
Neural Probabilistic
Language Model(2003)



word2Vec(2013)



Seq2Seq(2014)



SeqGAN(2017)
LeakGAN(2018)

Character CNN(2015)
self-attention(2017)

Jürgen Schmidhuber
IDSIA
Universal AI

Yoshua Bengio
University of Montreal
Deep learning

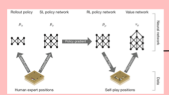
Deep Q-
learning(2013)

Double DQN(2015)
Dueling Net(2016)

David Silver
DeepMind
Reinforcement learning

DDPG(2015) → A3C(2016)

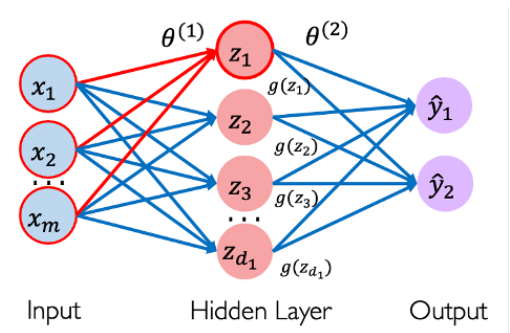
AlphaGo(2016)



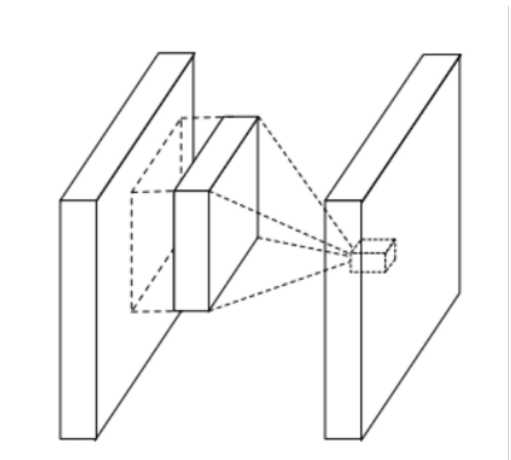
Alpha Zero(2017)

Today's neural network structure

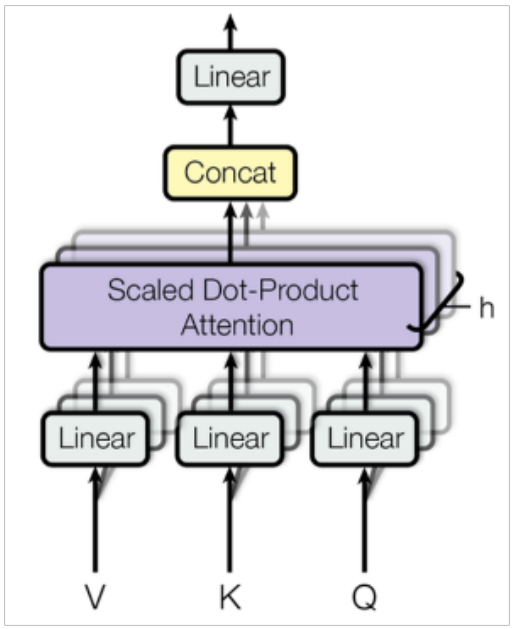
Typically, a deep neural network is composed of many layers of modules stacked together.
Basic modules: MLP block, CNN block, Attention block, RNN block ...



MLP block



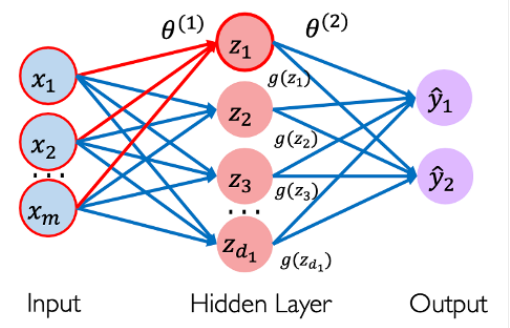
CNN block



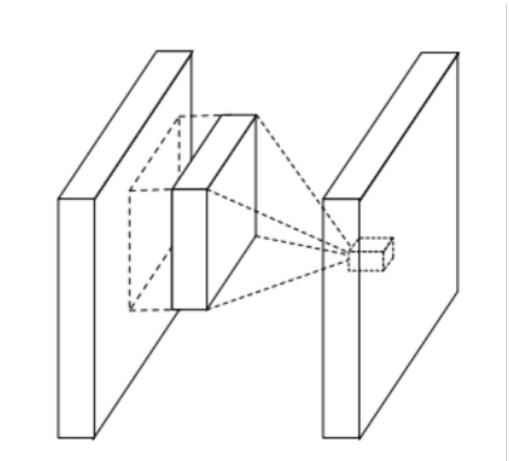
Attention block

Today's neural network structure

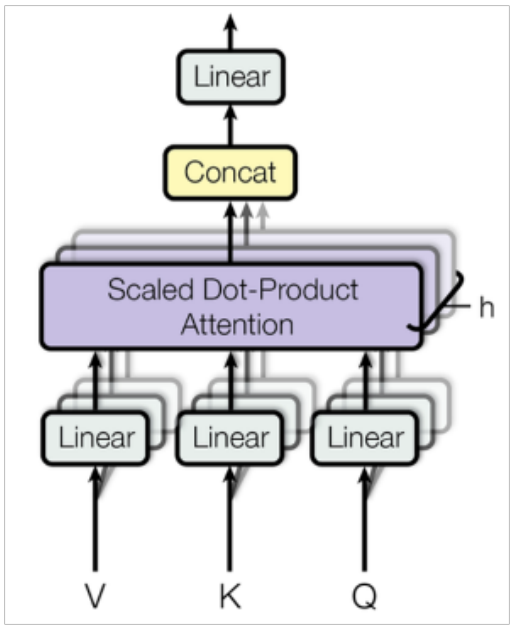
MLP block



CNN block



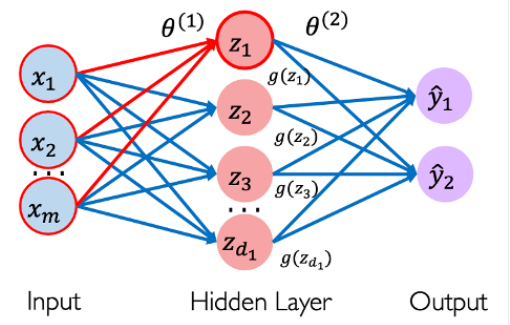
Attention block



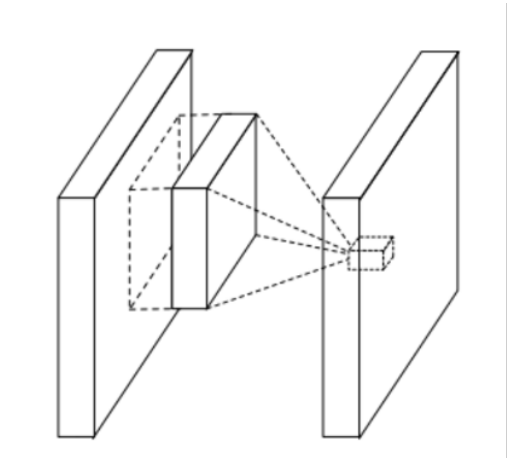
- Simple and flexible, capable of approximating any function
- Inefficient for input with large dimension, prone to overfitting.

Today's neural network structure

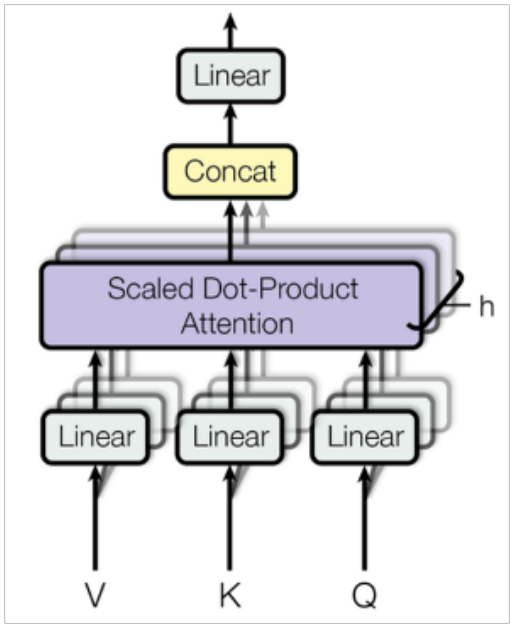
MLP block



CNN block



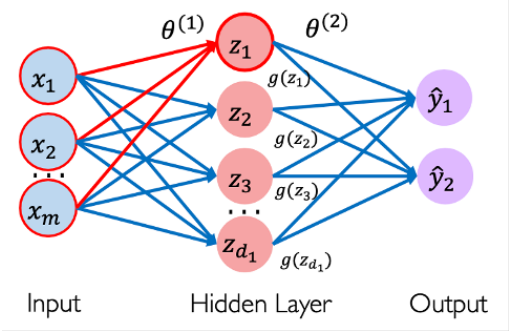
Attention block



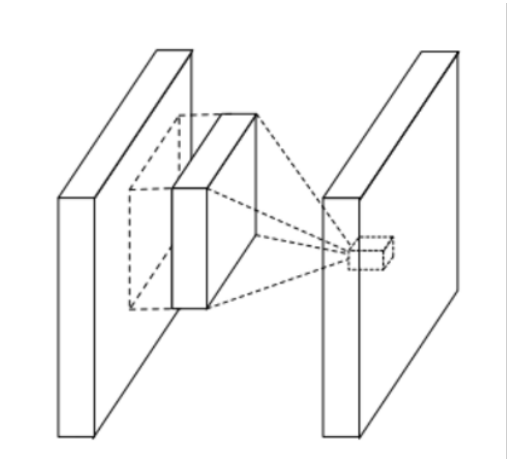
- Excellent for capturing local spatial patterns with fewer parameters. Used in Image task.
- Struggles with long-range dependencies and is less suited for unstructured data.

Today's neural network structure

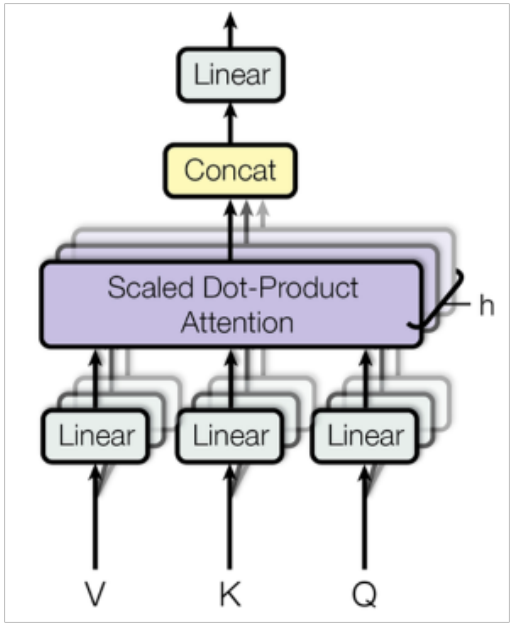
MLP block



CNN block



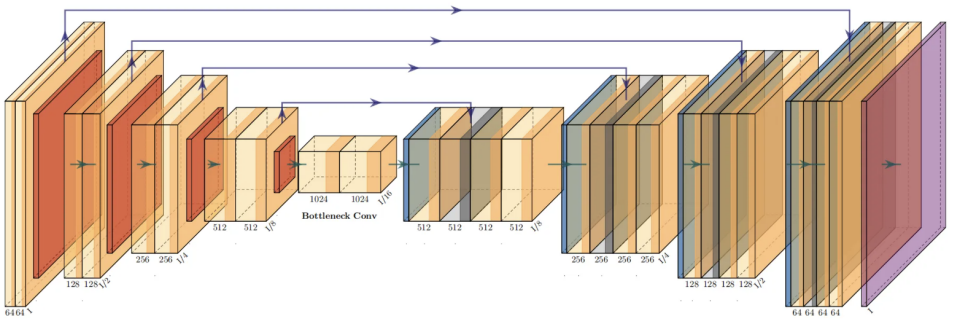
Attention block



- Efficient at capturing long-range dependencies and adaptable to various data types.
- Computationally expensive.

Today's neural network structure

- Example: UNet(stable diffusion version)

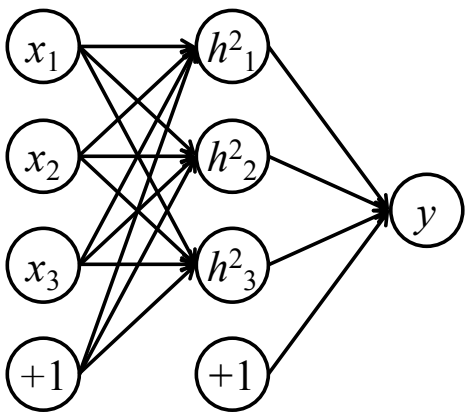


- In each resolution: N * CNN block + M * (Attetnion block + MLP block)
- And some trick for generation task: long residual connection

Outline

Summary

- Our first deep neural network—**deep forward network**.
- Model learning with forward and **backpropagation** algorithm.
- We also discuss the most popular **non-linear activation functions** used in hidden units.
- Finally, we discuss two most commonly-used **neural network architectures**.



Thanks.

HP: http://keg.cs.tsinghua.edu.cn/jietang/
Email: jietang@tsinghua.edu.cn