Neural networks

Victor Kitov

v.v.kitov@yandex.ru

Yandex School of Data Analysis

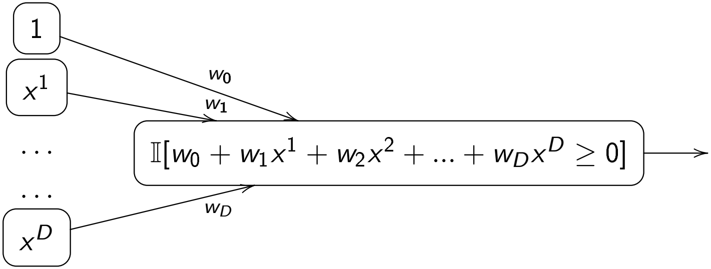
Table of Contents

History

- Neural networks originally appeared as an attempt to model human brain

- Human brain consists of multiple interconnected neuron cells
  - cerebral cortex (the largest part) is estimated to contain 15–33 billion neurons
  - communication is performed by sending electrical and electro-chemical signals
  - signals are transmitted through axons - long thin parts of neurons.

Simple model of a neuron

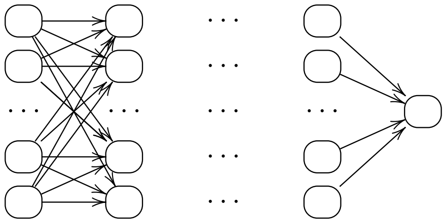

1
x^1
x^D
w_0
w_1
w_D
$\mathbb{I}[w_0 + w_1 x^1 + w_2 x^2 + ... + w_D x^D \geq 0]$

- Neuron get's activated in the half-space, defined by $w_0 + w_1 x^1 + w_2 x^2 + ... + w_D x^D \geq 0$.
- Each node is called a neuron.
- Each edge is associated a weight.
- Constant feature 1 stands for bias.

Multilayer perceptron architecture[1]

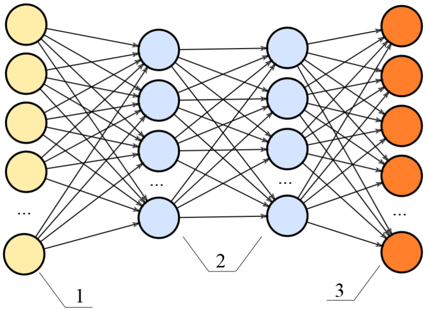
Multilayer perceptron:

- Each node has its own weights.
- Nodes&edges form directed acyclic graph.
- Multiple layers of neurons.
- Neurons of two consecutive layers are fully connected.



[1] Propose neural networks estimating OR,AND,XOR functions on boolean inputs.

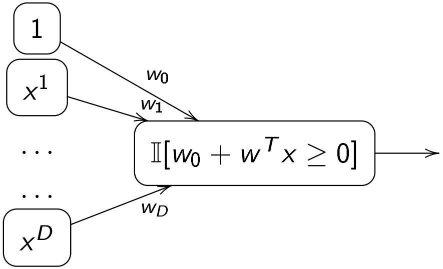
Layers

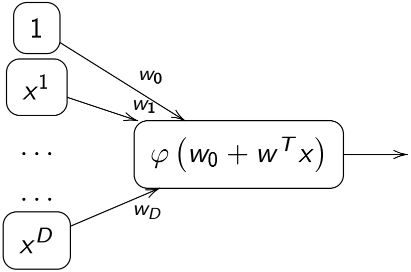


- Structure of neural network:
    - 1-input layer
    - 2-hidden layers
    - 3-output layer

Continious activations

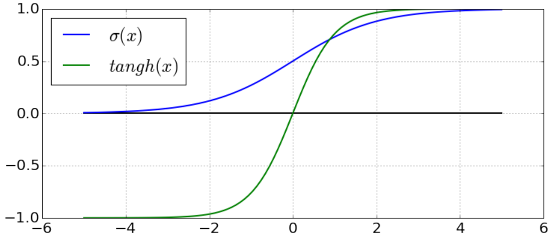
- Pitfall of $\mathbb{I}[]$: it causes stepwise constant outputs, weight optimization methods become inapliccable.
- We can replace $\mathbb{I}[w^T x + w_0 \geq 0]$ with smooth activation $\varphi(w^T x + w_0)$





Typical activation functions

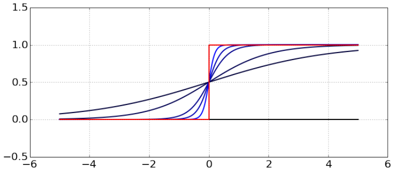
- sigmoidal: $\sigma(x) = \frac{1}{1+e^{-x}}$
  - 1-layer neural network with sigmoidal activation is equivalent to logistic regression
- hyperbolic tangent: $\text{tangh}(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$



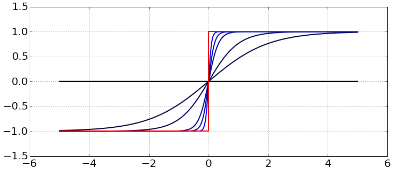
- ReLu: $\varphi(x) = [x]_{+}$.

Activation functions

Activation functions are smooth approximations of step functions:



$\sigma(ax)$ limits to 0/1-step function as $a \to \infty$



tangh(ax) limits to -1/1-step function as $a \to \infty$

Definition details

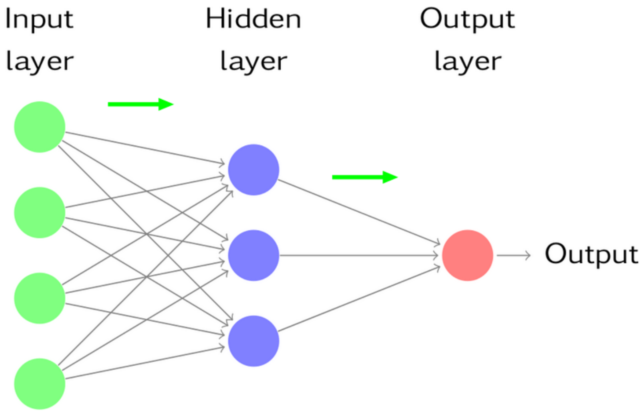
- Label each neuron with integer j.
- Denote: I_j - input to neuron j, O_j - output of neuron j
- Output of neuron j: $O_j = \varphi(I_j)$.
- Input to neuron j: $I_j = \sum_{k \in inc(j)} w_{kj} O_k + w_{0j}$,
  - w_{j0} is the bias term
  - $inc(j)$ is a set of neurons with outging edges incoming to neuron j.
  - further we will assume that at each layer there is a vertex with constant output $O_{const} \equiv 1$, so we can simplify notation

$$I_j = \sum_{k \in inc(j)} w_{kj} O_k$$

Table of Contents

Output generation

- Forward propagation is a process of successive calculations of neuron outputs for given features.



Activations at output layer

- Regression: $\varphi(I) = I$
- Classification:
  - binary: $y \in \{+1, -1\}$

$$\varphi(I) = p(y = +1|x) = \frac{1}{1 + e^{-I}}$$

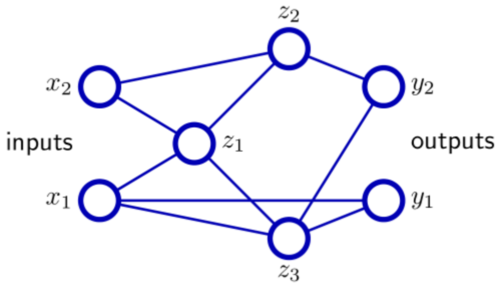
  - multiclass: $y \in \{1, 2, ...C\}$

$$\varphi(O_1, ...O_C) = p(y = j|x) = \frac{e^{O_j}}{\sum_{k=1}^{C} e^{O_k}}, j = 1, 2, ...C$$

where $O_1, ...O_C$ are outputs of output layer.

Generalizations

- each neuron j may have custom non-linear transformation φ_j
- weights may be constrained:
  - non-negative
  - equal weights
  - etc.
- layer skips are possible



- recurrent networks, RBF-activations.

Number of layers selection

- Number of layers usually denotes all layers except input layer (hidden layers+output layer)
- Classification with indicator activations:
    - single layer network selects arbitrary half-spaces
    - 2-layer network selects arbitrary convex polyhedron (by intersection of 1-layer outputs)
        - therefore it can approximate arbitrary convex sets
    - 3-layer network selects (by union of 2-layer outputs) arbitrary finite sets of polyhedra
        - therefore it can approximate all measurable sets

Number of layers selection

- Regression with indicator activations:
    - single layer can approximate arbitrary linear function
        - 2-layer network can model indicator function of arbitrary convex polyhedron
        - 3-layer network can uniformly approximate arbitrary continuous function (as sum weighted sum of indicators convex polyhedra)

Sufficient amount of layers

Any continuous function on a compact space can be uniformly approximated by 2-layer neural network with linear output and wide range of activation functions (excluding polynomial).

- In practice often it is more convenient to use more layers with less total amount of neurons
    - model becomes more interpretable and easy to fit.

Table of Contents

Network optimization: regression

- Single output:

$$\frac{1}{N}\sum_{n=1}^{N}(\widehat{y}_n(x_n|w) - y_n)^2 \to \min_w$$

Network optimization: regression

- Single output:

$$\frac{1}{N}\sum_{n=1}^{N}(\widehat{y}_n(x_n|w) - y_n)^2 \to \min_w$$

- K outputs

$$\frac{1}{NK}\sum_{n=1}^{N}\sum_{k=1}^{K}(\widehat{y}_{nk}(x_n|w) - y_{nk})^2 \to \min_w$$

Network optimization: classification

- Two classes, $y \in \{0, 1\}$:

$$\prod_{n=1}^{N} p(y_n = 1|x_n, w)^{y_n}[1 - p(y_n = 1|x_n, w)]^{1-y_n} \to \max_w$$

Network optimization: classification

- Two classes, $y \in \{0, 1\}$:

$$\prod_{n=1}^{N} p(y_n = 1|x_n, w)^{y_n}[1 - p(y_n = 1|x_n, w)]^{1-y_n} \to \max_w$$

- C classes, $y_{nc} = \mathbb{I}\{y_n = c\}$:

$$\prod_{n=1}^{N} \prod_{c=1}^{C} p(y_n = c|x_n, w)^{y_{nc}} \to \max_w$$

Network optimization: classification

- Two classes, $y \in \{0, 1\}$:

$$\prod_{n=1}^{N} p(y_n = 1|x_n, w)^{y_n}[1 - p(y_n = 1|x_n, w)]^{1-y_n} \to \max_{w}$$

- C classes, $y_{nc} = \mathbb{I}\{y_n = c\}$:

$$\prod_{n=1}^{N} \prod_{c=1}^{C} p(y_n = c|x_n, w)^{y_{nc}} \to \max_{w}$$

- In practice log-likelihood is maximized to avoid numerical underflow.

Neural network optimization

- Denote W - the total dimensionality of weights space, $w \in \mathbb{R}^W$.
- Let $E(\widehat{y}, y)$ denote the loss function of output
- We may optimize neural network using gradient descent:

```
$k = 0$
initialize randomly $w^0$ # small values for sigmoid and tangh

while (stop criteria not met):
    $w^{k+1} := w^k - \eta \nabla E(w^k)$
    $k := k + 1$
```

- Standardization of features makes gradient descend converge faster
- Other optimization methods are more efficient (such as conjugate gradients)

Gradient calculation

- Denote $\varepsilon_i = (0, ..., 0, \overbrace{\varepsilon}^{i\text{-th position}}, 0, ...0) \in \mathbb{R}^W$
- Direct $\nabla E(w)$ calculation, using

$$\frac{\partial E}{\partial w_i} = \frac{E(w+\varepsilon_i) - E(w)}{\varepsilon} + O(\varepsilon)$$

or better

$$\frac{\partial E}{\partial w_i} = \frac{E(w+\varepsilon_i) - E(w-\varepsilon_i)}{2\varepsilon} + O(\varepsilon^2)$$

has complexity $O(W^2)$ [W forward propagations to evaluate W derivatives]

Backpropagation algorithm needs only $O(W)$ to evaluate all derivatives.

Multiple local optima problem

- Optimization problem for neural nets is **non-convex**.
- Different optima will correspond to:
    - different starting parameter values
    - different training samples
- So we may solve task many times for different conditions and then
    - select best model
    - alternatively: average different obtained models to get ensemble

Table of Contents

Definitions

- Denote w_{ij} be the weight of edge, connecting i-th and j-th neuron.
- Define $\delta_j = \frac{\partial E}{\partial I_j} = \frac{\partial E}{\partial O_j}\frac{\partial O_j}{\partial I_j}$
- Since E depends on w_{ij} through the following functional relationship $E(w_{ij}) \equiv E(O_j(I_j(w_{ij})))$, using the chain rule we obtain:
$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial I_j}\frac{\partial I_j}{\partial w_{ij}} = \delta_j O_i$$
because $\frac{\partial I_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k\in inc(j)} w_{kj} O_k\right) = O_i$, where $inc(j)$ is a set of all neurons with outgoing edges to neuron j.
- $\frac{\partial E}{\partial I_j} = \frac{\partial E}{\partial O_j}\frac{\partial O_j}{\partial I_j} = \frac{\partial E}{\partial O_j}\varphi'(I_j)$, where φ is the activation function.

Output layer

- If neuron j belongs to the output node, then error $\frac{\partial E}{\partial O_j}$ is calculated directly.
- For output layer deltas are calculated directly:

$$\delta_j = \frac{\partial E}{\partial O_j}\frac{\partial O_j}{\partial I_j} = \frac{\partial E}{\partial O_j}\varphi'(I_j) \tag{1}$$

- example for training set = {single point x and true vector of outputs $(y_1, ...y_{|OL|})$}:
    - for $E = \frac{1}{2}\sum_{j\in OL}(O_j - y_j)^2$:

$$\frac{\partial E}{\partial O_j} = O_j - y_j$$

    - for $\varphi(I) = sigm(I)$:

$$\varphi'(I_j) = \sigma(I_j)\,(1-\sigma(I_j)) = O_j(1-O_j)$$

    - finally

$$\delta_j = (O_j - y_j)\,O_j(1-O_j)$$

Inner layer

- If neuron j belongs some hidden layer, denote $out(j) = \{k_1, k_2, ...k_m\}$ the set of all neurons, receiving output from neuron j.
- The effect of O_j on E is fully absorbed by $I_{k_1}, I_{k_2}, ...I_{k_m}$, so

$$\frac{\partial E(O_j)}{\partial O_j} = \frac{\partial E(I_{k_1}, I_{k_2}, ...I_{k_m})}{\partial O_j} = \sum_{k \in out(j)} \left(\frac{\partial E}{\partial I_k} \frac{\partial I_k}{\partial O_j} \right) = \sum_{k \in out(j)} (\delta_k w_{jk})$$

- So for layers other than output layer we have:

$$\delta_j = \frac{\partial E}{\partial I_j} = \frac{\partial E}{\partial O_j} \frac{\partial O_j}{\partial I_j} = \sum_{k \in out(j)} (\delta_k w_{jk}) \varphi'(I_j) \qquad (2)$$

- Weight derivatives are calculated using errors and outputs:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial I_j} \frac{\partial I_j}{\partial w_{ij}} = \delta_j O_i \qquad (3)$$

Backpropagation

- Backpropagation algorithm:
  1. Forward propagate x_n to the neural network, store all inputs I_i and outputs O_i for each neuron.
  2. Calculate δ_i for all $i \in$output layer using (1).
  3. Backpropagate δ_i from final layer backwards layer by layer using (2).
  4. Using calculated deltas and outputs calculate $\frac{\partial E}{\partial w_{ij}}$ with (3).
- Algorithm complexity: $O(W)$, where W is total number of edges.
- Updates:
  - batch
  - stochastic
    - using minibatches of objects

Regularization

- Constrain model complexity directly
    - constrain number of neurons
    - constrain number of layers
    - impose constraints on weights
- Take a flexible model
    - use early stopping during iterative evaluation (by controlling validation error)
    - quadratic regularization

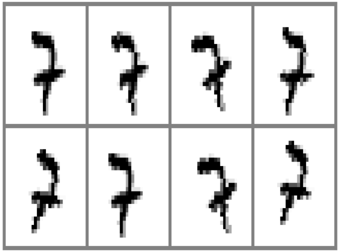
$$\tilde{E}(w) = E(w) + \lambda \sum_i w_i^2$$

- Stochastic simplification of model, such as dropout.

Table of Contents

Invariance to particular transformations

- It may happen that solution should not depend on certain kinds of transformations in the input space.
- Example: character recognition task
    - translation invariance
    - scale invariance
    - invariance to small rotations
    - invariance to small uniform noise



Invariance to particular transformations

- Approaches to build an invariant model to particular transformation:
  - use only features that are invariant to transformations
  - augment training objects with their transformed copies according to given invariances
    - amount of possible transformations grows exponentially with the number of invariances
  - add regularization term to the target cost function, which penalizes changes in output after invariant transformations
    - see tangent propagation
  - build the invariance properties into the structure of neural network
    - see convolutional neural networks

Augmentation of training samples

1. generate a random set of invariant transformations
2. apply these transformations to training objects
3. obtain new training objects

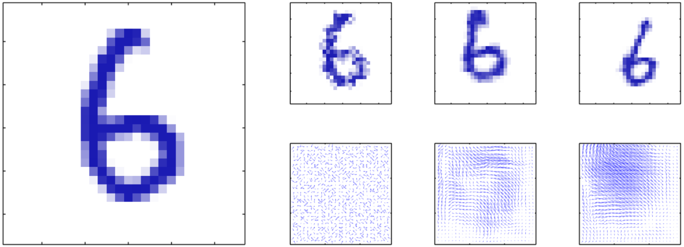
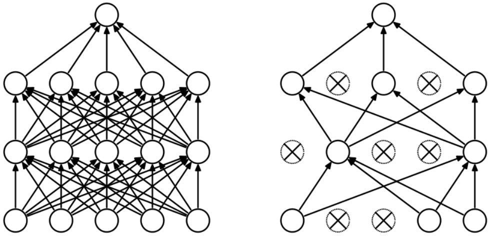


Table of Contents

Dropout idea

Each neuron is left or removed independently from decisions about other nodes:



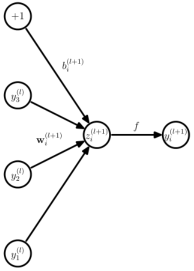
(a) Standard Neural Net (b) After applying dropout.

- Output layer nodes are never removed.
- Recommended[2] neuron removal probability $1 - p$:
    - $p = 0.5$ for inner layer nodes
    - $p = 0.8$ for input layer nodes (feature subsampling)

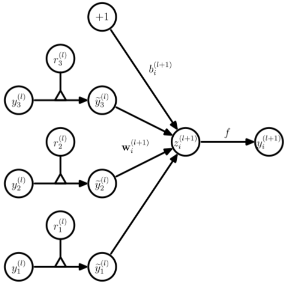
[2]Removal probabilities can be fine-tuned on cross-validation.

Dropout algorithm

- Dropout forces neurons to rely less on other neurons and to learn something valuable by themselves.
    - neurons learning becomes more robust to information learnt by other neurons.
    - resulting network becomes less overfitted.
- $r_i^{(l)}$ are i.i.d. random variables having *Bernoulli*(p) distribution.



(a) Standard network



(b) Dropout network

Definitions

Define:

- $f(x)$ - an activation function.
- y^l - vector of outputs at layer l
- z^l - vector of inputs to layer l
- L - number of layers in neural network
- $y^{(0)} = x$ - input feature vector
- $Bernoulli(p)$ returns a vector of independent Bernoulli random variables with parameter p.

Forward propagation algorithm

We need to repeat forward propagation recurrently for $l = 0, 1, ... L - 1$.

1. Usual feed-forward neural network:

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

2. Feed-forward network with dropout:

$$\mathbf{r_j^{(l)}} \sim \mathbf{Bernoulli(p)}$$
$$\mathbf{\tilde{y}^l} = \mathbf{r^{(l)}} * \mathbf{y^{(l)}}$$
$$z_i^{(l+1)} = \left(w_i^{(l+1)}\right)^T \tilde{y}^l + b_i^{(l+1)}$$
$$y_i^{(l+1)} = f(z_i^{(l+1)})$$

Application of dropout

- **Learning**
    - while weights not converge:
        1. sample random subnetwork (“thinned network”) with dropout
        2. apply one step of stochastic gradient descent to thinned network

  Comment: due to weights sharing across all thinned networks the number of parameters is the same as in original network.
- **Prediction**
    - use full networks with all nodes, but multiply each weight by p
    - such scaling approximates average output of randomly sampled thinned network[3].

[3]Approximation is precise for networks without non-linearities. With non-linearities - not. We may also use Monte-Carlo sampling in the latter case.

Conclusion on neural networks

- Advantages of neural networks:
  - can model accurately complex non-linear relationships
  - easily parallelizable
- Disadvantages of neural networks:
  - hardly interpretable (“black-box” algorithm)
  - optimization requires skill
    - too many parameters
    - may converge slowly
    - may converge to inefficient local minimum far from global one