

Basic Math for Neural Networks

Xiaodong Cui

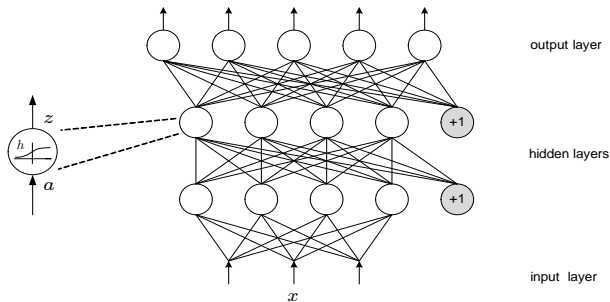
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598

Spring, 2017

Outline

- Feedforward neural networks
- Forward propagation
- Neural networks as universal approximators
- Back propagation
- Jacobian
- Vanishing gradient problem

Feedforward Neural Networks



$$a_i^{(l)} = \sum_j w_{ij}^{(l)} z_j^{(l-1)} + b_i^{(l)}$$

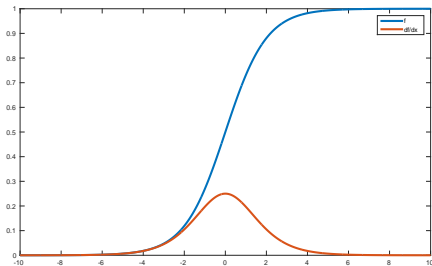
$$z_i^{(l)} = h_i^{(l)}(a_i^{(l)})$$

Activation Functions – Sigmoid

also known as logistic function

$$h(a) = \frac{1}{1 + e^{-a}}$$

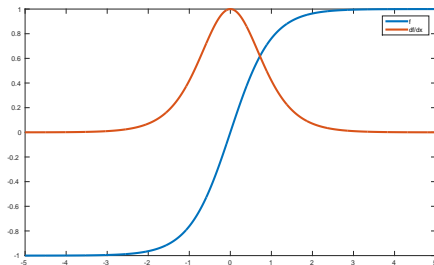
$$h'(a) = \frac{e^{-a}}{(1 + e^{-a})^2} = h(a)[1 - h(a)]$$



Activation Functions – Hyperbolic Tangent (tanh)

$$h(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

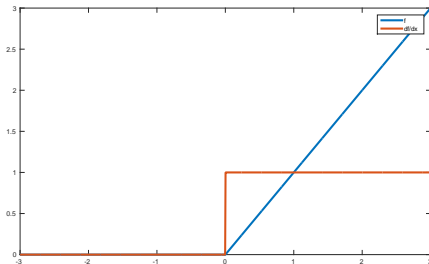
$$h'(a) = \frac{4}{(e^a + e^{-a})^2} = 1 - [h(a)]^2$$



Activation Functions – Rectified Linear Unit (ReLU)

$$h(a) = \max(0, a) = \begin{cases} a & \text{if } a > 0, \\ 0 & \text{if } a \leq 0. \end{cases}$$

$$h'(a) = \begin{cases} 1 & \text{if } a > 0, \\ 0 & \text{if } a \leq 0 \end{cases}$$



Activation Functions – Softmax

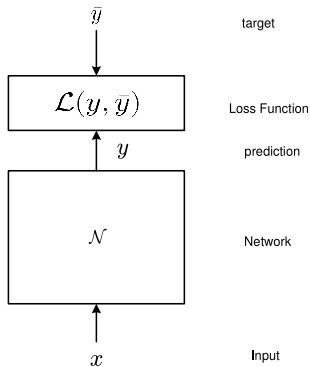
also known as normalized exponential function, a generalization of the logistic function to multiple classes.

$$h(a_k) = \frac{e^{a_k}}{\sum_j e^{a_j}}$$
$$\frac{\partial h(a_k)}{\partial a_j} = \frac{\partial h(a_k)}{\partial \log h(a_k)} \frac{\partial \log h(a_k)}{\partial a_j} = h(a_k) \left(\delta_{kj} - h(a_j) \right)$$

Why "softmax"?

$$\max\{x_1, \dots, x_n\} \leq \log(e^{x_1} + \dots + e^{x_n}) \leq \max\{x_1, \dots, x_n\} + \log n$$

Loss Functions



- Regression
- Classification

Regression: Least Square Error

Let $y_{n,k}$ and $\bar{y}_{n,k}$ be the output and target respectively. For regression, the typical loss function is least square error (LSE):

$$\begin{aligned}\mathcal{L}(y, \bar{y}) &= \frac{1}{2} \sum_n (y_n - \bar{y}_n)^2 \\ &= \frac{1}{2} \sum_n \sum_k (y_{nk} - \bar{y}_{nk})^2\end{aligned}$$

where n is the sample index and k is the dimension index. The derivative of LSE with respect to each dimension of each sample:

$$\frac{\partial \mathcal{L}(y, \bar{y})}{\partial y_{nk}} = (y_{nk} - \bar{y}_{nk})$$

Classification: Cross-Entropy (CE)

Suppose there are two (discrete) probability distributions p and q , the "distance" between the two distributions can be measured by the Kullback-Leibler divergence:

$$D_{\text{KL}}(p||q) = \sum_k p_k \log \frac{p_k}{q_k}$$

For classification, the targets \bar{y}_n are given

$$y_n = \{\bar{y}_{n1}, \dots, \bar{y}_{nk}, \dots, \bar{y}_{nK}\}$$

The predictions after the softmax layer are posterior probabilities after normalization

$$y_n = \{y_{n1}, \dots, y_{nk}, \dots, y_{nK}\}$$

Now measure the "distance" of these two distributions from all samples

$$\begin{aligned} \sum_n D_{\text{KL}}(\bar{y}_n||y_n) &= \sum_n \sum_k \bar{y}_{nk} \log \frac{\bar{y}_{nk}}{y_{nk}} = \sum_n \sum_k \bar{y}_{nk} \log \bar{y}_{nk} - \sum_n \sum_k \bar{y}_{nk} \log y_{nk} \\ &= - \sum_n \sum_k \bar{y}_{nk} \log y_{nk} + \mathcal{C} \end{aligned}$$

Classification: Cross-Entropy (CE)

Cross-entropy as the loss function:

$$\mathcal{L}(\bar{y}, y) = - \sum_n \sum_k \bar{y}_{nk} \log y_{nk}$$

Typically, targets are given in the form of 1-of-K coding

$$\bar{y}_n = \{0, \dots, 1, \dots, 0\}$$

where

$$\bar{y}_n = \begin{cases} 1 & \text{if } k = \tilde{k}_n, \\ 0 & \text{otherwise} \end{cases}$$

It follows that the cross-entropy has an even simpler form:

$$\mathcal{L}(\bar{y}, y) = - \sum_n \log y_{n\tilde{k}_n}$$

Classification: Cross-Entropy (CE)

derivative of the composite function of cross-entropy and softmax:

$$\mathcal{L}(\bar{y}, y) = \sum_n \mathcal{E}_n = \sum_n \left(- \sum_i \bar{y}_{ni} \log y_{ni} \right)$$

$$y_{ni} = \frac{e^{a_{ni}}}{\sum_j e^{a_{nj}}}$$

$$\begin{aligned} \frac{\partial \mathcal{E}_n}{\partial a_{nk}} &= \frac{\partial}{\partial a_{nk}} \left(- \sum_i \bar{y}_{ni} \log y_{ni} \right) = - \sum_i \bar{y}_{ni} \frac{\partial \log y_{ni}}{\partial a_{nk}} \\ &= - \sum_i \bar{y}_{ni} \frac{\partial \log y_{ni}}{\partial y_{ni}} \frac{\partial y_{ni}}{\partial a_{nk}} = - \sum_i \frac{\bar{y}_{ni}}{y_{ni}} \frac{\partial y_{ni}}{\partial a_{nk}} \frac{\partial \log y_{ni}}{\partial a_{nk}} \\ &= - \sum_i \frac{\bar{y}_{ni}}{y_{ni}} \cdot y_{ni} \cdot (\delta_{ik} - y_{nk}) = - \sum_i \bar{y}_{ni} (\delta_{ik} - y_{nk}) \\ &= - \sum_i \bar{y}_{ni} \delta_{ik} + \sum_i \bar{y}_{ni} y_{nk} = -\bar{y}_{nk} + y_{nk} \left(\sum_i \bar{y}_{ni} \right) \\ &= y_{nk} - \bar{y}_{nk} \end{aligned}$$

What's the derivative for an arbitrary differentiable function? – leave it for exercise

A DNN Example In Torch

Algorithm 1 Definition of a DNN with 3 hidden layers

```
function create_model()

  - MODEL
  local n_inputs = 460
  local n_outputs = 3000
  local n_hidden = 1024
  local model = nn.Sequential()
  model:add(nn.Linear(n_inputs, n_hidden))
  model:add(nn.Sigmoid())
  model:add(nn.Linear(n_hidden,n_hidden))
  model:add(nn.Sigmoid())
  model:add(nn.Linear(n_hidden,n_hidden))
  model:add(nn.Sigmoid())
  model:add(nn.Linear(n_hidden,n_outputs))
  model:add(nn.LogSoftMax())

  - LOSS FUNCTION
  local criterion = nn.ClassNLLCriterion()

  return model:cuda(), criterion:cuda()
end
```

Neural Networks As Universal Approximators

Universal Approximation Theorem[1][2][3] Let $\phi(\cdot)$ be a nonconstant, bounded, and monotonically-increasing continuous function. Let I_n represent an n -dimensional unit cube $[0, 1]^n$ and $C(I_n)$ be the space of continuous functions on I_n . Then given any function $f \in C(I_n)$ and $\epsilon > 0$, there exist an integer N and real constants $a_i, b_i \in \mathbb{R}$ and $w_i \in \mathbb{R}^n$, where $i = 1, 2, \dots, N$, such that one may define

$$F(x) = \sum_{i=1}^N a_i \phi(w_i^T x + b_i)$$

as an approximate realization of the function f where f is independent of ϕ such that

$$|F(x) - f(x)| < \epsilon$$

for all $x \in I_n$.

[1] Cybenko, G. (1989). "Approximation by Superpositions of a Sigmoidal Function," Math. Control Signals Systems, 2, 303-314.

[2] Hornik, K., Stinchcombe, M., and White, H. (1989). "Multilayer Feedforward Networks are Universal Approximators," Neural Networks, 2(5), 359-366.

[3] Hornik, K. (1991). "Approximation Capabilities of Multilayer Feedforward Networks," Neural Networks, 4(2), 251-257.

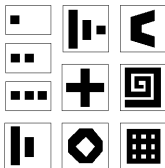
Neural Networks As Universal Classifiers

Extends $f(x)$ to decision functions of the form [1]

$$f(x) = j, \text{ iff } x \in P_j, \quad j = 1, 2, \dots, K$$

where P_j partition A_n into K disjoint measurable subsets and A_n is a compact subset of \mathbb{R}^n .

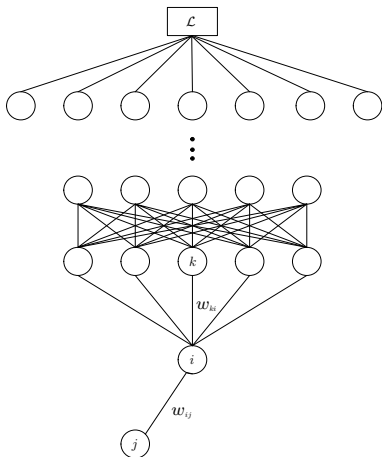
Arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity!



[1] Cybenko, G. (1989). "Approximation by Superpositions of a Sigmoidal Function," Math. Control Signals Systems, 2, 303-314.

[2] Huang, W. Y. and Lippmann, R. P. (1988). "Neural Nets and Traditional Classifiers," in Neural Information Processing Systems (Denver 1987), D. Z. Anderson, Editor. American Institute of Physics, New York, 387-396.

Back Propagation



How to compute the gradient of the loss function with respect to a weight w_{ij} ?

$$\frac{\partial \mathcal{L}}{\partial w_{ij}}$$

D. E. Rumelhart, G. E. Hinton and R. J. Williams (1986). "Learning representations by back-propagating errors," Nature, vol. 323, 533-536.

Back Propagation

$$\begin{cases} a_i^{(l)} &= \sum_j w_{ij}^{(l)} z_j^{(l-1)} \\ z_i^{(l)} &= h_i^{(l)}(a_i^{(l)}) \end{cases}$$

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial w_{ij}^{(l)}}$$

It follows that

$$\frac{\partial a_i^{(l)}}{\partial w_{ij}^{(l)}} = z_j^{(l)} \quad \text{and} \quad \delta_i^{(l)} \triangleq \frac{\partial \mathcal{L}}{\partial a_i^{(l)}}$$

δ_i is often referred to as “errors”. By the chain rule,

$$\delta_i^{(l)} = \frac{\partial \mathcal{L}}{\partial a_i^{(l)}} = \sum_k \frac{\partial \mathcal{L}}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial a_i^{(l)}} = \sum_k \delta_k^{(l+1)} \frac{\partial a_k^{(l+1)}}{\partial a_i^{(l)}}$$

Back Propagation

$$a_k^{(l+1)} = \sum_i w_{ki}^{(l+1)} h(a_i^{(l)})$$

therefore

$$\frac{\partial a_k^{(l+1)}}{\partial a_i^{(l)}} = w_{ki}^{(l+1)} h'(a_i^{(l)})$$

It follows that

$$\delta_i^{(l)} = h'(a_i^{(l)}) \sum_k w_{ki}^{(l+1)} \delta_k^{(l+1)}$$

which tells us that the errors can be evaluated recursively.

Back Propagation: An Example

- A neural network with multiple layers
- Softmax output layer
- Cross-entropy loss function
- Forward propagation:
 - ▶ push the input x_n through the network to get the activations to the hidden layers

$$a_i^{(l)} = \sum_j w_{ij}^{(l)} z_j^{(l-1)}, \quad z_i^{(l)} = h_i^{(l)}(a_i^{(l)})$$

- Back propagation:
 - ▶ start from the output layer

$$\delta_k^{(L)} = y_{nk} - \bar{y}_{nk}$$

- ▶ back-propagate errors from the output layer all the way down to the input layer

$$\delta_i^{(l)} = h'(a_i^{(l)}) \sum_k w_{ki}^{(l+1)} \delta_k^{(l+1)}$$

- ▶ evaluate gradients

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} z_j^{(l)}$$

Gradient Checking

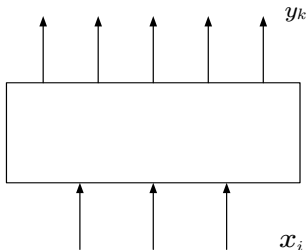
In practice, when you implement the gradient of a network or a module, one way to check the correctness of your implementation is via the following gradient checking by a (symmetric) finite difference:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\mathcal{L}(w_{ij} + \epsilon) - \mathcal{L}(w_{ij} - \epsilon)}{2\epsilon}$$

check the "closeness" of the two gradients (your own implementation and the one from the above difference)

$$\frac{\|\mathbf{g}_1 - \mathbf{g}_2\|}{\|\mathbf{g}_1 + \mathbf{g}_2\|}$$

Jacobian Matrix



- measure the sensitivity of outputs with respect to inputs of a (sub-)network
- can be used as a modular operation for error backprop in a larger network

Jacobian Matrix

The Jacobian matrix can be computed using a similar back-propagation procedure.

$$\frac{\partial y_k}{\partial x_i} = \sum_l \frac{\partial y_k}{\partial a_l} \frac{\partial a_l}{\partial x_i}$$

where a_l are the activation functions having immediate connections with the inputs x_i .

$$\frac{\partial y_k}{\partial x_i} = \sum_l w_{li} \frac{\partial y_k}{\partial a_l}$$

Analogous to the error propagation, define

$$\delta_{kl} \triangleq \frac{\partial y_k}{\partial a_l}$$

and it can be computed recursively

$$\delta_{kl} = \frac{\partial y_k}{\partial a_l} = \sum_j \frac{\partial y_k}{\partial a_j} \frac{\partial a_j}{\partial a_l} = \sum_j \frac{\partial y_k}{\partial a_j} [w_{jl} h'(a_l)] = \sum_j \delta_{kj} [w_{jl} h'(a_l)]$$

where j sweeps all the connections w_{jl} with l .

What's the Jacobian matrix of softmax outputs to the inputs? – leave it as an exercise

Vanishing Gradients

function	nonlinearity	gradient
sigmoid	$x = \frac{1}{1+e^{-a}}$	$\frac{\partial \mathcal{L}}{\partial a} = x(1-x) \frac{\partial \mathcal{L}}{\partial x}$
tanh	$x = \frac{e^a - e^{-a}}{e^a + e^{-a}}$	$\frac{\partial \mathcal{L}}{\partial a} = (1-x^2) \frac{\partial \mathcal{L}}{\partial x}$
softsign	$x = \frac{a}{1+ a }$	$\frac{\partial \mathcal{L}}{\partial a} = (1- x)^2 \frac{\partial \mathcal{L}}{\partial x}$

What's the problem???

$$|x| \leq 1 \quad \rightarrow \quad \frac{\partial \mathcal{L}}{\partial a} \leq \frac{\partial \mathcal{L}}{\partial x}$$

Nonlinearity causes vanishing gradients.