Natural Language Processing
Neural Networks

Felipe Bravo-Marquez

September 16, 2019

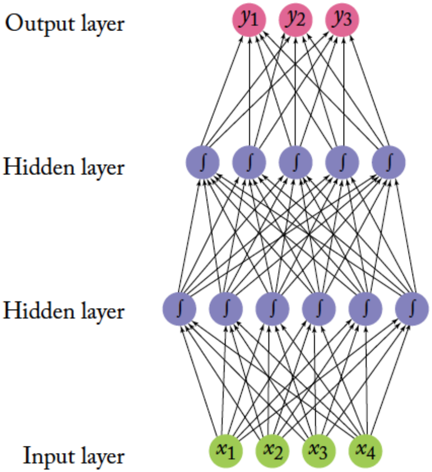
Introduction to Neural Networks

- Very popular machine learning models formed by units called **neurons**.
- A neuron is a computational unit that has scalar inputs and outputs.
- Each input has an associated weight w.
- The neuron multiplies each input by its weight, and then sums them (other functions such as **max** are also possible).
- It applies an activation function g (usually non-linear) to the result, and passes it to its output.
- Multiple layers can be stacked.

Activation Functions

- The nonlinear activation function g has a crucial role in the network's ability to represent complex functions.
- Without the nonlinearity in g, the neural network can only represent linear transformations of the input.

Feedforward Network with two Layers



[0]Source:[Goldberg, 2017]

Feedforward Network Neural Networks

- The feedforward network from the picture is a stack of linear models separated by nonlinear functions.
- The values of each row of neurons in the network can be thought of as a vector.
- The input layer is a 4-dimensional vector ($\vec{x}$), and the layer above it is a 6-dimensional vector ($\vec{h}^1$).
- The fully connected layer can be thought of as a linear transformation from 4 dimensions to 6 dimensions.
- A fully connected layer implements a vector-matrix multiplication, $\vec{h} = \vec{x}W$.
- The weight of the connection from the i-th neuron in the input row to the j-th neuron in the output row is $W_{[i,j]}$.
- The values of $\vec{h}$ are transformed by a nonlinear function g that is applied to each value before being passed on as input to the next layer.

---

[0] Vectors are assumed to be row vectors and superscript indices correspond to network layers.

Neural Netoworks as Mathematical Functions

- The Multilayer Perceptron (MLP) from the figure is called MLP2 because it has two hidden layers.
- A simpler model would be MLP1, a multilayer perceptron of one hidden layer:

$$\begin{gathered}\vec{y} = NN_{MLP1}(\vec{x}) = g(\vec{x}W^1 + \vec{b}^1)W^2 + \vec{b}^2 \\ \vec{x} \in \mathcal{R}^{in}, W^1 \in \mathcal{R}^{d_{in} \times d_1}, \vec{b}^1 \in \mathcal{R}^{d_{in}}, W^2 \in \mathcal{R}^{d_1 \times d_{out}}, \vec{b}^2 \in \mathcal{R}^{d_{out}}, \vec{y} \in \mathcal{R}^{d_{out}}\end{gathered} \tag{1}$$

- Here W^1 and $\vec{b}^1$ are a matrix and a bias term for the first linear transformation of the input.
- The function g is a nonlinear function that is applied element-wise (also called a nonlinearity or an activation function).
- W^2 and $\vec{b}^2$ are the matrix and bias term for a second linear transform.
- When describing a neural network, one should specify the dimensions of the layers (d_1), the input (d_{in}), and the output (d_{out}).

Neural Netoworks as Mathematical Functions

- MLP2 can be written as the following mathematical function:

$$
\begin{aligned}
NN_{MLP2}(\vec{x}) &= \hat{\vec{y}} \\
\vec{h}^1 &= g^1(\vec{x}W^1 + \vec{b}^1) \\
\vec{h}^2 &= g^2(\vec{h}^1 W^2 + \vec{b}^2) \\
\vec{y} &= \vec{h}^2 W^3 \\
\vec{y} &= (g^2(g^1(\vec{x}W^1 + \vec{b}^1)W^2 + \vec{b}^2))W^3.
\end{aligned}
\tag{2}
$$

- The matrices and the bias terms that define the linear transformations are the parameters of the network.
- Like in linear models, it is common to refer to the collection of all parameters as Θ.

Representation Power

- [Hornik et al., 1989] and [Cybenko, 1989] showed that a multilayer perceptron of one hidden later (MLP1) is a universal approximator.
- MLP1 can approximate all continuous functions on a closed and bounded subset of $\mathcal{R}^n$.
- This may suggest there is no reason to go beyond MLP1 to more complex architectures.
- The result does not say how easy or hard it is to set the parameters based on training data and a specific learning algorithm.
- It also does not guarantee that a training algorithm will find the correct function generating our training data.
- Finally, it does not state how large the hidden layer should be.

Representation Power

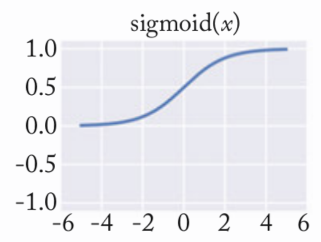
- In practice, we train neural networks on relatively small amounts of data using local search methods.
- We also use hidden layers of relatively modest sizes (up to several thousands).
- The universal approximation theorem does not give any guarantees under these conditions.
- However, there is definitely benefit in trying out more complex architectures than MLP1.
- In many cases, however, MLP1 does indeed provide strong results.

Activation Functions

- The nonlinearity g can take many forms.
- There is currently no good theory as to which nonlinearity to apply in which conditions.
- Choosing the correct nonlinearity for a given task is for the most part an empirical question.

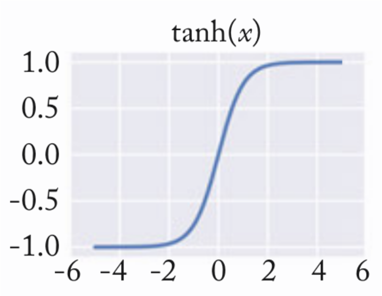
Sigmoid

- The sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$ is an S-shaped function, transforming each value x into the range [0, 1].
- The sigmoid was the canonical nonlinearity for neural networks since their inception.
- Is currently considered to be deprecated for use in internal layers of neural networks, as the choices listed next prove to work much better empirically.



Hyperbolic tangent (tanh)

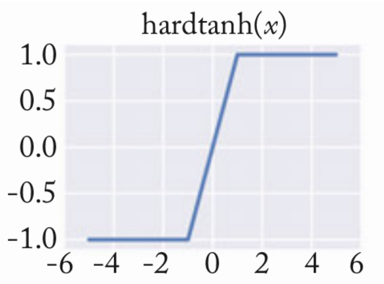
- The hyperbolic tangent $\mathtt{tanh}(x) = \frac{e^{2x}-1}{e^{2x}+1}$ activation function is an S-shaped function, transforming the values x into the range$[-1, 1]$.



Hard tanh

- The hard-tanh activation function is an approximation of the tanh function which is faster to compute and to find derivatives thereof:

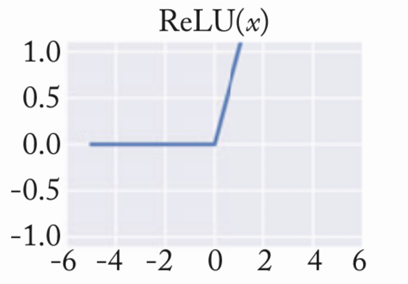
$$\mathrm{hardtanh}(x) = \left\{ \begin{array}{ll} -1 & x < -1 \\ 1 & x > 1 \\ x & \text{otherwise.} \end{array} \right\}$$



ReLU

- The rectifier activation function [Glorot et al., 2011], also known as the recti fied linear unit is a very simple activation function.
- It is easy to work with and was shown many times to produce excellent results.
- The ReLU unit clips each value $x < 0$ at 0.

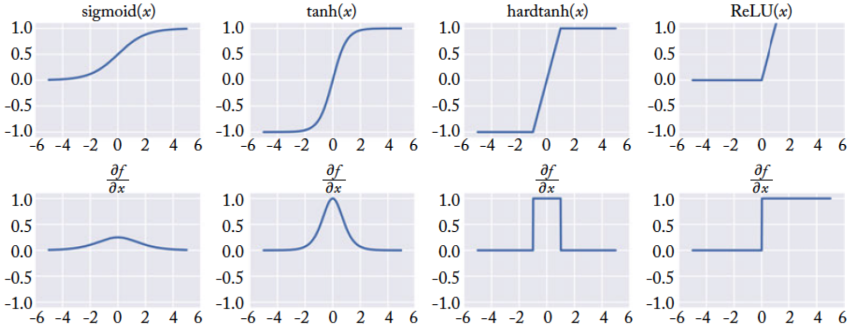
$$\mathrm{ReLU}(x) = \max(0, x)$$



- It performs well for many tasks, especially when combined with the dropout regularization technique (to be explained later).

Activation Functions

- As a rule of thumb, both ReLU and tanh units work well, and significantly outperform the sigmoid.
- You may want to experiment with both tanh and ReLU activations, as each one may perform better in different settings.
- The figure from below shows the shapes of the different activations functions, together with the shapes of their derivatives.



---

[0]Source:[Goldberg, 2017]

Backpropagation

- Neural networks are trained in the same way as linear models.
- The network's output is used to compute a loss fuction $L(\hat{y}, y)$ that is minimized across the training examples using gradient descent.
- Backpropagation is an efficient technique for evaluating the gradient of a loss function L for a feed-forward neural network with respect to all its parameters [Bishop, 2006]. [1]
- Those parameters are: $W^1, \vec{b}^1, \ldots, W^m, \vec{b}^m$, for a network of m layers.
- Recall that superscripts are used to denote layer indexes (not exponentiations).
- For simplicity, we will assume that L is calculated over a single example.
- Challenge: in neural networks the number of parameters can be huge and we need an efficient way to calculate the gradients.
- Idea: apply the derivate chain rule wisely.

[1] The following slides on backpropagation are based on [Bishop, 2006], we adapted the notation to be consistent with [Goldberg, 2017].

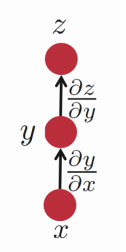
Derivative Chain Rule Recap

- Simple chain rule: let $z = f(y)$, $y = g(x)$,

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \times \frac{\partial y}{\partial x}$$

- Example: $z = e^y$, $y = 2x$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \times \frac{\partial y}{\partial x} = e^y \times 2 = 2e^{2x}$$



[1] Figure taken from:
`http://cs224d.stanford.edu/lectures/CS224d-Lecture5.pdf`

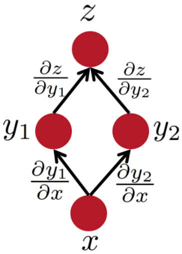
Derivative Chain Rule Recap

- Multiple path chain rule: let $z = f(y_1, y_2)$, $y_1 = g_1(x)$, $y_2 = g_2(x)$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1} \times \frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2} \times \frac{\partial y_2}{\partial x}$$

- Example: $z = e^{y_1 \times y_2}$, $y_1 = 2x$, $y_2 = x^2$

$$\frac{\partial z}{\partial x} = (e^{y_1 \times y_2} \times y_2) \times 2 + (e^{y_1 \times y_2} \times y_1) \times 2x = e^{2x^3} \times 6x^2$$

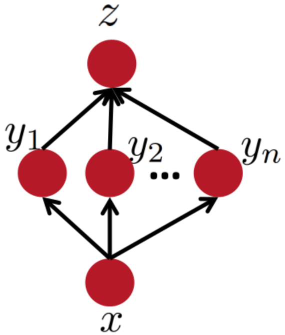


[1] Figure taken from:
`http://cs224d.stanford.edu/lectures/CS224d-Lecture5.pdf`

Derivative Chain Rule Recap

The general version of the multiple path chain rule would be:

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \times \frac{\partial y_i}{\partial x}$$



[1] Figure taken from:
`http://cs224d.stanford.edu/lectures/CS224d-Lecture5.pdf`

Backpropagation

- In a general feed-forward network, each unit computes a weighted sum of its inputs in the form:

$$\vec{h}^{l}_{[j]} = \left(\sum_{i} W^{l}_{[i,j]} \times \vec{z}^{(l-1)}_{[i]} \right) + \vec{b}^{l}_{[j]} \tag{3}$$

- The variable $\vec{z}^{(l-1)}_{[i]}$ is an input that sends a connection to unit $\vec{h}^{l}_{[j]}$, $W^{l}_{[i,j]}$ is the weight associated with that connection, and l is the layer index.
- The biases vectors $\vec{b}_{[j]}$ can be excluded from (eq.3) and included to the weight matrix $W^{l}_{[i,j]}$ by introducing an extra unit, or input, with activation fixed at +1.

Backpropagation

- The inputs at layer l, $\vec{z}^{(l-1)}_{[i]}$ are the result of applying the activation function g to units from the previous layer:

$$\vec{z}^{l}_{[j]} = g(\vec{h}^{l}_{[j]}) \tag{4}$$

- For the input layer ($l = 0$), $\vec{z}$ corresponds to the input vector $\vec{z} = \vec{x}$

$$\vec{z}^{0}_{[j]} = \vec{x}_{[j]} \tag{5}$$

- For each instance in the training set, we supply the corresponding input vector $\vec{x}$ to the network.
- Next we calculate the activations of all of the hidden and output units in the network by successive application of (eq.3) and (eq.4).
- This process is often called forward propagation because it can be regarded as a forward flow of information through the network.

Backpropagation

- Now consider the evaluation of the derivative of L with respect to a weight $W^{l}_{[i,j]}$.
- Assuming that the loss L is calculated over a single example, we can note that L depends on the weight $W^{l}_{[i,j]}$ only via the summed input $\vec{h}^{l}_{[j]}$.
- We can therefore apply the chain rule for partial derivatives to give

$$\frac{\partial L}{\partial W^{l}_{[i,j]}} = \frac{\partial L}{\partial \vec{h}^{l}_{[j]}} \times \frac{\partial \vec{h}^{l}_{[j]}}{\partial W^{l}_{[i,j]}} \tag{6}$$

Backpropagation

- We now introduce a useful notation:

$$\vec{\delta}^{l}_{[j]} \equiv \frac{\partial L}{\partial \vec{h}^{l}_{[j]}} \tag{7}$$

- Using (3), we can write

$$\frac{\partial \vec{h}^{l}_{[j]}}{\partial W^{l}_{[i,j]}} = \vec{z}^{(l-1)}_{[i]} \tag{8}$$

- Substituting (7) and (8) into (6), we then obtain

$$\frac{\partial L}{\partial W^{l}_{[i,j]}} = \vec{\delta}^{l}_{[j]} \times \vec{z}^{(l-1)}_{[i]} \tag{9}$$

Backpropagation

- Equation (9) tells us that the required derivative is obtained simply by multiplying the value of $\vec{\delta}^{l}_{[j]}$ by the value of $\vec{z}^{(l-1)}_{[i]}$.
- Thus, in order to evaluate the derivatives, we need only to calculate the value of $\vec{\delta}^{l}_{[j]}$ for each hidden and output unit in the network, and then apply (9).
- Calculating $\vec{\delta}^{m}_{[j]}$ for output units ($l = m$), is usually straightforward, since activation units $\vec{h}^{m}_{[j]}$ are directly observed in the loss expression.
- The same applies for shallow linear models.

Backpropagation

- To evaluate the $\vec{\delta}^{l}_{[j]}$ for hidden units, we again make use of the chain rule for partial derivatives:

$$\vec{\delta}^{l}_{[j]} \equiv \frac{\partial L}{\partial \vec{h}^{l}_{[j]}} = \sum_{k} \left(\frac{\partial L}{\partial \vec{h}^{l+1}_{[k]}} \times \frac{\partial \vec{h}^{l+1}_{[k]}}{\partial \vec{h}^{l}_{[j]}} \right) \tag{10}$$

- The sum runs over all units $\vec{h}^{l+1}_{[k]}$ to which unit $\vec{h}^{l}_{[j]}$ sends connections.
- We assume that connections go only to consecutive layers in the network (from layer l to layer $(l+1)$).
- The units $\vec{h}^{l+1}_{[k]}$ could include other hidden units and/or output units.
- If we now substitute the definition of $\vec{\delta}^{l}_{[j]}$ given by (eq.7) into (eq.10), we get

$$\vec{\delta}^{l}_{[j]} \equiv \frac{\partial L}{\partial \vec{h}^{l}_{[j]}} = \sum_{k} \left(\vec{\delta}^{(l+1)}_{[k]} \times \frac{\partial \vec{h}^{l+1}_{[k]}}{\partial \vec{h}^{l}_{[j]}} \right) \tag{11}$$

Backpropagation

- Now, for expression $\vec{h}^{l+1}_{[k]}$ we can go to its definition (eq.3):

$$\vec{h}^{(l+1)}_{[k]} = \left(\sum_i W^{l+1}_{[i,k]} \times \vec{z}^{l}_{[i]} \right) + \vec{b}^{(l+1)}_{[k]}$$

- Now, we replace (eq.4) ($\vec{z}^{l}_{[i]} = g(\vec{h}^{l}_{[i]})$) into previous equation and we obtain:

$$\vec{h}^{(l+1)}_{[k]} = \left(\sum_i W^{l+1}_{[i,k]} \times g(\vec{h}^{l}_{[i]}) \right) + \vec{b}^{(l+1)}_{[k]}$$

- Now when calculating $\frac{\partial \vec{h}^{l+1}_{[k]}}{\partial \vec{h}^{l}_{[j]}}$ all the terms in the summation where $i \neq j$ get canceled out.
- Hence:

$$\frac{\partial \vec{h}^{l+1}_{[k]}}{\partial \vec{h}^{l}_{[j]}} = W^{l+1}_{[j,k]} \times g'(\vec{h}^{l}_{[j]}) \tag{12}$$

Backpropagation

- Now, if we substitute (eq.12) into (eq.11)

$$\vec{\delta}^{l}_{[j]} \equiv \frac{\partial L}{\partial \vec{h}^{l}_{[j]}} = \sum_{k} \left(\vec{\delta}^{(l+1)}_{[k]} \times W^{l+1}_{[j,k]} \times g'(\vec{h}^{l}_{[j]}) \right) \tag{13}$$

- Since $g'(\vec{h}^{l}_{[j]})$ doesn't depend on k we can obtain the following backpropagation formula:

$$\vec{\delta}^{l}_{[j]} = g'(\vec{h}^{l}_{[j]}) \times \sum_{k} \left(\vec{\delta}^{(l+1)}_{[k]} \times W^{l+1}_{[j,k]} \right) \tag{14}$$

- Which tells us that the value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network. [Bishop, 2006].

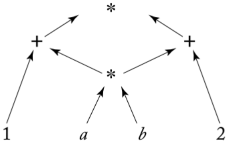
Backpropagation

The backpropagation procedure can be summarized as follows.

1. Apply an input vector $\vec{x}$ to the network and forward propagate through the network using (eq.3) and (eq.4) to find the activations of all the hidden and output units.
2. Evaluate the $\vec{\delta}^{m}_{[j]}$ for all the output units (recall that the derivatives involved here are easy to calculate).
3. Backpropagate the $\vec{\delta}^{(l+1)}_{[k]}$ using (eq.14) to obtain $\vec{\delta}^{l}_{[j]}$ for each hidden unit in the network. We go from higher to lower layers in the network.
4. Use (eq.9) ($\frac{\partial L}{\partial W^{l}_{[i,j]}} = \vec{\delta}^{l}_{[j]} \times \vec{z}^{(l-1)}_{[i]}$) to evaluate the required derivatives.

The Computation Graph Abstraction

- One can compute the gradients of the various parameters of a network by hand and implement them in code.
- This procedure is cumbersome and error prone.
- For most purposes, it is preferable to use automatic tools for gradient computation [Bengio, 2012].
- A computation graph is a representation of an arbitrary mathematical computation (e.g., a neural network) as a graph.
- Consider for example a graph for the computation of $(a * b + 1) * (a * b + 2)$:



- The computation of $a * b$ is shared.
- The graph structure defines the order of the computation in terms of the dependencies between the different components.

The Computation Graph Abstraction

- Te computation graph abstraction allows us to:
  1. Easily construct arbitrary networks.
  2. Evaluate their predictions for given inputs (forward pass)

Algorithm 5.3 Computation graph forward pass.

for i = 1 to N **do**
  Let $a_1, \ldots, a_m = \pi^{-1}(i)$
  $v(i) \leftarrow f_i(v(a_1), \ldots, v(a_m))$

  3. Compute gradients for their parameters with respect to arbitrary scalar losses (backward pass or backpropagation).

Algorithm 5.4 Computation graph backward pass (backpropagation).

$d(N) \leftarrow 1$ $\qquad \triangleright \frac{\partial N}{\partial N} = 1$
for i = N-1 to 1 **do**
  $d(i) \leftarrow \sum_{j \in \pi(i)} d(j) \cdot \frac{\partial f_j}{\partial i}$ $\qquad \triangleright \frac{\partial N}{\partial i} = \sum_{j \in \pi(i)} \frac{\partial N}{\partial j} \frac{\partial j}{\partial i}$

- The backpropagation algorithm (backward pass) is essentially following the chain-rule of differentiation[2].

[2] A comprehensive tutorial on the backpropagation algorithm over the computational graph abstraction: https://colah.github.io/posts/2015-08-Backprop/

Deep Learning Frameworks

Several software packages implement the computation-graph model. All these packages support all the essential components (node types) for defining a wide range of neural network architectures.

- TensorFlow (`https://www.tensorflow.org/`): an open source software library for numerical computation using data-flow graphs originally developed by the Google Brain Team.
- Keras: High-level neural network API that runs on top of Tensorflow as well as other backends (`https://keras.io/`).
- PyTorch: open source machine learning library for Python, based on Torch, developed by Facebook's artificial-intelligence research group. It supports dynamic graph construction, a different computation graph is created from scratch for each training sample. (`https://pytorch.org/`)

Questions?

Thanks for your Attention!

References I

Bishop, C. M. (2006).
Pattern recognition and machine learning.
springer.

Cybenko, G. (1989).
Approximation by superpositions of a sigmoidal function.
Mathematics of control, signals and systems, 2(4):303–314.

Glorot, X., Bordes, A., and Bengio, Y. (2011).
Deep sparse rectifier neural networks.
In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323.

Goldberg, Y. (2017).
Neural network methods for natural language processing.
Synthesis Lectures on Human Language Technologies, 10(1):1–309.

Goodfellow, I., Bengio, Y., and Courville, A. (2016).
Deep learning.
MIT press.

Hornik, K., Stinchcombe, M., and White, H. (1989).
Multilayer feedforward networks are universal approximators.
Neural networks, 2(5):359–366.