# Natural Language Processing
# Neural Networks

Felipe Bravo-Marquez

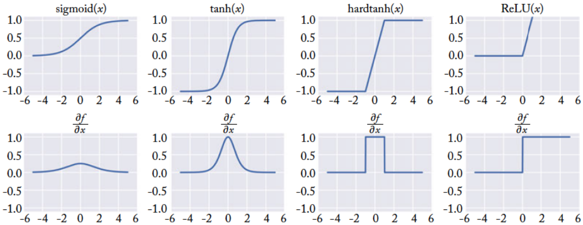
July 25, 2019

# 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.

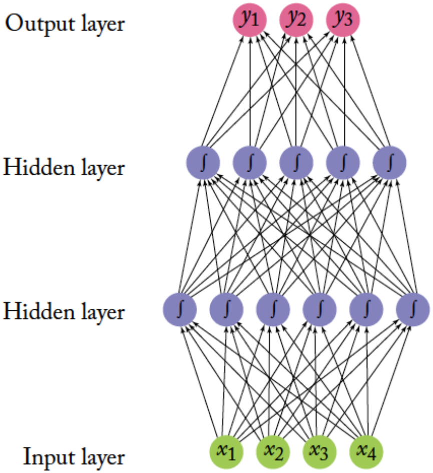
# Activation Functions

- 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.



[0] Source:[Goldberg, 2017]

# Feedforward Network with two Layers



[0] Source:[Goldberg, 2017]

# Brief Introduction to 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.

# Brief Introduction to Neural Networks

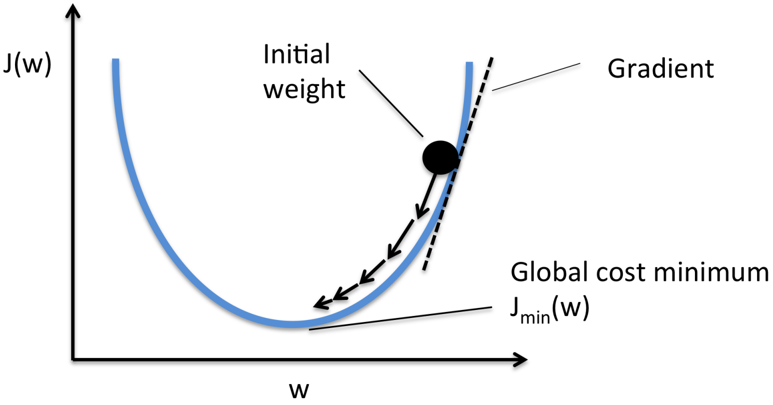
- The Multilayer Perceptron (MLP) from the figure can be written as the following mathematical function:

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

# Training

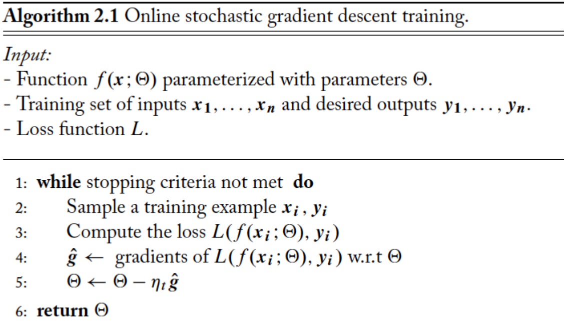
- When training a parameterized function (e.g., a linear model, a neural network) one defines a loss function $L(\hat{y}, y)$, stating the loss of predicting $\hat{y}$ when the true output is y.
- The training objective is then to minimize the loss across the different training examples.
- Functions are trained using gradient-based methods.
- They work by repeatedly computing an estimate of the loss $L$ over the training set.
- They compute gradients of the parameters with respect to the loss estimate, and moving the parameters in the opposite directions of the gradient.
- Different optimization methods differ in how the error estimate is computed, and how moving in the opposite direction of the gradient is defined.

# Gradient Descent



[0] Source: `https://sebastianraschka.com/images/faq/closed-form-vs-gd/ball.png`

# Online Stochastic Gradient Descent



**Algorithm 2.1** Online stochastic gradient descent training.

*Input:*

- Function $f(\boldsymbol{x};\Theta)$ parameterized with parameters $\Theta$.
- Training set of inputs $\boldsymbol{x_1},\dots,\boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1},\dots,\boldsymbol{y_n}$.
- Loss function $L$.

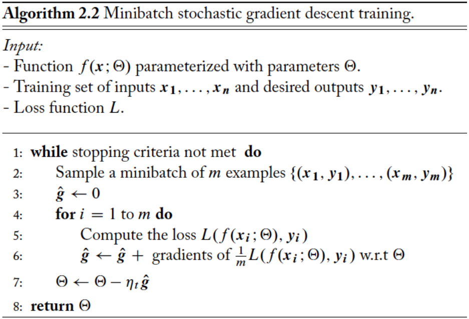
```
1: while stopping criteria not met do
2:     Sample a training example $\boldsymbol{x_i}, \boldsymbol{y_i}$
3:     Compute the loss $L(f(\boldsymbol{x_i};\Theta), \boldsymbol{y_i})$
4:     $\hat{\boldsymbol{g}} \leftarrow$ gradients of $L(f(\boldsymbol{x_i};\Theta), \boldsymbol{y_i})$ w.r.t $\Theta$
5:     $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
6: return $\Theta$
```

- The learning rate can either be fixed throughout the training process, or decay as a function of the time step *t*.
- The error calculated in line 3 is based on a single training example, and is thus just a rough estimate of the corpus-wide loss *L* that we are aiming to minimize.
- The noise in the loss computation may result in inaccurate gradients (single examples may provide noisy information).

[0]Source:[Goldberg, 2017]

# Mini-batch Stochastic Gradient Descent

- A common way of reducing this noise is to estimate the error and the gradients based on a sample of *m* examples.
- This gives rise to the minibatch SGD algorithm



**Algorithm 2.2** Minibatch stochastic gradient descent training.

```
Input:
- Function $f(x;\Theta)$ parameterized with parameters $\Theta$.
- Training set of inputs $x_1, \dots, x_n$ and desired outputs $y_1, \dots, y_n$.
- Loss function $L$.

1: while stopping criteria not met do
2:     Sample a minibatch of $m$ examples $\{(x_1, y_1), \dots, (x_m, y_m)\}$
3:     $\hat{g} \leftarrow 0$
4:     for $i = 1$ to $m$ do
5:         Compute the loss $L(f(x_i;\Theta), y_i)$
6:         $\hat{g} \leftarrow \hat{g} +$ gradients of $\frac{1}{m}L(f(x_i;\Theta), y_i)$ w.r.t $\Theta$
7:     $\Theta \leftarrow \Theta - \eta_t \hat{g}$
8: return $\Theta$
```

- Higher values of *m* provide better estimates of the corpus-wide gradients, while smaller values allow more updates and in turn faster convergence.
- For modest sizes of *m* , some computing architectures (i.e., GPUs) allow an efficient parallel implementation of the computation in lines 3-6.

[0] Source:[Goldberg, 2017]

# Some Loss Functions

- Hinge (or SVM loss): for binary classification problems, the classifier's output is a single scalar $\tilde{y}$ and the intended output y is in $\{+1, -1\}$. The classification rule is $\hat{y} = sign(\tilde{y})$, and a classification is considered correct if $y \cdot \tilde{y} > 0$.

$$L_{\text{hinge(binary)}}(\tilde{y}, y) = \max(0, 1 - y \cdot \tilde{y})$$

- Binary cross entropy (or logistic loss): is used in binary classification with conditional probability outputs. The classifier's output $\tilde{y}$ is transformed using the sigmoid function to the range $[0, 1]$ , and is interpreted as the conditional probability $P(y = 1|x)$.

$$L_{\text{logistic}}(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

# Some Loss Functions

- Categorical cross-entropy loss: is used when a probabilistic interpretation of multi-class scores is desired. It measures the dissimilarity between the true label distribution $y$ and the predicted label distribution $\tilde{y}$.

$$L_{\text{cross-entropy}}(\hat{y}, y) = -\sum_{i} y_{[i]} \log(\hat{y}_{[i]})$$

- The predicted label distribution of the categorical cross-entropy loss ($\hat{y}$) is obtained by applying the softmax function the last layer of the network $\tilde{y}$:

$$\hat{y}_{[i]} = \text{softmax}(\tilde{y})_{[i]} = \frac{e^{\tilde{y}_{[i]}}}{\sum_j e^{\tilde{y}_{[j]}}}$$

- The softmax function squashes the $k$-dimensional output to values in the range (0,1) with all entries adding up to 1. Hence, $\hat{y}_{[i]} = P(y = i|x)$ represent the class membership conditional distribution.

# Backpropagation

- 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: $W^1, \vec{b}^1, \dots, W^m, \vec{b}^m$, 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.
- In a general feed-forward network, each unit computes a weighted sum of its inputs in the form:

$$\vec{h}^l_{[j]} = \left( \sum_i W^l_{[i,j]} \times \vec{z}^{(l-1)}_{[i]} \right) + \vec{b}^l_{[j]} \tag{2}$$

- 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.2) and included to the weight matrix $W^l_{[i,j]}$ by introducing an extra unit, or input, with activation fixed at +1.

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

# Backpropagation

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

$$\vec{z}_{[j]}^{l} = g(\vec{h}_{[j]}^{l}) \tag{3}$$

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

$$\vec{z}_{[j]}^{0} = \vec{x}_{[j]} \tag{4}$$

- 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.2) and (eq.3).
- 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

$$\frac{\partial L}{\partial W^{l}_{[i,j]}} = \frac{\partial L}{\partial \vec{h}^{l}_{[j]}} \times \frac{\partial \vec{h}^{l}_{[j]}}{\partial W^{l}_{[i,j]}} \tag{5}$$

# Backpropagation

- We now introduce a useful notation:

$$\vec{\delta}^{l}_{[j]} \equiv \frac{\partial L}{\partial \vec{h}^{l}_{[j]}} \tag{6}$$

- Using (2), we can write

$$\frac{\partial \vec{h}^{l}_{[j]}}{\partial W^{l}_{[i,j]}} = \vec{z}^{(l-1)}_{[i]} \tag{7}$$

- Substituting (6) and (7) into (5), we then obtain

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

# Backpropagation

- Equation (8) 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 (8).
- 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:

$$\vec{\delta}^{l}_{[j]} \equiv \frac{\partial L}{\partial \vec{h}^{l}_{[j]}} = \sum_{k} \left( \frac{\partial L}{\partial \vec{h}^{l+1}_{[k]}} \times \frac{\partial \vec{h}^{l+1}_{[k]}}{\partial \vec{h}^{l}_{[j]}} \right) \tag{9}$$

- 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.6) into (eq.9), we get

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

# Backpropagation

- Now, for expression $\vec{h}_{[k]}^{l+1}$ we can go to its definition (eq.2):

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

- Now, we replace (eq.3) ($\vec{z}_{[i]}^{l} = g(\vec{h}_{[i]}^{l})$) into previous equation and we obtain:

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

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

$$\frac{\partial \vec{h}_{[k]}^{l+1}}{\partial \vec{h}_{[j]}^{l}} = W_{[j,k]}^{l+1} \times g'(\vec{h}_{[j]}^{l}) \qquad (11)$$

# Backpropagation

- Now, if we substitute (eq.11) into (eq.10)

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

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

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

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

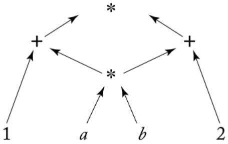
# Backpropagation

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.2) and (eq.3) 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.13) 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.8) ($\frac{\partial L}{\partial W^{l}_{[i,j]}} = \vec{\delta}^{l}_{[j]} \times \vec{z}^{(l-1)}_{[i]}$) to evaluate the required derivatives.

# The Computation Graph Abstraction

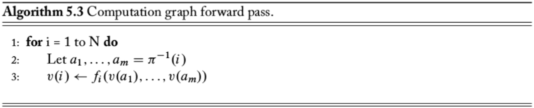
- One can compute the gradients of the various parameters of a network by hand and implement them in code.
- This procedure is cumbersome and error prone.
- For most purposes, it is preferable to use automatic tools for gradient computation [Bengio, 2012].
- A computation graph is a representation of 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

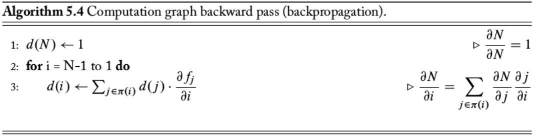
- Te computation graph abstraction allows us to:
  1. Easily construct arbitrary networks.
  2. Evaluate their predictions for given inputs (forward pass)



**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).

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

- 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/`

# Train, Test, and Validation Sets

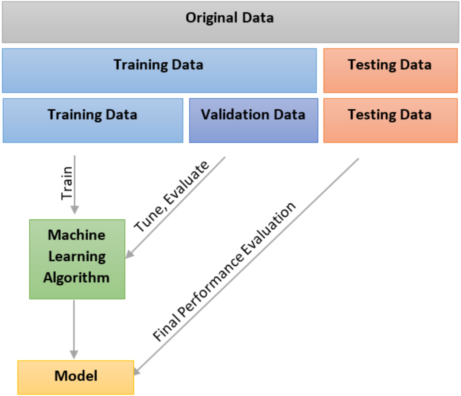
- Neural networks are prone to overfit the data.
- Hence, performance on training data can be misleading.
- Held-out set: split training set into training and testing subsets (80% and 20% splits). Train on training and compute accuracy on testing.
- Problem: in practice you often train several models, compare their quality, and select the best one.
- Selecting the best model according to the held-out set's accuracy will result in an overly optimistic estimate of the model's quality.
- You don't know if the chosen settings of the final classifier are good in general, or are just good for the particular examples in the held-out sets.

# Train, Test, and Validation Sets

- The accepted methodology is to use a three-way split of the data into train, validation (also called development ), and test sets [3].
- This gives you two held-out sets: a validation set (also called development set ), and a test set.
- All the experiments, tweaks, error analysis, and model selection should be performed based on the validation set.
- Then, a single run of the final model over the test set will give a good estimate of its expected quality on unseen examples.
- It is important to keep the test set as pristine as possible, running as few experiments as possible on it.
- Some even advocate that you should not even look at the examples in the test set, so as to not bias the way you design your model.

[3] An alternative approach is cross-validation, but it doesn't scale well for training deep neural networks.

# Train, Test, and Validation Sets



[3] source: `https://www.codeproject.com/KB/AI/1146582/validation.PNG`

# 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.

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