

# Natural Language Processing Neural Networks

Felipe Bravo-Marquez

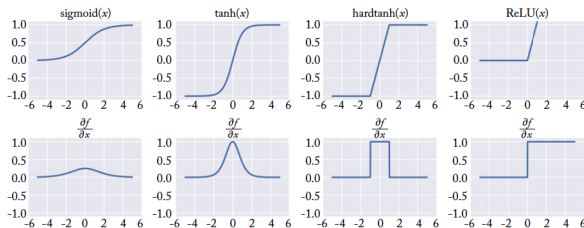
July 23, 2019

# Introduction to Neural Networks

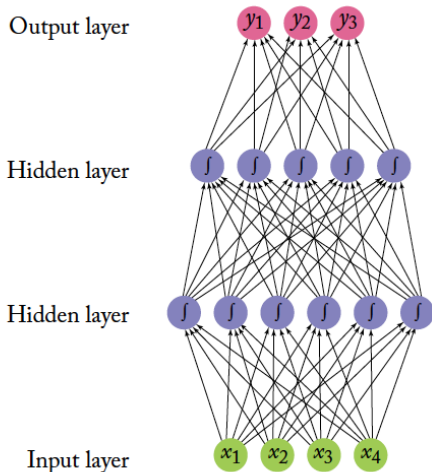
- Very popular machine learning models formed by units called **neurons**.
- A neuron is a computational unit that has scalar inputs and outputs.
- Each input has an associated weight  $w$ .
- The neuron multiplies each input by its weight, and then sums them (other functions such as **max** are also 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



# 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.

---

<sup>0</sup>Vectors are assumed to be row vectors and superscript indices correspond to network layers.

# Brief Introduction to Neural Networks

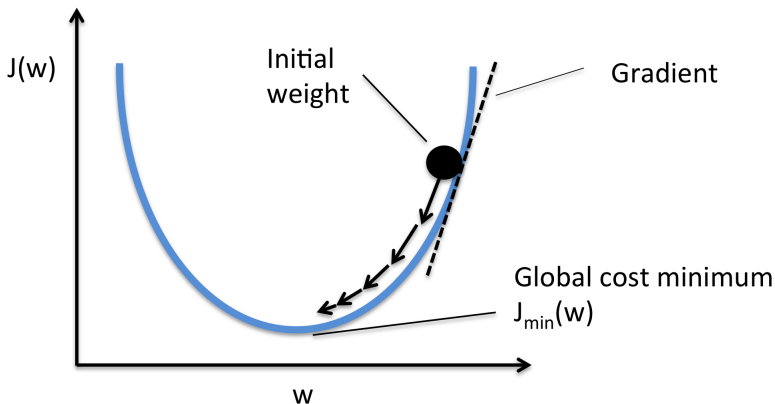
- The Multilayer Perceptron (MLP) from the figure can be written as the following mathematical function:

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

# 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



<sup>0</sup>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  $f(\mathbf{x}; \Theta)$  parameterized with parameters  $\Theta$ .
- Training set of inputs  $\mathbf{x}_1, \dots, \mathbf{x}_n$  and desired outputs  $y_1, \dots, y_n$ .
- Loss function  $L$ .

---

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

# 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:      $\hat{g} \leftarrow \hat{g} + \text{gradients of } \frac{1}{m}L(f(x_i; \Theta), y_i) \text{ w.r.t } \Theta$ 
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.

---

<sup>0</sup>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  $\hat{y} = \text{sign}(\tilde{y})$ , and a classification is considered correct if  $y \cdot \tilde{y} > 0$ .

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

- 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  $\hat{y}$ .

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

- 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 (without vectorial derivatives)

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

$$\vec{h}_{[l]}^l = \sum_i \mathbf{w}_{[i,l]}^l \times \vec{z}_{[l]}^{(l-1)} + \vec{b}_{[l]}^l \quad (2)$$

$$\frac{\partial L}{\partial \mathbf{w}_{[i,l]}^l} = \frac{\partial L}{\partial \vec{h}_{[l]}^l} \times \frac{\partial \vec{h}_{[l]}^l}{\partial \mathbf{w}_{[i,l]}^l} \quad (3)$$

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

$$\vec{z}_{[l]}^0 = \vec{x}_{[l]} \quad (5)$$

$$\vec{\delta}_{[l]}^l = \frac{\partial L}{\partial \vec{h}_{[l]}^l} \quad (6)$$

$$\frac{\partial \vec{h}_{[l]}^l}{\partial \mathbf{w}_{[i,l]}^l} = \vec{z}_{[l]}^{(l-1)} \quad (7)$$

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

$$\vec{\delta}_{[l]}^l = \frac{\partial L}{\partial \vec{h}_{[l]}^l} = \sum_{k, m \geq 0} \frac{\partial L}{\partial \vec{h}_{[l]}^m} \times \frac{\partial \vec{h}_{[k]}^m}{\partial \vec{h}_{[l]}^l} \quad (9)$$

# Backpropagation (without vectorial derivatives)

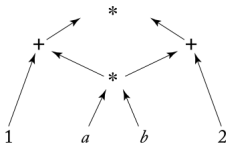
- In a general feed-forward network, each unit computes a weighted sum of its inputs in the form:

$$\vec{h}_{[j]}^l = \sum_i w_{[i,j]}^l \times \vec{z}_{[i]}^{(l-1)} + \vec{b}_{[j]}^l \quad (10)$$

- where  $z_i$  is the activation of a unit, or input, that sends a connection to unit  $j$ , and  $w_{ji}$  is the weight associated with that connection. superscripts correspond to layer indexes

# 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

- The 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.

---

```
1: for i = 1 to N do
2:   Let  $a_1, \dots, a_m = \pi^{-1}(i)$ 
3:    $v(i) \leftarrow f_i(v(a_1), \dots, v(a_m))$ 
```

---

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

---

**Algorithm 5.4** Computation graph backward pass (backpropagation).

---

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

---

- The backpropagation algorithm (backward pass) is essentially following the chain-rule of differentiation<sup>1</sup>.

---

<sup>1</sup>A comprehensive tutorial on the backpropagation algorithm over the computational graph abstraction:



# Train, Test, and Validation Sets

- Parameterized function 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.

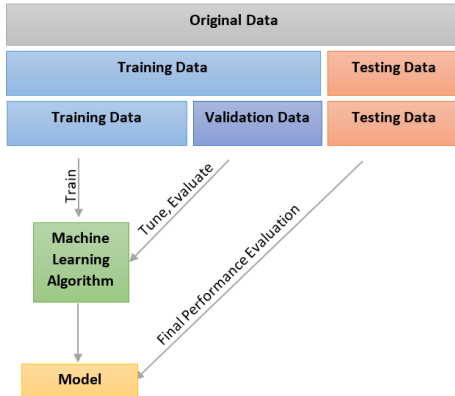
# Train, Test, and Validation Sets

- The accepted methodology is to use a three-way split of the data into train, validation (also called development ), and test sets <sup>2</sup>.
- 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.

---

<sup>2</sup>An alternative approach is cross-validation, but it doesn't scale well for training deep neural networks.

# Train, Test, and Validation Sets



<sup>2</sup>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



Goldberg, Y. (2017).

Neural network methods for natural language processing.

*Synthesis Lectures on Human Language Technologies*, 10(1):1–309.