An Introduction to Deep Learning

Clint P. George

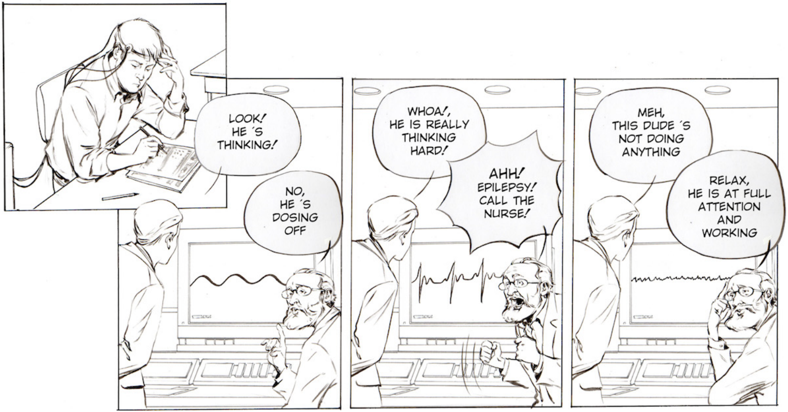
University of Florida Informatics Institute

March 14, 2017

Outline

1. Introduction to feed-forward networks
2. Relation to logistic regression
3. Notes on implementation
4. Illustration using synthetic and real data

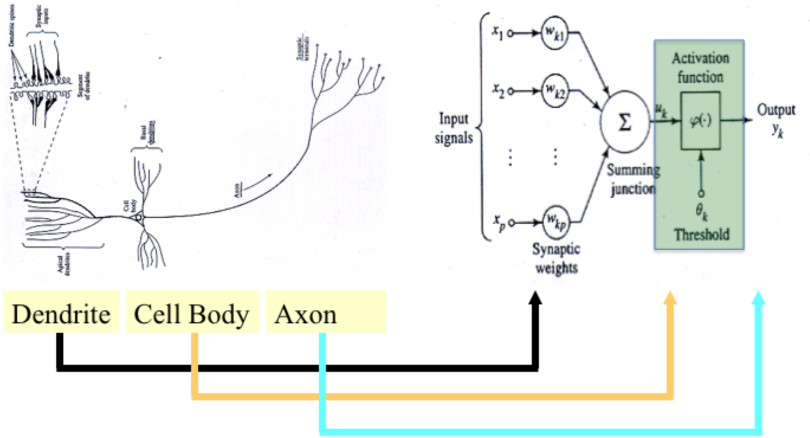
Motivation



The idea: Human intelligence may be due to a learning algorithm.
We aim to build algorithms that mimic the brain.[1]

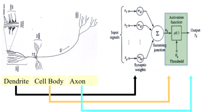
[1] image: https://backyardbrains.com/experiments/EEG

Imitate neurons in the brain: Artificial Neurons



Artificial Neuron (AN): input, weights, and output

Imitate neurons in the brain: Artificial Neurons



Artificial Neuron (AN): input, weights, and output

The inputs x_j's roughly correspond to the signals received by the dendrites of a biological neuron, and the output y_k models what the neuron soma (cell body) sends down the axon to its neighbors (i.e., the neurons output).

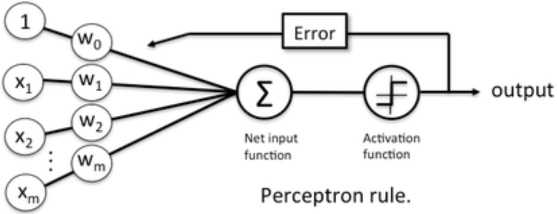
We calculate the pre-activation $\mu_k = \theta_k + \sum_{j=1}^{p} w_{kj} x_j$.

We calculate the output—the equivalent of what the biological neuron would send on its axon, by applying the activation function, $\psi(.)$, to μ_k.

Learning in an ANN is in large part about setting weights and biases

Activation functions

The Perceptron (Rosenblatt et al. 1957 & 1962) computes a step function as an activation function.



Perceptron rule.

$$\text{step}(z) = \begin{cases} 1 & z \geq t \\ 0 & z < t \end{cases}, \text{ where } t \text{ is a threshold}$$

As each input is applied to the perceptron its output is compared to the target. To keep the output closer to the target the **learning rule** adjusts the network parameters.

What can a single AN compute?

The Perceptron output is given by $y = \text{step}(b + \sum_{j=1}^{p} w_j x_j)$.

Perceptron can divide the input space into two regions.

The decision boundary is given by: $b + \sum_{j=1}^{p} w_j x_j = 0$.

Perceptron can learn to classify any linearly separable set of inputs—convergence theorem (Rosenblatt 1962)

What can a single AN compute?

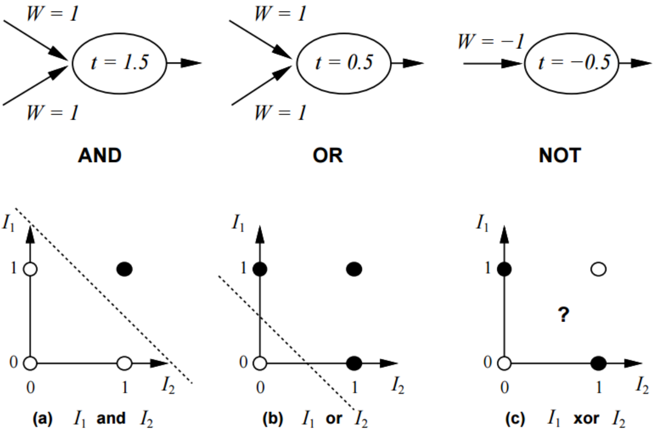
The Perceptron output is given by $y = \text{step}(b + \sum_{j=1}^{p} w_j x_j)$.

Perceptron can divide the input space into two regions.
The decision boundary is given by: $b + \sum_{j=1}^{p} w_j x_j = 0$.

Perceptron can learn to classify any linearly separable set of inputs—convergence theorem (Rosenblatt 1962)

The decision boundary is always orthogonal to the weight vector. This defines a line in the input space. On one side of the line the network output will be 0; on the line and on the other side of the line the output will be 1.

Boolean functions



Examples[2] of linearly separable and non separable problems

[2] Veloso, 2001

Learning the XOR function

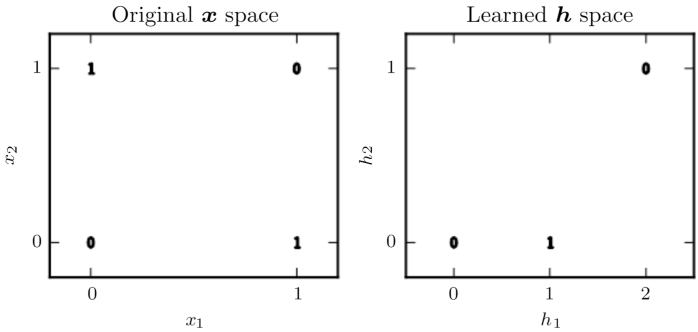
Consider this as a regression problem and use the MSE loss function:

$$\mathsf{MSE}(\theta) = \frac{1}{4} \sum_{\boldsymbol{x} \in \mathcal{X}} (y - f_\theta(\boldsymbol{x}))^2$$

where $\theta = (\boldsymbol{w}, b)$ and $f_\theta(\boldsymbol{x}) = b + \sum_{j=1}^{p} w_j x_j$—a linear model.

Using the normal equations we can minimize $\mathsf{MSE}(\theta)$ w.r.t. $\boldsymbol{w}$ and b in closed form. It gives $\boldsymbol{w} = 0$ and $b = .5$—This gives the model output $.5$ everywhere.

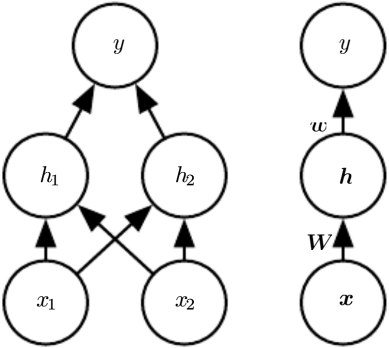
Learning the XOR function



- When $x_1 = 0$, the model's output must increase as x_2 increases
- When $x_1 = 1$, the model's output must decrease as x_2 increases
- A linear model applies a fixed coefficient w_2 to x_2. It cannot use the value of x_1 to change the coefficient w_2 on x_2 and cannot solve this problem.

Intuition behind multilayer neural network

One way to solve the XOR problem is to transform the input by introducing a feed-forward network:



The complete model will then be, in a function form:

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{w}, b, \boldsymbol{c}) = f^{(2)}\left(f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}); \boldsymbol{w}, b\right)$$

What function should $f^{(1)}$ be?

We consider $f^{(2)}$ as a linear function.

We can write the hidden layer output as

$$\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}) = g(\boldsymbol{W}^{\mathsf{T}} \boldsymbol{x} + \boldsymbol{c})$$

What function should $f^{(1)}$ be?

We consider $f^{(2)}$ as a linear function.

We can write the hidden layer output as

$$\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}) = g(\boldsymbol{W}^\mathsf{T}\boldsymbol{x} + \boldsymbol{c})$$

If $f^{(1)}$ is also linear, then the network as a whole would remain a linear function of its input.

What function should $f^{(1)}$ be?

We consider $f^{(2)}$ as a linear function.

We can write the hidden layer output as

$$\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}) = g(\boldsymbol{W}^{\mathsf{T}}\boldsymbol{x} + \boldsymbol{c})$$

So we must use a nonlinear activation function for g. A popular nonlinear activation function g is rectified linear unit (ReLU)

$$g(z) = \mathsf{max}\{0, z\}$$

A feed-forward network solution to XOR

We wish the network to perform well on the four cases

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}, \boldsymbol{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

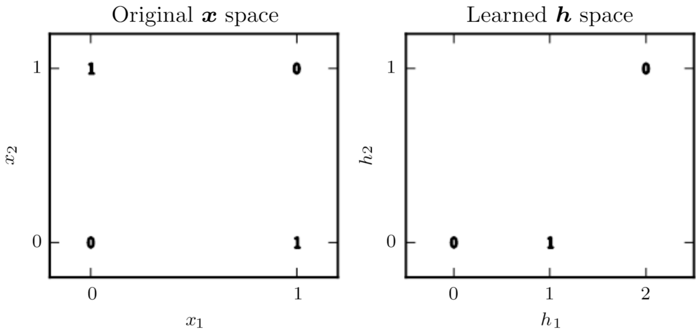
We can specify a two-layer network solution to XOR as

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0$$

The complete network is given by

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{w}, b, \boldsymbol{c}) = \boldsymbol{w}^{\mathsf{T}} \max\{0, \boldsymbol{W}^{\mathsf{T}} \boldsymbol{x} + \boldsymbol{c}\} + b$$

A feed-forward network solution to XOR



In the proposed network, the nonlinear hidden layer has mapped both $\boldsymbol{x} = [1, 0]$ and $\boldsymbol{x} = [0, 1]$ to a single point in feature space, $\boldsymbol{h} = [1, 0]$.

A linear model can now describe the function as increasing in h_1 and decreasing in h_2.

Activation functions

Modern ANs use a variety of activation functions that are smoother than the step function.

Linear - no input squashing

$$y = x$$

Rectified linear unit

$$y = \mathsf{max}\{0, x\}$$

Logistic sigmoid - squash input into $[0, 1]$

$$y = \mathsf{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

Hyperbolic tangent - squash input into $[-1, 1]$

$$y = \mathsf{tan}(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

Relation to logistic regression

Let $p(y = 1 \,|\, X = \boldsymbol{x}) = p(\boldsymbol{x}; \boldsymbol{w})$ be the conditional probability that a particular sample belongs to class 1 given its predictors $\boldsymbol{x}$. We write the logistic regression model as

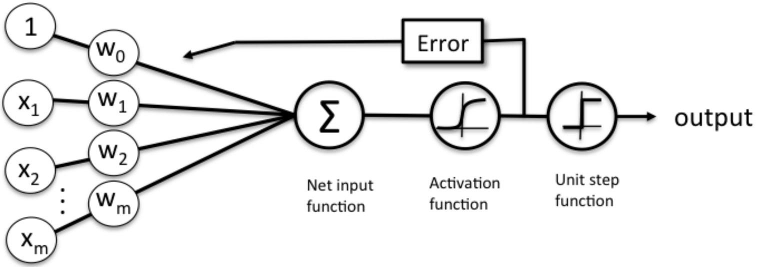
$$\text{logit}\left(p(\boldsymbol{x}; \boldsymbol{w})\right) = w_0 + \boldsymbol{w}^\mathsf{T}\boldsymbol{x}$$

Solving for $p(\boldsymbol{x}; \boldsymbol{w})$ gives

$$p(\boldsymbol{x}; \boldsymbol{w}) = \text{sigmoid}\left(w_0 + \boldsymbol{w}^\mathsf{T}\boldsymbol{x}\right)$$

To minimize misclassification rate, one should predict $y = 1$ when $p \geq 1$, and vice versa.—i.e. guess 1 whenever $w_0 + \boldsymbol{w}^\mathsf{T}\boldsymbol{x} \geq 0$ and 0 otherwise. So logistic regression gives a linear classifier.

Relation to logistic regression

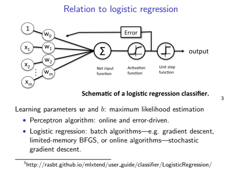


Schematic of a logistic regression classifier. [3]

Learning parameters $\boldsymbol{w}$ and b: maximum likelihood estimation

- Perceptron algorithm: online and error-driven.
- Logistic regression: batch algorithms—e.g. gradient descent, limited-memory BFGS, or online algorithms—stochastic gradient descent.

[3] http://rasbt.github.io/mlxtend/user_guide/classifier/LogisticRegression/



By construction an AN is a linear model. An AN with a sigmoid activation function is the same model as logistic regression.

$$y = \text{sigmoid}\left(b_k + \sum_{j=1}^{p} w_{kj} x_j\right)$$

Overfitting is high for ANN when we train using only a limited size data

In LR, the model complexity is less and overfitting is not a big issue.

Both AN and LR has a functional form for the prob. $p(y = 1 \mid X = \boldsymbol{x})$. LR is a parametric model and its parameters are interpretable; but, the weights in a AN is not necessary mean anything.

Gradient-based learning

The nonlinearity of a neural network causes most interesting loss functions to become non-convex

- Optimization procedure – using iterative, gradient-based optimizers that drive the cost function to a very low value
- Cost function, $C(\theta)$ – they are more or less the same as those for other parametric models, such as linear models

Gradient descent (GD) algorithm[4]

Let $C(\theta) = \int \mathcal{L}(f_\theta, z)P(z)dz$, where in supervised learning $z = (x, y)$ and $f_\theta(x)$ is the predictive function for y given θ.

Gradient descent: find a θ that minimizes the cost

- By solving $\frac{\partial C(\theta)}{\partial \theta} = 0$ we can find the minima, maxima, and saddle points.
- In general we cannot find the solutions of this equation, hence we seek numerical optimization methods
- *local descent*: iteratively modify θ so as to decrease $C(\theta)$, until we reach a local minima

$$\theta^{(t+1)} = \theta^{(t)} - \epsilon \frac{\partial C(\theta^{(t)})}{\partial \theta^{(t)}},$$

ϵ is the learning rate

[4] http://www.iro.umontreal.ca/~pift6266/H10/notes/gradient.html

Stochastic gradient descent (SGD) algorithm

Let $C(\theta) = \int \mathcal{L}(f_\theta, z)P(z)dz$, where in supervised learning $z = (x, y)$ and $f_\theta(x)$ is the predictive function for y given θ.

Stochastic gradient descent: find a θ that minimizes the cost

- Similar to GD, but exploits the fact that $C(\theta)$ is an average, generally over i.i.d. samples
- Make updates much often, the most common case

$$\theta^{(t+1)} = \theta^{(t)} - \epsilon \frac{\partial \mathcal{L}(\theta^{(t)}, z)}{\partial \theta^{(t)}},$$

  z is an the next sample from the training set.—It can be implemented online
- In SGD, the update direction is a random variable whose expectation is the true gradient of interest.

Stochastic gradient descent (SGD) algorithm

Let $C(\theta) = \int \mathcal{L}(f_\theta, z)P(z)dz$, where in supervised learning $z = (x, y)$ and $f_\theta(x)$ is the predictive function for y given θ.

Stochastic gradient descent: find a θ that minimizes the cost

- Similar to GD, but exploits the fact that $C(\theta)$ is an average, generally over i.i.d. samples
- Make updates much often, the most common case

$$\theta^{(t+1)} = \theta^{(t)} - \epsilon \frac{\partial \mathcal{L}(\theta^{(t)}, z)}{\partial \theta^{(t)}},$$

  z is an the next sample from the training set.—It can be implemented online
- In SGD, the update direction is a random variable whose expectation is the true gradient of interest.

The convergence conditions of SGD are similar to those for gradient descent, in spite of the added randomness.

SGD can be much faster than ordinary (also called batch) gradient descent, because it makes updates much more often. This is especially true for large datasets, or in the online setting. In fact, in machine learning tasks, one only uses ordinary gradient descent instead of SGD when the function to minimize cannot be decomposed as above (as a mean).

Learning Conditional Distributions with Maximum Likelihood

Popular choice: the neural network defines a distribution $p(\boldsymbol{y} \mid \boldsymbol{x}, \theta)$ and uses the principle of maximum likelihood

Cost function: the negative log-likelihood—the cross-entropy between the training data and the model distribution

$$C(\theta) = -\mathsf{E}_{\boldsymbol{x},\boldsymbol{y}} \log p_{\mathsf{model}}(\boldsymbol{y}|\boldsymbol{x}, \theta)$$

An example: if $p_{\mathsf{model}}(\boldsymbol{y}|\boldsymbol{x}, \theta) = \mathcal{N}(f_\theta(\boldsymbol{x}), \mathsf{I})$, then we have

$$C(\theta) = \frac{1}{2}\mathsf{E}_{\boldsymbol{x},\boldsymbol{y}}\|\boldsymbol{y} - f_\theta(\boldsymbol{x})\|^2 + \text{const.}$$

Learning Conditional Distributions with Maximum Likelihood

Popular choice: the neural network defines a distribution $p(\boldsymbol{y} \mid \boldsymbol{x}, \theta)$ and uses the principle of maximum likelihood

Cost function: the negative log-likelihood—the cross-entropy between the training data and the model distribution

$$C(\theta) = -\mathsf{E}_{\boldsymbol{x},\boldsymbol{y}} \log p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x}, \theta)$$

An example: if $p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x}, \theta) = \mathcal{N}(f_\theta(\boldsymbol{x}), \mathsf{I})$, then we have

$$C(\theta) = \frac{1}{2}\mathsf{E}_{\boldsymbol{x},\boldsymbol{y}}\|\boldsymbol{y} - f_\theta(\boldsymbol{x})\|^2 + \text{const.}$$

the deep learning book says "the constant is based on the variance of the gaussian, which we chose not to parameterize

We desire to have a large and predictable gradient of $C(\theta)$ to serve as good guide for the learning algorithm. Functions that saturate (become very at) undermine this objective because they make the gradient become very small. In many cases this happens because the activation functions used to produce the output of thehidden units or the output units saturate. The negative log-likelihood helps to avoid this problem for many models. One unusual property of the cross-entropy cost used in MLE is that it usually does not have a minimum value when applied to the models commonly used in practice (e.g. Logistic regression). For discrete output variables, most models are parametrized in such a way that they cannot represent a probability of zero or one, but can come arbitrarily close to doing so.—Regularization is a rescue technique"

Output Units

The output layer provides additional transformation from the hidden features to complete the network's indented task.

Linear units for Gaussian output distributions: the output units based on an affine transformation with no nonlinearity

- Given hidden features $\boldsymbol{h}$, we define outputs $\hat{\boldsymbol{y}} = \boldsymbol{w}^\mathsf{T}\boldsymbol{h} + b$
- Used to produce the mean of $p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\hat{\boldsymbol{y}}, \mathsf{I})$
- MLE is equivalent to minimizing the MSE
- The linear units do not saturate, hence ideal for gradient-based optimization algorithms

Output Units

Sigmoid units for Bernoulli output distributions: e.g. for binary classification

Given hidden features $\boldsymbol{h}$, we define output units

$$\hat{\boldsymbol{y}} = \text{sigmoid}(\boldsymbol{z}) = \text{sigmoid}\left(\boldsymbol{w}^\mathsf{T}\boldsymbol{h} + b\right)$$

The sigmoid can be motivated by constructing an unnormalized probability distribution that doesn't sums to 1. We then normalize to yield a Bernoulli distribution

$$p(y \,|\, \boldsymbol{x}) = \text{sigmoid}\left((2y-1)z\right)$$

The cost function for MLE is defined by the negative log-likelihood

Hidden Units

ReLU - the default choice, which is similar to linear

- $g(z) = \max\{0, z\}$, not differentiable at $z = 0$; but one can safely disregard this problem: "*... neural network training algorithms do not usually arrive at a local minimum of the cost function, but instead merely reduce its value significantly*"
- Drawback: they cannot learn via gradient-based methods on examples for which their activation is zero

Hidden Units

Sigmoid and Hyperbolic Tangent

- Sigmoidal units saturate across most of their domain—they saturate to a high value when z is very positive
- It can make gradient-based learning very difficult. Hyperbolic Tangent typically performs better than the logistic sigmoid and can be used

An overview of back-propagation algorithm

Computing an analytical expression for the gradient is straightforward, but numerical evaluation of such an expression can be expensive.—**backprop** gives an inexpensive procedure to evaluate this gradient.

backprop is a generic algorithm which can be applied to any problem where we need to evaluate $\nabla_\theta C(\theta)$

backprop uses the chain rule of calculus: Suppose $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$, $g : \mathbb{R}^m \to \mathbb{R}^n, f : \mathbb{R}^n \to \mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x}), z = g(\boldsymbol{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} . \frac{\partial y_j}{\partial x_i}.$$

We can apply this to vectors and tensors in the multilayer network

Notes on architectural considerations

In practice, the overall structure of the network is important: how many units it should have and how these units should be connected to each other.

Other examples

- Convolutional Neural Networks — imitates human memory
- Auto Encoders — unsupervised learning and dimensionality reduction

Hands on experiments

Datasets

- A synthetic spiral dataset with multiple classes
- The MNIST data set with 0-9 handwritten characters

Algorithm: A basic back-propagation algorithm implementation with one hidden layer

Link to R scripts: `http://bit.ly/deep-learning-r`

Links to serious implementations

Deep learning frameworks, which uses CPU and GPU

- TensorFlow with R
  `https://rstudio.github.io/tensorflow/index.html`
- Theano with python
  `http://deeplearning.net/software/theano`

Available R packages

- neuralnet
- deepnet
- h2o

Questions?