Applied Deep Learning - Notes

Dom Hutchinson

October 10, 2020

Contents

1 Machine Learning

Definition 1.1 - *Deep Representation Learning*
Representation Learning is a set of techniques in machine learning where a system can automatically learn representations needed for feature detection from the raw data without the need for hand-designed feature descriptions. *Deep Representation Learning* is then learning to classify using this feature detection.

2 Artificial Neural Networks

Remark 2.1 - *Biological Inspiration*
In the natural world *Neurons* are the basic working units of the brain. *Neurons* can be split into three main areas

i). *Dendrites* - Receives inputs from other neurons.

ii). *Axon* - Carries information.

iii). *Axon Terminals & Synapses* = Send information to other neurons.

Artificial Neural Networks seek to mimic this structure.

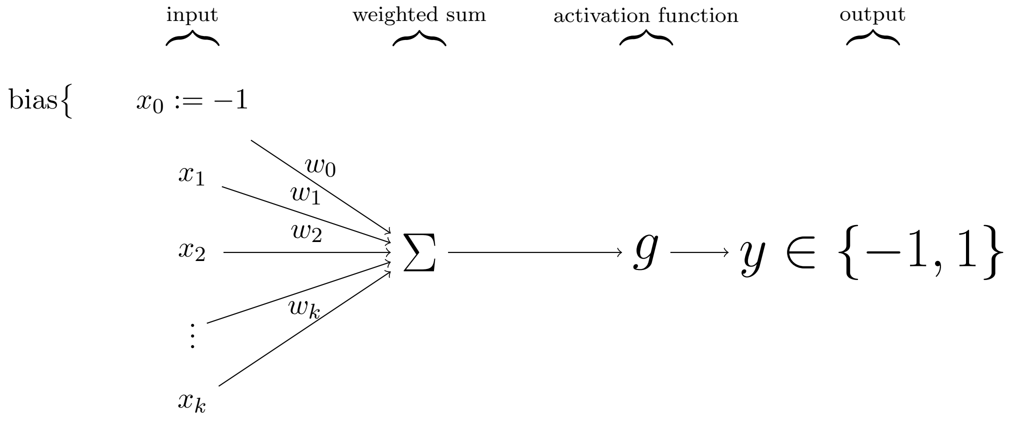
Definition 2.1 - *Neuro-Plasticity*
Neuro-Plasticity is the ability of a neural system to adapt its structure to accommodate new information (i.e. Learn). This can take several forms including growth & function changes.

2.1 Feed-Forward Networks

Definition 2.2 - *Feed-Forward Network*
is an artificial neural network where the connections between nodes are uni-directional. Data is provided to the input layer and then an output is returned from the output layer, no layers are visited twice.

2.1.1 Perceptron

Definition 2.3 - *Perceptron*
A *Perceptron* is an algorithm for supervised learning of a binary classifier. A perceptron defines a hyperplane which acts as a decision boundary which linearly separates the input-state space. These two regions correspond to the two-classes. A perceptron has the following structure.



x_0 is the bias element. It is always set to -1 in the input and the actual value is defined by its weight w_0.

$\boldsymbol{x} = (x_0, \dots, x_k)$ is the input. $(x_1, \dots, x_k)$ are the inputs for the item we wish to classify

$\boldsymbol{w} = (w_0, \dots, w_k)$ is the weights assigned to each input.

Σ is the weighted sum of the bias & inputs. $\Sigma := (\sum_{i=0}^k w_i x_i) = \boldsymbol{w}^T\boldsymbol{x}$

g is the *Activation function* which maps from Σ to $\{-1, 1\}$, effectively performing a binary classification. The user has several options for how to define this. (n.b. $g : \mathbb{R} \to \{-1, 1\}$)

y is the output of the *Activation function.* (i.e. the classification). Typically denoted as $f(\boldsymbol{x}; \boldsymbol{w})$

$$y = g\left(\sum_{i=0}^k x_i w_i)\right) = g(\boldsymbol{w}^T\boldsymbol{x})$$

Remark 2.2 - *Limitations of Perceptron*
A *Perceptron* can only perform linear binary classification so is not useful when two classes are not linearly separable. See `Section 2.2` for how to learn arbitrary decision boundaries.

Proposition 2.1 - *Activation Function*
There are several choice for the *Activation Function* including:

`sign` binarily assigns values depend on whether they are positive or negative.

$$\texttt{sign}(x) := \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases} = \frac{x}{|x|}$$

Proposition 2.2 - *Perceptron (Supervised) Learning Rule*
We need a way for a perceptron to learn when it makes a misclassification. This is done by adjusting the weight vector $\boldsymbol{w}$. A simple learning rule is to update the current weights by a certain proportion of the error made.

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \Delta\boldsymbol{w} \quad \text{where} \quad \Delta\boldsymbol{w} = \begin{cases} \eta f^*(\boldsymbol{x})\boldsymbol{x} & \text{if} \quad \overbrace{f^*(\boldsymbol{x})}^{\text{ground truth}} \neq \overbrace{f(\boldsymbol{x})}^{\text{prediction}} \\ 0 & \text{otherwise} \end{cases}$$

Here, $\eta \in \mathbb{R}^+$ is know as the *Learning Rate.* Remember that $f^*(\cdot) \in \{1, -1\}$.

Proposition 2.3 - *Training Process for a Single-Layer Perceptron*
Let $\{(\boldsymbol{x}_1, f^*(\boldsymbol{x}_1)), \dots, (\boldsymbol{x}_N, f^*(\boldsymbol{x}_N))\}$ be a set of training data. To learn a good set of weights $\boldsymbol{w}$ do the following process.

i). Initialise the weight vector $\boldsymbol{w} = \boldsymbol{0}$

ii). Consider next training datum $(\boldsymbol{x}_i, f^*(\boldsymbol{x}_i))$.

iii). Calculate prediction $f(\boldsymbol{x})$.

iv). Compare prediction $f(\boldsymbol{x})$ and ground truth $f^*(\boldsymbol{x})$.

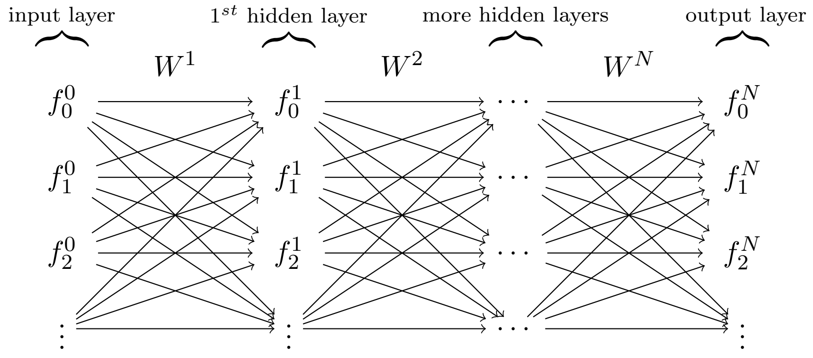
v). Update the weight vector $\boldsymbol{w} = \boldsymbol{w} + \Delta\boldsymbol{w}$ where $\Delta\boldsymbol{w} = \begin{cases} \eta f^*(\boldsymbol{x})\boldsymbol{x} & \text{if } f^*(\boldsymbol{x}) \neq f(\boldsymbol{x}) \\ 0 & \text{otherwise} \end{cases}$

vi). Repeat ii)-v) until the training set is exhausted.

2.1.2 Multi-Layer Perceptron

Remark 2.3 - *Learning Arbitrary Decision Boundaries*
To lean an arbitrary decision boundary (i.e. anything non-linear) can be done by using a *Multi-Layer Preceptron* with non-linear activation functions.

Definition 2.4 - *Multi-Layer Perceptron*
A *Multi-Layer Perceptron* has the same general structure as a perceptron but with multiple calculations occuring and multiple output values. Below is a diagram of a MLP of *depth N* (i.e. there are N layers of computation)



Note that each layer can have a different *width* (i.e. number of nodes in the layer). For each consecutive pair of layers $\boldsymbol{f}^i, \boldsymbol{f}^j$ (of widths n_i, n_j respectively) there is an associated weight matrix $W \in \mathbb{R}^{n_i \times n_j}$ st $\boldsymbol{f}^j = W^T \boldsymbol{f}^i$. VarThe values from the output layer are then passed to an *activation function* to make a classification.

Remark 2.4 - *Using MLPs*
An MLP with a *single* hidden layer is sufficient to represent any boolean or continuous function, althought the layer may be exponentially wider than the input.
An MPL with *two* hidden layers is sufficient to represent any mathematical function.

Proposition 2.4 - *MLPs as Computation Graphs*

$$\begin{aligned}
s_i^j &:= (W^j)^T f^{j-1} && \text{weighted sum of the } i^{th} \text{ node of the } j^{th} \text{ hidden layer} \\
\implies \frac{\partial s_i^j}{\partial w_{ii}^j} &= f_i^{j-1} \\
f_i^j &:= g_i^j(s_i^j) && i^{th} \text{ output vajue of } j^{th} \text{ hidden layer} \\
\implies \frac{\partial f_i^j}{\partial s_i^j} &= \text{depends on def of } g_i^j
\end{aligned}$$

3 Training Algorithms

Definition 3.1 - *Cost/Loss Function, J*
A *Cost Function* $J(\cdot;\cdot)$ is a real-valued measure of how inaccurate a classifier is for a given input configuration (test data & weights). Greater values imply the classifier is less accurate. Here are some common cost functions

$$\text{Expected Loss } J(X;\boldsymbol{w}) = \mathbb{E}[L(f(\boldsymbol{x},\boldsymbol{w}), f^*(\boldsymbol{x}))]$$

$$\text{Empirical Risk } J(X;\boldsymbol{w}) = \frac{1}{|X|}\sum_{\boldsymbol{x}\in X} L(f(\boldsymbol{x},\boldsymbol{w}), f^*(\boldsymbol{x}))$$

Here $L(x, x^*)$ is a measure of loss (distance) between two values. This is defined by the user on a case by case basis. Popular definitions are: $|x - x^*|$, $(x - x^*)^2$ & $\mathbb{1}\{x = x^*\}$

3.1 Gradient Descent

Definition 3.2 - *Gradient Descent*
Gradient Descent aims to learn a set of weight values $\boldsymbol{w}$ which produce a local minimum for a given cost function J. The update rule for gradient descent is

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \underbrace{\eta \cdot \nabla J(X; \boldsymbol{w}_t)}_{\Delta \boldsymbol{w}}$$

$\nabla J(X; \boldsymbol{w}_t)$ is the partial derivative of the cost function wrt to the weights and gives the direction of the greatest descent. We can calculate the i^{th} component of $\Delta \boldsymbol{w}$ after observing $(\boldsymbol{x}, f^*(\boldsymbol{x}))$

$$[\Delta \boldsymbol{w}]_i = \eta x_i (\underbrace{\boldsymbol{w}_t^T \boldsymbol{x}}_{f(\boldsymbol{x}; \boldsymbol{w}_t)} - f^*(\boldsymbol{x}))$$

3.1.1 Auto-Differentitation

Proposition 3.1 - *Calculating Partial Derivatives*
There are three ways to calculate the partial derivatives required for *Gradient Descent.*

- *Symbolic Differentiation* (i.e. algebra). Hard to define to work in all cases.
- *Numerical Differentiation* (i.e. check values in a neighbourhood and approximate the best direction). Easy to implement but low accuracy and high computational cost.
- *Automatic-Differentiation* using feedforward computation graphs. See below

Definition 3.3 - *Feedforward Computational Graph*
Given a series of equations we can construct a *feedforward computational graph. Feedforward computational graphs* have a node for each variable or constant, and then an edge between nodes which are dependent. Once values are defined for all variables at a given depth, values can easily be calculated for variables higher up the tree.

Example 3.1 - *Feedforward Computational Graph*
Consider the following series of equations

$$\begin{array}{rclcrcl} a & = & b \times c & \quad & b & = & d + e \\ c & = & e + 2 & & d & = & 3 + f \\ e & = & f \times g & & & & \end{array}$$

We can construct the following *Computational Graph*

$$\begin{tikzcd}
3 \arrow[dr] & & & \\
& d = 3 + f \arrow[dr] & & \\
f \arrow[ur] \arrow[dr] & & b = d + e \arrow[dr] & \\
& e = f \times g \arrow[ur] \arrow[dr] & & a = b \times c \\
g \arrow[ur] & & c = e + 2 \arrow[ur] & \\
& 2 \arrow[ur] & &
\end{tikzcd}$$

Definition 3.4 - *Auto-Differentiation using a Feedforward Computational Graph*
Consider two nodes in a computational graph x, y and suppose you want to find the partial derivative $\frac{\partial x}{\partial y}$.

i). Establish all the paths from y to x in the graph.

ii). Calculate the partial derivatives of each step of these graphs. (i.e. if there is a path $y \to a \to x$ calculate $\frac{\partial a}{\partial y}, \frac{\partial x}{\partial a}$).

iii). Apply the chain rule along each path (i.e. For $y \to a \to x$ calculate $\frac{\partial a}{\partial y} \cdot \frac{\partial x}{\partial a}$).

iv). Sum these calculations together to get the final result $\frac{\partial x}{\partial y}$.

v). Substitute variables to make computation easier.

Example 3.2 - *Auto-Differentiation using a Feedforward Computational Graph*
Consider the graph in `Example 3.1` and wanting to calculate $\frac{\partial f}{\partial a}$.

i). There are three paths from f to a in the graph: (1) $f \to d \to b \to a$; (2) $f \to e \to b \to a$; and, (3) $f \to e \to c \to a$.

ii). We need to calculate the following partial derivatives: $\frac{\partial d}{\partial f}, \frac{\partial b}{\partial d}, \frac{\partial a}{\partial b}$ for (1); $\frac{\partial e}{\partial f}, \frac{\partial b}{\partial e}, \frac{\partial a}{\partial b}$ for (2); and, $\frac{\partial e}{\partial f}, \frac{\partial c}{\partial e}, \frac{\partial a}{\partial c}$ for (3).

$$\begin{array}{ccc}
(1) & (2) & (3) \\
\begin{aligned} \frac{\partial d}{\partial f} &= 1 \\ \frac{\partial b}{\partial d} &= 1 \\ \frac{\partial a}{\partial b} &= c \end{aligned} &
\begin{aligned} \frac{\partial e}{\partial f} &= g \\ \frac{\partial b}{\partial e} &= 1 \\ \frac{\partial a}{\partial b} &= c \end{aligned} &
\begin{aligned} \frac{\partial e}{\partial f} &= g \\ \frac{\partial c}{\partial e} &= 1 \\ \frac{\partial a}{\partial c} &= b \end{aligned}
\end{array}$$

iii). Applying the chain rule to each path gives

$$\begin{array}{rcl}
(1) \ \frac{\partial d}{\partial f}\frac{\partial b}{\partial d}\frac{\partial a}{\partial b} & = & 1 \cdot 1 \cdot c = c \\
(2) \ \frac{\partial e}{\partial f}\frac{\partial e}{\partial f}\frac{\partial b}{\partial e} & = & g \cdot 1 \cdot 1 \cdot c = gc \\
(3) \ \frac{\partial e}{\partial f}\frac{\partial c}{\partial e}\frac{\partial a}{\partial c} & = & g \cdot 1 \cdot b = gb
\end{array}$$

iv). Summing the terms together we get

$$\frac{\partial a}{\partial f} = c + gc + gb$$

v). By substitution we get a final expression

$$\frac{\partial a}{\partial f} = 2 + 5g + 2fg + 2fg^2$$

So when $f = 4, g = 2$ we have that $a = 150$ and $\frac{\partial a}{\partial f} = 60$.