Machine Learning with Neural Networks

James L. Crowley[1]

[1] Grenoble Insitut Polytechnique, Univ. Grenoble Alpes
`http://crowley-coutaz.fr/jlc/jlc.html`

Abstract. Artificial neural networks provide a distributed computing technology that can be trained to approximate any computable function, and have enabled substantial advances in areas such as computer vision, robotics, speech recognition and natural language processing. This chapter provides an introduction to Artificial Neural Networks, with a review the early history of perceptron learning. It presents a mathematical notation for multi-layer neural networks and shows how such networks can be iteratively trained by back-propagation of errors using labeled training data. It derives the back-propagation algorithm as a distributed form of gradient descent that can be scaled to train arbitrarily large networks given sufficient data and computing power.

Learning Objectives: This chapter provides an introduction to the training and use of Artificial Neural Networks, and prepares students to understand fundamental concepts of deep-learning that are described and used in later chapters.

Keywords: Machine Learning, Perceptrons, Artificial Neural Networks, Gradient Descent, Back-Propagation.

Glossary of Symbols

x_d	A feature. An observed or measured value.
$\vec{X}$	A vector of D features.
D	The number of dimensions for the vector $\vec{X}$.
y	A dependent variable to be estimated.
$a = f(\vec{X};\vec{\omega},b)$	A model that predicts an activation a from a vector $\vec{X}$.
$\vec{w},b$	The parameters of the model $f(\vec{X};\vec{\omega},b)$.
$\{\vec{X}_m\}$ $\{y_m\}$	A set of training samples with indicator values.
$\vec{X}_m$	A feature vector from a training set.
y_m	An indicator or target value for $\vec{X}_m$
M	The number of training samples in the set $\{\vec{X}_m\}$
$a_m = f(\vec{X}_m;\vec{\omega},b)$	The output activation for a training sample, $\vec{X}_m$
$\delta_m^{out} = a_m - y_m$	The error from using $\vec{\omega}$ and b to compute y_m from $\vec{X}_m$
$C(\vec{X}_m,y_m;\vec{\omega},b)$	The cost (or loss) for computing a_m from $\vec{X}_m$ using $(\vec{\omega},b)$
$\vec{\nabla}C(\vec{X}_m,y_m;\vec{\omega},b)$	The gradient (vector derivative) of the cost.
L	The number of layers in a neural network.
l	An index for the l^{th} layer in a network $1 \leq l \leq L$
$a_j^{(l)}$	The activation output of the *j*th neuron of the *l*th layer.
$w_{ij}^{(l)}$	The weight from unit *i* of layer *l–1* to the unit *j* of layer *l*.
$b_j^{(l)}$	The bias for unit *j* of layer *l*.
η	A variable learning rate. Typically very small (0.01)
$\delta_{j,m}^{(l)}$	Error for the jth neuron of layer *l*, from sample $\vec{X}_m$
$\Delta w_{ij,m}^{(l)} = -a_i^{(l-1)}\delta_{j,m}^{(l)}$	Update for the weight from unit *i* of layer *l–1* to the unit *j* layer *l*.
$\Delta b_{j,m}^{(l)} = -\delta_{j,m}^{(l)}$	Update for bias for unit *j* of layer *l*.

1 Machine Learning

Machine learning explores the study and construction of algorithms that can learn from and make predictions about data. The term machine learning was coined in 1959 by Arthur Samuel, a pioneer in the field of computer gaming and inventor of the first computer program capable of learning to play checkers. Machine Learning is now seen as a core enabling technology for artificial intelligence.

Many of the foundational techniques for machine learning were originally developed in the late 19th and early 20th for problems of detecting signals for telegraph and radio communications. Throughout much of the 20th century, the science of machine learning was primarily concerned with recognizing patterns in data. For example, a key reference was the 1973 text-book by Duda and Hart named "Pattern Recognition and Scene Analysis" [1]. However, as the scientific study of machine learning has matured, and as computing and data have become increasingly available, it has become clear that, given enough data and computing, machine learning can be used to learn to imitate any computable function. In particular, machine learning and can be used to learn to control machines, to interact with humans with natural language, and to imitate intelligent behavior. Much of this progress has been due to the emergence of a technology for training artificial neural networks.

Over the last 50 years, artificial neural networks have emerged as a key enabling technology for machine learning. Such networks are composed of layers of identical units, each computing a weighted sum of inputs followed by a non-linear activation function. Such networks are computed as a form of Single-Instruction-Multiple-Data (SIMD) parallel algorithm, and thus can be scaled using parallel computers to arbitrarily large numbers of trainable parameters. The parameters can be learned from training data by a parallel form of gradient descent referred to as back propagation. Back propagation is also a SIMD algorithm. Thus both network computation (forward propagation) and network training (backward propagation) can be implemented using the highly parallel Graphical Processing Units (GPUs) now found in many personal computers, tablets and cell phones.

Many machine learning techniques, including neural networks, were originally developed for pattern recognition. In this case, the classic approach is to use a set of training data $\{\vec{X}_m\}$ to learn the parameters for a model that can compute a vector of activation functions $\vec{a}(\vec{X})$. A decision function, $d\left(\vec{a}\left(\vec{X}_m\right)\right)$, is then used to select a pattern label from a predefined set of possible labels. The full model has the form $f(\vec{X}_m) = d\left(\vec{a}\left(\vec{X}_m\right)\right) = \hat{y}_m$ where $\hat{y}_m$ is a predicted value for the target label, y_m.

For signal generation, a generator function $g\left(\vec{a}\left(\vec{X}\right)\right)$ can be designed (or trained) to approximate (or imitate) a desired output function such as a word, text, sound, image or video. In this case, the full model is $f(\vec{X}_m) = g\left(\vec{a}\left(\vec{X}_m\right)\right) = \vec{Y}_m$ where $\hat{Y}_m$ is a prediction of the desired output for $\vec{X}_m$. During learning, a cost function, *C(-)*, is used to evaluate the quality of the category label or generated signal. The learning process involves adjusting the parameters of the model in order to minimize this cost function, computed from the difference between the actual and desired function.

A variety of techniques have been developed to learn the function $f(\vec{X})$.

Supervised Learning. Most classical methods for machine learning learn to estimate a function from a set of training data with known ground truth labels. The data set is composed of M independent examples, $\{X_m\}$, for which target values $\{y_m\}$ are known. Having a target value for each training sample makes it much easier to estimate the function. Examples of popular techniques include K-nearest neighbors [1], Support Vector Machines [2], and Bayesian Estimation [3].

Semi-Supervised Learning. A number of hybrid algorithms exist that initiate learning from a labeled training set and then extend the learning with unlabeled data, using the initial algorithm to generate synthetic labels for new data [4].

Unsupervised Learning. Unsupervised Learning techniques learn the function without a labeled training set. Most unsupervised learning algorithms are based on clustering techniques that associate data based on statistical properties. Examples include K-nearest neighbors, and Expectation-Maximization [5].

Reinforcement Learning. Reinforcement learning refers to techniques were a system learns through interaction with an environment. While originally developed for training robots to interact with the world, reinforcement learning combined with deep learning has recently produced systems that outperform humans at games such as Go or Chess. Deep reinforcement learning uses training with realistic simulators adapted through additional training with a target domain by transfer learning [6].

Transfer Learning. With transfer learning a system is first trained using supervised learning with a very large general-purpose data set or simulator, and then refined through additional training in a target domain. Transfer learning has provided a very useful method for overcoming the need for very large training data sets for most modern machine learning techniques based on Neural networks. [7]

Self-Supervised Learning. Self-supervised learning learns to reconstruct missing data and to predict associated data from examples. Two classic self-supervised techniques are masked-token replacement and next-token prediction. With Self-Supervised learning, indicator variables are not needed. The data set is its own ground truth. Self-supervised learning makes it possible to pretrain a general purpose system with a very large collection unlabeled training data, followed by transfer learning using supervised learning or reinforcement learning to specialize for a particular problem [8].

2 Perceptrons

The Perceptron is an incremental learning algorithm for linear classifiers. The perceptron was first proposed by Warren McCullough and Walter Pitts in 1943 as a possible universal computational model [9]. In 1957 Frank Rosenblatt at Cornell University, constructed a Perceptron as a room-sized analog computer and demonstrated it for the popular press [10]. Rosenblatt claimed that the Perceptron was "the embryo of an electronic computer that would someday be able to walk, talk, see, write, reproduce itself and be conscious of its existence." Journalists called it the electronic brain. The obvious limitations of the perceptron led serious scientists to denounce such claims as fantasy.

The perceptron is an on-line learning algorithm that learns a linear decision boundary (hyper-plane) for separable training data. As an "on-line" learning algorithm, new training samples can be used at any time to update the recognition algorithm. The Perceptron algorithm uses errors in classifying the training data to iteratively update the hyper-plane decision boundary. Updates may be repeated until no errors exist. However, if the training data cannot be separated by a linear surface, the learning algorithm will not converge, and must be stopped after a certain number of iterations.

Assume a training set of M observations $\{\vec{X}_m\}$ of vectors composed of D components (features), with indicators variables, $\{y_m\}$ where $y_m = \{-1, +1\}$. The indicator variable, y_m, indicates the class label for a sample, $\vec{X}_m$. For a binary pattern detection,

$y_m = +1$ for examples of the target class (class 1)
$y_m = -1$ for all others (class 2)

The Perceptron will learn the coefficients, $\vec{w}, b$ for a linear boundary such that for all training data, $\vec{X}_m$,

$$f(\vec{X}_m) = \begin{cases} 1 & \text{if } \vec{w}^T\vec{X}_m + b \geq 0 \\ -1 & \text{if } \vec{w}^T\vec{X}_m + b < 0 \end{cases} \tag{1}$$

Note that $\vec{w}^T\vec{X}_m + b \geq 0$ is the same as $\vec{w}^T\vec{X}_m \geq -b$. Thus b can be considered as a threshold on the product: $\vec{w}^T\vec{X}_m$.

A training sample is correctly classified if:

$$y_m \cdot \left(\vec{w}^T\vec{X}_m + b\right) \geq 0 \tag{2}$$

The algorithm requires a learning rate, η, typically set to a very small number such as $\eta = 10^{-3}$. Lack of a theory for how to set the learning rate, or even an explanation for why such a small learning rate was required was a frequent criticism. In 1969, Marvin Minsky and Seymour Papert at MIT published a monography entitled "Perceptrons", that claimed to document the fundamental limitations of the perceptron approach [11]. Notably, they observed that a linear classifier could not be constructed to perform an Exclusive OR (XOR) function. Ironically, in their final chapter, they noted that while this is true for a single perceptron, it is not true for multi-layer perceptrons. However, they argued that a multi-layer perceptron would require excessively expensive computations, exceeding what was reasonable for computers at the time.

A more fundamental criticism was that if the data was not separable, then the Perceptron would not converge, and learning would continue as an infinite loop. Thus it was necessary to set a limit for the number of iterations.

3 Artificial Neural Networks

In the 1970s, frustrations with the limits of Artificial Intelligence based on Symbolic Logic led a small community of researchers to explore the use of multi-layer perceptrons for recognizing patterns. As noted above, many of the problems described by Minsky and Papert could be solved with multilayer networks. For example, around this time, Fukushima demonstrated a mult-layered network for the interpretation of handwritten characters known as the Cognitron [12]. It was soon recognized that a perceptron could be generalised to non-separable training data by replacing the "hard" discontinuous decision function with a soft, differentiable "activation function", making it possible to train a perceptron with the classic gradient descent algorithm.

In 1975, Paul Werbos reformulated gradient descent as a distributed algorithm referred to as "Back-propagation" [13]. Back-propagation implements gradient descent as a backwards flow of "correction energy" mirroring the forward flow of activation energy computed by a multi-layer network of perceptrons. Back-propagation made it possible to train and operate arbitrarily large layered networks using SIMD (single-instruction-multiple-data) parallel computing. Multi-layer perceptrons were increasingly referred to as "Artificial Neural Networks".

Artificial Neural Networks are computational structures composed a network of “neural” units. Each neural unit is composed of a weighted sum of inputs, followed by a non-linear decision function.

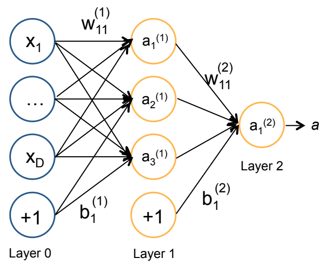

Fig. 1. A fully connected two-layer neural network, with 3 units in the hidden layer (layer 1) and a single unit in the output layer (layer 2). The input vector is composed of the *D* components of a feature vector, $\vec{X}$, plus a constant term that provides a trainable bias for each unit.

During the 1980's, neural networks went through a period of popularity with researchers showing that networks could be trained to provide simple solutions to problems such as recognizing handwritten characters, recognizing spoken words, and steering a car on a highway. However, the resulting systems were fragile and difficult to duplicate. The popularity of Artificial Neural Networks was overtaken by more mathematically sound approaches for statistical pattern recognition based on Bayesian learning [1], [3]. These methods were, in turn, overtaken by techniques such support vector machines and kernel methods [2].

The Artificial Neuron. The simplest possible neural network is composed of a single neuron.

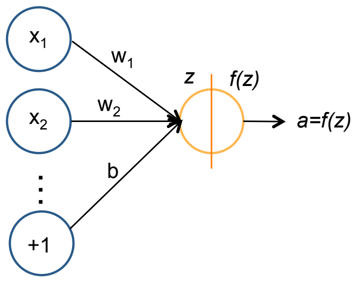

Fig. 2. A single neural unit. On the left side, z is the weighted sum of the coefficients of the input vector, X plus a bias. This is used to compute a non-linear activation, *a=f(z)* shown on the right.

A "neuron" is a computational unit that integrates information from a vector of features, $\vec{X}$, to compute an activation, *a*.

$$a = f(z) \tag{3}$$

The neuron is composed of a weighted sum of input values

$$z = w_1x_1 + w_2x_2 + \ldots + w_Dx_D + b = \vec{w}^T\vec{X} + b \tag{4}$$

followed by a non-linear activation function, $f(z)$.

A classic non-linear decision function is the hyperbolic tangent. This function provides a soft, differentiable, decision surface between +1 and +1, shown in Figure 4.

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{5}$$

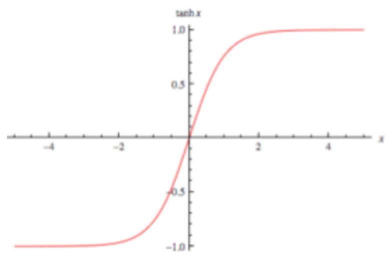

Fig. 4. The hyperbolic tangent is a soft decision function ranging from –1 to 1.

The hyperbolic tangent can be used to define a simple decision function:

$$\text{if } f(\vec{w}^T\vec{X}+b) \geq 0 \text{ then P else N} \tag{6}$$

Most importantly, the hyperbolic tangent has a simple derivative function, making it possible to use gradient descent to train the unit. The derivative of the hyperbolic tangent is:

$$\frac{\partial \tanh(z)}{\partial z} = 1 - \tanh^2(z) \tag{7}$$

3.1 Gradient Descent

Gradient descent is a first-order iterative optimization algorithm for finding the local minimum of a differentiable function. Gradient descent is generally attributed to Augustin-Louis Cauchy, who first suggested it in 1847, and has long been a popular algorithm for estimating parameters for a large variety of models.

The gradient of a scalar-valued differentiable function of several variables, $f(\vec{X})$ is a vector of derivatives:

$$\vec{\nabla} f(\vec{X}) = \frac{\partial f(\vec{X})}{\partial \vec{X}} = \begin{pmatrix} \frac{\partial f(\vec{X})}{\partial x_1} \\ \frac{\partial f(\vec{X})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\vec{X})}{\partial x_D} \end{pmatrix} \tag{8}$$

The gradient of the function $f(\vec{X})$ tells the direction and rate of change for the greatest slope of the function at the point $\vec{X}$. The direction of the gradient is the direction of greatest slope at that point. The magnitude of the gradient is the slope at that point. Following the direction of greatest change provides an iterative algorithm that leads to a local minimum of the function.

Interestingly, the components of the gradient of a function with respect to its parameters tells how much to correct each parameter to move towards the minimum. Subtracting the components of the gradient from the parameters of the function moves the function toward a local minimum. To use this to determine the parameters for a perceptron (or neural unit), we must introduce the notion of cost (or loss) for an error for that function, and then use the gradient to correct the parameters of the function until the cost reaches a minimum.

Cost (Loss) Function. A cost or loss function represent the cost of making an error when classifying or processing data. Assume a training sample $\vec{X}$ with ground truth y and a network function $a = f(\vec{X};\vec{w},b)$ with network parameters $\vec{w}$ and b. The error for a function is $\delta = a - y$. We can compute a cost for this error as 1/2 the square of the error.

$$C(\vec{X},y;\vec{\omega},b) = \frac{1}{2}\left(a-y\right)^2 \tag{9}$$

where the term 1/2 has been included to simplify the algebra when working with the derivative.

The gradient of the cost indicates how sensitive the cost is to each parameter of the network and can be used to determine how much to correct the parameter to minimize errors, as shown in equation 10.

$$\vec{\nabla}C(\vec{X},y;\vec{\omega},b) = \frac{\partial C(\vec{X},y;\vec{\omega},b)}{\partial(\vec{w},b)} = \begin{pmatrix} \frac{\partial C(\vec{X},y;\vec{\omega},b)}{\partial w_1} \\ \vdots \\ \frac{\partial C(\vec{X},y;\vec{\omega},b)}{\partial w_D} \\ \frac{\partial C(\vec{X},y;\vec{\omega},b)}{\partial b} \end{pmatrix} = \begin{pmatrix} \Delta w_1 \\ \vdots \\ \Delta w_D \\ \Delta b \end{pmatrix} \tag{10}$$

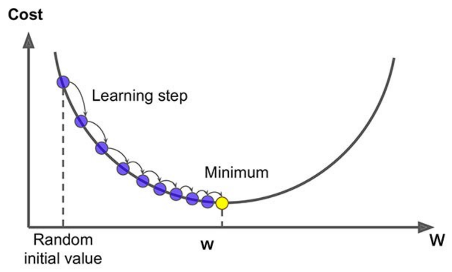

Fig. 5. Descending a gradient to the local minimum.

However, the gradient of the function depends on the point at which the function is evaluated. Real data tend to contain random variations (generally referred to as noise). As a result, while the ensemble of gradients will point in a general direction that improves the network, individual data points generally yield gradients that point in different directions. These random effects are minimized by multiplying the correction by a small learning rate, and using multiple passes through the training data to slowly converge towards a global minimum.

The derivative of the cost with respect to each parameter can be evaluated by using the chain rule. For example, suppose that we have a single neural unit composed of D inputs. The unit will have D weights indexed with the letter d, where $1 \leq d \leq D$. The contribution of the weight, w_d, to this error is given by the derivative of the cost with respect w_d . This can be evaluated by the chain rule, working back from the cost with respect to the activation, a, the derivative of the activation, a, with respect to the linear sum, z, and the derivative of the linear sum, z, with respect to w_d. as shown in equation 11.

$$\Delta w_d = \frac{\partial C(\vec{X},\vec{\omega},b)}{\partial w_d} = \frac{\partial C(\vec{X},\vec{\omega},b)}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_d} \tag{11}$$

The network can be improved by subtracting a fraction of gradient from each of the network parameters. Because the training data typically contains un-modelled phenomena, the correction is weighted by a (very small) learning rate "η" to stabilize learning, as shown in equation 12.

$$\vec{w}^{(i)} = \vec{w}^{(i-1)} - \eta \Delta \vec{w} \text{ and } b^{(i)} = b^{(i-1)} - \eta \Delta b \tag{12}$$

Typical values for η are from η=0.01 to η=0.001 and are often determined dynamically be an optimization algorithm. The superscripts *(i)* and *(i-1)* refer to successive iterations for the values of $\vec{w}$ and b.

The optimum network parameters are the parameters that provide the smallest cost or loss. To determine the best parameters, the network is iteratively refined by subtracting a small fraction of the gradient from the parameters. Ideally, at the optimum parameters, both the loss and the gradient would be zero. This can occur when training a network to approximate a mathematical function or a logic circuit from examples of uncorrupted outputs. However, this is almost never the case for real data, because of the intrinsic variations of samples within the data. As a result, if you evaluate gradient descent by hand with real world data samples, do not expect to easily see a path to convergence. Arriving at an optimum typically requires many passes through the training data, where each pass is referred to as an "epoch". Gradient descent may require many epochs to reach an optimal (minimum loss) model.

Feature Scaling. For a training set $\{\vec{X}_m\}$ of M training samples with D values, if the individual features do not have a similar range of values, than features with a larger range of values will dominate the gradient resulting in non-optimal convergence. One way to assure sure that features have similar ranges is to normalize the range of each feature with respect to the entire training set. For example, the range for each feature can be normalized by subtracting the minimum for that feature from the training data and dividing by the range, as shown in equation 13.

$$\forall_{m=1}^{M} : x'_{dm} := \frac{x_{dm} - \min(x_d)}{\max(x_d) - \min(x_d)} \tag{13}$$

This has the effect of mapping a highly elongated cloud of training data into a sphere, as shown in figure 6. Ideally this will lead to a spherical cost function with the minimum at the center.

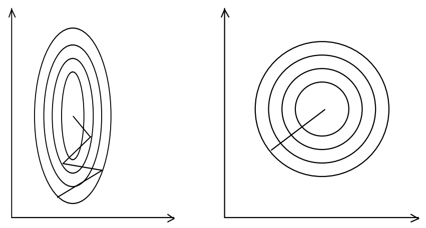

Fig. 6. Feature scaling improves gradient descent by projecting the data to a space where features have a similar range if values. This reduces the effects of small variations in the gradient.

Note that the 2-D surface shown in figure 6 is an idealized representation that portrays only two parameters, for example w, b for a single neural unit with a scalar input x. The actual cost surfaces are hyper-dimensional and not easy to visualize.

Local Minima. Gradient descent assumes that the loss function is convex. However, the loss function depends on real data with un-modeled phenomena (noise). Variations in the training samples can create a non-convex loss with local minima.

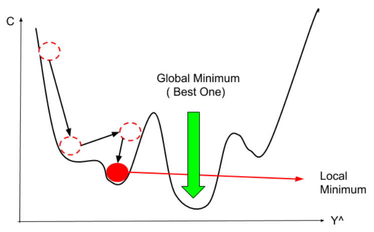

Fig. 7. Variations in the training samples can create a non-convex loss with local minima.

In most networks, the gradient has many parameters, and the cost function is evaluated in a very high dimensional space. It is helpful to see the data as a hyper-dimensional cloud descending (flowing over) a complex hyper-dimensional surface, as shown in figure 8.

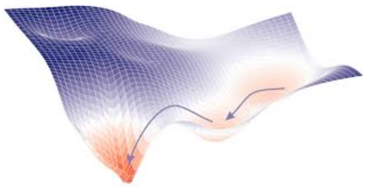

Fig. 8. The data can be seen as a hyper-dimensional cloud descending (flowing over) a complex hyper-dimensional surface. (Drawing recovered from the internet - Source unknown)

Batch Mode: A popular technique to reduce the random variations of corrections is to average the gradients from many samples, and then correct the network with this average. This is referred to as "“batch mode”. The training data is divided into “folds” composed of M samples, and an average correction vector is computed from the average of the gradients of the M samples.

$$\Delta\vec{w} = \frac{1}{M}\sum_{m=1}^{M}\Delta\vec{w}_m = \frac{1}{M}\sum_{m=1}^{M}\vec{\nabla}C_m \tag{14}$$

The model is then updated with the average gradient, using a learning rate “η” to stabilize learning as described above.

$$\vec{w}^{(i)} = \vec{w}^{(i-1)} - \eta\Delta\vec{w} \tag{15}$$

While batch mode tends to be more efficient than updating the network with each sample, it is more likely to learning that is stuck in a local minimum. This likelihood can be reduced using stochastic gradient descent.

Stochastic Gradient Descent. With Stochastic gradient descent, individual training samples are randomly selected and used to update the model. This will send the model in random directions that eventually flow to the global minima. While much less efficient than batch mode, this is less likely to become stuck in local minima.

3.2 Multilayer Neural Networks

Artificial Neural Networks are typically composed of multiple layers of neural units, where each each neural unit is composed of a weighted sum of inputs, followed by a non-linear decision function. Such structures are often referred to as multi-layer perceptrons, abbeviated as MLP. For example, figure 1 shows a 2-layer network with 3 hidden units at layer *l=1* and a single output unit at level *l=2*.

We will use the following notation to describe MLPs:

L	The number of layers (Layers of non-linear activations).
l	The layer index. l ranges from 0 (input layer) to L (output layer).
$N^{(l)}$	The number of units in layer l. $N^{(0)}=D$

$\vec{a}^{(0)} = \vec{X}$	is the input layer. For each component $a_i^{(0)} = x_d$
$a_j^{(l)}$	The activation output of the j^{th} neuron of the l^{th} layer.
$w_{ij}^{(l)}$	The weight from the unit *i* of layer *l-1* for the unit *j* of layer *l*.
$b_j^{(l)}$	The bias term for j^{th} unit of the l^{th} layer
f(z)	A non-linear activation function, such as a sigmoid, relu or tanh.

Note that all parameters carry a superscript, referring to their layer. For example:

$a_1^{(2)}$ is the activation output of the first neuron of the second layer.

$w_{13}^{(2)}$ is the weight for neuron 1 from the first level to neuron 3 in the second layer.

The network in figure 1 would be described by:

$$a_1^{(1)} = f(w_{11}^{(1)}X_1 + w_{21}^{(1)}X_2 + w_{31}^{(1)}X_3 + b_1^{(1)})$$

$$a_2^{(1)} = f(w_{12}^{(1)}X_1 + w_{22}^{(1)}X_2 + w_{32}^{(1)}X_3 + b_2^{(1)})$$

$$a_3^{(1)} = f(w_{13}^{(1)}X_1 + w_{23}^{(1)}X_2 + w_{33}^{(1)}X_3 + b_3^{(1)})$$

$$a_1^{(2)} = f(w_{11}^{(2)}a_1^{(1)} + w_{21}^{(2)}a_2^{(1)} + w_{31}^{(2)}a_3^{(1)} + b_1^{(2)}) \tag{16}$$

This can be generalized to multiple layers. For example, figure 9 shows a 3-layer network, with an output vector $\vec{a}^{(3)}$ at layer 3 composed of 2 units.

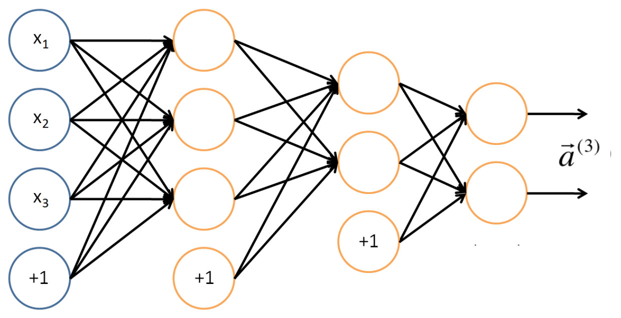

Fig. 9. A three layer network with an output vector of size *N*$^{(3)}$*=2* at level 3.

In general, unit *j* at layer *l* is computed as shown in equation 17, illustrated by figure 10.

$$a_j^{(l)} = f\left(\sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}\right) \tag{17}$$

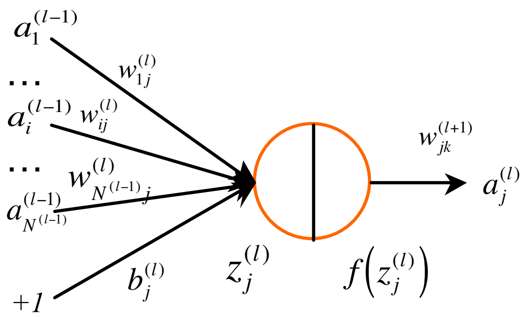

Fig. 10. The jth unit at layer *l*.

3.3 Activation Functions

Neural units are composed of a weighted sum of inputs followed by a non-linear activation function. The activation function must have a first derivative, so that the network can be trained using gradient descent.

For binary problems, the hard decision surface of the original perceptron is easily replaced with is the hyperbolic tangent:

$$f(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

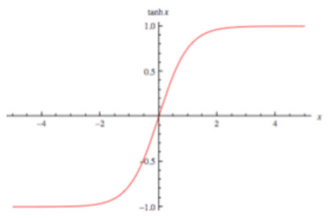

Fig 11. The hyperbolic Tangent activation function

The derivative for the hyperbolic tangent is:

$$\frac{\partial \tanh(z)}{\partial z} = \frac{1}{\cosh^2(z)} = \operatorname{sech}^2(z) \qquad (18)$$

A popular choice for activation function is the logistic sigmoid function. The logistic function is also known as the S-shaped curve or sigmoid curve and is widely used to model saturation phenomena in fields such as biology, chemistry, and economics.

$$f(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

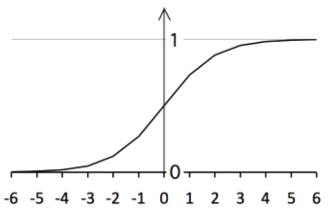

Fig. 12. The sigmoid function, σ(z).

The sigmoid has a simple derivative function

$$\frac{\partial\sigma(z)}{\partial z} = \sigma(z)(1-\sigma(z)) \tag{19}$$

Because the output of the sigmoid varies from 0 to 1, the sigmoid is easily generalized for multi-class problems, using the Softmax activation function.

$$f(z_k) = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}} \tag{20}$$

The softmax function takes as input a vector $\vec{z}$ of K real numbers, and normalizes it into a distribution consisting of vector of K values that sum to 1. The result of a softmax sums to 1, and can be interpreted as a probability distribution indicating the likelihood of each component. Softmax is widely used as the output activation function for network with multiclass outputs.

The rectified linear function is popular for intermediate levels in deep networks because of its trivial derivative:

$$\mathrm{ReLu}(z) = \max(0, z)$$

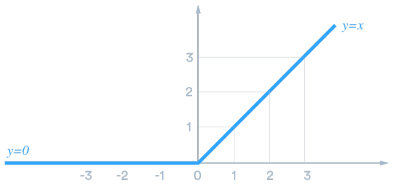

Fig. 13. The rectified linear (ReLu) activation function .

For z ≤ 0, the derivative is 0. For $z > 0$ the derivative is 1. Many other activations have been proposed, for a variety of uses.

3.4 Backpropagation as Distributed Gradient Descent

As stated above, the weights and biases for a multi-layer network can be computed using the back-propagation algorithm. Back-propagation is a distributed algorithm for computing gradient descent and can thus be scaled to extremely large networks composed of billions of parameters, provided sufficient computing and training data are available. The back propagation can be directly derived from the gradient using the chain rule. Back-propagation propagates error terms back through the layers, using the weights and activations of units to determine the component of the gradient for each parameter.

As a form of gradient descent, back-propagation determines the correction factors to adjust the network the weights $w_{ij}^{(l)}$ and biases $b_j^{(l)}$ so as to minimize an error function between the network output $\vec{a}_m^L$ and the target value $\vec{y}_m$ for the M training samples $\{\vec{X}_m\}$, $\{\vec{y}_m\}$. Each input sample is input to the network to compute an output activation, $\vec{a}_m^L$. This can be referred to as forward propagation. An error term is then computed from the difference of $\vec{a}_m^L$ with the target vector, $\vec{y}_m$. This error is propagated back through the network to determine the correction vector for each parameter.

To keep things simple, let us consider the case of a two class network with a single unit, so that δ_m^{out}, a_m, and y_m are scalars. The results are easily generalized to vectors for multi-class networks with multiple layers. For a single neuron, at the output layer, the "error" for each training sample is the error resulting from using a_m as a value for y_m.

$$\delta_m^{out} = \left(a_m - y_m\right) \tag{21}$$

The error term δ_m^{out} is the total error for the whole network for sample m. This error is used to compute an error term for the weights that activate the neuron: δ_m. As shown in figure 14.

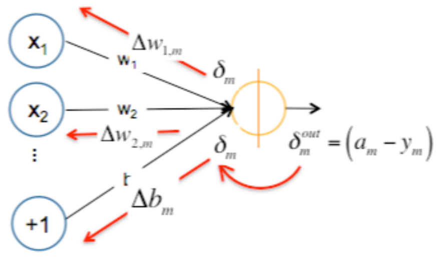

Fig. 14. The error term is propagated back through each unit to adjust the weights and biases.

The error term for the input activations is given equation 22:

$$\delta_m = \frac{\partial f(z)}{\partial z}\delta_m^{out} \tag{22}$$

This correction is then used to determine a correction term for the weights:

$$\Delta w_{d,m} = x_d \delta_m \tag{23}$$

$$\Delta b_m = \delta_m \tag{24}$$

Back propagation can be easily generalized for multiple neurons at multiple layers (*l=1* to *L*) with multiple outputs (*k=1 to K*). In this case, both the network output activation and the indicator variable would be vectors with k components, with one component for each class, giving an error vector, $\vec{\delta}_m^{out}$ as shown in equation 22.

$$\vec{\delta}_m^{out} = \left(\vec{a}_m^{(out)} - \vec{y}_m\right) \text{ with k components: } \delta_{k,m}^{out} = \left(a_{k,m}^{(out)} - y_{k,m}\right) \tag{22}$$

The error term for unit *k* at layer *L* is:

$$\delta_{k,m}^{(L)} = \frac{\partial f(z_k^{(L)})}{\partial z}\delta_m^{out} \tag{23}$$

For the hidden units in layers $l < L$ the error $\delta_j^{(l)}$ is based on a weighted average of the error terms for $\delta_k^{(l+1)}$.

$$\delta_{j,m}^{(l)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}}\sum_{k=1}^{N^{l+1}} w_{jk}^{(l+1)}\delta_{k,m}^{(l+1)} \tag{24}$$

An error term, $\delta_j^{(l)}$ is computed for each unit *j* in layer *l* and projected back to layer *l–1* using the sum of errors times the corresponding weights times the derivative of the activation function. This error term tells how much the activation of unit *j* was responsible for differences between the output activation vector of the network $\vec{a}_m^{(L)}$ and the target vector $\vec{y}_m$. This error term can then used to correct the weights and bias terms leading from layer *j* to layer *i*.

$$\Delta w_{ij,m}^{(l)} = a_i^{(l-1)}\delta_{j,m}^{(l)} \tag{25}$$

$$\Delta b_{j,m}^{(l)} = \delta_{j,m}^{(l)} \tag{26}$$

Note that the corrections $\Delta w_{ij,m}^{(l)}$ and $\Delta b_{j,m}^{(l)}$ are NOT applied until after the error has propagated all the way back to layer *l=1*, and that when *l=1,* $a_i^{(0)} = x_i$.

For "batch learning", the corrections terms, $\Delta w_{ji,m}^{(l)}$ and $\Delta b_{j,m}^{(l)}$ are averaged over M samples of the training data and then only an average correction is applied to the weights.

$$\Delta w_{ij}^{(l)} = \frac{1}{M}\sum_{m=1}^{M}\Delta w_{ij,m}^{(l)}$$
$$\Delta b_j^{(l)} = \frac{1}{M}\sum_{m=1}^{M}\Delta b_{j,m}^{(l)} \tag{27}$$

then

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta\cdot\Delta w_{ij}^{(l)}$$

$$b_j^{(l)} \leftarrow b_j^{(l)} - \eta\cdot\Delta b_j^{(l)} \tag{28}$$

where η is the learning rate.

Back-propagation is a parallel algorithm to compute a correction term for each network parameter from the gradient of the loss function. A common problem with gradient descent is that the loss function can have local minimum. This problem can be minimized by regularization. A popular regularization technique for back propagation is to use "momentum"

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \cdot \Delta w_{ij}^{(l)} + \mu \cdot w_{ij}^{(l)}$$
$$b_j^{(l)} \leftarrow b_j^{(l)} - \eta \cdot \Delta b_j^{(l)} + \mu \cdot b_j^{(l)} \tag{29}$$

where the terms $\mu \cdot w_j^{(l)}$ and $\mu \cdot b_j^{(l)}$ serves to stabilize the estimation.

Bibliography

1. Duda, Richard O. and Peter E. Hart, *Pattern classification and scene analysis*. New York, Wiley, 1973.

2 Cristianini, Nello, and John Shawe-Taylor. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

3. Bishop, Christopher. M., and Nasser M. Nasrabadi. *Pattern Recognition and Machine Learning*. New York: Springer, 2006.
4. van Engelen, J.E., Hoos, H. H. A survey on semi-supervised learning. Mach Learn 109, 373–440 (2020).
5. Dempster, Arthur P., Nan M. Laird, and Donald B. Rubin. "Maximum likelihood from incomplete data via the EM algorithm." Journal of the Royal Statistical Society: Series B (Methodological) 39, no. 1, pp1-22, 1977.
6. Sutton, Richard S., and Andrew G. Barto. "Reinforcement learning." Journal of Cognitive Neuroscience 11, no. 1, pp126-134, 1999.
7. Weiss, K., Khoshgoftaar, T.M. & Wang, D. A survey of transfer learning. J Big Data, Vol 3, No 9, 2016.
8. Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805, 2018.
9. McCulloch, Warren S., and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity." *The bulletin of mathematical biophysics* 5.4 : 115-133, 1943.
10. Rosenblatt, Frank: "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain.", *Psychological Review*, 65, No. 6, pp386-408 (1958)
11. Minsky, Marvin, and Seymour Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press, 1969.
12. Fukushima, Kunihiko, "Cognitron: A self-organizing multilayered neural network". *Biological Cybernetics*, vol 20, pp121-136, 1975.
13. Werbos, Paul J. "Backpropagation through time: what it does and how to do it." *Proceedings of the IEEE* Vol 78, No 10, pp1550-1560, 1990.