Fundamentals of Neural Networks

Seminar Data Mining

Mathias Jackermeier
Fakultät für Informatik
Technische Universität München
Email: mathias.jackermeier@tum.de

***Abstract*— Neural networks are biologically inspired computation models that have shaped the field of machine learning over the last decade, enabling breakthroughs in numerous application areas ranging from image processing to natural language understanding. In this paper, we review the fundamental principles behind neural networks, providing the reader with the necessary mathematical knowledge to be able to use them in practice. We examine the development of neural networks from simpler models, their architecture and related design decisions, and their mathematical specification. We then explore the most common training algorithm employed in neural networks. Finally, we briefly discuss more advanced network architectures and the continued impact we believe neural networks will have on machine learning.**

***Index Terms*— Artificial Intelligence, Machine Learning, Neural Networks, Stochastic Gradient Descent, Back-propagation**

I. Introduction

Artificial intelligence systems have been becoming more and more powerful over the last 10 years. We have seen outstanding advances in a variety of fields including computer vision, natural language processing and fraud detection, which power many end-user technologies such as digital assistants or self-driving cars. Much of the recent progress can be attributed to *deep learning*, a powerful set of techniques that enable computers to understand the world by decomposing complex concepts into a hierarchy of simpler abstractions.

While numerous other approaches to machine learning exist, deep learning has shown to outperform other methods in a wide variety of applications. To name a few examples, deep learning models dominate the task of object recognition in images [1], even surpassing human-level performance [2], have been successfully applied to sentiment analysis [3], and have significantly improved speech recognition systems [4]. Deep learning has also been used in problems such as style transfer between images [5], image description generation [6], and learning to play video games [7].

By learning everything required to solve a task purely from raw data, these techniques have alleviated the need for problem-specific expert knowledge. Thus, very similar models building on the same core ideas can be applied to a vast array of different tasks with outstanding success.

One such core idea that is fundamental to deep learning is the *neural network*, a computing model loosely inspired by neuroscience. While neural networks are not new, it was not until recently that enough data and computational resources became available to train them effectively and fully appreciate their power [8, Ch. 1, pp. 18-21].

Since neural networks have become so prevalent in modern machine learning applications, many libraries exist that abstract their concepts and provide simple programming interfaces. However, it does not suffice to be familiar with such libraries to use neural networks effectively; in order to understand which architectures perform well, and why, one must also know their mathematical foundations.

In this paper we thus aim to give a thorough overview of neural networks and the fundamental techniques and algorithms associated with them. We first briefly examine the motivation and history behind neural networks in Section II by introducing the *perceptron* model. Section III then shows how this model has been adjusted and extended to obtain the neural network, focusing in particular on *feedforward neural networks*. In Section IV, we explain how these networks can be trained, introducing ideas such as *stochastic gradient descent* and *back-propagation*. Section V then briefly examines extensions of feedforward neural networks that are commonly used in practice, before we conclude our paper in Section VI.

II. The Perceptron

When researchers developed the first machine learning models, they often used ideas based closely on our understanding of the brain. One such model, inspired by the biological neuron, is the perceptron [9].

Like its biological counterpart, the perceptron receives information and produces an output. More specifically, it accepts n input values $x_1, \dots, x_n$ and calculates a corresponding output value $\hat{y} \in \{-1, 1\}$ by computing

$$\hat{y} = \operatorname{sign}\left(\sum_{i=1}^{n} w_i x_i\right), \tag{1}$$

where the weights w_i are the parameters of the model, and $\operatorname{sign}(x)$ is defined as

$$\operatorname{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x \leq 0. \end{cases} \tag{2}$$

By representing the input values and weights as vectors $\boldsymbol{x}$ and $\boldsymbol{w}$, we can rewrite Eq. (1) as

$$\hat{y} = \operatorname{sign}(\boldsymbol{w}^\top \boldsymbol{x}). \tag{3}$$

For a visual representation of this model, see Fig. 1.

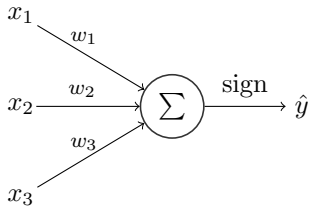

Fig. 1. An illustration of the perceptron model. In this example, the perceptron accepts three inputs x_1, x_2, x_3, has the parameters w_1, w_2, w_3, and computes $\hat{y} = \text{sign}(w_1 x_1 + w_2 x_2 + w_3 x_3)$.

Perceptron models can be used to solve binary classification problems. In this scenario, we are given a list of m training examples $\mathbb{X} = (\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)})$ and their corresponding binary labels $\mathbb{Y}$, and wish to predict the most probable label for an unseen vector $\boldsymbol{x} \notin \mathbb{X}$.

For example, the vectors $\boldsymbol{x}^{(i)}$ might describe features of an email using a *bag-of-words* representation. That is, we define a fixed vocabulary, and the j^{th} entry in the vector $\boldsymbol{x}^{(i)}$ specifies how often the j^{th} word of the vocabulary occurs in the particular email represented by $\boldsymbol{x}^{(i)}$. The corresponding label $y^{(i)} = 1$ then might signify that the email is a legitimate email, whereas a value of $y^{(i)} = -1$ might label the email as spam.

In the beginning, the weights are randomly initialized and the model thus makes arbitrary predictions. During the process of *training* the perceptron, we iteratively adjust the weights in order to improve the prediction accuracy on the training set.

One common method of training is the perceptron learning algorithm proposed by Rosenblatt [10]. Essentially, the algorithm iterates through the training data $\mathbb{X}$ and makes small adjustments to the weights if a particular training example $\boldsymbol{x}^{(i)}$ is misclassified. For example, if the perceptron predicts $\hat{y} = 1$ and the actual label is $y^{(i)} = -1$, the weights are corrected in the negative direction. Since the perceptron learning algorithm is not directly applicable to neural networks, we will not discuss it further; a more in depth explanation can be found in Ref. [11, Ch. 8, pp. 265-267].

A major shortcoming of the perceptron is that it can only learn to classify linearly separable data [12]. For example, the XOR function, where

$$\text{XOR}(\boldsymbol{x}) = \begin{cases} 0 & \text{if } \boldsymbol{x} = [0,0] \vee \boldsymbol{x} = [1,1] \\ 1 & \text{if } \boldsymbol{x} = [1,0] \vee \boldsymbol{x} = [0,1], \end{cases} \tag{4}$$

cannot be learned with the perceptron. The discovery of these limitations has greatly reduced interest in the field of biological learning, until more sophisticated models, such as neural networks, were developed [8, Ch. 1, pp. 12-18].

III. Feedforward Neural Networks

A natural extension of the perceptron model is to combine multiple perceptrons in a network architecture called a neural network. It is intuitively clear that, much like in an organic brain, a complex arrangement of many simple computing units can learn much more complicated functions than those simple units alone. In this section, we will examine how such a network architecture based on perceptrons can be constructed. The ideas that we develop are mostly based on Ref. [8, Ch. 6].

A. Extensions to the Perceptron

Before explaining the composition of perceptrons to neural networks, we will first explore two common extensions to the perceptron model, which are required in the network architecture we develop.

First, we introduce an additional term called *bias* to the output calculation. The new output $\hat{y}$ becomes

$$\hat{y} = \text{sign}(\boldsymbol{w}^\top \boldsymbol{x} + b), \tag{5}$$

where the scalar b is the bias. This additional learnable parameter shifts the function computed by the model *independently of its input*. If the bias is large and positive, the model is generally more inclined to predict a positive label, while a negative bias makes it more likely that negative labels are predicted.

Second, we generalize the perceptron by replacing the sign function with an arbitrary function f called *activation function*. In contrast to $\text{sign}(x)$, most activation functions used in neural networks are continuous, since this enables us to use a variety of *gradient-based* learning algorithms for training as we will see in Section IV. Concrete examples of activation functions will be discussed later in this section.

We also introduce the quantity z, the *weighted input*, which simply is defined as

$$z = \boldsymbol{w}^\top \boldsymbol{x} + b. \tag{6}$$

The computing units we have obtained with these modifications to the perceptron are generally called *neurons* or simply *units*.

B. Network architecture

Any arrangement of neurons in a network architecture can be considered a neural network. The most influential such architecture is the feedforward neural network, which forms the basis for many other more advanced neural networks [8, Ch. 6, p. 163]. Feedforward neural networks are sometimes also called *multilayer perceptrons* (MLPs).

In feedforward neural networks, the computing units are arranged in layers. We distinguish between the *output layer*, the *hidden layers*, and the *input layer*. The output layer is the final layer in the network were its actual output is produced. The input layer is the first layer in the network, and all layers in between are called hidden layers. The input layer is special in that it does not compute anything; it merely represents the input that is passed into the neural network. In general, a L-layer feedforward neural network consists of one input layer, $(L-2)$ hidden layers, and an output layer. We call the number of layers L the *depth* of the model.

Every neuron in a layer l receives input from all neurons in the $(l-1)^{\text{th}}$ layer. There are no connections between neurons in the same layer, and we also do not allow feedback connections into previous layers. Neurons in the hidden and

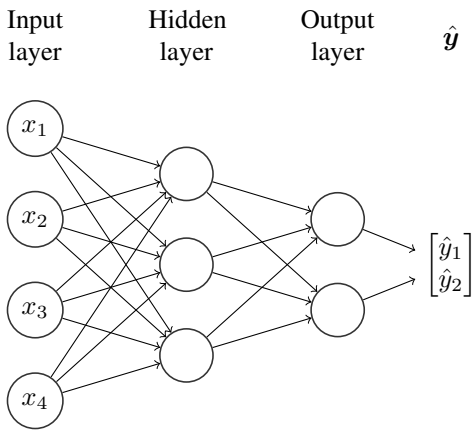

Fig. 2. A three-layer neural network. The network accepts an input $\boldsymbol{x} \in \mathbb{R}^4$, propagates it through its hidden layer consisting of three units, and finally produces an output $\hat{\boldsymbol{y}} \in \mathbb{R}^2$ in the output layer. The weights, sums, and activation functions have been omitted.

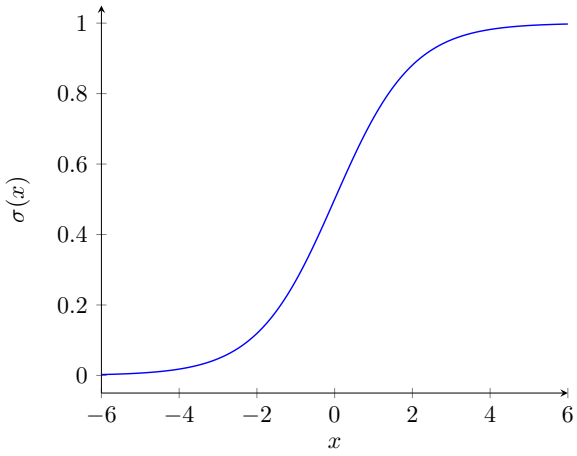

Fig. 3. The logistic sigmoid function.

output layers behave exactly like the modified perceptron; the only important detail is that their input is the output from the previous layer. An illustration of a feedforward neural network can be found in Fig. 2.

In the remainder of this section, we discuss the individual layers and corresponding design decisions in more detail.

1) Output layer: The design of the output layer depends mostly on the task that we wish to perform with the neural network.

If we want to predict a numerical value, a problem known as *regression*, we only use a single linear neuron in the output layer. A linear neuron simply uses the identity function as its activation function.

In *classification*, we wish to predict the class of an input vector $\boldsymbol{x}$, given a set of classes. For example, as with the perceptron, we might want to distinguish normal emails from spam emails. In this case of *binary* classification, it is common to use the activation function

$$\sigma(x) = \frac{1}{1 + \exp(-x)}, \tag{7}$$

called the logistic sigmoid, in combination with a single output unit. As shown in Fig. 3, $\sigma(x)$ squashes its input to a value between 0 and 1, which can be interpreted as a probability distribution. Thus, it is an excellent choice for binary classification problems: the output $\hat{y}$ of the neural network is the probability that $\boldsymbol{x}$ belongs to class 1, while $(1 - \hat{y})$ is the probability that $\boldsymbol{x}$ belongs to class 0.

In *multiclass* classification problems, where we wish to predict a probability distribution over k different classes, we construct an output layer with k units. A generalization of the logistic sigmoid, called the $\mathrm{softmax}$ function

$$\mathrm{softmax}(x) = \frac{\exp(x)}{\sum_{i=1}^{k} \exp(z_i)}, \tag{8}$$

is commonly used as activation function in this scenario. In Eq. (8), z_i represents the weighted input z of the i^{th} neuron in the output layer. We can easily see that the $\mathrm{softmax}$ function creates a valid probability distribution and that neurons that have a large weighted input produce a higher probability. The output $\hat{y}_i$ of the neural network is the probability it assigns to the i^{th} class.

Feedforward neural networks can also be applied to many other tasks such as *structured output prediction*, *anomaly detection*, and *synthesis* [8, Ch. 5, pp. 96-100]. Many specialized architectures exist for these types of problems, but they are beyond the scope of this paper.

2) Hidden layers: In contrast to the output layer, the task that we want so solve does not give us any information about how to design the hidden layers. The first and foremost design decision that comes to mind is the depth of the neural network.

From a purely theoretical point of view, the depth of a network is irrelevant. It can be shown that a feedforward neural network with just one hidden layer can approximate any function arbitrarily well, given that enough hidden units are available [13], [14]. However, this does not mean that we know how to design or train such a network, and deeper networks almost always perform better in practice.

We can think of the hidden layers as a means to learn a different representation of the input. The neural network transforms the input by propagating it through its hidden layers until it obtains a representation where it can perform the assigned task much easier. For example, in object recognition, one can show that neural networks learn to detect different features such as edges or individual object parts in different hidden layers [15]. It is thus not surprising that deeper neural networks often perform better than shallow ones. On the other hand, deep neural networks are often difficult and computationally expensive to train.

Another important consideration is the activation function used in hidden layers. Common activation functions include the logistic sigmoid, which we have already presented, the $\tanh$ function, which is only a variation of the logistic sigmoid function, and the rectified linear function $g(x) = \max\{0, x\}$.

Activation functions are continuous, a property needed

for the training algorithm, and generally non-linear, since a network consisting of only linear hidden layers is linear as a whole, suffering from the same drawbacks that we saw in the perceptron model [8, Ch. 6, p. 190].

The design of neural networks is mostly driven by experimentation and not many theoretical tools exist that justify the use of one architecture over another. For example, rectified linear units often outperform other types of hidden units [16], [17], but we are far from a rigorous understanding of why this is the case. Thus, a network design for a particular task is mostly chosen via trial and error [8, Ch. 6, pp. 185-187].

3) Input layer: Since the input layer only represents the input passed to the neural network, there are not many design decisions to be made. We can only choose how we wish to represent the input. Often, this representation follows immediately from the raw data–for example, in image classification, we can simply represent an input image as a vector of pixel values. In other cases, the input representation might not be that obvious and we may need carefully hand-crafted features.

C. Mathematical formulation

As we have seen, a feedforward neural network is simply a combination of many perceptron-like computing units. We can combine all parameters of these neurons in weight matrices and bias vectors to obtain one single mathematical specification of the whole network.

In particular, we define a weight matrix $\boldsymbol{W}^{(l)}$ for every layer l except the input layer, where the entry in the j^{th} row and k^{th} column equals the weight of the j^{th} neuron in the $(l-1)^{\text{th}}$ layer to the k^{th} neuron in the l^{th} layer. We specify the entries of the weight matrices with $w_{jk}^{(l)}$. We similarly define bias vectors $\boldsymbol{b}^{(l)}$ whose j^{th} entry contains the bias of the j^{th} neuron in the l^{th} layer. We denote the activation function used in the l^{th} layer with $f^{(l)}$. Activation functions are applied element-wise to vectors.

We can now denote the output $\boldsymbol{a}^{(l)}$ of the l^{th} layer as

$$\boldsymbol{a}^{(l)} = f^{(l)}\left(\boldsymbol{W}^{(l)\top}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}\right), \tag{9}$$

where $\boldsymbol{a}^{(0)} = \boldsymbol{x}$. The output of the whole network then is simply given by $\hat{\boldsymbol{y}} = \boldsymbol{a}^{(L)}$. We also define the vector of weighted inputs $\boldsymbol{z}^{(l)}$ at layer l as

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)\top}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}. \tag{10}$$

Note that these expressions are very similar to the formulation of the perceptron model in Eq. (5) and (6). The only difference is that we now use matrices, since there are multiple neurons in a layer, and that the total output consists of a chain of multiple such functions.

As shorthand notation, we will sometimes refer to the parameters of a neural network as $\boldsymbol{\theta}$ and to the whole neural network as $f(\boldsymbol{\theta})$, abstracting away the individual layers.

IV. Training Feedforward Neural Networks

Similar to the perceptron model, when training neural networks, we have a list of m training examples $\mathbb{X} = (\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)})$ with corresponding labels $\mathbb{Y}$, and wish to iteratively adjust the parameters $\boldsymbol{\theta}$ of the neural network to learn a mapping from $\mathbb{X}$ to $\mathbb{Y}$. The parameters are usually randomly initialized.

In a binary classification setting, the labels $y^{(i)}$ are either 0 or 1 to indicate one of the two classes. In multiclass problems, they encode one of k classes in a *one-hot* fashion: in this case, $\boldsymbol{y}^{(i)}$ is a k-dimensional vector whose i^{th} entry is 1 if $\boldsymbol{y}^{(i)}$ represents the i^{th} class. All other entries are equal to 0. Finally, in regression, the labels are simply the scalar values that we want to predict.

Suppose we are given such a set of training data and a neural network $f(\boldsymbol{\theta})$. In this section, we explore how we can choose $\boldsymbol{\theta}$ in order that the network learns the mapping described by the training data, mainly utilizing concepts presented in Ref. [8] and [18].

A. Cost functions

Since we start with a random set of parameters which we wish to improve, we need some kind of measure of how good the network performs. For this, we introduce the *cost function* $J(\boldsymbol{\theta})$, sometimes also referred to as *loss* or *error* function. $J(\boldsymbol{\theta})$ produces a scalar cost which is non-negative and the closer it is to 0, the better our network performs. We can thus reframe the training problem as *minimizing the cost function*.

The cost functions that we will consider can all be represented as sums over the costs of the individual training examples:

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta}), \tag{11}$$

where $\mathcal{L}$ is the loss for an individual training example. We scale J by $1/m$ to make the cost independent of the number of training examples m.

The particular per-example cost function $\mathcal{L}$ is chosen based on the task that we wish to perform.

In regression, a good choice is the *mean squared error*

$$\mathcal{L}(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{2}(\hat{y} - y)^2, \tag{12}$$

which simply corresponds to the distance of the desired output y from the actual output $\hat{y}$. It can easily be seen that it satisfies the properties of a cost function: it is always non-negative and if $\hat{y}$ is similar to y, then $(\hat{y} - y)^2/2 \approx 0$.

The mean squared error is also a valid cost function in binary classification problems. However, it has some undesirable properties that can make training very difficult in this setting [8, Ch. 6, p. 178]. Hence, a different cost function, called the *cross-entropy*, is commonly chosen. The cross-entropy loss is defined as

$$\mathcal{L}(\boldsymbol{x}, y, \boldsymbol{\theta}) = -y \ln \hat{y} - (1 - y)\ln(1 - \hat{y}). \tag{13}$$

We observe that, if $y = 1$, the loss simply becomes $-\ln \hat{y}$, which is large if $\hat{y}$ is close to 0, and small if $\hat{y}$ is close to 1. A similar analysis can be conducted for the case $y = 0$ and we can see that this cost function also satisfies the desired properties.

In case of multiclass classification, the cross-entropy becomes

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta}) = -\ln \hat{y}_i \tag{14}$$

if $\boldsymbol{y}$ represents the i^{th} class (i.e. the i^{th} position in the vector $\boldsymbol{y}$ is 1). Similar arguments as in the case of binary classification apply as to why this is a valid cost function.

While these functions might seem rather different, they can all be derived with the same principle, called *maximum likelihood estimation* (MLE) [8, Ch. 5, pp. 128-131]. This estimation comes from a probabilistic perspective and interprets the training data as samples drawn from an unknown probability distribution $P(\mathbb{Y} \mid \mathbb{X})$. From this perspective, the goal of training is to choose parameters $\boldsymbol{\theta}$ such that the probability distribution $P_{\text{model}}(\mathbb{Y} \mid \mathbb{X}; \boldsymbol{\theta})$ described by the model matches the true distribution as closely as possible. The MLE states that we should choose the parameters that maximize P_{model}, i.e. the optimal parameters $\hat{\boldsymbol{\theta}}$ are

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} P_{\text{model}}(\mathbb{Y} \mid \mathbb{X}; \boldsymbol{\theta}). \tag{15}$$

We obtain the mean squared error from the MLE if we regard the true probability distribution of the data as a Gaussian distribution. In the case of binary and multiclass classification, we regard the true probability distribution as a Bernoulli or Multinoulli distribution, respectively [8, Ch. 6, pp. 175-185].

B. Stochastic Gradient Descent

Minimizing cost functions is similar to any other minimization problem. A variety of algorithms exist to minimize functions, but stochastic gradient descent, an extension of gradient descent, is particularly dominant in neural network training.

As the name suggests, gradient descent makes use of the gradient of the cost function to iteratively improve the parameters to decrease the cost. In particular, we know from calculus that a small change $\Delta\boldsymbol{\theta}$ in $\boldsymbol{\theta}$ corresponds roughly to the change

$$\Delta J(\boldsymbol{\theta}) \approx \nabla J(\boldsymbol{\theta})^\top \Delta\boldsymbol{\theta} \tag{16}$$

in $J(\boldsymbol{\theta})$. It can be shown that the choice of $\Delta\boldsymbol{\theta}$ which decreases the cost the fastest and thus minimizes $\Delta J(\boldsymbol{\theta})$ is

$$\Delta\boldsymbol{\theta} = -\eta \nabla J(\boldsymbol{\theta}), \tag{17}$$

where η is the *learning rate* [8, Ch. 4, p. 82]. This means that we can improve a neural network by making small changes to the parameters in the negative direction of the gradient of the cost function.

The learning rate η controls the size of the update steps performed by gradient descent. It should neither be too small, nor too large: if it is small, the model only learns very slowly, and if it is large, we risk making updates that unintentionally increase the loss, since the gradient is only an approximation of the real cost function.

The problem with using gradient descent to train neural networks is that computing the gradient is linear in the number of training examples m. To see this, note that

$$\nabla J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \nabla \mathcal{L}(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta}). \tag{18}$$

Stochastic gradient descent resolves this dependency by computing an approximation of the gradient by averaging over only a subset of training examples with corresponding labels $\mathbb{B}$ called a *minibatch*. The gradient computation becomes

$$\nabla J(\boldsymbol{\theta}) = \frac{1}{|\mathbb{B}|} \sum_{(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{B}} \nabla \mathcal{L}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta}), \tag{19}$$

which is independent of m. The size $|\mathbb{B}|$ of the minibatches is not increased with the amount of training data.

Many variations of stochastic gradient descent exist. They usually make use of higher order derivatives or approximations thereof and dynamically adjust the learning rate [19].

C. The Back-propagation Algorithm

An efficient way of computing the gradient of the cost function is the back-propagation algorithm [20]. Beginning from the output layer, the algorithm computes the partial derivatives of all the weights and biases up to the first hidden layer. In the remainder of this section, we derive this procedure mathematically and show how it can be used in combination with stochastic gradient descent to train feedforward neural networks. We assume familiarity with basic multivariable calculus. The derivation we present is closely based on Ref. [18, Ch. 2].

The ultimate goal of the algorithm is to compute the partial derivatives $\partial J / \partial w_{ij}^{(l)}$ and $\partial J / \partial b_j^{(l)}$. To achieve this, we introduce the intermediate quantity

$$\delta_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}}, \tag{20}$$

which is a measure of the error in the j^{th} neuron of the l^{th} layer: if $\delta_j^{(l)}$ is large, changing the parameters of this neuron can decrease the cost considerably, and if it is small, the neuron is already near-optimal. The error is closely related to the quantities of interest: with the chain rule, we obtain

$$\frac{\partial J}{\partial b_j^{(l)}} = \frac{\partial J}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \frac{\partial \left(\sum_i w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)} \right)}{\partial b_j^{(l)}} = \delta_j^{(l)}, \tag{21}$$

and similarly,

$$\frac{\partial J}{\partial w_{ij}^{(l)}} = \frac{\partial J}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}. \tag{22}$$

We can arrange those derivatives in vectors and matrices matching the bias vectors and weight matrices in dimension. The bias derivative vector is simply given by $\boldsymbol{\delta}^{(l)}$ and the matrix of weight derivatives is $\boldsymbol{a}^{(l-1)} \boldsymbol{\delta}^{(l)\top}$.

All that remains is the calculation of $\boldsymbol{\delta}^{(l)}$ itself. We start with the output layer L:

$$\delta_j^{(L)} = \frac{\partial J}{\partial z_j^{(L)}} = \frac{\partial J}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial z_j^{(L)}} = \frac{\partial J}{\partial \hat{y}_j} f^{(L)\prime} \left(z_j^{(L)} \right), \tag{23}$$

which, in vector notation, becomes

$$\boldsymbol{\delta}^{(L)} = \nabla_{\hat{\boldsymbol{y}}} J \odot f^{(L)\prime}\left(\boldsymbol{z}^{(L)}\right). \tag{24}$$

Here, $\boldsymbol{x} \odot \boldsymbol{y}$ denotes the entry-wise product between $\boldsymbol{x}$ and $\boldsymbol{y}$.

Finally, the output error can be propagated back to earlier layers. Again applying the chain rule,

$$\delta_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}} = \sum_k \frac{\partial J}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k \delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}, \tag{25}$$

where the sum is over all neurons in the $(l+1)^{\text{th}}$ layer. Note that

$$\begin{aligned} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} &= \frac{\partial \left(\sum_j w_{jk}^{(l+1)} f^{(l)}\left(z_j^{(l)}\right) + b_k^{(l+1)}\right)}{\partial z_j^{(l)}} \\ &= w_{jk}^{(l+1)} f^{(l)\prime}\left(z_j^{(l)}\right). \end{aligned} \tag{26}$$

Substituting back, we obtain

$$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} w_{jk}^{(l+1)} f^{(l)\prime}\left(z_j^{(l)}\right), \tag{27}$$

which is easily expressed in matrix notation:

$$\boldsymbol{\delta}^{(l)} = \boldsymbol{W}^{(l+1)} \boldsymbol{\delta}^{(l+1)} \odot f^{(l)\prime}\left(\boldsymbol{z}^{(l)}\right). \tag{28}$$

These expressions allow for a very efficient computation of the gradient. First, we propagate the input vector of a particular training example forward through the network to obtain the output $\hat{\boldsymbol{y}}$. From this, we can calculate the cost $J(\boldsymbol{\theta})$ and the error $\boldsymbol{\delta}^{(L)}$ in the output layer. With Eq. (28), the error is then propagated backwards through the hidden layers, where we can easily compute the derivatives using Eq. (21) and (22).

Combining back-propagation with stochastic gradient descent yields the complete learning algorithm for feedforward neural networks, which is formalized in algorithm 1 [8, Ch. 6, pp. 205-206].

Algorithm 1 The learning algorithm for feedforward neural networks. The algorithm computes one stochastic gradient descent update, given a minibatch $\mathbb{B}$.

Require: A minibatch of training examples $\mathbb{B}$
Require: The model composed of weight matrices $\boldsymbol{W}^{(i)}$, bias vectors $\boldsymbol{b}^{(i)}$, and activation functions $f^{(i)}$
Require: The learning rate η
  for $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{B}$ **do**
    $\boldsymbol{a}^{(\boldsymbol{x},1)} = \boldsymbol{x}$
    for $l = 2, \ldots, L$ **do** ▷ forward propagation
      $\boldsymbol{z}^{(\boldsymbol{x},l)} = \boldsymbol{W}^{(l)\top} \boldsymbol{a}^{(\boldsymbol{x},l-1)} + \boldsymbol{b}^{(l)}$
      $\boldsymbol{a}^{(\boldsymbol{x},l)} = f^{(l)}\left(\boldsymbol{z}^{(\boldsymbol{x},l)}\right)$
    end for
    $\hat{\boldsymbol{y}} = \boldsymbol{a}^{(\boldsymbol{x},L)}$
    $\boldsymbol{\delta}^{(\boldsymbol{x},L)} = \nabla_{\hat{\boldsymbol{y}}} J \odot f^{(L)\prime}\left(\boldsymbol{z}^{(\boldsymbol{x},L)}\right)$
    for $l = L-1, \ldots, 2$ **do** ▷ back-propagation
      $\boldsymbol{\delta}^{(\boldsymbol{x},l)} = \boldsymbol{W}^{(l+1)} \boldsymbol{\delta}^{(\boldsymbol{x},l+1)} \odot f^{(l)\prime}\left(\boldsymbol{z}^{(\boldsymbol{x},l)}\right)$
    end for
  end for
  for $l = L, \ldots, 2$ **do** ▷ gradient descent
    $\boldsymbol{W}^{(l)} = \boldsymbol{W}^{(l)} - \frac{\eta}{|\mathbb{B}|} \sum_{(\boldsymbol{x},\boldsymbol{y}) \in \mathbb{B}} \boldsymbol{a}^{(\boldsymbol{x},l-1)} \boldsymbol{\delta}^{(\boldsymbol{x},l)\top}$
    $\boldsymbol{b}^{(l)} = \boldsymbol{b}^{(l)} - \frac{\eta}{|\mathbb{B}|} \sum_{(\boldsymbol{x},\boldsymbol{y}) \in \mathbb{B}} \boldsymbol{\delta}^{(\boldsymbol{x},l)}$
  end for

V. Extensions

The basic feedforward neural networks we have discussed so far are rarely used in practice. However, they form the foundation for a variety of more sophisticated models, which we briefly examine in this section.

In computer vision, the most common type of neural networks are *convolutional neural networks* [21]. These models have been explicitly designed to exploit the spatial structure of images. For example, they can easily detect the same feature in different parts of an image. Convolutional neural networks also require far less parameters than traditional feedforward neural networks and are thus much more efficient to train and evaluate.

Recurrent neural networks [20] are neural networks that can process sequential data such as natural language and speech. In contrast to feedforward networks, they allow feedback connections from a layer into previous layers. Therefore, the output at a particular step t in a sequence $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}$ depends not only on the input $\boldsymbol{x}^{(t)}$, but also on all the previous inputs $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t-1)}$. For example, if a recurrent neural network predicts the meaning of a word in a sentence, it can take into account the previous words in the sentence.

While convolutional and recurrent neural networks are the most common extensions of feedforward neural networks, many other specialized networks exist. Neural networks have been adapted to almost every task in machine learning and continue to produce excellent results [8, Ch. 5, pp. 96-100].

VI. Conclusion

We have discussed neural networks, a biologically inspired computing model that dominates modern machine learning. Neural networks are complex networks of simple units which are arranged in a layered architecture. By propagating an input through its layers, a neural network can make sophisticated predictions, such as which class the input is most likely to belong to.

Feedforward neural networks can be represented as a chain of matrix operations and applications of nonlinear functions. The parameters of the model are chosen during training, where a cost function that measures the performance of the network is minimized. The most common algorithm to achieve this is stochastic gradient descent, which requires the computation of the gradient of the cost function. This is efficiently handled by the back-propagation algorithm.

We believe that, with ever-increasing datasets and computational power, as well as the development of more complex and refined models, neural networks will continue to have a great impact on technology and will power more and more intelligent systems in the future.

REFERENCES

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015. [Online]. Available: https://doi.org/10.1007/s11263-015-0816-y

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*. IEEE Computer Society, 2015, pp. 1026–1034. [Online]. Available: https://doi.org/10.1109/ICCV.2015.123

[3] A. Severyn and A. Moschitti, "Twitter sentiment analysis with deep convolutional neural networks," in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval, Santiago, Chile, August 9-13, 2015*, R. A. Baeza-Yates, M. Lalmas, A. Moffat, and B. A. Ribeiro-Neto, Eds. ACM, 2015, pp. 959–962. [Online]. Available: http://doi.acm.org/10.1145/2766462.2767830

[4] A. Mohamed, G. E. Dahl, and G. E. Hinton, "Acoustic modeling using deep belief networks," *IEEE Trans. Audio, Speech & Language Processing*, vol. 20, no. 1, pp. 14–22, 2012. [Online]. Available: https://doi.org/10.1109/TASL.2011.2109382

[5] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image style transfer using convolutional neural networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 2414–2423. [Online]. Available: https://doi.org/10.1109/CVPR.2016.265

[6] A. Karpathy and L. Fei-Fei, "Deep visual-semantic alignments for generating image descriptions," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 4, pp. 664–676, 2017. [Online]. Available: https://doi.org/10.1109/TPAMI.2016.2598339

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://doi.org/10.1038/nature14236

[8] I. J. Goodfellow, Y. Bengio, and A. C. Courville, *Deep Learning*, ser. Adaptive computation and machine learning. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org/

[9] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, 1943. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-51249194645&doi=10.1007%2fBF02478259&partnerID=40&md5=edb67afceee33d22eaabbf1f8c1dca90

[10] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-11144273669&doi=10.1037%2fh0042519&partnerID=40&md5=f6ad02750f121e6d5d33566003d5ac8b

[11] K. P. Murphy, *Machine learning - a probabilistic perspective*, ser. Adaptive computation and machine learning series. MIT Press, 2012.

[12] M. Minsky and S. Papert, *Perceptrons - an introduction to computational geometry*. MIT Press, 1987.

[13] K. Hornik, M. B. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989. [Online]. Available: https://doi.org/10.1016/0893-6080(89)90020-8

[14] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *MCSS*, vol. 2, no. 4, pp. 303–314, 1989. [Online]. Available: https://doi.org/10.1007/BF02551274

[15] M. D. Zeiler and R. Fergus., "Visualizing and understanding convolutional networks," in *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. J. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds., vol. 8689. Springer, 2014, pp. 818–833. [Online]. Available: https://doi.org/10.1007/978-3-319-10590-1_53

[16] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, ser. JMLR Proceedings, G. J. Gordon, D. B. Dunson, and M. Dudík, Eds., vol. 15. JMLR.org, 2011, pp. 315–323. [Online]. Available: http://www.jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 1106–1114. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

[18] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/

[19] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[20] D. Rumelhart, G. Hinton, and R. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0022471098&doi=10.1038%2f323533a0&partnerID=40&md5=2e1eae82a89c7503560b15a74e4c698d

[21] Y. LeCun, "Generalization and network design strategies," University of Toronto, Tech. Rep. CRG-TR-89-4, 1989.