Mathematical Engineering of Deep Learning

Book Draft

Benoit Liquet, Sarat Moka and Yoni Nazarathy

February 28, 2024

Contents

5 Feedforward Deep Networks - DRAFT

We now enter the heart of deep learning models by presenting key concepts for *feedforward deep neural networks* also known as *general fully connected neural networks*. These models are a powerful extension of shallow neural networks presented previously and are the central entities of deep learning. The key mathematical objects used for such networks are highlighted in this chapter. These include layers with neurons, activation function alternatives, and the backpropagation algorithm for gradient evaluation. A universal approximation theorem is stated and further demonstrations highlight the benefit of exploiting deep neural networks for machine learning tasks. Key steps for training a deep neural network are presented. These include weight initialization and batch normalization. We then focus on key strategies for handling overfitting. These are dropout and addition of regularization terms.

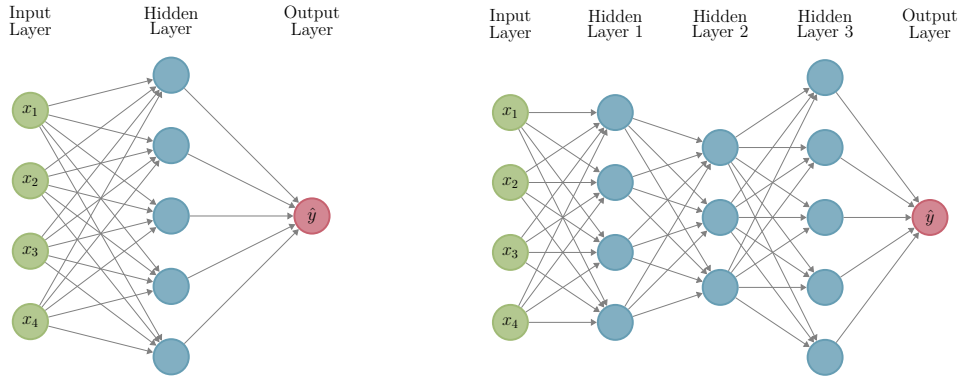
In Section 5.1 we present details for the general fully connected architecture. In Section 5.2 we explore the expressive power of neural networks to get a feel for why the general fully connected architecture is a very useful machine learning model. Towards that end, we explore a few theoretical underpinnings for the power of deep learning. In Section 5.3 we introduce activation functions beyond the sigmoid and softmax functions which were already presented in Chapter 3. In Section 5.4 we study the backpropagation algorithm used for gradient evaluation. This algorithm which is a form of backward mode automatic differentiation, is at the core of training deep learning models. In Section 5.5 we study various methods for weight initialization. In Section 5.6 we introduce the idea of batch-normalzation. In Section 5.7 we explore methods for mitigating overfitting. These include the addition of regularization terms and the concept of dropout.

5.1 The General Fully Connected Architecture

We refer to the generic deep learning model as *the general fully connected architecture* and also mention that such a model can be called a *fully connected network*, a *feedforward network*, or a *dense network* where each of these terms may also be augmented with the phrases “deep”, “neural”, and “general”. Another name for such a model is a *multi-layer perceptron* (MLP) or a *multi-layer dense network*.

This architecture involves multiple layers, each with non-linear transformations, and is an extension to the shallow neural networks presented in Chapter 3. Schematic illustrations of this architecture are presented in Figure 5.1. In that figure, each circle may be called a *neuron* or *unit* and each vertical set of neurons is a *layer*. A network with a single hidden layer is in (a) and a more general multi-layer network is in (b). Compare these illustrations with Figure 3.3 which has no hidden layers and has a single output (logistic regression), or Figure 3.6 which also has no hidden layers and has multiple outputs (multinomial regression).

Returning to Figure 5.1, the left most layer is called the *input layer* and it has the input features vector x. Each element of this layer is sometimes called an *input neuron* even though



(a) **(b)**

Figure 5.1: Fully connected feedforward neural networks. (a) A network with a single hidden layer. (b) A deep neural network with multiple hidden layers.

it does not involve any computation. The right most or *output layer* contains the output neurons (just one in this example). The layers in the middle are called *hidden layers*, since the neurons in these layers are neither inputs nor outputs. When using the network in production for prediction (regression or classification), we do not directly observe what goes on in the hidden layers, but rather observe network outputs resulting from inputs. However in certain cases, the values of neurons in hidden layers are also called "features" or more precisely, *extracted features* (also known as *computed features* or *derived features*) since they may encode some intermediate summary of the input data.

The design of the input and output layers in a network is often straightforward. There are as many neurons in the input layer as the number of features. As for the output layer, the number of output neurons depends on the application. In certain cases the output can be a scalar, determining either a probability in binary classification, or a response in a regression problem. In other cases, the output may be a vector, such as for example in multi-class classification where there will typically be as many output neurons as the number of possible labels (classes) and the output is a probability vector. In contrast to the input and output layers, when selecting the number and size of the hidden layers, there is much room for variation of model choice.

For example, assume we wish to create a model for classification of a type of plant based on $p = 120$ measured indicators (features). Assume further that our classifier supports 30 different types of plants. Then in this case we may use a model where the input layer has 120 input neurons and the output layer has 30 output neurons which may be interpreted as probabilities, similar to the multinomial regression model of Chapter 3. With this, there still remains a choice of how many hidden layers to use, and how many neurons to use for each of those hidden layers. These choices determine the number of trained parameters, the model expressivity, and the ease/difficulty of training the model.

Finally note that in our terminology, when counting layers in a multi-layer network, we count hidden layers as well as the output layer, but we do not count an input layer. Hence with our terminology, Figure 5.1 (a) has 2 layers, and (b) has 4 layers.

A Model Based on Function Composition

The goal of a feedforward network is to approximate some function $f^* : \mathbb{R}^p \longrightarrow \mathbb{R}^q$. A feedforward network model defines a mapping $f_\theta : \mathbb{R}^p \longrightarrow \mathbb{R}^q$ and learns the value of the parameters θ that ideally result in

$$f_\theta(x) \approx f^*(x).$$

The function f_θ is recursively composed via a chain of functions:

$$f_\theta(x) = f^{[L]}_{\theta^{[L]}}(f^{[L-1]}_{\theta^{[L-1]}}(\ldots(f^{[1]}_{\theta^{[1]}}(x))\ldots)), \tag{5.1}$$

where $f^{[\ell]}_{\theta^{[\ell]}} : \mathbb{R}^{N_{\ell-1}} \longrightarrow \mathbb{R}^{N_\ell}$ is associated with the ℓ-th layer which depends on parameters $\theta^{[\ell]} \in \Theta^{[\ell]}$, where $\Theta^{[\ell]}$ is the parameter space for the ℓ-th layer. The *depth* of the network is L. We have that $N_0 = p$ (the number of features) and $N_L = q$ (the number of output variables). Note that in case of networks used for classification we typically have $q = K$, the number of classes. The number of neurons in the network is $\sum_{\ell=1}^{L} N_\ell$.

Affine Transformations Followed by Activations

In deep learning, the function $f^{[\ell]}_{\theta^{[\ell]}}$ is generally defined by an affine transformation followed by an activation function. Activation functions are the means of introducing non-linearity into the model. The output of layer ℓ is represented by the vector $a^{[\ell]}$ and the intermediate result of the affine transformation is represented by the vector $z^{[\ell]}$ (see also Figure 3.3 in Chapter 3). We typically denote the output of the model via $\hat{y}$ and hence,

$$\hat{y} = a^{[L]} = f_\theta(x).$$

The action of $f^{[\ell]}_{\theta^{[\ell]}}$ can be schematically represented as follows,

$$a^{[\ell-1]} \xrightarrow[\text{Transformation}]{\text{Affine}} z^{[\ell]} := W^{[\ell]}a^{[\ell-1]} + b^{[\ell]} \xrightarrow{\text{Activation}} a^{[\ell]} := S^{[\ell]}(z^{[\ell]}), \tag{5.2}$$

$$f^{[\ell]}_{\theta^{[\ell]}}$$

where $a^{[0]} = x$. Hence the parameters of the ℓ-th layer, $\theta^{[\ell]}$, are given by the $N_\ell \times N_{\ell-1}$ *weight matrix* $W^{[\ell]} = \left(w^{[\ell]}_{i,j}\right)$ and the N_ℓ dimensional *bias vector* $b^{[\ell]} = (b^{[\ell]}_i)$. Thus the parameter space of the layer is $\Theta^{[\ell]} = \mathbb{R}^{N_\ell \times N_{\ell-1}} \times \mathbb{R}^{N_\ell}$.

The activation function $S^{[\ell]} : \mathbb{R}^{N_\ell} \longrightarrow \mathbb{R}^{N_\ell}$ is a non-linear multivalued function. For $\ell = 1, \ldots, L-1$ it is generally of the form

$$S^{[\ell]}(z) = \left[\sigma^{[\ell]}(z_1) \ \ldots \ \sigma^{[\ell]}(z_{N_\ell})\right]^\top, \tag{5.3}$$

where $\sigma^{[\ell]} : \mathbb{R} \to \mathbb{R}$ is typically an activation function common to all hidden layers. For the output layer, $\ell = L$, it is often of a different form depending on the task at hand.

In the popular case of multi-class classification, a *softmax* function as (3.25) is used, or more specifically for binary classification, a *sigmoid* function as (3.4) is typically used; see Chapter 3 for background. Thus, in such a classification framework, the output of the network is a vector of probability values determining class membership. In order to get a class label prediction one can convert the predicted probability scores into a class label using a chosen *threshold*, or more simply *maximum a posteriori probability*, as described in (3.34).

The Forward Pass

The *forward pass* equation, (5.1), of a deep neural network can be expanded out as follows,

$$\begin{aligned}
&\text{Layer 1} \left\{ \begin{array}{rcl} z^{[1]} &=& W^{[1]} \overbrace{x}^{\text{Input}} + b^{[1]} \\ a^{[1]} &=& S^{[1]}(z^{[1]}) \end{array} \right. \\
&\text{Layer 2} \left\{ \begin{array}{rcl} z^{[2]} &=& W^{[2]} a^{[1]} + b^{[2]} \\ a^{[2]} &=& S^{[2]}(z^{[2]}) \end{array} \right. \\
&\quad \vdots \\
&\text{Layer L} \left\{ \underbrace{\hat{y}}_{\text{output}} = \begin{array}{rcl} z^{[L]} &=& W^{[L]} a^{[L-1]} + b^{[L]} \\ a^{[L]} &=& S^{[L]}(z^{[L]}). \end{array} \right.
\end{aligned} \tag{5.4}$$

Thus, for a given input x, when computing $f_\theta(x)$, we sequentially execute the affine transformations and activation functions from layer 1 to layer L. The computational cost of such a forward pass is at an order of the total number of weights. To see this, observe that at the ℓ-th layer, the cost to compute $z^{[\ell]} = W^{[\ell]} a^{[\ell-1]} + b^{[\ell]}$ and $a^{[\ell]} = S^{[\ell]}(z^{[\ell]})$ are of the order $N_\ell \times N_{\ell-1} + N_\ell$ and N_ℓ, respectively. Hence, the total computational cost is of the order $\sum_{\ell=1}^{L} N_\ell(N_{\ell-1} + 2) \approx \sum_{\ell=1}^{L} N_\ell N_{\ell-1}$, which is the total number of weights.

An Example with Concrete Dimensions

As an illustration, let us return to the networks depicted in Figure 5.1. Consider the network in (a) with one-hidden layer. Here $N_0 = p = 4$, $N_1 = 5$, and $N_2 = q = 1$. Hence the dimension of $W^{[1]}$ is 5×4, the dimension of $b^{[1]}$ is 5, the dimension of $W^{[2]}$ is 1×5, and the dimension

of $b^{[2]}$ is 1. Putting these elements together we obtain,

$$f_\theta(x) = \overbrace{S^{[2]}(W^{[2]}\underbrace{S^{[1]}(\overbrace{W^{[1]}x + b^{[1]}}^{z^{[1]}})}_{a^{[1]}} + b^{[2]})}^{z^{[2]}}, \tag{5.5}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{a^{[2]}}$$

where the number of parameters in θ is $5 \times 4 + 5 + 1 \times 5 + 1 = 31$. Similarly, the deeper network on the right of Figure 5.1 is represented via,

$$f_\theta(x) = S^{[4]}(W^{[4]}S^{[3]}(W^{[3]}S^{[2]}(W^{[2]}S^{[1]}(W^{[1]}x + b^{[1]}) + b^{[2]}) + b^{[3]}) + b^{[4]}). \tag{5.6}$$

We may work out that the number of parameters is,

$$\underbrace{4 \times 4 + 4}_{\text{Hidden layer 1}} + \underbrace{3 \times 4 + 3}_{\text{Hidden layer 2}} + \underbrace{5 \times 3 + 5}_{\text{Hidden layer 3}} + \underbrace{1 \times 5 + 1}_{\text{Output layer}} = 61.$$

The Scalar Based View of the Model

It is also instructive to consider the scalar view of the system. The i-th neuron of layer ℓ, with $i = 1, \ldots, N_\ell$, is typically composed of both $z_i^{[\ell]}$ and $a_i^{[\ell]}$. The transition from layer $\ell - 1$ to layer ℓ takes the output of layer $\ell - 1$, an $N_{\ell-1}$ dimensional vector, and operates on it as follows,

$$\underset{\text{Transformation}}{\text{Affine}}: \begin{cases} z_1^{[\ell]} &= w_{(1)}^{[\ell]\top} a^{[\ell-1]} + b_1^{[\ell]} \\ z_2^{[\ell]} &= w_{(2)}^{[\ell]\top} a^{[\ell-1]} + b_2^{[\ell]} \\ &\vdots \\ z_{N_\ell}^{[\ell]} &= w_{(N_\ell)}^{[\ell]\top} a^{[\ell-1]} + b_{N_\ell}^{[\ell]} \end{cases} \Rightarrow \underset{\text{Step}}{\text{Activation}}: \begin{cases} a_1^{[\ell]} &= \sigma\left(z_1^{[\ell]}\right) \\ a_2^{[\ell]} &= \sigma\left(z_2^{[\ell]}\right) \\ &\vdots \\ a_{N_\ell}^{[\ell]} &= \sigma\left(z_{N_\ell}^{[\ell]}\right) \end{cases}, \tag{5.7}$$

where,

$$w_{(j)}^{[\ell]\top} = \begin{bmatrix} w_{j,1}^{[\ell]} & \ldots & w_{j,N_{\ell-1}}^{[\ell]} \end{bmatrix}, \qquad \text{for} \qquad j = 1, \ldots, N_\ell,$$

is the j-th row of the weight matrix $W^{[\ell]}$, and $b_j^{[\ell]}$ is the j-th element of the bias vector $b^{[\ell]}$. Hence the parameters associated with neuron j in layer ℓ, are $w_{(j)}^{[\ell]\top}$ and $b_j^{[\ell]}$.

Vectorizing Across Multiple Samples

So far we have defined our neural network using only one input feature vector x to generate a prediction $\hat{y}$. Namely,

$$x \longrightarrow a^{[L]} = \hat{y}.$$

Let us now consider mini-batches as introduced in Section 4.2. Here we use n_b training samples $x^{(1)}, \ldots, x^{(n_b)}$ where n_b is the size of the mini-batch and $x^{(i)} \in \mathbb{R}^p$. We can use the

feedforward pass equation to get for each training sample $x^{(i)}$, a prediction

$$x^{(i)} \longrightarrow a^{[L](i)} = \hat{y}^{(i)} \quad i = 1, \ldots, n_b.$$

One can clearly iterate using a loop for getting all the predictions. However, a matrix representation is often useful for describing the prediction of the whole mini-batch together. This is particularly useful in GPU implementations when the mini-batch size, n_b, is appropriately chosen to fit GPU memory. Let us first define the matrix $X^\top$ in which every column is a feature vector for one training sample:

$$X^\top = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \ldots & x^{(n_b)} \\ | & | & & | \end{bmatrix}. \tag{5.8}$$

Then, for each ℓ, we define the matrix $Z^{[\ell]}$ with columns $z^{[\ell](1)}, \ldots, z^{[\ell](n_b)}$ as,

$$Z^{[\ell]} = \begin{bmatrix} | & | & & | \\ z^{[\ell](1)} & z^{[\ell](2)} & \ldots & z^{[\ell](n_b)} \\ | & | & & | \end{bmatrix}.$$

The activation matrix $A^{[\ell]}$ for each layer ℓ is defined similarly via,

$$A^{[\ell]} = \begin{bmatrix} | & | & & | \\ a^{[\ell](1)} & a^{[\ell](2)} & \ldots & a^{[\ell](n_b)} \\ | & | & & | \end{bmatrix},$$

where for example the element in the first row and in the second column of a matrix $A^{[\ell]}$ is an activation of the first hidden unit from the layer ℓ and the second training example. Based on this matrix and the forward pass representation (5.4), we get,

$$\begin{cases} Z^{[1]} = & W^{[1]}X^\top + B^{[1]} \\ A^{[1]} = & \sigma^{[1]}(Z^{[1]}) \\ Z^{[2]} = & W^{[2]}A^{[1]} + B^{[2]} \\ A^{[2]} = & \sigma^{[2]}(Z^{[2]}) \\ \vdots & \\ Z^{[L]} = & W^{[L]}A^{[L-1]} + B^{[L]} \\ \underbrace{A^{[L]}}_{[\hat{y}^{(1)}, \ldots, \hat{y}^{(n_b)}]} = & S^{[L]}(Z^{[L]}) \end{cases} \tag{5.9}$$

where the scalar activation functions, as in (5.3), $\sigma^{[\ell]}(\cdot)$ are applied independently to each element of the matrix $Z^{[\ell]}$ for $\ell = 1, \ldots, L-1$, while $S^{[L]}(\cdot)$ is applied independently on each column of $Z^{[L]}$. For this representation, the matrix $B^{[\ell]}$ for each layer ℓ is based on the bias vector $b^{[\ell]}$ and is constructed via,

$$B^{[\ell]} = \begin{bmatrix} | & | & & | \\ b^{[\ell]} & b^{[\ell]} & \ldots & b^{[\ell]} \\ | & | & & | \end{bmatrix}. \tag{5.10}$$

An Overview of Model Training

Training of feedforward deep neural networks follows the same iterative optimization based learning paradigm presented in Section 2.4 of Chapter 2. This paradigm was also applied to simple neural networks in Chapter 3. Specifically a standardized training dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$ is used to find network parameters θ (weights and biases) that minimize a loss function $C(\theta\,;\mathcal{D})$. As illustrated in previous chapters, common loss functions include the cross entropy loss (binary or categorical) in the context of classification or the square loss function in the context of regression.

In practice, gradient based optimization generalizing the gradient descent of Algorithm 2.1 is the typical technique of training. Further, as discussed in detail in Chapter 4, a very common variant is ADAM presented in Algorithm 4.2. In any case, such learning algorithms require the evaluation of gradients, e.g., Step 3 of Algorithm 2.1 or Step 4 of Algorithm 4.2. For deep learning models, such gradient evaluation is carried out via the backpropagation algorithm which we study in detail in Section 5.4 below.

Another important aspect of iterative optimization involves parameter intialization. This is Step 1 of Algorithm 2.1 or Step 2 of Algorithm 4.2. In the context of deep learning this is called *weight initialization*, a topic we discuss in Section 5.5.

Finally, other aspects of training deep neural networks include batch normalization presented in Section 5.6 and dropout presented in Section 5.7, as well as other methods of regularization also presented in that section.

5.2 The Expressive Power of Neural Networks

Deep neural network models are extremely expressive and versatile. Research about their strength dates all the way back to the early days of artificial intelligence and continues in current times. We now explore the expressivity of such models where we simply touch the tip of the iceberg, attempting to illustrate why the model is sensible and versatile.

Neural networks are known for being able to approximate arbitrarily complex functions. Our exposition in this section aims to illustrate this while also presenting intuition about the benefits of deep models. We begin with a simple constructive example for scalar real valued functions, and we then progress to present an overview key results and intuition which highlight why these models are so powerful.

Simple Function Approximation

We now see one possible way to approximate scalar functions via a neural network. Say we have a function $f^* : [\underline{l}, \bar{l}] \to \mathbb{R}$ and we are given the values of the function at $v_i = f^*(u_i)$ for $u_1, \ldots, u_r$ with $\underline{l} \leq u_1 < u_2 < \ldots < u_r \leq \bar{l}$.

See for example Figure 5.2 focusing on this arbitrary example,

$$f^*(x) = \cos\left(2\cos\left(x^2\right) + \frac{1}{4}(x-1)^2\right) + \frac{x}{2} + 1, \quad \text{for} \quad x \in [0,4]. \tag{5.11}$$

Approximation functions obtained for $f^*(\cdot)$ of (5.11) are also illustrated in Figure 5.2 where the case of $r = 10$ is in (b) and the case of $r = 30$ is in (c). Our approximation of $f^*(\cdot)$ uses a feedforward neural network $f_\theta(\cdot)$ as illustrated in (a). In this case we apply a construction of a piecewise continuous affine function $f_\theta : \mathbb{R} \to \mathbb{R}$ with breakpoints at $u_1, \ldots, u_r$ and $f_\theta(u_i) = f^*(u_i)$ for $i = 1, \ldots, r$. For completeness, we now present the details.

Our constructed network has one hidden layer ($L = 2$), and model dimensions $N_0 = 1$, $N_1 = r - 1$, and $N_2 = 1$. In this case, the approximation is obtained by setting the scalar activation functions as in (5.3) to be $\sigma^{[1]}(u) = \max(u, 0)$ and $\sigma^{[2]}(u) = u$, where the former is called the ReLU function, see also (5.17), and the latter is simply the identity function. We set the first $N_1 \times 1$ dimensional weight matrix, $W^{[1]}$, to have all entries 1; the first N_1 dimensional bias vector to have entries $b_i^{[1]} = -u_i$; the second $1 \times N_1$ dimensional weight matrix is set with entries $w_{1,i}^{[2]} = s_i - s_{i-1}$ for $i = 1, \ldots, r-1$ with $s_0 = 0$ and $s_i = \frac{v_{i+1} - v_i}{u_{i+1} - u_i}$; and finally the second scalar bias $b^{[2]}$ is set to be v_1.

In summary, the construction uses a linear combination of shifted ReLU functions (see also Figure 5.6) where each shifted ReLU (shifted by the bias $-u_i$) is constructed to match the interval $[u_i, u_{i+1}]$ with a slope s_i. As the construction moves from left to right, slopes are cancelled out via subtraction of the s_{i-1} term in $w_{1,i}^{[2]}$.

With such a construction we see that essentially any continuous real valued function on a bounded domain can be approximated via a neural network.[1] It is evident that as $r \to \infty$ the approximation becomes exact, and this can be made rigorous for any continuous target function $f^*(\cdot)$ on a bounded interval.

A General Approximation Result

The expressive power of feedforward neural networks generalizes to functions that have p inputs and q outputs. In fact, as the result below shows, similarly to the construction above, a single hidden layer ($L = 2$) and identity activations in the second layer suffices for this.

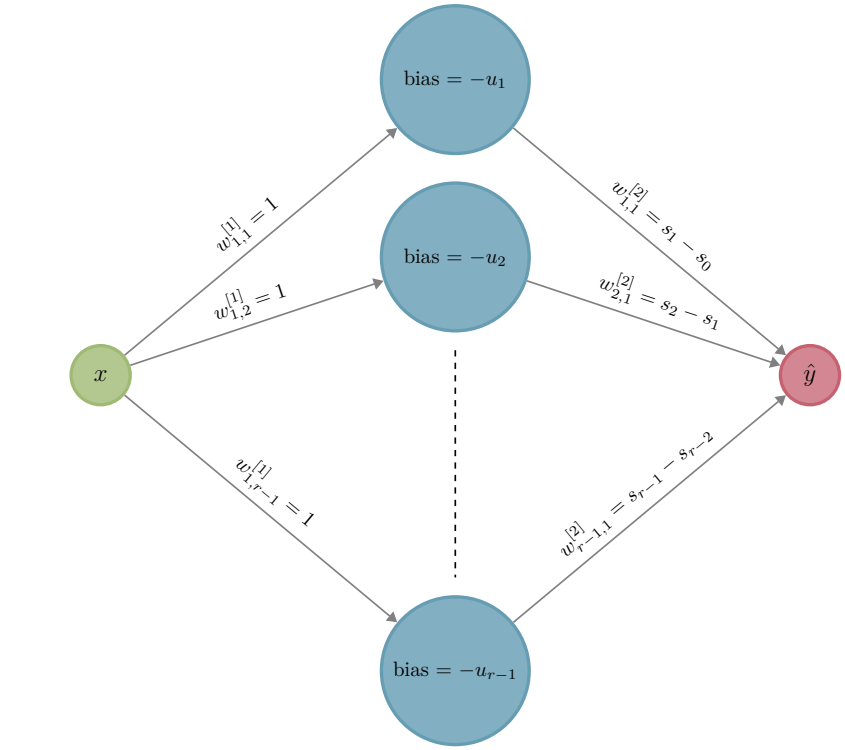
Theorem 5.1. *Consider a continuous function $f^* : \mathcal{K} \to \mathbb{R}^q$ where $\mathcal{K} \subseteq \mathbb{R}^p$ is a compact set. Then for any non-polynomial activation function $\sigma^{[1]}(\cdot)$ and any $\varepsilon > 0$, there exists an N_1 and parameters $W^{[1]} \in \mathbb{R}^{N_1 \times p}$, $b^{[1]} \in \mathbb{R}^{N_1}$, and $W^{[2]} \in \mathbb{R}^{q \times N_1}$, such that the function*

$$f_\theta(x) = W^{[2]} S^{[1]}(W^{[1]} x + b^{[1]}), \qquad \text{with} \qquad S^{[1]} \text{ as in (5.3)},$$

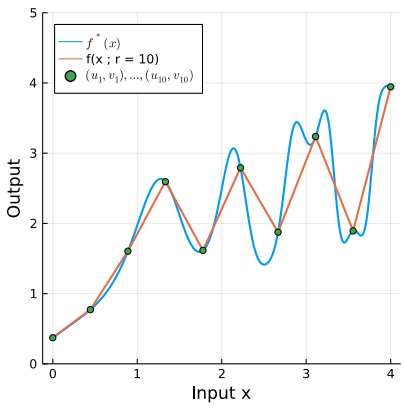
satisfies $||f_\theta(x) - f^*(x)|| < \varepsilon$ for all $x \in \mathcal{K}$.

Hence this theorem states that essentially all functions can be approximated to arbitrary precision dictated via ε. Practically for complicated functions $f^*(\cdot)$ and small ε one may need large N_1. Yet, the theorem states that it is always possible. A reference to a proof is provided in the notes and references section at the end of this chapter. The constructive example in Figure 5.2 above (for $p = 1$ and $q = 1$) may hint at the validity of the result. Note also that the tanh, sigmoid, and ReLU activation functions described in the next section are some of the valid activation functions for this result.

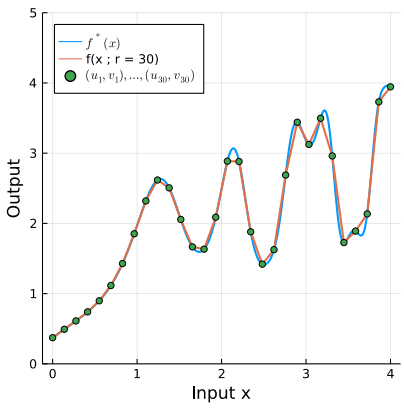
[1] This construction is one of many options one could use.



(a)



(b)

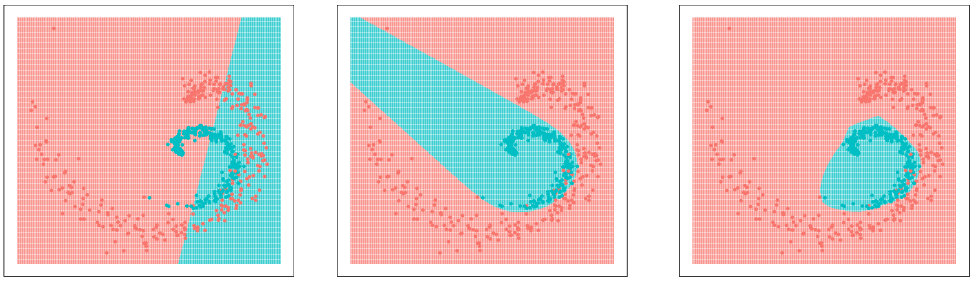


(c)

Figure 5.2: Approximation of an arbitrary real valued function on a bounded domain via a piecewise continuous function obtained by a single hidden layer feedforward model with ReLU activation functions. (a) A neural network with one hidden layer that constructs such an approximation. (b) Approximation based on $r = 10$ sampled points of the function. (c) Approximation based on $r = 30$ sampled points of the function.

The Strength of a Hidden Layer

As we saw in Chapter 3 shallow neural networks such as logistic regression or softmax regression can be used to create classifiers with linear decision boundaries. Further, as in Section 3.4, for cases where more general decision boundaries are needed, one may attempt to create additional transformed features while still using these basic models. However, the expressiveness of models with a single hidden layer (or more), as introduced in the current chapter, can yield a versatile alternative to the shallow networks of Chapter 3.



(a) **(b)** **(c)**

Figure 5.3: Binary classification example with $x \quad \mathbb{R}^2$, moving from a shallow neural network to a model with a hidden layer and then increasing the number of neurons. (a) Sigmoid model ($L = 1$). (b) One-hidden layer ($L = 2$) $N_1 = 4$ neurons. (c) One-hidden layer ($L = 2$) $N_1 = 10$ neurons.

Consider Figure 5.3 (a) for a classification task based on two inputs $x \in \mathbb{R}^2$ using logistic regression. By adding a single hidden layer with 4 neurons (*sigmoid* activation function is used for all units), we can move beyond the linear boundaries to obtain Figure 5.3 (b). Then, by increasing the number of neurons in the hidden layer from 4 to 10 units, the model further refines the decision boundary as in Figure 5.3 (c).

We thus see that obtaining non-linear decision boundries is possible not only via feature enginering as discussed in Section 3.4 and illustrated there in Figure 3.9, but can also be done via neural networks as hinted by Theorem 5.1 and illustrated here in Figure 5.3.

Stylized Functions via Simple Models

To further appreciate the expressive power of feedforward neural networks we also consider specific stylized functions. For example, historically, in the study of neural networks much effort has gone into characterizing the set of logical functions, sometimes called *gates*, that may be described via certain classes of models. In our exploration we focus on a different type of specific task, *multiplication of two inputs*, and demonstrate a simple constructive network that approximates this task. We thus construct what may be called a *multiplication gate*.

A simple construction of a single hidden layer network with $p = 2$ and $q = 1$, allows us to create a function $f_\theta(\cdot)$, parametrized by $\lambda > 0$, such that for input $x = (x_1, x_2)$, the function approximately implements multiplication of inputs,

$$f_\theta(x_1, x_2) \approx x_1 x_2. \tag{5.12}$$

Importantly, the approximation error vanishes as $\lambda \to 0$ and achieving (5.12) to arbitrary accuracy requires only $N_1 = 4$ neurons in the single hidden layer. Similarly to the model in Theorem 5.1 and to the simple function approximation example of Figure 5.2, the activation function of the output layer is the identity. There are no bias terms, and the weight matrices are,

$$W^{[1]} = \begin{bmatrix} \lambda & \lambda \\ -\lambda & -\lambda \\ \lambda & -\lambda \\ -\lambda & \lambda \end{bmatrix}, \qquad \text{and} \qquad W^{[2]} = \begin{bmatrix} \mu & \mu & -\mu & -\mu \end{bmatrix}, \tag{5.13}$$

with $\mu = \big(4\lambda^2 \ddot{\sigma}(0)\big)^{-1}$. Here $\ddot{\sigma}(0)$ represents the second derivative of the scalar activation function of the hidden layer ($\ell = 1$) at 0. Hence the model assumes $\sigma^{[1]}(\cdot)$ is twice differentiable (at 0) with a non-zero second derivative at zero. A schematic representation of $f_\theta(\cdot)$ is in Figure 5.4.

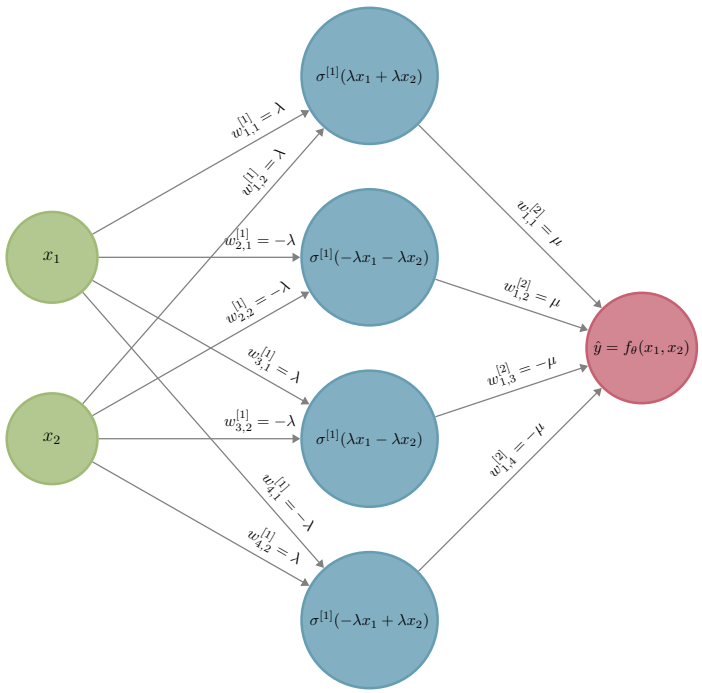


Figure 5.4: A simple neural network model with a single hidden layer composed of four neurons. This network approximates multiplication, or a multiplication gate, in the sense that $\hat{y} \approx x_1 x_2$.

We may verify that the model $f_\theta(\cdot)$ evaluates to,

$$f_\theta(x_1, x_2) = \frac{\sigma\big(\lambda(x_1 + x_2)\big) + \sigma\big(\lambda(-x_1 - x_2)\big) - \sigma\big(\lambda(x_1 - x_2)\big) - \sigma\big(\lambda(-x_1 + x_2)\big)}{4\lambda^2 \ddot{\sigma}(0)}, \tag{5.14}$$

where we use $\sigma(\cdot)$ to denote $\sigma^{[1]}(\cdot)$. We may now use a Taylor expansion (see Theorem A.1 in Appendix A) of $\sigma(\cdot)$ around the origin,

$$\sigma(u) = \sigma(0) + \dot{\sigma}(0)u + \ddot{\sigma}(0)\frac{u^2}{2} + O\left(u^3\right), \tag{5.15}$$

with $O(h^k)$ denoting a function such that $O(h^k)/h^k$ goes to a constant as $h \to 0$. We can now use (5.14) and (5.15) represent the model as,

$$f_\theta(x_1, x_2) = x_1 x_2 \left(1 + O\left(\lambda(x_1^2 + x_2^2)\right)\right).$$

Hence as $\lambda \to 0$ the desired goal (5.12) becomes exact. Note that this continuous multiplication gate is mostly a theoretical construct and for many popular activation functions the condition $\ddot{\sigma}(0) = 0$ does not hold. Nevertheless, this problem can be overcome by introducing biases to shift the origin and hence use a different input into the activation. Below we use this construction to further argue about the power of deep learning models.

Feature Focus with Neural Networks

We now contrast the usage of feedforward neural networks with the more classic practice of *feature engineering* where one would consider data and add additional features by transforming existing features. See also Section 3.4 where feature engineering is illustrated for simple neural networks. As an example we consider a case where there are originally p features $x_1, \ldots, x_p$ and we wish to construct $\tilde{p} = p(p+1)/2$ features based on all possible pairwise interactions[2] (multiplications) $x_i x_j$ for $i, j = 1, \ldots, p$. For instance if $p = 1,000$ then we arrive at $\tilde{p} \approx 500,000$. Clearly for non-small p we quickly arrive at huge number of transformed features $\tilde{p}$.

Let us contrast the usage of two alternatives. On the one hand consider a linear model acting on the transformed features $\tilde{x}$ where for simplicity we ignore the bias (intercept). On the other hand let us consider a neural network with a single hidden layer acting on the original features x. In the linear model, we have $\tilde{f}_\theta(\tilde{x}) = \tilde{w}^\top \tilde{x}$ where the learned weight vector $\tilde{w}$ has $\tilde{p}$ parameters. In the single hidden layer neural network, there are p inputs, $q = 1$ output, and N_1 units in the hidden layer. Thus the number of parameters is $N_1 \times p + N_1 + N_1 + 1$.

It is often the case that not all interactions (product features) are relevant. As an example let us consider that only a fraction α of the interactions are relevant. We now argue that the neural network model with N_1 hidden units sufficiently large, but not necessarily huge, can in principle capture these interactions. To do so, revisit the multiplication example presented above where 4 hidden units were needed to approximate an interaction (multiplication). While the construction using weight parameters as in (5.13) is artificial, the example hints at the fact that with $N_1 \approx 4\,\alpha\,p$ hidden units we may be able to capture the key interactions. In such a framework, the basic linear model still requires the full set of parameters. With this, compare the number of parameters,

$$\underbrace{\frac{1}{2}p(p+1)}_{\text{Linear Model}} \qquad \text{vs.} \qquad \underbrace{4\alpha p(p+2) + 1}_{\text{Neural Network}}.$$

[2] Typically in statistics interactions refer to terms such as $x_i x_j$ for $i \neq j$. However in this example we also allow for terms such as x_i^2.

Observe that p^2 is the dominant term in both models but for $\alpha < 1/8$ and large p, the neural network model has less parameters. Keep in mind that often α is very small while p may grow. Returning to the numerical case of $p = 1,000$, if $\alpha = 0.02$ (20 significant interactions) then the linear model has an order of $500,000$ parameters while the neural network only has order of $80,000$ parameters and is thus parisiminous in comparison to the linear model.

Improvement in Expressivity by an Increase of Depth

Despite the fact that Theorem 5.1 states that almost any function can be approximated using a neural network model with a single hidden layer, practice and research has shown that to gain high expressive power, this model might require a very large number of units (N_1 needs to be very large). Hence gaining significant expressive power may require a very large number of parameters. The power of deep learning then arises via repeated composition of non-linear activations functions via an increase of depth (an increase of L).

Note first that if the identity activation function is used in each hidden layer, then the network reduces to a shallow neural network,

$$f_\theta(x) = S^{[L]}(\tilde{W}x + \tilde{b}),$$

where,[3]

$$\tilde{W} = W^{[L]}W^{[L-1]} \cdot \ldots \cdot W^{[1]}, \qquad \text{and} \qquad \tilde{b} = \sum_{\ell=1}^{L} \Big(\prod_{\tilde{\ell}=\ell+1}^{L} W^{[\tilde{\ell}]} \Big) b^{[\ell]}.$$

In the case where the identity function is also used for the output layer, the model reduces to be a linear (affine) model. Thus, we have no gain by going deeper and adding multiple layers with identity activations. The expressivity of the neural network comes from the composition of non-linear activation functions. The repeated compositions of such functions has significant expressive power and can reduce the number of units needed in each layer in comparison to a network with a single hidden layer. A consequence is that the parameter space is reduced as well.

Let us consider an artificial example to demonstrate why exploiting the depth of the network is crucial for modelling complex relationships between input and output. We revisit the previous example involving models using interactions of order 2 (i.e., $x_i x_j$). Let us consider a higher complexity model by exploiting potential high-order interactions, namely products of r inputs, similarly to the discussion in Section 3.4.

Consider a fully connected network with p inputs, $q = 1$ output, and $L \geq 2$ layers with same size ($N^{[\ell]} = N$ in each layer). We re-use the idea appearing in (5.13) that with $N \approx 4\,\alpha\,p$ hidden units we may be able to capture the $\alpha\,p$ relevant interactions of order $r = 2$. Then by moving forward in the network, the subsequent addition of a layer with $N \approx 4\,\alpha\,p$ hidden units will capture interactions of order $r = 2^2$ and so on, until we capture interaction of order $r = 2^L$ at the output layer. Hence to achieve interactions of order r we may require $L \approx \log_2 r$, e.g., $L = \lceil \log_2 r \rceil$. Such network is depicted in Figure 5.5 where the number of

[3]Note that in the expression $\prod_{\tilde{\ell}=\ell+1}^{L} W^{[\tilde{\ell}]}$ we assume that the product multiplies the matrices in the left to right order $W^{[L]}W^{[L-1]}W^{[L-2]}\ldots$. Further, for $\ell = L$ the product is taken as the identity matrix.

parameters is,

$$\underbrace{N \times p + N}_{\text{First hidden layer}} + \underbrace{(L-2) \times (N^2 + N)}_{\text{Inner hidden layers}} + \underbrace{N + 1}_{\text{Output layer}} \approx LN^2.$$

For example, assume we wish to have a model for $p = 1,000$ features that supports about 20 meaningful interactions of order $r = 500$. Hence we can consider $\alpha = 0.02$. With a model not involving hidden layers (e.g., a linear model or a logistic model), as shown in Section 3.4, we cannot specialize for an order of 20 interactions and thus we require a full model of order $p^r/r! \approx 10^{365}$ parameters. In contrast, with a deep model we require about $L = 10 \approx \log_2 500$ layers and $N = 4\alpha p = 80$ neurons per layer, and thus the total number of parameters is at the order of $LN^2 = 57,600$. This deep construction which can capture a desired set of meaningful interactions is clearly more feasible and efficient than the shallow construction of astronomical size.

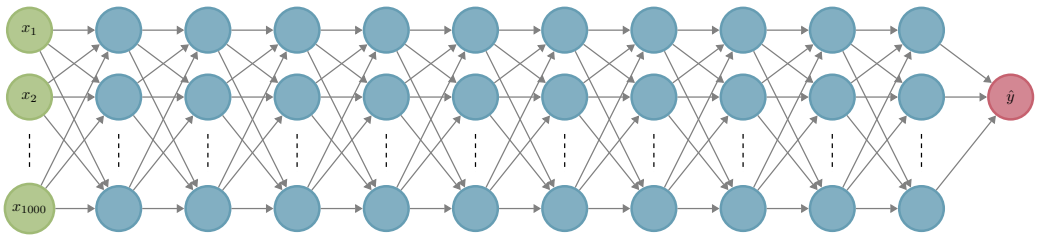


Figure 5.5: A deep learning model with $L = 10$ hidden layers may express many meaningful interactions for the inputs.

5.3 Activation Function Alternatives

As presented in (5.2), each layer ℓ of the network incorporates an activation function which is generally a non-linear transformation of $z^{[\ell]}$ to arrive at $a^{[\ell]}$. For most layers of the network, the activation function $S^{[\ell]}(\cdot)$ is composed of a sequence of identical scalar valued activation functions as in (5.3). That is, the ℓ-th layer incorporates a scalar activation function $\sigma^{[\ell]} : \mathbb{R} \to \mathbb{R}$ and it is applied to each of the N_ℓ coordinates of $z^{[\ell]}$ separately. In some models, all the scalar activation functions across all layers will be of the same form, while in other models different layers will sometimes incorporate different forms of scalar activation functions.

Scalar Activations and their Derivatives

When it comes to the choice of the scalar activation functions, there are different heuristic considerations that depend on model expressivity and learning ability. These considerations are often not based on theory, yet practice and experience over the years has shown that some scalar activation functions perform much better than others. In Section 5.5 we outline how such choices interface with optimization and the associate vanishing gradient problem is discussed in Section 5.5. Here we simply outline the key activation functions.

At the onset of the development of deep learning, in the late 1950's, the *step* scalar activation function was used. That is,

$$\sigma_{\text{Step}}(u) = \begin{cases} -1 & u < 0, \\ +1 & u \geq 0. \end{cases} \tag{5.16}$$

However, σ_{Step} is not used in modern neural network models, primarily because its derivative $\dot{\sigma}_{\text{Step}}(u) = 0$ for all $u \neq 0$. Indeed the derivative of activation functions is important since it is used in the backpropagation algorithm (see the next section) to compute the gradient of the loss function with respect to the model parameters.

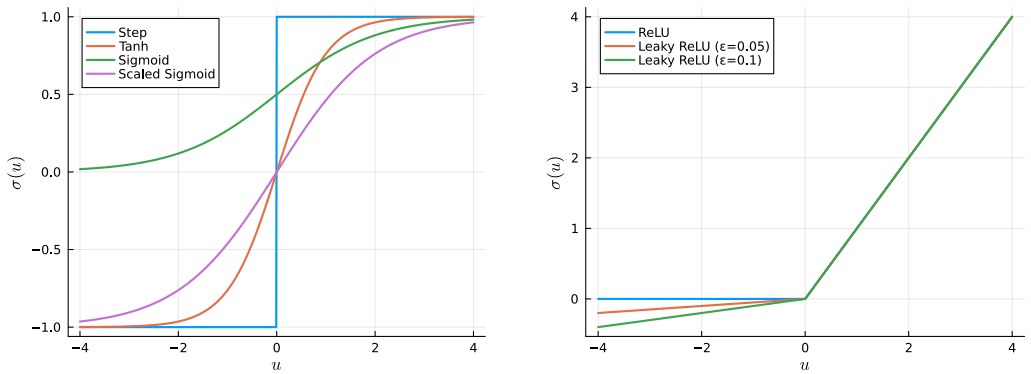
The *sigmoid* activation function (also known as the *logistic* function), used heavily in Chapter 3, see (3.4), is a much more popular choice. Note also that its derivative $\dot{\sigma}_{\text{Sig}}$ can be expressed in terms of the function σ_{Sig} itself,

$$\sigma_{\text{Sig}}(u) = \frac{e^u}{1+e^u} = \frac{1}{1+e^{-u}}, \qquad \text{with} \qquad \dot{\sigma}_{\text{Sig}}(u) = \sigma_{\text{Sig}}(u)\big(1 - \sigma_{\text{Sig}}(u)\big).$$

A similar popular function is *hyperbolic tangent*, denoted *tanh*,

$$\sigma_{\text{Tanh}}(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}, \qquad \text{with} \qquad \dot{\sigma}_{\text{Tanh}}(u) = 1 - \sigma_{\text{Tanh}}(u)^2.$$

Both σ_{Sig} and σ_{Tanh} share similar qualitative properties as σ_{Step}. They are non-decreasing and bounded. At $u \to \infty$ both functions converge to unity just like σ_{Step} and at $u \to -\infty$ the sigmoid function converges to 0 while the tanh function converges to -1 like σ_{Step}. This minor quantitative difference is not significant as it may generally be compensated by learned weights and biases or a shifting and scaling of the functions, yet in practice when the output $\hat{y}$ is a probability in $[0, 1]$ using σ_{Sig} is much more common. See Figure 5.6 for plots of several activation functions.



(a) **(b)**

Figure 5.6: Several common scalar activation functions. (a) The step and tanh function have a range of $(-1, 1)$ while the sigmoid function has a range of $(0, 1)$. We also plot a scaled sigmoid to the range $(-1, 1)$ so it can be compared to tanh. (b) The ReLU activation function and the leaky ReLU variant with different leaky ReLU parameters.

In earlier applications of deep learning, it was a matter of empirical research, practice, and heuristics to choose between models or layers that use σ_{Sig}, σ_{Tanh}, or similar forms. However in more recent years, a completely different type of scalar activation function became popular,

namely the *rectified linear unit* [4] or *ReLU*,

$$\sigma_{\text{ReLU}}(u) = \max(0, u) = \begin{cases} 0 & u < 0, \\ u & u \geq 0, \end{cases} \qquad \text{with,} \tag{5.17}$$

$$\dot{\sigma}_{\text{ReLU}}(u) = \mathbf{1}\{u \geq 0\} = \begin{cases} 0 & u < 0, \\ 1 & u \geq 0. \end{cases}$$

Note that while $\sigma_{\text{ReLU}}(u)$ is not differentiable at $u = 0$ we still arbitrarily define the derivative at $u = 0$ to be the same as the derivative for $u > 0$.

As we describe in Section 5.5, when parameters are initialized properly, the unboundedness of σ_{ReLU} often presents an advantage in training over bounded activation functions such as σ_{Sig} since it overcomes a training problem known as the vanishing gradient problem. However in certain cases it also introduces a problem called *dying ReLU* since the derivative is 0 for negative inputs. While in practice, this is often not considered a major problem, to handle dying ReLU, one may use a related activation function, *leaky ReLU*, parameterized by a fixed small $\varepsilon \geq 0$ (e.g., $\varepsilon = 0.01$) and defined via,

$$\sigma_{\text{LeakyReLU}}(u) = \max(0, u) + \min(0, \varepsilon u) = \begin{cases} \varepsilon u & u < 0, \\ u & u \geq 0, \end{cases} \qquad \text{with,}$$

$$\dot{\sigma}_{\text{LeakyReLU}}(u) = \mathbf{1}\{u \geq 0\} + \varepsilon \mathbf{1}\{u < 0\} = \begin{cases} \varepsilon & u < 0, \\ 1 & u \geq 0. \end{cases}$$

Observe that when $\varepsilon = 0$, this is just the ReLU activation function. Another variant called *PReLU* (parametric ReLU) considers the leaky ReLU parameter ε as a learned parameter. That is, the gradient based optimization for the parameters of the network also includes improvement steps of ε, incorporating it as part of the parameters for ℓ-th layer, $\theta^{[\ell]}$.

Note that feedforward models that only use piecewise affine scalar activation functions such as the step, ReLU, leaky ReLU, or PReLU have some mathematical appeal. When only such activations are used in a deep learning model, these functions imply that the complete model can be described as a *piecewise affine function* of the input. That is, the trained model implicitly partitions the input space $\mathbb{R}^p$ into polytopes (regions defined by intersections of half spaces), and for each polytope, when the input x is in that polytope, the response of the model is a fixed affine transformation of x. This property is simply a consequence of the fact that compositions of piecewise affine functions remains a piecewise affine function. There are no immediate practical applications for this property but it further hints at the expressive power of neural network models further to the description in Section 5.2.

Also note that many other forms of scalar activation functions have been introduced and experimented with. These include the *arctan*, *softsign*, *softplus*, *elu*, *selu*, and *swish* among others. Figure 5.6 presents the key activation functions σ_{Step}, σ_{Sig}, σ_{Tanh}, σ_{ReLU}, and $\sigma_{\text{LeakyReLU}}$ (with $\epsilon = 0.01$), along with a few of these alternatives.

[4] Note that source of the name is from electrical engineering where a rectifier is a device that converts alternating current to direct current.

Non-Scalar Activations and their Derivatives

Some layers also use non-scalar activation functions. That is, $S^{[\ell]} : \mathbb{R}^{N_\ell} \to \mathbb{R}^{N_\ell}$ is a vector to vector function that cannot be decomposed as in (5.3). The most common example of this is the softmax activation function, typically used for classification in the last layer $\ell = L$. We now denote $N_L = K$ since we often deal with classification of K classes as in Section 3.3. In such a case,

$$a^{[L]} = S_{\text{Softmax}}(z^{[L]}). \tag{5.18}$$

The $\mathbb{R}^K \to \mathbb{R}^K$ softmax activation function is defined as,

$$S_{\text{Softmax}}(z) = \frac{1}{\sum_{i=1}^K e^{z_i}} \begin{bmatrix} e^{z_1} & \cdots & e^{z_K} \end{bmatrix}^\top,$$

which was also defined in (3.25) of Section 3.3. Note that other examples of non-scalar activations include max pooling layers, introduced in Section 6.4 of the next chapter.

As will become evident in the next section, denoting the training loss by C, the gradient $\partial C / \partial z^{[L]}$ needs to be evaluated as part of the backpropagation algorithm. One approach for this evaluation is

$$\frac{\partial C}{\partial z^{[L]}} = \underbrace{\frac{\partial a^{[L]}}{\partial z^{[L]}}}_{N_L \times N_L} \underbrace{\frac{\partial C}{\partial a^{[L]}}}_{N_L \times 1},$$

where we use the multivariate chain rule (see also Appendix A) and keep in mind that $a^{[L]}$ is a vector, $z^{[L]}$ is a vector, and C is a scalar. With this approach, when the last layer is a softmax as (5.18), one may essentially need to evaluate $\partial a^{[L]} / \partial z^{[L]}$ which is the transpose of the Jacobian of $S_{\text{Softmax}}(\cdot)$. The elements of this Jacobian can be represented using the (scalar) quotient rule for differentiation, and are,

$$[J_{S_{\text{Softmax}}}(z)]_{ij} = \frac{\partial}{\partial z_j} \frac{e^{z_i}}{\sum_{k=1}^K e^{z_k}} = \begin{cases} [S_{\text{Softmax}}(z)]_i (1 - [S_{\text{Softmax}}(z)]_i), & i = j, \\ -[S_{\text{Softmax}}(z)]_i [S_{\text{Softmax}}(z)]_j, & i \neq j. \end{cases} \tag{5.19}$$

However, one rarely uses (5.19) since in the typical case where the loss function is the cross entropy loss (see also Section 3.3), we have a direct expression for $\partial C / \partial z^{[L]}$. To see, this suppose the label y equals k, an element from $\{1, \ldots, K\}$. In this case, as was shown in (3.29) from Section 3.3, the loss for a specific observation is,

$$C = -\log [S_{\text{Softmax}}(z^{[L]})]_k = \log \sum_{i=1}^K e^{z_i^{[L]}} - z_k^{[L]}.$$

Now to obtain $\partial C / \partial z^{[L]}$ we compute the derivative with respect to j for every $j = 1, \ldots, K$. This yields,

$$\frac{\partial C}{\partial z_j^{[L]}} = \frac{e^{z_j^{[L]}}}{\sum_{i=1}^K e^{z_i^{[L]}}} - \mathbf{1}\{j = k\} = [S_{\text{Softmax}}(z^{[L]})]_j - \mathbf{1}\{j = k\}.$$

Hence in this case the direct expression is,

$$\frac{\partial C}{\partial z^{[L]}} = S_{\text{Softmax}}(z) - e_k, \tag{5.20}$$

where e_k is the K-dimensional vector with 1 in the k-th position and 0 in other coordinates. Note that in practice when using deep learning frameworks, it is often recommended to use (5.20) directly in such a case.

5.4 The Backpropagation Algorithm

Now that we understand the model, we focus on gradient computation so as to facilitate learning using variants of gradient descent as covered in Chapter 4. A key algorithm is the *backpropagation algorithm* which implements backward mode automatic differentiation which is overviewed in Section 4.4 of Chapter 4. Now we build the related backpropagation algorithm of deep learning. We start with a general recursive model and then specialize to feedforward neural networks where the parameters are weights and biases.

Backpropagation for the General Recursive Model

It is instructive to first consider a general recursive feedforward model as appearing in (5.1). For such a model, the recursive step is of the form $a^{[\ell]} = f^{[\ell]}_{\theta^{[\ell]}}(a^{[\ell-1]})$. However, in this section it is convenient to use notation that treats the function $f^{[\ell]}$ separately as a function of $a^{[\ell-1]}$ and of the parameter $\theta^{[\ell]}$:

$$f^{[\ell]}(\cdot\,;\theta^{[\ell]}) : \mathbb{R}^{N_{\ell-1}} \longrightarrow \mathbb{R}^{N_\ell}, \qquad \text{and} \qquad f^{[\ell]}(a^{[\ell-1]}\,;\cdot) : \Theta^{[\ell]} \longrightarrow \mathbb{R}^{N_\ell}.$$

Using this notation, the recursive step is,

$$a^{[\ell]} = f^{[\ell]}(a^{[\ell-1]}\,;\theta^{[\ell]}), \qquad \text{for} \qquad \ell = 1, \ldots, L, \tag{5.21}$$

where $a^{[0]} = x$ and $\hat{y} = a^{[L]}$. Given a single data sample (x, y) we assume there is a loss function which depends on the given parameters θ, on the label value y, and on the output of the model, $a^{[L]}$. We denote this loss via $C(a^{[L]}, y\,;\theta)$.

Our purpose is to optimize the loss with respect to $\theta = (\theta^{[1]}, \ldots, \theta^{[L]})$. For this, we require the gradient with respect to θ and we denote its components via,

$$g^{[\ell]}_\theta := \frac{\partial C(a^{[L]}, y\,;\theta)}{\partial \theta^{[\ell]}}. \tag{5.22}$$

A key aspect of the automatic differentiation setup is that we are presented with computable expressions or code, for evaluation of certain derivatives. In our context these are the derivative of the loss C with respect to $a^{[L]}$, and the derivatives of $f^{[\ell]}(\cdot\,;\cdot)$ with respect to the input arguments (both layer input and parameters). These known derivatives are denoted via,

$$\dot{C}(u) := \frac{\partial C(u, y\,;\theta)}{\partial u}, \quad \dot{f}^{[\ell]}_a(u) := \frac{\partial f^{[\ell]}(u\,;\theta^{[\ell]})}{\partial u}, \quad \dot{f}^{[\ell]}_\theta(u) := \frac{\partial f^{[\ell]}(a^{[\ell-1]}\,;u)}{\partial u}. \tag{5.23}$$

Note that the shape of these expressions varies based on the domain and co-domain of $C(\cdot)$ and $f(\cdot\,;\cdot)$. The derivative $\dot{C}(u)$ is typically a gradient and thus vector valued with length N_L. The derivative $\dot{f}^{[\ell]}_a(u)$ is typically an $N_{\ell-1} \times N_\ell$ matrix obtained via a transpose of a Jacobian and thus matrix valued. This is because the input to the layer is $N_{\ell-1}$ dimensional and the output argument is N_ℓ dimensional. Finally, the derivative $\dot{f}^{[\ell]}_\theta(u)$ may take on

various shapes depending on the form of θ. See Appendix A for a review of matrix derivatives. On route to compute the desired gradients (5.22) we require intermediate gradients of the

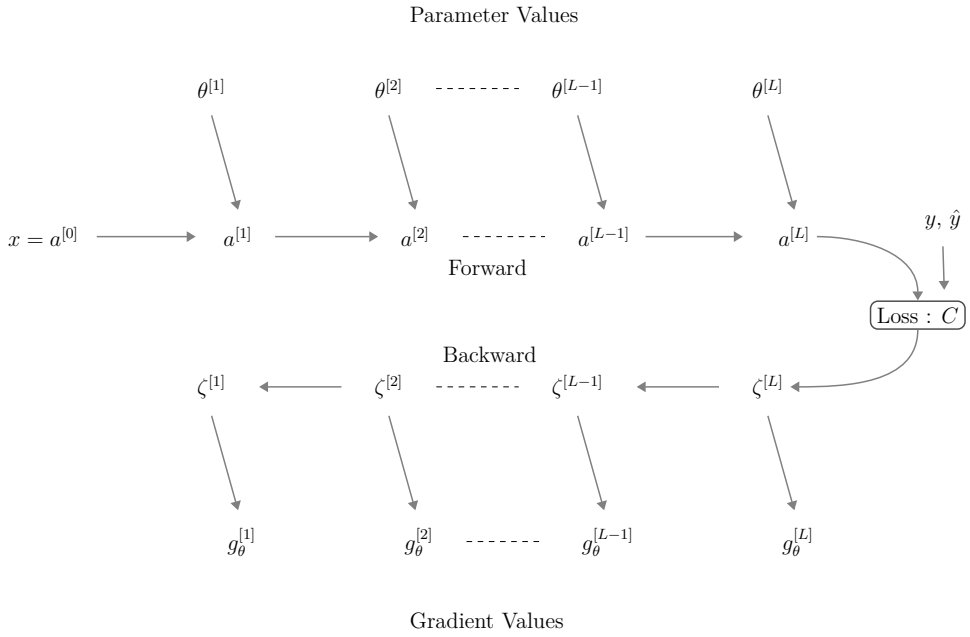


Figure 5.7: The variables and flow of information in the backpropagation algorithm for the general recursive model.

loss with respect to the activation values $a^{[1]}, \ldots, a^{[L]}$. Keeping in mind that $a^{[L]} = \hat{y}$ is a function of these activation values, the intermediate gradients[5] are denoted,

$$\zeta^{[\ell]} := \frac{\partial C(a^{[L]}, y\,;\,\theta)}{\partial a^{[\ell]}}, \qquad \ell = 1, \ldots, L. \tag{5.24}$$

Now based on the multivariate chain rule, the recursive step (5.21), and the definitions above, we observe,

$$\zeta^{[\ell]} = \frac{\partial a^{[\ell+1]}}{\partial a^{[\ell]}} \frac{\partial C}{\partial a^{[\ell+1]}} = \dot{f}_a^{[\ell+1]}(a^{[\ell]})\, \zeta^{[\ell+1]}, \qquad g_\theta^{[\ell]} = \frac{\partial a^{[\ell]}}{\partial \theta^{[\ell]}} \frac{\partial C}{\partial a^{[\ell]}} = \dot{f}_\theta^{[\ell]}(\theta^{[\ell]})\, \zeta^{[\ell]}. \tag{5.25}$$

Note that in the application of the chain rule above, we use the notation specified in Appendix A. Hence once the activation values $a^{[1]}, \ldots, a^{[L]}$ are populated via forward propagation of (5.21), backward computation can be carried out via,

$$\zeta^{[\ell]} = \begin{cases} \dot{C}(a^{[L]}), & \ell = L, \\ \dot{f}_a^{[\ell+1]}(a^{[\ell]})\, \zeta^{[\ell+1]}, & \ell = L-1, \ldots, 1, \end{cases} \tag{5.26}$$

and at each step the gradient can be obtained via $g_\theta^{[\ell]} = \dot{f}_\theta^{[\ell]}(\theta^{[\ell]})\, \zeta^{[\ell]}$.

[5] These are also called adjoints. See (4.30) in Chapter 4.

This process is summarized in Algorithm 5.1. See also Figure 5.7 for an illustration of the flow of information.

Algorithm 5.1: Backpropagation for the general recursive model

Input: Dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})\}$,
objective function $C(\cdot) = C(\cdot\,; \mathcal{D})$, and
parameter values $\theta = (\theta^{[1]}, \dots, \theta^{[L]})$

Output: gradients of the loss $g_\theta^{[1]}, \dots, g_\theta^{[L]}$

Compute $a^{[\ell]}$ for $\ell = 1, \dots, L$ using (5.21) (Forward pass)
Compute $\zeta^{[L]} = \dot{C}(a^{[L]})$
Compute $g_\theta^{[L]} = \dot{f}_\theta^{[L]}(\theta^{[L]})\,\zeta^{[L]}$
for $\ell = L-1, \dots, 1$ **do**
  compute $\zeta^{[\ell]} = \dot{f}_a^{[\ell+1]}(a^{[\ell]})\,\zeta^{[\ell+1]}$
  compute $g_\theta^{[\ell]} = \dot{f}_\theta^{[\ell]}(\theta^{[\ell]})\,\zeta^{[\ell]}$

An Unrolled Example

To get a better feel for Algorithm 5.1, the operation of backpropagation, and the associated notation, we consider a simple hypothetical example as illustrated in Figure 5.8. This is a feedforward neural network with $L = 4$, arbitrary input dimension p, output dimension $q = 1$, and a single neuron on each hidden layer. That is $N_0 = p$ and $N_1, N_2, N_3, N_4 = 1$. Assume the loss function is

$$C(a^{[L]}, y\,;\, \theta) = \frac{1}{2}(a^{[L]} - y)^2,$$

and assume a model structure,

$$a^{[\ell]} = f^{[\ell]}(a^{[\ell-1]}\,;\, \theta^{[\ell]}) = \sigma^{[\ell]}({\theta^{[\ell]}}^\top a^{[\ell-1]}) \qquad \text{for} \qquad \ell = 1, 2, 3, 4, \tag{5.27}$$

similarly to the neural network structure of Section 5.1 where affine transformations are followed by activation functions, yet without bias terms. Since $N_0 = p$, we have that $\Theta^{[1]} = \mathbb{R}^p$ and since $N_1, \dots, N_4$ are all unity, we have that $\Theta^{[2]}, \Theta^{[3]}$, and $\Theta^{[4]}$ are each $\mathbb{R}$ and the transpose in (5.27) is not needed for $\ell = 2, 3, 4$.

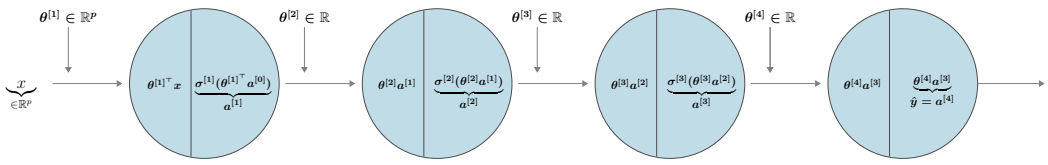


Figure 5.8: A simple hypothetical example with $L = 4$ and scalar hidden units.

In this hypothetical illustrative example, we use various activation functions. Namely,

$$\sigma^{[1]}(\cdot) = \sigma_{\text{ReLU}}(\cdot), \qquad \sigma^{[2]}(\cdot) = \sigma_{\text{Tanh}}(\cdot), \qquad \sigma^{[3]}(\cdot) = \sigma_{\text{Sig}}(\cdot),$$

and for the last step, we use the identity function, i.e., $\sigma^{[4]}(u) = u$. With the model specified we now have computable expressions for known derivatives as in (5.23), see also Section 5.3,

$$\begin{aligned}
\dot{C}(u) &= u - y, & \dot{f}_\theta^{[4]}(u) &= a^{[3]}, \\
\dot{f}_a^{[4]}(u) &= \theta^{[4]}, & \dot{f}_\theta^{[3]}(u) &= a^{[2]} \underbrace{\sigma_{\mathrm{Sig}}(ua^{[2]})(1 - \sigma_{\mathrm{Sig}}(ua^{[2]}))}_{\dot{\sigma}_{\mathrm{Sig}}(ua^{[2]})}, \\
\dot{f}_a^{[3]}(u) &= \theta^{[3]}\dot{\sigma}_{\mathrm{Sig}}(u\theta^{[3]}), & \dot{f}_\theta^{[2]}(u) &= a^{[1]} \underbrace{(1 - \sigma_{\mathrm{Tanh}}(ua^{[1]})^2)}_{\dot{\sigma}_{\mathrm{Tanh}}(ua^{[1]})}, \\
\dot{f}_a^{[2]}(u) &= \theta^{[2]}\dot{\sigma}_{\mathrm{Tanh}}(u\theta^{[2]}), & \dot{f}_\theta^{[1]}(u) &= a^{[0]} \underbrace{\mathbf{1}\{u^T a^{[0]} \geq 0\}}_{\dot{\sigma}_{\mathrm{ReLU}}(u^T a^{[0]})}.
\end{aligned} \tag{5.28}$$

Note that in this example we choose to treat $\dot{f}_\theta^{[1]}(u)$ as a p dimensional vector, while the other functions in this example are scalar valued both in their domain and co-domain.

Using these computable expressions for the derivatives, Algorithm 5.1 can be used to compute the gradient of the loss function with respect to θ. For illustration we unroll the algorithm where we use the notation $u \leftarrow v$ to indicate assignment of v to the variable u.

`Step 1`: We compute forward propagation,

$$\begin{aligned}
a^{[1]} &\leftarrow \sigma_{\mathrm{ReLU}}(x^\top \theta^{[1]}), \\
a^{[2]} &\leftarrow \sigma_{\mathrm{Tanh}}(\theta^{[2]} a^{[1]}), \\
a^{[3]} &\leftarrow \sigma_{\mathrm{Sig}}(\theta^{[3]} a^{[2]}), \\
\hat{y} = a^{[4]} &\leftarrow \theta^{[4]} a^{[3]}.
\end{aligned}$$

`Steps 2 and 3`:

$$\zeta^{[4]} \leftarrow a^{[4]} - y, \qquad g_\theta^{[4]} \leftarrow a^{[3]} \zeta^{[4]}.$$

Then the loop in steps 4–6 executes for three iterations for $\ell = 3, 2, 1$:

`Iteration` $\ell = 3$ `(steps 5 and 6)`:

$$\zeta^{[3]} \leftarrow \theta^{[4]} \zeta^{[4]}, \qquad g_\theta^{[3]} \leftarrow a^{[2]} \dot{\sigma}_{\mathrm{Sig}}(\theta^{[3]} a^{[2]}) \zeta^{[3]}.$$

`Iteration` $\ell = 2$ `(steps 5 and 6)`:

$$\zeta^{[2]} \leftarrow \theta^{[3]} \dot{\sigma}_{\mathrm{Sig}}(\theta^{[3]} a^{[2]}) \zeta^{[3]}, \qquad g_\theta^{[2]} \leftarrow a^{[1]} \dot{\sigma}_{\mathrm{Tanh}}(\theta^{[2]} a^{[1]}) \zeta^{[2]}.$$

`Iteration` $\ell = 1$ `(steps 5 and 6)`:

$$\zeta^{[1]} \leftarrow \theta^{[2]} \dot{\sigma}_{\mathrm{Tanh}}(\theta^{[2]} a^{[1]}) \zeta^{[2]}, \qquad g_\theta^{[1]} \leftarrow x\, \dot{\sigma}_{\mathrm{ReLU}}(x^\top \theta^{[1]}) \zeta^{[1]}.$$

By expanding out the resulting expressions and using the chain rule, we may verify that each of the intermediate outputs $g_\theta^{[4]}, g_\theta^{[3]}, g_\theta^{[2]}$, and $g_\theta^{[1]}$ yields the correct gradient expression.

Accounting for $\delta^{[\ell]}$ Instead of $\zeta^{[\ell]}$

Algorithm 5.1 summarizes backpropagation since it deals with gradient evaluation for the general recursive model (5.21). However, it does not make any use of the more specific structure of (5.2) which has an affine transformation followed by a non-linear activation function. It turns out that in the context of deep learning models as in (5.2), one can simplify the algorithm by keeping track of an alternative set of intermediate derivative values, $\delta^{[\ell]} \in \mathbb{R}^{N_\ell}$, instead of $\zeta^{[\ell]}$. These are,

$$\delta^{[\ell]} := \frac{\partial C(a^{[L]}, y\,;\,\theta)}{\partial z^{[\ell]}}, \qquad \ell = 1, \ldots, L,$$

where we observe that in contrast to the $\zeta^{[\ell]}$ values from above, these values are derivatives of the loss with respect to $z^{[\ell]}$ values instead of with respect $a^{[\ell]}$ values. Usage of $\delta^{[\ell]}$ is standard in backpropagation for deep learning and serves the basis for the main backpropagation algorithm that we present below. With this notation, the key recursive relationship for backward computation is,

$$\delta^{[\ell]} = \frac{\partial a^{[\ell]}}{\partial z^{[\ell]}} \frac{\partial z^{[\ell+1]}}{\partial a^{[\ell]}} \frac{\partial C}{\partial z^{[\ell+1]}} = \frac{\partial a^{[\ell]}}{\partial z^{[\ell]}} \frac{\partial z^{[\ell+1]}}{\partial a^{[\ell]}} \delta^{[\ell+1]}, \qquad \ell = L-1, \ldots, 1, \tag{5.29}$$

which in comparison to the left hand side of (5.25) breaks up the step $a^{[\ell]} \Rightarrow a^{[\ell+1]}$ into two steps: $a^{[\ell]} \Rightarrow z^{[\ell+1]}$ followed by $z^{[\ell+1]} \Rightarrow a^{[\ell+1]}$. Further, for the final layer, $\ell = L$, we have,

$$\delta^{[L]} = \frac{\partial a^{[L]}}{\partial z^{[L]}} \frac{\partial C}{\partial a^{[L]}}. \tag{5.30}$$

Backpropagation for Fully Connected Networks

We now expand and adapt Algorithm 5.1 using the key recursive relationships for $\delta^{[\ell]}$ (5.29) and (5.30). Consider first the component $\partial a^{[\ell]}/\partial z^{[\ell]}$ associated with the (vector) activation function $S^{[\ell]}(\cdot)$. In a typical layer ℓ, the vector activation function is composed of identical scalar activation functions as in (5.3) and hence the transposed Jacobian (see also (A.10) in Appendix A) is,

$$\frac{\partial a^{[\ell]}}{\partial z^{[\ell]}} = J_{S^{[\ell]}}(z^{[\ell]})^\top = \mathrm{Diag}\big(\dot{\sigma}^{[\ell]}(z^{[\ell]})\big). \tag{5.31}$$

Here $\mathrm{Diag}(\cdot)$ transforms a vector into a diagonal matrix and $\dot{\sigma}^{[\ell]}(\cdot)$ is the derivative of the scalar activation which is interpreted as operating element wise on the input vector. Examples of scalar activation functions and their derivatives are in Section 5.3.

In other cases, $S^{[\ell]}(\cdot)$ is not separable as in (5.3) and the transposed Jacobian of $S^{[\ell]}$ does not have a simple diagonal form as in (5.31). One such potential Jacobian is computed for the softmax in (5.19). However, the most common application of softmax is in the last layer $\ell = L$ together with cross entropy loss. In such a case, using (5.20) directly in place of (5.30) is more efficient and practical.

Continuing now with the recursive relationship (5.29), we consider the component $\partial z^{[\ell+1]}/\partial a^{[\ell]}$. Here since $z^{[\ell+1]} = W^{[\ell+1]} a^{[\ell]} + b^{[\ell+1]}$, we have

$$\frac{\partial z^{[\ell+1]}}{\partial a^{[\ell]}} = {W^{[\ell+1]}}^\top.$$

Hence, similarly to (5.26), putting the pieces together we have the recursive relationship,

$$\delta^{[\ell]} = \begin{cases} \frac{\partial C}{\partial z^{[L]}}, & \ell = L, \\ \mathrm{Diag}\big(\dot{\sigma}^{[\ell]}(z^{[\ell]})\big) {W^{[\ell+1]}}^\top \delta^{[\ell+1]}, & \ell = L-1, \ldots, 1. \end{cases} \tag{5.32}$$

As discussed above, the case $\ell = L$ can be computed using the two components of (5.30) separately or can be computed according to (5.20) in case of softmax combined with cross entropy loss.

Now with a recursive relationship supporting backpropagation of the $\delta^{[\ell]}$ values, we are ready to deal with the desired derivative values of the parameters $\theta^{[\ell]} = (W^{[\ell]}, b^{[\ell]})$. Define,

$$g_W^{[\ell]} = \frac{\partial C}{\partial W^{[\ell]}}, \qquad \text{and} \qquad g_b^{[\ell]} = \frac{\partial C}{\partial b^{[\ell]}}, \qquad \ell = 1, \ldots, L.$$

Here $g_W^{[\ell]}$ is an $N_\ell \times N_{\ell-1}$ matrix and $g_b^{[\ell]}$ is an N_ℓ-vector. Paralleling the right hand equation in (5.25) (which involves $\zeta^{[\ell]}$ for the more general model), we seek equations to retrieve these target values (gradient components) in terms of $\delta^{[\ell]}$. This is done via,

$$g_W^{[\ell]} = \frac{\partial C}{\partial z^{[\ell]}} \frac{\partial z^{[\ell]}}{\partial W^{[\ell]}} = \delta^{[\ell]} {a^{[\ell-1]}}^\top, \qquad \text{and} \qquad g_b^{[\ell]} = \frac{\partial z^{[\ell]}}{\partial b^{[\ell]}} \frac{\partial C}{\partial z^{[\ell]}} = \delta^{[\ell]}. \tag{5.33}$$

where the first expression results from (A.15) and the second expression from (A.14), both in Appendix A.

All of these relationships are packaged in the backpropagation algorithm for fully connected networks. See also Figure 5.9 for an illustration of the flow of information.

Algorithm 5.2: Backpropagation for fully connected networks

Input: Dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$,
objective function $C(\cdot) = C(\cdot\,; \mathcal{D})$, and
parameter values $\theta = (\theta^{[1]}, \ldots, \theta^{[L]})$

Output: derivatives of the loss $(g_W^{[1]}, g_b^{[1]}), \ldots, (g_W^{[L]}, g_b^{[L]})$

Compute $a^{[\ell]}$ and $z^{[\ell]}$ for $\ell = 1, \ldots, L$ (Forward pass)
Compute $\delta^{[L]} = \frac{\partial C}{\partial z^{[L]}}$
Set $g_W^{[L]} = \delta^{[L]} {a^{[L-1]}}^\top$ and set $g_b^{[L]} = \delta^{[L]}$
for $\ell = L-1 \ldots, 1$ **do**
    Compute $\delta^{[\ell]} = \mathrm{Diag}\big(\dot{\sigma}^{[\ell]}(z^{[\ell]})\big) {W^{[\ell+1]}}^\top \delta^{[\ell+1]}$
    Set $g_W^{[\ell]} = \delta^{[\ell]} {a^{[\ell-1]}}^\top$ and set $g_b^{[\ell]} = \delta^{[\ell]}$

Backpropagation on a Whole Mini Batch

In Section 4.2 we discussed the concept of iterative optimization using mini-batches. With this approach, instead of considering all of the data points, we only use n_b samples and use the gradient $\sum_i \nabla C_i / n_b$, summed over the samples in the mini-batch i. See also equation (4.13) in Chapter 4.

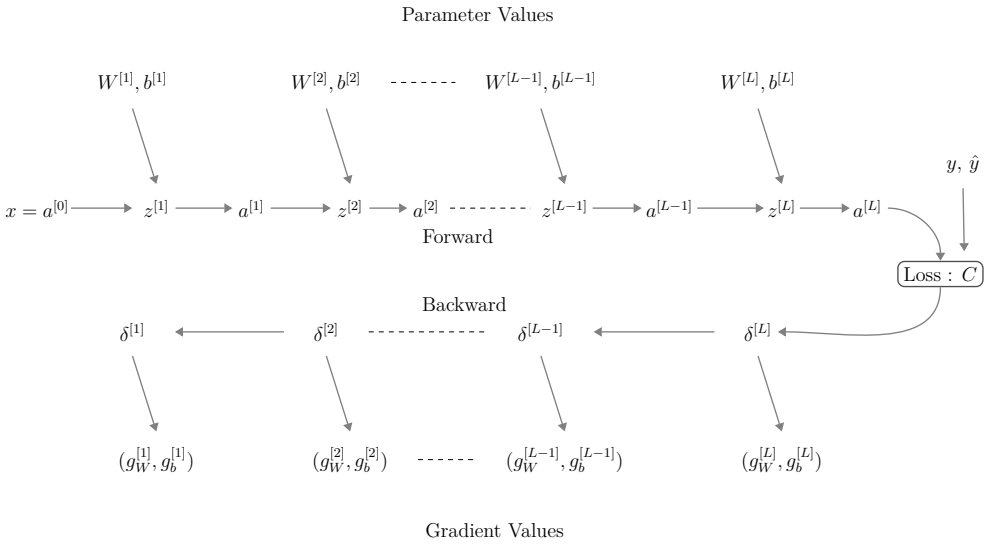


Figure 5.9: The variables and flow of information in the backpropagation algorithm for fully connected networks.

In fact, earlier in this chapter, in (5.9), we introduced notation for executing the forward pass simultaneously on a whole mini-batch. The key here is to properly organize the activation values and the data values in matrices. This notation can be further extended to adapt the backpropagation algorithm for computation of the mini-batch gradient. Hence once may also specify a form of Algorithm 5.2, suitable for mini-batches.

While specific implementation details are not our focus, we mention that in most practical deep learning frameworks, the common interface for gradient evaluation using backpropagation is based on input tensors, where each tensor is associated with a mini-batch. Then one of the indices of the tensor indexes specific data samples within the mini-batch. For example common formats for image data are based on 4 dimensional tensors, with a format referred to as *NHWC* or *NCHW*. Here 'N' represents the index within the mini-batch, while 'H', 'W', and 'C' are for height, width, and channels as appropriate for color image data. Note that the concept of channels is taken from the context of convolutional neural networks which are the topic of study in Chapter 6.

Vanishing and Exploding Gradients

The key backpropagation recursions are the forward step $a^{[\ell+1]} = S^{[\ell]}\big(W^{[\ell]}a^{[\ell-1]} + b^{[\ell]}\big)$ as in (5.4) and the backward step $\delta^{[\ell]} = \mathrm{Diag}\big(\dot{\sigma}^{[\ell]}(z^{[\ell]})\big){W^{[\ell+1]}}^{\top}\delta^{[\ell+1]}$, as in (5.32). From a practical perspective, these steps are sometimes subject to instability when the number of layers L is large.

To see this, let us first simplify the situation by ignoring the activation functions and assuming that $W^{[\ell]}$ is with a fixed square dimension and the same weight matrix W, for $\ell = 1, \ldots, L-1$, and the last weight matrix is $W^{[L]}$. Further, ignore the bias terms $b^{[\ell]}$. In

this simplified case,

$$\hat{y} = a^{[L]} = W^{[L]}W^{[L-1]}W^{[L-2]} \ \cdots \ W^{[3]}W^{[2]}W^{[1]}x = W^{[L]}W^{L-1}x, \tag{5.34}$$

where W^{L-1} is the $L-1$ power of W.

As is well known in linear algebra and systems theory, unless the maximal eigenvalues of W are exactly with a magnitude of unity, as L grows we have that $\hat{y}$ either vanishes (towards 0) or explodes (with values of increasing magnitude). As a further simplified illustration of this, if $W = wI$ (a constant multiple of the identity matrix), then $\hat{y} = W^{[L]}w^{L-1}x$, and for any $w \neq 1$, a vanshing $\hat{y}$ or exploding $\hat{y}$ phenomena persists. This illustration shows that for non-small network depths (large L), instability issues may arise in the forward pass.

The same type of instability problem can then also persist in the backward pass since the backward recursion $\delta^{[\ell]} = \text{Diag}\big(\dot{\sigma}^{[\ell]}(z^{[\ell]})\big)W^{[\ell+1]\top}\delta^{[\ell+1]}$ also includes repeated matrix multiplications, and if for simplicity we ignore the activation functions and again take a constant matrix W, then,

$$\delta^{[\ell]} = \left(W^{\top}\right)^{L-\ell}\delta^{[L]}. \tag{5.35}$$

Hence there is often a vanishing or exploding nature of $\delta^{[\ell]}$ for large L and low values of ℓ (the first layers of the network). Now with (5.33) in mind, we see that in general, the gradient values $g_W^{[\ell]}$ and $g_b^{[\ell]}$ may get smaller and smaller (vanishing) or larger and larger (exploding) as we go backward with every layer during backpropagation. These are respectively called the *vanishing gradient* or *exploding gradient* problems.

In the worst case, vanishing gradients, may completely stop the neural network from training, or exploding gradients may throw parameter values towards arbitrary directions. This may result in oscillations around the minima or even overshooting the optimum again and again. Another impact of exploding gradients is that huge values of the gradients may cause number overflow resulting in incorrect computations or introductions of `NaN` floating point values ("not a number").

Gradient descent improvements such as RMSProp, integrated in ADAM (see Section 4.3), can help normalize such variation in the gradients. Nevertheless, numerical instability can still persist. Further, with activation functions such as sigmoid or tanh, in cases of inputs far from 0 the gradient components of $\text{Diag}\big(\dot{\sigma}^{[\ell]}(z^{[\ell]})\big)$ may also vanish. Activation functions such as ReLU or leaky ReLU handle such problems, yet the overarching phenomenon associated with repeated matrix multiplication as exemplified in (5.34) and (5.35) still persists. A key strategy for mitigating such a problem is based on weight initialization as we discuss in the section below.

To handle exploding gradients, sometimes *gradient clipping* is employed. This approach adjusts the gradient so that it, or its individual components, are not too big in magnitude. One approach is to clip each coordinate of the gradient so that it does not exceed a pre-specified value in absolute value. This approach can obviously change the direction of the gradient since some coordinates may be clipped while others not. Another approach is to scale the whole gradient by its norm and to multiply by a fixed factor. This maintains the direction of the gradient as originally computed and ensures its magnitude is at a fixed threshold.

5.5 Weight Initialization

While gradient based optimization generally works in advancing towards local minima, as highlighted above, vanishing gradients or exploding gradients may significantly hinder progress. In fact, starting with initial values that are either constant or 0 for the weights and bias parameters may throw the learning process off. Such constant initial parameters may impose symmetry on the activation values of the hidden units and in turn prohibit the model from exploiting its expressive power.

Random initialization enables us to break any potential symmetries and is almost always preferable. Below we outline specific principles of random parameter intilization, yet even if these are not applied, the most basic random intilization approach is to set all parameters of the weight matrices $W^{[1]}, \ldots, W^{[L]}$ as independent and identically distributed standard normal random variables and to set all the entries of the bias vectors $b^{[1]}, \ldots, b^{[L]}$ at 0.

In view of the potential vanishing gradient and exploding gradient problems highlighted above, there is room for smarter weight intialization techniques. Specifically, a general principle that is followed focuses on the activation values $a^{[1]}, \ldots, a^{[L]}$ associated with the initial weight and bias parameters. If we momentarily consider these activation values as random entries, an overarching goal is that the distribution of each entry of $a^{[\ell+1]}$ is approximately similar to that of each entry of $a^{[\ell]}$. In such a case, recursing the forward pass (5.4), a form of distributional stability can persist when the number of layers L is large. With such an approach, if we are to choose the distribution of the initial weight matrix parameters judiciously, vanishing and exploding gradients may be mitigated at the onset of learning.

More concretely, the goal of equating the distribution of $a^{[\ell+1]}$ entries with $a^{[\ell]}$ entries is viewed via the first two moments of the distribution (mean and variance). Specifically, with activation function $\sigma(\cdot)$ we have in the ℓ-th layer,

$$a_i^{[\ell]} = \sigma\left(z_i^{[\ell]}\right), \qquad \text{where} \qquad z_i^{[\ell]} = \sum_{j=1}^{N_{\ell-1}} w_{i,j}^{[\ell]} a_j^{[\ell-1]} + b_i^{[\ell]}. \tag{5.36}$$

To develop an initialization approach, we assume that all $a_j^{[\ell-1]}$ values for $j = 1, \ldots, N_{\ell-1}$, are identically distributed with mean 0, some variance $\breve{v}$, and are statistically independent. Further, we wish to initialize parameters randomly such that $a_i^{[\ell]}$ is also with a mean of 0 and the same variance $\breve{v}$. Our strategy for this is to initialize the bias $b^{[\ell]}$ with entries 0 and the weight parameters $w_{i,j}^{[\ell]}$ as normally distributed random variables with a mean of 0 and some specified variance.

It turns out, that when the activation function, $\sigma(\cdot)$, is tanh, a sensible variance to use for $w_{i,j}^{[\ell]}$ is $1/N_{\ell-1}$. This form of initialization is called *Xavier initialization*. Further, when the activation function is ReLU, a sensible variance to use is $2/N_{\ell-1}$ and this form of initialization is called *He initialization*. Another commonly used heuristic is to use variance $2/(N_{\ell-1} + N_\ell)$ which is the harmonic mean of $\frac{1}{N_{\ell-1}}$ and $\frac{1}{N_\ell}$. In any case, all of these are heuristics, often implemented in practical deep learning frameworks.

Derivation of Xavier initialization

To get a feel for the considerations of the $1/N_{\ell-1}$ variance rule of Xavier initialization we now present the derivation of this rule. The approach is to equate activation variances at some $\breve{v}$, based on (5.36). To do so, we use an approximation of $\sigma_{\text{Tanh}}(\cdot)$ as $\sigma(u) = u$. This is a first-order Taylor approximation of the tanh activation function. With this approximation and with 0 bias entries, (5.36) yields,

$$\mathbb{V}\text{ar}\left(a_i^{[\ell]}\right) = \mathbb{V}\text{ar}\left(z_i^{[\ell]}\right) = \mathbb{V}\text{ar}\left(\sum_{j=1}^{N_{\ell-1}} w_{i,j}^{[\ell]} a_j^{[\ell-1]}\right) = \sum_{j=1}^{N_{\ell-1}} \mathbb{V}\text{ar}\left(w_{i,j}^{[\ell]} a_j^{[\ell-1]}\right). \tag{5.37}$$

This is based on the assumption that all random variables, both activation values and weight parameters, are statistically independent. Now to evaluate the variance summands on the right hand side, we use the following property of the variance of a product of two independent random variables X and Y,

$$\mathbb{V}\text{ar}\,(XY) = \mathbb{E}[X]^2\mathbb{V}\text{ar}\,(Y) + \text{Var}(X)\mathbb{E}[Y]^2 + \mathbb{V}\text{ar}\,(X)\mathbb{V}\text{ar}\,(Y).$$

With this we obtain,

$$\mathbb{V}\text{ar}\left(w_{i,j}^{[\ell]} a_j^{[\ell-1]}\right) = \mathbb{E}\left[w_{i,j}^{[\ell]}\right]^2 \mathbb{V}\text{ar}\left(a_j^{[\ell-1]}\right) + \mathbb{V}\text{ar}\left(w_{i,j}^{[\ell]}\right) \mathbb{E}\left[a_j^{[\ell-1]}\right]^2 + \mathbb{V}\text{ar}\left(w_{i,j}^{[\ell]}\right) \mathbb{V}\text{ar}\left(a_j^{[\ell-1]}\right).$$

Now since we seek activation values with a mean of 0 and a variance of $\breve{v}$, it turns out that $\mathbb{V}\text{ar}\left(w_{i,j}^{[\ell]} a_j^{[\ell-1]}\right) = \mathbb{V}\text{ar}\,(w_{i,j}^{[\ell]})\breve{v}$. Now also assuming this variance $\breve{v}$ for activations of layer ℓ, and combining in (5.37) we have,

$$\breve{v} = \sum_{j=1}^{N_{\ell-1}} \mathbb{V}\text{ar}\,(w_{i,j}^{[\ell]})\,\breve{v}.$$

By further requiring that all weight initialization variance entries for layer ℓ are constant, say at $\mathbb{V}\text{ar}\,(w_{i,j}^{[\ell]}) = \breve{w}_\ell$, we have $\breve{v} = N_{\ell-1}\,\breve{w}_\ell\,\breve{v}$. Then, this shows that setting $\breve{w}_\ell = 1/N_{\ell-1}$ achieves the desired result.

Further Insight Regarding Vanishing or Exploding Values

The derivation of the Xavier initialization offers further insight about the nature of vanishing or exploding values. Assume that the input features vector x is also random with each entry having the same variance $\breve{v}_x$. Then by recursing the forward pass (5.4), continuing to approximate the activation function as identity, and using the variance calculations above we have,

$$\mathbb{V}\text{ar}\left(a_j^{[L]}\right) = \left[\prod_{\ell=1}^{L} N_{\ell-1}\breve{w}_\ell\right] \breve{v}_x.$$

This further highlights that setting $\breve{w}_\ell = 1/N_{\ell-1}$ yields stability of the variance of the outputs which is especially important when the number of layers L is large. Other choices where $N_{\ell-1}\breve{w}_\ell < 1$ across all layers ℓ may yield vanishing activations, and similarly if $N_{\ell-1}\breve{w}_\ell > 1$ we may observe exploding activations.

5.6 Batch Normalization

In Section 2.1 we discussed standardization of data. As seen in equations (2.1) and (2.2), this is simply a transformation for each feature of the input data based on subtraction of the mean and division by the standard deviation. There are multiple benefits in carrying out standardization, one of which is a reshaping of the loss landscape. As an illustration, in Section 2.4 we explored the effect that such standardization has on simple problems. It turns out that with much more complicated deep neural networks, standardization also called *normalization*, may also be very beneficial. In this section we present an extended idea called *batch normalization* where the outputs of intermediate hidden layers are also normalized in an adaptive manner.

The overarching idea of batch normalization is to normalize (or standardize) not just the input data but also individual neuron values within the intermediate hidden layers or final layer of the network. In a nutshell returning to the display (5.2) and taking j as an index of a neuron in layer ℓ, we may wish to have either $z_j^{[\ell]}$ or $a_j^{[\ell]}$ exhibit near-normalized values over the input dataset. Such normalization of the neuron values then yields more consistent training and mitigates vanishing or exploding gradient problems. It also has a slight regularization effect which may prevent overfitting.

Our exposition here outlines normalization of the $z_j^{[\ell]}$ values, yet the reader should keep in mind that one may choose to do so for the $a_j^{[\ell]}$ values instead. When applying the batch normalization technique that we describe here, the output of the training process involves further parameters, some of which are trained via optimization (gradient descent), and others are based on running averages in the training process. We now present the details.

The Idea of Per Unit Normalization

The main idea of batch normalization is to consider neuron j in layer ℓ and instead of using $z_j^{[\ell]}$ as in (5.2), to use a transformed version $\tilde{z}_j^{[\ell]}$. Such a transformation takes place both in training time and when using the model in production, with subtle differences between the two cases as we describe below. The transformation aims to position the $\tilde{z}_j^{[\ell]}$ values so that they have approximately zero mean and unit standard deviation over the data. Further, the transformation involves a correction using trainable parameters.

The transformation requires estimates of the unit's mean and standard deviation so that the unit value can be normalized. During training time, at a given training epoch and for a given mini-batch of size n_b, such estimates are obtained via,

$$\hat{\mu}_j^{[\ell]} = \frac{1}{n_b}\sum_{i=1}^{n_b} z_j^{[\ell](i)} \qquad \text{and} \qquad \hat{\sigma}_j^{[\ell]} = \sqrt{\frac{1}{n_b}\sum_{i=1}^{n_b}(z_j^{[\ell](i)} - \hat{\mu}_j^{[\ell]})^2}, \tag{5.38}$$

where $z_j^{[\ell](i)}$ is the value at unit j, at layer ℓ, and sample i within the mini-batch, prior to carrying out normalization.

With $\hat{\mu}_j^{[\ell]}$ and $\hat{\sigma}_j^{[\ell]}$ available, we compute

$$\bar{z}_j^{[\ell](i)} = \frac{z_j^{[\ell](i)} - \hat{\mu}_j^{[\ell]}}{\sqrt{(\hat{\sigma}_j^{[\ell]})^2 + \varepsilon}}, \tag{5.39}$$

where $\varepsilon > 0$ is a small fixed quantity that ensures that we do not divide by zero. At this point $\bar{z}_j^{[\ell](i)}$ has nearly zero mean and nearly unit standard deviation for all data samples i in the mini-batch (it is nearly and not exactly only due to ε).

Now finally, an additional transformation takes place in the form,

$$\tilde{z}_j^{[\ell](i)} = \gamma_j^{[\ell]} \bar{z}_j^{[\ell](i)} + \beta_j^{[\ell]}, \tag{5.40}$$

where $\gamma_j^{[\ell]}$ and $\beta_j^{[\ell]}$ are trainable parameters. A consequence of (5.39) and (5.40) is that $\tilde{z}_j^{[\ell](i)}$ has a standard deviation of approximately $\gamma_j^{[\ell]}$ and a mean of approximately $\beta_j^{[\ell]}$ over the data samples i in the mini-batch. These parameters are respectively initialized at 1 and 0, and then as training progresses, $\gamma_j^{[\ell]}$ and $\beta_j^{[\ell]}$ are updated using the same learning mechanisms applied to the weights and biases of the network. Namely they are updated using backpropagation, and gradient based learning. Specific backpropagation details are presented at the end of this section.

As presented above, each unit or neuron has their own set of trainable parameters, $\gamma_j^{[\ell]}$ and $\beta_j^{[\ell]}$. However, in practice, multiple neurons in the same layer often share the same batch normalization parameters. This implies adjusting the mean and standard deviation estimates (5.38) to have summations over multiple neurons j. For example in convolutional neural networks as presented in the next chapter, all neurons of the same channel typically have the same batch normalization applied to them.

Batch Normalization in Production

When using the model in production we need to be able to apply the model to a single input data sample in which case evaluation of $\hat{\mu}_j^{[\ell]}$ and $\hat{\sigma}_j^{[\ell]}$ as in (5.38) is impossible. Instead, averages from the training set are collected during train time and these are supplied and deployed with the model. Practically, since parameters are updated during a training run and this updating in turn affects the $z_j^{[\ell]}$ values, averages are often collected in parallel to training. A common technique is to use exponential smoothing as in (4.14) in Chapter 4, and apply it to the sequence of computed values $\hat{\mu}_j^{[\ell]}$ and $\hat{\sigma}_j^{[\ell]}$ between mini-batches during training. The result of this exponential smoothing, denoted here as $\hat{\hat{\mu}}_j^{[\ell]}$ and $\hat{\hat{\sigma}}_j^{[\ell]}$, is then deployed together with the model.

As a summary, in a deployed model, each unit (j in layer ℓ), or set of units, to which batch normalization is applied is deployed with the trained $\gamma_j^{[\ell]}$ and $\beta_j^{[\ell]}$ values as well as with the exponentially smoothed estimates $\hat{\hat{\mu}}_j^{[\ell]}$ and $\hat{\hat{\sigma}}_j^{[\ell]}$. Then in production,

$$\tilde{z}_j^{[\ell]} = \gamma_j^{[\ell]} \frac{z_j^{[\ell]} - \hat{\hat{\mu}}_j^{[\ell]}}{\sqrt{\left(\hat{\hat{\sigma}}_j^{[\ell]}\right)^2 + \varepsilon}} + \beta_j^{[\ell]}, \tag{5.41}$$

is used in place of $z_j^{[\ell]}$.

Interestingly, returning to (5.2) and observing that the vector $z^{[\ell]} = W^{[\ell]}a^{[\ell-1]} + b^{[\ell]}$ is an affine function of the previous activation vector, we see that when combined with (5.41) we are left with an affine transformation $\tilde{z}^{[\ell]} = \tilde{W}^{[\ell]}a^{[\ell-1]} + \tilde{b}^{[\ell]}$ with a modified weight matrix $\tilde{W}^{[\ell]}$ and bias vector $\tilde{b}^{[\ell]}$. Hence at least in principle, deployment of batch normalization of this sort can be done without the production model needing to be aware of batch normalization at all since we can encode it in the weight matrices and bias vectors.[6]

Backpropagation of Batch Normalization Parameters

We close this section with an exposition of how the batch normalization parameters $\gamma_j^{[\ell]}$ and $\beta_j^{[\ell]}$ can be updated as part of the backpropagation algorithm. To simplify the notation we omit the superscript $[\ell]$ and the subscript j as the batch learning parameters are either neuron-specific or are shared by a group of neurons. Our focus is thus on one pair γ and β for which we require $\partial C/\partial\gamma$ and $\partial C/\partial\beta$ respectively. We also require computing the intermediate derivative value $\partial C/\partial z^{(i)}$ for backpropagation to operate.

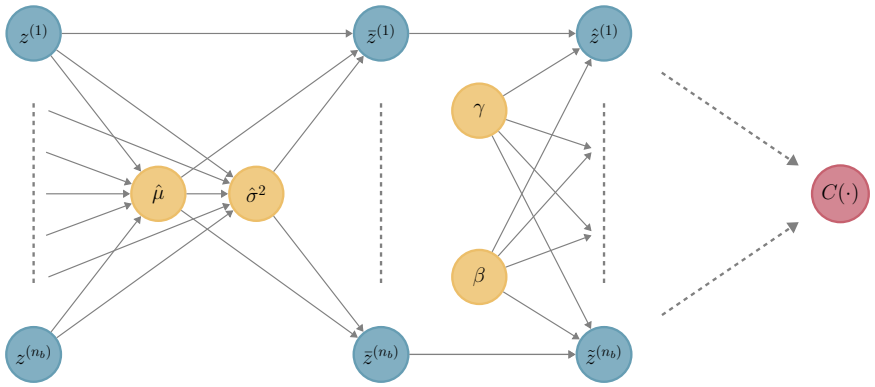


Figure 5.10: A schematic of the computational graph for batch normalization at an arbitrary layer. The goal is to compute the gradients of the loss with respect to γ, β and each $z^{(i)}$.

At each iteration, the network estimates the mean $\hat{\mu}$ and the standard deviation $\hat{\sigma}$ corresponding to the current batch. In the forward pass, given inputs (output of the previous layer) $z^{(i)}$ for $i = 1, \ldots, n_b$, we calculate the outputs of the batch normalization procedure $\tilde{z}^{(i)}$. Then, the backpropagation pass will propagate the gradient of the loss function $C(\cdot)$ through this transformation and compute the gradients with respect to the parameters γ and β. We then also compute the intermediate derivatives $\partial C/\partial z^{(i)}$.

A schematic of the computational graph associated to the batch normalization steps is presented in Figure 5.10. Backpropagation of the higher layers yields the gradient $\frac{\partial C}{\partial \tilde{z}^{(i)}}$. With this we want to compute $\frac{\partial C}{\partial \gamma}$, $\frac{\partial C}{\partial \beta}$, and $\frac{\partial C}{\partial z^{(i)}}$.

[6] This consolidation of batch normalization into the weight matrix and bias vector is often not done in practice, especially due to the fact that W is often a convolution matrix as described in the next chapter.

The gradients $\partial C/\partial\gamma$ and $\partial C/\partial\beta$ are simple. By applying the chain rule with (5.40), we get

$$\frac{\partial C}{\partial \gamma} = \sum_{i=1}^{n_b} \frac{\partial C}{\partial \tilde{z}^{(i)}} \bar{z}^{(i)}, \qquad \text{and} \qquad \frac{\partial C}{\partial \beta} = \sum_{i=1}^{n_b} \frac{\partial C}{\partial \tilde{z}^{(i)}}. \tag{5.42}$$

In order to compute $\frac{\partial C}{\partial z^{(i)}}$, we need also to evaluate $\frac{\partial C}{\partial \hat{\mu}}$, $\frac{\partial C}{\partial \hat{\sigma}^2}$, and $\frac{\partial C}{\partial \bar{z}^{(i)}}$ since,

$$\frac{\partial C}{\partial z^{(i)}} = \frac{\partial C}{\partial \bar{z}^{(i)}} \frac{\partial \bar{z}_{(i)}}{\partial z^{(i)}} + \frac{\partial C}{\partial \hat{\sigma}^2} \frac{\partial \hat{\sigma}^2}{\partial z^{(i)}} + \frac{\partial C}{\partial \hat{\mu}} \frac{\partial \hat{\mu}}{\partial z^{(i)}}. \tag{5.43}$$

With the multivariate chain rule and (5.39) and (5.38) we have,

$$\begin{aligned}
\frac{\partial C}{\partial \bar{z}^{(i)}} &= \frac{\partial C}{\partial \tilde{z}^{(i)}} \gamma, \\
\frac{\partial C}{\partial \hat{\sigma}^2} &= \sum_{i=1}^{n_b} \frac{\partial C}{\partial \bar{z}^{(i)}} \frac{\partial \bar{z}^{(i)}}{\partial \hat{\sigma}^2} = \sum_{i=1}^{n_b} \frac{\partial C}{\partial \bar{z}^{(i)}} \left(z^{(i)} - \hat{\mu}\right) \left(-\frac{1}{2}\right) \left(\hat{\sigma}^2 + \varepsilon\right)^{-3/2}, \\
\frac{\partial C}{\partial \hat{\mu}} &= \sum_{i=1}^{n_b} \frac{\partial C}{\partial \bar{z}^{(i)}} \frac{\partial \bar{z}^{(i)}}{\partial \hat{\mu}} + \frac{\partial C}{\partial \hat{\sigma}^2} \frac{\partial \hat{\sigma}^2}{\partial \hat{\mu}} = \sum_{i=1}^{n_b} \frac{\partial C}{\partial \bar{z}^{(i)}} \frac{-1}{\sqrt{\hat{\sigma}^2 + \varepsilon}} + \frac{\partial C}{\partial \hat{\sigma}^2} \frac{\sum_{i=1}^{n_b} -2\left(z^{(i)} - \hat{\mu}\right)}{n_b}.
\end{aligned}$$

These formulas present us with an explicit expression by transforming (5.43) to

$$\frac{\partial C}{\partial z^{(i)}} = \frac{\partial C}{\partial \bar{z}^{(i)}} \frac{1}{\sqrt{\hat{\sigma}^2 + \varepsilon}} + \frac{\partial C}{\partial \hat{\sigma}^2} \frac{2(z^{(i)} - \hat{\mu})}{n_b} + \frac{\partial C}{\partial \hat{\mu}} \frac{1}{n_b}. \tag{5.44}$$

This representation of (5.44) can then be integrated with backpropagation through the neural network.

5.7 Mitigating Overfitting with Dropout and Regularization

In Section 2.5 we discussed the challenges and tradeoffs associated with overfitting and the need for generalization. On the one hand we seek a model that will make use of the available data and properly capture the dependence on the input features, while on the other hand we wish to avoid overfitting. As presented in Section 2.5, one general approach for mitigating overfitting is called *regularization*, where as in (2.33) we may augment the loss function $C(\cdot)$ with a regularization term $R_\lambda(\theta)$, and optimize $\min_\theta \; C(\theta\,;\mathcal{D}) + R_\lambda(\theta)$, in place of just minimization of the loss function. Such practice using additive regularization is possible with deep neural networks as well. However, there is also an alternative popular approach called *dropout* which is specific to deep neural networks. We first describe the dropout approach and then highlight relationships between dropout, ensemble methods, as well as addition of regularization terms.

Dropout

The idea of *dropout* is to randomly zero out certain neurons during the training process. This allows training to focus on multiple random subsets of the parameters and yields a form of regularization.

With dropout, at any backpropagation iteration (forward pass and backward pass) on a mini-batch, only some random subset of the neurons is active. Practically neurons in layer ℓ,

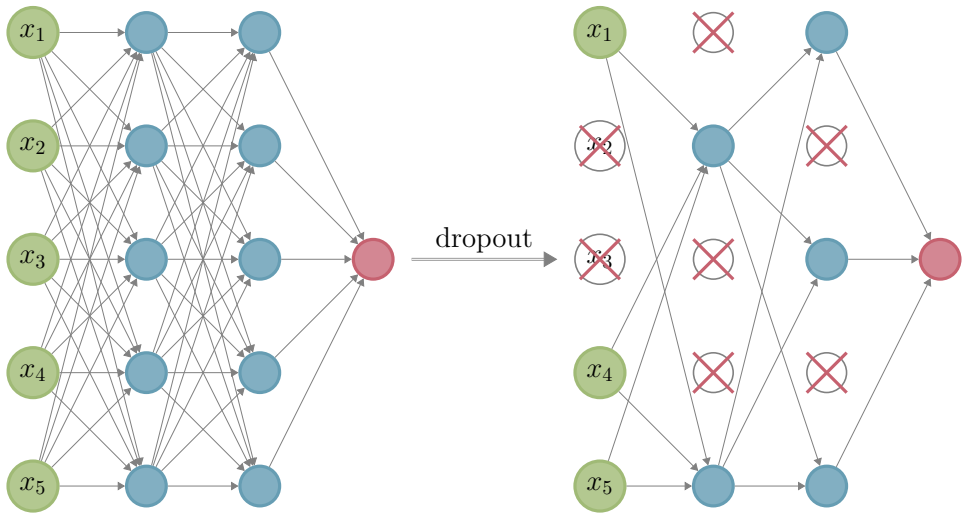


Figure 5.11: An illustration of dropout during training for a network with $L = 3$ layers, $p = 5$ input features, and $q = 1$ output. In each iteration during training, the network is transformed to the network on the right where the dropped out units are randomly selected.

for $\ell = 0, \ldots, L-1$, have a specified probability $p_{\text{keep}}^{[\ell]} \in (0, 1]$ where if $p_{\text{keep}}^{[\ell]} = 1$ dropout does not affect the layer, and otherwise each neuron i of the layer is "dropped out" with probability $1 - p_{\text{keep}}^{[\ell]}$. This is simply a zeroing out of the neuron activation $a_i^{[\ell]}$ as we illustrate in Figure 5.11.

In the forward pass, when we get to the neurons of layer $\ell + 1$, all the neurons in layer ℓ that were zeroed out do not participate in the computation. Specifically, the update for a neuron j in the next layer, assuming a scalar activation function $\sigma(\cdot)$, becomes,

$$a_j^{[\ell+1]} = \sigma\big(b_j^{[\ell+1]} + \sum_{i \text{ kept}} w_{i,j}^{[\ell+1]} a_i^{[\ell]}\big). \tag{5.45}$$

If $p_{\text{keep}}^{[\ell]} = 1$ all neurons are kept and the summation is over $i = 1, \ldots, N_\ell$ as in (5.7), but otherwise the summation is over the random subset of kept neurons. The Bernoulli coin flips determining which neurons are kept and which are not are all carried out independently within the layer, across layers, and throughout the training iterations.

In the backward pass, the effect of dropping out a neuron is evident via (5.33). When neuron i is dropped out in layer ℓ, the weights $w_{i,j}^{[\ell+1]}$ for all neurons $j = 1, \ldots, N_{\ell+1}$ are updated based on the gradient $[g_W^{[\ell]}]_{i,j}$ which is set at 0. With a pure gradient descent optimizer this means that weights $w_{i,j}^{[\ell+1]}$ are not updated at all during the given iteration, whereas with a momentum based optimizer such as ADAM it means that the descent step for those weights has a smaller magnitude; see Section 4.3 for a review of such optimizers.

At the end of training, even with dropout implemented, the trained model still has a complete set of weight matrices without any zeroed out elements, similar to the case in which we do not have dropout. Hence to account for the fact that neuron i in layer ℓ only took part in a proportion of iterations $p_{\text{keep}}^{[\ell]}$ during the training, when using the model in production (test time), we would like to use a weight matrix $\tilde{W}^{[\ell+1]} = p_{\text{keep}}^{[\ell]} W^{[\ell+1]}$ in place of $W^{[\ell+1]}$. The rationale here is to have the production forward pass similar to the training pass. Namely such a production forward pass has,

$$a_j^{[\ell+1]} = \sigma\big(b_j^{[\ell+1]} + \sum_{i=1}^{N_\ell} p_{\text{keep}}^{[\ell]} w_{i,j}^{[\ell+1]} a_i^{[\ell]}\big), \tag{5.46}$$

and this serves as an approximation to (5.45). To see the basis of this approximation, treat the summands in (5.45), $w_{i,j}^{[\ell+1]} a_i^{[\ell]}$, as identically distributed random variables, say with expected value μ_j. Now observe that the expected value of both the summation (over a random number of elements) in (5.45) and the expected value of the summation in (5.46) are both $N_\ell \, p_{\text{keep}}^{[\ell]} \, \mu_j$.

In practice, the training–production pair (5.45)–(5.46) is not typically used per-se. The more practical alternative is instead of remembering $p_{\text{keep}}^{[\ell]}$ and deploying it with the production model, the training forward pass is modified to have the reciprocal of $p_{\text{keep}}^{[\ell]}$ as a scaling factor of the weights. Namely, the training forward pass is,

$$a_j^{[\ell+1]} = \sigma\big(b_j^{[\ell+1]} + \sum_{i \text{ kept}} \frac{1}{p_{\text{keep}}^{[\ell]}} w_{i,j}^{[\ell+1]} a_i^{[\ell]}\big). \tag{5.47}$$

This form allows to use the resulting model normally in production without having to take dropout into consideration at all. Namely, the production forward pass is,

$$a_j^{[\ell+1]} = \sigma\big(b_j^{[\ell+1]} + \sum_{i=1}^{N_\ell} w_{i,j}^{[\ell+1]} a_i^{[\ell]}\big), \tag{5.48}$$

With this setup, the expected value of the summations in (5.47) and (5.48) agree and further the forward step (5.48) agrees with the standard model as in (5.2), (5.4), and (5.7).

In practice, this simple and easy to implement idea of dropout has improved performance of deep neural networks in many empirically tested cases. It is now an integral part of deep learning training. We now explore the idea a bit further though the viewpoint of ensemble methods.

Viewing Dropout as an Ensemble Approximation

Dropout can be viewed as an approximation of an *ensemble method*, a general concept from machine learning. Let us first present an overview of ensemble methods or *ensemble learning* and then argue why dropout serves as an approximation.

In general when we seek a model $\hat{y} = f_\theta(x)$, we may use the same dataset to train multiple models that all try to achieve the same task. We may then combine the models into an *ensemble* (model). The latter is usually more accurate than each of the individual models.

Let us illustrate this in the case of a scalar output model. We can choose to use M models and denote each model via $\hat{y}^{\{i\}} = f^{\{i\}}_{\theta_{\{i\}}}(x)$ for $i = 1, \ldots, M$, where $\theta_{\{i\}}$ is taken here as the set of parameters of the i-th model. The ensemble model on an input x is then the average,

$$f_\theta(x) = \frac{1}{M}\sum_{i=1}^{M} f^{\{i\}}_{\theta_{\{i\}}}(x), \qquad \text{where} \qquad \theta = (\theta_{\{1\}}, \ldots, \theta_{\{M\}}). \tag{5.49}$$

Clearly $f_\theta(\cdot)$ is more computationally costly since it requires M models instead of a single model. This implies storing M parameter sets, training M times instead of once, and evaluating M models in (5.49) instead of once during production. Nevertheless, there are benefits.

To illustrate the strength of this technique assume for simplicity that the models are homogenous in nature and only differ due to randomness in the training process and not the model choice or hyper-parameters. In this case, for some fixed unseen input $\tilde{x}$ we may treat the output of model i, denoted $\hat{y}^{\{i\}}_{\theta_{\{i\}}}(\tilde{x})$, as a random variable that is identical in distribution to every other model output $\hat{y}^{\{j\}}_{\theta_{\{j\}}}(\tilde{x})$, yet generally not independent. We further assume that any pair of model outputs is identically distributed to any other pair. In this case we may denote,

$$\mathbb{E}\big[\hat{y}^{\{i\}}_{\theta_{\{i\}}}(\tilde{x})\big] = \mu, \qquad \mathbb{V}\text{ar}\left(\hat{y}^{\{i\}}_{\theta_{\{i\}}}(\tilde{x})\right) = \sigma^2, \qquad \text{and} \qquad \text{cor}\big(\hat{y}^{\{i\}}_{\theta_{\{i\}}}(\tilde{x}), \hat{y}^{\{j\}}_{\theta_{\{j\}}}(\tilde{x})\big) = \rho,$$

respectively, where $\text{cor}(\cdot,\cdot)$ is the correlation between two models $i \neq j$ and is assumed to be the same for all i, j pairs. Such an assumption on the correlation also imposes[7] a lower bound on the correlation,

$$-\frac{1}{M-1} \leq \rho, \qquad \text{or} \qquad 0 \leq \rho + \frac{1-\rho}{M}. \tag{5.50}$$

We can now evaluate the mean and variance of the ensemble model (5.49). Namely,

$$\mathbb{E}[f_\theta(\tilde{x})] = \frac{1}{M}\mathbb{E}\big[\sum_{i=1}^{M} f^{\{i\}}_{\theta_{\{i\}}}(\tilde{x})\big] = \mu, \tag{5.51}$$

and further noting that $\rho\sigma^2$ is the covariance between any two models we obtain,[8]

$$\mathbb{V}\text{ar}\left(f_\theta(\tilde{x})\right) = \frac{1}{M^2}\mathbb{V}\text{ar}\Big(\sum_{i=1}^{M} f^{\{i\}}_{\theta_{\{i\}}}(\tilde{x})\Big) = \frac{1}{M^2}\big(M\sigma^2 + M(M-1)\rho\sigma^2\big) = \Big(\rho + \frac{1-\rho}{M}\Big)\sigma^2. \tag{5.52}$$

With (5.50) it is confirmed that as required this variance expression in (5.52) is non-negative. Further, we see that as the number of models in the ensemble, M, grows, the variance of the ensemble model converges to $\rho\sigma^2$. Since in addition to (5.50), $\rho \leq 1$ and practically $\rho < 1$, this limiting variance is less than σ^2. For example if $\rho = 0.5$ as M grows the variance of the estimator drops by 50%.

Putting the computational costs aside, these properties of ensemble models make them very attractive because the bias does not change as shown in (5.51), but the variance decreases;

[7] This may be shown based on the constraint that the covariance matrix is a positive semi-definite matrix.
[8] The variance of a sum of random variables is the sum of the elements of the covariance matrix.

recall also the general discussion of the bias variance tradeoff in Section 2.5. Nevertheless, deep learning models are not easily amenable for ensemble models in the form of (5.49) because the number of parameters and computational cost (both for training and production) is too high. Training a single model may sometimes take days and the computational costs of a single evaluation $f^{\{i\}}_{\theta_{\{i\}}}(\tilde{x})$ are also non-negligible. This is where dropout comes in.

We may loosely view dropout as an ensemble of M models where M is the number of training iterations. In a practical training scenario, M can be on the order of hundreds, thousands, or tens of thousands, depending on the number of epochs during training and the size of the mini-batch in comparison to the number of training samples. For example, if $n = 10^5$ training samples and the mini-batch size is $n_b = 100$, then each epoch has 1000 iterations and if we execute, say 1000 epochs of training then $M = 10^6$. Then, since the production model involves weights accumulated throughout all 10^6 iterations, we may loosely view the production model as an ensemble model and we may expect its variance to be reduced from σ^2 to approximately $\rho\sigma^2$. Now clearly each of the M iterations did not execute training of the model fully but rather only trained for a single iteration. Hence this ensemble view of dropout is merely a heuristic description.

Addition of Regularization Terms and Weight Decay

In addition to dropout, as already introduced in Section 2.5, addition of a regularization term is another key approach to prevent overfitting and improve generalization performance. Augmenting the loss with a regularization term $R_\lambda(\theta)$ restricts the flexibility of the model, and this restriction is sometimes needed to prevent overfitting. In the context of deep learning, and especially when ridge regression style regularization is applied, this practice is sometimes called *weight decay* when considering gradient based optimization. We now elaborate.

Take the original loss function $C(\theta)$ and augment it to be $\tilde{C}(\theta) = C(\theta) + R_\lambda(\theta)$. In our discussion here, let us focus on the ridge regression type regularization with parameter $\lambda > 0$, and,

$$R_\lambda(\theta) = \frac{\lambda}{2} R(\theta), \qquad \text{with} \qquad R(\theta) = \|\theta\|^2 = \theta_1^2 + \ldots + \theta_d^2.$$

Here for notational simplicity, we consider all the d parameters of the model as scalars, θ_i for $i = 1, \ldots, d$ (not to be confused with the notation used above for the parameters of a model as part of an ensemble). Nevertheless, note that typically regularization is only applied to the weight matrix parameters and not to the bias vectors. Further, we may even restrict regularization to certain layers and not others.

Now assume we execute basic gradient descent steps as in (2.20). With a learning rate $\alpha > 0$, the update at iteration t is,

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla \tilde{C}(\theta^{(t)}).$$

In our ridge regression style penalty case we have $\nabla \tilde{C}(\theta) = \nabla C(\theta) + \lambda\theta$, and hence the gradient descent update can be represented as

$$\theta^{(t+1)} = (1 - \alpha\lambda)\theta^{(t)} - \alpha \nabla C(\theta^{(t)}). \tag{5.53}$$

Now the interesting aspect of (5.53), assuming that $\alpha\lambda < 2$, is that it involves shrinkage or weight decay directly on the parameters in addition to gradient based learning. That is, independently of the value of the gradient $\nabla C(\theta^{(t)})$, in every iteration, (5.53) continues to decay the parameters, each time multiplying the previous parameter by a factor $1 - \alpha\lambda$.

This weight decay phenomena can then be extended algorithmically to enforce regularization not directly via addition of a regularization term, but rather simply by augmenting the gradient descent updates to include weight decay. For example we may consider popular gradient based algorithms such as ADAM in Section 4.3 and the other algorithms in Section 4.5, and in each case add an additional step which incorporates multiplying the weights by a constant less than but close to unity.

Notes and References

The origins of deep learning date back to the same era during which the digital computer materialised. In fact, early ideas of artificial neural networks were introduced first in 1943 with [283]. Then, in the post WWII era, Frank Rosenblatt's *perceptron* was the first working implementation of a neural network model, [353]. The perceptron and follow up work drew excitement in the 1960's yet with the 1969 paper [291], there was formal analysis that shone negative light on limited capabilities of single layer neural networks and this eventually resulted in a decline of interest, which is sometimes termed the "AI winter" of 1974–1980. Before this period there were even implementations of deep learning architectures with [206], and with [205] where 8 layers were implemented. In 1967 such *multi-layer perceptrons* were even trained using *stochastic gradient descent*, [12].

Some attribute the end of the 1974–1980 AI winter to a few developments that drew attention and resulted in impressive results. Some include *Hopfield networks* which are recurrent in nature (see Chapter 7), and also formalism and implementation of the *backpropagation algorithm* in 1981, [421]. In fact, early ideas of backpropagation can be attributed to a PhD thesis in 1970, [265] by S. Linnainmaa. See also our notes and references on early developments of reverse mode automatic differentiation at the end of Chapter 4. In parallel to this revival of interest in artificial intelligence in the early 1980's there were many developments that led to today's contemporary convolutional neural networks. See the notes and references at the end of Chapter 6. Historical accounts of deep learning can be found in [367] and [370] as well as a website by A. Kurenkov.[9] The book [169] was a key reference of neural networks, summarizing developments up to the turn of the twenty first century. The 2015 Nature paper by Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, [249] captures more contemporary developments.

Positive results about the universal approximation ability of neural networks, such as Theorem 5.1 presented in this chapter, appear in [97] for a class of sigmoid activations and in [186] for a larger class of non-polynomial functions. With such results, it became evident that with a single hidden layer, neural networks are very expressive. Still, the practical insight to add more hidden layers to increase expressivity arose with the work of Geoffrey Hinton et al. in 2006 [180], Yoshua Bengio et al. 2006 [30] and other influential including works such as [28], [86] and [248]. The big explosion was in 2012 with *AlexNet* in [239].

Our example of a multiplication gate as in Figure 5.4 comes from [259]. Our motivation to use this example and the analysis we present around Figure 5.5 dealing with the expressivity of deeper networks is motivated by a 2017 talk of Niklas Wahlström.[10] Beyond our elementary presentation, many researchers have tried to provide theoretical justifications to why deep neural networks are so powerful. Some justifications of the power of deep learning are in [376] using *information theory*, and [90], [116],[334] , and [399], using other mathematical reasoning approaches. See also [285] and [333] for surveys of theoretical analysis of neural networks.

In terms of activation functions, the sigmoid function was the most popular function in early neural architectures with the *tanh* function serving as an alternative. The popularity of *ReLU* grew with [301], especially after its useful application in the AlexNet convolutional neural network [239]. Note that ReLU was first used in 1969, see [126]. Other activation functions such as *leaky ReLU* were introduced in [276], as well as parameterized activation functions such as *PReLU* studied in [171] in order to mitigate vanishing and exploding gradient issues. A general survey of activation functions is in [111].

The backpropagation algorithm can be attributed to [357], yet has earlier origins with general *backward mode automatic differentiation* surveyed in [24] (see also notes of Chapter 4). Our presentation in Algorithm 5.2 is specific to the precise form of feedforward networks that we considered, yet variants can be implemented. Importantly, with the advent of automatic differentiation frameworks such as *TensorFlow* [1] followed by *PyTorch* [324], *Flux.jl* [201], *JAX* [57], and others, the use of backpropagation as part of deep learning has become standard. Such software frameworks automatically implement backpropagation as a special case of backward mode automatic differentiation where the computational graph is constructed, often automatically based on code. Early deep learning work up to about 2014, did not have such software frameworks available and hence "hand coding" of backpropagation was more delicate and time consuming. It is fair to say that with the proliferation of deep learning software frameworks, innovations in research and industry accelerated greatly. We

[9] https://www.skynettoday.com/overviews/neural-net-history.
[10] See https://www.it.uu.se/katalog/nikwa778/talks/DL_EM2017_online.pdf.

also note that properly considering matrix analysis aspects of backpropagation often requires *matrix calculus* for which useful references are [141] and [330].

Extensive discussion of *vanishing* and *exploding gradients* is in [323]. Related aspects of training including *weight initialization* are discussed in chapter 7 of [336]. The *Xavier initialization* technique was introduced in [137]. Related initialization techniques and their analysis are studied in [171]. The idea of *batch normalization* was initially introduced in [203] and analyzed in [17] and [202]. Since then, batch normalization has been extended in several ways including instance normalization in [406]. A survey is in [194]. Our analysis of backpropagation of batch-normalization parameters is from [203]. *Dropout* was initially introduced in [181] and [388]. Analysis of dropout as an ensemble approximation is in [159]. See also [263] for an overview of ensemble methods. Further study of dropout and its implications is in [128]. *Regularization* in feedforward networks was reviewed in [243] and analysis of weight decay is in [241] and [270].