Lecture 5

Training Neural Networks: A Deep Dive

By

Dr. Muhammad Alrabeiah

Electrical Engineering Dept., KSU

Spring 2022

March 8th, 2022

Disclaimer These notes are still under development, and they have not been subjected to proper review and revision.

Contents

Chapter 1

Training Neural Networks

With the foundation of iterative training laid in Lecture 4, we are now ready to take a deep dive into the training of neural networks. As we have established, we will follow an iterative training approach that is based on gradient descent, for a close-form solution is out of reach. However, training neural networks is not that simple; they commonly comprise multiple layers and are heavily parameterized, forming complex augmented functions. This makes gradient computations cumbersome and expensive (in terms of computational resources).

In this chapter, we will learn a way to deal with that complexity. We will discuss the vastly popular *backpropagation algorithm*, which is the key to train modern neural networks. The basic idea behind the algorithm rests on the use of the chain rule of derivatives. It is a point of strength for the algorithm as well as a point of weakness. We will discuss both in the next two sections.

1 The Backpropagation Algorithm

The layered nature of neural networks sort of implies that parameters should be adjusted sequentially and in accordance to their reliance on each other. Recall from Lecture 3 that a neural network is formed by stacking layers of neurons one after another, mush like the simple schematic in Figure 1.1. This creates a form of dependence between the parameters of layer 1 and layer 2. We will see now how this dependence is decoupled using the chain rule.

Let's learn backpropagation with a couple of examples from Figure 1.1. First let the

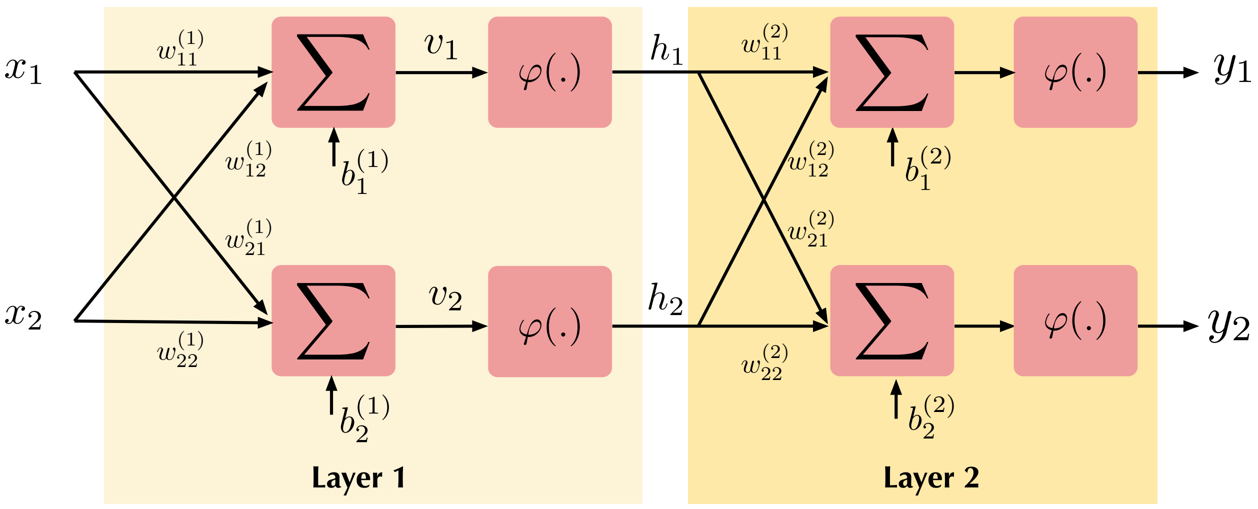


Figure 1.1: An example of simple two-layer neural network.

weights of layers 1 and 2 be represented as follows

$$\mathbf{W}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \end{bmatrix} = \begin{bmatrix} \mathbf{w_1^{(1)}}^T \\ \mathbf{w_2^{(1)}}^T \end{bmatrix} \tag{1.1}$$

$$\mathbf{W}^{(2)} = \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} \end{bmatrix} = \begin{bmatrix} \mathbf{w_1^{(2)}}^T \\ \mathbf{w_2^{(2)}}^T \end{bmatrix}, \tag{1.2}$$

and let the activations of layer 2 be linear.

Eample 1: now, say we are interested in the gradient of the loss, $\mathcal{L}$, w.r.t. the weights of layer 2, in particular $\mathbf{w}_1^{(2)}$. This gradient could be decomposed using the chain rule (see Lecture 4) into

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_1^{(2)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{w}_1^{(2)}} \tag{1.3}$$

where $\mathbf{y} = [y_1 \ y_2]^T$. Let compute each partial separately starting with $\frac{\partial \mathbf{y}}{\partial \mathbf{w}_1^{(2)}}$. This partial is a Jacobian as both $\mathbf{y}$ and $\mathbf{w}_1^{(2)}$ are vectors. It is given by

$$\frac{\partial \mathbf{y}}{\partial \mathbf{w}_1^{(2)}} = \begin{bmatrix} \partial y_1/\partial w_{11}^{(2)} & \partial y_1/\partial w_{12}^{(2)} \\ \partial y_2/\partial w_{11}^{(2)} & \partial y_2/\partial w_{12}^{(2)} \end{bmatrix}. \tag{1.4}$$

The other partial, namely $\frac{\partial \mathcal{L}}{\partial \mathbf{y}}$, is a gradient since $\mathcal{L}$ is a scalar and $\mathbf{y}$ is a vector, and it is written as

$$\frac{\partial \mathcal{L}}{\partial \mathbf{y}} = [\partial \mathcal{L}/\partial y_1 \; \partial \mathcal{L}/\partial y_2] \, . \tag{1.5}$$

Therefore, the gradient of $\mathcal{L}$ w.r.t. $\mathbf{w}_1^{(2)}$ is the result of multiplying the two factors as follows

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_1^{(2)}} = [\partial \mathcal{L}/\partial y_1 \; \partial \mathcal{L}/\partial y_2] \begin{bmatrix} \partial y_1/\partial w_{11}^{(2)} & \partial y_1/\partial w_{12}^{(2)} \\ \partial y_2/\partial w_{11}^{(2)} & \partial y_2/\partial w_{12}^{(2)} \end{bmatrix} \tag{1.6}$$

$$= \left[\frac{\partial \mathcal{L}}{\partial y_1}\frac{\partial y_1}{\partial w_{11}^{(2)}} + \frac{\partial \mathcal{L}}{\partial y_2}\frac{\partial y_2}{\partial w_{11}^{(2)}} \quad \frac{\partial \mathcal{L}}{\partial y_1}\frac{\partial y_1}{\partial w_{12}^{(2)}} + \frac{\partial \mathcal{L}}{\partial y_2}\frac{\partial y_2}{\partial w_{12}^{(2)}} \right]. \tag{1.7}$$

Evaluating the partials in Equation 1.7 results in the gradient of $\mathcal{L}$ w.r.t. $\mathbf{w}_1^{(2)}$. Note the following:

1. The partials $\partial y_2/\partial w_{11}^{(2)}$ and $\partial y_2/\partial w_{12}^{(2)}$ equal zeros, for y_2 is not a function of $w_{11}^{(2)}$ and $w_{12}^{(2)}$. This is true for any input vector $\mathbf{x}$ and target vector $\mathbf{t}$.
2. The gradient of $\mathcal{L}$ w.r.t. $\mathbf{y}$ is written as a row vector so that it could be multiplied by the Jacobian $\frac{\partial \mathbf{y}}{\partial \mathbf{w}_1^{(2)}}$. This makes the gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_1^{(2)}}$ a row vector, as well.

Example 2: in the above example, we looked at a weight vector in the last layer, so let's now turn our attention to a weight vector in a deeper layer. Consider computing the gradient of $\mathcal{L}$ w.r.t. $\mathbf{w}_1^{(1)}$ in layer 1 of the schematic in Figure 1.1. We will follow a similar approach where we start by decomposing the gradient using the chain rule and compute each partial separately. The gradient in this case is given by

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_1^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{h}}\frac{\partial \mathbf{h}}{\partial \mathbf{v}}\frac{\partial \mathbf{v}}{\partial \mathbf{w}_1^{(1)}}, \tag{1.8}$$

where $\mathbf{h} = [h_1 \; h_2]^T$ and $\mathbf{v} = [v_1, v_2]^T$. Note that $\frac{\partial \mathcal{L}}{\partial \mathbf{y}}$ is already computed in Equation 1.5, so we need to evaluate the rest of the partials as follows

- We start with the Jacobian of $\mathbf{y}$ w.r.t. the input of layer 2

$$\frac{\partial \mathbf{y}}{\partial \mathbf{h}} = \begin{bmatrix} \partial y_1/\partial h_1 & \partial y_1/\partial h_2 \\ \partial y_2/\partial h_1 & \partial y_2/\partial h_2 \end{bmatrix} \tag{1.9}$$

- We then compute the Jacobian of $\mathbf{h}$ w.r.t. $\mathbf{v}$, which are the partials of the output of layer 1 w.r.t. the input to the activation functions of the same layer

$$\frac{\partial \mathbf{h}}{\partial \mathbf{v}} = \begin{bmatrix} \partial h_1/\partial v_1 & \partial h_1/\partial v_2 \\ \partial h_2/\partial v_1 & \partial v_2/\partial v_2 \end{bmatrix} \tag{1.10}$$

$$= \begin{bmatrix} \partial h_1/\partial v_1 & 0 \\ 0 & \partial v_2/\partial v_2 \end{bmatrix}, \tag{1.11}$$

  where the zeros are a result of the fact that h_1 and h_2 do not depend on v_2 and v_1, respectively.

- Finally, we compute the Jacobian $\frac{\partial \mathbf{v}}{\partial \mathbf{w}_1^{(1)}}$, which relates the output of the linear combiner to the weights of layer 1.

$$\frac{\partial \mathbf{v}}{\partial \mathbf{w}_1^{(1)}} = \begin{bmatrix} \partial v_1/\partial w_{11}^{(1)} & \partial v_1/\partial w_{12}^{(1)} \\ \partial v_2/\partial w_{11}^{(1)} & \partial v_2/\partial w_{12}^{(1)} \end{bmatrix} \tag{1.12}$$

$$= \begin{bmatrix} \partial v_1/\partial w_{11}^{(1)} & \partial v_1/\partial w_{12}^{(1)} \\ 0 & 0 \end{bmatrix} \tag{1.13}$$

  where the zeros result from v_2 not depending on $w_{11}^{(1)}$ and $w_{12}^{(1)}$ in any way, see Figure 1.1.

With all partials computed, we just need to multiply them to get our gradient as follows

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_1^{(1)}} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial y_1}\frac{\partial y_1}{\partial h_1}\frac{\partial h_1}{\partial v_1}\frac{\partial v_1}{\partial w_{11}^{(1)}} + \frac{\partial \mathcal{L}}{\partial y_2}\frac{\partial y_2}{\partial h_1}\frac{\partial h_1}{\partial v_1}\frac{\partial v_1}{\partial w_{11}^{(1)}} & \frac{\partial \mathcal{L}}{\partial y_1}\frac{\partial y_1}{\partial h_1}\frac{\partial h_1}{\partial v_1}\frac{\partial v_1}{\partial w_{12}^{(1)}} + \frac{\partial \mathcal{L}}{\partial y_2}\frac{\partial y_2}{\partial h_1}\frac{\partial h_1}{\partial v_1}\frac{\partial v_1}{\partial w_{12}^{(1)}} \end{bmatrix} \tag{1.14}$$

The above two examples show how the chain rule of derivatives could be used to evaluate the gradient of the loss function with respect to any parameter in the neural network. The Backpropagation algorithm is merely an implementation of the chain rule; it evaluates the

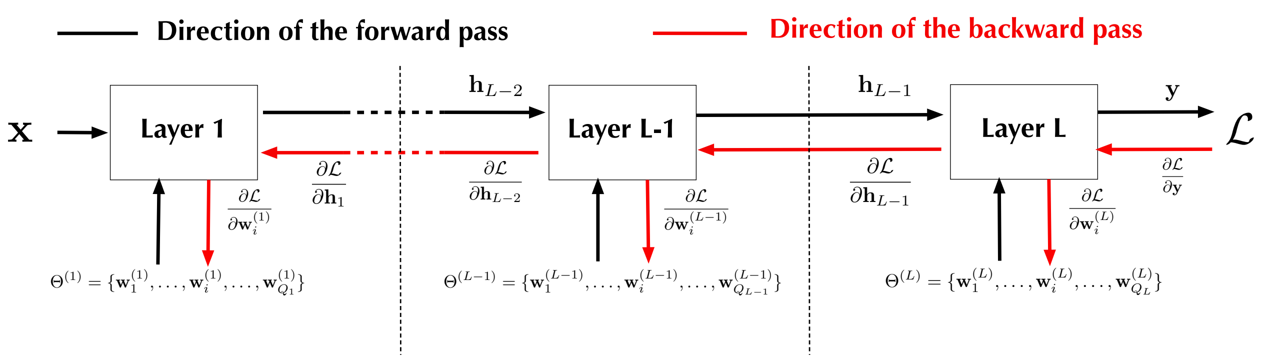

Direction of the forward pass
Direction of the backward pass
$\mathbf{x}$
Layer 1
Layer L-1
Layer L
$\mathcal{L}$
$\mathbf{h}_{L-2}$
$\mathbf{h}_{L-1}$
$\mathbf{y}$
$\frac{\partial\mathcal{L}}{\partial\mathbf{h}_1}$
$\frac{\partial\mathcal{L}}{\partial\mathbf{h}_{L-2}}$
$\frac{\partial\mathcal{L}}{\partial\mathbf{h}_{L-1}}$
$\frac{\partial\mathcal{L}}{\partial\mathbf{y}}$
$\frac{\partial\mathcal{L}}{\partial\mathbf{w}_i^{(1)}}$
$\frac{\partial\mathcal{L}}{\partial\mathbf{w}_i^{(L-1)}}$
$\frac{\partial\mathcal{L}}{\partial\mathbf{w}_i^{(L)}}$
$\Theta^{(1)} = \{\mathbf{w}_1^{(1)},\dots,\mathbf{w}_i^{(1)},\dots,\mathbf{w}_{Q_1}^{(1)}\}$
$\Theta^{(L-1)} = \{\mathbf{w}_1^{(L-1)},\dots,\mathbf{w}_i^{(L-1)},\dots,\mathbf{w}_{Q_{L-1}}^{(L-1)}\}$
$\Theta^{(L)} = \{\mathbf{w}_1^{(L)},\dots,\mathbf{w}_i^{(L)},\dots,\mathbf{w}_{Q_L}^{(L)}\}$

Figure 1.2: Implementing backpropagation

gradient of the loss w.r.t. any parameter, and applies a *form of gradient descent* to update the parameter.

1.1 How Is It Implemented?

This is an important question. Evaluating partials may seem simple to us, people, but it may not be that simple from a programming perspective. As the network grows deeper, the number of factors in the chain rule grows, see how many factors are in Equation 1.8 compared to Equation 1.3. Do we really need to calculate all those factors as we go deeper? Or is there a more modular approach to keep computations simple.

Let's answer this question by first defining the following two operations:

Definition 1.1. Forward Pass is the operation where the network sees the input and propagates it all the way to the output, obtaining the predictions. Then, it computes the loss based on the available groundtruth targets. This operation corresponds to the first two steps in the iterative training procedure, see Lecture 4.

Definition 1.2. Backward Pass is the operation where the training algorithm (backpropagation for example) starts computing the derivative of the loss w.r.t. the parameters of the network. Then, it updates those parameters in preparation for the next forward pass.

The backward pass, and similarly the forward pass, is sequential in nature. Backpropagation works by breaking down the gradient into a sequence of partials. These partials follow the same order of the layers (operations) in the network. Let's try to understand this by considering Equations 1.3 and 1.8. The first factor $\frac{\partial\mathcal{L}}{\partial\mathbf{y}}$ represents a common factor between the two Equations. Hence, once it is computed for Equation 1.3 (the last layer), it should not be re-calculated again. The algorithm should just save the value for deeper layers.

This observation hints at a way to implement backpropagation, moving backwards from the loss to each layer, one by one. The following summarizes the steps:

1. Start with the loss $\mathcal{L}$ and compute its derivative w.r.t. the output of the network $\mathbf{y}$.

2. Use the partial $\frac{\partial \mathcal{L}}{\partial \mathbf{y}}$ to compute the gradient of $\mathcal{L}$ w.r.t. $\mathbf{w}_i^{(L)}$ where $i \in \{1, \dots, Q_L\}$. Then, update the parameters. See Equation 1.3 for example.

3. Compute the Jacobian of $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_{L-1}}$ and pass it back to the proceeding layer, e.g., $\frac{\partial \mathcal{L}}{\partial \mathbf{h}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{h}}$ in Equation 1.8.

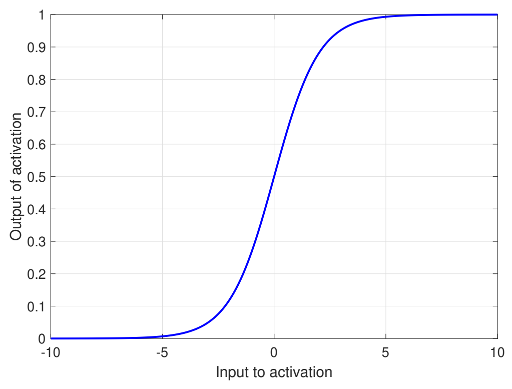
4. Use the Jacobine to compute the gradient of $\mathcal{L}$ w.r.t. $\mathbf{w}_i^{(L-1)}$ where $i \in \{1, \dots, Q_{L-1}\}$. See the last two factors of Equation 1.8 as an example, i.e., $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_1^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{v}} \frac{\partial \mathbf{v}}{\partial \mathbf{w}_1^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{v}} \frac{\partial \mathbf{v}}{\partial \mathbf{w}_1^{(1)}}$.

5. Repeat points 3 and 4 for layer $L - 2$.

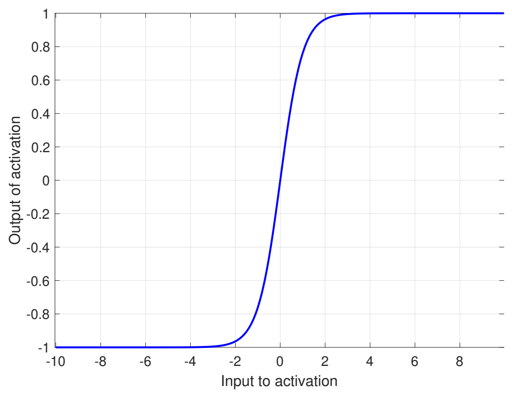
Figure 1.2 is a visual illustration of the above points. Notice how at any layer $L - j$ (where $j \in \{1, \dots, L-1\}$), we need to get the Jacobian of $\mathcal{L}$ w.r.t. $\mathbf{h}_{L-j}$ to compute the gradient of $\mathcal{L}$ w.r.t. $\mathbf{w}_i^{(L-j)}$. Then, we prepare the new Jacobian of $\mathcal{L}$ w.r.t. $\mathbf{h}_{L-j-1}$ to pass it back to the proceeding layer $L - j - 1$ so it can update its parameters. This sequential nature of backpropagation makes it modular and makes the computations manageable.

2 The Vanishing Gradient Problem

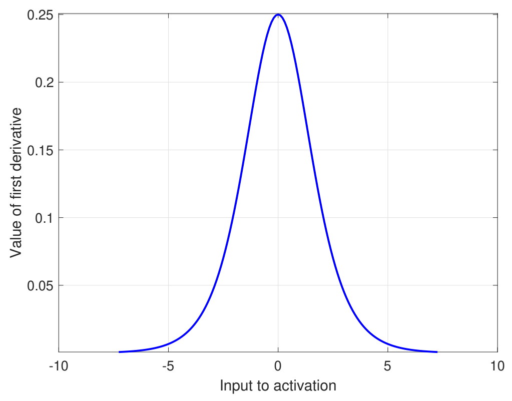
The elegance of backpropagation is commonly shadowed by the vanishing gradient problem. Such problem is a consequence of factorizing the gradient and the use of bounded activation functions, e.g., sigmoid and hyperbolic tangent. It is one of the most pervasive issues that causes training failures [1, 2], and various solutions have been proposed along the years to deal with it. Here, we are interested in developing a good understanding for the problem, and we will not go deep into how to deal with it. We will just discuss a simple solution and scatter the rest of the discussion across other chapters.



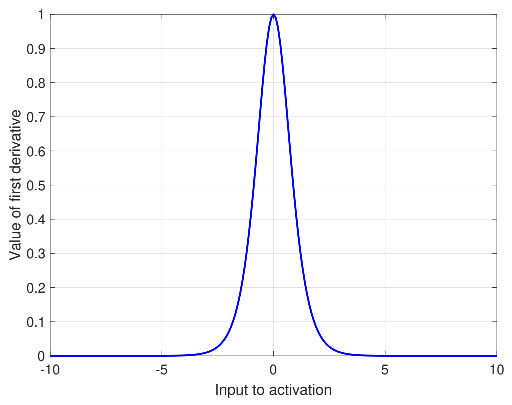
(a) Sigmoid



(b) tanh



(c) Derivative of sigmoid



(d) Derivative of tanh

What is the vanishing gradient problem? This is the key question of this section. Let's answer it by examining Equation 1.8, which is composed of many factors. What happens if one of the partials in the middle dies out, i.e., a Jacobian full of zeros? This causes the gradient flow (i.e., backward pass) to be blocked. *That means the gradient of any parameter relying on that partial might either be zero or partly diminished in value.* For instance, if the partial $\frac{\partial h_1}{\partial v_1}$ is zero, then the first element of the gradient in Equation 1.14 dies out (i.e., vanishes), and the weight $w_{11}^{(1)}$ dose not get updated! In a network deeper than our example in Figure 1.1, this vanishing gradient will cause some parameters in earlier layers (closer to the input side) to get stuck at certain values, as well; their gradients depend on the dead partial and, therefore, they vanish!

2.1 The Role of The Activation Function

A natural question to ask now is what causes a gradient to vanish in the first place? One simple answer is the use of bounded (i.e., squashing) activation functions. Let's dig deeper here. Recall from Lecture 3 the sigmoid and tanh functions which are given by

$$\varphi_1(x) = \frac{1}{1+e^{-x}} \tag{1.15}$$

$$\varphi_2(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{1.16}$$

Both functions are bounded above and below, see Figures 1.3a and 1.3b. As such, their first derivatives have a small activation (i.e., non-zero) range, which is illustrated in Figures 1.3c and 1.3d. Notice how the gradient dies out as a function of the output of the linear combiner (i.e., the input to the activation). What the figures tell us is that if the output of the i-th neuron in the l-th layer of a network approaches one of the edges of the activation range, then its partial $\frac{\partial h_i}{\partial v_i}$ rapidly dies out, causing the gradient of some parameters in layers $l, l-1, l-2, \dots, 1$ to go to zero as well.

There are different ways to combat vanishing gradients, one of which is the choice of an activation function. Consider replacing a sigmoid or a tanh function with a ReLU. This makes the activation function unbounded from above, expanding the activation range. However, it is still bounded from below; this makes it prone to vanishing gradient when the linear combiner outputs negative values. Another possible choice is parameterized ReLU that is defined as

$$\varphi_{\text{ParReLU}}(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}, \tag{1.17}$$

where α is a learnable parameter. This function is not bounded at all (i.e., has infinite activation range), and, thus, it has a gradient for any linear combiner output. Although semi-bounded and unbounded activations can mitigate the vanishing gradient problem, they sometime cause a mirror problem called "exploding gradient." As the name indicates, this problem happens when the output of a neuron blows up in value, driving the value of the gradient to be huge. Such huge value could bring about overflow issues or even cause the weight update to be unreasonable[1].

REMARK: There are other more powerful approaches to deal with vanishing gradients such as *batch normalization* and *dropout regularization*. We will discuss them in later chapters and lectures.

3 Parameter Initialization

The parameters of a neural network has to be initialized to some values before the training starts. This might seem like an easy requirement, but it is not. The function a neural network is representing is complex, and this leads to a complex loss function. Such complexity means the loss function is full of *bad minima*, several critical points that result in an underfitted or overfitted model[2]. Therefore, where the algorithm starts (the initial set of parameters) partially determines how likely it is going to fall in one of those bad minima.

It is always preferable to initialize the parameters such that they are close to either the global or a good local minimum. Unfortunately, this is a difficult requirement to satisfy, and to add insult to injury, how to best initialize a neural network is, to this day, not well understood. The only thing known with certainty is that we need to break symmetry when initializing the parameters [2].

Breaking symmetry means the initial parameters must be all different form each other. Such requirement guarantees that different neurons in a layer would learn to respond to different patterns in their inputs. To understand this, think of this simple example. When two neurons are initialized to have the same weights in a fully-connected network, they will compute the same output as their input is the same. This makes their gradients more likely to be the same, leading to a fruitless update; they keep computing the same function through out training, and this produces redundant patterns, which could harm the learning process.

A solution to breaking symmetry is to initialize the parameters by sampling a *high-entropy probability distribution*—please see the remark below. Sampling a distribution like the uniform or Gaussian distributions results in parameters that are likely to be very different from one another. Such difference indicates they could respond to different patterns in the

[1]remember that the learning rate is usually fixed at the start of training

[2]Please note that not all local minima are bad, leading to under- or over- fitting; a small bunch results in reasonably good and equivalent models.

input, regardless of whether the response is good or bad. This leads to a better learning experience.

REMARK: Entropy characterizes uncertainty in a probability distribution and it is given for a probability mass function $p_X(x)$ by

$$H(p) = -\sum_{x \in S_X} p_X(x) \log\left[p_X(x)\right] \tag{1.18}$$

where S_X is the sample space of the random variable X. Simply, the higher the entropy is the higher the uncertainty. For example, the discrete uniform distribution has the highest possible entropy amongst discrete distributions. This means when we draw a sample from the distribution, it is very likely to be any of the elements of the sample space! The reader should refer to an information theory book for more details such as [3].

Chapter 2

Regularization

As we have already established, training neural networks is not an easy job. We have seen how vanishing gradient could hinder the training process and how bad initialization could lead to bad local minimum. We have discussed some simple techniques to tackle training challenges like choosing semi- or non-saturating activations and sampling high-entropy distributions, and in this chapter, we will expand on them by discussing a more sophisticated family of techniques, which is collectively referred to as training regularization.

1 The Intuition

Recall from Lecture 4 that training a machine learning algorithm is equivalent to searching a hypothesis space that is picked beforehand. For neural networks, the hypothesis space is not easily characterized like in the case of linear regression. However, we can safely say that the space is defined once the architecture of the network is defined—architecture here refers to the depth, breadth, type of activation, and so forth—and we can also safely assume it is relatively large given the universal approximation theorem [4]. Searching such space is not easy, and we have already established some of the challenges associated with this search in Chapter 1. This gives rise to the question: *is there a way to guide the training algorithm during the search process (i.e., training) such that it avoids bad solutions?* This is the core question behind training regularization.

Let's try to develop some intuition about what regularization means before we go ahead and discuss some popular methods. Consider Figure 2.1. The training process could be

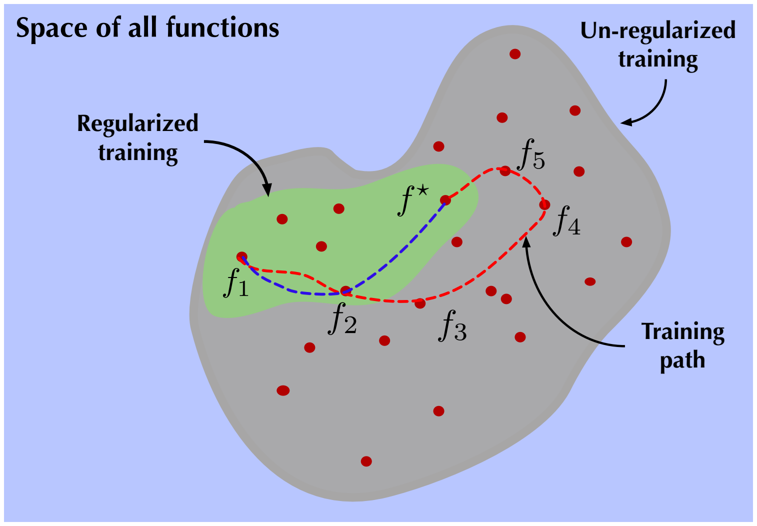


Figure 2.1: Regularization guides training by restricting the feasible hypothesis space.

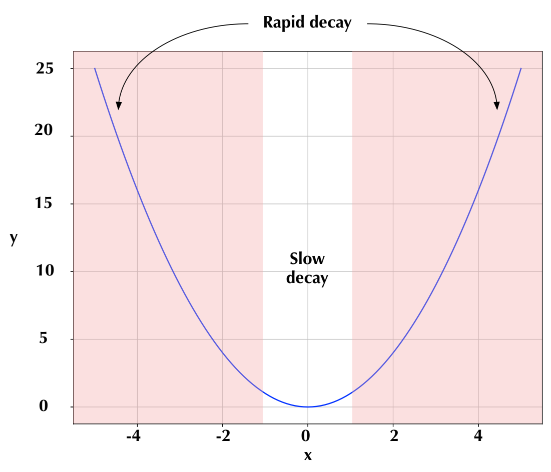
visualized as a path traversing a set of functions in the hypothesis space defined by the network architecture, e.g., the red dashed line passing through the gray set of functions (hypothesis space) from f_1 to $f^\star$ in Figure 2.1. When a regularization method is considered, the hypothesis space of the same network shrinks, changing the training path along with it, e.g., the dashed blue path passing through the green set from f_1 to $f^\star$ in Figure 2.1. If we consider the length of the path in the figure a reflection of how difficult the training process is, then the path in the regularized training case is expect to be shorter than that of the un-regularized case.

Regularization could be encoded into the performance metric used to train an algorithm in the form of a penalty. More specifically, the optimization problem describing the training process could be re-expressed with an extra term, the regularization term, as follows

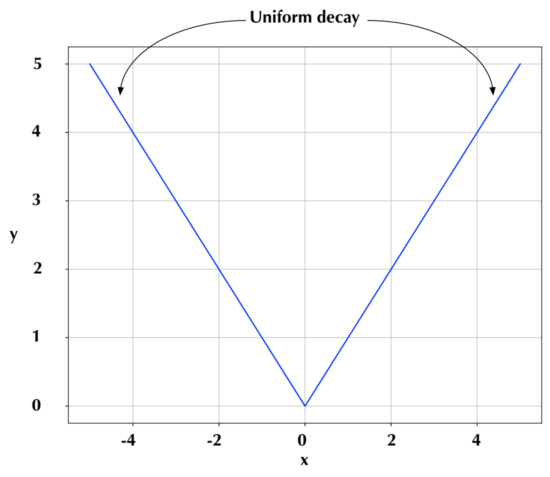
$$\min_{\Theta} \ \tilde{\mathcal{L}} = \min_{\Theta} \ \mathcal{L} + \lambda\Omega(\Theta), \tag{2.1}$$

where $\Omega(\Theta)$ is the regularization function, and $\lambda \geq 0$ is a scaling factor. Note that:

- Regularization is some function of the algorithm parameters enforcing a certain behavior.
- The scaling factor λ plays the role of balancing Equation 2.1. Its value enforces a form of trade-off between minimizing the loss function $\mathcal{L}$ and minimizing the regularization



(a)



(b)

Figure 2.2

function $\Omega(\Theta)$.

2 Weight Decay and Sparsity Regularization

There are may regularization methods in training neural networks, and machine learning algorithms in general. In this section we will discuss two very popular methods, which are *weight decay* and *sparsity regularization*. Both methods have their roots in statistical learning theory, yet since we do not focus on the statistical view of machine learning in this course, we will discuss them from a functional perspective.

2.1 Weight Decay

This is arguably the most popular form of regularization. It involves penalizing the weights of the network so that they become close to zero. Formally, this is given by

$$\min_{\Theta} \; \mathcal{L} + \lambda \sum_{i=1}^{|\Theta|} w_i^2, \tag{2.2}$$

where $|\Theta|$ is the total number of parameters of the algorithm. To understand the role of the regularization function, let us first visualize it for a signal variable in Figure 2.2a. This is a simple square function, where the rate of decay changes as a function of x; notice how fast the value of y decays for the region defined by $|x| > 1$ (the red regions in the figure), but such decay slows down once we are in the region $|x| \leq 1$ (close to zero). This behavior indicates that the regularization function tends to push the weights to become small but not zeros.

We can also consolidate the above observation analytically. Consider the gradient descent update rule for the ith weight of a neural network

$$w_i^{(t)} = w_i^{(t-1)} - \eta \nabla \tilde{\mathcal{L}}|_{w_i = w_i^{(t-1)}} \tag{2.3}$$

the gradient for the new loss function $\tilde{\mathcal{L}}$ in Equation 2.2 is given by

$$\nabla \tilde{\mathcal{L}} = \nabla \mathcal{L} + 2\lambda w_i \tag{2.4}$$

substituting back into Equation 2.3

$$w_i^{(t)} = w_i^{(t-1)} - \eta \nabla \mathcal{L}|_{w_i = w_i^{(t-1)}} - 2\eta\lambda w_i|_{w_i = w_i^{(t-1)}}, \tag{2.5}$$

and re-arranging gives us

$$w_i^{(t)} = (1 - 2\eta\lambda) w_i^{(t-1)} - \eta \nabla \mathcal{L}|_{w_i = w_i^{(t-1)}}. \tag{2.6}$$

Let's explore the behavior of Equation 2.5 over several updates (iterations t)

$$\begin{aligned}
t = 1, \quad w_i^{(1)} &= (1 - 2\eta\lambda) w_i^{(0)} - \eta \nabla \mathcal{L}|_{w_i = w_i^{(0)}} \\
t = 2, \quad w_i^{(2)} &= (1 - 2\eta\lambda) w_i^{(1)} - \eta \nabla \mathcal{L}|_{w_i = w_i^{(1)}} \\
&= (1 - 2\eta\lambda)^2 w_i^{(0)} - (1 - 2\eta\lambda)\eta \nabla \mathcal{L}|_{w_i = w_i^{(0)}} - \eta \nabla \mathcal{L}|_{w_i = w_i^{(1)}} \\
t = 3, \quad w_i^{(3)} &= (1 - 2\eta\lambda) w_i^{(2)} - \eta \nabla \mathcal{L}|_{w_i = w_i^{(2)}} \\
&= (1 - 2\eta\lambda)^3 w_i^{(0)} - (1 - 2\eta\lambda)^2 \eta \nabla \mathcal{L}|_{w_i = w_i^{(0)}} - (1 - 2\eta\lambda)\eta \nabla \mathcal{L}|_{w_i = w_i^{(1)}} - \eta \nabla \mathcal{L}|_{w_i = w_i^{(2)}}.
\end{aligned}$$

This gives raise to the following formula

$$w_i^{(t)} = (1 - 2\eta\lambda)^{(t)} w_i^{(0)} - \eta \left[\sum_{j=0}^{t-1} (1 - 2\eta\lambda)^{t-j-1} \nabla\mathcal{L}\big|_{w_i = w_i^{(j)}} \right] \tag{2.7}$$

The product $2\eta\lambda$ is typically less than one[1], and this means the factor $(1-2\eta\lambda) < 1$. Equation 2.7 is interesting; its first term decays exponentially faster than the other term with increasing number of iterations. Its second term, on the other hand, is a sum of exponentially decaying gradients. Notice how the oldest gradients (i.e., evaluated at early iterations $t = 1, 2, 3, \dots$) has higher decay rate than those evaluated close to the end of training, and since the gradient dies out as we get closer to the end (an intrinsic property of gradient descent), the whole term approaches, but mostly never gets to, zero. Putting it all together, we can see how Equation 2.7 could push the weight closer to zero as training progresses, explaining why the regularization technique is referred to as weight decay.

2.2 Sparsity Regularization

This is another popular regularization technique. It aims at introducing a form of sparsity in the algorithm parameters. This means it encourages most of the parameters in the set Θ to be zeros. This is an important property for some applications, e.g., compact neural networks. Sparsity is achieved in this form of regularization by penalizing the first norm of the weights. Formally this is given by

$$\min_{\Theta} \quad \mathcal{L} + \lambda \sum_{i=1}^{|\Theta|} |w_i|. \tag{2.8}$$

We will only follow a visualization approach to develop an understanding of how this technique works, for a formal analysis is a little advanced and beyond the scope of this course. Consider Figure 2.2b. The absolute function produces a uniform decay around zero; it pushes the weight to approach zero at a constant speed, and never slows down like the weight decay regularization. This results in many weights going too zero as the training progresses, and only those who has a strong impact on the loss (have quite large gradients) could escape the

[1]We usually want a balanced relation between Ω and $\mathcal{L}$, which means λ should not be quite large, and we would like to take very small steps when updating the parameters, which means η is very small.

push toward zero.

Bibliography

[1] S. Haykin, *Neural Networks and Learning Machines.* Prentice Hall, 2009. [Online]. Available: https://books.google.com.sa/books?id=K7P36lKzI_QC

[2] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* Cambridge, MA, USA: MIT Press, 2016, http://www.deeplearningbook.org.

[3] T. M. Cover and J. A. Thomas, *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing).* USA: Wiley-Interscience, 2006.

[4] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.