20

Neural Networks

An artificial neural network is a model of computation inspired by the structure of neural networks in the brain. In simplified models of the brain, it consists of a large number of basic computing devices (neurons) that are connected to each other in a complex communication network, through which the brain is able to carry out highly complex computations. Artificial neural networks are formal computation constructs that are modeled after this computation paradigm.

Learning with neural networks was proposed in the mid-20th century. It yields an effective learning paradigm and has recently been shown to achieve cutting-edge performance on several learning tasks.

A neural network can be described as a directed graph whose nodes correspond to neurons and edges correspond to links between them. Each neuron receives as input a weighted sum of the outputs of the neurons connected to its incoming edges. We focus on *feedforward* networks in which the underlying graph does not contain cycles.

In the context of learning, we can define a hypothesis class consisting of neural network predictors, where all the hypotheses share the underlying graph structure of the network and differ in the weights over edges. As we will show in Section 20.3, every predictor over n variables that can be implemented in time $T(n)$ can also be expressed as a neural network predictor of size $O(T(n)^2)$, where the size of the network is the number of nodes in it. It follows that the family of hypothesis classes of neural networks of polynomial size can suffice for all practical learning tasks, in which our goal is to learn predictors which can be implemented efficiently. Furthermore, in Section 20.4 we will show that the sample complexity of learning such hypothesis classes is also bounded in terms of the size of the network. Hence, it seems that this is the ultimate learning paradigm we would want to adapt, in the sense that it both has a polynomial sample complexity and has the minimal approximation error among all hypothesis classes consisting of efficiently implementable predictors.

The caveat is that the problem of training such hypothesis classes of neural network predictors is computationally hard. This will be formalized in Section 20.5.

A widely used heuristic for training neural networks relies on the SGD framework we studied in Chapter 14. There, we have shown that SGD is a successful learner if the loss function is convex. In neural networks, the loss function is highly nonconvex. Nevertheless, we can still implement the SGD algorithm and hope it will find a reasonable solution (as happens to be the case in several practical tasks). In Section 20.6 we describe how to implement SGD for neural networks. In particular, the most complicated operation is the calculation of the gradient of the loss function with respect to the parameters of the network. We present the *backpropagation* algorithm that efficiently calculates the gradient.

20.1 FEEDFORWARD NEURAL NETWORKS

The idea behind neural networks is that many neurons can be joined together by communication links to carry out complex computations. It is common to describe the structure of a neural network as a graph whose nodes are the neurons and each (directed) edge in the graph links the output of some neuron to the input of another neuron. We will restrict our attention to feedforward network structures in which the underlying graph does not contain cycles.

A feedforward neural network is described by a directed acyclic graph, $G = (V, E)$, and a weight function over the edges, $w : E \to \mathbb{R}$. Nodes of the graph correspond to neurons. Each single neuron is modeled as a simple scalar function, $\sigma : \mathbb{R} \to \mathbb{R}$. We will focus on three possible functions for σ: the sign function, $\sigma(a) = \text{sign}(a)$, the threshold function, $\sigma(a) = \mathbb{1}_{[a>0]}$, and the sigmoid function, $\sigma(a) = 1/(1+\exp(-a))$, which is a smooth approximation to the threshold function. We call σ the "activation" function of the neuron. Each edge in the graph links the output of some neuron to the input of another neuron. The input of a neuron is obtained by taking a weighted sum of the outputs of all the neurons connected to it, where the weighting is according to w.

To simplify the description of the calculation performed by the network, we further assume that the network is organized in *layers*. That is, the set of nodes can be decomposed into a union of (nonempty) disjoint subsets, $V = \cup_{t=0}^{T} V_t$, such that every edge in E connects some node in V_{t-1} to some node in V_t, for some $t \in [T]$. The bottom layer, V_0, is called the input layer. It contains $n+1$ neurons, where n is the dimensionality of the input space. For every $i \in [n]$, the output of neuron i in V_0 is simply x_i. The last neuron in V_0 is the "constant" neuron, which always outputs 1. We denote by $v_{t,i}$ the ith neuron of the tth layer and by $o_{t,i}(\mathbf{x})$ the output of $v_{t,i}$ when the network is fed with the input vector $\mathbf{x}$. Therefore, for $i \in [n]$ we have $o_{0,i}(\mathbf{x}) = x_i$ and for $i = n+1$ we have $o_{0,i}(\mathbf{x}) = 1$. We now proceed with the calculation in a layer by layer manner. Suppose we have calculated the outputs of the neurons at layer t. Then, we can calculate the outputs of the neurons at layer $t+1$ as follows. Fix some $v_{t+1,j} \in V_{t+1}$. Let $a_{t+1,j}(\mathbf{x})$ denote the input to $v_{t+1,j}$ when the network is fed with the input vector $\mathbf{x}$. Then,

$$a_{t+1,j}(\mathbf{x}) = \sum_{r:(v_{t,r},v_{t+1,j})\in E} w((v_{t,r}, v_{t+1,j}))\, o_{t,r}(\mathbf{x}),$$

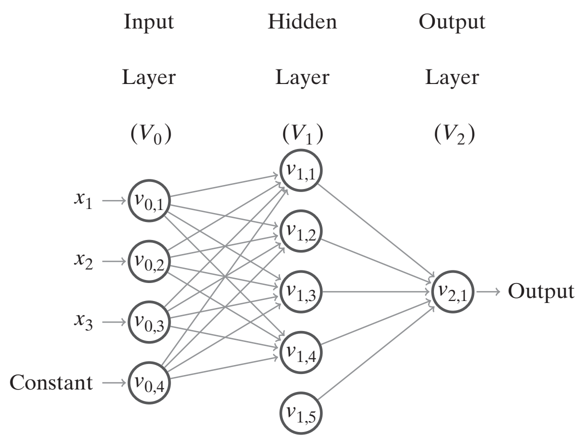
and

$$o_{t+1,j}(\mathbf{x}) = \sigma\left(a_{t+1,j}(\mathbf{x})\right).$$

That is, the input to $v_{t+1,j}$ is a weighted sum of the outputs of the neurons in V_t that are connected to $v_{t+1,j}$, where weighting is according to w, and the output of $v_{t+1,j}$ is simply the application of the activation function σ on its input.

Layers $V_1, \ldots, V_{T-1}$ are often called *hidden layers*. The top layer, V_T, is called the output layer. In simple prediction problems the output layer contains a single neuron whose output is the output of the network.

We refer to T as the number of layers in the network (excluding V_0), or the "depth" of the network. The size of the network is $|V|$. The "width" of the network is $\max_t |V_t|$. An illustration of a layered feedforward neural network of depth 2, size 10, and width 5, is given in the following. Note that there is a neuron in the hidden layer that has no incoming edges. This neuron will output the constant $\sigma(0)$.



20.2 LEARNING NEURAL NETWORKS

Once we have specified a neural network by (V, E, σ, w), we obtain a function $h_{V,E,\sigma,w} : \mathbb{R}^{|V_0|-1} \to \mathbb{R}^{|V_T|}$. Any set of such functions can serve as a hypothesis class for learning. Usually, we define a hypothesis class of neural network predictors by fixing the graph (V, E) as well as the activation function σ and letting the hypothesis class be all functions of the form $h_{V,E,\sigma,w}$ for some $w : E \to \mathbb{R}$. The triplet (V, E, σ) is often called the *architecture* of the network. We denote the hypothesis class by

$$\mathcal{H}_{V,E,\sigma} = \{h_{V,E,\sigma,w} : w \text{ is a mapping from } E \text{ to } \mathbb{R}\}. \tag{20.1}$$

That is, the parameters specifying a hypothesis in the hypothesis class are the weights over the edges of the network.

We can now study the approximation error, estimation error, and optimization error of such hypothesis classes. In Section 20.3 we study the approximation error of $\mathcal{H}_{V,E,\sigma}$ by studying what type of functions hypotheses in $\mathcal{H}_{V,E,\sigma}$ can implement, in terms of the size of the underlying graph. In Section 20.4 we study the estimation error of $\mathcal{H}_{V,E,\sigma}$, for the case of binary classification (i.e., $V_T = 1$ and σ is the sign function), by analyzing its VC dimension. Finally, in Section 20.5 we show that it is computationally hard to learn the class $\mathcal{H}_{V,E,\sigma}$, even if the underlying graph is small, and in Section 20.6 we present the most commonly used heuristic for training $\mathcal{H}_{V,E,\sigma}$.

20.3 THE EXPRESSIVE POWER OF NEURAL NETWORKS

In this section we study the expressive power of neural networks, namely, what type of functions can be implemented using a neural network. More concretely, we will fix some architecture, V, E, σ, and will study what functions hypotheses in $\mathcal{H}_{V,E,\sigma}$ can implement, as a function of the size of V.

We start the discussion with studying which type of Boolean functions (i.e., functions from $\{\pm 1\}^n$ to $\{\pm 1\}$) can be implemented by $\mathcal{H}_{V,E,\mathrm{sign}}$. Observe that for every computer in which real numbers are stored using b bits, whenever we calculate a function $f : \mathbb{R}^n \to \mathbb{R}$ on such a computer we in fact calculate a function $g : \{\pm 1\}^{nb} \to \{\pm 1\}^b$. Therefore, studying which Boolean functions can be implemented by $\mathcal{H}_{V,E,\mathrm{sign}}$ can tell us which functions can be implemented on a computer that stores real numbers using b bits.

We begin with a simple claim, showing that without restricting the size of the network, every Boolean function can be implemented using a neural network of depth 2.

Claim 20.1. *For every n, there exists a graph (V, E) of depth 2, such that $\mathcal{H}_{V,E,sign}$ contains all functions from $\{\pm 1\}^n$ to $\{\pm 1\}$.*

Proof. We construct a graph with $|V_0| = n+1$, $|V_1| = 2^n + 1$, and $|V_2| = 1$. Let E be all possible edges between adjacent layers. Now, let $f : \{\pm 1\}^n \to \{\pm 1\}$ be some Boolean function. We need to show that we can adjust the weights so that the network will implement f. Let $\mathbf{u}_1, \ldots, \mathbf{u}_k$ be all vectors in $\{\pm 1\}^n$ on which f outputs 1. Observe that for every i and every $\mathbf{x} \in \{\pm 1\}^n$, if $\mathbf{x} \neq \mathbf{u}_i$ then $\langle \mathbf{x}, \mathbf{u}_i \rangle \le n - 2$ and if $\mathbf{x} = \mathbf{u}_i$ then $\langle \mathbf{x}, u_i \rangle = n$. It follows that the function $g_i(\mathbf{x}) = \mathrm{sign}(\langle \mathbf{x}, \mathbf{u}_i \rangle - n + 1)$ equals 1 if and only if $\mathbf{x} = \mathbf{u}_i$. It follows that we can adapt the weights between V_0 and V_1 so that for every $i \in [k]$, the neuron $v_{1,i}$ implements the function $g_i(\mathbf{x})$. Next, we observe that $f(\mathbf{x})$ is the disjunction of the functions $g_i(\mathbf{x})$, and therefore can be written as

$$f(\mathbf{x}) = \mathrm{sign}\left(\sum_{i=1}^{k} g_i(\mathbf{x}) + k - 1 \right),$$

which concludes our proof. □

The preceding claim shows that neural networks can implement any Boolean function. However, this is a very weak property, as the size of the resulting network might be exponentially large. In the construction given at the proof of Claim 20.1, the number of nodes in the hidden layer is exponentially large. This is not an artifact of our proof, as stated in the following theorem.

Theorem 20.2. *For every n, let $s(n)$ be the minimal integer such that there exists a graph (V, E) with $|V| = s(n)$ such that the hypothesis class $\mathcal{H}_{V,E,sign}$ contains all the functions from $\{0,1\}^n$ to $\{0,1\}$. Then, $s(n)$ is exponential in n. Similar results hold for $\mathcal{H}_{V,E,\sigma}$ where σ is the sigmoid function.*

Proof. Suppose that for some (V, E) we have that $\mathcal{H}_{V,E,\mathrm{sign}}$ contains all functions from $\{0,1\}^n$ to $\{0,1\}$. It follows that it can shatter the set of $m = 2^n$ vectors in $\{0,1\}^n$ and hence the VC dimension of $\mathcal{H}_{V,E,\mathrm{sign}}$ is 2^n. On the other hand, the VC dimension of $\mathcal{H}_{V,E,\mathrm{sign}}$ is bounded by $O(|E| \log(|E|)) \le O(|V|^3)$, as we will show in the next section. This implies that $|V| \geq \Omega(2^{n/3})$, which concludes our proof for the case of networks with the sign activation function. The proof for the sigmoid case is analogous. $\square$

Remark 20.1. It is possible to derive a similar theorem for $\mathcal{H}_{V,E,\sigma}$ for any σ, as long as we restrict the weights so that it is possible to express every weight using a number of bits which is bounded by a universal constant. We can even consider hypothesis classes where different neurons can employ different activation functions, as long as the number of allowed activation functions is also finite.

Which functions can we express using a network of polynomial size? The preceding claim tells us that it is impossible to express all Boolean functions using a network of polynomial size. On the positive side, in the following we show that all Boolean functions that can be calculated in time $O(T(n))$ can also be expressed by a network of size $O(T(n)^2)$.

Theorem 20.3. *Let $T : \mathbb{N} \to \mathbb{N}$ and for every n, let $\mathcal{F}_n$ be the set of functions that can be implemented using a Turing machine using runtime of at most $T(n)$. Then, there exist constants $b, c \in \mathbb{R}_+$ such that for every n, there is a graph (V_n, E_n) of size at most $c\,T(n)^2 + b$ such that $\mathcal{H}_{V_n,E_n,sign}$ contains $\mathcal{F}_n$.*

The proof of this theorem relies on the relation between the time complexity of programs and their circuit complexity (see, for example, Sipser (2006)). In a nutshell, a Boolean circuit is a type of network in which the individual neurons implement conjunctions, disjunctions, and negation of their inputs. Circuit complexity measures the size of Boolean circuits required to calculate functions. The relation between time complexity and circuit complexity can be seen intuitively as follows. We can model each step of the execution of a computer program as a simple operation on its memory state. Therefore, the neurons at each layer of the network will reflect the memory state of the computer at the corresponding time, and the translation to the next layer of the network involves a simple calculation that can be carried out by the network. To relate Boolean circuits to networks with the sign activation function, we need to show that we can implement the operations of conjunction, disjunction, and negation, using the sign activation function. Clearly, we can implement the negation operator using the sign activation function. The following lemma shows that the sign activation function can also implement conjunctions and disjunctions of its inputs.

Lemma 20.4. *Suppose that a neuron v, that implements the sign activation function, has k incoming edges, connecting it to neurons whose outputs are in $\{\pm 1\}$. Then, by adding one more edge, linking a "constant" neuron to v, and by adjusting the weights on the edges to v, the output of v can implement the conjunction or the disjunction of its inputs.*

Proof. Simply observe that if $f : \{\pm 1\}^k \to \{\pm 1\}$ is the conjunction function, $f(\mathbf{x}) = \wedge_i x_i$, then it can be written as $f(\mathbf{x}) = \text{sign}\left(1 - k + \sum_{i=1}^k x_i\right)$. Similarly, the disjunction function, $f(\mathbf{x}) = \vee_i x_i$, can be written as $f(\mathbf{x}) = \text{sign}\left(k - 1 + \sum_{i=1}^k x_i\right)$. $\square$

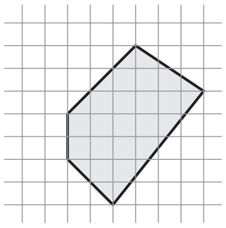
So far we have discussed Boolean functions. In Exercise 20.1 we show that neural networks are *universal approximators*. That is, for every fixed precision parameter, $\epsilon > 0$, and every Lipschitz function $f : [-1,1]^n \to [-1,1]$, it is possible to construct a network such that for every input $\mathbf{x} \in [-1,1]^n$, the network outputs a number between $f(\mathbf{x}) - \epsilon$ and $f(\mathbf{x}) + \epsilon$. However, as in the case of Boolean functions, the size of the network here again cannot be polynomial in n. This is formalized in the following theorem, whose proof is a direct corollary of Theorem 20.2 and is left as an exercise.

Theorem 20.5. *Fix some $\epsilon \in (0,1)$. For every n, let $s(n)$ be the minimal integer such that there exists a graph (V,E) with $|V| = s(n)$ such that the hypothesis class $\mathcal{H}_{V,E,\sigma}$, with σ being the sigmoid function, can approximate, to within precision of ϵ, every 1-Lipschitz function $f : [-1,1]^n \to [-1,1]$. Then $s(n)$ is exponential in n.*

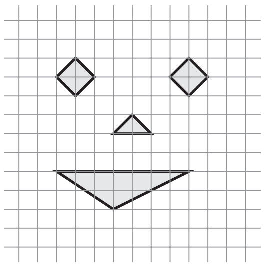
20.3.1 Geometric Intuition

We next provide several geometric illustrations of functions $f : \mathbb{R}^2 \to \{\pm 1\}$ and show how to express them using a neural network with the sign activation function.

Let us start with a depth 2 network, namely, a network with a single hidden layer. Each neuron in the hidden layer implements a halfspace predictor. Then, the single neuron at the output layer applies a halfspace on top of the binary outputs of the neurons in the hidden layer. As we have shown before, a halfspace can implement the conjunction function. Therefore, such networks contain all hypotheses which are an intersection of $k-1$ halfspaces, where k is the number of neurons in the hidden layer; namely, they can express all convex polytopes with $k-1$ faces. An example of an intersection of 5 halfspaces is given in the following.



We have shown that a neuron in layer V_2 can implement a function that indicates whether $\mathbf{x}$ is in some convex polytope. By adding one more layer, and letting the neuron in the output layer implement the disjunction of its inputs, we get a network that computes the union of polytopes. An illustration of such a function is given in the following.



20.4 THE SAMPLE COMPLEXITY OF NEURAL NETWORKS

Next we discuss the sample complexity of learning the class $\mathcal{H}_{V,E,\sigma}$. Recall that the fundamental theorem of learning tells us that the sample complexity of learning a hypothesis class of binary classifiers depends on its VC dimension. Therefore, we focus on calculating the VC dimension of hypothesis classes of the form $\mathcal{H}_{V,E,\sigma}$, where the output layer of the graph contains a single neuron.

We start with the sign activation function, namely, with $\mathcal{H}_{V,E,\text{sign}}$. What is the VC dimension of this class? Intuitively, since we learn $|E|$ parameters, the VC dimension should be order of $|E|$. This is indeed the case, as formalized by the following theorem.

Theorem 20.6. *The VC dimension of $\mathcal{H}_{V,E,sign}$ is $O(|E|\log(|E|))$.*

Proof. To simplify the notation throughout the proof, let us denote the hypothesis class by $\mathcal{H}$. Recall the definition of the growth function, $\tau_{\mathcal{H}}(m)$, from Section 6.5.1. This function measures $\max_{C\subset\mathcal{X}:|C|=m}|\mathcal{H}_C|$, where $\mathcal{H}_C$ is the restriction of $\mathcal{H}$ to functions from C to $\{0,1\}$. We can naturally extend the definition for a set of functions from $\mathcal{X}$ to some finite set $\mathcal{Y}$, by letting $\mathcal{H}_C$ be the restriction of $\mathcal{H}$ to functions from C to $\mathcal{Y}$, and keeping the definition of $\tau_{\mathcal{H}}(m)$ intact.

Our neural network is defined by a layered graph. Let $V_0,\ldots,V_T$ be the layers of the graph. Fix some $t\in[T]$. By assigning different weights on the edges between V_{t-1} and V_t, we obtain different functions from $\mathbb{R}^{|V_{t-1}|}\to\{\pm1\}^{|V_t|}$. Let $\mathcal{H}^{(t)}$ be the class of all possible such mappings from $\mathbb{R}^{|V_{t-1}|}\to\{\pm1\}^{|V_t|}$. Then, $\mathcal{H}$ can be written as a composition, $\mathcal{H}=\mathcal{H}^{(T)}\circ\ldots\circ\mathcal{H}^{(1)}$. In Exercise 20.4 we show that the growth function of a composition of hypothesis classes is bounded by the products of the growth functions of the individual classes. Therefore,

$$\tau_{\mathcal{H}}(m)\le\prod_{t=1}^{T}\tau_{\mathcal{H}^{(t)}}(m).$$

In addition, each $\mathcal{H}^{(t)}$ can be written as a product of function classes, $\mathcal{H}^{(t)}=\mathcal{H}^{(t,1)}\times\cdots\times\mathcal{H}^{(t,|V_t|)}$, where each $\mathcal{H}^{(t,j)}$ is all functions from layer $t-1$ to $\{\pm1\}$ that the jth neuron of layer t can implement. In Exercise 20.3 we bound product classes, and this yields

$$\tau_{\mathcal{H}^{(t)}}(m)\le\prod_{i=1}^{|V_t|}\tau_{\mathcal{H}^{(t,i)}}(m).$$

Let $d_{t,i}$ be the number of edges that are headed to the ith neuron of layer t. Since the neuron is a homogenous halfspace hypothesis and the VC dimension of homogenous halfspaces is the dimension of their input, we have by Sauer's lemma that

$$\tau_{\mathcal{H}^{(t,i)}}(m)\le\left(\frac{em}{d_{t,i}}\right)^{d_{t,i}}\le(em)^{d_{t,i}}.$$

Overall, we obtained that

$$\tau_{\mathcal{H}}(m)\le(em)^{\sum_{t,i}d_{t,i}}=(em)^{|E|}.$$

Now, assume that there are m shattered points. Then, we must have $\tau_{\mathcal{H}}(m) = 2^m$, from which we obtain

$$2^m \leq (em)^{|E|} \;\Rightarrow\; m \leq |E| \log(em)/\log(2).$$

The claim follows by Lemma A.2. $\square$

Next, we consider $\mathcal{H}_{V,E,\sigma}$, where σ is the sigmoid function. Surprisingly, it turns out that the VC dimension of $\mathcal{H}_{V,E,\sigma}$ is lower bounded by $\Omega(|E|^2)$ (see Exercise 20.5.) That is, the VC dimension is the number of tunable parameters squared. It is also possible to upper bound the VC dimension by $O(|V|^2\,|E|^2)$, but the proof is beyond the scope of this book. In any case, since in practice we only consider networks in which the weights have a short representation as floating point numbers with $O(1)$ bits, by using the discretization trick we easily obtain that such networks have a VC dimension of $O(|E|)$, even if we use the sigmoid activation function.

20.5 THE RUNTIME OF LEARNING NEURAL NETWORKS

In the previous sections we have shown that the class of neural networks with an underlying graph of polynomial size can express all functions that can be implemented efficiently, and that the sample complexity has a favorable dependence on the size of the network. In this section we turn to the analysis of the time complexity of training neural networks.

We first show that it is NP hard to implement the ERM rule with respect to $\mathcal{H}_{V,E,\text{sign}}$ even for networks with a single hidden layer that contain just 4 neurons in the hidden layer.

Theorem 20.7. *Let $k \geq 3$. For every n, let (V,E) be a layered graph with n input nodes, $k+1$ nodes at the (single) hidden layer, where one of them is the constant neuron, and a single output node. Then, it is NP hard to implement the ERM rule with respect to $\mathcal{H}_{V,E,sign}$.*

The proof relies on a reduction from the k-coloring problem and is left as Exercise 20.6.

One way around the preceding hardness result could be that for the purpose of learning, it may suffice to find a predictor $h \in \mathcal{H}$ with low empirical error, not necessarily an exact ERM. However, it turns out that even the task of finding weights that result in close-to-minimal empirical error is computationally infeasible (see (Bartlett & Ben-David 2002)).

One may also wonder whether it may be possible to change the architecture of the network so as to circumvent the hardness result. That is, maybe ERM with respect to the original network structure is computationally hard but ERM with respect to some other, larger, network may be implemented efficiently (see Chapter 8 for examples of such cases). Another possibility is to use other activation functions (such as sigmoids, or any other type of efficiently computable activation functions). There is a strong indication that all of such approaches are doomed to fail. Indeed, under some cryptographic assumption, the problem of learning intersections of halfspaces is known to be hard even in the representation independent model of learning (see Klivans & Sherstov (2006)). This implies that, under the same cryptographic assumption, any hypothesis class which contains intersections of halfspaces cannot be learned efficiently.

A widely used heuristic for training neural networks relies on the SGD framework we studied in Chapter 14. There, we have shown that SGD is a successful learner if the loss function is convex. In neural networks, the loss function is highly nonconvex. Nevertheless, we can still implement the SGD algorithm and hope it will find a reasonable solution (as happens to be the case in several practical tasks).

20.6 SGD AND BACKPROPAGATION

The problem of finding a hypothesis in $\mathcal{H}_{V,E,\sigma}$ with a low risk amounts to the problem of tuning the weights over the edges. In this section we show how to apply a heuristic search for good weights using the SGD algorithm. Throughout this section we assume that σ is the sigmoid function, $\sigma(a) = 1/(1+e^{-a})$, but the derivation holds for any differentiable scalar function.

Since E is a finite set, we can think of the weight function as a vector $\mathbf{w} \in \mathbb{R}^{|E|}$. Suppose the network has n input neurons and k output neurons, and denote by $h_{\mathbf{w}} : \mathbb{R}^n \to \mathbb{R}^k$ the function calculated by the network if the weight function is defined by $\mathbf{w}$. Let us denote by $\Delta(h_{\mathbf{w}}(\mathbf{x}), \mathbf{y})$ the loss of predicting $h_{\mathbf{w}}(\mathbf{x})$ when the target is $\mathbf{y} \in \mathcal{Y}$. For concreteness, we will take Δ to be the squared loss, $\Delta(h_{\mathbf{w}}(\mathbf{x}), y) = \frac{1}{2}\|h_{\mathbf{w}}(\mathbf{x}) - \mathbf{y}\|^2$; however, similar derivation can be obtained for every differentiable function. Finally, given a distribution $\mathcal{D}$ over the examples domain, $\mathbb{R}^n \times \mathbb{R}^k$, let $L_{\mathcal{D}}(\mathbf{w})$ be the risk of the network, namely,

$$L_{\mathcal{D}}(\mathbf{w}) = \mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}} \left[\Delta(h_{\mathbf{w}}(\mathbf{x}), \mathbf{y})\right].$$

Recall the SGD algorithm for minimizing the risk function $L_{\mathcal{D}}(\mathbf{w})$. We repeat the pseudocode from Chapter 14 with a few modifications, which are relevant to the neural network application because of the nonconvexity of the objective function. First, while in Chapter 14 we initialized $\mathbf{w}$ to be the zero vector, here we initialize $\mathbf{w}$ to be a randomly chosen vector with values close to zero. This is because an initialization with the zero vector will lead all hidden neurons to have the same weights (if the network is a full layered network). In addition, the hope is that if we repeat the SGD procedure several times, where each time we initialize the process with a new random vector, one of the runs will lead to a good local minimum. Second, while a fixed step size, η, is guaranteed to be good enough for convex problems, here we utilize a variable step size, η_t, as defined in Section 14.4.2. Because of the nonconvexity of the loss function, the choice of the sequence η_t is more significant, and it is tuned in practice by a trial and error manner. Third, we output the best performing vector on a validation set. In addition, it is sometimes helpful to add regularization on the weights, with parameter λ. That is, we try to minimize $L_{\mathcal{D}}(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$. Finally, the gradient does not have a closed form solution. Instead, it is implemented using the backpropagation algorithm, which will be described in the sequel.

SGD for Neural Networks

```
parameters:
  number of iterations $\tau$
  step size sequence $\eta_1, \eta_2, \ldots, \eta_\tau$
  regularization parameter $\lambda > 0$
input:
  layered graph $(V, E)$
  differentiable activation function $\sigma : \mathbb{R} \to \mathbb{R}$
initialize:
  choose $\mathbf{w}^{(1)} \in \mathbb{R}^{|E|}$ at random
    (from a distribution s.t. $\mathbf{w}^{(1)}$ is close enough to $\mathbf{0}$)
for $i = 1, 2, \ldots, \tau$
  sample $(\mathbf{x}, \mathbf{y}) \sim \mathcal{D}$
  calculate gradient $\mathbf{v}_i = \mathrm{backpropagation}(\mathbf{x}, \mathbf{y}, \mathbf{w}, (V, E), \sigma)$
  update $\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \eta_i(\mathbf{v}_i + \lambda \mathbf{w}^{(i)})$
output:
  $\bar{\mathbf{w}}$ is the best performing $\mathbf{w}^{(i)}$ on a validation set
```

Backpropagation

```
input:
  example $(\mathbf{x}, \mathbf{y})$, weight vector $\mathbf{w}$, layered graph $(V, E)$,
  activation function $\sigma : \mathbb{R} \to \mathbb{R}$
initialize:
  denote layers of the graph $V_0, \ldots, V_T$ where $V_t = \{v_{t,1}, \ldots, v_{t,k_t}\}$
  define $W_{t,i,j}$ as the weight of $(v_{t,j}, v_{t+1,i})$
    (where we set $W_{t,i,j} = 0$ if $(v_{t,j}, v_{t+1,i}) \notin E$)
forward:
  set $\mathbf{o}_0 = \mathbf{x}$
  for $t = 1, \ldots, T$
    for $i = 1, \ldots, k_t$
      set $a_{t,i} = \sum_{j=1}^{k_{t-1}} W_{t-1,i,j}\, o_{t-1,j}$
      set $o_{t,i} = \sigma(a_{t,i})$
backward:
  set $\boldsymbol{\delta}_T = \mathbf{o}_T - \mathbf{y}$
  for $t = T-1, T-2, \ldots, 1$
    for $i = 1, \ldots, k_t$
      $\delta_{t,i} = \sum_{j=1}^{k_{t+1}} W_{t,j,i}\, \delta_{t+1,j}\, \sigma'(a_{t+1,j})$
output:
  foreach edge $(v_{t-1,j}, v_{t,i}) \in E$
    set the partial derivative to $\delta_{t,i}\, \sigma'(a_{t,i})\, o_{t-1,j}$
```

Explaining How Backpropagation Calculates the Gradient:
We next explain how the backpropagation algorithm calculates the gradient of the loss function on an example $(\mathbf{x}, \mathbf{y})$ with respect to the vector $\mathbf{w}$. Let us first recall

a few definitions from vector calculus. Each element of the gradient is the partial derivative with respect to the variable in $\mathbf{w}$ corresponding to one of the edges of the network. Recall the definition of a partial derivative. Given a function $f : \mathbb{R}^n \to \mathbb{R}$, the partial derivative with respect to the ith variable at $\mathbf{w}$ is obtained by fixing the values of $w_1, \ldots, w_{i-1}, w_{i+1}, w_n$, which yields the scalar function $g : \mathbb{R} \to \mathbb{R}$ defined by $g(a) = f((w_1, \ldots, w_{i-1}, w_i + a, w_{i+1}, \ldots, w_n))$, and then taking the derivative of g at 0. For a function with multiple outputs, $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, the *Jacobian* of $\mathbf{f}$ at $\mathbf{w} \in \mathbb{R}^n$, denoted $J_{\mathbf{w}}(\mathbf{f})$, is the $m \times n$ matrix whose i, j element is the partial derivative of $f_i : \mathbb{R}^n \to \mathbb{R}$ w.r.t. its jth variable at $\mathbf{w}$. Note that if $m = 1$ then the Jacobian matrix is the gradient of the function (represented as a row vector). Two examples of Jacobian calculations, which we will later use, are as follows.

- Let $\mathbf{f}(\mathbf{w}) = A\mathbf{w}$ for $A \in \mathbb{R}^{m,n}$. Then $J_{\mathbf{w}}(\mathbf{f}) = A$.
- For every n, we use the notation $\boldsymbol{\sigma}$ to denote the function from $\mathbb{R}^n$ to $\mathbb{R}^n$ which applies the sigmoid function element-wise. That is, $\boldsymbol{\alpha} = \boldsymbol{\sigma}(\boldsymbol{\theta})$ means that for every i we have $\alpha_i = \sigma(\theta_i) = \frac{1}{1+\exp(-\theta_i)}$. It is easy to verify that $J_{\boldsymbol{\theta}}(\boldsymbol{\sigma})$ is a diagonal matrix whose (i,i) entry is $\sigma'(\theta_i)$, where σ' is the derivative function of the (scalar) sigmoid function, namely, $\sigma'(\theta_i) = \frac{1}{(1+\exp(\theta_i))(1+\exp(-\theta_i))}$. We also use the notation $\text{diag}(\boldsymbol{\sigma}'(\boldsymbol{\theta}))$ to denote this matrix.

The *chain rule* for taking the derivative of a composition of functions can be written in terms of the Jacobian as follows. Given two functions $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ and $\mathbf{g} : \mathbb{R}^k \to \mathbb{R}^n$, we have that the Jacobian of the composition function, $(\mathbf{f} \circ \mathbf{g}) : \mathbb{R}^k \to \mathbb{R}^m$, at $\mathbf{w}$, is

$$J_{\mathbf{w}}(\mathbf{f} \circ \mathbf{g}) = J_{g(\mathbf{w})}(\mathbf{f}) J_{\mathbf{w}}(\mathbf{g}).$$

For example, for $\mathbf{g}(\mathbf{w}) = A\mathbf{w}$, where $A \in \mathbb{R}^{n,k}$, we have that

$$J_{\mathbf{w}}(\boldsymbol{\sigma} \circ \mathbf{g}) = \text{diag}(\boldsymbol{\sigma}'(A\mathbf{w}))\, A.$$

To describe the backpropagation algorithm, let us first decompose V into the layers of the graph, $V = \cup_{t=0}^{T} V_t$. For every t, let us write $V_t = \{v_{t,1}, \ldots, v_{t,k_t}\}$, where $k_t = |V_t|$. In addition, for every t denote $W_t \in \mathbb{R}^{k_{t+1}, k_t}$ a matrix which gives a weight to every potential edge between V_t and V_{t+1}. If the edge exists in E then we set $W_{t,i,j}$ to be the weight, according to $\mathbf{w}$, of the edge $(v_{t,j}, v_{t+1,i})$. Otherwise, we add a "phantom" edge and set its weight to be zero, $W_{t,i,j} = 0$. Since when calculating the partial derivative with respect to the weight of some edge we fix all other weights, these additional "phantom" edges have no effect on the partial derivative with respect to existing edges. It follows that we can assume, without loss of generality, that all edges exist, that is, $E = \cup_t (V_t \times V_{t+1})$.

Next, we discuss how to calculate the partial derivatives with respect to the edges from V_{t-1} to V_t, namely, with respect to the elements in W_{t-1}. Since we fix all other weights of the network, it follows that the outputs of all the neurons in V_{t-1} are fixed numbers which do not depend on the weights in W_{t-1}. Denote the corresponding vector by $\mathbf{o}_{t-1}$. In addition, let us denote by $\ell_t : \mathbb{R}^{k_t} \to \mathbb{R}$ the loss function of the subnetwork defined by layers $V_t, \ldots, V_T$ as a function of the outputs of the neurons in V_t. The input to the neurons of V_t can be written as $\mathbf{a}_t = W_{t-1}\mathbf{o}_{t-1}$ and the output of the neurons of V_t is $\mathbf{o}_t = \boldsymbol{\sigma}(\mathbf{a}_t)$. That is, for every j we have $o_{t,j} = \sigma(a_{t,j})$. We obtain that the loss, as a function of W_{t-1}, can be written as

$$g_t(W_{t-1}) = \ell_t(\mathbf{o}_t) = \ell_t(\boldsymbol{\sigma}(\mathbf{a}_t)) = \ell_t(\boldsymbol{\sigma}(W_{t-1}\mathbf{o}_{t-1})).$$

It would be convenient to rewrite this as follows. Let $\mathbf{w}_{t-1} \in \mathbb{R}^{k_{t-1}k_t}$ be the column vector obtained by concatenating the rows of W_{t-1} and then taking the transpose of the resulting long vector. Define by O_{t-1} the $k_t \times (k_{t-1}k_t)$ matrix

$$O_{t-1} = \begin{pmatrix} \mathbf{o}_{t-1}^\top & 0 & \cdots & 0 \\ 0 & \mathbf{o}_{t-1}^\top & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{o}_{t-1}^\top \end{pmatrix}. \tag{20.2}$$

Then, $W_{t-1}\mathbf{o}_{t-1} = O_{t-1}\mathbf{w}_{t-1}$, so we can also write

$$g_t(\mathbf{w}_{t-1}) = \ell_t(\boldsymbol{\sigma}(O_{t-1}\,\mathbf{w}_{t-1})).$$

Therefore, applying the chain rule, we obtain that

$$J_{\mathbf{w}_{t-1}}(g_t) = J_{\boldsymbol{\sigma}(O_{t-1}\mathbf{w}_{t-1})}(\ell_t)\,\mathrm{diag}(\boldsymbol{\sigma}'(O_{t-1}\mathbf{w}_{t-1}))\,O_{t-1}.$$

Using our notation we have $\mathbf{o}_t = \boldsymbol{\sigma}(O_{t-1}\mathbf{w}_{t-1})$ and $\mathbf{a}_t = O_{t-1}\mathbf{w}_{t-1}$, which yields

$$J_{\mathbf{w}_{t-1}}(g_t) = J_{\mathbf{o}_t}(\ell_t)\,\mathrm{diag}(\boldsymbol{\sigma}'(\mathbf{a}_t))\,O_{t-1}.$$

Let us also denote $\boldsymbol{\delta}_t = J_{\mathbf{o}_t}(\ell_t)$. Then, we can further rewrite the preceding as

$$J_{\mathbf{w}_{t-1}}(g_t) = \left(\delta_{t,1}\,\sigma'(a_{t,1})\,\mathbf{o}_{t-1}^\top\,,\ \ldots\,,\ \delta_{t,k_t}\,\sigma'(a_{t,k_t})\,\mathbf{o}_{t-1}^\top\right). \tag{20.3}$$

It is left to calculate the vector $\boldsymbol{\delta}_t = J_{\mathbf{o}_t}(\ell_t)$ for every t. This is the gradient of ℓ_t at $\mathbf{o}_t$. We calculate this in a recursive manner. First observe that for the last layer we have that $\ell_T(\mathbf{u}) = \Delta(\mathbf{u},\mathbf{y})$, where Δ is the loss function. Since we assume that $\Delta(\mathbf{u},\mathbf{y}) = \frac{1}{2}\|\mathbf{u}-\mathbf{y}\|^2$ we obtain that $J_{\mathbf{u}}(\ell_T) = (\mathbf{u}-\mathbf{y})$. In particular, $\boldsymbol{\delta}_T = J_{\mathbf{o}_T}(\ell_T) = (\mathbf{o}_T - \mathbf{y})$. Next, note that

$$\ell_t(\mathbf{u}) = \ell_{t+1}(\boldsymbol{\sigma}(W_t\mathbf{u})).$$

Therefore, by the chain rule,

$$J_{\mathbf{u}}(\ell_t) = J_{\boldsymbol{\sigma}(W_t\mathbf{u})}(\ell_{t+1})\mathrm{diag}(\boldsymbol{\sigma}'(W_t\mathbf{u}))W_t.$$

In particular,

$$\begin{aligned}\boldsymbol{\delta}_t = J_{\mathbf{o}_t}(\ell_t) &= J_{\boldsymbol{\sigma}(W_t\mathbf{o}_t)}(\ell_{t+1})\mathrm{diag}(\boldsymbol{\sigma}'(W_t\mathbf{o}_t))W_t \\ &= J_{\mathbf{o}_{t+1}}(\ell_{t+1})\mathrm{diag}(\boldsymbol{\sigma}'(\mathbf{a}_{t+1}))W_t \\ &= \boldsymbol{\delta}_{t+1}\,\mathrm{diag}(\boldsymbol{\sigma}'(\mathbf{a}_{t+1}))W_t.\end{aligned}$$

In summary, we can first calculate the vectors $\{\mathbf{a}_t,\mathbf{o}_t\}$ from the bottom of the network to its top. Then, we calculate the vectors $\{\boldsymbol{\delta}_t\}$ from the top of the network back to its bottom. Once we have all of these vectors, the partial derivatives are easily obtained using Equation (20.3). We have thus shown that the pseudocode of backpropagation indeed calculates the gradient.

20.7 SUMMARY

Neural networks over graphs of size $s(n)$ can be used to describe hypothesis classes of all predictors that can be implemented in runtime of $O(\sqrt{s(n)})$. We have also shown that their sample complexity depends polynomially on $s(n)$ (specifically, it depends on the number of edges in the network). Therefore, classes of neural network hypotheses seem to be an excellent choice. Regrettably, the problem of training the network on the basis of training data is computationally hard. We have presented the SGD framework as a heuristic approach for training neural networks and described the backpropagation algorithm which efficiently calculates the gradient of the loss function with respect to the weights over the edges.

20.8 BIBLIOGRAPHIC REMARKS

Neural networks were extensively studied in the 1980s and early 1990s, but with mixed empirical success. In recent years, a combination of algorithmic advancements, as well as increasing computational power and data size, has led to a breakthrough in the effectiveness of neural networks. In particular, "deep networks" (i.e., networks of more than 2 layers) have shown very impressive practical performance on a variety of domains. A few examples include convolutional networks (LeCun & Bengio 1995), restricted Boltzmann machines (Hinton, Osindero & Teh 2006), auto-encoders (Ranzato et al. 2007, Bengio & LeCun 2007, Collobert & Weston 2008, Lee et al. 2009, Le et al. 2012), and sum-product networks (Livni, Shalev-Shwartz & Shamir 2013, Poon & Domingos 2011). See also (Bengio 2009) and the references therein.

The expressive power of neural networks and the relation to circuit complexity have been extensively studied in (Parberry 1994). For the analysis of the sample complexity of neural networks we refer the reader to (Anthony & Bartlet 1999). Our proof technique of Theorem 20.6 is due to Kakade and Tewari lecture notes.

Klivans and Sherstov (2006) have shown that for any $c > 0$, intersections of n^c halfspaces over $\{\pm 1\}^n$ are not efficiently PAC learnable, even if we allow representation independent learning. This hardness result relies on the cryptographic assumption that there is no polynomial time solution to the unique-shortest-vector problem. As we have argued, this implies that there cannot be an efficient algorithm for training neural networks, even if we allow larger networks or other activation functions that can be implemented efficiently.

The backpropagation algorithm has been introduced in Rumelhart, Hinton, and Williams (1986).

20.9 EXERCISES

20.1 **Neural Networks are universal approximators:** Let $f : [-1,1]^n \to [-1,1]$ be a ρ-Lipschitz function. Fix some $\epsilon > 0$. Construct a neural network $N : [-1,1]^n \to [-1,1]$, with the sigmoid activation function, such that for every $\mathbf{x} \in [-1,1]^n$ it holds that $|f(\mathbf{x}) - N(\mathbf{x})| \le \epsilon$.
Hint: Similarly to the proof of Theorem 19.3, partition $[-1,1]^n$ into small boxes. Use the Lipschitzness of f to show that it is approximately constant at each box.

Finally, show that a neural network can first decide which box the input vector belongs to, and then predict the averaged value of f at that box.

20.2 Prove Theorem 20.5.
Hint: For every $f : \{-1,1\}^n \to \{-1,1\}$ construct a 1-Lipschitz function $g : [-1,1]^n \to [-1,1]$ such that if you can approximate g then you can express f.

20.3 **Growth function of product:** For $i = 1,2$, let $\mathcal{F}_i$ be a set of functions from $\mathcal{X}$ to $\mathcal{Y}_i$. Define $\mathcal{H} = \mathcal{F}_1 \times \mathcal{F}_2$ to be the Cartesian product class. That is, for every $f_1 \in \mathcal{F}_1$ and $f_2 \in \mathcal{F}_2$, there exists $h \in \mathcal{H}$ such that $h(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}))$. Prove that $\tau_{\mathcal{H}}(m) \leq \tau_{\mathcal{F}_1}(m)\,\tau_{\mathcal{F}_2}(m)$.

20.4 **Growth function of composition:** Let $\mathcal{F}_1$ be a set of functions from $\mathcal{X}$ to Z and let $\mathcal{F}_2$ be a set of functions from Z to $\mathcal{Y}$. Let $\mathcal{H} = \mathcal{F}_2 \circ \mathcal{F}_1$ be the composition class. That is, for every $f_1 \in \mathcal{F}_1$ and $f_2 \in \mathcal{F}_2$, there exists $h \in \mathcal{H}$ such that $h(\mathbf{x}) = f_2(f_1(\mathbf{x}))$. Prove that $\tau_{\mathcal{H}}(m) \leq \tau_{\mathcal{F}_2}(m)\tau_{\mathcal{F}_1}(m)$.

20.5 **VC of sigmoidal networks:** In this exercise we show that there is a graph (V,E) such that the VC dimension of the class of neural networks over these graphs with the sigmoid activation function is $\Omega(|E|^2)$. Note that for every $\epsilon > 0$, the sigmoid activation function can approximate the threshold activation function, $\mathbb{1}_{[\sum_i x_i]}$, up to accuracy ϵ. To simplify the presentation, throughout the exercise we assume that we can exactly implement the activation function $\mathbb{1}_{[\sum_i x_i > 0]}$ using a sigmoid activation function.
Fix some n.

1. Construct a network, N_1, with $O(n)$ weights, which implements a function from $\mathbb{R}$ to $\{0,1\}^n$ and satisfies the following property. For every $\mathbf{x} \in \{0,1\}^n$, if we feed the network with the real number $0.x_1x_2\ldots x_n$, then the output of the network will be $\mathbf{x}$.
   Hint: Denote $\alpha = 0.x_1x_2\ldots x_n$ and observe that $10^k\alpha - 0.5$ is at least 0.5 if $x_k = 1$ and is at most -0.3 if $x_k = -1$.
2. Construct a network, N_2, with $O(n)$ weights, which implements a function from $[n]$ to $\{0,1\}^n$ such that $N_2(i) = \mathbf{e}_i$ for all i. That is, upon receiving the input i, the network outputs the vector of all zeros except 1 at the i'th neuron.
3. Let $\alpha_1,\ldots,\alpha_n$ be n real numbers such that every α_i is of the form $0.a_1^{(i)}a_2^{(i)}\ldots a_n^{(i)}$, with $a_j^{(i)} \in \{0,1\}$. Construct a network, N_3, with $O(n)$ weights, which implements a function from $[n]$ to $\mathbb{R}$, and satisfies $N_2(i) = \alpha_i$ for every $i \in [n]$.
4. Combine N_1, N_3 to obtain a network that receives $i \in [n]$ and output $\mathbf{a}^{(i)}$.
5. Construct a network N_4 that receives $(i,j) \in [n] \times [n]$ and outputs $a_j^{(i)}$.
   Hint: Observe that the AND function over $\{0,1\}^2$ can be calculated using $O(1)$ weights.
6. Conclude that there is a graph with $O(n)$ weights such that the VC dimension of the resulting hypothesis class is n^2.

20.6 Prove Theorem 20.7.
Hint: The proof is similar to the hardness of learning intersections of halfspaces – see Exercise 32 in Chapter 8.