Deep Learning

January 15, 2024

Contents

1 Neural Networks

Definition 1.1. An *artificial neuron* (sometimes also called *perceptron*) is a function $f : \mathbb{R}^n \to \mathbb{R}$ given by

$$x \mapsto \rho(\langle x, w \rangle + b)$$

where:

- $w \in \mathbb{R}^n$, whose components are called the *weights* of the neuron
- $b \in \mathbb{R}$ is called the *bias*
- $\rho : \mathbb{R} \to \mathbb{R}$ is called the *activation function*.

1.2. There are several standard activation functions to consider. For example, we could simply use the identity function. In the literature this is called *linear activation function* and in this case a neuron is simply a linear map. Another example is the sigmoid function

$$\rho(x) = \frac{1}{1 - e^{-x}}$$

In fact, a single neuron with sigmoid activation function is a model for logistic regression. We have discussed this briefly in the exercise class but since we are already here, let's have a look.

1.3. Logistic regression is a binary classification algorithm, by definition we are learning the function

$$f(x;\theta) = p(y|\rho(\langle x, w\rangle + b))$$

Thus we are computing the conditional probability that we have the value y given θ. To use this as classification we simply put $y = 1$ if this probability is greater than $1/2$.

1.4. An artificial neuron is thus related to linear models for classification and regression. Now the idea of a neural network is to stack artificial neurons onto layers and compose them. More precisely, given $m \in \mathbb{N}$ artificial neurons of dimension n with activation function ρ, we obtain a map $F : \mathbb{R}^n \to \mathbb{R}^m$. Another way of viewing this map is to assemble the weights of the neurons into a *weight matrix* $W \in \mathbb{R}^n \times \mathbb{R}^m$ with $W = (w^1, \ldots, w^m)^T$. We may then look at the affine linear function

$$x \mapsto Wx + b$$

and applying the activation function component-wise gives the function $F : \mathbb{R}^n \to \mathbb{R}^m$.

Definition 1.5. A *neural network* is a composition of layers of artificial neurons.

1.6. Let's break down the definition. Since a neural network is a composition of layers, let's fix how many layers we have (thus how many compositions we take). So let $L \in \mathbb{N}$ be the number of layers. Thus for each $l = 1, \ldots, L$ we have a map

$$F_l : \mathbb{R}^{N_{l-1}} \to \mathbb{R}^{N_l}$$

Here, N_{l-1} is the dimension of the neurons of the l-th layer and N_l is the number of neurons in the layer. Then the neural network is the composition

$$R^n = R^{N_0} \xrightarrow{F_1} R^{N_1} \xrightarrow{F_2} \cdots \xrightarrow{F_l} \mathbb{R}^{N_l} = \mathbb{R}^m$$

1.7. Note that in principle each layer can have different activation functions. In practice, the hidden layers have the same activation function and this activation function is in general different from the activation function of the output layer. Depending on the task, there are several activation functions to consider. Here are some examples:

- identity function
- sigmoid function
- tanh
- relu
- softmax

1.8. Here is another way of thinking about neural networks. We can think of a neural network as being specified a directed graph $G = (V, E)$, an activation function $\rho : \mathbb{R} \to \mathbb{R}$ and weights for each edge of the graph $w : E \to \mathbb{R}$. Note that we consider only very special types of graphs for neural networks as specified by in our description above. Most notably, (V, E) is acyclic and the vertices can be decomposed into a disjoint union of layers $V = \coprod_{l=0}^{L} V_l$ with edges only going from V_{l-1} to V_l. We move forward through the neural network by taking the weighted sum over the vertices and applying the activation function.

1.9. V_0 is called the *input layer* and V_L is the *output layer*. The layers corresponding to V_l for $l = 1, \ldots, L-1$ are called *hidden layers*. The integer L is the *depth* of the neural network and this is what makes deep learning *deep*. For example, a neural network of depth 2 will have 1 input layer, 1 output layer and 1 hidden layer.

Definition 1.10. The tuple (G, ρ) is called the *architecture* of the neural network.

1.11. Having specified the architecture of the neural network, for each choice of weights $w : E \to \mathbb{R}$ we obtain a function

$$\Phi : \mathbb{R}^{|V_0|} \to \mathbb{R}^{|V_L|}$$

Thus, the class of neural networks of a given architecture defines a hypothesis class $\mathscr{H}_{(G,\rho)}$. Which we may study using the tools of statistical learning theory.

1.12. We collect some results for neural networks before we discuss how to train them. The first result concerns the expressive power of neural networks. Recall from the lectures on statistical learning theory that we can decompose the error of a prediction function $h \in \mathscr{H}$ as

$$L_D(h) = \varepsilon_{\text{app}} + \varepsilon_{\text{est}}$$

where $\varepsilon_{\text{app}} = \min_{h^* \in \mathscr{H}} L_D(h^*)$ and the estimation error is the difference between true error and approximation error. From the point of view of statistical learning theory, the higher the expressive power, the lower the approximation error. For example, neural networks are able to represent all boolean functions.

Theorem 1.13 (Boolean functions)**.** *For every $n \in \mathbb{N}$, there exists a graph G of depth 2, such that $\mathscr{H}_{G,sign}$ contains all boolean functions $\{0,1\}^n \to \{0,1\}$.*

1.14. Thus, if there is a boolean functional relation between samples and target, the class of neural networks has the realizability property. However,it can be shown that the number of nodes grows exponentially in n.

Example 1.15. As an example, let us show how to represent the XOR function by a neural network. We may treat this as a classification task.

1.16. One might ask if the class of neural networks is Turing complete. This doesn't seem to be the case, but we could not find a proof in the literature. However, if we are allowed to have loops in the graph representing the neural network, i.e. if we allow for more complex architectures, then one can show that neural networks are Turing complete for rational weights and more expressive for real weights. The class of such neural networks is called *recurrent neural networks* and we will come back to them later.

1.17. Feed forward neural networks are however *universal approximators*.

Theorem 1.18 (Universal approximation theorem)**.** *Let $f : [-1,1]^n \to [-1,1]$ be Lipschitz. Let $\varepsilon > 0$. Then there exists a neural network Φ with sigmoid activation function such that $|f(x) - \Phi(x)| \leq \varepsilon$.*

1.19. Although one can actually show that shallow neural networks are able to approximate any function as above, it turns out that in practice deep neural networks work better.

1.20. Finally, let us discuss the VC-dimension of neural networks. Recall that the VC-dimeonsion is defined for binary classification tasks. It is defined as the largest cardinality for which the hypothesis class is able to shatter a finite set of samples. The fundamental theorem of PAC-learning asserts that a hypothesis class is (agnostic) PAC-learnable if and only if it is of finite VC-dimension. In this case ERM is a successful PAC-learner. We will show that neural networks have finite VC-dimension. In order to do so, we need the notion of growth function.

Definition 1.21. Let $\mathscr{H}$ be a hypothesis class. The *growth function* of $\mathscr{H}$, denoted $\tau_{\mathscr{H}} : \mathbb{N} \to \mathbb{N}$, is defined as

$$m \mapsto \max_{C \subset X : |C| = m} |\mathscr{H}|_C|$$

1.22. Let us also call the definition of VC-dimension. It is the maximal cardinality of a subset $C \subset X$ that can be shattered by $\mathscr{H}$. In particular, if $VCdim(\mathscr{H}) = d$, then for any $m \leq d$ we have $\tau_{\mathscr{H}}(m) = 2^m$.

Theorem 1.23 (VC-dimension)**.** *The VC-dimension of $\mathscr{H}_{G,sign}$ is $\mathcal{O}(|E| \log(|E|))$.*

Sketch of Proof. We give a quick sketch of proof. It's easy to fill in the details yourself.

- Our neural network is defined by a layered graph with vertices $V_0, \ldots, V_L$. Fix some $l = 0, \ldots, L$. A choice of weights for the lth layer specifies a function

  $$\mathbb{R}^{|V_{l-1}|} \to \{-1,1\}^{|V_l|}$$

  We denote $\mathscr{H}^{(l)}$ the class of all such possible mappings. Then the class $\mathscr{H}$ is a composition $\mathscr{H}^{(L)} \circ \cdots \circ \mathscr{H}^{(1)}$.

- For the growth function we have

  $$\tau_{\mathscr{H}}(m) \leq \prod_{l=1}^{L} \tau_{\mathscr{H}^{(l)}}(m)$$

- Let $\mathscr{H}^{(l,i)}$ be the class of all function from layer $l-1$ to $\{-1,1\}$ that the ith neuron (of layer V_l can implement. We have

  $$\tau_{\mathscr{H}^{(l)}}(m) \leq \prod_{i=1}^{|V_l|} \tau_{\mathscr{H}^{(l,i)}}(m)$$

- (Sauers Lemma): $\tau_{\mathscr{H}}(m) \leq \sum_{i=0}^{d} \binom{m}{i}$. In particular if $m > d+1$ then $\tau_{\mathscr{H}}(m) \leq (em/d)^d$.
- Each neuron is a homogenous halfspace and their VC-dimension is equal to the number of inputs. Let $d_{l,i}$ be the number of edges with target the ith neuron of layer V_l. Thus by Sauers Lemma we have
$$\tau_{\mathscr{H}^{(l,i)}}(m) \leq (em)^{d_{l,i}}$$
- Overall we have
$$\tau_{\mathscr{H}}(m) \leq (em)^{|E|}$$
- Assuming we have m shattered points. Then we must have $\tau_{\mathscr{H}}(m) = 2^m$. Therefore
$$2^m \leq (em)^{|E|}$$
and hence
$$m \leq |E| log(em)/log(2)$$

□

1.24. The previous theorem shows that neural networks are PAC-learnable and that ERM will be a successful learner for the class of neural networks. However, it is in fact NP-hard to implement the ERM-rule for neural networks.

Theorem 1.25. *Let $k \geq 3$. For every $n \in \mathbb{N}$, 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 $\mathscr{H}_{V,E,sign}$.*

1.26. So, instead of trying to implement the ERM-rule for neural networks. One uses the stochastic gradient descent framework for training.

1.27. The conclusion here is that statistical learning theory seems to be inadequate to describe the performance of neural networks. For one, the universal approximation theorem tells us that we have a small approximation error. But this might come at the cost of having a lot of parameters. In this case the bounds from statistical learning theory seem to be vacous. For example, in practice one often has more parameters than than training samples (overparametrization) and at the same fit very closely to the training data so that one would expect overfitting, but this does not seem to be the case. In other words, neural networks seem to contradict the classical bias-variance tradeoff of statistical learning theory.

1.28. We conclude this section with a few empirical observations:

-

2 Geometric intuition for binary classification

2.1. We have previously seen that feed forward neural networks can implement all boolean functions. Let's look at a concrete implementation of the conjunction and disjunctin operation. This will help to get some geometric intuition on neural networks.

2.2. Let's start with the conjunction operation. It is a function

$$\{\pm 1\}^k \to \{\pm 1\}$$

and one easily checks that it has the form

$$x \mapsto sgn\left(1-k+\sum_{i=1}^{k} x_i\right)$$

thus can be implemented by a single neuron. The interpretation here is that this neuron implements the conjunction of its inputs. For example, let's consider a neural network of depth 2, where the hidden layer consists of k neurons with sign activation function and the output layer consists of the conjunction neuron just defined. Each neuron in the hidden layer implements a linear classification of its input represented by a hyperplane. Taking the conjunction then can be viewed as taking the intersection of these hyperplanes forming a polytope. The output of this neural network then classifies whether a point lies within the polytope or not.

2.3. Let's add another layer. We can implement the disjunction operation by

$$x \mapsto sgn\left(k-1+\sum_{i=1}^{k} x_i\right)$$

and can thus get a disjoint union of polytopes.

3 Training: Stochastic Gradient descent

3.1. We begin with a discussion on gradient descent. This is an algorithmGradient descent is ana first order algorithm for finding a local minimum of a real valued differentiable function. The idea is to follow along the negative gradient to reach a local minimum.

3.2. Notice that we have as output the sum of each iteration step. One would expect to output to be the last value or thi minimum over all the computed values. However, taking the average turns out to be useful especially for non-differentiable functions and stochastic gradient descent.

3.3. Recall that a real valued function with convex domain is called *convex*, if for all $x, y \in U$ and for all $0 \leq \lambda \leq 1$ we have

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$$

For convex functions we have the following result on the convergence of gradient descent.

Algorithm 1 Gradient descent

Parameters: integer $T > 0$, $\eta_1, \dots \eta_T \in \mathbb{R}$
initialize: $w^{(1)} = 0$
  for $t = 1, \dots, T$ **do**
    $w^{(t)} \leftarrow w^{(t-1)} - \eta_t \nabla f(w^{(t-1)})$
  end for

return : $w = \frac{1}{T} \sum^T w^{(t)}$

Theorem 3.4. *Let f be convex, differentiable and ρ-Lipschitz and let w^* be a minimum for f defined on a closed ball of radius B. If we run GD on f for T steps with $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$ then the output w satisfies*

$$|f(w) - f(w^*)| \leq \frac{B\rho}{\sqrt{T}}$$

Furthermore, to achieve an approximation of $\varepsilon > 0$ it suffices to run GD for a number of iterations that satisfies

$$T \geq \frac{B^2 \rho^2}{\varepsilon^2}$$

3.5. For non-differentiable functions one can use subgradients to perform the gradient descend algorithm. This is in parrticular applicable if we have a piecewise smooth/differentiable function such as ReLU.

Definition 3.6. Let D be a convex set and let $f : D \to \mathbb{R}$ be a function. A point $v \in D$ is called *subgradient* of f at a point $w \in D$ if and only if

$$\forall u \in D, \quad f(u) \geq f(w) + \langle u - w, v \rangle$$

The set of subgradients at w is called *differential set* and denoted by $\partial f(w)$.

3.7. The definition of subgradient comes from an alternative characterization of convexity. A function f is convex if and only if for every $w \in D$ there exists $v \in D$ satisfying the above equation.

3.8. In the gradient descend algorithm we can replace the gradient by an element of $\partial f(w)$. The analysis for convex functions does not change.

3.9. In fact, we don't even have to use the gradient or subgradient at all. We only need a *descend direction*, that is a sequence of vectors in which we move using the step-size/learning rate satisfying some conditions to ensure convergence.

Theorem 3.10. *Let $v_1, \dots, v_T$ be an arbitrary sequence of vectors. Any algorithm with initialization $w^{(1)} = 0$ and an update rule of the form*

$$w^{(t+1)} = w^{(t)} - \eta v_t$$

satisfies

$$\sum_{t=1}^{T}\langle w^{(t)}-w^*, v_t\rangle \leq \frac{\|w^*\|^2}{2\eta}+\frac{\eta}{2}\sum_{i=1}^{T}\|v_t\|^2$$

In particular, for every $B,\rho>0$, if for all t we have $\|v_t\|\leq\rho$ and if we set $\eta=\sqrt{\frac{B^2}{\rho^2 T}}$ then for every w^* with $\|w^*\|\leq B$ we have

$$\frac{1}{T}\sum_{i=1}^{T}\langle w^{(t)}-w^*, v_t\rangle \leq \frac{B\rho}{\sqrt{T}}$$

3.11. An important part is the *step size*, also called *learning rate* in machine learning. In the theorem above we have a constant step size η. Choosing the right step size is an important part and above we have a bound for which GD converges. However in practice, choosing the step size too small will result in slow convergence while too large may cause divergence (we can overshoot the local minimum).

3.12. We can instead use a variable stepsize to obtain a variant of GD. We get a similar bound by using $\eta_t=\frac{B}{\rho\sqrt{t}}$ in the theorem above for example. The idea to use a variable step size is to take more careful steps as we approach the minimum so as not to overshoot. This is in practice implemented in a *learning rate scheduler*.

3.13. There is a way to find the optimal step size called *line search*. We won't go into details here.

3.14. GD can move slowly in flat regions of the loss landscape. A solution is the *heavy ball* or *momentum* heuristic. This is for example implemented in the *Adam* optimizer of *Keras*. The idea is to move faster along directions that were previously good and slow down along directions where the gradient has suddenly changed. This can be implemented as follows.

$$m_t=\beta m_{t-1}+g_{t-1}$$
$$\theta_t=\theta_{t-1}-\eta_t m_t$$

where m_t is the momentum and $\beta\in(0,1)$. We see that

$$m_t=\sum_{\tau=0}^{t-1}\beta^\tau g_{t-\tau-1}$$

so the momentum is a weighted average of previous gradients. In particular, past gradients have an influence.

3.15. Finally, there is a variant called *Nesterov momentum*. We won't go into detail here.

3.16. This describes gradient descend. In practice we want to apply this to the true risk function. Thus this provides another learning paradigm in contrast to ERM. In particular, a statistical version of gradient descent is in general used for neural networks as ERM can't be implemented.

4 Training: Backpropagation

5 Other types of neural networks

5.1. Recurrent neural networks

5.2. Convolutional neural networks