Deep Learning: Basic Concepts

February 8, 2021

This note introduces the problems of classification and regression in machine learning, and then discusses issues related to the data that are necessary to train these predictors. Basic architectures of neural networks are described next, followed by a discussion of general principles underlying their training. Back-propagation (a specific training algorithm) and more task-specific neural network architectures will be discussed in subsequent notes.

1 Classification and Regression

Machine learning develops algorithms that discover patterns in data. Consider the following examples of two different types of supervised machine learning, classification and regression, drawn from computer vision.

Examples:

- **Classification:** The US Postal Service (USPS) uses digit recognition, a machine learning technique, to read handwritten ZIP codes on envelopes. Given the image X of a single digit, a *classifier* h outputs a *label* out of the set $Y = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ that tells what digit the image represents.
- **Regression:** A self-driving vehicle must know where potential obstacles are in its immediate surroundings. These obstacles include other vehicles, pedestrians, and any object that can potentially obstruct the vehicle's path. Given the position of an object in the image taken by one of the car's cameras at some point in time, a *tracker* h, which is an example of a *regressor*, outputs the object's position in a new image X taken a fraction of a second later. This position is a *value* out of a set $Y \subset \mathbb{R}^2$.

In both examples, the *predictor* is a function $h : X \to Y$ from the set X of all possible images that are relevant to the task (handwritten digits, traffic scenes). The difference between classification and regression is that h outputs a *categorical* variable for classification and a *real-valued* vector for regression. This means that the label set Y is finite and unstructured for classification, while for regression Y is an uncountable subset of $\mathbb{R}^e$ for some integer $e > 0$.

In terms of definitions, the difference between classification and regression is minor. In practice, however, the finite nature of the label set often suggests techniques for classification that could not be applied to regression. Conversely, regression techniques often rely on a *metric* defined in Y, and these techniques cannot be used for classification. The separation between regression and classification is not always crisp, as some machine learning algorithms can be adapted to either type of problem with minor changes. As we will see, this is the case for deep-learning methods.

2 The Training Set

To train the digit classifier, the USPS provides many thousands or even millions of digit images, and someone looks at each image and records a label for it. To train the object tracker, the car manufacturer provides hours of video. Once an object of interest is found in a frame of the video, someone draws a box around it in that frame and in subsequent ones.

The product of this painstaking work is a *training set* of the form

$$T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\} \quad \text{with} \quad \mathbf{x}_n \in X \quad \text{and} \quad y_n \in Y \ ,$$

where each *data point* $\mathbf{x}_n$ is an input to the predictor and y_n is the corresponding desired output. In the digit recognition example, $\mathbf{x}_n$ is an image and y_n is a digit label. In the tracking example, $\mathbf{x}_n$ is perhaps the new video frame together with the position of the object in the old frame, and y_n is the object's position in the new frame.

Each of the thousands or millions of ordered pairs $(\mathbf{x}_n, y_n)$ is called a *training sample.* The set X is the *data space* and the set Y is called the *label set* for a classifier and the *value set* for a regressor. When the distinction between classification and regression is ignored, the term "*target*" is sometimes used to denote either a label or a value.

3 Data Annotation

In the days of Google Images and web crawlers, collecting data for training is often but not always inexpensive. *Data annotation*, on the other hand, that is, the process of associating targets to data, is invariably a time consuming, expensive, and error-prone activity. The high cost of data annotation is arguably the main hurdle to a broader use of machine learning and especially of deep learning, whose predictors require large amounts of data to be trained.

For instance, to produce the images $\mathbf{x}_n$ for digit classification, one could place a camera above one of the conveyer belts that move envelopes around in a USPS distribution center, and take a picture of each envelope. That picture, however, contains much more than a digit, and someone must therefore sit at a computer, display each envelope image on the screen in turn, drag a bounding box around each digit with a mouse, save each cropped digit as a separate image file, and associate a label y_n to the file through some user interface.

Different images may have different quality depending on factors such as lighting, the ink used to write the digit, the color of the envelope paper, the size of the digit, and so forth. In addition, the digit images may be cropped inconsistently if the operator gets tired, or multiple operators are employed to crop and label images.

These inconsistencies often make machine learning harder, and training sets are sometimes *curated* because of this. In the USPS example, curation involves running manual procedures to make the images more uniform in terms of the factors mentioned above.[1] Curation helps machine learning research by making the problem easier for initial study. However, curation cannot be used to train real-life machine learning classifiers, because the whole point of computing h is to eliminate manual intervention during deployment (testing). In these cases, only automatic preprocessing of the images is a viable option, because the preprocessing can then be applied during both training and testing. Preprocessing techniques are beyond the scope of these notes, although by now you do know how to reduce noise by filtering or how to resize an image.

Similar complications occur for the object tracking example. In that case, the fact that positions are real-valued makes it even harder for a person to determine the correct position, and annotations for the same image by different people may be different. Even discrete labels are sometimes

[1]Think calibrating, touching up, and re-cropping or resizing images in Photoshop.

inconsistent across annotators, and various schemes have been devised to address this issue. For instance, multiple people are sometimes employed to annotate each training sample, and a majority vote (for classification) or a mean or median (for regression) then determines the "correct" target.

A common method for labeling large amounts of data is to publish it to the Amazon Mechanical Turk, an online marketplace for any type of repetitive work, including image labeling. People around the world access the marketplace searching for jobs they can perform and are paid a small amount per label. Recently, many companies have sprung up that offer labeling services for a fee. Others provide software platforms that make labeling easier for the user. An example of a labeling company is Hive and V7 Labs' Darwin is a sample annotation platform.

4 Deep Neural Networks

A deep neural network implements a function that takes an input $\mathbf{x}$ and outputs a target value[2] $\hat{y}$. Inputs are often images. Neural networks output (vectors of) real values, so they are a natural match for regression. When the desired output is a label, that is, a member of a discrete set, the network computes a *score* for each possible label. An $\arg\max$ operator at the output then returns the label with highest score. For instance, the network of a digit classifier computes a vector $\mathbf{p}$ of ten scores $p_0, \ldots, p_9$, one per digit, and the answer is

$$\hat{y} = \arg\max_k p_k \ ,$$

the digit that receives the highest score.

Because of this, networks for regression are essentially the same as networks for classification, except that the latter have an $\arg\max$ operation at their output[3].

Neural networks are *parametric*, in that their behavior depends on the values of a large number of *weights*, which are numerical parameters whose values are determined when a network is trained. In this Section we ignore training, and the values of the weights are therefore not determined. Principles for choosing their values are discussed in Section 5, while a specific training algorithm will be the subject of a later note.

Section 4.1 describes deep neural networks in general and Section 4.2 shows how to convert a regression network to a classifier. A later note describes deep *convolutional* neural networks, which are predominant in computer vision applications of deep learning.

4.1 The Generic Architecture of a Neural Network

A neural network is a cascade of *layers*, in which each layer's output is fed to the subsequent layer, and each layer is a set of *neurons*. The network is *deep* if it has many layers. We describe neurons first and layers and complete networks thereafter.

Neurons A *neuron* (in the computational sense) is a function $\mathbb{R}^d \to \mathbb{R}$ of the form

$$y = \rho(a(\mathbf{x})) \quad \text{where} \quad a = \mathbf{w}^T\tilde{\mathbf{x}} \quad , \quad \tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} \ .$$

[2]The hat distinguishes the estimate $\hat{y}$ from the true target y.

[3]The $\arg\max$ is usually not considered to be part of the network, but the literature is not consistent on this point.

The entries of the vector $\mathbf{w} \in \mathbb{R}^{d+1}$ are called the *weights*, and the *activation function* is a nonlinear and weakly monotonic function $\mathbb{R} \to \mathbb{R}$. The input $a(\mathbf{x})$ to ρ is called the *activation* of the neuron, and the particular type of activation function

$$\rho(a) = \max(0, a)$$

is called the *Rectified Linear Unit* (ReLU, Figure 1).

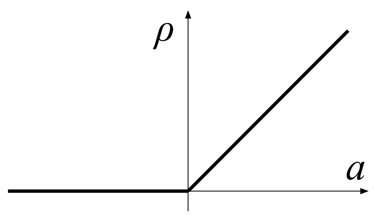


Figure 1: The Rectified Linear Unit (ReLU).

We view the tilde (as in $\tilde{\mathbf{x}}$) as an operator: Given any vector $\mathbf{x}$, this operator appends a 1 at the end of $\mathbf{x}$.

The activation can be rewritten as follows

$$a = \mathbf{v}^T\mathbf{x} + b \quad \text{where} \quad \mathbf{v}^T = [w_1, \ldots, w_d] \quad \text{and} \quad b = w_{d+1} \ ,$$

and is an inner product between a *gain*[4] vector $\mathbf{v}$ and the input $\mathbf{x}$, plus a *bias* b. Figure 2 shows a neuron in diagrammatic form.

For different inputs $\mathbf{x}$ of the same magnitude[5], the activation is maximum when $\mathbf{x}$ is parallel to $\mathbf{v}$, and the latter can be viewed as a *pattern* or *template* to which $\mathbf{x}$ is compared. The bias b then raises or lowers the activation before it is passed through the activation function.

The ReLU will *respond* (that is, return a nonzero output) if the inner product $\mathbf{v}^T\mathbf{x}$ is greater than $-b$ (so that a is positive), and the response thereafter increases with the value of a. So the negative of the bias can be viewed as a *threshold* that the inner product between pattern and input must exceed before it is deemed to be significant, and the neuron can be viewed as a *score function* that measures the similarity of the suitably normalized input $\mathbf{x}$ to the pattern $\mathbf{v}$ when the similarity is significant (that is, greater than $-b$). When the similarity is not significant, the neuron does not respond.

A *pattern classifier* would add a stage that decides if the score is large enough to declare the input $\mathbf{x}$ to contain the pattern represented by $\mathbf{v}$. So another way to view a neuron is a pattern classifier without the decision stage. This "interpretation" of a neuron has only psychological usefulness. Mathematically and even practically, all that really matters is that with enough layers with enough neurons one can implement very complex functions.

Layers A neural-net *layer* is a vector of e neurons, that is, a function $\mathbb{R}^d \to \mathbb{R}^e$

$$\mathbf{y} = \rho(\mathbf{a}(\mathbf{x})) \quad \text{where} \quad \mathbf{a}(\mathbf{x}) = W\tilde{\mathbf{x}} \ ,$$

the weight matrix W is $e \times (d+1)$, and the activation function ρ is applied to each entry of the activation vector $\mathbf{a}(\mathbf{x}) \in \mathbb{R}^e$. So a neural-net layer can be viewed as a *bank* of pattern scoring devices, one pattern per neuron. Figure 3 illustrates.

[4]Gains are often called weights as well.

[5]As measured by their Euclidean norm $\|\mathbf{x}\|$.

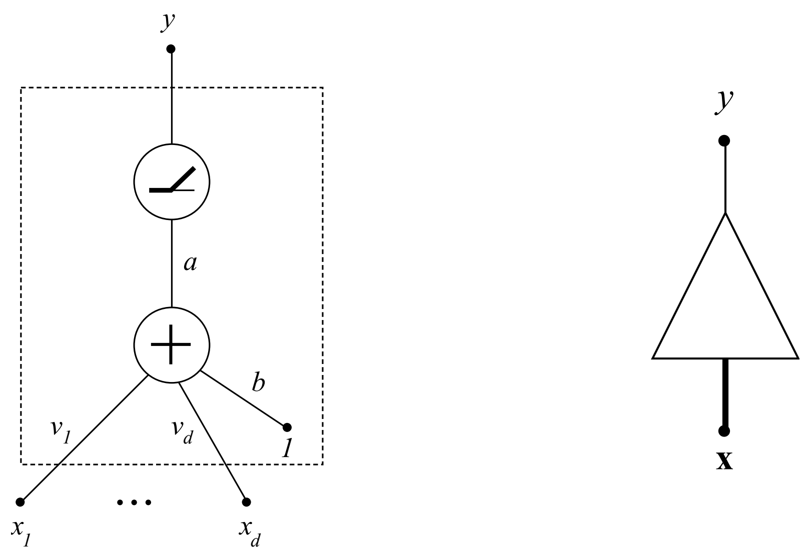


Figure 2: The internal structure of a neuron (left) and a neuron as a black box (right). The black box corresponds to the part inside the dashed rectangle on the left.

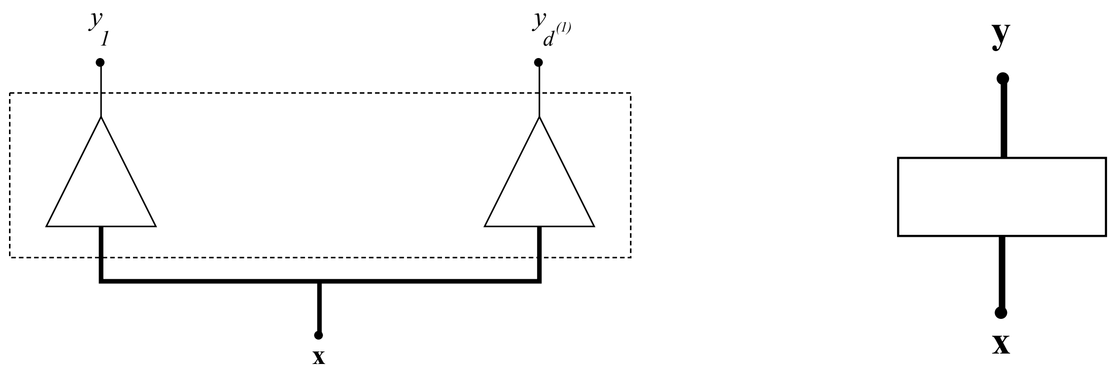


Figure 3: The internal structure of a layer (left) and a layer as a black box (right). The black box corresponds to the part inside the dashed rectangle on the left.

To compute the output of a layer from its input one needs to perform ed multiplications and as many additions to compute the activation vector, and then compute the activation function e times. So if e is of the same order of magnitude as d, the cost of this computation is quadratic in the size d of the input $\mathbf{x}$. Even more importantly, there are $O(d^2)$ parameters (the entries of W) that need to be determined when the layer is trained.

Networks A generic neural network is simply a cascade of layers, in which each layer takes the output of the previous layer as its input. Each neuron of each layer has weights (several gains and a bias), and it is useful to collect all the weights of layer number ℓ into a column vector $\mathbf{w}_\ell$. A network with L layers then has weight vector

$$\mathbf{w} = \begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_L \end{bmatrix} .$$

Training a neural network amounts to determining values for $\mathbf{w}$ so that the network's outputs are consistent with those specified in a *training set*, as discussed in general terms in Section 5. Consistency here entails that if $\hat{y}_n = h(\mathbf{x}_n; \mathbf{w})$ is the target estimate that the network computes on training data point $\mathbf{x}_n$, then $\hat{y}_n$ should be close, on average, to the true target y_n for that sample. A specific algorithm for training a network, called back-propagation, will be discussed in a later note.

4.2 Converting a Regression Network to a Classifier

The output from a deep CNN is fed to a computation that depends on the purpose of the net. For regression, for instance, the outputs may be used as they are. For classification, the last stage is typically chosen so that it outputs a vector of activations that has as many entries as there are labels in the label set Y. These activations are real numbers.

Training is easier to formulate, as we will see, if these numbers are positive and add up to one, so that it is easier to compare these numbers to each other during training. Because of this, the output from the last layer of a network that is to be used for classification is fed to a so-called *softmax* function. Let $\mathbf{p}$ be the network output. Then the softmax function is

$$z_k = \sigma_k(\mathbf{p}) = \frac{\exp(p_k)}{\sum_{j=1}^{K} \exp(p_j)} \quad \text{for } k = 1, \ldots, K$$

and where K is the number of labels in Y. The exponential makes all quantities positive, and normalization makes sure the entries of the output vector $\mathbf{z} = (z_1, \ldots, z_K)$ add up to 1. In this way, the entries of the softmax output can be viewed as *scores* for each of the categories, and the result of classification is then class

$$\hat{y} = h(\mathbf{x}; \mathbf{w}) = \arg\max_k z_k .$$

Note that the softmax function has no parameters. From the point of view of determining the highest activation, the softmax is irrelevant, since the exponential is a monotonically increasing function, so that

$$\arg\max_k z_k = \arg\max_k p_k .$$

Because of this, the softmax layer is typically removed after training.

5 Principles of Network Training

Deep neural networks have recently been shown empirically to achieve very good performance on tasks whose inputs are images, speech, or audio signals. They have also been applied to inputs of other types, with varied results. The reasons why these predictors work so well are still unclear. What *is* clear is that they are very expressive, in the sense that they can implement almost arbitrarily complex functions.

Specifically, so-called *universal approximation theorems* [2] show that *any* Lipschitz function[6] from a hypercube in $\mathbb{R}^d$ to a hypercube in $\mathbb{R}^e$ can be approximated arbitrarily closely (that is, within any pre-specified $\epsilon > 0$) with some neural net. As a consequence, if there exists a good predictor for a given task, there is likely *some* neural network that can implement that predictor.[7] Finding that network, on the other hand, is a whole different story. To understand why, we need to look at the basics of training.

5.1 Training

When a predictor h with weights $\mathbf{w} \in \mathbb{R}^m$ predicts value $\hat{y} = h(\mathbf{x}; \mathbf{w})$ for a data point $\mathbf{x}$ and the true value associated with $\mathbf{x}$ is y, we experience a *loss* $\ell(y, \hat{y})$. The loss is zero when $y = \hat{y}$ and positive otherwise. For instance, the *zero-one loss* is very commonly used for classification:

$$\ell(y, \hat{y}) = \begin{cases} 0 & \text{if } \hat{y} = y \\ 1 & \text{otherwise.} \end{cases}$$

For regression, the *quadratic loss* is a popular choice:

$$\ell(y, \hat{y}) = (y - \hat{y})^2 \; .$$

The *empirical risk* over the training set

$$T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\} \quad \text{with} \quad \mathbf{x}_n \in X \quad \text{and} \quad y_n \in Y \; ,$$

of data point/target pairs is the average loss over that set:

$$L_T(\mathbf{w}) \stackrel{\text{def}}{=} \frac{1}{|T|} \sum_{(\mathbf{x}_n, y_n) \in T} \ell(y_n, h(\mathbf{x}_n; \mathbf{w})) \; .$$

Given a training set T, *Empirical Risk Minimization* (ERM) is the problem of finding a weight vector $\hat{\mathbf{w}}$ with the lowest possible empirical risk[8] on T:

$$\hat{\mathbf{w}} \in \arg \min_{\mathbf{w} \in \mathbb{R}^m} L_T(\mathbf{w}) \; . \tag{1}$$

Notation: The quantity $\min_{\mathbf{w} \in \mathbb{R}^m} L_T(h; \mathbf{w})$ is a single (nonnegative) number, the smallest achievable risk over all choices of $\mathbf{w}$. The notation $\arg\min_{\mathbf{w} \in \mathbb{R}^m} L_T(\mathbf{w})$ represents the *set* of all weight vectors in $\mathbb{R}^m$ that achieve that minimal risk (there could be more than one such vector). Finally, $\hat{\mathbf{w}}$ is one of these vectors, it does not matter which.

ERM is a *fitting* problem, as it seeks a parameter vector for which the function h best fits the data in T.

[6] Somewhat loosely speaking, a differentiable function is Lipschitz when its gradient is uniformly bounded by a constant. This notion can be defined more generally without reference to differentiability.

[7] This holds even for classifiers, since any finite label set Y can be viewed as a subset of the reals.

[8] We are assuming that the minimum in this expression exists. Discussion of this assumption is beyond the scope of these notes.

Generalization What is the point of fitting a predictor to a training set T if we only look at how well the predictor does on that set, as the risk L_T does? A predictor estimates a value $\hat{y}_n$ from a training data point $\mathbf{x}_n$, but we already know the correct answer y_n. What is the point of estimating the answer again?

This question goes to the key difference between data fitting, or empirical risk minimization, and machine learning: Data fitting is asked to do well on the training set, and is used when something about the parameters of the predictor itself is of interest. Perhaps an economist postulates that price and demand are linearly correlated, and wants to determine the regression coefficients. The goal of such a study is not so much to predict new prices for new levels of demand, but rather to use the slope of the regression line to understand the effects of a demand fluctuation on changes of price.

In contrast, machine learning focuses on prediction, and is asked to do well *on previously unseen data.* The parameters of the predictor are not of interest *per se*, and the goal is instead to estimate values $\hat{y}$ corresponding to new data points $\mathbf{x}$. A predictor that does this well is said to *generalize well.* Failure to generalize well can be caused by a predictor that is not flexible enough (*underfitting*) or by one that is too flexible (*overfitting*), as we will see soon.

Statistical Risk Minimization Conceptually, one can link training data to "previously unseen data" by assuming that all data, both those in the training set and those seen after training, are drawn independently and at random from some joint probability distribution $p(\mathbf{x}, y)$ of data points and targets called the *generative model of the data*, or "model" for short.

The goal of machine learning is then not to minimize the empirical risk, bur rather the *statistical risk*:

$$L_p(\mathbf{w}) = \mathbb{E}_p[\ell(y, h(\mathbf{x}; \mathbf{w}))] \ ,$$

that is, the statistical expectation of the loss over the model p. This is a measure of how poorly h does not just over the training set, but rather over all data that could possibly be drawn from the model. The problem of machine learning is then Statistical Risk Minimization (SRM), that is, the problem of finding

$$\hat{\mathbf{w}} \in \arg \min_{\mathbf{w} \in \mathbb{R}^m} L_p(\mathbf{w}) \ ,$$

a much taller order than ERM.

Overfitting The main practical problem with SRM is that the model p is generally unknowable. What is, for instance, the probability distribution over the set of all possible images? Or over all possible English sentences? All we usually have is data. Even if we have large amounts of data, estimating probability distributions over spaces with many dimensions is hopeless. Thus, for all but the simplest cases, minimizing the statistical risk is not possible.

To see how to address this difficulty, we need to understand first what happens when training a neural network through empirical risk minimization alone, that is, by fitting the predictor's weights to the training set T.

Fitting occurs through gradient descent and, more specifically, through Stochastic Gradient Descent (SGD), because the empirical risk is the average of a large number of terms, one term per training sample. Starting with a random initial weight vector $\mathbf{w}_0$, SGD applied to the empirical risk minimization problem (1) produces a sequence of vectors

$$\mathbf{w}_0, \mathbf{w}_1, \ldots$$

that yield values of the empirical risk $L_T(\mathbf{w})$ that decrease on average. Since deep networks are very expressive, it is often possible to reduce the training risk $L_T(\mathbf{w})$ to zero or near zero. This occurs because the predictor has more weights than strictly necessary to fit the training set.

This situation is to some extent analogous to what occurs when a high degree polynomial is fit to data that approximately follow a a lower-degree polynomial. If there is any noise in the data, that is, if y is not exactly a polynomial function of $\mathbf{x}$, then the higher-degree polynomial will fit the data *including noise*.

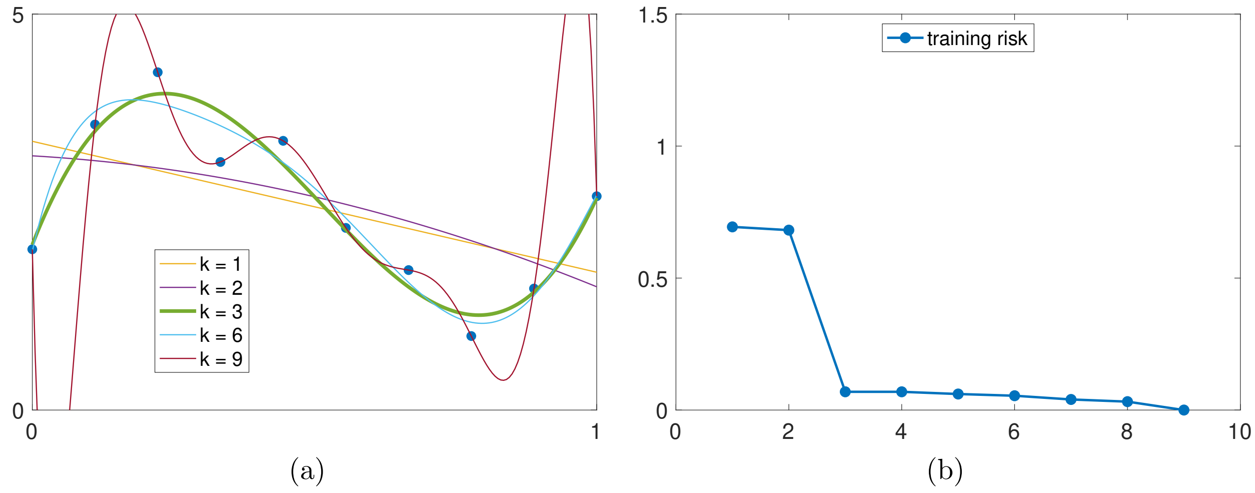


Figure 4: (a) Polynomial fits of degrees between 1 and 9 to ten training data points (blue dots). The dots were generated with a third degree polynomial plus random noise. The third-degree polynomial (green curve, $k = 3$) fits the dots well but not exactly. (b) Training risk as a function of the degree k of the polynomial. The training risk keeps decreasing until it reaches zero for $k = 9$.

Consider for instance the blue dots in Figure 4 (a). These dots are our tiny training set T. They were generated with a third degree polynomial, except that some random noise was added to their y coordinate. The distribution of the coordinates of these dots are the generative data model $p(x, y)$, which is unknown to the data fitting algorithm. The green, thicker curve ($k = 3$) results from fitting a third degree polynomial to the dots. The green curve fits the data quite well, but not exactly, because of the added noise: Not all blue dots are on the green curve.

If one were to use a lower-degree polynomial, as with the yellow and purple curves for degrees $k = 1$ and 2 in Figure 4 (a), *underfitting* would occur, that is, the curves would not pass anywhere close to the blue dots. Higher-degree polynomials, on the other hand, do fit the data well, as shown by the light blue and magenta curves in the Figure, corresponding to degrees $k = 6$ and 9.

In this analogy, the training risk, when using the quadratic loss, is the root-mean-squared average of the vertical distances between the dots and the approximating curve. This risk is plotted in Figure 4 (b) for the five polynomials in Figure 4 (a) plus a few more, as a function of their degree k. The empirical training risk is large for $k = 1$ and 2, then drops considerably for $k = 3$ and keeps decreasing after that. Since there are ten dots, a degree-9 polynomial can always interpolate them, that is, it fits them with zero residual: The training risk is zero when $k = 9$, and would remain zero even for higher degrees.

The empirical training risk when a deep network is trained by SGD follows a similar trend *as a* *function of the epoch number* (as opposed to a polynomial degree) and keeps decreasing. Since the weight vector $\mathbf{w} \in \mathbb{R}^m$ has many entries (m is large), the empirical training risk often approaches zero over time. We then say that the predictor has *memorized the training set T*, in that it is able to give a perfect answer for all of its samples.

This is not necessarily good news. For instance, a network designed to recognize motorcycles may have been presented a training set containing outdoor motorcycle images (with a "motorcycle" label) and other images of different objects ("not a motorcycle" label), many of which are indoors. Rather than learning the "concept" of motorcycle, this network may have learned that images with a blue sky and gray asphalt on the ground must be motorcycles, simply because T is biased in this way. The network is then unlikely to do well on new motorcycle images, when these are taken indoors, or on non-motorcycle images taken outdoors. This biased training set is the machine learning counterpart of the noisy data in the polynomial fitting scenario: The degree-9 polynomial did not capture the "concept" of the third degree polynomial underlying the data, but it rather adapted to all the oscillations caused by noise. The network (or the polynomial fit) *overfit* the data.

The Validation Set One way to avoid both underfitting and overfitting for a neural network would be to choose the number m of weights so as to match the complexity of the problem somehow. In this way, even if training were to proceed to completion, the network would be "just right," analogously to the third-degree polynomial in the example above.

However, tuning the number m of weights in the predictor is difficult. First, designing a deep network to have a predefined number m of weights is not straightforward, as we will see when we study network architectures. In addition, knowing when m is "just right" is problematic as well: If m were too small (underfitting), we would notice that the empirical training risk flattens out before approaching zero. However, if the risk were to go zero, we would only know that m is *at least* as large as needed for perfect fitting, but could also be much larger.

In addition, even if we were able somehow to determine a value of m that fits the training data well but not perfectly (whatever *that* means!), good *prediction* performance could still require a larger value of m, that is, a larger neural network. This is because we do not know how large a training set needs to be so as to represent the underlying data well. In other words, it could happen that a given value of m allows for interpolation merely because the training set T is too small, and the risk could be worse if T contained more data.

Because of these difficulties, instead of somehow tuning the value of m to the data precisely, one addresses over- and under-fitting in deep learning by introducing a second data set V, called the *validation set*, which is meant to represent "previously unseen data." The weight vector $\mathbf{w}$ is still fitted to the training set T, but in a way that the resulting predictor $h(\mathbf{x}; \hat{\mathbf{w}})$ does well on the validation set, rather than on the training set, as described next.

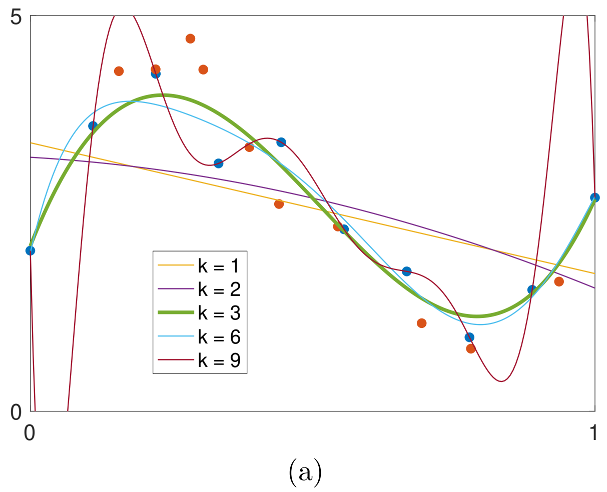
Early Stopping by Validation After each training epoch (recall that an epoch of SGD is a complete scan of all the data in the training set), the empirical risk $L_V(\mathbf{w})$ on the validation set V is computed. This risk is typically greater than the empirical training risk $L_T(\mathbf{w})$ for all values of $\mathbf{w}$, because the SGD algorithm does not see the data in V, and therefore cannot adapt to it as it does to the data in T.

In the early stages of training, the empirical validation risk $L_V(\mathbf{w})$ decreases as the network improves its performance. After a while, however, the validation risk levels off and then typically *increases* as SGD continues. This increase is a symptom of *overfitting*: While the predictor $h(\mathbf{x}; \mathbf{w})$ does better and better on the *training* data, it eventually does so by adjusting to idiosyncrasies of T (as in the motorcycle example above), but then fails to predict correct outputs for many of the samples in V.

Training is then stopped when the validation risk reaches a minimum. The epoch in which this occurs is detected *post facto*, because one has to see an increase before a minimum can be declared. The weights $\mathbf{w}_{\text{best}}$ for the lowest validation risk encountered so far are therefore stored away during training. Once $L_V(\mathbf{w})$ starts to increase, $\mathbf{w}_{\text{best}}$ is returned as the result of training.

Patience As a practical matter, detecting when $L_V(\mathbf{w})$ starts to increase is nontrivial: Since the validation risk is just an estimate of the statistical risk and the validation set V has finite size, the validation risk may fluctuate during training. It is then typical to set a so-called *patience* parameter, which is a positive integer π. If none of the last π epochs have resulted in an update of $\mathbf{w}_{\text{best}}$ (that is, no improvement on $L_V(\mathbf{w})$ has been observed over π epochs), then training is stopped and $\mathbf{w}_{\text{best}}$ is returned.This technique has been studied empirically [1], but no clear guidelines have emerged on an optimal patience parameter π.

Validation in Polynomial Fitting Overfitting can be measured also for polynomial fitting in a similar way. In addition to the same blue dots as in Figure 4 (a), Figure 5 (a) also shows ten red dots, which represent a validation set V. These red dots are generated with the same third degree polynomial and noise distribution used for the blue dots (training set T): They are drawn from the same generative data model $p(x, y)$ from which the blue dots were drawn.



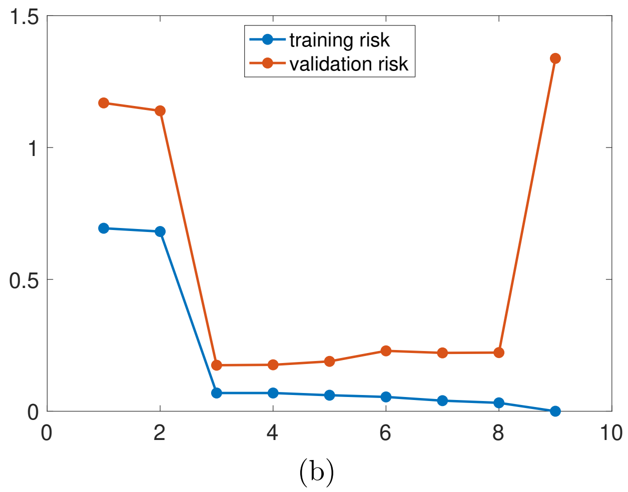


Figure 5: (a) Polynomials of degrees between 1 and 9 fit the training data. Blue points are the training set and red points are the validation set. Only some of the polynomials are shown, to reduce figure clutter. The polynomial that generalizes best to the validation data has degree $k = 3$ and is shown as a thicker, green line. (b) Training and validation risk as a function of the degree k of a polynomial fit to the blue dots in (a). The lowest validation risk is achieved for $k = 3$, while the training risk keeps decreasing even beyond that.

However, the red dots are never used in training, and polynomials are still fit only to the blue dots. The resulting polynomials in Figure 5 (a) are therefore identical to those in Figure 4 (a). However, the red dots are now used to measure a validation risk, shown as a red plot as a function of degree k in Figure 5 (b).

The polynomials for $k = 1$ or 2 do poorly on V, since the predictor is not sufficiently expressive. The polynomial for $k = 3$ does best on V, and those for k between 4 and 8 do only moderately worse. When $k = 9$, on the other hand, overfitting is glaringly obvious: The training risk is zero and yet the validation risk skyrockets: The degree-9 polynomial has learned mostly the noise in the data, rather than the underlying generative law.

Caveat While trends are similar for training a deep network and fitting polynomials, and so are the reasons for them, the big difference between the two scenarios is the parameter being used to modulate the extent of fitting. For polynomials it is the degree k. For deep networks, it is the number of epochs used for training.

5.2 Dropout

Since deep nets have a large number of parameters, they would need impractically large training sets to avoid overfitting if no measures are taken during training. Early termination by monitoring the validation risk, as described above, is one such measure.

More generally, an excellent way to avoid overfitting in the presence of limited data would be to build one network for every possible setting of the parameters, compute the posterior probability of each setting given the training set, and then aggregate the nets into a single predictor that computes the average output weighted by the posterior probabilities. This approach is obviously infeasible to implement even for small nets.

One way to approximate this scheme in a computationally efficient way is called the *dropout* method [3]. Given a deep network to be trained, a *dropout network* is obtained by flipping a biased coin with $\mathbb{P}[\text{heads}] = p$ for each neuron of the original network and "dropping" that neuron if the flip turns out tail. Dropping the neuron means that the output from that neuron is clamped to zero, so that the neuron becomes effectively inactive.

One then trains the network by using mini-batches of training data, and performs one iteration of training on each mini-batch after turning off neurons independently with probability $1-p$. When training is done, all the weights in the network are multiplied by p, and this somewhat resembles averaging the outputs of the nets with weights that depend on how often a unit participated in training. The value of p is typically set to $1/2$.

Each dropout network can be viewed as a different network, and the dropout method somewhat resembles sampling a large number of nets efficiently.

The theoretical justification of this method is dubious, but empirical studies show its effectiveness in reducing overfitting.

6 Testing

Suppose that you train a deep network on a data set T and use a validation set V to determine when to stop training. Once the optimal weight vector $\hat{\mathbf{w}}$ has been found, the following question arises: How well can you expect your predictor to do on new data? That is, how well does the predictor generalize?

One may be tempted to use the validation risk $L_V(\hat{\mathbf{w}})$ as a measure of generalization performance, since this risk is computed on a data set V that has not been used for training the predictor.

However, just as training "taints" the training data, so that validation must be performed on a data set separate from the training set, validation "taints" the validation data. In other words, validation, just like training, is an estimation procedure that finds an optimal number of training epochs by evaluating the performance of the predictor on the validation set V after every epoch. If V is too small or is biased, overfitting may occur during validation as well, and the number of epochs chosen may be a consequence of the idiosyncrasies of V. Perhaps if we were to use a different V we would obtain a different number of epochs.

The availability of a test set S, entirely disjoint from all other data sets used during training, is indispensable. No research article in machine learning is ever accepted if there is even the suspicion that any part of S has been used during training, either directly or indirectly.

In summary, the issues in validation are analogous to those encountered in training as far as overfitting is concerned. Thus, we generally need the following three sets:

- A *training set* T to train the predictor.
- A *validation set* V to determine when to stop training.
- A *test* set S to evaluate the generalization performance of the predictor.

All these sets contain data point/value pairs $(\mathbf{x}, y)$, that is, they are all subsets of $X \times Y$.

Termination by Cross-Validation If collecting a training set is difficult (and we saw that it is), collecting *three* data sets is even harder, and may even be unrealistic in applications where each sample is very costly. Even when expense is not an overriding issue, using a separate training set T, validation set V, and testing set S means that we forgo using all the available data in $T \cup V \cup S$ for training, and we therefore make the predictor intentionally worse than it could be, in order to hold out data for validation and testing.

Because of these considerations, some *resampling* techniques have been developed that allow using a single data set for both training and model selection. The most popular of these techniques is called *cross-validation.*

In a nutshell, before training starts the test set S is separated out from the rest of the data. Let D be the remaining data. At the beginning of a new epoch e, the set D is scrambled as usual. If cross-validation is used, D is split at random into a training set T_e and a validation set V_e specific to that epoch and in some fixed proportion (for instance, 3 to 1). Training is then stopped if the sequence of validation risks $L_{V_0}(\mathbf{w}_0), L_{V_1}(\mathbf{w}_1), \ldots$ measured on V_e with the weights $\mathbf{w}_e$ obtained at the end of epoch e shows signs of increase.

It was observed under the heading *Patience* above that it is difficult to detect this increase reliably. With cross-validation these considerations still hold, and are exacerbated by the fact that the validation risk is computed over a different set V_e for every epoch. The resulting randomness of course worsens the fluctuations of the validation risk, and one may need a larger patience parameter π for a reliable detection of overfitting.

References

[1] L. Prechelt. Early stopping—but when? In *Neural networks: Tricks of the trade*, pages 53–67. Springer, 2012. G. Montavon, G. B. Orr and K.-R. Müller, editors.

[2] S. Sonoda and N. Murata. Neural network with unbounded activations is universal approximator. Technical Report 1505.3654 [cs.NE], arXiv, 2015.

[3] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.