1

Introduction to Neural Networks

Isha Garg and Kaushik Roy

Elmore School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA

The availability of compute power and abundance of data has resulted in the tremendous success of deep learning algorithms. Neural Networks often outperform their human counterparts in a variety of tasks, ranging from image classification to sentiment analysis of text. Neural networks can even play video games [1] and generate artwork [2]. In this chapter we introduce the basic concepts needed to understand neural networks. The simplest way to think of how these networks learn is to think of how a human learns to play a sport, let's say tennis. When someone who has never played tennis is put on a court, it takes them just a few volleys to figure out how to respond to an incoming shot. It might take a long time to get good at a sport, but it is quite magical that only through some trial and error, we can learn how to swing a racquet to a tennis ball heading our way. If we were to write a mathematical model to calculate the angle of swing, it would require many complicated variables such as the wind velocity, the incoming angle, the height from the ground, etc. Yet, we just learn from real-life examples that swinging a particular way has a particular impact, without knowing these variables. This implicit learning from examples is what sets machine learning models apart from their rule-based computational counterparts, such as a calculator. These models learn an implicit structure of the data they see without any explicit definitions on what to look for. The learning is guided by a lot of examples available with ground truth, and the models learn what is needed from these datapoints. Not only do they learn the datapoints they have seen, they are also able to generalize to unseen examples. For instance, we can train a model to differentiate between cats and dogs with, say, 100 examples. Now when we show them new examples of cats that were not present in the training set, they are still able to classify them correctly. There are enormous applications of the field, and a lot of ever-evolving subfields. In Section 1.1, we introduce some basic taxonomy of concepts that will help us understand the basics of neural networks.

1.1 Taxonomy

1.1.1 Supervised Versus Unsupervised Learning

In the unsupervised learning scenario, datapoints are present without labels. The aim is to learn an internal latent representation of data that catches repeated patterns and can make some decisions based on it. By latent representations, we mean an unexposed representation of data that is no longer in the original format, such as pixels of images. The hope is that repetition magnifies

Advances in Electromagnetics Empowered by Artificial Intelligence and Deep Learning, First Edition.
Edited by Sawyer D. Campbell and Douglas H. Werner.

the significant aspects of images, or aids in learning compressed internal representations that can remove noisy artifacts or just learn lower dimensional representations for compressed storage, such as in AutoEncoders [3]. An advantage of having a meaningful latent space is that it can be used to generate new data. Most concepts covered in this chapter pertain to discriminatory models for regression or classification. However, in models such as Variational AutoEncoders [4], the latent space can be perturbed in order to create new data that is not present in the dataset. These models are called generative models. Unsupervised learning problems are harder and an active area of research, since it removes the need to label data. In this chapter, we will stick to supervised learning problems of the discriminatory kind.

1.1.2 Regression Versus Classification

The kind of output expected from a supervised learning discriminatory task dictates whether the problem is one of regression or classification. In regression, the output is a continuous value, such as predicting the price of a house. In classification, the task corresponds to figuring out which of the predefined classes an input belongs to. For example, determining whether an image is of a cat or a dog is a two-class classification problem. In this chapter, we will give examples of both regression and classification tasks.

1.1.3 Training, Validation, and Test Sets

An important concept in deep learning is that of inference versus training. Training is the method of determining the parameters of a model. Inference is making a prediction on an input once the training is complete. When we have a dataset, we can have many different resulting models and predictions for the same query, depending on initialization and the choice of some tunable parameters, which are called hyperparameters. This means we need a way of testing different models. We cannot test on the same data as that used for training, since models with enough complexity tend to memorize training data, leading to the problem of overfitting. Overfitting can be mitigated by using regularization techniques, discussed later. To avoid memorizing data, there is a need to partition the entire set of data into a training and a testing set. We choose a certain value of the hyperparameters and train on the training dataset, get a trained model, and then test it on the testing dataset to get the final reported accuracy. However, this is not the best practice. Let's say a particular choice of hyperparameters did not yield good testing accuracy, and hence we alter these hyperparameters and retrain the model until we get the best testing accuracy. In this process, we are overfitting to the testing set, because we, as the hyperparameter tuners, are exposed to the testing accuracy. This is not a fair generalized testing scenario. Thus, we need a third partition of the data, called the validation set, upon which the hyperparameters should be tuned. The testing set is shown only once to the final chosen model, and the accuracy obtained on that is reported as the models final accuracy. A commonly used split percentage for the dataset is 70%-15%-15% for training, validation, and testing, respectively.

The rest of this chapter is organized as follows. We first introduce linear regression models in Section 1.2 and then extend them to logistic classification in Section 1.3. These sections make up the base for the neurons that serve as building blocks for neural networks. In each section, we introduce the corresponding objective functions and training methodologies. Changes to the objective function to tackle overfitting are discussed in Section 1.4. We then discuss stacking neurons into layers and layers into fully connected neural networks in Section 1.5. We introduce more complex layers that make up convolutional neural networks in Section 1.5 and conclude the chapter in Section 1.6.

1.2 Linear Regression

In this section, we consider a supervised regression problem and explore a simple technique that makes the assumption of linearity in modeling. The canonical example often used to explain this is that of predicting the price of a house. Let's say that you are a realtor with a lot of experience. You have sold a lot of houses and maintain a neat log of all their details. Now you get a new house to sell and need to price it based on your experience with all the previous houses. If you were going to do this manually, you might consider all your records and look for a house that is "similar" to the one that you have sold and give an estimate close to that. This is similar to what a nearest neighbor algorithm would do. But we are going to go a step further and use a linear model to fit a high dimensional curve as best as we can to the data of the old houses, and then determine where a new house would perform according to this model.

Being a diligent realtor, you noted all the features you thought were relevant to the price of a house, such as the length of the house, the width of the house, the number of rooms, and the zip code. Let's say, you have D such features, and N such houses in your log. One can imagine each house as a point in D dimensional space. We have N such points, and the problem of regression essentially reduces to the best curve we can fit to this data. Since we are assuming a linear model, we will try and fit a line to this data. The hypothesis underlying the model can be denoted as $h(\boldsymbol{\theta})$, with $\boldsymbol{\theta}$ being the parameters to be learned. The equation for this model for a single datapoint, x, thus becomes:

$$\hat{\boldsymbol{y}} = h(\boldsymbol{\theta}, \mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_D x_D \tag{1.1}$$

We can write this in abridged form using matrix or vector multiplication. Each house is represented as a D-dimensional vector, and all the N houses together can be concatenated into the columns of an $D \times N$ matrix, $\mathbf{X}$. Since θ_0 does not multiply with any input, it is known as the bias, denoted by b. The remaining parameters are denoted by the matrix $\boldsymbol{\theta}$.

$$\hat{\boldsymbol{y}} = h(\boldsymbol{\theta}, \mathbf{X}) = \boldsymbol{\theta}^{\mathbf{T}}\mathbf{X} + \mathbf{b} \tag{1.2}$$

We use parameters and weights interchangeably to refer to $\boldsymbol{\theta}$. For the single point case, $\boldsymbol{\theta}$ and $\mathbf{X}$ are D-dimensional vectors and b and $\hat{y}$ are scalar values. For multiple points, $\mathbf{X} \in \mathbf{R^{D \times N}}$, $\boldsymbol{\theta} \in \mathbf{R^{D \times 1}}$, b is a scalar repeated for each sample, and the output is a value for each sample, i.e. $\hat{\boldsymbol{y}} \in \mathbf{R}^N$. Often, for ease of notation, b is absorbed into $\boldsymbol{\theta}$, with a corresponding 1 appended to each datapoint in $\mathbf{X}$ so that the equation is simplified to $\hat{\boldsymbol{y}} = \boldsymbol{\theta}^T\mathbf{X}$. $\boldsymbol{\theta}$ is the matrix that takes inputs from D-dimensional input space to the output space, which in this case is one-dimensional, the price of the house. This equation is shown pictorially as a single node, also called a neuron, in Figure 1.1.

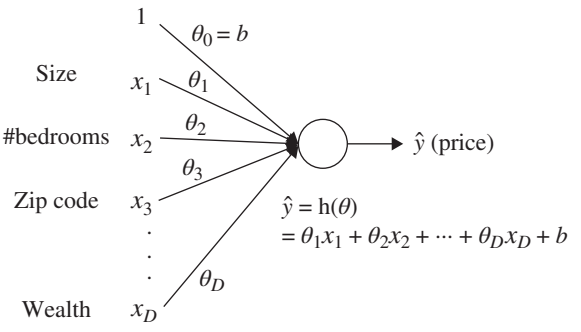

Figure 1.1 A simple linear model that takes as input a D dimensional feature vector and predicts a single dimensional output, in this case, price. The linear hypothesis is shown.

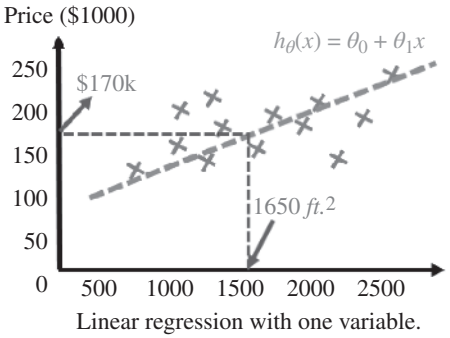

Figure 1.2 Linear Regression with one variable for visualization purposes. The task of regression can be viewed as fitting the best curve to the data. Since we assume linear models, we show the best fit line. Inference of a single point is shown; when a new data point comes in, we can predict its price by seeing where it lies on the line.

Since D dimensional space is really hard to imagine and pictorially represent, for the sake of visualization, we assume $D = 1$ in Figure 1.2. This means that we have only one feature: that of, say, square feet area. We can imagine all the points laid out on the 2D space with the X-axis being the area feature value, and the Y-axis representing the price that the house sold for. The true data distribution might not be linear, and hence the datapoints might not match the best fit line as shown in the figure. The mismatch between these points from the line is called the training error, also referred to as cost or loss of the model. The error between the testing points and their true value is correspondingly referred to as testing error. Note that during inference, the parameters are held constant and Eq. (1.2) can be written as a function of $\mathbf{X}$ alone. However, we need to train this model, i.e. we need to learn the parameters $\boldsymbol{\theta}$ such that the predicted output matches the ground truth. In order to find such $\boldsymbol{\theta}$, we employ objective functions which minimize the expected empirical training error.

1.2.1 Objective Functions

An objective function, as apparent from the name, is a mathematical formulation of what we want to achieve with our model. The objective function is also called the loss function or the cost function, since it encapsulates the costs or losses incurred by the model. In the linear regression model shown above, we want the prediction to align with the ground truth price of the house. In the case of classification, the objective would be to minimize the number of misclassifications.

In the example of predicting the house price, a possible objective function is the distance between the best fit curve and the ground truth. The distance is often measured in terms of norm. The Lp-norm of an n-dimensional vector is defined as:

$$\|x\|_p = (|x_1|^p + |x_2|^p + \cdots |x_n|^p)^{1/p} \tag{1.3}$$

Commonly used norms are $p = 1$ and $p = 2$, which translate to Manhattan (L1) and Euclidean (L2) distance, respectively. Let's assume our objective is to minimize the L2 distance between the predicted house price $\hat{y}_i$ and the ground truth value y_i for all the N samples of houses in our log. The datapoint specific cost is averaged into the overall cost, depicted by $J(\boldsymbol{\theta})$.

$$\hat{y}_i = \boldsymbol{\theta}^T x_i + b \tag{1.4}$$

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{N} \|\hat{y}_i - y_i\|_2^2 \tag{1.5}$$

$$\texttt{where for any vector a, } \|\mathbf{a}\|_2 = \sqrt{\sum_j a_j^2} \tag{1.6}$$

where datapoints are indexed by the subscript i. Note that the cost is a scalar value. If the cost is high, the model does not fit the data well. In Section 1.2.2, we discuss how to find the parameters to obtain the best fit to the training data, or minimize the cost function, $J(\boldsymbol{\theta})$. This process is called optimization and the commonly used method for performing this optimization is Stochastic Gradient Descent (SGD).

1.2.2 Stochastic Gradient Descent

In Section 1.2.1, we introduced the cost function that captures the task we want our model to perform. We now discuss how this cost function is used in training, to find the right parameters for the task. Most cost functions used in neural networks are much more complicated and non-convex, and we used Stochastic Gradient Descent to minimize them. Hence, even though the cost function we discussed for linear regression is convex and can be minimized analytically, we explore how to utilize gradient descent to minimize it. The analogy used to understand gradient descent is usually one of a hiker finding themselves blindfolded on a hill, and trying to find their way to the bottom of the hill. In this analogy, the hill refers to the landscape of the loss function such that the height corresponds to the cost. At the bottom of the hill lies the minima, corresponding to the optimized set of weights that minimize the cost function and achieve the objective. The hiker in question, which represents the set of current weights, desires to move down the hill iteratively, until they reach the minima or close enough to it. A two-dimensional loss landscape (with two weights) is shown in Figure 1.3.

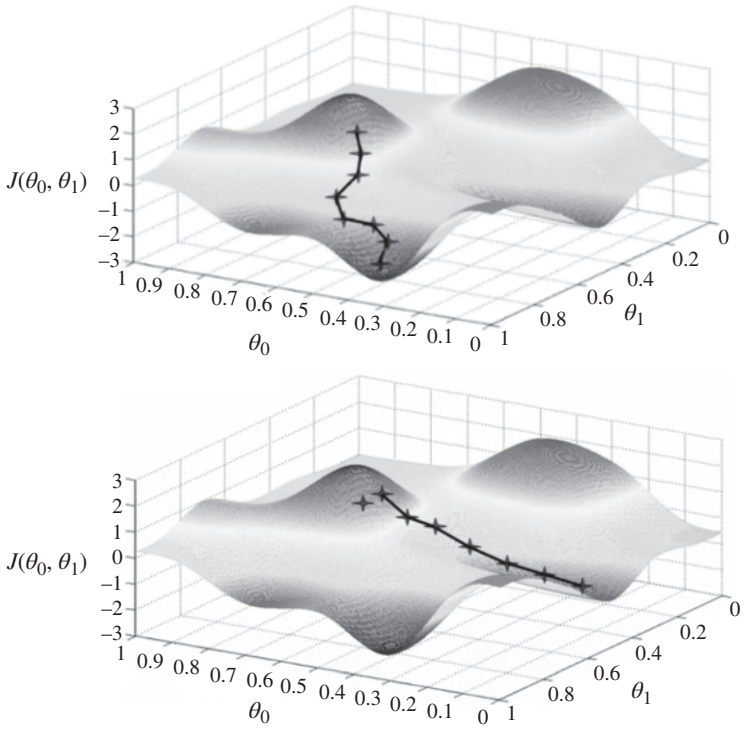

Figure 1.3 A non-convex loss landscape with multiple different local minimas. Different initializations and hyperparameters can result in convergence to different minimas.

The hiker has two immediate questions to answer: in which direction should they step, and how long should the step be (or equivalently, how many steps) in that direction. The latter is called the learning rate, denoted by α. Right away, it is easy to see that too small a step will take too long to get to the minima, and if too large, it is possible to completely miss the minima and instead diverge away as shown in Figure 1.4. In addition, the landscape may not be convex and too small a learning rate can get stuck in local minimas, which can be hard to get out of. The learning rate is a hyperparameter, and is often decayed over the course of learning to smaller values to trade off the number of iterations and the closeness to minima. The final minima that the algorithm converges to is also dependent on the initialization, and hence the entire process is stochastic in nature, and we can get many different sets of weights (corresponding to different local minimas) upon training the same model multiple times, as shown in Figure 1.3.

Now, let's consider the direction of descent. The quickest way to the bottom is via the direction of steepest descent, which is the negative of the gradient at that point. Let's take a deeper look at the gradient, also known as the derivative. Assume x is single dimensional, and the loss is always a scalar value, and hence $f(x)$ is a function from $\mathbf{R} \rightarrow \mathbf{R}$. The derivative of $f(x)$ with respect to x:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{1.7}$$

It essentially captures the sensitivity of the function to a small change in the value of x. When x is multi-dimensional, f maps from $\mathbf{R}^d \rightarrow \mathbf{R}$. In this case, we use partial derivatives, that show the sensitivity of f($\mathbf{x}$) with respect to each dimension of $\mathbf{x}$ (x_i), denoted by $\frac{\partial f(\mathbf{x})}{\partial x_i}$. This computation, however, is difficult to approximate as the change needs to be infinitesimal to calculate correctly. The final expression for the weight update in each iteration is given by:

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \alpha \frac{\partial L}{\partial \boldsymbol{\theta}^{(i)}} \tag{1.8}$$

Henceforth, for clarification, the iterations will be shown as a superscript, and each sample is shown as subscript. Let's make this clearer with an example. Let's return to the case of the linear model shown in Equation (1.2), used for regression to predict house prices. Let's assume the loss function is a simple L2 loss, shown below.

$$\hat{y}_i = {\boldsymbol{\theta}^{(i)}}^T \mathbf{x_i} + b \tag{1.9}$$

$$J = \frac{1}{2N} \sum_{i=1}^{N} \|\hat{y}_i - y_i\|_2^2 \tag{1.10}$$

In each iteration, we take the derivative of the total loss with respect to the weights, and take a direction in the negative of the derivative, scaled by the learning rate α. The iterative weight update can be calculated using the chain rule of derivatives, shown below:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x} \tag{1.11}$$

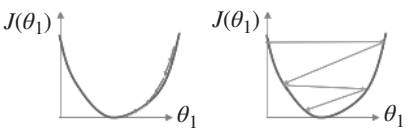

Figure 1.4 A pictorial representation of the role of learning rate in convergence to a minima. Too small a step takes a long time to reach the minima, and too big a step can diverge and miss the minima altogether.

We put all this together to get one iteration of weight update for the linear regression case with L2 loss:

$$\frac{\partial J}{\partial \boldsymbol{\theta}^{(i)}} = \frac{\partial J}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial \boldsymbol{\theta}^{(i)}} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)\mathbf{x_i} \tag{1.12}$$

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \alpha\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)\mathbf{x_i} \tag{1.13}$$

Here, N is the size of the entire training dataset. In practical scenarios, N is usually large, which means that we cannot take a step until we have parsed all the data. In practical settings, we partition the training dataset into minibatches of size 32, 64, 128, or 256 and take one step per minibatch. This is called Batch Gradient Descent and helps us converge to the minima faster. The extreme case of updating after every single data point, i.e. minibatch size = 1, is called Stochastic Gradient Descent. This is because the underlying operations of batched gradient descent are matrix multiplications, which can be implemented by general purpose hardwares, such as GPUs, very efficiently. The minibatch size is upper-bounded by the GPU memory, since the GPU has to hold the entire matrix in its memory to perform the matrix multiplication. We generally refer to updates with minibatches as SGD in the literature, and optimize the batchsize as a hyperparameter. Ruder [5] provides a more detailed discussion. Advanced versions of SGD introduce concepts such as momentum to recall past gradients, and automatically tuned parameter-specific learning rates, such as in Adagrad [6].

The standard practice also involves normalizing the input data, so that the values of all features have similar impact. For the housing price example, the number of rooms will often be values between 1 and 9 and area might be values in the hundreds to thousands of square feet. If the raw values are used in the hypothesis, the area will dominate the house price, or the weight assigned to the number of rooms would have to be really large to have a similar impact as area. To avoid this, the features are first normalized to values between 0 and 1, and then centered by subtracting the mean. They are also then scaled by dividing by the variance, q, a process referred to as mean-std normalization. In practical scenarios, the input data matrix can be very high dimensional in terms of features. This adds significant computational expense to the learning procedure. In addition, a lot of features are correlated to each other and do not offer additional information for learning. For example, if the features are in terms of length, width, and area, just the area is enough to capture the concept of size. To counter this, a standard practice is feature selection, by say, Principal Component Analysis [7], which will remove redundant features.

1.3 Logistic Classification

In Section 1.2, we considered a regression problem, that of predicting the price of a house, which is a real valued output. Now, we look at a classification problem, with $C = 2$ classes. Let's consider a tumor classification example. Given some input feature of a tumor, one classification task could be to predict whether the tumor is malignant ($y = 1$) or benign ($y = 0$). Let's assume the input data, as before, is D-dimensional. Our model is thus a mapping from $\mathbf{R}^D \rightarrow \mathbf{R}^C$, where each of the C parameters of the output corresponds to a score of the input belonging to that class. Let's assume that the same linear model still applies,

$$h(\boldsymbol{\theta}, \mathbf{x_i}) = \boldsymbol{\theta}^T\mathbf{x_i} + \mathbf{b} \qquad i = 1, 2\ldots\ldots, N \tag{1.14}$$

where, as before, N is the number of datapoints, $\mathbf{x} \in \mathbf{R}^D$ is the input, and the output $\hat{\boldsymbol{y}} \in \mathbf{R}^C$. The predicted output, $\hat{\mathbf{y}}_\mathbf{i}$, is distinct from the ground truth label, $\mathbf{y_i}$. The weight matrix, $\boldsymbol{\theta}$, is a $\mathbf{D} \times \mathbf{C}$

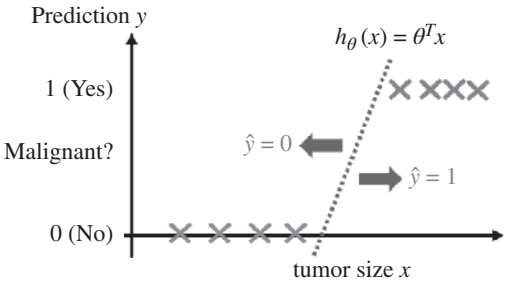

Figure 1.5 The linear classification model: data with ground truth 0 shown in light gray, and for ground truth 1 shown in dark gray. In the classification case, the learned line, shown as the dotted gray line, serves as a boundary, delineating regions for each class.

matrix that takes us from the input space of data to the output space of class scores. Now, instead of drawing the best-fit line, the task is to find the best linear boundary delineating space for each class as shown in Figure 1.5. Additionally, we have a vector of biases, $\mathbf{b} \in \mathbf{R^c}$, one for each class. The predicted output $\mathbf{y_i}$ consists of C values and can be interpreted as the score of each class. Hence a simple classification decision can be based on thresholding. If the output is greater than 0.5, predict class 1; otherwise, predict class 0, as shown below:

$$\hat{y}_i = \begin{cases} 0 & \text{if } h(\boldsymbol{\theta}, \mathbf{x_i}) < 0.5 \\ 1 & \text{if } h(\boldsymbol{\theta}, \mathbf{x_i}) \geq 0.5 \end{cases}$$

The single node discussed until now, also known as a neuron, is the form of the earliest perceptron, introduced in 1958 by Frank Rosenblatt in [8]. We mentioned that the output is interpreted as the scores allotted to each class. A better way to understand the output is if we interpreted them as probabilities. To do this, we have to normalize the scores to lie between 0 and 1. There are two popular ways of doing this, one via the sigmoid function, which turns linear regression/classification into logistic regression/classification, and another via the softmax function. The sigmoid function for a scalar value x is shown below, with the corresponding graph plotted in Figure 1.6.

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

The sigmoid function acts independently on each output score, and squashes it to a value between 0 and 1. While it ensures that each score lies between 0 and 1, it does not ensure that they sum to 1. Hence, it is not strictly a probability metric. However, it comes in handy in case of classification problems where the labels are not mutually exclusive, i.e. multiple classes can be correct for the same sample. The pre-normalized outputs are referred to as logits, often denoted by $\mathbf{z}$. The corresponding changed hypothesis function now becomes

$$\mathbf{z_i} = \boldsymbol{\theta}^T \mathbf{x_i} \tag{1.16}$$

$$h(\boldsymbol{\theta}, \mathbf{x_i}) = \sigma(\mathbf{z_i}) \tag{1.17}$$

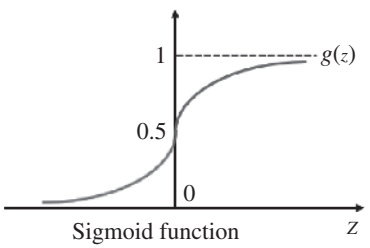

Figure 1.6 The graph of the sigmoid function that squashes outputs into a range of 0 to 1.

The softmax function is an extension of the sigmoid. It normalizes the score of each class after taking the scores of other classes into account. It ensures that the resulting scores sum to 1, so each value can be interpreted as the confidence of belonging to that class. It is useful when the labels are exclusive, i.e. only one class can be present at a time. The equation for softmax is shown below for a vector $\mathbf{x}$ of dimension D:

$$softmax(\mathbf{x}) = \frac{e^{\mathbf{x}}}{\sum_{j=1}^{D} e^{x_j}} \tag{1.18}$$

where x_j corresponds to the jth element in the vector $\mathbf{x}$. Similar to the case with sigmoid, the new hypothesis function now becomes:

$$\mathbf{z_i} = \theta^T \mathbf{x_i} \tag{1.19}$$

$$h(\boldsymbol{\theta}, \mathbf{x_i}) = \hat{\mathbf{y}}_\mathbf{i} = softmax(\mathbf{z_i}) \tag{1.20}$$

We no longer need to threshold, since the output is directly the score of the class, and the class with the maximum probability can be predicted as the classification output. Softmax is a commonly used last layer for typical classification problems with more complex models as well. The probability outputs available from softmax are often used in an information-theoretic objective function, called the cross entropy loss function. Since the outputs function as probabilities, a way of measuring the distance between them is the Kullback–Liebler (KL) divergence [9], and is closely related to cross entropy. For two distributions $p(x)$ and $q(x)$, where p is considered the true distribution, and q is the distribution that approximates p, the cross entropy is defined by:

$$H(p, q) = -\sum_x p(x) \log q(x) \tag{1.21}$$

In the case of a C class classification, we want to measure the error between the true distribution y_i, which is a just point mass on the correct class and zero elsewhere, and the predicted distribution $\hat{\mathbf{y}}_\mathbf{i}$. Hence the loss for the datapoint becomes:

$$l_i = -\log \hat{y}_{i,y_i} \tag{1.22}$$

where $\hat{y}_{i,y_i}$ is the predicted score of the ground truth class for sample i. To optimize this objective function, we employ gradient descent in the direction of the derivative of this cost with respect to the parameters, with a tunable learning rate, similar to logistic regression example.

1.4 Regularization

Let's return to the example of predicting house prices. In Section 1.2, we tried to fit a linear line to the data. It is possible that the best fit of the line may not fit the data well. It could be because the underlying distribution was not linear or the linear model did not have the sufficient complexity to fit the data. This problem is referred to as underfitting, also known as a high bias problem. One way to fix this is to use a more complex model, such as a polynomial of degree 2. Let's say in the example of house price prediction, we only had length and breadth of the house of features. Having second-degree polynomials would allow us to multiply them and have area ($length \times breadth$) as well as one of the features, which might give us a better fit. Taking this further, we can fit an increasingly higher degree polynomial to our data, and it will in most cases end up fitting our training data near perfectly. However, this near-perfect fit of training data to the complex hypothesis means that

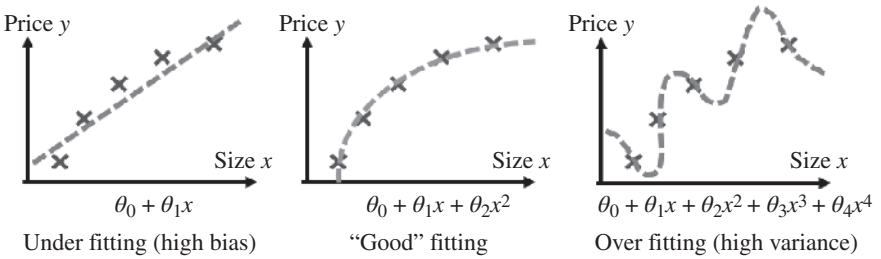

Figure 1.7 Different complexity models can fit the data to different degrees. Too simple a model cannot explain the data well and underfits. Too complex a model memorizes the data and overfits the training set.

the model is memorizing the data, in which case it would not generalize well to the testing or the validation set. This is shown as overfitting in Figure 1.7, also referred to as a high variance problem.

The cost functions used in practical scenarios are highly non-convex, which means there are many local minimas and non-unique sets of weights that the optimization process can converge to, as shown in Figure 1.3. We can encode preferences for certain kinds of weights by expanding the cost function. Since complex models often overfit the data, hurting their generalization performance, we wish to encode a preference for simpler models. By simpler models, we mean those in which no one weight or parameter has the capability to largely effect the cost by itself. A simple way to achieve this is to minimize the Lp norm of all the weights [10]. This means that as any weight grows large, a large value is added to the cost function, which is not preferred since the objective is to minimize this cost function. Thus, the minimization procedure would naturally prefer smaller weights. A common form is to add the L2 norm of weights as a regularization objective to the original cost function, as shown below.

$$L = \frac{1}{2N}\left[\sum_i l_i + \lambda \|\boldsymbol{\theta}\|_2^2\right] \tag{1.23}$$

where λ is the tradeoff parameter that decides the strength of regularization. Note that the regularization function is independent of the data samples, and is just a function of the weights. Let's derive the weight update rule in case of the linear regression problem discussed earlier.

$$\hat{y}_i = \boldsymbol{\theta}^{(i)^T}\mathbf{x_i} \tag{1.24}$$

$$J = \frac{1}{2N}\left[\sum_{i=1}^{N} \|\hat{y}_i - y_i\|_2^2 + \lambda \|\boldsymbol{\theta}^{(i)}\|_2^2\right] \tag{1.25}$$

$$\frac{\partial J}{\partial \boldsymbol{\theta}^{(i)}} = \frac{1}{N}\left[\sum_{i=1}^{N} (\hat{y}_i - y_i)\mathbf{x_i} + \lambda \boldsymbol{\theta}^{(i)}\right] \tag{1.26}$$

The corresponding weight update rule is given by:

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \alpha \frac{\partial J}{\partial \boldsymbol{\theta}^{(i)}} \tag{1.27}$$

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)}\left[1 - \alpha\frac{\lambda}{N}\right] - \alpha\left[\frac{1}{N}\sum_{i=1}^{N} (\hat{y}_i - y_i)\mathbf{x_i}\right] \tag{1.28}$$

1.5 Neural Networks

In Section 1.4, we introduced polynomial functions as an alternative to linear functions in order to better fit more complex data. However, data can be high dimensional and polynomial models suffer from the curse of dimensionality. Let's return to the house prediction example one last time, and assume we have $D = 100$ features in our input data. Let's take a very reasonable hypothesis, that of polynomial functions with a degree of 2. We now have $O(D^2)$ combinations in the input, in this case, 5000. Hence, the model will also have that many extra parameters. It is common to have millions of parameters in the input, such as images that are made of $D = 224 \times 224 \times 3$ pixels, and this is significantly computationally prohibitive. If we expand the hypothesis to a kth order polynomial, the features would grow by $O(D^k)$. We would also need correspondingly larger number of data samples that can be quite expensive to obtain. Clearly, we need different model structures to process such complex data.

Neural networks (NNs) were introduced to counter this explosion in input dimensionality, yet enable rich and complex learning. They stack together multiple perceptrons (or neurons) in a layer together to aid in complex data mapping. In order to aid non-linear mapping, multiple layers are stacked together, with non-linearities in the middle. Training NNs is a simple extension of the chain rule of gradient descent, a process known as backpropagation. NNs have shown astounding amounts of success in all forms of data ranging from images, texts, audio to medical data. We will now discuss their structure and training in detail via the example of training on images. We borrow a lot of the concepts from linear models to build our way up to the final neural network structure.

The basic neuron of the NNs remains the same as in the linear regression model from the house prediction model, shown in Figure 1.8. The input, $\mathbf{x}$, multiplies with a weight represented by an edge in the figure and denoted as before by $\boldsymbol{\theta}$. The output of the neuron is $\boldsymbol{\theta}^T\mathbf{x}$.

Many of these neurons are stacked together in one layer, with each input connecting to each neuron for now. This connectivity pattern results in what is referred to as a fully connected layer, as all neurons are connected to all inputs. We will explore more connectivity patterns when we discuss convolutional neural networks. Let's assume we have s_1 such inputs and the data, as before, is D dimensional. Therefore, the weight matrix dimensions change, $\boldsymbol{\theta} \in \mathbf{R}^{D \times s_1}$. Let's call the output of each neuron as its activation, since it represents how active that neuron is, represented by $a_i,\ i = 1, 2 \ldots s_1$. We can represent all activations as a vector $T^{(j)} \in \mathbf{R}_1^N$, corresponding to a the jth layer.

Till now, we described the first layer, i.e. $j = 1$. We can stack together multiple layers, as shown in Figure 1.9. The layers $T_j, j = 1, 2$ are called the hidden layer representations of the neural networks. In each layer j, neuron i produces activation $a_i^{(j)}$. Each layer gets a bias, with the corresponding input set to 1, represented by the subscript 0. Each layer's weight matrix, connecting layer j to $j + 1$

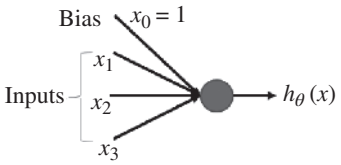

Figure 1.8 Each neuron in a Neural Network essentially performs regression.

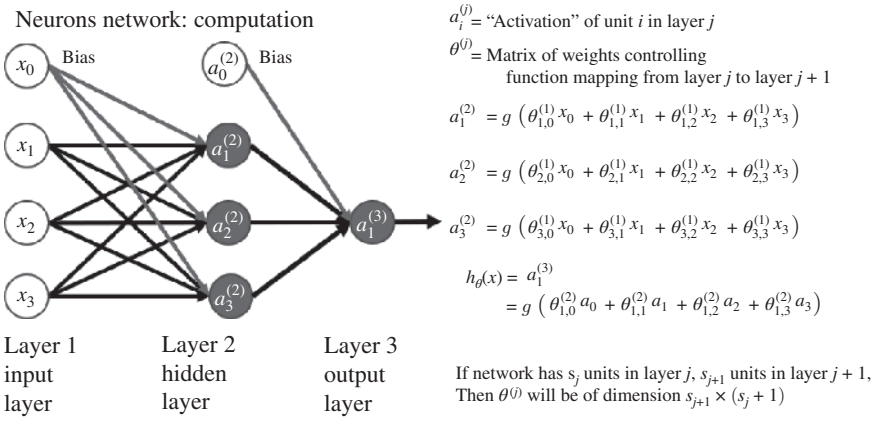

Figure 1.9 A three-layer neural network with the corresponding activation shown.

is denoted by $\theta^{(j)} \in \mathbf{R}^{s_j \times s_{j+1}}$. The equation for the two hidden-layered NN shown in Figure 1.9, is expressed by:

$$\mathbf{T}^{(1)} = \boldsymbol{\theta}^{(1)T}\mathbf{x} \tag{1.29}$$

$$\mathbf{T}^{(2)} = \boldsymbol{\theta}^{(2)T}\mathbf{T}^{(1)} \tag{1.30}$$

$$= (\boldsymbol{\theta}^{(2)T} \cdot \boldsymbol{\theta}^{(1)T})\mathbf{x} \tag{1.31}$$

$$= \boldsymbol{\theta}^{(3)T}x \tag{1.32}$$

$$\text{where } \boldsymbol{\theta}^{(3)} = \boldsymbol{\theta}^{(1)}\boldsymbol{\theta}^{(2)} \tag{1.33}$$

Equation (1.32) implies that stacking two layers with s_1 and s_2 adds no extra complexity than a single layer with just s_2 neurons. This is due to linearity of matrix multiplications: sequential multiplication with two matrices can be denoted as multiplication with a different matrix. We avoid this by adding a non-linearity after each neuron. As discussed earlier, this linearity could be sigmoid, or the more commonly used ReLU, a Rectified Linear Unit. ReLU returns 0 for all inputs less than 0, and passes the input unaltered after 0, and became the default choice for non-linearity after its success in [11]. Let's represent the choice of non-linearity by $g(.)$, and Equations (1.29)–(1.33) are updated as follows:

$$\mathbf{T}^{(1)} = g(\boldsymbol{\theta}^{(1)T}\mathbf{x}) \tag{1.34}$$

$$\mathbf{T}^{(2)} = g(\boldsymbol{\theta}^{(2)T}\mathbf{T}^{(1)}) \tag{1.35}$$

$$= g(\boldsymbol{\theta}^{(2)T}g(\boldsymbol{\theta}^{(1)T}\mathbf{x})) \tag{1.36}$$

$$\neq g(\boldsymbol{\theta}^{(3)T})\mathbf{x}, \text{ for some } \boldsymbol{\theta}^{(3)} \tag{1.37}$$

The forward pass of the input through all layers, generating activations at each neuron and representations at each layer, right up to the cost is called a forward pass or forward propagation. Let's explore this with the multi-class classification example shown in Figure 1.10. In this case, we get an input image and have to predict which class the image belongs to: pedestrian, car, motorcycle, or dog. Earlier we had binary classification, and we only need a single neuron to perform that, since we can threshold on its output. But if we have C classes, we need C neurons in the last layer, corresponding to the C scores for each class. The ground truth labels are now interpreted as one-hot vectors, i.e. $\mathbf{y_i} \in \mathbf{R^C}$, where all elements of $\mathbf{y_i}$ are 0 except the true label, which is a 1 as shown in the figure. Each neuron still performs a one versus all binary classification. For example, the last

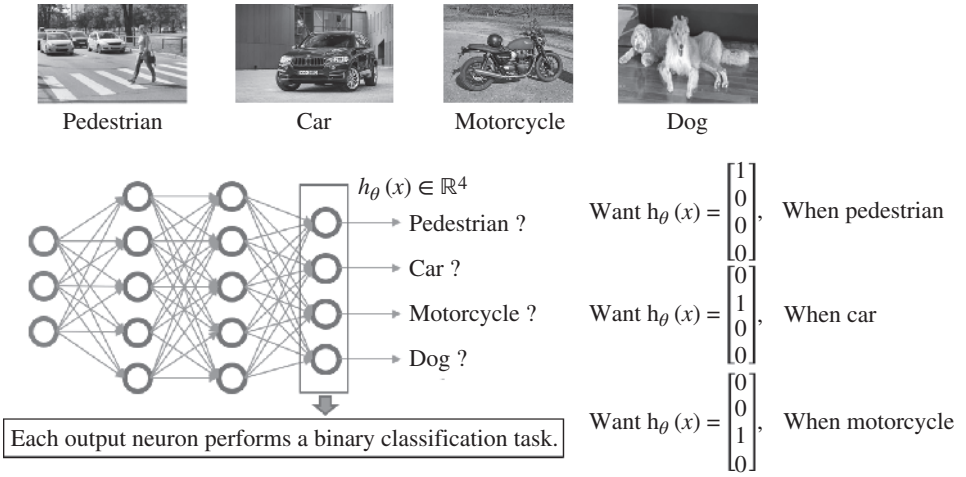

Figure 1.10 Using a NN to perform a 4 class classification task. Source: Sergey Ryzhov/Adobe Stock; Moose/Adobe Stock; brudertack69/Adobe Stock.

neuron in the last layer corresponding to truck class essentially predicts the confidence of the image containing a truck versus not containing a truck. In this form, all the layers of the NN until the last layer perform some sort of feature extraction. This feature vector corresponding to each input is fed into the last layer, which serves as the classifier.

Let's look at the cost function for this example. Let's assume there are L layers in the NN. Each layer has $s_l \quad l = 1, 2, \ldots L$ neurons. There are N training samples shown as pairs of input and ground truth labels: $(\mathbf{x_i}, \mathbf{y_i}), \quad i = 1, 2, \ldots . N$. The previous equations showed how to forward propagate the input to all layers. The activation of layer l is represented by $\mathbf{T^{(l)}}$. The last layer logits are therefore $\mathbf{T^{(L)}}$. Similar to logistic classification, the logits represent unnormalized scores for classes and will be passed through a softmax function for a cross entropy objective function.

$$\hat{\mathbf{y}}_\mathbf{i} = softmax(\mathbf{T}_\mathbf{i}^{(\mathbf{L})}) \tag{1.38}$$

$$J(\boldsymbol{\theta}) = \sum_{c=1}^{C} \mathbf{y_i} \log \hat{\mathbf{y}}_{\mathbf{i,c}} \tag{1.39}$$

$$= -\log \hat{y}_{i,y_i} \tag{1.40}$$

The ground truth label $\mathbf{y_i}$ is one-hot encoded; hence, it only has one non-zero element corresponding to the ground truth class, getting rid of the sum in equation 1.39. The regularization term, if included, would be the L2 norm of all the weights for all layers in the neural network.

We now show how to train all the layers simultaneously. We discussed how the forward pass generates activations at all the intermediate layers, and the loss to be optimized as the objective function. The gradient of the loss for optimization is first calculated at the last layer which has direct access to the cost function, and then flows backward to each parameter using the chain rule. This process is called a backward pass or backward propagation. The weight update rule remains the same for all neurons as earlier with their respective gradients. Let's look at the gradient for the weights in layer j, represented by $\boldsymbol{\theta}^{(j)}$. For ease of notation, we return to the case of three layers and revisit the corresponding forward pass that was described in equations 1.36 for an input $\mathbf{x_i}$.

$$\mathbf{T}_\mathbf{i}^{(\mathbf{1})} = g(\boldsymbol{\theta}^{(1)T}\mathbf{x_i}) \tag{1.41}$$

$$\mathbf{T}_\mathbf{i}^{(\mathbf{2})} = g(\boldsymbol{\theta}^{(2)T}\mathbf{T}_\mathbf{i}^{(\mathbf{1})}) \tag{1.42}$$

$$\hat{\mathbf{y}}_\mathbf{i} = softmax(\mathbf{T}_\mathbf{i}^{(2)}) \tag{1.43}$$

$$J(\boldsymbol{\theta}) = -\log \hat{y}_{i,y_i} \tag{1.44}$$

For each of these equations, we can write the gradient rule and then string them together to get the required gradients.

$$\frac{\partial J}{\partial \boldsymbol{\theta}^{(1)}} = \frac{\partial J}{\partial \hat{\mathbf{y}}_\mathbf{i}} \times \frac{\partial \hat{\mathbf{y}}_\mathbf{i}}{\partial \mathbf{T}_\mathbf{i}^{(2)}} \times \frac{\partial \mathbf{T}_\mathbf{i}^{(2)}}{\partial \mathbf{T}_\mathbf{i}^{(1)}} \times \frac{\partial \mathbf{T}_\mathbf{i}^{(1)}}{\partial \boldsymbol{\theta}^{(1)}} \tag{1.45}$$

This means that NNs avoid the curse of dimensionality in input features and are able to be trained using SGD. Stacking many such layers allows us to achieve more complex learning. This stacking is what gave rise to the term "deep learning." In practice, the softwares used for training NNs use automatic differentiation, a powerful procedure that can calculate gradients quickly. Calculation of gradients is basically matrix–vector or matrix–matrix multiplications, something GPUs are really good at. Combining this computational power with the advent of big data has made NNs very powerful. A particular kind of NN, called the Convolutional Neural Network, allows us to take this even further, and we will discuss that in detail next.

1.6 Convolutional Neural Networks

The NNs introduced in section 1.5 had all layers fully connected. Since these networks regularly deal with high dimensional data, the weight matrices for the fully connected layers can grow quite large. To counter this, we utilize a special class of NNs, called Convolutional Neural Nets, or CNNs, which are particularly useful for extracting features from image and audio data. Additionally, each pixel in the image domain does not form a feature by itself. Quite often a group of neighboring pixels form features relevant to concepts in images that might help make classification decisions, for examples. This informs the connectivity pattern shift from fully connected into convolutional styles

Convolutional neural networks were first introduced in [12] for document character recognition. This network, called Le-Net, performed really well on a handwritten digit dataset, known as MNIST [13]. Since then, there have been many complex networks and datasets introduced. The dataset used for characterizing real-life images is made of millions of $224 \times 224 \times 3$ resolution images, known as ImageNet [14]. The state-of-the-art performance on this dataset is held by powerful networks like ResNets [15] and Vision Transformers [16]. These networks are trained and follow the same inference methodology as NNs, discussed in Section 1.5. However, they differ in the structure of layers. An example of one such CNN structure is shown in Figure 1.11. The first layer in this example is a convolutional layer, made up of convolutional kernels that act on the input. The output of this layer is referred to as feature maps. Often, convolutional layers are followed by subsampling layers to reduce the dimensionality of the maps. The `[conv-maxpool]` structure repeats for a while, ending with one or more fully connected layers. The last layer is the fully connected classifier, which acts upon the extracted features to make the classification decision.

An important concept is that of the receptive field. The receptive field of a kernel is the size of input it accesses. Hence, for the first layer, it is directly the size of the kernel. But as shown in Figure 1.11, each layer takes as input the output of the previous layer, and hence each layer's receptive fields translate backward into larger areas of input. This implies that the receptive field grows as we go deeper into the network. In Section 1.6.1, we discuss some of the layers that make up CNNs, and their functionality.

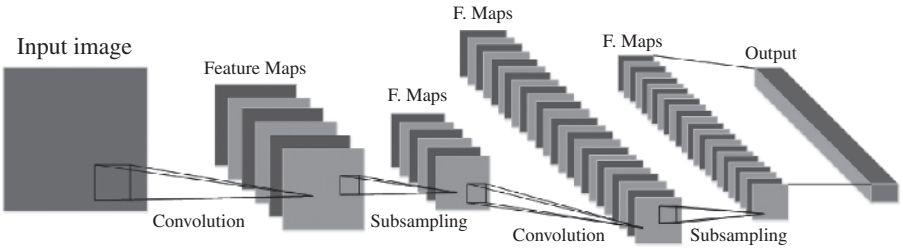

Figure 1.11 A simple CNN structure made of convolutional, subsampling and fully connected layers.

1.6.1 Convolutional Layers

Convolutional layers derive their name from the convolutional operation in signal processing, with the exception that the kernel is not flipped as it passes over the input (and hence, it is actually performing correlation instead of convolution). Let's first consider the simplified one-dimensional convolution operation that occurs between a kernel and a single channel input. A convolutional kernel acts on a patch of the input the same size as the kernel. This is shown in Figure 1.12. The input image, I, and the kernel, K, both have a single channel in this example. Let's assume we have a kernel of size 3×3, shown using binary values. The full input is larger sized, but only a 3×3 patch is taken, as highlighted. A dot product is performed between the image patch and the kernel, which is just element-wise multiplication as expanded above the arrow. The resulting value of 2 is the first pixel of the output feature map. The next patch that convolves with the kernel happens when the highlighted 3×3 slides over to the right. The amount of pixels it moves to the right is known as stride. A stride of 1 is shown in the figure. The same dot product repeats on the slided patch to give the next output of 1. This 3×3 window slides over the entire image, giving rise to the entire feature map shown on the right in Figure 1.12.

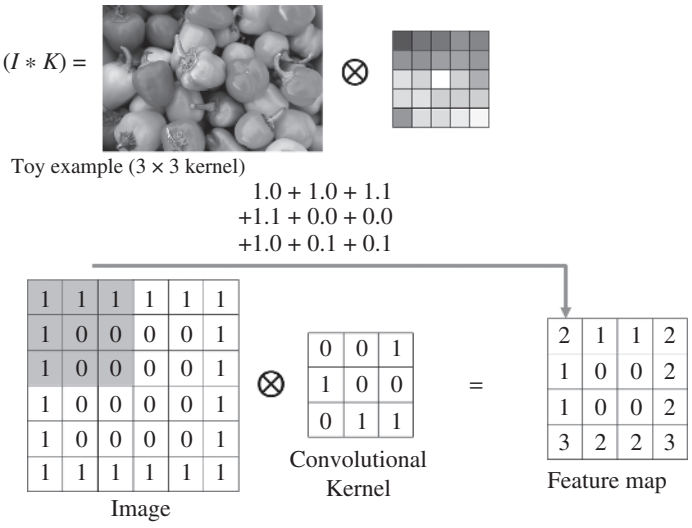

Figure 1.12 The operations involved in convolution are shown. The highlighted part of the input is the patch the kernel convolves with to give a single output in the feature map. This window then slides over the entire image resulting in the whole feature map. Source: DAVID CARREON/Adobe Stock.

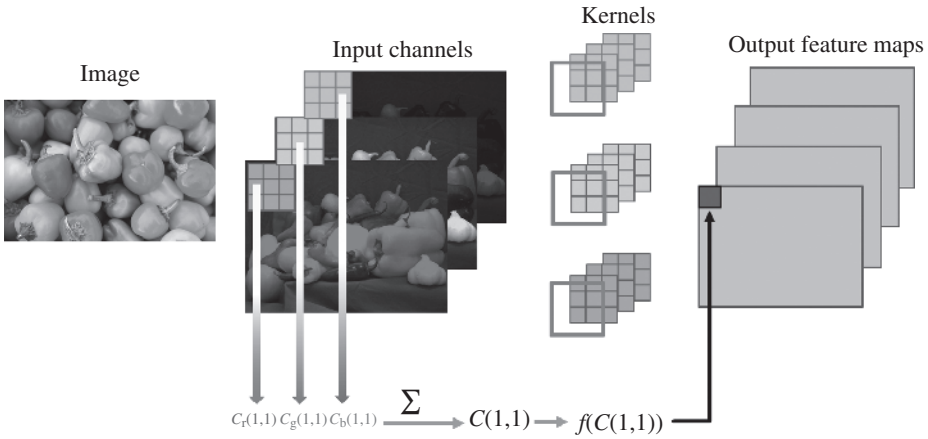

Figure 1.13 One input patch of the image of the same size as the kernel convolves with all channels of one kernel to give rise to a single pixel in the output feature map. Source: DAVID CARREON/Adobe Stock.

Now let us look at what happens with multi-channeled inputs and multiple kernels. Figure 1.13 shows the first convolutional layer. In this case the input, $\mathbf{X}$, is an RGB image, and hence, has 3 channels, 1 for each color. Let's assume the image height and width are H_i and W_i respectively. Hence $X \in \mathbf{R}^{H_i \times W_i \times 3}$. The kernels are usually square, let's say of size $K_i \times K_i \times 3$. In Figure 1.13, the image is shown expanded into its 3 different channels. Correspondingly 4 different kernels are shown stacked one upon the other, with the 3 channels for each kernel expanded vertically. The first kernel, with its 3 channels, convolves with a similarly sized 3×3 patch of the input, and the resulting convolution outputs one pixel. As shown in the image, each channel of the kernel convolves with the corresponding channel of the input. The kernel then slides over by the value of stride to the next patch in the input, creating the next pixel in the feature map. This computation can be performed in parallel for each kernel, with different kernels adding pixels to the depth of the feature map. Hence, if there are M kernels in the layers ($M = 4$ in this example), the output feature map will have a depth of M. This convolution process repeats for later layers, where the input is now the feature maps from previous layers.

Ultimately when the learning is complete, the convolutional kernels learn features relevant to recognizing objects in images. The early layer filters respond to edges and color blobs, but later layers build upon these simpler features and learn more complex features, responding to concepts like faces and patterns. The intuition behind sliding the same kernel across the image is that in images, a feature could be located at any part of the image, such as the center or the bottom left or top right. This connectivity pattern encodes a preference for feature location invariance into the CNN. The size and number of convolutional filters, along with the stride are hyperparameters, along with the total number of layers.

1.6.2 Pooling Layers

Pooling layers, also known as subsampling layers, are useful to reduce the size of the feature maps resulting from convolutions. There are two typical pooling types: max pooling and average pooling. Let's say we have pooling kernels of size 2×2. Similar to the convolutional kernels, pooling kernels will slide over the image in blocks of size of size 2×2, with the prescribed stride. Each block will result in one output. For max pool the output would be the maximum of the 4 values in the 2×2

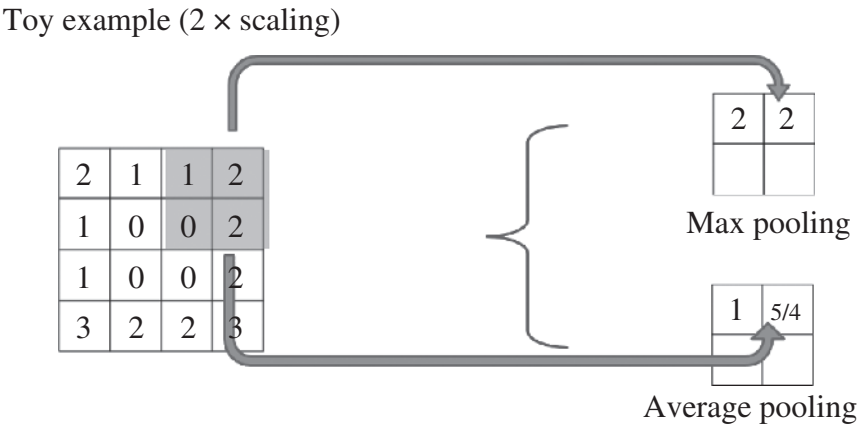

Figure 1.14 The output of max and average pooling applied to blocks of size 2 × 2.

block, and for average pooling, it would be the average of all 4 values, as shown in Figure 1.14. This subsampling introduces some robustness or slight invariance to the feature extraction as well: if a feature is detected at a particular location or a slightly shifted location (with a receptive field of the surrounding 3 values), the output is the maximum or average detection in that region.

1.6.3 Highway Connections

The first CNNs were made of repeating blocks of convolutional layers followed by pooling layers. However, many such blocks were needed to learn complex features. Going too deep results in the problem of vanishing gradients. As gradients backpropagate from the classifier backwards through all the layers, they tend to shrink in magnitude such that the earlier layers get very small gradients and hence, may not be able to learn much. To solve this problem, ResNet [15]-style architectures were introduced that utilize shortcut, highway, or residual connections between the outputs of multiple layers. The output of normal convolution and the highway convolution get concatenated. Hence, when the gradient flows backward, there is a strong gradient flowing back to the earlier layers via the means of these highway connections.

1.6.4 Recurrent Layers

Until now, we have only introduced feedforward layers, layers that only pass an input in one direction. Sometimes, it is useful for layers to have recurrent connections [17], that is layers that are also connected to themselves. This helps a lot in sequential learning, where any output is dependent on the previous outputs, such as in text or video frames. Recurrent networks were introduced to hold some memory of previous inputs by introducing self-connected layers. To train them, however, backpropagation had to unroll the network "in time," resulting in a very large network. This unrolling causes RNNs to suffer from the same problem of vanishing gradients that very deep networks suffered from without highway connections. To counter such problems, Long Short Term Memory Networks (LSTMs) [18] were introduced, which have proven successful at sequence learning tasks. An emerging paradigm of networks that incorporate inherent recurrence are networks that mimic the spike-based learning that occurs in mammalian brains, known as spiking neural networks (SNNs) [19, 20]. They operate on spikes, which can be thought of events that occur when something changes. They accumulate spikes over time to make any inference and use integrate and

fire neurons. Each neuron activates spikes when the accumulated spikes crosses a threshold. These are particularly useful in the case of event-driven sensors that naturally emit data as a time series of spikes. However, they can be used with static data such as images as well, by encoding the pixel intensity in say, the number of spikes over a certain time or the time between subsequent spikes.

There have been many more networks that have grown to billions of parameters in size. They are being utilized for a plethora of tasks, outperforming humans at quite a few of them. A lot of ongoing research focuses on making networks more accurate, making training faster, introducing more learning complexity and generalizability across a range of tasks and newer application domains. We encourage readers to seek out some interesting state-of-the-art challenges or domains of interest and explore state-of-the-art methods in those areas.

1.7 Conclusion

In this chapter, we endeavored to introduce some concepts of machine learning that help us build an intuition for understanding the nuts and bolts of neural networks. We introduced neural networks and convolutional neural networks that have shown tremendous success in many deep learning applications. We showed objective functions that encapsulate our learning goals and how backpropagation can be used to train these networks to achieve these objective. This is a fast-evolving field with applications in nearly every domain. We encourage readers to utilize this as a base and find their application of choice and dive into how deep learning can be utilized to revolutionize that area.

References

1 Mnih, V., Kavukcuoglu, K., Silver, D. et al. (2013). Playing Atari with deep reinforcement learning. *CoRR*, abs/1312.5602. http://arxiv.org/abs/1312.5602.

2 Ramesh, A., Dhariwal, P., Nichol, A. et al. (2022). Hierarchical text-conditional image generation with clip latents. https://arxiv.org/abs/2204.06125.

3 Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

4 Kingma, D.P. and Welling, M. (2013). Auto-encoding variational bayes. https://arxiv.org/abs/1312.6114.

5 Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

6 Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12 (7): 2121–2159.

7 Karl Pearson, F.R.S. (1901). LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2 (11): 559–572. https://doi.org/10.1080/14786440109462720.

8 Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review* 65 (6): 386.

9 Kullback, S. and Leibler, R.A. (1951). On information and sufficiency. *The Annals of Mathematical Statistics* 22 (1): 79–86.

10 Ng, A.Y. (2004). Feature selection, L^1 vs. L^2 regularization, and rotational invariance. *Proceedings of the 21st International Conference on Machine Learning*, p. 78.

11 Nair, V. and Hinton, G.E. (2010). Rectified linear units improve restricted Boltzmann machines. *ICML*.

12 LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86 (11): 2278–2324.

13 Deng, L. (2012). The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29 (6): 141–142.

14 Deng, J., Dong, W., Socher, R. et al. (2009). ImageNet: a large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255. IEEE.

15 He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385. http://arxiv.org/abs/1512.03385.

16 Dosovitskiy, A., Beyer, L., Kolesnikov, A. et al. (2020). An image is worth 16x16 words: transformers for image recognition at scale. *CoRR*, abs/2010.11929. https://arxiv.org/abs/2010.11929.

17 Rumelhart, D.E., Hinton, G.E., and Williams, R.J. (1986). Learning internal representations by error propagation. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations MIT Press Cambridge, MA, USA. 318–362.

18 Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation* 9 (8): 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735.

19 Pfeiffer, M. and Pfeil, T. (2018). Deep learning with spiking neurons: opportunities and challenges. *Frontiers in Neuroscience* 12. https://doi.org/10.3389/fnins.2018.00774.

20 Roy, K., Panda, P., and Jaiswal, A. (2019). Towards spike-based machine intelligence with neuromorphic computing. *Nature* 575: 607–617.