A convolutional neural network primer

For the Oxford C18 and AIMS Big Data courses

Andrea Vedaldi
vedaldi@robots.ox.ac.uk

Version 1.2 June 2019

Contents

Chapter 1

Introduction

These notes summarize the lectures on convolutional neural networks for the Oxford C18 Computer Vision and the Oxford AIMS CDT Big Data courses. The notes are self-contained and can be used as a stand-alone introduction to deep convolutional neural networks in general.

Modern deep neural networks have been one of the most significant technological advances in machine learning in recent memory. Convolutional networks are an important family of deep networks optimized for the analysis of images. These models have state-of-the-art performance in almost all image understanding applications, from object recognition and detection, to image indexing, medical image analysis, perception for autonomous driving, face recognition, and many other.

The notes start by discussing a very simple type of neural network, namely the perceptron (chapter 2), and use it to introduce important concepts such as prediction, loss, learning from example data, generalization, energy optimization via stochastic gradient descent, and data representations. Then, convolutional neural networks (chapter 3) are introduced, with a discussion of most important types of layers, from linear convolution to pooling, non-linear

activation functions, softmax, cross-entropy loss, etc. The concept of receptive field is also introduced and its calculation is derived. Lastly, the discussion shifts to automatic differentiation and back-propagation (chapter 4).

Chapter 2

The perceptron

This chapter introduces the perceptron, one of the earliest and simplest examples of neural network. It also introduces several important concepts in machine learning: building predictors using linear and non-linear operators, training them via energy minimization using example data, generalization, optimization of the model parameters via gradient descent, and data representations.

While artificial neural networks are inspired by biological neural networks, they bear only a superficial resemblance to the latter. Artificial neural networks are best regarded as *parametric functions* that, due to their peculiar structure, can be learned efficiently and result in excellent performance in applications. This chapter introduces the perceptron from this viewpoint, starting from a discussion of generalization, perhaps the most important concept in machine learning.

2.1 Generalization

The key goal of machine learning is to extrapolate *general* properties of data from limited empirical observations. Consider for instance

the sequence of numbers $1, 2, 3, 4$. Asked to predict how the sequence continues, you would likely guess $5, 6, 7, \ldots$. Implicitly, by looking at the first four numbers, you have learned that each number is obtained by adding 1 to the previous one. This rule *generalizes* to the whole sequence and allows one to *predict* its continuation.

Consider a different example: Let the vector $\mathbf{x} \in \mathbb{R}^d$ be an image, where each vector component encodes the brightness of a certain image pixel. Only a tiny subset of such vectors correspond to images that look natural, whereas the vast majority of them look like random noise. Such *natural images* can be modelled as samples from an unknown probability distribution $p(\mathbf{x})$, and we can *generate* such images by drawing samples form this distribution. The distribution itself is unknown, but we can obtain information about it by observing n such samples $\mathcal{D} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$, for instance by downloading random images from the Internet. Learning in this case amounts to estimating $p(\mathbf{x})$ from the available samples. Knowing $p(\mathbf{x})$ generalizes the samples because it allows to *predict* how natural images look like in general.

The final example is closer to the topic of these notes, namely the problem of *analysing the content* of data. In the simplest case, the content is described by a binary class variable $c \in \{0, 1\}$. For example, $c = 1$ might mean that an image contains a "bicycle" (*positive* hypothesis) and $c = 0$ that it does not (*negative* or *null* hypothesis).

The relationship between data and labels is described by an unknown conditional probability distribution $P[c|\mathbf{x}]$. Similar to before, we obtain information about this distribution by drawing n samples $\mathcal{D} = ((\mathbf{x}_1, c_1), \ldots, (\mathbf{x}_n, c_n))$ from it.[1] For example, we could download from the Internet images that contain "bicycles" as well as other images that do not (both kind of images are required to learn a rule that tells bicycles from non-bicycles). The goal of learning is to estimate the conditional distribution $P[c|\mathbf{x}]$ from the available samples, capturing the *general* relationship between data $\mathbf{x}$ and labels c and

[1] More accurately we draw samples from the joint $p(\mathbf{x}, c) = P[c|\mathbf{x}]p(\mathbf{x})$.

thus *predicting* the labels of all possible images.[2]

In the last two examples, generalization was formalized as the problem of estimating certain (conditional) probability distributions from limited sample sets. Next, we introduce the perceptron as a mean to do this in practice.

2.2 The perceptron

Consider the simplest of the problems described above, namely binary classification. In this case, the goal is to estimate, given samples $\mathcal{D}$, the conditional probability $P[c = 1|\mathbf{x}]$ that any datum $\mathbf{x}$ belongs to the positive class.[3] Since $P[c = 1|\mathbf{x}]$ can be interpreted as a function f that maps data $\mathbf{x} \in \mathcal{X}$ to numbers $f(\mathbf{x}) \in [0, 1]$, we can regard this as the problem of estimating a function $f : \mathcal{X} \to [0, 1]$.

A reasonable condition on f is that it should predict accurately the labels of the given samples, i.e. $f(\mathbf{x}_i) \approx c_i$ for all $(\mathbf{x}_i, c_i) \in \mathcal{D}$. However, this tells us *nothing* about the value of f *out of sample*, namely for points $\mathbf{x}$ that are not in the sample set $\mathcal{D}$, and so this is useless for generalization.

To fix this problem, we restrict the choice of f to a family $\mathcal{F}$ of functions $\mathcal{X} \to [0, 1]$ that are defined *a-priori* on the entire input space $\mathcal{X}$. Then, we seek *within this family* for a function f that satisfies the constraints $f(\mathbf{x}_i) \approx c_i$ sufficiently well. In this manner, f is implicitly defined out of sample. Such a function may not generalize well, but it is at least defined for all inputs.

Concretely, the family $\mathcal{F}$ is obtained by considering a *parametric* function and varying its parameters. The perceptron is an important example of such a parametric function.

The *perceptron* $y = f(\mathbf{x}; \mathbf{w}, b)$ takes as input a vector $\mathbf{x} \in \mathbb{R}^d$ and returns as output a scalar y in the $[0, 1]$ range, just as we need.

[2]Note that we could, as before, try to recover $p(\mathbf{x})$ as well. This is often uninteresting in applications.

[3]The probability that a datum is negative is just the complement $1 - P[c = 1|\mathbf{x}]$ and does not need to be estimated separately.

Evaluating the perceptron comprises two steps. First, one computes the inner product between the input vector $\mathbf{x}$ and a parameter vector $\mathbf{w} \in \mathbb{R}^d$ and adds a constant *bias* $b \in \mathbb{R}$, resulting in the scalar *score*

$$\langle \mathbf{w}, \mathbf{x} \rangle + b = \mathbf{w}^\top \mathbf{x} + b.$$

Since the score can be any number from $-\infty$ to $+\infty$, the second step is to apply the *sigmoid* activation function to compress the score to the $[0, 1]$ interval.[4] The sigmoid function is given by the expression:

$$S(z) = \frac{1}{1 + e^{-z}}. \tag{2.1}$$

The perceptron is the combination of the two functions above:

$$f(\mathbf{x}; \mathbf{w}, b) = S(\langle \mathbf{w}, \mathbf{x} \rangle + b). \tag{2.2}$$

Shorthand for the bias term

It is possible to incorporate the bias term directly into the parameter vector $\mathbf{w}$. This is achieved by extending the input vectors $\mathbf{x}$ with an additional component equal to the constant 1 and $\mathbf{w}$ by a corresponding component equal to b. With this substitution, the perceptron can be written more compactly as

$$f(\mathbf{x}; \mathbf{w}) = S(\langle \mathbf{w}, \mathbf{x} \rangle).$$

In the rest of the chapter, we often use this convention to shorten the notation.

2.2.1 Linear classifiers

Dropping the sigmoid activation function from the perceptron (2.2) leaves us with a *linear scoring function* $h(\mathbf{x}; \mathbf{w}, b) = \langle \mathbf{w}, \mathbf{x} \rangle + b$. Such

[4] $S(z)$ is also known as logistic function.

functions can be used directly as predictor by looking at the *sign* of the score: a positive score $h(\mathbf{x}; \mathbf{w}, b) > 0$ is interpreted as predicting class $c = 1$ and a negative score as predicting class $c = 0$.

In this manner, the input space $\mathbb{R}^d$ is divided in two regions containing vectors predicted to be respectively positive and negative. The two regions are separated by a *decision boundary* of equation $h(\mathbf{x}; \mathbf{w}, b) = 0$. For a linear scoring function, the decision boundary is the hyperplane given by the linear equation $\langle \mathbf{w}, \mathbf{x} \rangle = -b$.

Predictors with a linear decision boundary are called *linear classifiers*. A perceptron is also a linear classifier, although the decision boundary is obtained by thresholding its probabilistic output at 50% rather than 0.

Intuitively, a linear classifier may be insufficient to correctly solve a complex prediction problem. This occurs because the *expressive power* of the predictor is insufficient to model the data. The expressive power of linear classifiers, including the perceptron, can be greatly extended by the use of data representations, as will be explained in section 2.4.

2.3 Learning the perceptron

In order to get the perceptron function $f(\mathbf{x}; \mathbf{w})$ to do something useful, such as recognizing bicycles in images, one must set the parameters $\mathbf{w}$ appropriately. Doing so manually is generally hopeless; instead, one automatizes this construction by *fitting* the parameters $\mathbf{w}$ to example data $\mathcal{D}$, a process known as *training* the model.

As explained before, the dataset $\mathcal{D}$ contains n samples $(\mathbf{x}_i, c_i)$ drawn from the unknown joint data-label probability distribution. Training should make the perceptron label all, or at least most, of the training examples correctly and confidently. Hence, one seeks for a parameter vector $\mathbf{w}^*$ such that $f(\mathbf{x}_i; \mathbf{w}^*) = \hat{P}[c = 1 | \mathbf{x}_i] = 1$ if the label of example $\mathbf{x}_i$ is $c_i = 1$ and $f(\mathbf{x}_i; \mathbf{w}^*) = 0$ otherwise. This parameter vector is usually characterized as the minimizer of

an energy function like:

$$E(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\left(f(\mathbf{x}_i;\mathbf{w}) - c_i\right)^2. \tag{2.3}$$

Note that the energy $E(\mathbf{w}) \geq 0$ is always non-negative and the minimum $E(\mathbf{w}^*) = 0$ is achieved if, and only if, all terms in the summation are identically zero. The latter in turn means $f(\mathbf{x}_i;\mathbf{w}^*) = c_i$ as required.

A minimizer of eq. (2.3) is also known as the *least square* fit of the preceptron function to the data. You are likely familiar with the concept of least square fit for *linear* models. This is the same, except that the perceptron is a non-linear function due to the sigmoid activation. It is therefore not possible to find a minimizer of (2.3) in closed form as in standard least square; instead, as explained in section 2.3.3, this is done by using methods such as gradient descent.

2.3.1 Generalization and regularization

If one can find a parameter vector $\mathbf{w}^*$ so that the energy function (2.3) is equal to zero, the resulting perceptron fits perfectly all the training data. Furthermore, the perceptron is defined out-of-sample by construction, so it computes *some* class probability value for all possible input vectors. Unfortunately, even this is insufficient to guarantee that generalization is good. Namely, given a second independent *test set* $\mathcal{D}_{\text{test}}$ of samples, the value of the energy (2.3) may be large even if the energy is small when evaluated on the training set $\mathcal{D}$.

The issue is that, while both training and test data $\mathcal{D}$ and $\mathcal{D}_{\text{test}}$ are drawn from the *same* distribution $P[c|\mathbf{x}]p(\mathbf{x})$, the distribution itself is unknown. Instead, the only information available for learning the perceptron are the training samples $\mathcal{D}$. Finding out when such samples are sufficient to learn a good predictor is the core problem of *statistical learning*, a very rich and complex field. While a discussion of the latter is out of scope, it is not difficult to build some

basic intuition here.

Recall that optimizing the parameters of the perceptron has the effect of *choosing a particular prediction function* f from a family of implementable functions $\mathcal{F}$ (given in our case by all possible perceptron functions as parameters are varied). The key intuition is that, if is this selection process is not "too specific" to the given training data $\mathcal{D}$, then the chosen function is likely to work well on test data as well.

A simple way to avoid overly-specific choices is to *reduce the space of available choices* before the optimization begins. So, instead of considering all possible perceptron parameters, one can constrain them to a smaller space of acceptable parameters. While this can be done in many ways, we are interested in a strategy which is simple, general, and still likely to allow fitting most training sets $\mathcal{D}$ well. A criterion that satisfies these requirements is to restrict functions to be *smooth* in some sense. Smoothness captures the idea that, in most applications, similar data vectors $\mathbf{x}$ tend to receive similar labels c, and so smooth predictor functions may generalize well.

Here, the appropriate approach is to look at the smoothness of the linear scoring function inside the perceptron. As a measure of smoothness, one can look at how much the score $\langle \mathbf{w}, \mathbf{x} \rangle$ can change in response to a change in the input vector $\mathbf{x}$. The amount of change can be bounded from above by using the *Cauchy-Schwarz inequality* as follows:

$$\left(\langle \mathbf{w}, \mathbf{x}' \rangle - \langle \mathbf{w}, \mathbf{x} \rangle\right)^2 = \left(\langle \mathbf{w}, \mathbf{x}' - \mathbf{x} \rangle\right)^2 \leq \|\mathbf{w}\| \cdot \|\mathbf{x}' - \mathbf{x}\|. \tag{2.4}$$

The scoring function is smoother if the output changes more slowly in response to a change in the input. Hence, this inequality shows that smoothens can be increased by reducing the *norm of the parameter vector* $\mathbf{w}$.

It is then possible to restrict the optimization to a subset of smooth perceptron functions by making sure that the norm of the parameter $\mathbf{w}$ is never too large. This is normally done by adding the squared norm $\|\mathbf{w}\|^2$ as a penalty or *regularization* term to (2.3),

resulting in the regularized energy function

$$E(\mathbf{w}) = \frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{|\mathcal{D}|}\sum_{(\mathbf{x},c)\in\mathcal{D}} \big(c - h(\mathbf{x};\mathbf{w})\big)^2. \tag{2.5}$$

The parameter λ controls smoothness, trading-off lowering the fitting error on the training data with the ability to generalize to future data. Formulation (2.5) is a particular example of *regularized risk minimization* known as *ridge regression*, where one balances squared error and squared norm regularizer.

2.3.2 Cross-entropy and other losses

The least square formulation (2.3) can be generalized by considering a different loss function. A *loss function* $\ell(y, c)$ compares the ground-truth class label c to the output y of the perceptron so that (1) $\ell(y, c) \geq 0$ is non-negative and (2) $\ell(y, c) = 0$ when y is a "correct" prediction for c. Generalizing eq. (2.3) to use a generic loss function, the energy becomes

$$E(\mathbf{w}) = \frac{1}{|\mathcal{D}|}\sum_{(\mathbf{x},c)\in\mathcal{D}} \ell(f(\mathbf{x};\mathbf{w}), c). \tag{2.6}$$

All loss functions are zero when predictions are perfect, but can weigh in different manner imperfect predictions, resulting in different trade-offs. In (2.3) the loss function is the *squared loss*:

$$\ell(y, c) = (y - c)^2.$$

Another popular loss is the *cross-entropy loss*, given by

$$\ell(y, c) = -c \log y - (1 - c)\log(1 - y). \tag{2.7}$$

This loss is generally preferable for classification problems because it very strongly discourages the prediction y from putting more mass on the incorrect class. For example, if $c = 1$, then the cross-entropy loss assigns infinitely-large loss to the (incorrect) estimate $y = 0$ whereas the squared loss assigns loss 1.

Consistent posterior probability estimators. An interesting property of the squared and cross-entropy losses is that they are consistent estimator of the true conditional class probabilities. To understand this, consider the case in which the label c itself is uncertain (for example, the label may be produced by a human annotator, and different values may be estimated by different annotators), so that $c = 1$ with some probability P_1. Then, for a prediction y, the *average value* of the loss is given by:

$$E_c[-c \log y - (1 - c) \log(1 - y)] = -P_1 \log y - (1 - P_1) \log(1 - y).$$

The minimum of the average can be found by setting the derivative of the right hand side of this expression to zero. In this manner, it is easy to show that the minimum is achieved for $y = P_1$. Hence minimising the expected loss results in the correct probability value being estimated. The squared loss has the same property.

Sigmoid cross-entropy and logistic regression. A perceptron trained using the cross-entropy loss can also be interpreted as a so-called *logistic regressor*. In order to do so, move the activation function $S(z)$ from the predictor function into the loss function. In this manner, the predictor reduces to the linear function $g(\mathbf{x}; \mathbf{w}) = \langle \mathbf{x}, \mathbf{w} \rangle$ and the loss becomes

$$\ell(y, c) = -c \log S(y) - (1 - c) \log(1 - S(y)).$$

Using the fact that $1 - S(y) = S(-y)$, this expression can be simplified to:

$$\ell(y, c) = -\log S(\bar{c}y) = \log \left(1 + e^{-\bar{c}y}\right), \quad \bar{c} = 2c - 1.$$

The latter is called the *logistic loss* and is applied directly to the output of a linear predictor.

Hinge loss and support vector machines. The *hinge loss* is similar in shape to the logistic loss, but is not smooth everywhere:

$$\ell(y, c) = \max\{0, 1 - \bar{c}y\}, \quad \bar{c} = 2c - 1.$$

Just like the logistic loss, the hinge loss can be used to train a linear predictor. A linear predictor learned using this loss in a regularized formulation such as eq. (2.5) is also called a *support vector machine* (SVM).

2.3.3 Optimization via gradient descent

We now consider the practical problem of learning a predictor by optimizing energies such as eq. (2.3). Due to the non-linearity of the perceptron, even the least square formulation (2.3) cannot be solved in closed form as in standard least square. Instead, one optimizes $E(\mathbf{w})$ incrementally, usually via a process known as *gradient descent*. Gradient descent is based on approximating around a current estimate $\mathbf{w}_t$ of the parameter vector via the *gradient* $\nabla E(\mathbf{w}_t)$ (first order derivative) of the energy function at that point, which results in the first-order Tayolor expansion $E(\mathbf{w}_{t+1}) \approx E(\mathbf{w}_t) + \langle \nabla E(\mathbf{w}_t), \mathbf{w}_{t+1} - \mathbf{w}_t \rangle$, also known as *linearization*. Updating the parameter $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla E(\mathbf{w})$ by subtracting a positive multiple $\eta > 0$ of the gradient reduces the approximated energy:

$$\begin{aligned} E(\mathbf{w}_{t+1}) &\approx E(\mathbf{w}_t) + \langle E(\mathbf{w}_t), \mathbf{w}_{t+1} - \mathbf{w}_t \rangle \\ &= E(\mathbf{w}_t) - \eta \|E(\mathbf{w}_t)\|^2 \\ &\leq E(\mathbf{w}_t). \end{aligned}$$

Since the approximation is valid only if $\mathbf{w}_{t+1}$ is sufficiently close to $\mathbf{w}_t$, the the *step size* η must be chosen sufficiently small. If this is done properly, by starting from an initial random parameter vector $\mathbf{w}_0$ and by iterating this update, one can gradually reduce the energy until a (locally) optimal parameter $\mathbf{w}^*$ is found.

As seen above, the key step in learning the perceptron is to compute the gradient of the terms $\ell(f(\mathbf{x}; \mathbf{w}), c)$ in the energy (2.6). Unpacking the perceptron, we see that each term is the composition of three functions: the loss, the sigmoid activation, and the inner product. Hence, in order to compute this derivative we need to use

the chain rule:

$$\frac{d\ell(f(\mathbf{x};\mathbf{w}),c)}{d\mathbf{w}} = \frac{d\ell}{dy}(f(\mathbf{x};\mathbf{w}),c) \cdot \frac{dS}{dz}(\langle \mathbf{w},\mathbf{x}\rangle) \cdot \frac{d\langle \mathbf{w},\mathbf{x}\rangle}{d\mathbf{w}}.$$

Note that each derivative is computed at the point determined by evaluating the function chain: $\ell(y,c)$ at $y = f(\mathbf{x};\mathbf{w})$, $S(z)$ at $z = \langle \mathbf{w},\mathbf{x}\rangle$, and $\langle \mathbf{w},\mathbf{x}\rangle$ at $\mathbf{x}$.

For example, if ℓ is the cross-entropy loss and if $c = 1$, the expression above simplifies to:

$$\frac{d\ell(f(\mathbf{x};\mathbf{w}),c)}{d\mathbf{w}} = \mathbf{x} \cdot \dot{S}(\langle \mathbf{w},\mathbf{x}\rangle) \cdot \begin{cases} -\frac{1}{S(\langle \mathbf{w},\mathbf{x}\rangle)}, & \text{if } c = 1, \\ +\frac{1}{1-S(\langle \mathbf{w},\mathbf{x}\rangle)}, & \text{if } c = 0. \end{cases}$$

Here the derivative of the sigmoid function is given by

$$\dot{S}(z) = \frac{e^z}{(1+e^z)^2} = S(z)(1-S(z)). \tag{2.8}$$

Recall that gradient descent adds a multiple of the negative gradient to the current parameter estimate $\mathbf{w}$. Hence, if $c = 1$, this adds to $\mathbf{w}$ a fraction of the vector $\mathbf{x}$, increasing the response of $\langle \mathbf{w},\mathbf{x}\rangle$ to this input. Furthermore, the speed of descent is modulated by the factor

$$G(\langle \mathbf{w},\mathbf{x}\rangle) \quad \text{where} \quad G(z) = \frac{\dot{S}(z)}{S(z)} = 1 - S(z).$$

Hence, if $\langle \mathbf{w},\mathbf{x}\rangle$ has already a strong positive response to input $\mathbf{x}$, the parameters are not changed by much because $G(z) \approx 0$. On the other hand, if $\langle \mathbf{w},\mathbf{x}\rangle$ has a strong negative (hence incorrect) response, the speed of change is maximum because $G(z) \approx 1$, which rapidly steers the scoring function to respond correctly. These considerations are reversed if $c = 0$.

Shrinkage. Note that, when regularization is used as in (2.5), then the gradient $\lambda\mathbf{w}$ of the regularizer must be considered as well. When subtracted from the parameter vector $\mathbf{w} - \eta\lambda\mathbf{w} = (1-\eta\lambda)\mathbf{w}$, the effect is to *shrink* the parameter towards zero. This is why in neural networks this form of regularization is often called *shrinkage*.

Stochastic gradient descent (SGD). Note that gradient descent requires to compute the gradient $\nabla E(\mathbf{w}_t)$ of the energy function at each step. In modern machine learning applications, the number of training samples n can be in the order of millions, so each evaluation of the gradient requires millions of evaluations of predictor, loss, and their gradients. Given that the energy is improved only slightly at each iteration of gradient descent, this is just too slow.

The solution to this problem is *stochastic gradient descent* (SGD). The idea is rather simple. Instead of evaluating the gradient exactly, one approximates it by considering a tiny subset $\mathcal{B} \subset \mathcal{D}$ of the training data at every iteration, called a *mini-batch*:

$$\nabla E(\mathbf{w}_t) \approx \frac{1}{|\mathcal{B}|} \sum_{(\mathbf{x},c)\in\mathcal{B}} \nabla \ell(f(\mathbf{x};\mathbf{w}), c) + \lambda \mathbf{w}. \tag{2.9}$$

Here the last term is the gradient of the squared norm regularizer. Given this approximated gradient, parameters are updated as usual.

In the limit, the mini-batch can comprise a single training sample at a time. However, there is a trade-off between how noisy the gradient estimate is and how quickly parameters are updated which is typically optimized for batches containing 10-1000 samples.

Batches are in principle randomly sampled, but in practice it is more effective to randomly permute the training data $\mathcal{D}$ and then partition the latter in mini-batches of equal size. A full pass through the training data is called an *epoch*. After each epoch, the data is shuffled again and the process is repeated.

Learnign rate schedule. The learning rate η is gradually decreased during (S)GD. While there are many possible *schedules*, a simple approach is to run (S)GD until there is no more progress in the energy function and then lower the learning rate by a factor of 10.

Momentum. As gradients are noisy, and since gradient descent does not have a way of "sensing" the curvature of the energy function as second-order methods would, several tricks have been developed to improve the conditioning of the algorithm.

One of the simplest and most effective is the idea of *momentum*. The momentum is a variable $\mathbf{m}_t$ containing a *moving average* of the gradient:

$$\mathbf{m}_{t+1} = \rho\mathbf{m}_t + (1-\rho)\nabla E(\mathbf{w}_t).$$

where $0 \leq \rho < 1$ is the inertia (often set to $\rho = 0.9$). Then parameters are updated as follows:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta\mathbf{m}_{t+1}.$$

2.4 Data representations

Perceptrons and other linear predictors such as SVMs can rarely be applied to data directly. First, often data is *not* presented in the form of vectors, so it is not directly compatbile with the perceptron. Second, even when data is in vectorial form, as it may be the case for an RGB-coded image, the perceptron may be too simplistic to successfully analyse the data as required.

This problem is usually addressed by considering a suitable *data representation*. A representation function Φ maps data sample $\mathbf{x}$ belonging to some space $\mathcal{X}$ to vectors $\Phi(\mathbf{x}) \in \mathbb{R}^d$. With this transformation, the perceptron can be applied to the data as follows:

$$f(\mathbf{x}; \mathbf{w}, \Phi) = S(\langle \mathbf{w}, \Phi(\mathbf{x}) \rangle).$$

The representation Φ can be handcrafted, or it can be though of as an additional parameter of the perceptron and can be optimized using the same energy as before (in *deep neural networks*, the representation itself is computed by a neural network, recursively).

The obvious advantage of using a representation is that it converts arbitrary data into vectors, so that a linear classifier can be

used. However, representations are much more powerful than that and are used even if the input space is a vector space. Representations can in fact dramatically expand the expressive power of a linear classifier by implementing a non-linear scoring rule:

$$h(\mathbf{x}; \mathbf{w}) = \langle \mathbf{w}, \Phi(\mathbf{x}) \rangle = \sum_{k=1}^{d} w_k \Phi_k(\mathbf{x}). \tag{2.10}$$

Each dimension of the representation can be though of as a basis function Φ_k and w_k as the corresponding coefficient.

While (2.10) may not seem all that expressive, quite the opposite is true. For example, if we pick the functions $\Phi_k(\mathbf{x})$ to be elements of a Fourier basis (i.e. sinusoidal functions), in the limit almost any function can be approximated in the form of $h(\mathbf{x}; \mathbf{w})$.

For machine learning applications, however, the ability to potentially approximate any function is not necessarily the most important property of a representation. As discussed in section 2.3.1, the most important goal is to make predictor generalize to future data, which is usually obtained via smoothing/regularization.

Representations play a fundamental role here as well as they determine what it means for functions to be smooth. Generalizing eq. (2.4), we have in fact:

$$\left(\langle \mathbf{w}, \Phi(\mathbf{x}') \rangle - \langle \mathbf{w}, \Phi(\mathbf{x}) \rangle\right)^2 \leq \|\mathbf{w}\| \cdot \|\Phi(\mathbf{x}') - \Phi(\mathbf{x})\|.$$

Hence, we see that two inputs $\mathbf{x}$ and $\mathbf{x}'$ are automatically assigned similar scores if their representation vectors $\Phi(\mathbf{x})$ and $\Phi(\mathbf{x}')$ are similar and (possibly) different scores otherwise.

This suggest that the representation should reflect the *congruency* of the input data in the similarity between representation vectors. Namely, two input points that are congruent, i.e. that have similar meaning and thus similar classification, should be mapped to similar vectors and two input points that are incongruent should be mapped to different vectors.

The latter often requires the representation to be *invariant* to irrelevant factors of variation in the data, also called *nuisance factors*. For instance, if the goal is to recognize bicycles in images, since the specific location of the bicycle is irrelevant, a representation which is invariant to image translations can be helpful in solving this problem.

Universal representations

Congruency is often too strong a condition because the same representation is often used to solve multiple and complementary analysis problem. For instance, we may use the same representation to tell whether an image contains a *car* or not, or a *red object* or not. Given that cars need not to be red and that red objects need not to be cars, it is clear that we cannot make $\Phi(\mathbf{x})$ congruent to *both* concepts at the same time.
Fortunately, we do not need to. In fact suffices for $\Phi(\mathbf{x})$ to *disentangle* the two concepts, by containing *subset of components* congruous to each concept separately. Then different linear projections $\langle \mathbf{w}, \mathbf{x} \rangle$ can easily extract the two concepts.

Chapter 3

Convolutional neural networks

Deep *Convolutional Neural Networks* (CNNs) are one of the most important families of modern neural networks. They have outstanding performance in image generation and analysis tasks, such as image classification, semantic segmentation, object detection, image denoising, colorization, image synthesis, face recognition, text spotting, pose recognition, scene understanding, and many more.

This chapter introduces CNNs. After a high-level overview of CNNs (section 3.1), section 3.2 is dedicated to discuss the concept of tensor, the fundamental data unit that is processed by CNNs. Tensors are processed by "convolutional" operators, which are discussed in detail in the second part of the chapter (section 3.3).

3.1 Filter chains

In the simplest case, CNNs operate on 3D arrays or tensors (section 3.2), with two spatial dimensions, height and width, and a channel dimension. A tensor is a generalization of an image and

associates to each spatial location a vector of features. In the case of an image, these feature vectors have three channels, capturing the intensity of the primary colours red, green and blue which specify the color of each pixel.

A CNN processes tensors via a chain of non-linear filters. These filters gradually transform the low-level RGB data, which captures raw light measurements, into progressively more abstract representations, capturing concepts such as edges, boundaries, textures, and eventually objects, enabling their recognition.

Formally, a CNN is as sequence of *layers* $f_1, f_2, \ldots, f_n$. Each layer takes as input a tensor $\mathbf{x}_{i-1}$ and produces as output a tensor $\mathbf{x}_i$:

$$\mathbf{x}_1 = f_1(\mathbf{x}_0), \quad \mathbf{x}_2 = f_2(\mathbf{x}_1), \quad \ldots, \quad \mathbf{x}_n = f_n(\mathbf{x}_n).$$

Most of the functions f_i are *local* and *translation invariant* operators and thus can be thought as filters. The prototypical layer is in fact *linear convolution* (section 3.3.1), but in general these filters may be non-linear.

Feature channels in tensors often are not interpretable; instead, their meaning is determined automatically by learning the CNN from example data, similar to the perceptron (section 2.3). An exception is the output of the last layer, $\mathbf{x}_n$, which generally conveys an explicit and desirable meaning. For instance, $\mathbf{x}_n$ can be a vector of class probabilities, encoding the CNN belief that the input image belongs to a certain class.

The next section discusses tensors in more detail, slightly generalizing the concepts reviewed so far.

3.2 Tensors

As we have suggested in the previous section, a CNN operates on *tensor* variables; while in many branches of mathematics "tensors" have a rich structure, in CNNs a tensor is simply a synonymy of multi-dimensional array.

Usually, tensors in CNNs have the format $\mathbf{x} \in \mathbb{R}^{N \times C \times U_1 \times \cdots \times U_D}$, where the first dimension indexes different data samples, the second dimension indexes feature channels, and the remaining dimensions index spatial locations. The tuple $N \times C \times U_1 \times \cdots \times U_D$ listing the dimensions of the tensor is also called the *tensor shape*. Dimension N is often called **batch size** and is used to concatenate several input samples for efficient parallel processing, for example to compute mini-batches in stochastic gradient descent (section 2.3.3). The number of feature channels C is arbitrary in general, although in some cases it has a specific meaning (e.g. $C = 3$ when representing RGB images). Often, as for a standard image, there are two spatial dimensions, so that $U_1 = H$ is called the **height** of the tensor and $U_2 = W$ its **width**. Other common cases is to have 1 or 3 spatial dimensions; in the latter case, the third dimension U_3 is usually called **depth** and can be used to represent volumetric data, common in medical imaging.

Note that the conventions above are common but not universal. Tensor dimensions can be assigned different meanings by different CNN architectures. For example, sometimes in some applications it is useful to use specific dimensions to represent time rather than space.

In order to refer to a specific component of a tensor $\mathbf{x}$, we will use the notation x_{ncu}, where $0 \leq n < N$ is the sample index, $0 \leq c < C$ the channel index, and $(0, \ldots, 0) \leq u < U = (U_1, \ldots, U_D)$ the location multi-index. Note that, unless otherwise specified, tensor elements are indexed starting from 0 rather than 1 — the reason is that this often simplifies calculations substantially.

A scalar quantity $x \in \mathbb{R}$ can be interpreted as a tensor with a single dimension equal to 1. However, it can equivalently be interpreted as a tensor with an arbitrary number of dimensions all equal to 1 (i.e. a $1 \times 1 \times \cdots \times 1$ tensor,[1] or a tensor with no dimensions at all. Since the latter is the convention used by the PyTorch tool-

[1] More in general any tensor with dimensions U can be thought of as a tensor with dimensions $U \times 1 \times \cdots \times 1$.

box, we will adopt it here. The reason for distinguishing $\mathbb{R}$, $\mathbb{R}^1$ and $\mathbb{R}^{1\times 1\times\cdots\times 1}$ is that, while they are isomorphic (equivalent) spaces, they are treated differently by operations that manipulate the tensor dimensions, such as reshaping, slicing, or concatenating.

Tensors in practice

Tensors are a fundamental building block in many machine learning toolboxes. The notation we use here directly maps to tensors as represented in PyTorch. Other toolboxes may use equivalent but different conventions. For example, MatConvNet uses the $HWCN$ order (instead of $NCHW$) and 1-based indexing (instead of 0-based). The existence of different conventions is unfortunate, but usually it is not a source of major confusion.

3.2.1 The vec operator

The different dimensions of a tensor organize data components by meaning, for example by associating them to different spatial locations, feature channels, or data samples. However, when we look at tensors as variables in a calculation, for example to compute the derivatives required in gradient descent, these distinctions are moot. In such cases, it is sometimes useful to reinterpret tensors as mere data blobs or vectors.

A tensor is converted into a vector by the *vec* or *stacking* operator

$$\operatorname{vec} \quad : \quad \mathbb{R}^{U_1\times\cdots\times U_D} \to \mathbb{R}^{\prod_{d=1}^{D} U_d}. \tag{3.1}$$

If $\mathbf{y}$ is a tensor, then $\operatorname{vec}\mathbf{y}$ is a vector whose entries are obtained by scanning in some conventional order the entries of $\mathbf{y}$. Hence, the dimension of $\operatorname{vec}\mathbf{y}$ is the product of the dimensions of $\mathbf{y}$.

An important application of vec is to transform tensors in to vectors so that the standard real analysis notation (gradients, Jacobian

matrices, etc.) can be used to work out the derivatives of neural networks (section 4.2).

Conventions for vectorizing tensors

The specific scanning order used to vectorize a tensor, which determines how the elements of $\mathbf{y}$ are mapped to elements of $\operatorname{vec}\mathbf{y}$, is not important, provided that it is fixed. In practice, most programming environments provide the ability to reshape tensors (or multi-dimensional arrays) using a certain default order, which normally corresponds to the order in which the tensor elements are stored in memory. There are two main conventions for the scanning order:

- **Row major.** Dimensions are scanned from the right to the left, i.e. $(y_{0,\dots,0,0}, y_{0,\dots,0,1}, y_{0,0,\dots,2}, \dots, y_{0,\dots,1,0}, \dots)$ This is the convention used by PyTorch and Caffe.
- **Column major.** Dimensions are scanned from the left to the right, i.e. $(y_{0,0,\dots,0}, y_{1,0,\dots,0}, y_{2,0,\dots,0}, \dots, y_{0,1,\dots,0}, \dots)$ This is the convention used by MATLAB and MatConvNet.

Packages that use opposite scanning orders also tend to use opposite conventions for ordering dimensions in a tensor. So for example PyTorch and Caffe use the dimension order $N \times C \times H \times W$, whereas MatConvNet uses $H \times W \times C \times N$. As a consequence of this "double reversal", stacked tensors end-up being almost the same in the two cases (almost because the width and height dimensions are still swapped). This is likely due to the fact that the stacking order is also the order of the tensor elements in memory and certain layouts are preferable for the hardware implementing the calculations.

3.2.2 Slices

Slicing a tensor is another common operation. Slicing amounts to considering a subtensor obtained by fixing some dimensions to given values and letting the other ones vary. We will use a slightly informal notation for slicing. Namely, let $\mathbf{x} \in \mathbb{R}^{U_1 \times \cdots \times U_M}$ be a tensor. The tensor $\mathbf{x}_{u_1:}$ denotes the slice of $\mathbf{x}$ obtained by fixing the first index to value u_1:

$$[\mathbf{x}_{u_1:}]_{u_2 \cdots u_M} = x_{u_1 u_2 \cdots u_M}.$$

The notations $\mathbf{x}_{:u_M}$ and $\mathbf{x}_{:u_k:}$ are similar, but denote fixing the last or intermediate indexes. Fixing several dimensions is also possible, as in $\mathbf{x}_{u_1:u_k:u_M}$, with obvious meaning.

3.3 Layers

This section describes the most important layer types in CNNs.

3.3.1 Linear convolution

The linear convolution operator takes as input a tensor $\mathbf{x} \in \mathbb{R}^{N \times C \times I}$ and a filter bank $\mathbf{f} \in \mathbb{R}^{K \times C \times F}$ and produces as output the tensor $\mathbf{y}$ given by:

$$y_{nkv} = \sum_{c=0}^{C-1} \sum_{u=0}^{F-1} f_{kcu} \cdot x_{n,c,v+u}. \tag{3.2}$$

This expression has the obvious meaning for tensors with a single spatial dimension ($D = 1$). For more than one dimension, u must be interpreted as a multi-index and is summed over the product of intervals $\Pi_{i=1}^{D}[0, F_i - 1]$. A more explicit version of this expression for the multi-dimensional case is thus given by:

$$y_{nkv_1 \cdots v_D} = \sum_{c=0}^{C-1} \sum_{u_1=0}^{F_1-1} \cdots \sum_{u_D=0}^{F_D-1} f_{kcu} \cdot x_{n,c,v_1+u_1,\ldots,v_D+u_D}.$$

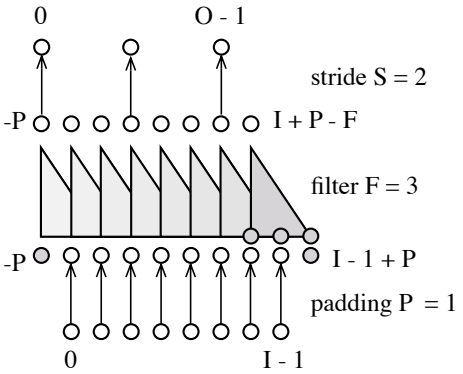

Figure 3.1: Calculation of the size of the output of a convolution-like layer for a 1D tensor. From bottom to top: the 1D input tensor has I samples; padding ($P = 1$ in the example) adds P samples to each side; applying a filter of size F drops $F - 1$ samples to the right; downsampling ($S = 3$ in the example) retains one every S samples.

Note that working with multiple spatial dimensions is a direct and trivial extension of the 1D case. Likewise, processing multiple input samples as indexed by n is a trivial extension of processing a single sample. Hence, in order to simplify the notation, in the rest of the section we will assume $D = 1$ and $N = 1$ and drop the sample index n. In this case, expression (3.2) simplifies to:

$$y_{kv} = \sum_{c=0}^{C-1} \sum_{u=0}^{F-1} f_{kcu} \cdot x_{c,v+u}.$$

Tensor dimensions. Each filter $\mathbf{f}_{k:}$ in the bank processes the whole input tensor $\mathbf{x}$ to produce a specific channel $\mathbf{y}_{k:}$ of the output; therefore, the number K of filters in the bank is independent of the shape of $\mathbf{x}$ and sets the number of output channels. On the

other hand, the filter bank and the input tensor must have the same number of channels C.

Next, we compute the spatial dimensions of the output $\mathbf{y}$ as a function of the input and the filter bank shapes (see also fig. 3.1). Each output element $y_{:v}$ is computed by applying a filter to a window of the input tensor $\mathbf{x}$, starting at location v and ending at location $v + U - 1$. Since the largest such locations must still be within the bounds of the input tensor and since the latter has has size I, it must be $v + U - 1 \leq I - 1$. Hence, v, which indexes output elements, must be in the interval $0 \leq v \leq I - U$. We conclude that the output size is $O = I - U + 1$.

Convolution vs correlation

In mathematics and signal processing, eq. (3.2) does not define the convolution operator, but rather then *correlation* operator. The only significant difference is that convolution is defined by flipping (spatially) the filters. In these notes we still call eq. (3.2) convolution as this is standard in the CNN literature.

Padding, stride, and dilation. The basic convolution operator can be extended to include three useful features:

- **Padding.** The input tensor can be virtually padded with P_- **zeros** to the left and P_+ zeros to the right, allowing the filter to slide more to the left and to the right than otherwise possible. For example, if the filter bank size $F = 2P + 1$ is odd, then using $P_- = P_+ = P$ as padding causes the input and output tensors to have the same size (instead of having a smaller output).

- **Stride.** It is possible to retain only one every S elements of the output tensor, thus sampling the output with stride S and reducing the resolution of the output tensor by the same

factor. This is useful because after convolution retaining the full resolution is usually overkill.

- **Dilation.** It is also possible to dilate the filter support without increasing the filter size. This is done by skipping over D input locations for every filter spatial location, a technique borrowed from the wavelet literature also known as "à trous" or "perforated".

In order to account for padding, stride, and dilation, eq. (3.2) is modified to

$$\boxed{y_{kv} = \sum_{c=0}^{C-1} \sum_{u=0}^{F-1} f_{kcv} \cdot x_{c,Dv+Sv-P_-}} \tag{3.3}$$

The following lemma gives shape of the output tensor with these modifications:

Lemma 1. *The size O of the output of a filter of size F applied to an input of size I with padding (P_-, P_+), stride S, and dilation D is given by:*

$$O = 1 + \left\lfloor \frac{I + P_+ + P_- - D \cdot (F-1) - 1}{S} \right\rfloor. \tag{3.4}$$

Proof. Since the input tensor is virtually extended by P_- and P_+ pixels to the left and to the right, we must have the following constraints on the spatial index applied to $\mathbf{x}$:

$$-P_- \leq S \cdot 0 + Dv - P_- \quad \text{and} \quad D \cdot (F-1) + Sv - P_- \leq I + P_+ - 1.$$

Rearranging, the output index v must vary in the range:

$$0 \leq v \leq \left\lfloor \frac{I + P_+ + P_- - D \cdot (F-1) - 1}{S} \right\rfloor.$$

From this expression, we immediately derive eq. (3.4). □

3.3.2 Activation functions

Activation functions are scalar functions $y = S(x)$ applied component-wise to tensors. Given an input tensor $\mathbf{x}$, we will use the "broadcast" notation $\mathbf{y} = S(\mathbf{x})$ to denote the effect of applying the function to each entry of the input, i.e.:

$$y_{ncu} = S(x_{ncu}).$$

Note that the output tensor $\mathbf{y}$ has the same shape as the input $\mathbf{x}$.

Activation functions are usually non-linear. The perceptron uses the *sigmoid* function, already given in eq. (2.1) and repeated here for convenience:

$$S(z) = \frac{1}{1 + e^{-z}}.$$

Modern neural networks prefer the *Rectified Linear Units* (ReLU), given by

$$S(z) = \max\{0, z\}. \tag{3.5}$$

ReLU exists in several variants, such as the *soft ReLU*

$$S(z) = \log(1 + e^z) \tag{3.6}$$

and the *leaky ReLU*

$$S(z) = \epsilon z + (1 - \epsilon) \max\{0, z\}, \tag{3.7}$$

where $0 < \epsilon \ll 1$ is a small positive number. Another choice popular in certain applications is the *hyperbolic tangent* (tanh) activation:

$$S(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{3.8}$$

3.3.3 Pooling

The *max pooling* computes the maximum of each channel of an input tensor in small sliding windows:

$$y_{cv} = \max_{0 \le u < F} x_{c,v+u}. \tag{3.9}$$

Geometrically, the operation is similar to linear convolution. In particular, the shape of the output tensor can be computed in the same manner as in section 3.3.1: if the input tensor has dimensions $C \times I$ and the pooling window has dimension F, the output tensor has dimensions $C \times O$ where $O = I - F + 1$.

Pooling can be implemented with operators other than max. The general form is

$$y_{cv} = S\left(\operatorname*{pool}_{0 \leq u < F} x_{c,v+u}\right). \tag{3.10}$$

For example, *average pooling* is given by:

$$y_{cv} = \frac{1}{F_1 \times \cdots \times F_D} \sum_{0 \leq u < F} x_{c,v+u}. \tag{3.11}$$

Sum pooling is the same as average pooling, except that the result is not normalized. *Squared sum pooling* is given by

$$y_{cv} = \sqrt{\sum_{0 \leq u < F} x_{c,v+u}^2}. \tag{3.12}$$

Just like for convolution in eq. (3.3), it is possible to introduce padding (P_-, P_+), stride S and even dilation D for pooling operators (although the latter is not commonly used):

$$y_{cv} = S\left(\operatorname*{pool}_{0 \leq u < F} x_{c,Sv+Du-P_-}\right). \tag{3.13}$$

In this case, lemma 1 can be used to compute the output dimension. There is a subtelty with padding: the input tensor $\mathbf{x}$ is (virtually) padded with different values depending on the pooling operator: $-\infty$ for max-pooling and 0 for the others.

3.3.4 Softmax

The perceptron was introduced as a predictor suitable for *binary* (two-class) classification problems. There, the sigmoid function was

used to convert a real class score into a probability value. However, in applications we are often interested in classification problems where there are $C > 2$ classes. In this case, we can use the *softmax* function to convert a vector of scores $\mathbf{x} \in \mathbb{R}^C$ into a vector of probabilities $\mathbf{y} \in [0,1]^C$ as follows:

$$y_c = \frac{e^{x_c}}{\sum_{k=0}^{C-1} e^{x_k}}. \tag{3.14}$$

The output vector can now interpreted as a probability vector because it is non-negative and because it sums to one ($\sum_{c=0}^{C-1} y_c = 1$).

Shift invariance. An important property of the softmax operator is that it is *shift invariant*; namely, one can subtract a constant M to all elements of the input tensor $\mathbf{x}$ without changing the output:

$$y_c = \frac{e^{x_c - M}}{\sum_{k=0}^{C-1} e^{x_k - M}} = \frac{e^{-M} e^{x_c}}{e^{-M} \sum_{k=0}^{C-1} e^{x_k}} = \frac{e^{x_c}}{\sum_{k=0}^{C-1} e^{x_k}}. \tag{3.15}$$

Numerical stability

Computing the softmax operator may lead to numerical instabilities due to the exponential functions and the division. A simple trick to stabilize softmax is to subtract from the input $\mathbf{x}$ its maximum component $M = \max_c x_c$. Due to shift invariance (3.15), this does not change the result. The advantage is that in this manner the numerator is at most 1 and that the denominator is at least 1 and at most C.

Relation to sigmoid. For a two-class problems, $C = 2$ and the softmax output is given by:

$$\begin{aligned} y_0 &= \frac{e^{x_0}}{e^{x_0} + e^{x_1}} = \frac{1}{1 + e^{-(x_0 - x_1)}} = S(x_0 - x_1), \\ y_1 &= 1 - S(x_0 - x_1) = S(x_1 - x_0), \end{aligned}$$

where S is the sigmoid function. This shows that softmax is a direct generalization of the sigmoid. It also shows that, for a binary classification problem, one can use the softmax operator on top of two class scores x_0 and x_1, but this is not very meaningful as only the score difference $x_0 - x_1$ matters.

Multi-dimensional softmax. Equation (3.14) is defined for a vector input $\mathbf{x}$ and produces a normalized version of the vector $\mathbf{y}$ as output. However, in CNNs we are interested in defining operators on tensors. Softmax is extended to operate on tensors by normalizing *slices* of the input tensor independently.

In applications, where the dimensions $N \times C \times U$ of a tensor $\mathbf{x}$ are interpreted as sample, channel, and spatial, softmax is often applied along the channel dimension, to produce a field of class probabilities for dense image labelling (segmentation):

$$y_{ncu} = \frac{e^{x_{ncu}}}{\sum_{k=0}^{C-1} e^{x_{nku}}}. \tag{3.16}$$

Another common case is to sum over spatial dimensions, to produce a set of heat-maps for localizing a bank of objects or keypoints in an image:

$$y_{ncv} = \frac{e^{x_{ncv}}}{\sum_{u=0}^{U-1} e^{x_{ncu}}}. \tag{3.17}$$

These definitions can be generalized as follows. Given a tensor of dimensions $U_1 \times \cdots \times U_M$, a slice is a selection $D \subset (1, \ldots, M)$ of the list[2] of the M dimensions. Using the multi-index $u = (u_1, \ldots, u_M)$ to index entries in the tensor and using the shorthand $u_D = (u_i : i \in D)$ for the part of the multi-index spanning the selected slice, the softmax operator can be written as

$$y_v = \frac{e^{x_v}}{\sum_{u:u_D=v_D} e^{x_u}}. \tag{3.18}$$

[2]Mathematically, a list is a sequence of tuple.

3.3.5 Losses

In order to learn a CNN, its output must ultimately be converted to a scalar score E that is then optimized. Normally, this score is the result of composing the CNN with a loss function $\ell(\mathbf{y}, c)$, which compares the CNN output $\mathbf{y}$ to some ground-truth output value c. This section discusses the most common losses used in applications, starting from a discussion of the tensor dimensions involved.

Dimensions. Different losses interpret data in a different manners, which has an effect on the dimensions of the corresponding tensors. For instance, in a classification problem, the tensor $\mathbf{y} \in [0,1]^C$ is a vector of class probabilities and $c \in \{0, \dots, C-1\}$ is a single ground-truth class index. In a regression problem, on the other hand, $\mathbf{y}, \mathbf{c} \in \mathbb{R}^C$ are two vectors.

In practice, one seldom works with simple vectors. Most losses operate in fact by averaging errors over samples in a batch. In this case, the tensor $\mathbf{y} \in \mathbb{R}^{N \times C}$ represents N C-dimensional outputs, one for each sample in a batch. In this case, for classification-like problems individual outputs are compared to a vector of ground truth values $\mathbf{c} \in \mathbb{R}^N$ and the results averaged:

$$\ell(\mathbf{y}, \mathbf{c}) = \frac{1}{N} \sum_{n=0}^{N-1} \ell(\mathbf{y}_{n:}, c_n).$$

For regression-like problems, $\mathbf{c} \in \mathbb{R}^{N \times C}$ has the same dimensionality as $\mathbf{y}$, and the expression is changed accordingly to compare corresponding tensor slices.

This can be generalized further. For instance, it is not uncommon for a network to estimate a *field of values* (for example, pixel-level classes in image segmentation). In this case, the loss is applied to certain slices of the tensor $\mathbf{y}$ and the results are averaged. The most common of these examples is to average over the batch and spatial

indexes:

$$\ell(\mathbf{y}, \mathbf{c}) = \frac{1}{NU_1 \cdots U_D} \sum_{n=0}^{N-1} \sum_{u=0}^{U-1} \ell(\mathbf{y}_{n,:,u}, c_{nu}).$$

Slicing conventions

Note that, in the last example, $\mathbf{y}$ has dimensions $N \times C \times U_1 \times \cdots \times U_D$ whereas $\mathbf{c}$ has dimensions $N \times U_1 \times \cdots \times U_D$. An alternative convention, used by some machine learning toolboxes, is that the slicing dimension C in $\mathbf{y}$ is mapped to a singleton dimension in $\mathbf{c}$, which is therefore $N \times 1 \times U_1 \times \cdots \times U_D$. The singleton dimension is redundant, but has the advantage of keeping the dimensions of $\mathbf{y}$ and $\mathbf{c}$ aligned.

Normalization factors

While dividing the loss output by the number of elements in a batch (and/or the number of spatial locations) is common, in some cases these factors are suppressed. This may be used, for example, to facilitate the distributed calculation of a loss over multiple GPUs.

Cross-entropy loss. A CNN configured for image classification produces a tensor $\mathbf{y} \in [0,1]^{N \times C}$ of class probabilities, compared to a corresponding vector $\mathbf{c} \in \{0, \ldots, C-1\}^N$ of ground-truth class labels. In this case a common loss is the *cross-entropy loss*, given by:

$$\ell(\mathbf{y}, \mathbf{c}) = -\frac{1}{N} \sum_{n=0}^{N-1} \log y_{n,c_n} \tag{3.19}$$

Multi-class logistic loss. If $\mathbf{y}$ is computed from a score vector $\mathbf{x}$ using the softmax operator (3.14), then naively composing with

the loss (3.19) may be numerically unstable. Instead, one uses the *multi-class logistic loss*, which combines softmax and cross-entropy into a single expression:

$$\ell(\mathbf{x}, \mathbf{c}) = \frac{1}{N} \sum_{n=0}^{N-1} \left[-x_{n,c_n} + \log \sum_{k=0}^{C-1} e^{x_{nk}} \right] \tag{3.20}$$

As for softmax, this expression can be stabilized numerically by subtracting $M_n = \max_c x_{nc}$ from $\mathbf{x}$ before calculation.

3.4 Receptive fields

A key property of many neural network layers is to be *local*. A layer $\mathbf{y} = f(\mathbf{x})$ is local if every element of $\mathbf{y}$ depends only on a subset of the elements of $\mathbf{x}$.

Locality is usually *spatial* as many layers operate as filters: each element y_v of the output, depends on a corresponding window x_u, $u \in \Omega_v$ of elements of the input. The set Ω_v is called the *receptive field* of the "neuron" or operator outputting y_v.

In this section, we show how to compute the receptive field for the most common neural network layers and of their composition. The latter is required to determine which image pixels influence a certain neuron deep down a multi-layer CNN.

3.4.1 The receptive field of a layer

In this section we compute the receptive field of a single layer $\mathbf{y} = f(\mathbf{x})$. Many layers can be interpreted as non-linear filter. In such cases, as shown by eq. (3.3), the output element y_v depends on the following elements x_ω in the input tensor:

$$x_\omega \quad \text{where} \quad \omega = vS + uD - P_- \quad \text{and} \quad 0 \leq u < F.$$

Thus the relevant input elements x_ω are in the window $\Omega_v = [\omega_v^-, \omega_v^+]$, which depends on the filter size F, the left padding P_-, the stride S

and the dilation D. Plugging $u = 0$ and $u = F - 1$ in the equation above gives us the smallest and largest input indices ω involved in the calculation of y_v:

$$\begin{aligned} \omega_v^- &= vS - P_-, \\ \omega_v^+ &= vS - P_- + (F-1) \cdot D. \end{aligned} \tag{3.21}$$

Rather than manipulating these two quantities directly, it is easier to consider the receptive field *size* and *center*, given by:

$$\begin{aligned} R = \omega_v^+ - \omega_v^- + 1 \quad &= (F-1) \cdot D + 1, \\ \mu_v = \frac{\omega_v^+ + \omega_v^-}{2} \quad &= vS - P_- + \frac{R-1}{2}. \end{aligned} \tag{3.22}$$

With these two quantities, we can express the receptive field as $\Omega_v = [\mu_v - (R-1)/2, \mu_v + (R-1)/2]$. The receptive field size is the same as the filter size after dilation has been applied to it (so if the dilation is $D = 1$, then $R = F$). The receptive filter size R does not depend on the output location v because filters are spatially invariant and so operate in the same way at all spatial locations.

Centered filters

Due to boundary effects, filters may apply a spatial shift to the information in the tensors. In the calculations above, this is captured by the left padding P_-. In particular, if the left padding is half the receptive field, or more exactly $P_- = (R-1)/2$, which requires R to be odd, then the filter is *centered*. By this, we mean that every output element y_v depends on a window centered at location $\mu_v = Sv$ in the input, so there is no shift. This can be better appreciated if the stride is $S = 1$, as in this case $\mu_v = v$.

Activation functions

Element-wise layers such as activation functions can be though of as filters of size $F = 1$, dilation $D = 1$, and stride $S = 1$. Hence, their receptive field has size $R = 1$ and it is centered, in the sense that $\mu_v = v$.

3.4.2 Receptive field of two layers

Next, we consider the composition $\mathbf{z} = \Phi \circ f$ of a subnetwork Φ with a filter-like layer $\mathbf{y} = f(\mathbf{x})$. We can visualize the dependency chain as follows:

$$\{x_\omega,\ \omega \in \Omega_v\} \overset{f}{\mapsto} \{y_\lambda,\ \lambda \in \Lambda_v\} \overset{\Phi}{\mapsto} z_v.$$

In other words, the output element z_v depends on a window Λ_v of elements of the intermediate tensor $\mathbf{y}$, or on a window Ω_v of elements in the tensor before it in the chain, namely $\mathbf{x}$.

Assume that we are given window $\Lambda_v = [\lambda_w^-, \lambda_w^+]$. We can calculate Ω_v exactly as we did in the previous question. In this case, we must find which elements x_ω influence of any of the elements $y_\lambda, \lambda \in \Lambda_v$. As before, this depends on the parameters (F, P_-, S, D) of the filter f. The minimum and maximum ω are given

$$\begin{aligned}\omega_v^- &= \lambda_v^- S - P_-,\\ \omega_v^+ &= \lambda_v^+ S - P_- + \hat{F} - 1,\end{aligned}$$

where $\hat{F} = D \cdot (F - 1) + 1$ is the *dilated filter size*.

These formulas allow us to compute the receptive field *recursively*, from the end of the network back to the input. As before, these formulas can be simplified by considering the receptive field size and center instead of the extrema. The receptive field size R in $\mathbf{x}$ is given by

$$R = \omega_v^+ - \omega_v^- + 1 = (\lambda_v^+ - \lambda_v^+)S + \hat{F} - 1 = R' \cdot S + \hat{F} - 1$$

where R' is the receptive field size in $\mathbf{y}$. The receptive field center μ_v in $\mathbf{x}$ is given by

$$\mu_v = \frac{\omega_v^+ + \omega_v^-}{2} = \frac{\lambda_v^+ + \lambda_v^+}{2} S - P_- + \frac{\hat{F} - 1}{2} = \mu'_v S + \frac{\hat{F} - 1}{2} - P_-$$

where μ'_v is the receptive field center in $\mathbf{y}$.

3.4.3 Receptive field of several layers

Consider a deep neural network formed by the chain

$$\mathbf{x}_0 \overset{f_1}{\longmapsto} \mathbf{x}_1 \overset{f_2}{\longmapsto} \dots \overset{f_n}{\longmapsto} \mathbf{x}_n.$$

The receptive field of an element in the output tensor $\mathbf{x}_n$ within the same tensor is obviously the element itself, so we have $(R_n, \mu_{vn}) = (1, v)$. Then, we can find the receptive field sizes in tensors $\mathbf{x}_{n-1}$, $\mathbf{x}_{n-2}$,... as:

$$\begin{aligned}
R_{n-1} &= (R_n - 1)S_n + \hat{F}_n = \hat{F}_n \\
R_{n-2} &= (R_{n-1} - 1)S_{n-1} + \hat{F}_{n-1} \\
&= (\hat{F}_n - 1)S_{n-1} + (\hat{F}_{n-1} - 1) + 1 \\
R_{n-3} &= (R_{n-2} - 1)S_{n-2} + \hat{F}_{n-2} \\
&= (\hat{F}_n - 1)S_{n-1}S_{n-2} + (\hat{F}_{n-1} - 1)S_{n-2} + (\hat{F}_{n-2} - 1) + 1 \\
&\vdots \\
R_0 &= 1 + \sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} S_j \right) (\hat{F}_i - 1)
\end{aligned}$$

Note that the total receptive field size is given by the sum of the individual filter sizes *rescaled* by the product of filter strides. Hence, downsampling results in much larger receptive fields. A similar rea-

soning gives use the centers:

$$
\begin{aligned}
\mu_{v,n-1} &= vS_n + \frac{\hat{F}_n - 1}{2} - P_{n,-} \\
\mu_{v,n-2} &= \mu_{v,n-1}S_{n-1} + \frac{\hat{F}_{n-1} - 1}{2} - P_{n-1,-} \\
&= vS_nS_{n-1} + \left(\frac{\hat{F}_n - 1}{2} - P_{n,-}\right)S_{n-1} + \left(\frac{\hat{F}_{n-1} - 1}{2} - P_{n-1,-}\right) \\
&\vdots \\
\mu_{v0} &= v \cdot \left(\prod_{j=1}^{n} S_j\right) + \sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} S_j\right) \left(\frac{\hat{F}_i - 1}{2} - P_{i,-}\right)
\end{aligned}
$$

The stride of the total receptive field is the product of the strides along the chain. Furthermore, in order ot have a "centered" receptive field (so that the second part of the last equation goes to zero) the (dilated) filter size and the left padding must satisfy the relation $\hat{F}_i = 2P_{i,-} + 1$.

Receptive fields: 'of' and 'in'

The calculations above give the receptive fields (R_i, μ_{vi}) of the function $f_n \circ \cdots \circ f_{i+1}$ that maps a certain intermediate tensor $\mathbf{x}_i$ to the output tensor $\mathbf{x}_n$. We thus say that this is the receptive field *of* $\mathbf{x}_n$ *in* $\mathbf{x}_i$. In particular, if $i = 0$, then this is the receptive field of the output of the deep neural network into the input image (or tensor).

Often, we are interested in computing different receptive fields. In particular, it is interesting to consider the function $f_i \circ f_{i-1} \cdots \circ f_1$ that goes from the input image/tensor $\mathbf{x}_0$ to a certain intermediate tensor $\mathbf{x}_i$. In this case, we are may want to compute the receptive field of $\mathbf{x}_i$ in $\mathbf{x}_0$. Can you see how to use the formulas above to do so?

Chapter 4

Automatic differentiation

The goal of automatic differentiation (shortened AutoDiff) is to efficiently and transparently compute the derivative of functions resulting from the calculations performed in a program. In our context, the program evaluates a certain deep neural network, and AutoDiff provides automatically the gradients required for optimization.

This chapter provides a high-level introduction to AutoDiff (section 4.1). It then develops a notation for the derivative of tensor functions (section 4.2) and uses it to illustrate in more detail how AutoDiff works (section 4.3). It also discusses the backpropagation algorithm (section 4.4), the key building block of AutoDiff.

4.1 Overview of AutoDiff

When training a CNN, one start from an image $\mathbf{x}_0$, applies to it a sequence of operations $\mathbf{x}_1 = f_1(\mathbf{x}_0, \mathbf{w}_1)$, $\dots$, $\mathbf{x}_n = f_n(\mathbf{x}_{n-1}, \mathbf{w}_n)$ to evaluate the CNN, and eventually computes the loss $\ell(\mathbf{y}, c)$ to asses the quality of the CNN prediction with respect to the ground truth

label c.

This sequence of steps can be summarised in pseudo-code as follows:

Require: tensors $c, \mathbf{x}_0, \mathbf{w}_1, \dots, \mathbf{w}_n$
1: $\mathbf{x}_1 = f_1(\mathbf{x}_0, \mathbf{w}_0)$
2: $\mathbf{x}_2 = f_2(\mathbf{x}_1, \mathbf{w}_1)$
3: $\cdots$
4: $\mathbf{x}_n = f_n(\mathbf{x}_{n-1}, \mathbf{w}_n)$
5: $E = \ell(\mathbf{x}_n, c)$

In fact, we can write this more compactly by iterating over the CNN layers using a for loop:

Require: tensors $c, \mathbf{x}_0, \mathbf{w}_1, \dots, \mathbf{w}_n$
1: **for** $i = 1, \dots n$ **do**
2: $\quad \mathbf{x}_i = f_i(\mathbf{x}_{i-1}, \mathbf{w}_i)$
3: **end for**
4: $E = \ell(\mathbf{x}_n, c)$

A neural network toolbox such as PyTorch or MatConvNet allows to evaluate the operators f_i, and thus compute E, given the various data and parameter tensors. After E is computed, such toolboxes allow to automatically compute the value of all derivatives $dE/dc, dE/d\mathbf{x}_0, dE/d\mathbf{w}_1, \dots, dE/d\mathbf{w}_n$ (which, since E is a scalar quantity, are all *gradients*). Due to AutoDiff, doing so does not require write any additional code. These derivatives can then be used in optimization algorithms such as stochastic gradient descent to learn the parameters of the model.

AutoDiff works by implicitly keeping track of all operations performed and intermediate results obtained while executing the program. Instructions in the program define a forward chain of calculations, in which each application of an operator f_i produces a new tensor $\mathbf{x}_i$. These steps can be tracked in a so-called *compute graph*, which captures all dependencies between input, output and intermediate variables. The graph is directed and acyclic, as operators are

only applied to inputs that have already been calculated. By quickly working backward through the compute graph, AutoDiff can then compute the required derivatives.

The advantage of AutoDiff, in addition to computing derivatives automatically, is that it is embedded (more or less) transparently in a programming language such as Python or MATLAB. This means that, differently from early CNN toolboxes such as Caffe that require explicitly defining neural networks as compute graphs, one simply executes the CNN layers as instructions in the program. A corresponding compute graph is built transparently and on the fly. This approach affords tremendous development efficiency and flexibility. A clear advantage to work with such "on the fly" graphs is that calculations can vary from batch to batch, which can be useful to process sequences of variable length or to perform conditional processing (`if...then`) in neural networks. In fact, AutoDiff makes the concept of a deep network as a static graph somewhat obsolete; instead, a more flexible viewpoint is to think of programs that contain some parameters that can be optimized over, using AutoDiff and gradient descent.

4.2 Derivatives of tensor functions

In order to describe AutoDiff in more detail, we need to develop a notation for the derivatives of tensor functions. Recall that a *function* $f : X \to Y$ is a rule that maps elements x of the set X (the domain) to elements $y = f(x)$ of the set Y (the codomain).

In machine learning, we are often interested in functions that map a subset of real numbers $X \subset \mathbb{R}$ to another subset $Y \subset \mathbb{R}$ (sometimes $\mathbb{R}$ is replaced by the field of complex numbers $\mathbb{C}$). The *derivative* of the scalar function $y = f(x)$ is given by:

$$\frac{df}{dx}(x) = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta}. \tag{4.1}$$

Note that the derivative is evaluated at a chosen point x, which is

either implied or specified after the df/dx symbol. Assuming that the limit exists for all $x \in X$, then, if f is a function from X to Y, the derivative df/dx is a function from X to $\mathbb{R}$.

The derivative provides an approximation to the function f around point x:

$$f(x + dx) \approx f(x) + \frac{df}{dx}(x) \cdot dx.$$

Such an approximation is also called a *linearization* of f at x.

When working with neural networks, the derivative formula (4.1) needs to be generalized to functions that have a tensor input $\mathbf{x} \in X \subset \mathbb{R}^U$ and a tensor output $\mathbf{y} \in Y \subset \mathbb{R}^V$. The difficulty is that input and output tensors have multiple dimensions $U = U_1 \times \cdots \times U_D$ and $V = V_1 \times \cdots V_E$, respectively. So, given a tensor function $f : X \to Y$, what is the meaning of the symbol $df/d\mathbf{x}$?

Extending the considerations above, the derivative should contain approximate linear dependencies of all the elements of the output tensor $\mathbf{y}$ with respect to all the elements of the input tensor $\mathbf{x}$. Such dependencies can be collected in a table or tensor of dimension $V \times U = V_1 \times \cdots V_E \times U_1 \times \cdots \times U_D$. This tensor is given by the formula:

$$\left[\frac{df}{d\mathbf{x}}(\mathbf{x})\right]_{vu} = \lim_{\delta \to 0} \frac{f_v(\mathbf{x} + \mathbf{e}_u\delta) - f_v(\mathbf{x})}{\delta},$$

where $\mathbf{e}_u \in \mathbb{R}^U$ is the indicator tensor of element u in the input:

$$\forall : 0 \leq u' < U : \quad [\mathbf{e}_u]_{u'} = \delta_{u=u'}.$$

There are three cases of interest. The first is when the output y is a *scalar*. As suggested in section 3.2, in this case is convenient to interpret this scalar as having no dimensions ($V = \phi$) so that the derivative $V \times U = U$ is a tensor with the same shape as the input $\mathbf{x}$:

$$\frac{df}{d\mathbf{x}}(\mathbf{x}) \in \mathbb{R}^U.$$

In this case, the derivative is also called a *gradient*.

The second notable case is when both $\mathbf{x}$ and $\mathbf{y}$ are vectors, in the sense that they have single dimensions $U = U_1$ and $V = V_1$ respectively. In this case:

$$\frac{df}{d\mathbf{x}}(\mathbf{x}) \in \mathbb{R}^{V_1 \times U_1}$$

is a $V_1 \times U_1$ matrix, known ans the *Jacobian*. The Jacobian, as a matrix, is a convenient quantity to work with. An important application is to express the chain rule for the derivative of function chains as a product of matrices (appendix A.1).

The third notable case is when the function $\mathbf{y} = f(\mathbf{x})$ operates on general tensors. In this case, in addition to considering the derivatie a tensor with "combined" dimensions, we can use the vec operator (section 3.2.1) to reshape first tensors into vectors. The advantage is that allows to use the Jacobian notation:

$$\frac{d \operatorname{vec} f}{d \operatorname{vec} \mathbf{x}} \in \mathbb{R}^{(V_1 \cdots V_E) \times (U_1 \cdots U_D)}.$$

Note that the Jacobian is still a matrix, but the first dimension is the product of the dimensions of the output tensor $\mathbf{y}$ and the second dimension is the product of the dimensions of input tensor $\mathbf{x}$. Hence, the Jacobian can be a very large matrix.

4.3 Using AutoDiff

In order to use AutoDiff correctly, it is important to understand what it does and how it works.

AutoDiff keeps track of calculations performed in a program using a *compute graph*. Importantly, most AutoDiff implementations do so only for calculations involving variables that have a certain special type. In PyTorch, for example, these are variables of class `torch.Tensor`.

Whenever a new variable $\mathbf{x}$ is obtained as a result of a calculation, it is added to the graph together with the operator f used to compute

it. In principle, f can take as input all variables that have already been computed prior to it. In practice, the inputs of f are only a small subset of those, called the parent variables $\pi \subset (\mathbf{x}_1, \ldots, \mathbf{x}_n)$. Note that the order of the variables in π is important as these are used as positional arguments for the operator f.

The compute graph $G = (V, E)$ then has:

- As nodes V a set of variable-operator-parent triplets $v = (\mathbf{x}, f, \pi)$.
- As edges the pairs $E = \{(u, v) : v = (\mathbf{x}, f, \pi) \in V,\ u \in \pi\}$ connecting parent variables to their children.

The inputs to the graph are nodes $v = (\mathbf{x}, f, \pi)$ that are associated the null operator $f = \epsilon$ and have no parents $\pi = \phi$.[1] The outputs of the graph are the nodes that do not have children — still, any variable in the graph be read off as output in practice.

Every time a new calculation is performed, a new node and corresponding edges are added to the compute graph. Since every new calculation produces a *new* variable as output, the resulting graph is acyclic. While this may seem obvious, it is worth remarking. In fact, it is not uncommon to find code like:

Require: tensors $c, \mathbf{x}, \mathbf{w}_1, \ldots, \mathbf{w}_n$
1: $\mathbf{x} = f_1(\mathbf{x}, \mathbf{w}_0)$
2: $\mathbf{x} = f_2(\mathbf{x}, \mathbf{w}_1)$
3: $\cdots$
4: $\mathbf{x} = f_n(\mathbf{x}, \mathbf{w}_n)$
5: $E = \ell(\mathbf{x}, c)$

where apparently the *same* variable $\mathbf{x}$ is assigned multiple times. In fact, what happens is that the variable *name* $\mathbf{x}$ is reassigned, but each equation causes a *new* variable to be created under the hood, and a new node to be added to the compute graph. This is consistent with the semantics of imperative programming languages

[1] Alternatively, they can be modelled as the output of nullary functions f that output a constant value.

where names can be reused when intermediate results are not needed anymore. However, this distinction becomes essential for AutoDiff, which must keep track of all intermediate calculations even if variable names are reassigned.

Once the graph has been (implicitly) assembled as calculations are performed, AutoDiff is ready to compute derivatives. In order to do so, one must specify which gradients are required with respect to which variable. Normally, this is done as follows:

1. While the graph is constructed by performing calculations, a subset of the variables $\mathbf{w}_1, \ldots, \mathbf{w}_k$ is declared as "requiring gradients".

2. An output variable z is selected. This variable is a *scalar* $z \in \mathbb{R}$ representing an energy value, error, or loss to be optimized.

3. AutoDiff is invoked to compute gradients $dz/d\mathbf{w}_1, \ldots, dz/d\mathbf{w}_k$.

Under the hood, AutoDiff will also compute gradients with respect to all intermediate variables between $\mathbf{w}_k$ and z that are required to perform the desired calculation via the chain rule.

Gradient accumulation

In PyTorch, each invocation of AutoDiff does not simply write gradients, but *accumulates* new gradients to gradient "buffers" that are associated to each variable. This is useful because in this manner it is possible to compute gradients with respect to the sum $z_1 + \cdots + z_k$ over several output variables (which could be multiple losses) by invoking AutoDiff multiple times. This features, however, requires to *clear* the gradient buffers (`zero_grads` in PyTorch) before starting a new cycle.

4.4 Backpropagation

In order to carry out the actual derivative calculations, AutoDiff uses the backpropagation (BackProp) algorithm. BackProp is essentially an efficient implementation of the chain rule to differentiate composition of functions operating on large data variables such as tensors.

Understanding backpropagation is important in order to understand the cost and limitations of AutoDiff, as well as to extend neural network toolboxes with efficient implementations of new layers.

In order to understand backpropagation, consider first a simple chain or composition of vector functions:

$$x_n = (f_n \circ f_{n-1} \circ f_{n-2} \circ \cdots \circ f_1)(\mathbf{x}_0) = h(\mathbf{x}_0).$$

We thus assume that all variables are vectors $\mathbf{x}_i \in \mathbb{R}^{U_i}$ (i.e. tensors with one dimension), and we further assume that the last variable is a scalar. If the functions work on multi-dimensional tensors, as it is commonly the case, we can fall back to this case by reshaping all tensors into vectors using the vec operator (section 3.2).

Suppose that our goal is to calculate the derivative of the output x_n with respect to the input of the chain, namely $dh/d\mathbf{x}_0$. Applying the chain rule (appendix A.1), we find immediately:

$$\underbrace{\frac{d(f_n \circ f_{n-1} \circ \cdots \circ f_1)}{d\mathbf{x}_0}}_{1 \times U_0} = \underbrace{\frac{df_n}{d\mathbf{x}_{n-1}}}_{1 \times U_{n-1}} \cdot \underbrace{\frac{df_{n-1}}{d\mathbf{x}_{n-2}}}_{U_{n-1} \times U_{n-2}} \cdot \underbrace{\frac{df_{n-2}}{d\mathbf{x}_{n-3}}}_{U_{n-2} \times U_{n-3}} \cdots \underbrace{\frac{df_1}{d\mathbf{x}_0}}_{U_1 \times U_0}. \tag{4.2}$$

Here each derivative is a Jacobian matrix, and all we need to do is to multiply them together to obtain the solution.

The problem of this approach is the sheer dimensionality of these matrices in practical neural networks. When data are tensors, each U_i in eq. (4.2) is in fact the product of the corresponding tensor dimensions. In practical networks, this means that a single $U_i \times U_{i-1}$ Jacobian matrix, may occupy several GBs of memory, making calculations unfeasible.

Notably, however, the final result of (4.2) is a $1 \times U_0$ (or simply U_0) dimensional Jacobian matrix, a special case due to the fact that the output is a *scalar* score x_n. This is nearly always the case, as the goal is to compute the derivative of an energy function (average error) to optimize it. Hence, at least the *result* of the computation $dh/d\mathbf{x}_0$ has the same dimension as $\mathbf{x}_0$ and hence can be stored without problems.

Interestingly, if we group terms in eq. (4.2) from the left to the right, we see that all intermediate products have a similar sturcture:

$$\underbrace{\frac{d(f_n \circ \cdots \circ f_1)}{d\mathbf{x}_0}}_{1\times U_0} = \underbrace{\underbrace{\underbrace{\underbrace{\frac{df_n}{d\mathbf{x}_{n-1}}}_{1\times U_{n-1}} \cdot \underbrace{\frac{df_{n-1}}{d\mathbf{x}_{n-2}}}_{U_{n-1}\times U_{n-2}}}_{1\times U_{n-2}} \cdot \underbrace{\frac{df_{n-2}}{d\mathbf{x}_{n-3}}}_{U_{n-2}\times U_{n-3}}}_{1\times U_{n-3}} \cdots \underbrace{\frac{df_1}{d\mathbf{x}_0}}_{U_1\times U_0}}_{1\times U_0}. \tag{4.3}$$

Hence, if we multiply matrices from left to right, all intermediate results can be stored without problems.

We now examine such intermediate results in more detail, giving them names $\mathbf{p}_{n-1}, \mathbf{p}_{n-2}, \ldots, \mathbf{p}_0$:

$$\underbrace{\frac{d(f_n \circ \cdots \circ f_1)}{d\mathbf{x}_0}}_{\mathbf{p}_0} = \underbrace{\underbrace{\underbrace{\underbrace{\frac{df_n}{d\mathbf{x}_{n-1}}}_{\mathbf{p}_{n-1}} \cdot \frac{df_{n-1}}{d\mathbf{x}_{n-2}}}_{\mathbf{p}_{n-2}} \cdot \frac{df_{n-2}}{d\mathbf{x}_{n-3}}}_{\mathbf{p}_{n-3}} \cdots \frac{df_1}{d\mathbf{x}_0}}_{\mathbf{p}_0}. \tag{4.4}$$

We see that each $\mathbf{p}_i$ is simply the derivative of the output variable x_n with respect to the intermediate variable $\mathbf{x}_i$:

$$\boxed{\mathbf{p}_i = \frac{d(f_n \circ \cdots \circ f_{i+1})}{d\mathbf{x}_i}(\mathbf{x}_i) \in \mathbb{R}^{1\times U_i}.}$$

This suggests the following iteration, which computes $\mathbf{p}_i$ backward, from left to right:

$$\mathbf{p}_{i-1} \leftarrow \mathbf{p}_i \cdot \frac{df_i}{d\mathbf{x}_{i-1}}. \tag{4.5}$$

The iteration starts with $p_n = 1$ (the derivative of x_n w.r.t. x_n) and terminates with $\mathbf{p}_0 = dh/d\mathbf{x}_0$ as required. The advantage of this iteration is that the output of each step is a tensor $\mathbf{p}_i$ that has the same size as the corresponding variable $\mathbf{x}_i$, and is therefore manageable.

The problem, of course, is that in order to compute each iteration eq. (4.5) one still needs to compute a potentially unfeasibly-large Jacobian matrix $df_i/d\mathbf{x}_{i-1}$. However, such matrices have in practice a lot of structure, and in particular are very sparse, as they reflect the computations performed by each layer f_i. The key idea of backpropagation is to exploit in ad-hoc manner such layer-specific structures in order to greatly optimize the computation of (4.5).

The key ingredient is the following lemma:

Lemma 2. *Let $\mathbf{y} = f(\mathbf{x})$ be a vector function where $\mathbf{y} \in \mathbb{R}^V$ and $\mathbf{x} \in \mathbb{R}^U$ and let $\mathbf{p} \in \mathbb{R}^{1\times V}$ a fixed row vector. Then:*

$$\mathbf{p} \cdot \frac{df}{d\mathbf{x}}(\mathbf{x}) = \frac{d(\mathbf{p} \cdot f)}{d\mathbf{x}}(\mathbf{x}).$$

In other words, the vector-matrix product in the left hand side can be computed as the derivative of the scalar-valued projected function *$\mathbf{p} \cdot f$ to the right.*

Proof. This is a direct application of the linearity of the derivative operator:

$$\mathbf{p} \cdot \frac{df}{d\mathbf{x}}(\mathbf{x}) = \sum_v p_v \cdot \frac{df_v}{d\mathbf{x}}(\mathbf{x}) = \frac{d}{d\mathbf{x}}\left[\sum_v p_v f_v(\mathbf{x})\right] = \frac{d(\mathbf{p} \cdot f)}{d\mathbf{x}}.$$

□

We call the quantity

$$\boxed{f^{\mathrm{BP}}(\mathbf{x}, \mathbf{p}) = \frac{d(\mathbf{p} \cdot f)}{d\mathbf{x}}(\mathbf{x})} \tag{4.6}$$

projected function derivative. The reason is that this is the derivative of the function $\mathbf{p} \cdot f(\mathbf{x})$, which is the projection of the vector function f on the constant vector $\mathbf{p}$.

Dimensions and simplified notation

We note that $\mathbf{p}$ as defined above is a row vector $\mathbb{R}^{1 \times V}$. This notation is fine, but cumbersome. Instead, we often just think of $\mathbf{p}$ as having the same dimension $\mathbb{R}^V$ as the corresponding output variable $\mathbf{y}$. Then, and also to emphasize the projection operator, we write (4.6) as

$$\frac{d\langle \mathbf{p}, f \rangle}{d\mathbf{x}} \in \mathbb{R}^U.$$

A nice thing about this version of the equation is that it works out of the box if, instead of vectors, we work with multi-dimensional tensors (so that $V = V_1 \times \cdots \times V_E$ and $U = U_1 \times \cdots \times U_D$).

The importance of the projected function derivative is that machine learning toolboxes, or you as an author of a new layer, can focus on writing an optimized algorithm to compute quantity (4.6), exploiting any layer-specific simplification.

In practice, a machine learning toolbox provides for each "atomic layer" f (i.e. layers that are not obtained as the composition of other more elementary layers, to which AutoDiff can be applied) two implementations:

- A **forward mode** implementation that takes as input $\mathbf{x}$ and computes $\mathbf{y} = f(\mathbf{x})$.

- A **backward mode** implementation that takes as input $\mathbf{x}$ and $\mathbf{p}$ and computes $\mathbf{p}' = f^{BP}(\mathbf{x}, \mathbf{p})$, i.e. the derivative not of f, but of $\langle \mathbf{p}, f \rangle$, with respect to the argument $\mathbf{x}$.

While toolboxes make these functions available, and you many need to author them to add new layers, the backward mode function is usually invoked automatically by AutoDiff when needed.

Appendix A

Elements of linear algebra and multivariate analysis

We use boldface symbols such as $\mathbf{x}$ to denote multi-dimensional objects such as vectors and tensors. In most cases, we consider real vector spaces $\mathbb{R}^d$.

A.1 Composition and the chain rule

Deep learning is based on computing derivatives of compositions of functions.

Two functions $f : X \to Y$ and $g : Y \to Z$ can be *composed* to result in a third function $g \circ f$ that maps elements $x \in X$ to elements $z = g(f(x))$ of Z. The composition symbol $\circ$ looks like a product but is not (although it is essentially a product if the functions are all linear). Function composition is associative $(h \circ (g \circ f) = (h \circ g) \circ f)$, so parentheses are not required and one may just write $h \circ g \circ f$ for

the composition of several functions.

Computing the derivative of composition of functions is based on the*chain rule*: the derivative of a composition is the product of the individual derivatives. Consider first the scalar case $y = f(x)$ and $z = g(y)$. If we linearise the functions we geometry

$$f(x + dx) \approx f(x) + dy, \quad dy = \frac{df}{dx}(x) \cdot dx,$$
$$g(y + dy) \approx g(y) + dz, \quad dz = \frac{dg}{dy}(y) \cdot dy,$$

so that

$$dz = \frac{dg}{dy}(y) \cdot \frac{df}{dx}(x) \cdot dx.$$

This substitution suggests (but does not prove!) that the derivative of the composition $g \circ f$ is given by:

$$\frac{d(g \circ f)}{dx}(x) = \frac{dg}{dy}(y) \cdot \frac{df}{dx}(x), \qquad \text{where } y = f(x).$$

If $\mathbf{x} \in \mathbb{R}^{U_1}$, $\mathbf{y} \in \mathbb{R}^{V_1}$ and $\mathbf{z} \in \mathbb{R}^{W_1}$ are vectors, then the chain rule has exactly the same form (and intuitive justification), except that derivatives are matrices:

$$\underbrace{\frac{d(g \circ f)}{d\mathbf{x}}(\mathbf{x})}_{W_1 \times U_1} = \underbrace{\frac{dg}{d\mathbf{y}}(\mathbf{y})}_{W_1 \times V_1} \cdot \underbrace{\frac{df}{d\mathbf{x}}(\mathbf{x})}_{V_1 \times U_1}, \qquad \text{where } \mathbf{y} = f(\mathbf{x}).$$

Finally, if $\mathbf{x} \in \mathbb{R}^{U}$, $\mathbf{y} \in \mathbb{R}^{V}$ and $\mathbf{z} \in \mathbb{R}^{W}$ are tensors $U = U_1 \times \cdots \times U_D$, $V = V_1 \times \cdots \times V_E$ and $W = W_1 \times \cdots \times W_F$, then we can use the vec operator to obtain:

$$\underbrace{\frac{d \operatorname{vec}(g \circ f)}{d \operatorname{vec} \mathbf{x}}(\mathbf{x})}_{\prod_k W_k \times \prod_i U_i} = \underbrace{\frac{d \operatorname{vec} g}{d \operatorname{vec} \mathbf{y}}(\mathbf{y})}_{\prod_k W_k \times \prod_j V_i} \cdot \underbrace{\frac{d \operatorname{vec} f}{d \operatorname{vec} \mathbf{x}}(\mathbf{x})}_{\prod_j V_j \times \prod_i U_i}, \qquad \text{where } \mathbf{y} = f(\mathbf{x}).$$