CHAPTER 5

Neural Network Training

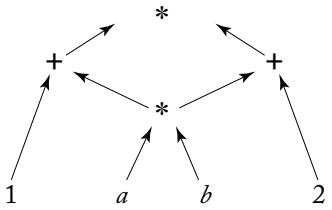
Similar to linear models, neural network are differentiable parameterized functions, and are trained using gradient-based optimization (see Section 2.8). The objective function for nonlinear neural networks is not convex, and gradient-based methods may get stuck in a local minima. Still, gradient-based methods produce good results in practice.

Gradient calculation is central to the approach. The mathematics of gradient computation for neural networks are the same as those of linear models, simply following the chain-rule of differentiation. However, for complex networks this process can be laborious and error-prone. Fortunately, gradients can be efficiently and automatically computed using the *backpropagation algorithm* [LeCun et al., 1998b, Rumelhart et al., 1986]. The backpropagation algorithm is a fancy name for methodically computing the derivatives of a complex expression using the chain-rule, while caching intermediary results. More generally, the backpropagation algorithm is a special case of the reverse-mode automatic differentiation algorithm [Neidinger, 2010, Section 7], [Baydin et al., 2015, Bengio, 2012]. The following section describes reverse mode automatic differentiation in the context of the *computation graph* abstraction. The rest of the chapter is devoted to practical tips for training neural networks in practice.

5.1 THE COMPUTATION GRAPH ABSTRACTION

While one can compute the gradients of the various parameters of a network by hand and implement them in code, this procedure is cumbersome and error prone. For most purposes, it is preferable to use automatic tools for gradient computation [Bengio, 2012]. The computation-graph abstraction allows us to easily construct arbitrary networks, evaluate their predictions for given inputs (forward pass), and compute gradients for their parameters with respect to arbitrary scalar losses (backward pass).

A computation graph is a representation of an arbitrary mathematical computation as a graph. It is a directed acyclic graph (DAG) in which nodes correspond to mathematical operations or (bound) variables and edges correspond to the flow of intermediary values between the nodes. The graph structure defines the order of the computation in terms of the dependencies between the different components. The graph is a DAG and not a tree, as the result of one operation can be the input of several continuations. Consider for example a graph for the computation of $(a * b + 1) * (a * b + 2)$:



The computation of $a * b$ is shared. We restrict ourselves to the case where the computation graph is connected (in a disconnected graph, each connected component is an independent function that can be evaluated and differentiated independently of the other connected components).

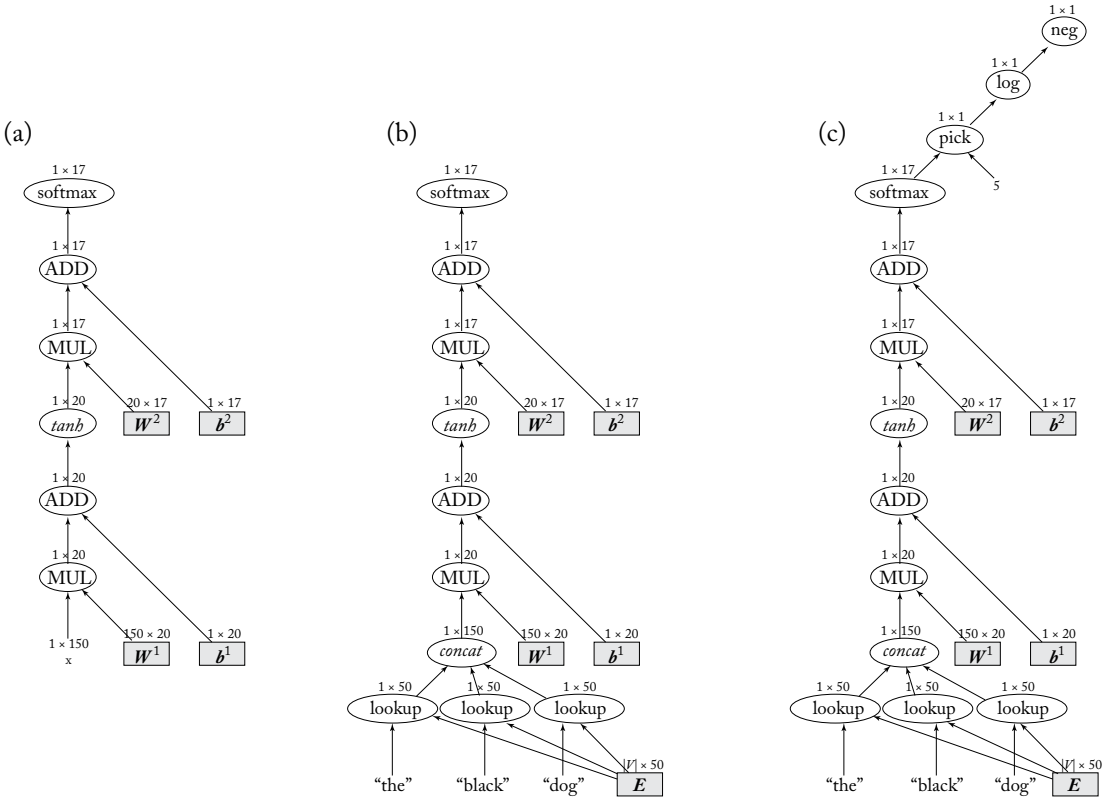


Figure 5.1: (a) Graph with unbound input. (b) Graph with concrete input. (c) Graph with concrete input, expected output, and a final loss node.

Since a neural network is essentially a mathematical expression, it can be represented as a computation graph. For example, Figure 5.1a presents the computation graph for an MLP with one hidden-layer and a softmax output transformation. In our notation, oval nodes represent

mathematical operations or functions, and shaded rectangle nodes represent parameters (bound variables). Network inputs are treated as constants, and drawn without a surrounding node. Input and parameter nodes have no incoming arcs, and output nodes have no outgoing arcs. The output of each node is a matrix, the dimensionality of which is indicated above the node.

This graph is incomplete: without specifying the inputs, we cannot compute an output. Figure 5.1b shows a complete graph for an MLP that takes three words as inputs, and predicts the distribution over part-of-speech tags for the third word. This graph can be used for prediction, but not for training, as the output is a vector (not a scalar) and the graph does not take into account the correct answer or the loss term. Finally, the graph in Figure 5.1c shows the computation graph for a specific training example, in which the inputs are the (embeddings of) the words "the," "black," "dog," and the expected output is "NOUN" (whose index is 5). The *pick* node implements an indexing operation, receiving a vector and an index (in this case, 5) and returning the corresponding entry in the vector.

Once the graph is built, it is straightforward to run either a forward computation (compute the result of the computation) or a backward computation (computing the gradients), as we show below. Constructing the graphs may look daunting, but is actually very easy using dedicated software libraries and APIs.

5.1.1 FORWARD COMPUTATION

The forward pass computes the outputs of the nodes in the graph. Since each node's output depends only on itself and on its incoming edges, it is trivial to compute the outputs of all nodes by traversing the nodes in a topological order and computing the output of each node given the already computed outputs of its predecessors.

More formally, in a graph of N nodes, we associate each node with an index i according to their topological ordering. Let f_i be the function computed by node i (e.g., *multiplication*. *addition*, etc.). Let $\pi(i)$ be the parent nodes of node i, and $\pi^{-1}(i) = \{j \mid i \in \pi(j)\}$ the children nodes of node i (these are the arguments of f_i). Denote by $v(i)$ the output of node i, that is, the application of f_i to the output values of its arguments $\pi^{-1}(i)$. For variable and input nodes, f_i is a constant function and $\pi^{-1}(i)$ is empty. The computation-graph forward pass computes the values $v(i)$ for all $i \in [1, N]$.

Algorithm 5.3 Computation graph forward pass.

1: **for** i = 1 to N **do**
2: Let $a_1, \ldots, a_m = \pi^{-1}(i)$
3: $v(i) \leftarrow f_i(v(a_1), \ldots, v(a_m))$

5.1.2 BACKWARD COMPUTATION (DERIVATIVES, BACKPROP)

The backward pass begins by designating a node N with scalar (1×1) output as a loss-node, and running forward computation up to that node. The backward computation computes the gradients of the parameters with respect to that node's value. Denote by $d(i)$ the quantity $\frac{\partial N}{\partial i}$. The backpropagation algorithm is used to compute the values $d(i)$ for all nodes i.
The backward pass fills a table of values $d(1), \ldots, d(N)$ as in Algorithm 5.4.

Algorithm 5.4 Computation graph backward pass (backpropagation).

1: $d(N) \leftarrow 1$ ▷ $\frac{\partial N}{\partial N} = 1$
2: **for** i = N-1 to 1 **do**
3: $\quad d(i) \leftarrow \sum_{j \in \pi(i)} d(j) \cdot \frac{\partial f_j}{\partial i}$ ▷ $\frac{\partial N}{\partial i} = \sum_{j \in \pi(i)} \frac{\partial N}{\partial j} \frac{\partial j}{\partial i}$

The backpropagation algorithm (Algorithm 5.4) is essentially following the chain-rule of differentiation. The quantity $\frac{\partial f_j}{\partial i}$ is the partial derivative of $f_j(\pi^{-1}(j))$ w.r.t the argument $i \in \pi^{-1}(j)$. This value depends on the function f_j and the values $v(a_1), \ldots, v(a_m)$ (where $a_1, \ldots, a_m = \pi^{-1}(j)$) of its arguments, which were computed in the forward pass.

Thus, in order to define a new kind of node, one needs to define two methods: one for calculating the forward value $v(i)$ based on the node's inputs, and the another for calculating $\frac{\partial f_i}{\partial x}$ for each $x \in \pi^{-1}(i)$.

Derivatives of "non-mathematical" functions While defining $\frac{\partial f_i}{\partial x}$ for mathematical functions such is as log or $+$ is straightforward, some find it challenging to think about the derivative of operations as as $pick(\boldsymbol{x}, 5)$ that selects the fifth element of a vector. The answer is to think in terms of the contribution to the computation. After picking the ith element of a vector, only that element participates in the remainder of the computation. Thus, the gradient of $pick(\boldsymbol{x}, 5)$ is a vector $\boldsymbol{g}$ with the dimensionality of $\boldsymbol{x}$ where $\boldsymbol{g}_{[5]} = 1$ and $\boldsymbol{g}_{[i \neq 5]} = 0$. Similarly, for the function $\max(0, x)$ the value of the gradient is 1 for $x > 0$ and 0 otherwise.

For further information on automatic differentiation, see Neidinger [2010, Section 7] and Baydin et al. [2015]. For more in depth discussion of the backpropagation algorithm and computation graphs (also called flow graphs), see Bengio et al. [2016, Section 6.5] and Bengio [2012], LeCun et al. [1998b]. For a popular yet technical presentation, see Chris Olah's description at http://colah.github.io/posts/2015-08-Backprop/.

5.1.3 SOFTWARE

Several software packages implement the computation-graph model, including Theano,[1] [Bergstra et al., 2010], TensorFlow[2] [Abadi et al., 2015], Chainer,[3] and DyNet[4] [Neubig et al., 2017]. All these packages support all the essential components (node types) for defining a wide range of neural network architectures, covering the structures described in this book and more. Graph creation is made almost transparent by use of operator overloading. The framework defines a type for representing graph nodes (commonly called *expressions*), methods for constructing nodes for inputs and parameters, and a set of functions and mathematical operations that take expressions as input and result in more complex expressions. For example, the python code for creating the computation graph from Figure 5.1c using the DyNet framework is:

```
import dynet as dy
# model initialization.
model = dy.Model()
mW1 = model.add_parameters((20,150))
mb1 = model.add_parameters(20)
mW2 = model.add_parameters((17,20))
mb2 = model.add_parameters(17)
lookup = model.add_lookup_parameters((100, 50))
trainer = dy.SimpleSGDTrainer(model)

def get_index(x):
    pass # Logic omitted.
Maps words to numeric IDs.

# The following builds and executes the computation graph,
# and updates model parameters.
# Only one data point is shown, in practice the following
# should run in a data-feeding loop.

# Building the computation graph:
dy.renew_cg() # create a new graph.
# Wrap the model parameters as graph-nodes.
W1 = dy.parameter(mW1)
b1 = dy.parameter(mb1)
W2 = dy.parameter(mW2)
b2 = dy.parameter(mb2)
# Generate the embeddings layer.
vthe   = dy.lookup[get_index("the")]
vblack = dy.lookup[get_index("black")]
vdog   = dy.lookup[get_index("dog")]

# Connect the leaf nodes into a complete graph.
x = dy.concatenate([vthe, vblack, vdog])
output = dy.softmax(W2*(dy.tanh(W1*x+b1))+b2)
loss = -dy.log(dy.pick(output, 5))
```

[1] http://deeplearning.net/software/theano/
[2] https://www.tensorflow.org/
[3] http://chainer.org
[4] https://github.com/clab/dynet

```
loss_value = loss.forward()
loss.backward() # the gradient is computed
                # and stored in the corresponding
                # parameters.
trainer.update() # update the parameters according to the gradients.
```

Most of the code involves various initializations: the first block defines model parameters that are be shared between different computation graphs (recall that each graph corresponds to a specific training example). The second block turns the model parameters into the graph-node (Expression) types. The third block retrieves the Expressions for the embeddings of the input words. Finally, the fourth block is where the graph is created. Note how transparent the graph creation is—there is an almost a one-to-one correspondence between creating the graph and describing it mathematically. The last block shows a forward and backward pass. The equivalent code in the TensorFlow package is:[5]

```
import tensorflow as tf

W1 = tf.get_variable("W1", [20, 150])
b1 = tf.get_variable("b1", [20])
W2 = tf.get_variable("W2", [17, 20])
b2 = tf.get_variable("b2", [17])
lookup = tf.get_variable("W", [100, 50])

def get_index(x):
    pass # Logic omitted

p1 = tf.placeholder(tf.int32, [])
p2 = tf.placeholder(tf.int32, [])
p3 = tf.placeholder(tf.int32, [])
target = tf.placeholder(tf.int32, [])

v_w1 = tf.nn.embedding_lookup(lookup, p1)
v_w2 = tf.nn.embedding_lookup(lookup, p2)
v_w3 = tf.nn.embedding_lookup(lookup, p3)

x = tf.concat([v_w1, v_w2, v_w3], 0)
output = tf.nn.softmax(
    tf.einsum("ij,j->i", W2, tf.tanh(
        tf.einsum("ij,j->i", W1, x) + b1)) + b2)
loss = -tf.log(output[target])
trainer = tf.train.GradientDescentOptimizer(0.1).minimize(loss)

# Graph definition done, compile it and feed concrete data.
# Only one data-point is shown, in practice we will use
# a data-feeding loop.
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    feed_dict = {
        p1: get_index("the"),
        p2: get_index("black"),
        p3: get_index("dog"),
```

[5] TensorFlow code provided by Tim Rocktäschel. Thanks Tim!

```
    target: 5
}
loss_value = sess.run(loss, feed_dict)
# update, no call of backward necessary
sess.run(trainer, feed_dict)
```

The main difference between DyNet (and Chainer) to TensorFlow (and Theano) is that the formers use *dynamic graph construction* while the latters use *static graph construction*. In dynamic graph construction, a different computation graph is created from scratch for each training sample, using code in the host language. Forward and backward propagation are then applied to this graph. In contrast, in the static graph construction approach, the shape of the computation graph is defined once in the beginning of the computation, using an API for specifying graph shapes, with place-holder variables indicating input and output values. Then, an optimizing graph compiler produces an optimized computation graph, and each training example is fed into the (same) optimized graph. The graph compilation step in the static toolkits (TensorFlow and Theano) is both a blessing and a curse. On the one hand, once compiled, large graphs can be run efficiently on either the CPU or a GPU, making it ideal for large graphs with a fixed structure, where only the inputs change between instances. However, the compilation step itself can be costly, and it makes the interface more cumbersome to work with. In contrast, the dynamic packages focus on building large and dynamic computation graphs and executing them "on the fly" without a compilation step. While the execution speed may suffer compared to the static toolkits, in practice the computation speeds of the dynamic toolkits are very competitive. The dynamic packages are especially convenient when working with the recurrent and recursive networks described in Chapters 14 and 18 as well as in structured prediction settings as described in Chapter 19, in which the graphs of different data-points have different shapes. See Neubig et al. [2017] for further discussion on the dynamic-vs.-static approaches, and speed benchmarks for the different toolkits. Finally, packages such as Keras[6] provide a higher level interface on top of packages such as Theano and TensorFlow, allowing the definition and training of complex neural networks with even fewer lines of code, provided that the architectures are well established, and hence supported in the higher-level interface.

5.1.4 IMPLEMENTATION RECIPE

Using the computation graph abstraction and dynamic graph construction, the pseudo-code for a network training algorithm is given in Algorithm 5.5.

Here, build_computation_graph is a user-defined function that builds the computation graph for the given input, output, and network structure, returning a single loss node. update_parameters is an optimizer specific update rule. The recipe specifies that a new graph is created for each training example. This accommodates cases in which the network structure varies between training examples, such as recurrent and recursive neural networks, to be discussed in

[6] https://keras.io

Algorithm 5.5 Neural network training with computation graph abstraction (using minibatches of size 1).

```
Define network parameters.
for iteration = 1 to T do
    for Training example x_i, y_i in dataset do
        loss_node ← build_computation_graph(x_i, y_i, parameters)
        loss_node.forward()
        gradients ← loss_node().backward()
        parameters ← update_parameters(parameters, gradients)
return parameters.
```

Chapters 14–18. For networks with fixed structures, such as an MLPs, it may be more efficient to create one base computation graph and vary only the inputs and expected outputs between examples.

5.1.5 NETWORK COMPOSITION

As long as the network's output is a vector ($1 \times k$ matrix), it is trivial to compose networks by making the output of one network the input of another, creating arbitrary networks. The computation graph abstractions makes this ability explicit: a node in the computation graph can itself be a computation graph with a designated output node. One can then design arbitrarily deep and complex networks, and be able to easily evaluate and train them thanks to automatic forward and gradient computation. This makes it easy to define and train elaborate recurrent and recursive networks, as discussed in Chapters 14–16 and 18, as well as networks for structured outputs and multi-objective training, as we discuss in Chapters 19 and 20.

5.2 PRACTICALITIES

Once the gradient computation is taken care of, the network is trained using SGD or another gradient-based optimization algorithm. The function being optimized is not convex, and for a long time training of neural networks was considered a "black art" which can only be done by selected few. Indeed, many parameters affect the optimization process, and care has to be taken to tune these parameters. While this book is not intended as a comprehensive guide to successfully training neural networks, we do list here a few of the prominent issues. For further discussion on optimization techniques and algorithms for neural networks, refer to Bengio et al. [2016, Chapter 8]. For some theoretical discussion and analysis, refer to Glorot and Bengio [2010]. For various practical tips and recommendations, see Bottou [2012], LeCun et al. [1998a].

5.2.1 CHOICE OF OPTIMIZATION ALGORITHM

While the SGD algorithm works well, it may be slow to converge. Section 2.8.3 lists some alternative, more advanced stochastic-gradient algorithms. As most neural network software frameworks provide implementations of these algorithms, it is easy and often worthwhile to try out different variants. In my research group, we found that when training larger networks, using the Adam algorithm [Kingma and Ba, 2014] is very effective and relatively robust to the choice of the learning rate.

5.2.2 INITIALIZATION

The non-convexity of the objective function means the optimization procedure may get stuck in a local minimum or a saddle point, and that starting from different initial points (e.g., different random values for the parameters) may result in different results. Thus, it is advised to run several restarts of the training starting at different random initializations, and choosing the best one based on a development set.[7] The amount of variance in the results due to different random seed selections is different for different network formulations and datasets, and cannot be predicted in advance.

The magnitude of the random values has a very important effect on the success of training. An effective scheme due to Glorot and Bengio [2010], called *xavier initialization* after Glorot's first name, suggests initializing a weight matrix $\boldsymbol{W} \in \mathbb{R}^{d_{in} \times d_{out}}$ as:

$$\boldsymbol{W} \sim U\left[-\frac{\sqrt{6}}{\sqrt{d_{in}+d_{out}}}, +\frac{\sqrt{6}}{\sqrt{d_{in}+d_{out}}}\right], \tag{5.1}$$

where $U[a,b]$ is a uniformly sampled random value in the range $[a,b]$. The suggestion is based on properties of the tanh activation function, works well in many situations, and is the preferred default initialization method by many.

Analysis by He et al. [2015] suggests that when using ReLU nonlinearities, the weights should be initialized by sampling from a zero-mean Gaussian distribution whose standard deviation is $\sqrt{\frac{2}{d_{in}}}$. This initialization was found by He et al. [2015] to work better than xavier initialization in an image classification task, especially when deep networks were involved.

5.2.3 RESTARTS AND ENSEMBLES

When training complex networks, different random initializations are likely to end up with different final solutions, exhibiting different accuracies. Thus, if your computational resources allow, it is advisable to run the training process several times, each with a different random initialization, and choose the best one on the development set. This technique is called *random restarts*. The average model accuracy across random seeds is also interesting, as it gives a hint as to the stability of the process.

[7] When debugging, and for reproducibility of results, it is advised to used a fixed random seed.

While the need to "tune" the random seed used to initialize models can be annoying, it also provides a simple way to get different models for performing the same task, facilitating the use *model ensembles*. Once several models are available, one can base the prediction on the ensemble of models rather than on a single one (for example by taking the majority vote across the different models, or by averaging their output vectors and considering the result as the output vector of the ensembled model). Using ensembles often increases the prediction accuracy, at the cost of having to run the prediction step several times (once for each model).

5.2.4 VANISHING AND EXPLODING GRADIENTS

In deep networks, it is common for the error gradients to either vanish (become exceedingly close to 0) or explode (become exceedingly high) as they propagate back through the computation graph. The problem becomes more severe in deeper networks, and especially so in recursive and recurrent networks [Pascanu et al., 2012]. Dealing with the vanishing gradients problem is still an open research question. Solutions include making the networks shallower, step-wise training (first train the first layers based on some auxiliary output signal, then fix them and train the upper layers of the complete network based on the real task signal), performing batch-normalization [Ioffe and Szegedy, 2015] (for every minibatch, normalizing the inputs to each of the network layers to have zero mean and unit variance) or using specialized architectures that are designed to assist in gradient flow (e.g., the LSTM and GRU architectures for recurrent networks, discussed in Chapter 15). Dealing with the exploding gradients has a simple but very effective solution: clipping the gradients if their norm exceeds a given threshold. Let $\hat{\boldsymbol{g}}$ be the gradients of all parameters in the network, and $\|\hat{\boldsymbol{g}}\|$ be their L_2 norm. Pascanu et al. [2012] suggest to set: $\hat{\boldsymbol{g}} \leftarrow \frac{\text{threshold}}{\|\hat{\boldsymbol{g}}\|}\hat{\boldsymbol{g}}$ if $\|\hat{\boldsymbol{g}}\| > \text{threshold}$.

5.2.5 SATURATION AND DEAD NEURONS

Layers with tanh and sigmoid activations can become saturated—resulting in output values for that layer that are all close to one, the upper-limit of the activation function. Saturated neurons have very small gradients, and should be avoided. Layers with the ReLU activation cannot be saturated, but can "die"—most or all values are negative and thus clipped at zero for all inputs, resulting in a gradient of zero for that layer. If your network does not train well, it is advisable to monitor the network for layers with many saturated or dead neurons. Saturated neurons are caused by too large values entering the layer. This may be controlled for by changing the initialization, scaling the range of the input values, or changing the learning rate. Dead neurons are caused by all signals entering the layer being negative (for example this can happen after a large gradient update). Reducing the learning rate will help in this situation. For saturated layers, another option is to normalize the values in the saturated layer after the activation, i.e., instead of $g(\boldsymbol{h}) = \tanh(\boldsymbol{h})$ using $g(\boldsymbol{h}) = \frac{\tanh(\boldsymbol{h})}{\|\tanh(\boldsymbol{h})\|}$. Layer normalization is an effective measure for countering saturation, but is also expensive in terms of gradient computation. A related technique is *batch normalization*, due

to Ioffe and Szegedy [2015], in which the activations at each layer are normalized so that they have mean 0 and variance 1 across each mini-batch. The batch-normalization techniques became a key component for effective training of deep networks in computer vision. As of this writing, it is less popular in natural language applications.

5.2.6 SHUFFLING

The order in which the training examples are presented to the network is important. The SGD formulation above specifies selecting a random example in each turn. In practice, most implementations go over the training example in random order, essentially performing random sampling without replacement. It is advised to shuffle the training examples before each pass through the data.

5.2.7 LEARNING RATE

Selection of the learning rate is important. Too large learning rates will prevent the network from converging on an effective solution. Too small learning rates will take a very long time to converge. As a rule of thumb, one should experiment with a range of initial learning rates in range $[0, 1]$, e.g., 0.001, 0.01, 0.1, 1. Monitor the network's loss over time, and decrease the learning rate once the loss stops improving on a held-out development set. *Learning rate scheduling* decreases the rate as a function of the number of observed minibatches. A common schedule is dividing the initial learning rate by the iteration number. Léon Bottou [2012] recommends using a learning rate of the form $\eta_t = \eta_0(1 + \eta_0\lambda t)^{-1}$ where η_0 is the initial learning rate, η_t is the learning rate to use on the tth training example, and λ is an additional hyperparameter. He further recommends determining a good value of η_0 based on a small sample of the data prior to running on the entire dataset.

5.2.8 MINIBATCHES

Parameter updates occur either every training example (minibatches of size 1) or every k training examples. Some problems benefit from training with larger minibatch sizes. In terms of the computation graph abstraction, one can create a computation graph for each of the k training examples, and then connecting the k loss nodes under an averaging node, whose output will be the loss of the minibatch. Large minibatched training can also be beneficial in terms of computation efficiency on specialized computing architectures such as GPUs, and replacing vector-matrix operations by matrix-matrix operations. This is beyond the scope of this book.