Chapter 2

Neural Networks

A neural network is a machine learning technique that takes a number of inputs and predicts outputs. In many ways, they are not very different from other machine learning methods but have distinct strengths.

2.1 Linear Models

Linear models are a core element of statistical machine translation. A potential translation x of a sentence is represented by a set of features $h_i(x)$. Each feature is weighted by a parameter λ_i to obtain an overall score. Ignoring the exponential function that we used previously to turn the linear model into a log-linear model, the following formula sums up the model.

$$\text{score}(\lambda, x) = \sum_j \lambda_j \, h_j(x) \tag{2.1}$$

Graphically, a linear model can be illustrated by a network, where feature values are input nodes, arrows are weights, and the score is an output node (see Figure 2.1).

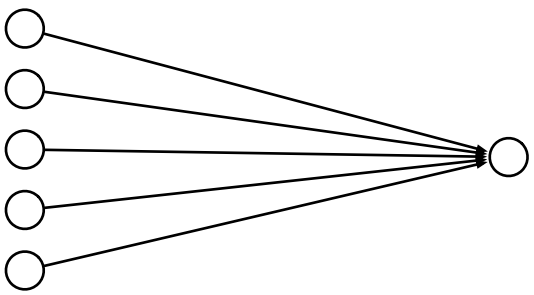


Figure 2.1: Graphical illustration of a linear model as a network: feature values are input nodes, arrows are weights, and the score is an output node.

Most prominently, we use linear models to combine different components of a machine translation system, such as the language model, the phrase translation model, the reordering model, and properties such as the length of the sentence, or the accumulated jump distance between phrase translations. Training methods assign a weight value λ_i to each such feature $h_i(x)$, related to their importance in contributing to scoring better translations higher. In statistical machine translation, this is called **tuning**.

However, linear models do not allow us to define more complex relationships between the features. Let us say that we find that for short sentences the language model is less important than the translation model, or that average phrase translation probabilities higher than 0.1 are similarly reasonable but any value below that is really terrible. The first hypothetical example implies dependence between features and the second example implies non-linear relationship between the feature value and its impact on the final score. Linear models cannot handle these cases.

A commonly cited counter-example to the use of linear models is XOR, i.e., the boolean operator $\oplus$ with the truth table $0 \oplus 0 = 0$, $1 \oplus 0 = 1$, $0 \oplus 1 = 1$, and $1 \oplus 1 = 0$. For a linear model with two features (representing the inputs), it is not possible to come up with weights that give the correct output in all cases. Linear models assume that all instances, represented as points in the feature space, are linearly separable. This is not the case with XOR, and may not be the case for type of features we use in machine translation.

2.2 Multiple Layers

Neural networks modify linear models in two important ways. The first is the use of multiple layers. Instead of computing the output value directly from the input values, a **hidden layer** is introduced. It is called hidden, because we can observe inputs and outputs in training instances, but not the mechanism that connects them — this use of the concept *hidden* is similar to its meaning in hidden Markov models.

See Figure 2.2 for on illustration. The network is processed in two steps. First, a linear combination of weighted input node is computed to produce each hidden node value. Then a linear combination of weighted hidden nodes is computed to produce each output node value.

At this point, let us introduce mathematical notations from the neural network literature. A neural network with a hidden layer consists of

- a vector of input nodes with values $\vec{x} = (x_1, x_2, x_3, ...x_n)^T$
- a vector of hidden nodes with values $\vec{h} = (h_1, h_2, h_3, ...h_m)^T$
- a vector of output nodes with values $\vec{y} = (y_1, y_2, y_3, ...y_l)^T$
- a matrix of weights connecting input nodes with hidden nodes $W = \{w_{ij}\}$
- a matrix of weights connecting hidden nodes with output nodes $U = \{u_{ij}\}$

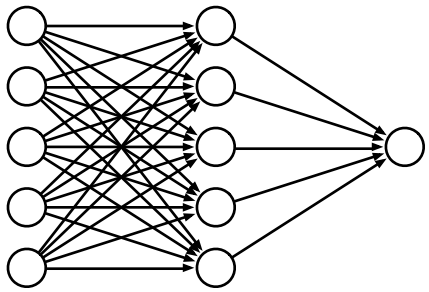


Figure 2.2: A neural network with a hidden layer.

The computations in a neural network with a hidden layer, as sketched out so far, are

$$h_j = \sum_i x_i w_{ji} \tag{2.2}$$

$$y_k = \sum_j h_j u_{kj} \tag{2.3}$$

Note that we snuck in the possibility of multiple output nodes y_k, although our figures so far only showed one.

2.3 Non-Linearity

If we carefully think about the addition of a hidden layer, we realize that we have not gained anything so far to model input/output relationships. We can easily do away with the hidden layer by multiplying out the weights

$$\begin{aligned} y_k &= \sum_j h_j u_{kj} \\ &= \sum_j \sum_i x_i w_{ji} u_{kj} \\ &= \sum_i x_i \left(\sum_j u_{kj} w_{ji} \right) \end{aligned} \tag{2.4}$$

Hence, a salient element of neural networks is the use of a **non-linear activation function**. After computing the linear combination of weighted feature values $s_j = \sum_i x_i w_{ji}$, we obtain the value of a node only after applying such a function $h_j = f(s_j)$.

Popular choices are the **hyperbolic tangent** tanh(x) and the **logistic function** sigmoid(x). See Figure 2.3 for more details on these functions. A good way to think about these activation functions is that they segment the range of values for the linear combination s_j into

- a segment where the node is turned off (values close to 0 for tanh, or –1 for sigmoid)

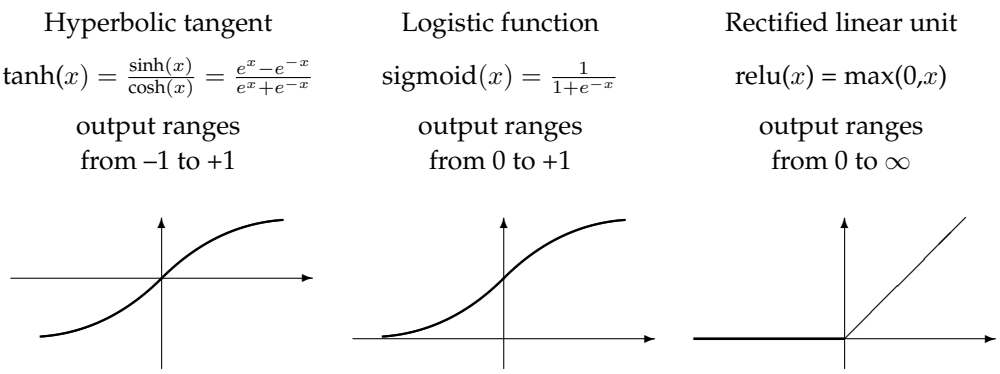


Figure 2.3: Typical activation functions in neural networks.

- a transition segment where the node is partly turned on
- a segment where the node is turned on (values close to 1)

A different popular choice is the activation function for the **rectified linear unit** (ReLU). It does not allow for negative values and floors them at 0, but does not alter the value of positive values. It is simpler and faster to compute than $\tanh(x)$ or $\text{sigmoid}(x)$.

You could view each hidden node as a feature detector. For a certain configurations of input node values, it is turned on, for others it is turned off. Advocates of neural networks claim that the use of hidden nodes obviates (or at least drastically reduces) the need for feature engineering: Instead of manually detecting useful patterns in input values, training of the hidden nodes discovers them automatically.

We do not have to stop at a single hidden layer. The currently fashionable name **deep learning** for neural networks stems from the fact that often better performance can be achieved by deeply stacking together layers and layers of hidden nodes.

2.4 Inference

Let us walk through neural network inference (i.e., how output values are computed from input values) with a concrete example. Consider the neural network in Figure 2.4. This network has one additional innovation that we have not presented so far: bias units. These are nodes that always have the value 1. Such bias units give the network something to work with in the case that all input values are 0. Otherwise, the weighted sum s_j would be 0 no matter the weights.

Let us use this neural network to process some input, say the value 1 for the first input node x_0 and 0 for the second input node x_1. The value of the bias input node (labelled x_2) is fixed to 1. To compute the value of the first hidden node h_0, we have to carry out the following calculation.

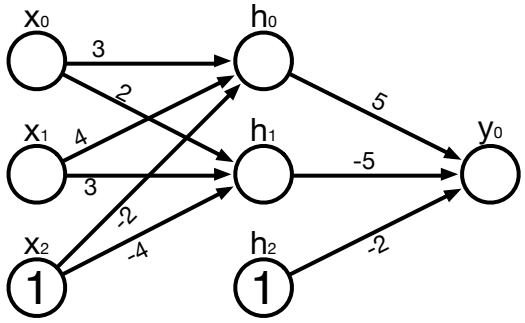


Figure 2.4: A simple neural network with bias nodes in input and hidden layers.

Layer	Node	Summation	Activation
hidden	h_0	$1 \times 3 + 0 \times 4 + 1 \times -2 = 1$	0.731
hidden	h_1	$1 \times 2 + 0 \times 3 + 1 \times -4 = -2$	0.119
output	y_0	$0.731 \times 5 + 0.119 \times -5 + 1 \times -2 = 1.060$	0.743

Table 2.1: Calculations for input (1,0) to the network in Figure 2.4.

$$\begin{aligned} h_0 &= \text{sigmoid}\left(\sum_i x_i w_{1i}\right) \\ &= \text{sigmoid}\left(1 \times 3 + 0 \times 4 + 1 \times -2\right) \\ &= \text{sigmoid}\left(1\right) \\ &= 0.73 \end{aligned} \tag{2.5}$$

The calculations for the other nodes are summarized in Table 2.1. The output value in node y_0 for the input (0,1) is 0.743. If we expect binary output, we would understand this result as the value 1, since it is over the threshold of 0.5 in the range of possible output values [0;1].

Here, the output for all possible binary inputs:

Input x_0	Input x_1	Hidden h_0	Hidden h_1	Output y_0
0	0	0.119	0.018	$0.183 \rightarrow 0$
0	1	0.881	0.269	$0.743 \rightarrow 1$
1	0	0.731	0.119	$0.743 \rightarrow 1$
1	1	0.993	0.731	$0.334 \rightarrow 0$

Our neural network computes XOR. How does it do that? If we look at the hidden nodes h_0 and h_1, we notice that h_0 acts like the Boolean OR: Its value is high if at least of the two input values is 1 (h_0 = 0.881, 0.731, 0.993, for the three configurations), it otherwise has a low value (0.119). The other hidden node h_1 acts like the Boolean AND — it only has a high value (0.731) if both inputs are 1. XOR is effectively implemented as the subtraction of the AND from the OR hidden node.

Note that the non-linearity is key here. Since the value for the OR node h_0 is not that much higher for the input of (1,1) opposed to a single 1 in the input (0.993 vs. 0.881 and 0.731), the distinct high value for the AND node h_1 in this case (0.731) manages to push the final output y_0 below the threshold. This would not be possible if the values of the inputs would be simply summed up as in linear models.

As mentioned before, recently the use of the name **deep learning** for neural networks has become fashionable. It emphasizes that often higher performance can be achieved by using networks with multiple hidden layers. Our XOR example hints at where this power comes from. With a single input-output layer network it is possible to mimic basic Boolean operations such as AND and OR since they can be modeled with linear classifiers. XOR can be expressed as x AND $y - x$ OR y, and our neural network example implements the Boolean operations AND and OR in the first layer, and the subtraction in the second layer. For functions that require more intricate computations, more operations may be chained together, and hence a neural network architecture with more hidden layers may be needed. It may be possible (with sufficient training data) to build neural networks for any computer program, if the number of hidden layers matches the depth of the computation. There is a line of research under the banner **neural Turing machines** that explores what kind of architectures are needed to implement basic algorithms (Gemici et al., 2017). For instance, a neural network with two hidden layers is sufficient to implement an algorithm that sorts n-bit numbers.

2.5 Back-Propagation Training

Training neural networks requires the optimization of weight values so that the network predicts the correct output for a set of training examples. We repeatedly feed the input from the training examples into the network, compare the computed output of the network with the correct output from the training example, and update the weights. Typically, several passes over the training data are carried out. Each pass over the data is called an **epoch**.

The most common training method for neural networks is called **back-propagation**, since it first updates the weights to the output layer, and propagates back error information to earlier layers. Whenever a training example is processed, then for each node in the network, an error term is computed which is the basis for updating the values for incoming weights.

The formulas used to compute updated values for weights follows principles of **gradient descent** training. The error for a specific node is understood as a function of the incoming weights. To reduce the error given this function, we compute the gradient of the error function with respect to each of the weights, and move against the gradient to reduce the error.

Why is moving alongside the gradient a good idea? Consider that we optimize multiple dimensions at the same time. If you are looking for the lowest point in an area (maybe you are looking for water in a desert), and the ground falls off steep to the west of you, and also slightly south of you, then you would go in a direction that is mainly west — and only slightly south. In other words, you go alongside the gradient. See Figure 2.5 for an illustration.

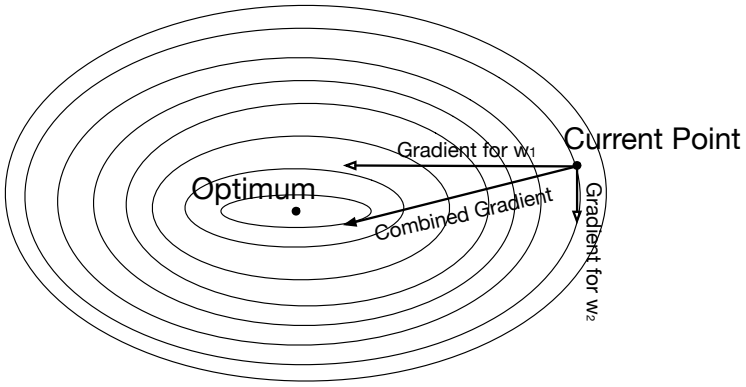


Figure 2.5: Gradient descent training: We compute the gradient with regard to every dimension. In this case the gradient with respect to weight w_2 smaller than the gradient with respect to the weight w_1, so we move more to the left than down (note: arrows point in negative gradient direction, pointing to the minimum).

In the following two sections, we will derive the formulae for updating weights for our example network. If you are less interested in the *why* and more in the *how*, you can skip these sections and continue reading when we summarize the update formulae on page 20.

2.5.1 Weights to the output nodes

Let us first review and extend our notation. At an output node y_i, we first compute a linear combination of weight and hidden node values.

$$s_i = \sum_j w_{i \leftarrow j} h_j \tag{2.6}$$

This sum s_i is passed through an activation function such as sigmoid to compute the output value y.

$$y_i = \text{sigmoid}(s_i) \tag{2.7}$$

We compare the computed output values y_i against the target output values t_i from the training example. There are various ways to compute an error value E from these values. Let us use the L2 norm.

$$E = \sum_i \frac{1}{2}(t_i - y_i)^2 \tag{2.8}$$

As we stated above, our goal is to compute the gradient of the error E with respect to the weights w_k to find out in which direction (and how strongly) we should move the weight value. We do this for each weight w_k separately. We first break up the computation of the gradient into three steps, essentially unfolding the Equations 2.6 to 2.8.

$$\frac{dE}{dw_{i \leftarrow j}} = \frac{dE}{dy_i} \frac{dy_i}{ds_i} \frac{ds_i}{dw_{i \leftarrow j}} \tag{2.9}$$

Let us now work through each of these three steps.

- Since we defined the error E in terms of the output values y_i, we can compute the first component as follows.

$$\frac{dE}{dy_i} = \frac{d}{dy_i}\ \frac{1}{2}(t_i - y_i)^2 = -(t_i - y_i) \tag{2.10}$$

- The derivative of the output value y_i with respect to s_i (the linear combination of weight and hidden node values) depends on the activation function. In the case of sigmoid, we have:

$$\frac{dy_i}{ds_i} = \frac{d\ \text{sigmoid}(s_i)}{ds_i} = \text{sigmoid}(s_i)(1 - \text{sigmoid}(s_i)) = y_i(1 - y_i) \tag{2.11}$$

  To keep our treatment below as general as possible and not commit to the sigmoid as an activation function, we will use the shorthand y_i' for $\frac{dy_i}{ds_i}$ below. Note that for any given training example and any given differentiable activation function, this value can always be computed.

- Finally, we compute the derivative of s_i with respect to the weight $w_{i \leftarrow j}$, which turns out to be quite simply the value to the hidden node h_j.

$$\frac{ds}{dw_{i \leftarrow j}} = \frac{d}{dw_{i \leftarrow j}} \sum_j w_{i \leftarrow j} h_j = h_j \tag{2.12}$$

Where are we? In Equations 2.10 to 2.12, we computed the three steps needed to compute the gradient for the error function given the unfolded laid out in Equation 2.9. Putting it all together, we have

$$\begin{aligned} \frac{dE}{dw_{i \leftarrow j}} &= \frac{dE}{dy_i}\frac{dy_i}{ds_i}\frac{ds}{dw_{i \leftarrow j}} \\ &= -(t_i - y_i) \quad y_i' \quad h_j \end{aligned} \tag{2.13}$$

Factoring in a **learning rate** μ gives us the following update formula for weight $w_{i \leftarrow j}$. Note that we also remove the minus sign, since we move against the gradient towards the minimum.

$$\Delta w_{i \leftarrow j} = \mu (t_i - y_i)\ y_i'\ h_j$$

It is useful to introduce the concept of an **error term** δ_i. Note that this term is associated with a node, while the weight updates concern weights. The error term has to be computed only once for the node, and it can be then used for each of the incoming weights.

$$\delta_i = (t_i - y_i)\ y_i' \tag{2.14}$$

This reduces the update formula to:

$$\Delta w_{i \leftarrow j} = \mu\ \delta_i\ h_j \tag{2.15}$$

2.5.2 Weights to the hidden nodes

The computation of the gradient and hence the update formula for hidden nodes is quite analogous. As before, we first define the linear combination z_j (previously s_i) of input values x_k (previously hidden values h_j) weighted by weights $u_{j\leftarrow k}$ (previously weights $w_{i\leftarrow j}$).

$$z_j = \sum_k u_{j\leftarrow k} x_k \tag{2.16}$$

This leads to the computation of the value of the hidden node h_j.

$$h_j = \text{sigmoid}(z_j) \tag{2.17}$$

Following the principles of gradient descent, we need to compute the derivative of the error E with respect to the weights $u_{j\leftarrow k}$. We decompose this derivative as before.

$$\frac{dE}{du_{j\leftarrow k}} = \frac{dE}{dh_j}\frac{dh_j}{dz_j}\frac{dz_j}{du_{j\leftarrow k}} \tag{2.18}$$

However, the computation of $\frac{dE}{dh_j}$ is more complex than in the case of output nodes, since the error is defined in terms of output values y_i, not values for hidden nodes h_j. The idea behind back-propagation is to track how the error caused by the hidden node contributed to the error in the next layer. Applying the chain rule gives us:

$$\frac{dE}{dh_j} = \sum_i \frac{dE}{dy_i}\frac{dy_i}{ds_i}\frac{ds_i}{dh_j} \tag{2.19}$$

We already encountered the first two terms $\frac{dE}{dy_i}$ (Equation 2.10) and $\frac{dy_i}{ds_i}$ (Equation 2.11) previously. To recap:

$$\begin{aligned}
\frac{dE}{dy_i}\frac{dy_i}{ds_i} &= \frac{d}{dy_i}\sum_{i'}\frac{1}{2}(t_i - y_{i'})^2 \quad y_i' \\
&= \frac{d}{dy_i}\frac{1}{2}(t_i - y_i)^2 \quad y_i' \\
&= -(t_i - y_i) \quad y_i' \\
&= \delta_i
\end{aligned} \tag{2.20}$$

The third term in Equation 2.19 is computed straightforward.

$$\frac{ds_i}{dh_j} = \frac{d}{dh_j}\sum_i w_{i\leftarrow j} h_j = w_{i\leftarrow j} \tag{2.21}$$

Putting Equation 2.20 and Equation 2.21 together, Equation 2.19 can be solved as:

$$\frac{dE}{dh_j} = \sum_i \delta_i w_{i\leftarrow j} \tag{2.22}$$

This gives rise to a quite intuitive interpretation. The error that matters at the hidden node h_j depends on the error terms δ_i in the subsequent nodes y_i, weighted by $w_{i \leftarrow j}$, i.e., the impact the hidden node h_j has on the output node y_i.

Let us tie up the remaining loose ends. The missing pieces from Equation 2.18 are the second term

$$\frac{dh_j}{dz_j} = \frac{d\,\text{sigmoid}(z_j)}{dz_j} = \text{sigmoid}(z_j)(1 - \text{sigmoid}(z_j)) = h_j(1 - h_j) = h'_j \tag{2.23}$$

and third term

$$\frac{dz_j}{du_{j \leftarrow k}} = \frac{d}{du_{j \leftarrow k}} \sum_k u_{j \leftarrow k} x_k = x_k \tag{2.24}$$

Putting Equation 2.22, Equation 2.23, and Equation 2.24 together gives us the gradient

$$\begin{aligned} \frac{dE}{du_{j \leftarrow k}} &= \frac{dE}{dh_j} \frac{dh_j}{dz_j} \frac{dz_j}{du_{j \leftarrow k}} \\ &= \sum_i (\delta_i w_{i \leftarrow j}) \quad h'_j \quad x_k \end{aligned} \tag{2.25}$$

If we define an error term δ_j for hidden nodes analogous to output nodes

$$\delta_j = \sum_i (\delta_i w_{i \leftarrow j}) \quad h'_j \tag{2.26}$$

then we have an analogous update formula

$$\Delta u_{j \leftarrow k} = \mu \; \delta_j \; x_k \tag{2.27}$$

2.5.3 Summary

We train neural networks by processing training examples, one at a time, and update weights each time. What drives weight updates is the gradient towards a smaller error. Weight updates are computed based on error terms δ_i associated with each non-input node in the network.

For output nodes, the error term δ_i is computed from the actual output y_i of the node for our current network, and the target output t_i for the node.

$$\delta_i = (t_i - y_i) \; y'_i \tag{2.28}$$

For hidden nodes, the error term δ_j is computed via back-propagating the error term δ_i from subsequent nodes connected by weights $w_{i \leftarrow j}$.

$$\delta_j = \sum_i (\delta_i w_{i \leftarrow j}) \; h'_j \tag{2.29}$$

Computing y'_i and h'_j requires the derivative of the activation function, to which the weighted sum of incoming values is passed.

Given the error terms, weights $w_{i \leftarrow j}$ (or $u_{j \leftarrow k}$) from each proceeding node h_j (or x_k) are updated, tempered by a learning rate μ.

$$\begin{aligned} \Delta w_{i \leftarrow j} &= \mu \; \delta_i \; h_j \\ \Delta u_{j \leftarrow k} &= \mu \; \delta_j \; x_k \end{aligned} \tag{2.30}$$

Once weights are updated, the next training example is processed. There are typically several passes over the training set, called epochs.

2.5.4 Example

Given the neural network in Figure 2.4, let us see how the training example (1,0) $\rightarrow$ 1 is processed.

Let us start with the calculation of the error term δ for the output node y_0. During inference (recall Table 2.1 on page 15), we computed the linear combination of weighted hidden node values $s_0 = 1.060$ and the node value $y_0 = 0.743$. The target value is $t_0 = 1$.

$$\delta = (t_0 - y_0) \; y_0' = (1 - 0.743) \times \text{sigmoid}'(1.060) = 0.257 \times 0.191 = 0.049 \tag{2.31}$$

With this number, we can compute weight updates, such as for weight $w_{0 \leftarrow 0}$.

$$\Delta w_{0 \leftarrow 0} = \mu \; \delta_0 \; h_0 = \mu \times 0.049 \times 0.731 = \mu \times 0.036 \tag{2.32}$$

Since the hidden node h_0 leads only to one output node y_0, the calculation of its error term δ_0 is not more computationally complex.

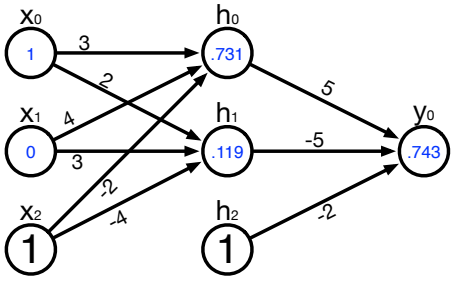
$$\delta_j = \sum_i (\delta_i u_{i \leftarrow 0}) \;\; h_0' = (\delta \times w_{0 \leftarrow 0}) \times \text{sigmoid}'(z_0) = 0.049 \times 5 \times 0.197 = 0.049 \tag{2.33}$$

Table 2.2 summarizes the updates for all weights.

2.6 Refinements

We conclude our introduction to neural networks with some basic refinements and considerations. To motivate some of the refinements, consider Figure 2.6. While gradient descent training is a fine idea, it may run into practical problems.

- Setting the learning rate too high leads to updates that overshoot the optimum. Conversely, a too low learning rate leads to slow convergence.
- Bad initialization of weights may lead to long paths of many update steps to reach the optimum. This is especially a problem with activation functions like sigmoid which only have a short interval of significant change.
- The existence of local optima lead the search to get trapped and miss the global optimum.



Node	Error term	Weight updates
	$\delta = (t_0 - y_0)$ sigmoid(s_0)	$\Delta w_{0 \leftarrow j} = \mu\ \delta\ h_j$
y_0	$\delta = (1 - 0.743) \times 0.191 = 0.049$	$\Delta w_{0 \leftarrow 0} = \mu \times 0.049 \times 0.731 = 0.036$
		$\Delta w_{0 \leftarrow 1} = \mu \times 0.049 \times 0.119 = 0.006$
		$\Delta w_{0 \leftarrow 2} = \mu \times 0.049 \times 1 = 0.049$
	$\delta_j = \delta\ w_{i \leftarrow j}$ sigmoid(z_j)	$\Delta u_{j \leftarrow i} = \mu\ \delta_j\ x_i$
h_0	$\delta_0 = 0.049 \times 5 \times 0.197 = 0.048$	$\Delta u_{0 \leftarrow 0} = \mu \times 0.048 \times 1 = 0.048$
		$\Delta u_{0 \leftarrow 1} = \mu \times 0.048 \times 0 = 0$
		$\Delta u_{0 \leftarrow 2} = \mu \times 0.048 \times 1 = 0.048$
h_1	$\delta_1 = 0.049 \times -5 \times 0.105 = -0.026$	$\Delta u_{1 \leftarrow 0} = \mu \times -0.026 \times 1 = -0.026$
		$\Delta u_{1 \leftarrow 1} = \mu \times -0.026 \times 0 = 0$
		$\Delta u_{1 \leftarrow 2} = \mu \times -0.026 \times 1 = -0.026$

Table 2.2: Weight updates (with unspecified learning rate μ) for the neural network in Figure 2.4 (repeated above the table) when the training example (1,0) $\rightarrow$ 1 is presented.

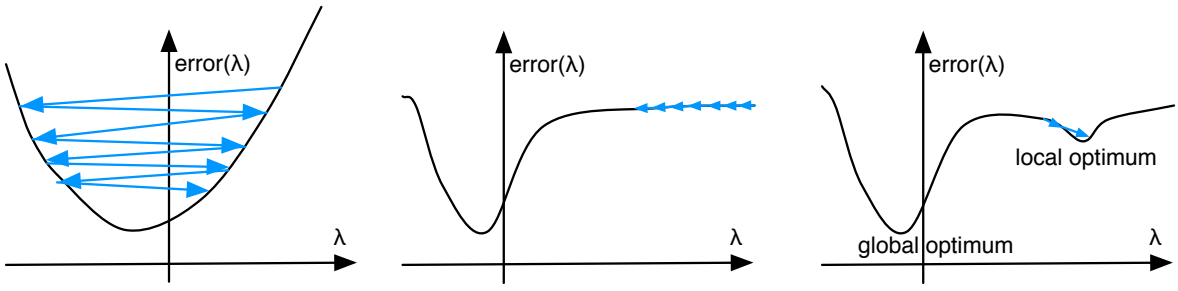


Figure 2.6: Problems with gradient descent training that motivate some of the refinements detailed in Section 2.6: (a) a too high learning rate may lead to too drastic parameter updates, overshooting the optimum, (b) bad initialization may require many updates to escape a plateau, and (c) the existence of local optima which trap training.

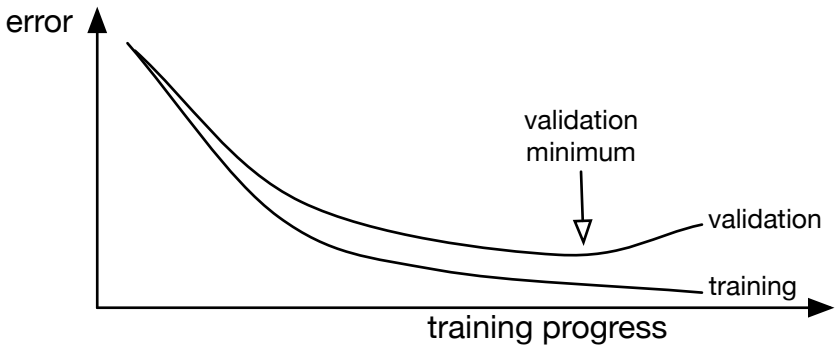


Figure 2.7: Training progress over time. The error on the training set continuously decreases. However, on a validation set (not used for training), at some point the error increases. Training is stopped at the validation minimum before such over-fitting sets in.

2.6.1 Validation Set

Neural network training proceeds for several epochs, i.e., full iterations over the training data. When to stop? When we track training progress, we see that the error on the training set continuously decreases. However, at some point **over-fitting** sets in, where the training data is memorized and not sufficiently generalized.

We can check this with an additional set of examples, called the **validation set**, that is not used during training. See Figure 2.7 for an illustration. When we measure the error on the validation set at each point of training, we see that at some point this error increases. Hence, we stop training, when the minimum on the validation set is reached.

2.6.2 Weight Initialization

Before training starts, weights are initialized to random values. The values are choses from a uniform distribution. We prefer initial weights that lead to node values that are in the transition area for the activation function, and not in the low or high shallow slope where it would take a long time to push towards a change. For instance, for the sigmoid activation function, feeding values in the range of, say, $[-1;1]$ to the activation function leads to activation values in the range of [0.269;0.731].

For the sigmoid activation function, commonly used formula for weights to the final layer of a network are

$$\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right] \tag{2.34}$$

where n is the size of the previous layer. For hidden layers, we chose weights from the range

$$\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right] \tag{2.35}$$

where n_j is the size of the previous layer, n_j size of next layer.

2.6.3 Momentum Term

Consider the case where a weight value is far from its optimum. Even if most training examples push the weight value in the same direction, it may still take a while for each of these small updates to accumulate until the weight reaches its optimum. A common trick is to use a **momentum term** to speed up training. This momentum term m_t gets updated at each time step t (i.e., for each training example). We combine the previous value of the momentum term m_{t-1} with the current raw weight update value Δw_t and use the resulting momentum term value to update the weights.

For instance, with a decay rate of 0.9, the update formula changes to

$$\begin{aligned} m_t &= 0.9 m_{t-1} + \Delta w_t \\ w_t &= w_{t-1} - \mu\, m_t \end{aligned} \tag{2.36}$$

2.6.4 Adapting Learning Rate per Parameter

A common training strategy is to reduce the learning rate μ over time. At the beginning the parameters are far away from optimal values and have to change a lot, but in later training stages we are concerned with fine tuning, and a large learning rate may cause a parameter to bounce around an optimum.

But different parameters may be at different stages on the path to their optimal values, so a different learning rate for each parameter may be helpful. One such method, called **Adagrad**, records the gradients that were computed for each parameter and accumulates their square values over time, and uses this sum to adjust the learning rate.

The Adagrad update formula is based on the sum of gradients of the error E with respect to the weight w at all time steps t, i.e., $g_t = \frac{dE_t}{dw}$. We divide the learning rate μ for this weight this accumulated sum.

$$\Delta w_t = \frac{\mu}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2}}\, g_t \tag{2.37}$$

Intutively, big changes in the parameter value (corresponding to big gradients g_t), lead to a reduction of the learning rate of the weight parameter.

Combining the idea of momentum term and adjusting parameter update by their accumulated change is the inspiration of **Adam**, another method to transform the raw gradient into a parameter update.

First, there is the idea of momentum, which is computed as in Equation 2.36 above.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.38}$$

Then, there is the idea of the squares of gradients (as in Adagrad) for adjusting the learning rate. Since raw accumulation does run the risk of becoming too large and hence permanently depressing the learning rate, Adam uses exponential decay, just like for the momentum term.

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{2.39}$$

The hyper parameters β_1 and β_2 are set typically close to 1, but this also means that early in training the values for m_t and v_t are close to their initialization values of 0. To adjust for that, they are corrected for this bias.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{2.40}$$

With increasing training time steps t, this correction goes away: $\lim_{t\to\infty} \frac{1}{1-\beta^t} \to 1$.

Having these pieces in hand (learning rate μ, momentum $\hat{m}_t$, accumulated change $\hat{v}_t$), weight update per Adam is computed as

$$\Delta w_t = \frac{\mu}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t \tag{2.41}$$

Common values for the hyper parameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

There are various other adaptation schemes. This is an active area of research. For instance, the second order gradient gives some useful information about the rate of change. However, it is often expensive to compute, so other shortcuts are taken.

2.6.5 Dropout

The parameter space in which back-propagation learning and its variants are operating is littered with local optima. The hill-climbing algorithm may just climb a mole hill and be stuck there, instead of moving towards a climb of the highest mountain. Various methods have been proposed to get training out of these local optima. One currently popular method in neural machine translation is called **drop-out**.

It sounds a bit simplistic and wacky. During training, some of the nodes of the neural network are ignored. Their values are set to 0, and their associated parameters are not updated. These dropped-out nodes are chosen at random, and may account for as much as 10%, 20% or even more of all the nodes. Training resumes for some number of iterations without the nodes, and then a different set of drop-out nodes are selected.

The dropped-out nodes played some useful role in the model trained up to the point where they are ignored. After that, other nodes have to pick up the slack. The end result is a more robust model where several nodes share similar roles.

2.6.6 Layer Normalization

Layer normalization addresses a problem that arises especially in the deep neural networks that we are using in neural machine translation, where computing proceeds through a large sequence of layers. For some training examples, average values at one layer may become very large, which feed into the following layer, also producing large output values, and so on. This is especially a problem with activation functions that do not limit the output to a narrow interval, such as rectified linear units. For other training examples the average values at the same layers

may be very small. This causes a problem for training. Recall from Equation 2.30, that gradient updates are strongly effected by node values. Too large node values lead to exploding gradients and too small node values lead to diminishing gradients.

To remedy this, the idea is to normalize the values on a per-layer basis. This is done by adding additional computational steps to the neural network. Recall that a feed-forward layer consists of the the matrix multiplication of the weight matrix W with the node values from the previous layer $\vec{h^{l-1}}$, resulting in a weighted sum $\vec{s^l}$, followed by an activation function such as sigmoid.

$$\begin{aligned} \vec{s^l} &= W\ \vec{h^{l-1}} \\ \vec{h^l} &= \text{sigmoid}(\vec{h^l}) \end{aligned} \tag{2.42}$$

We can compute the mean and variance of the values in the weighted sum vector $\vec{s^l}$ by

$$\begin{aligned} \mu^l &= \frac{1}{H}\sum_{i-1}^{H} s_i^l \\ \sigma^l &= \sqrt{\frac{1}{H}\sum_{i-1}^{H}(s_i^l - \mu^l)^2} \end{aligned} \tag{2.43}$$

Using these values, we normalize the vector $\vec{s^l}$ using two additional bias vectors $\vec{g}$ and $\vec{b}$

$$\vec{\hat{s}^l} = \frac{\vec{g}}{\sigma^l} \cdot (\vec{s^l} - \mu^l) + \vec{b} \tag{2.44}$$

where $\cdot$ is element-wise multiplication and the difference subtracts the scalar average from each vector element.

The formula first normalizes the values in $\vec{s^l}$ by shifting them against their average value, hence ensuring that their average afterwards is 0. The resulting vector is then divided by the variance σ^l. The additional bias vectors give some flexibility, they may be shared across multiple layers of the same type, such as multiple time steps in a recurrent neural network (we will introduce these in Section 4.4 on page 44).

2.6.7 Mini Batches

Each training example yields a set of weight updates Δw_i. We may first process all the training examples and only afterwards apply all the updates. But neural networks have the advantage that they can immediately learn from each training example. A training method that updates the model with each training example is called **online learning**. The online learning variant of gradient descent training is called **stochastic gradient descent**.

Online learning generally takes fewer passes over the training set (called **epochs**) for convergence. However, since training constantly changes the weights, it is hard to parallelize. So, instead, we may want to process the training data in batches, accumulate the weight updates, and then apply them collectively. These smaller sets of training examples are called

mini batches to distinguish this approach from **batch training** where the entire training set is considered one batch.

There are other variations to organize the processing of the training set, typically motivated by restrictions of parallel processing. If we process the training data in mini batches, then we can parallelize the computation of weight update values Δw, but have to synchronize their summation and application to the weights. If we want to distribute training over a number of machines, it is computationally more convenient to break up the training data in equally sized parts, perform online learning for each of the parts (optionally using smaller mini batches), and then average the weights. Surprisingly, breaking up training this way, often leads to better results than straightforward linear processing.

Finally, a scheme called **Hogwild** runs several training threads that immediately update weights, even though other threads still use the weight values to compute gradients. While this is clearly violates the safe guards typically taken in parallel programming, it does not hurt in practical experience.

2.6.8 Vector and Matrix Operations

We can express the calculations needed for handling neural networks as vector and matrix operations.

- Forward computation: $\vec{s} = W\ \vec{h}$
- Activation function: $\vec{y} = \text{sigmoid}(\vec{h})$
- Error term: $\vec{\delta} = (\vec{t} - \vec{y})\ \text{sigmoid'}(\vec{s})$
- Propagation of error term: $\vec{\delta}_i = W\ \vec{\delta}_{i+1} \cdot \text{sigmoid'}(\vec{s})$
- Weight updates: $\Delta W = \mu\ \vec{\delta}\ \vec{h}^T$

Executing these operations is computationally expensive. If our layers have, say, 200 nodes, then the matrix operation $W\vec{h}$ requires $200 \times 200 = 40,000$ multiplications. Such matrix operations are also common in another highly used area of computer science: graphics processing. When rendering images on the screen, the geometric properties of 3-dimensional objects have to be processed to generate the color values of the 2-dimensional image on the screen. Since there is high demand for fast graphics processing, for instance for the use in realistic looking computer games, specialized hardware has become commonplace: **graphics processing units (GPUs)**.

These processors have a massive number of cores (for example, the NVIDIA GTX 1080ti GPU provides 3584 thread processors) but a rather lightweight instruction set. GPUs provide instructions that are applied to many data points at once, which is exactly what is needed out the vector space computations listed above. Programming for GPUs is supported by various libraries, such as CUDA for C++, and has become an essential part of developing large scale neural network applications.

The general term for scalars, vectors, and matrices is **tensors**. A tensor may also have more dimensions: a sequence of matrices can be packed into a 3-dimensional tensor. Such large objects are actually frequently used in today's neural network toolkits.

Further Readings A good introduction to modern neural network research is the textbook ”Deep Learning” (Goodfellow et al., 2016). There is also book on neural network methods applied to the natural language processing in general (Goldberg, 2017).

A number of key techniques that have been recently developed have entered the standard repertoire of neural machine translation research. Training is made more robust by methods such as drop-out (Srivastava et al., 2014), where during training intervals a number of nodes are randomly masked. To avoid exploding or vanishing gradients during back-propagation over several layers, gradients are typically clipped (Pascanu et al., 2013). Layer normalization (Lei Ba et al., 2016) has similar motivations, by ensuring that node values are within reasonable bounds.

An active topic of research are optimization methods that adjust the learning rate of gradient descent training. Popular methods are Adagrad (Duchi et al., 2011), Adadelta (Zeiler, 2012), and currently Adam (Kingma and Ba, 2015).