1 Gradient descent

Gradient descent is an iterative minimization method. The gradient of the error function always shows in the direction of the steepest ascent of the error function. Thus, we can start with a random weight vector and subsequently follow the *negative* gradient (using a learning rate η)

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \Delta\mathbf{w}_t, \quad \Delta\mathbf{w}_t = -\eta \left.\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}^T\right|_{\mathbf{w}_t} \tag{1}$$

Note that we will only find a local minimum, thus it is recommended to repeat the gradient descent procedure for several random initial states.

1.1 Variants

Impulse term:

$$\Delta\mathbf{w}_t = \alpha\Delta\mathbf{w}_{t-1} - (1-\alpha)\eta \left.\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}^T\right|_{\mathbf{w}_t}, \tag{2}$$

for $0 \leq \alpha \leq 1$.

Adaptive learning rate: Increase η if error has decreased, decrease η otherwise.

2 Gradient descent for neural networks: The Backpropagation algorithm

Consider a two-layer neural network with the following structure (blackboard):

Hidden Layer:

$$a_j^{(1)} = \sum_i w_{ji}^{(1)} x_i = \mathbf{w}_j^{(1)}\mathbf{x}, \quad \mathbf{a}^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{bmatrix} = \mathbf{W}^{(1)}\mathbf{x}, \tag{3}$$

with $\mathbf{W}_{ji}^{(1)} = w_{ji}^{(1)}$ being the weight matrix of the hidden layer, the jth row contains all weights of neuron j.

$$z_j = h_{(1)}(a_j^{(1)}), \quad \frac{\partial z_j}{\partial a_j^{(1)}} = h'_{(1)}(a_j^{(1)}), \quad \mathbf{z} = \begin{bmatrix} h_{(1)}(a_1^{(1)}) \\ h_{(1)}(a_2^{(1)}) \\ \vdots \\ h_{(1)}(a_m^{(1)}) \end{bmatrix}, \frac{\partial \mathbf{z}}{\partial \mathbf{a}^{(1)}} = \mathrm{diag}([h'_{(1)}(a_j^{(1)})]) = \mathbf{H}_{(1)} \tag{4}$$

Output Layer:

$$a_k^{(2)} = \sum_j w_{kj}^{(2)} z_j = \mathbf{w}_k^{(2)}\mathbf{z}, \quad \mathbf{a}^{(2)} = \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ \vdots \\ a_n^{(2)} \end{bmatrix} = \mathbf{W}^{(2)}\mathbf{z} \tag{5}$$

$$o_k = h_{(2)}(a_k^{(2)}), \quad \frac{\partial o_k}{\partial a_k^{(2)}} = h'_{(2)}(a_k^{(2)}), \quad \mathbf{o} = \begin{bmatrix} h_{(2)}(a_1^{(2)}) \\ h_{(2)}(a_2^{(2)}) \\ \vdots \\ h_{(2)}(a_n^{(2)}) \end{bmatrix}, \frac{\partial \mathbf{o}}{\partial \mathbf{a}_{(2)}} = \text{diag}([h'_{(2)}(a_k^{(2)})]) = \mathbf{H}_{(2)} \quad (6)$$

The network has d inputs, m hidden neurons and n output neurons. The transfer function of the network is therefore given by

$$o_k = h_{(2)}(\sum_{j=1}^{m} w_{kj}^{(2)} h_{(1)}(\sum_{i=1}^{d} w_{ji}^{(1)} x_i))$$

or in matrix form

$$\mathbf{o} = \mathbf{h}_{(2)} \left(\mathbf{W}^{(2)} \mathbf{h}_{(1)} (\mathbf{W}^{(1)} \mathbf{x}) \right)$$

Calculating the outputs $\mathbf{o}$ as a function of the inputs $\mathbf{x}$ is also denoted as forward sweep in the backpropagation algorithm.

2.1 Derivation via matrix derivatives (single training example)

For a single training example $\langle \mathbf{x}, \mathbf{y} \rangle$ the squared error function is given by

$$E = \sum_{k=1}^{n} (o_k - y_k)^2 = (\mathbf{o} - \mathbf{y})^T (\mathbf{o} - \mathbf{y})$$

When calculating the derivative w.r.t. the weights $\mathbf{w}_i^{(l)}$ of neuron i in layer l, we can use the chain rule to decompose the derivatives

$$\frac{\partial E}{\partial \mathbf{w}_i^{(l)}} = \frac{\partial E}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial \mathbf{w}_i^{(l)}}$$

We will denote $\delta_i^{(l)} = \frac{\partial E}{\partial a_i^{(l)}}$ as sensitivity or error of neuron i in layer l and $\boldsymbol{\delta}_{(l)}$ as error vector of layer l.

Note that the derivative $\frac{\partial a_i^{(l)}}{\partial \mathbf{w}_i^{(l)}}$ of the summed input $a_i^{(l)}$ w.r.t the weights is given by the input of the corresponding layer

$$\frac{\partial a_j^{(1)}}{\mathbf{w}_j^{(1)}} = \mathbf{x}^T, \quad \frac{\partial a_k^{(2)}}{\partial \mathbf{w}_k^{(2)}} = \mathbf{z}^T.$$

Hence, the gradient of the error function is always given by the product of the error with the inputs

$$\frac{\partial E}{\partial \mathbf{w}_j^{(1)}} = \frac{\partial E}{\partial a_j^{(1)}} \frac{\partial a_j^{(1)}}{\partial \mathbf{w}_j^{(1)}} = \delta_j^{(1)} \mathbf{x}^T, \quad \frac{\partial E}{\partial w_{ji}^{(1)}} = \delta_j^{(1)} x_i \quad (7)$$

$$\frac{\partial E}{\partial \mathbf{w}_k^{(2)}} = \frac{\partial E}{\partial a_k^{(2)}} \frac{\partial a_k^{(2)}}{\partial \mathbf{w}_k^{(2)}} = \delta_k^{(2)} \mathbf{z}^T, \quad \frac{\partial E}{\partial w_{kj}^{(2)}} = \delta_k^{(2)} z_j \quad (8)$$

Output layer:

$$\boldsymbol{\delta}_{(2)}^T = \frac{\partial E}{\partial \mathbf{a}^{(2)}} = \frac{\partial (\mathbf{o} - \mathbf{y})^T (\mathbf{o} - \mathbf{y})}{\partial \mathbf{a}^{(2)}} \quad (9)$$

$$= 2(\mathbf{o} - \mathbf{y})^T \mathbf{H}_{(2)}, \quad (10)$$

with

$$\mathbf{H}_{(2)} = \frac{\partial \mathbf{o}}{\partial \mathbf{a}^{(2)}} = \text{diag}([h'_{(2)}(a_k^{(2)})]).$$

For a single error $\delta_k^{(2)}$ this yields

$$\delta_k^{(2)} = 2(o_k - y_k)h'_{(2)}(a_k^{(2)}). \tag{11}$$

Hidden layer:

$$\begin{aligned}
\boldsymbol{\delta}_{(1)}^T &= \frac{\partial E}{\partial \mathbf{a}^{(1)}} = \frac{\partial E}{\partial \mathbf{a}^{(2)}} \frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{a}^{(1)}} && (12)\\
&= \boldsymbol{\delta}_{(2)}^T \frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{a}^{(1)}} && (13)\\
&= \boldsymbol{\delta}_{(2)}^T \frac{\partial \mathbf{W}^{(2)}\mathbf{z}}{\partial \mathbf{a}^{(1)}} && (14)\\
&= \boldsymbol{\delta}_{(2)}^T \mathbf{W}^{(2)} \frac{\partial \mathbf{z}}{\partial \mathbf{a}^{(1)}} && (15)\\
&= \boldsymbol{\delta}_{(2)}^T \mathbf{W}^{(2)} \mathbf{H}_{(1)} && (16)
\end{aligned}$$

For a single error $\delta_j^{(1)}$ this results in the familiar recursive update equation

$$\delta_j^{(1)} = \sum_{k=1}^{n} \delta_k^{(2)} w_{kj}^{(2)} h'_{(1)}(a_j^{(1)}) \tag{17}$$

This equation is known as the backwards sweep of backpropagation. Starting at the output neurons, we calculate the error and propagate the error backwards through the network. As we can see the iterative equations solely result from the application of the chain rule. Note that the sum in Equation 17 goes over all error terms $\delta_k^{(2)}$ of the (subsequent) output layer. The weights $w_{kj}^{(2)}$ indicate how much neuron j in the hidden layer has contributed to the output (and hence the error) of neuron k in the output layer.

2.2 Summary

The weight derivatives are always given by the product of the error $\delta^{(l)}$ with the corresponding inputs

Output Layer

$$\frac{\partial E}{\partial w_{kj}^{(2)}} = \frac{\partial E}{\partial a_k^{(2)}} \frac{\partial a_k^{(2)}}{\partial w_{kj}^{(2)}} = \delta_k^{(2)} z_j \tag{18}$$

Hidden Layer

$$\frac{\partial E}{\partial w_{ji}^{(1)}} = \frac{\partial E}{\partial a_j^{(1)}} \frac{\partial a_j^{(1)}}{\partial w_{ji}^{(1)}} = \delta_j^{(1)} x_i \tag{19}$$

The error $\delta^{(l)}$ can be calculated by:

Output Layer

$$\delta_k^{(2)} = h'_{(2)}(a_k^{(2)}) 2(o_k - y_k) \tag{20}$$

Hidden Layer

$$\delta_j^{(1)} = h'_{(1)}(a_j^{(1)}) \sum_{k=1}^{n} w_{kj}^{(2)} \delta_k^{(2)} \tag{21}$$

2.3 Additional hints

Learning rules for biases

If a neural network has additional constant inputs $x_0 = 1$ and $z_0 = 1$, the derivatives for the weights $w_{k0}^{(2)}$ and $w_{j0}^{(1)}$ can also be obtained from Eq. 18 and 19, respectively.

Multiple training examples

The performed derivations apply to a single training example $\langle \mathbf{x}, \mathbf{y} \rangle$. For a set of N training examples $\langle \mathbf{x}_l, \mathbf{y}_l \rangle$, $l = 1 \ldots N$, the cost function is defined as the sum of individual costs,

$$E(\mathbf{w}) = \sum_{l=1}^{N} E_l(\mathbf{w}) = \sum_{l=1}^{N} (\mathbf{o}_l - \mathbf{y}_l)^T (\mathbf{o}_l - \mathbf{y}_l) \quad , \tag{22}$$

($\mathbf{o}_l$ denotes the output vector computed by the network for the input $\mathbf{x}_l$). The gradient of this total cost function is then simply the sum of individual gradients,

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \sum_{l=1}^{N} \frac{\partial E_l(\mathbf{w})}{\partial \mathbf{w}} \quad . \tag{23}$$

3 Practical Issues

3.1 Activation functions

For regression we typically use the sigmoid function $h_{(1)}(x) = \sigma(x) = 1/(1+\exp(-x))$ as activation function in the hidden layer and a linear activation function in the output layer $h_{(2)}(x) = x$.

For classification we can't directly use the class labels as target values. Instead, we will use multidimensional target vectors which are obtained from a 1 out of n coding. 1 out of n coding returns a vector being zero everywhere except for the index of the class label, where the value is 1. For example, if we have 4 classes, and our class label is $c = 3$, the target vector is given by $\mathbf{y} = [0, 0, 1, 0]^T$. We will again use sigmoid activation functions for the hidden and also sigmoid activation functions for the output layer. Note that we have K output neurons, where K denotes the number of classes.

3.2 Model Selection with Cross Validation

Suppose we want to select one model out of M models but we do NOT have a large test set to calculate the true error of the learned functions. We could use a certain percentage of our training data as test set, however, in order to have an accurate estimate of the error we need a large test set. But we also do not want to waste too much of our precious training data just for testing. In this case we can use cross validation. The principle is simple. Divide the training data into K disjoint subsets. Now train on $K - 1$ subsets and test on the remaining subset. This has to be repeated K times for all training set / test set combinations. The true error of the hypothesis is then estimated by the average over all test set errors.

3.3 Neural network toolbox

The usage of the toolbox is very simple, we can create a neural network with *newff*, initialize the network with *init*, train the network with *train* and simulate the network with *sim*. Consult the online tutorial on the course website for a more detailed explanation. Note that the toolbox requires that you provide the training data in a column matrix (i.e. each column is a training example!).

3.4 Normalizing the training data

We should always normalize the training data such that each dimension of the input has zero mean and unit variance. This prevents that dimensions with a high range of values dominate the output of the network when learning starts. In matlab we can use the *mapstd* function for this normalization

- Normalize training data

```
[x_train_n, ps] = mapstd(x_train);
```

- Apply the same normalization to test set

```
[x_test_n] = mapstd('apply', x_test, ps);
```

- Transform into original input space

```
[x_train] = mapstd('reverse', x_train_n, ps);
```

3.5 Weight decay

Weight decay just denotes the usage of the regularized error function

$$E = 1/N \sum_i (o(x_i) - y_i)^2 + \alpha \sum_j w_j^2$$

Again, the intuition behind this error function is to keep the weights small in order to prevent overfitting. Hence, we can use a neural network which is too complex (too many neurons), but regularize the network by setting α. In matlab this error function is implemented by using the *msereg* performance function

```
net.performFcn = 'msereg';
```

The exact function used by matlab is given by

$$E = \alpha 1/N \sum_i (o(x_i) - y_i)^2 + (1 - \alpha) \sum_j w_j^2,$$

for $0 \leq \alpha \leq 1$. Hence, for α close to 1.0, no regularization is used.

3.6 Early stopping

Early stopping (ES) is another regularization method. The key idea behind ES is a simple observation: during training with gradient descent, the error on the training set always decreases, however, the error on the test set might increase again (due to overfitting). Thus, we just stop the training where the error on the test set has reached its minimum. Since we usually also start the algorithm with rather small weights ES also avoids that the weights of the network grow too large.

In matlab, ES can be easily implemented by searching for the minimum index of the test set performance during training (the performance during training on a test set is returned by the train function). Then we can retrain a 2nd neural network for this optimal number of epochs. Note that the 2nd neural network has to use the same initial weights as the first, because the determined optimal number of epochs is just valid for this initial weight vector.