1. Neural Networks

This section presents a powerful and widely used learning algorithm inspired by natural neural networks in a human brain. Basically, neural networks are a generalization of the perceptron that uses a feature transform learnt from the data. Perceptron cannot implement nonlinearly-separable functions, but a composition of perceptrons (multi-layer perceptron (MLP)) can. In fact, a large 3-layer MLP can closely approximate any target function. The difficulty with MLP is that more weights should be learnt. If the hard threshold is used as an activation function, learning weights will become a hard combinatorial problem. That's why it is approximated by a smooth and differentiable tanh function. Such 'softened' MLP is called a neural network. The section then emphasizes the importance of a proper notation for neural networks and introduces forward propagation to compute the output of a neural network and backward propagation to compute sensitivity vectors. Backpropagation algorithm which is based on the applications of the chain rule can be used to calculate the gradient efficiently and obtain the learnt weights.

1.1 The Multi-layer Perceptron

1. *Please see the exercises on the next pages*

1.2 Neural Networks

1. *Please see the exercises on the next pages*

2. Algorithm to compute $E_{in}(w)$ and $g = \nabla E_{in}(w)$

To compute the gradient of E_{in}, its derivative with respect to each weight matrix $W^{(l)}$ is needed. Backpropagation is based on the applications of the chain rule and allows us to compute the partial derivatives with respect to each weight efficiently. As a result, $\frac{\partial e}{\partial W^{(l)}} = x^{(l-1)}\left(\delta^{(l)}\right)^T$. Therefore, we need the outputs $x^{(l)}$ and the sensitivities $\delta^{(l)}$ for each layer. The outputs can be computed using the forward propagation ($x^{(l)}$ is obtained from $x^{(l-1)}$), while the sensitivities are calculated with the backward propagation ($\delta^{(l)}$ is obtained from $\delta^{(l+1)}$).

So, the algorithm takes as inputs a dataset and weights that should be initialized to small random values (for example, to Gaussian random weights). In-sample error and its gradient should be initialized to zero. Then, for each data point, we run one forward and backward propagation and using the results from them, compute E_{in} through the output from the last layer $x^{(L)}$ and the gradient. Combining the results from all data points, we calculate E_{in} as a mean-squared error and get the full batch gradient. After that, the weights can be updated via the gradient.

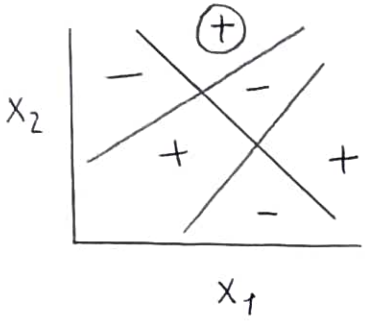
1.1 The Multi-layer Perceptron

1. Exercise 7.1

Show that $f = \bar{h}_1 h_2 h_3 + h_1 \bar{h}_2 h_3 + h_1 h_2 \bar{h}_3$

Using hint, we can conclude that there is a systematic way of going from a target function to a Boolean formula.

If we consider the '+' region of f at the top right:

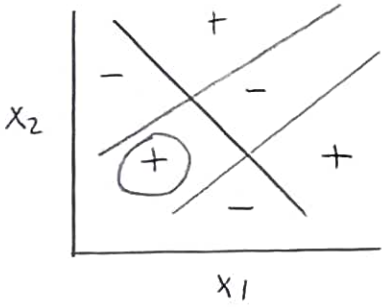


we have

h_1	h_2	h_3
−	+	+

$\Rightarrow$ we should use $\bar{h}_1 h_2 h_3$ to obtain '+' region

For the '+' region at the bottom left:

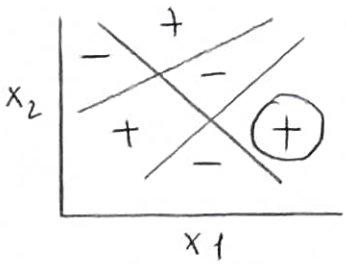


we have

h_1	h_2	h_3
+	+	−

$\Rightarrow$ we should have $h_1 h_2 \bar{h}_3$

For the '+' region at the bottom right:



we have

h_1	h_2	h_3
+	−	+

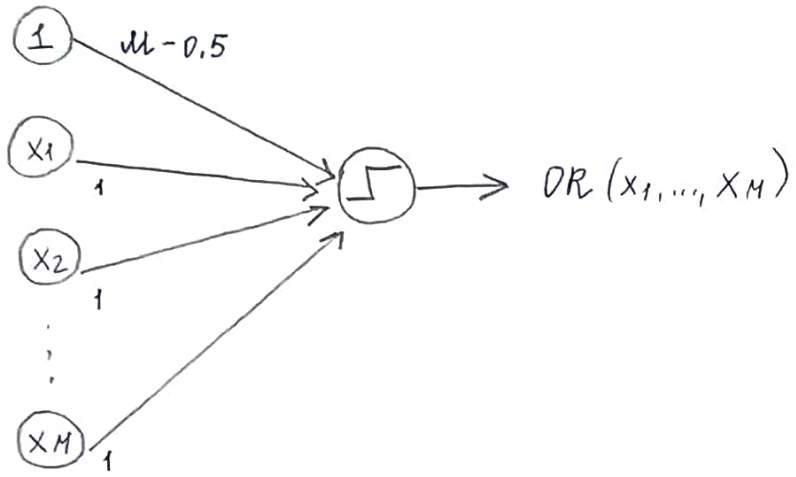
$\Rightarrow$ we should have $h_1 \bar{h}_2 h_3$

If either of them is 'TRUE', then f is 'TRUE'

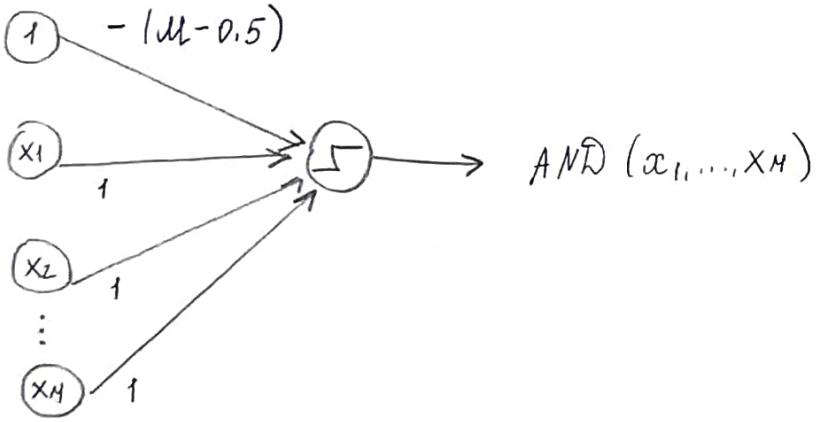
$\Rightarrow f = \bar{h}_1 h_2 h_3 + h_1 \bar{h}_2 h_3 + h_1 h_2 \bar{h}_3$ (OR of ANDs)

Exercise 7.2

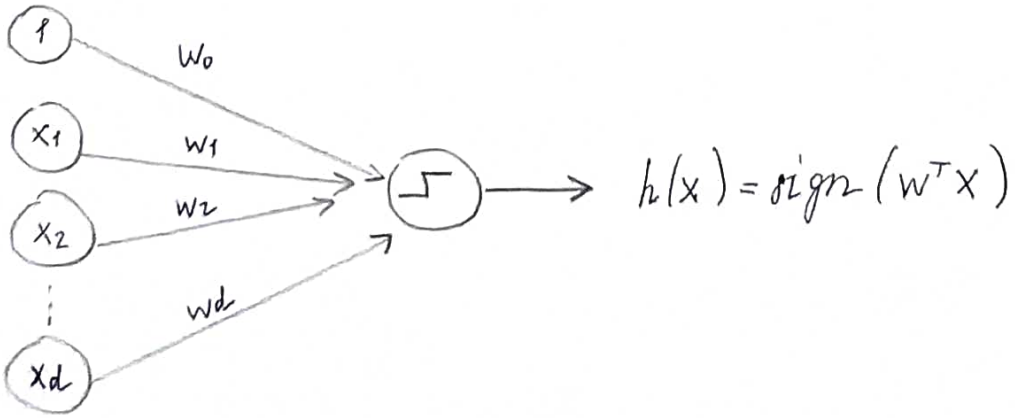
a) $OR(x_1, \dots, x_M)$



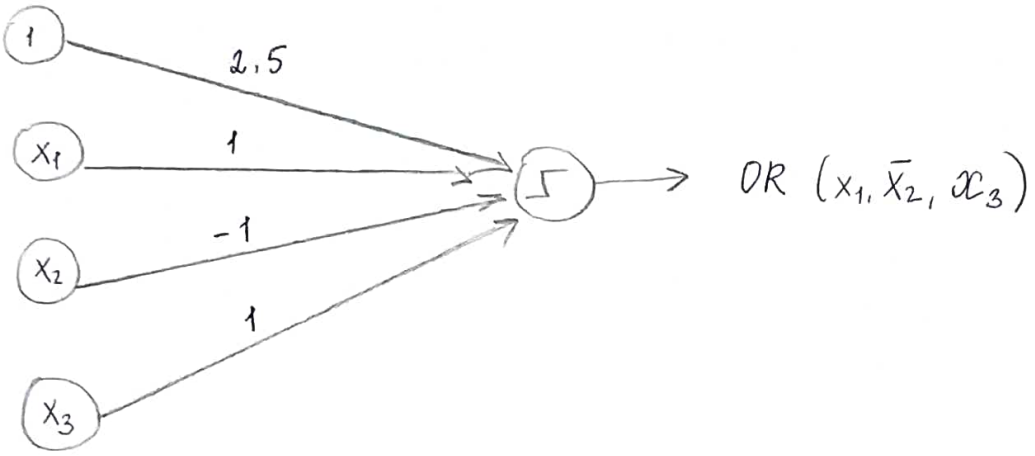
$AND(x_1, \dots, x_M)$



b) $h(x) = sign(w^T x)$ Assume the dimension of input is d



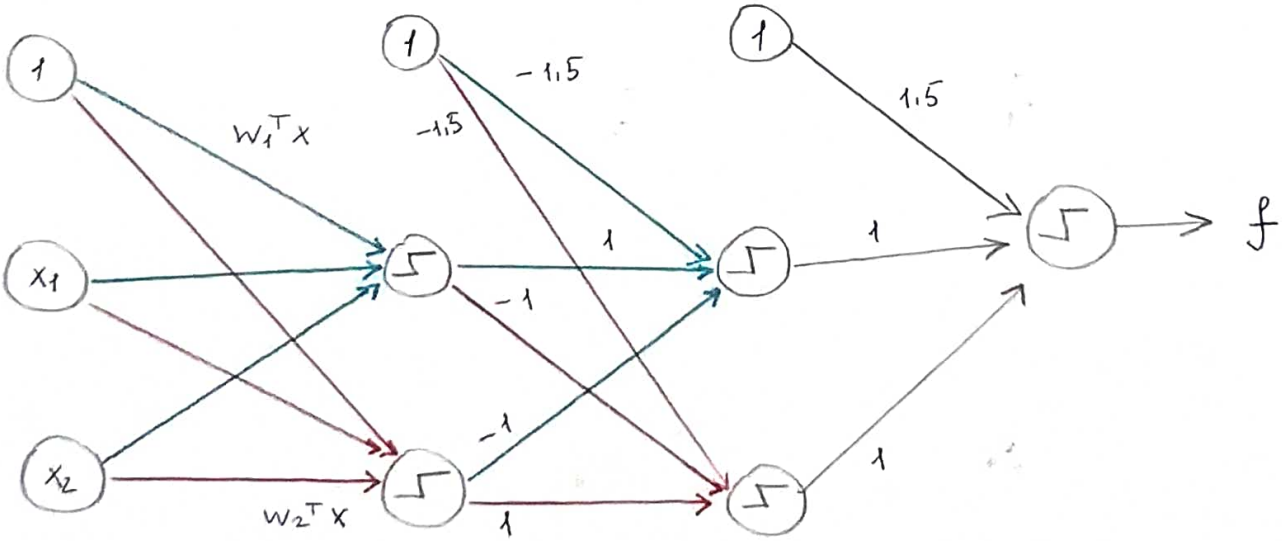
c) OR $(x_1, \bar{x}_2, x_3)$



Exercise 7.3

Show that $f(x) = \text{sign}\left[\text{sign}\left(h_1(x) - h_2(x) - \frac{3}{2}\right) - \text{sign}\left(h_1(x) - h_2(x) + \frac{3}{2}\right) + \frac{3}{2}\right]$

where $h_1(x) = \text{sign}(w_1^T x)$ and $h_2(x) = \text{sign}(w_2^T x)$



From the graph representation, it is clearly seen that:

$$f = \text{sign}\left(1.5 + \text{sign}\left(-1.5 + h_1(x) - h_2(x)\right) + \text{sign}\left(-1.5 - h_1(x) + h_2(x)\right)\right)$$

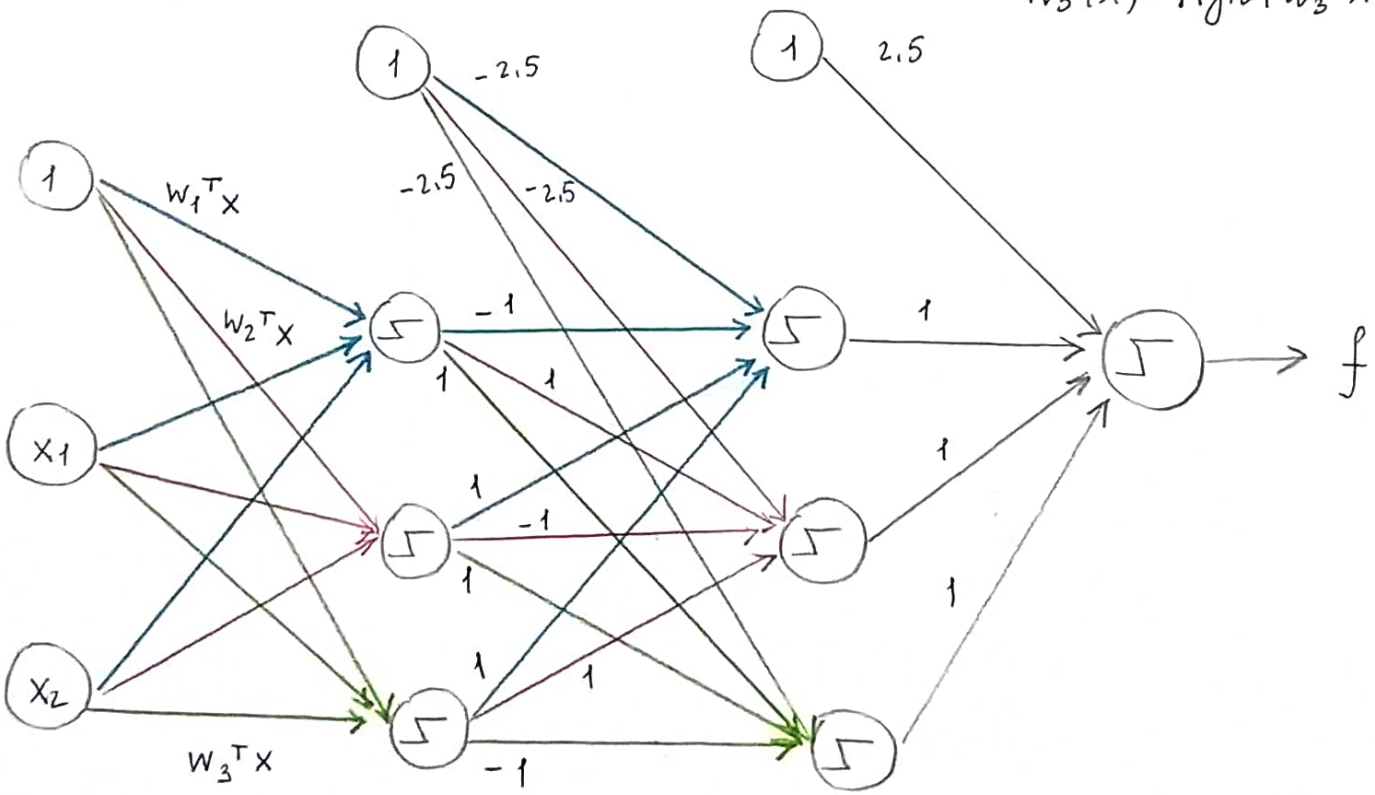
$$= \text{sign}\left[\text{sign}\left(h_1(x) - h_2(x) - \frac{3}{2}\right) - \text{sign}\left(h_1(x) - h_2(x) + \frac{3}{2}\right) + \frac{3}{2}\right]$$

where $h_1(x) = \text{sign}(w_1^T x)$ and $h_2(x) = \text{sign}(w_2^T x)$

Exercise 7.4

$$f = \bar{h}_1 h_2 h_3 + h_1 \bar{h}_2 h_3 + h_1 h_2 \bar{h}_3$$

$h_1(x) = \text{sign}(w_1^T x)$
$h_2(x) = \text{sign}(w_2^T x)$
$h_3(x) = \text{sign}(w_3^T x)$



$$f = \text{sign}\left[\text{sign}\left(-h_1(x) + h_2(x) + h_3(x) - 2.5\right) + \text{sign}\left(h_1(x) - h_2(x) + h_3(x) - 2.5\right) + \text{sign}\left(h_1(x) + h_2(x) - h_3(x) - 2.5\right) + 2.5\right]$$

1.2 Neural Networks

1. Exercise 7.6

$$V = \sum_{\ell=0}^{L} d^{(\ell)} \qquad Q = \sum_{\ell=1}^{L} d^{(\ell)} \left(d^{(\ell-1)} + 1 \right)$$

In forward propagation, for each layer ℓ

$s^{(\ell)} \leftarrow (W^{(\ell)})^T x^{(\ell-1)}$ is calculated

$\Rightarrow d^{(\ell)} (d^{(\ell-1)} + 1)$ multiplications and additions are made

Then, $x^{(\ell)} \leftarrow \begin{bmatrix} 1 \\ \theta(s^{(\ell)}) \end{bmatrix}$ is computed

$\Rightarrow d^{(\ell)}$ evaluations of θ are made

If we add the computations for all layers (from $\ell = 1$ to L),

then $O(Q)$ multiplications and additions, and $O(V)$

θ-evaluations are made.

Exercise 7.7

$$E_{in}(w) = \frac{1}{N} \sum_{n=1}^{N} (\tanh(w^T x_n) - y_n)^2$$

Show that

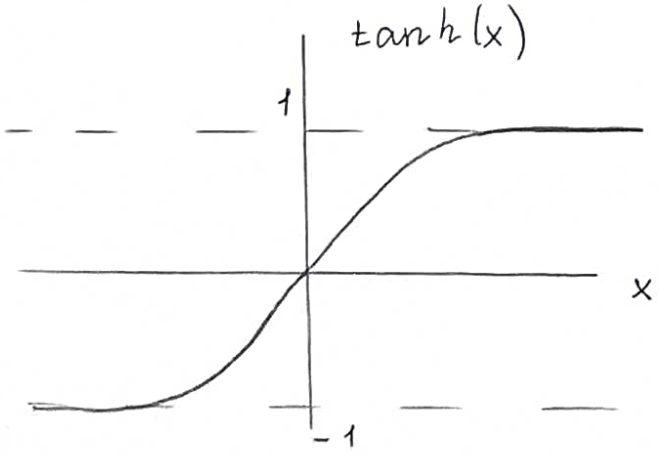
$$\nabla E_{in}(w) = \frac{2}{N} \sum_{n=1}^{N} (\tanh(w^T x_n) - y_n)(1 - \tanh^2(w^T x_n)) x_n$$

We know that $\frac{d}{dx}(\tanh x) = \operatorname{sech}^2 x = 1 - \tanh^2 x$

Using chain rule,

$$\nabla E_{in}(w) = \frac{\partial E_{in}(w)}{\partial w_i} = \frac{1}{N} \sum_{n=1}^{N} 2(\tanh(w^T x_n) - y_n) \cdot (1 - \tanh^2(w^T x_n)) \cdot x_n =$$

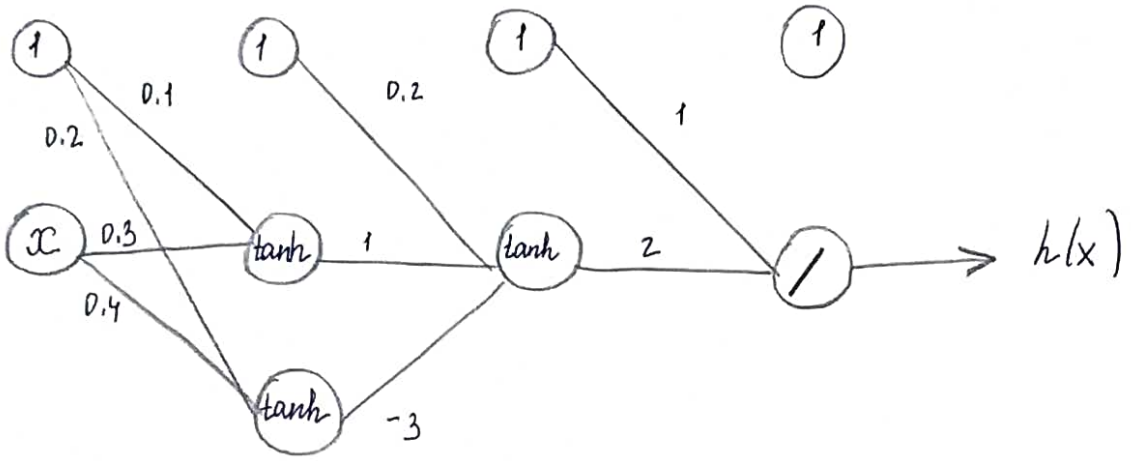
$$\frac{2}{N} \sum_{n=1}^{N} (\tanh(w^T x_n) - y_n)(1 - \tanh^2(w^T x_n)) x_n$$



=> if $w \to \infty$, $\tanh(w^T x_n) \to 1$ and $\nabla E_{in}(w) \to 0$

=> the gradient won't change => weights won't be updated => hard to optimize the perceptron

Exercise 7.8



$$W^{(1)} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix} \qquad W^{(2)} = \begin{bmatrix} 0.2 \\ 1 \\ -3 \end{bmatrix} \qquad W^{(3)} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

For the data point $x = 2$, $y = 1$, $x^{(0)} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$

As only the output transformation becomes the identity,

$s^{(1)}$, $x^{(1)}$, $s^{(2)}$, $x^{(2)}$, $s^{(3)}$ remain the same as in Example 7.1

$$\Rightarrow \quad s^{(1)} = \begin{bmatrix} 0.7 \\ 1 \end{bmatrix} \qquad x^{(1)} = \begin{bmatrix} 1 \\ 0.60 \\ 0.76 \end{bmatrix} \qquad s^{(2)} = [-1.48] \qquad x^{(2)} = \begin{bmatrix} 1 \\ -0.90 \end{bmatrix}$$

$$s^{(3)} = [-0.8] \qquad \Rightarrow \quad x^{(3)} = -0.8 \qquad \text{(output transformation is identity)}$$

We know that $\delta^{(3)} = 2\,(x^{(3)} - y)\,\theta'(s^{(3)})$ and

$\theta'(s^{(3)}) = 1$ (the identity)

$$\Rightarrow \delta^{(3)} = \alpha\,(-0.8 - 1)\cdot 1 = [-3.6]$$

Also, $\delta^{(l)} = \theta'(s^{(l)}) \otimes [W^{(l+1)}\,\delta^{(l+1)}]_1^{d^{(l)}}$

where $\theta'(s^{(l)}) = [1 - x^{(l)} \otimes x^{(l)}]_1^{d^{(l)}}$ for $l = 1, 2$

$$\Rightarrow \delta^{(2)} = \left(\begin{bmatrix}1\\1\end{bmatrix} - \begin{bmatrix}1\\-0.9\end{bmatrix}\begin{bmatrix}1\\-0.9\end{bmatrix}\right)\begin{bmatrix}1\\2\end{bmatrix}[-3.6] = \begin{bmatrix}0\\0.19\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix}[-3.6] =$$

$$[0.38]\,[-3.6] = [-1.368]$$

$$\delta^{(1)} = \left(\begin{bmatrix}1\\1\\1\end{bmatrix} - \begin{bmatrix}1\\0.6\\0.76\end{bmatrix}\begin{bmatrix}1\\0.6\\0.76\end{bmatrix}\right)\begin{bmatrix}0.2\\1\\-3\end{bmatrix}\begin{bmatrix}-1.368\\-1.368\end{bmatrix} =$$

$$\begin{bmatrix}0\\0.64\\0.4224\end{bmatrix}\begin{bmatrix}0.2\\1\\-3\end{bmatrix}\begin{bmatrix}-1.368\\-1.368\end{bmatrix} = \begin{bmatrix}0.64\\-1.2672\end{bmatrix}\begin{bmatrix}-1.368\\-1.368\end{bmatrix} =$$

$$\begin{bmatrix}-0.876\\1.734\end{bmatrix}$$

$$\frac{\partial e}{\partial W^{(1)}} = X^{(0)} \left(\delta^{(1)}\right)^T = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \begin{bmatrix} -0.876 & 1.734 \end{bmatrix} =$$

$$\begin{bmatrix} -0.876 & 1.734 \\ -1.752 & 3.468 \end{bmatrix}$$

$$\frac{\partial e}{\partial W^{(2)}} = X^{(1)} \left(\delta^{(2)}\right)^T = \begin{bmatrix} 1 \\ 0.6 \\ 0.76 \end{bmatrix} \begin{bmatrix} -1.368 \end{bmatrix} = \begin{bmatrix} -1.368 \\ -0.821 \\ -1.040 \end{bmatrix}$$

$$\frac{\partial e}{\partial W^{(3)}} = X^{(2)} \left(\delta^{(3)}\right)^T = \begin{bmatrix} 1 \\ -0.9 \end{bmatrix} \begin{bmatrix} -3.6 \end{bmatrix} = \begin{bmatrix} -3.6 \\ 3.24 \end{bmatrix}$$

3. Exercise 7.9

If we initialize all the weights to exactly zero, all $x^{(\ell)}$ and $\delta^{(\ell)}$ will be zero $\Rightarrow$ gradient will always be zero and the weights will not be updated

4. Exercise 7.10

2 hidden layer network with 10 hidden nodes in each layer

For layers $\ell = 1$ to L, weights in $W^{(\ell)}$ have $(d^{(\ell-1)}+1) \times d^{(\ell)}$ dimensions

$\Rightarrow$ the number of weights in the neural network is

$$Q = \sum_{\ell=1}^{L} d^{(\ell)}(d^{(\ell-1)} + 1)$$

In our case, $L = 3$, $d^{(1)} = d^{(2)} = 10$, $d^{(3)} = 1$ (output layer)

$d^{(0)} = d^{(0)}$ (we don't know the dimension of input)

$$\Rightarrow Q = 10 \cdot (d^{(0)}+1) + 10 \cdot (10+1) + 1(10+1) =$$

$$10d^{(0)} + 10 + 110 + 11 = 131 + 10d^{(0)}$$

2. Regularization and Validation

As multi-layer neural networks are very powerful, they have a tendency to overfit. The section provides some useful regularization techniques to fight it. First, weight decay used for linear models and its modified version – weight elimination regularizers were presented. Next, it was stated that early stopping can also be used as a form of regularization as in this way, a smaller hypothesis set is used. The rest part of the section discusses the methods to improve the gradient descent algorithm, which include variable learning rate gradient descent, steepest descent, and conjugate gradient descent. The last one has the best performance in terms of the in-sample error.

1. Summary of Example 7.2

This example verifies that using more iterations, we explore a larger hypothesis set and, therefore, ger a worse generalization and overfitting (E_{in} is decreasing, and d_{VC} is increasing). Hence, it may be better to stop early, constraining the hypothesis set. This way, early stopping is related to weight decay regularization as early stopping indirectly achieves smaller weights.

2. Exercise 7.12

From the hint, we can see that the wisdom that learning with more data is better applies to a fixed model ($\mathcal{H}, \mathcal{A}$). However, early stopping is model selection on a nested hypothesis sets $\mathcal{H}_1 \subset \mathcal{H}_2 \subset \cdots$ determined by $\mathcal{D}_{train}$ (as t^* increases, hypothesis set becomes larger). If the full data $\mathcal{D}$ is used, a completely different set of weights can be obtained (validation estimate of performance only holds for w_{t^*})

3. Explanations of Figures 7.2 and 7.3

Figure 7.2 shows the classification boundary for slightly noisy data obtained after more than 2 million iterations of gradient descent without regularization. Overfitting is clear in this case as the boundary is too complex. Then, weight decay regularization is applied to fight the overfitting, and as a result, a much better boundary is obtained. Figure 7.3 shows the results

for early stopping. From Figure 7.3 (a), it is seen that after iteration t^*, we have bad generalization and overfitting. Therefore, it is better to stop here, and Figure 7.3 (b) verifies that as the classification boundary with early stopping at t^* is similar to one with weight decay. The quantitative statistics demonstrates that E_{out} is even a little better for early stopping in comparison with weight decay.

4. The main differences among the following methods

Gradient descent – a simple method to minimize E_{in} that has problems converging. It takes a step of size η in the negative gradient direction. It is possible to improve the algorithm by using a variable learning rate. The method is also called batch gradient descent because the gradient is computed for the error on the whole data set before a weight update is done.

Stochastic gradient descent – a sequential version of gradient descent in which instead of considering the full batch gradient on all N training data points, we pick a random data point and consider only the error on that data point

Steepest descent (gradient descent + line search) – a method for which we do not need to pick a learning rate η arbitrarily but choose the optimal one that minimizes $E_{in}\big(w(t+1)\big)$. When the direction in which to move, $v(t)$, has been determined, we simply continue moving along the line $w(t) + \eta v(t)$ and stop when E_{in} is minimum.

Conjugate gradient descent – a method in which the line search direction is different from the negative gradient. When the new gradient and the previous line search direction are orthogonal, we have set one of the components of the gradient to zero. The conjugate gradient descent chooses the next direction $v(t\ +\ 1)$ so that the gradient along this direction will remain perpendicular to the previous search direction $v(t)$. The process is repeated until all components of the gradient are set to zero (a local minimum is achieved).

5. Discussion of Example 7.3

This example compares the performances of gradient descent, stochastic gradient descent, gradient descent with variable learning rate, and steepest descent in terms of the in-sample error at various points in the optimization. It is seen that stochastic gradient descent is quite effective and competitive as steepest descent starts to outperform it only after the optimization time of 50,000 sec. Figure 7.4 shows that it is hard to know when to stop minimizing because it is difficult to differentiate between a flat region and a true local minimum.

3. Deep Learning: Networks with Many Layers

Though any target function can be approximated with a single hidden layer with enough hidden units, this section shows that deep learning (networks with many layers) often more closely mimics human learning. This is demonstrated on the digit recognition problem. From this example, we understand that deep networks provide some human-like insight into the solution of a problem by constructing a higher-level representation from a low-level representation from the previous layers. As deep networks are hard to train and have a tendency to overfit, one may favor single hidden layer neural networks over them. However, the section presents a greedy deep learning algorithm that is the current best algorithm for digit recognition. Also, the section presents two approaches for deep learning: unsupervised auto-encoder in which hidden layers provide a hierarchical representation of inputs and supervised deep network in which a network is trained on targets. In the last example of the section, we can see that a deep network trained with the greedy deep learning algorithm for digit recognition performs really well in terms of out-of-sample error, giving 99.8% accuracy.

1. Exercise 7.17

Each person writes digits in his/her own way, therefore, the exercise is quite subjective. My opinion on it is presented below

2, 3, 7, 9 can contain φ_1

0, 3, 4, 6, 7, 8, 9 may contain φ_2 (if they are written in a digital format)

0, 2, 3, 6, 8, 9 may contain φ_3

0, 6, 7, 8, 9 can contain φ_4

4, 6, 8, 9 may contain φ_5

3, 6, 8 can contain φ_6

Additional basic shapes should not be too large or too small, and they should not overlap. Also, the collection of basic shapes should be able to construct the digits.

Exercise 7.18

Black ($x_{ij} = 1$) white ($x_{ij} = 0$)

a) Show that $\phi_k(x) = \tanh\left(w_0 + \sum_{ij} w_{ij} x_{ij}\right)$

If input image x has feature φ_k, $\varphi_k(x)$ should compute $+1$

If it does not, $\varphi_k(x)$ should compute -1

Since there are many black pixels in φ_k, the absolute value of the weighted sum of pixels will be large $\Rightarrow$ the input of tanh (its absolute value) will be large $\Rightarrow$ tanh will work as the hard threshold (output $+1$ or -1)

Also, bias w_0 is added to account for differences in basic shapes' representations in images of written digits

$$\Rightarrow \quad \varphi_k(x) = \tanh\left(w_0 + \sum_{ij} w_{ij} x_{ij}\right)$$

b) The inputs to the neural network node are pixels (matrix $[x_{ij}]$ of pixels)

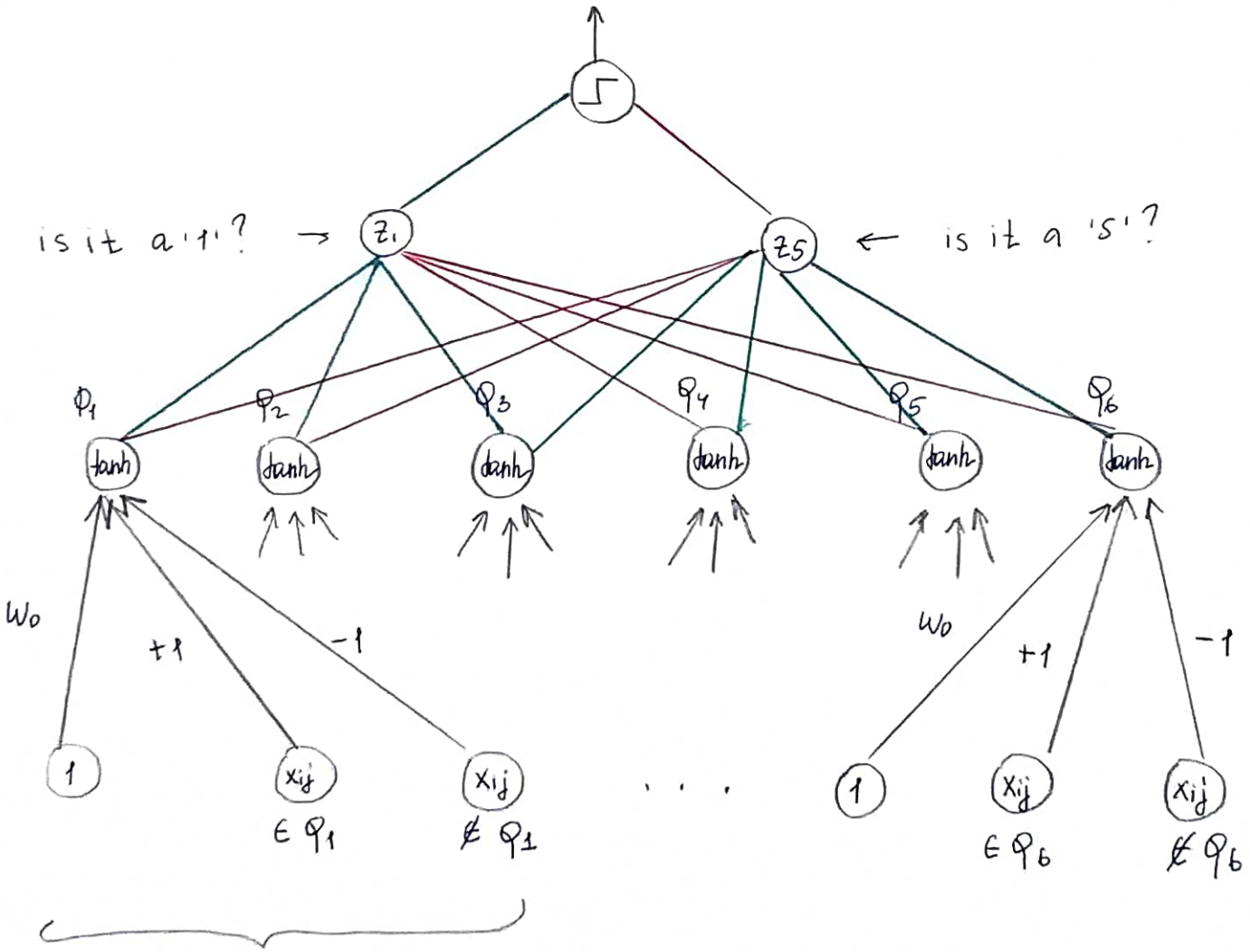
c) The weights of the pixels for those $x_{ij} \in \phi_k$ are chosen as $+1$ so that their weighted sum is positive

$\Rightarrow \varphi_k(x) = +1$

The weights of the pixels for those $x_{ij} \notin \varphi_k$ are chosen as -1 so that their weighted sum is negative $\Rightarrow \varphi_k(x) = -1$.

d) w_0 should be positive and the exact value should be chosen so that $\varphi_k(x) = +1$ if the input has feature φ_k but it is represented slighly differently in the image (i.e, the input of tanh must be positive) and $\varphi_k(x) = -1$ if the input doesn't have the feature at all (i.e, the input of tanh must be negative)

e)



should be repeated for other φ_k-s $(k = 2, 3, 4, 5)$

2. Explanation of Figure 7.6

Figure 7.6 presents a schematic of the greedy deep learning algorithm. The first three figures (a – c) illustrate the essence of the algorithm – to learn first layer weights, fix them, and move on to learn the second layer weights. The process continues by learning the third layer weights after fixing the weights of two previous layers, and so on. Figure 7.6 (d) shows that learned and fixed weights can be then used as a starting point to fine-tune the entire network.

3. Greedy Deep Learning algorithm

In this algorithm, we learn the first layer weights $W^{(1)}$ and fix them. The output of the first hidden layer is a nonlinear transformation of the inputs, which is used to train the second layer weights $W^{(2)}$ keeping the first layer weights fixed. This way, a single hidden layer neural network is trained, and we ignore the possibility that better first layer weights may exist if the second layer is considered. The process continues with the output from the previous layer used to learn the weight of the next layers, keeping weights from all previous layers fixed.

4. Exercise 7.19

For deep networks, symmetry and intensity can be extracted after several hidden layers as high-level features can be learnt from low-level features. Therefore, there is no need to generate them manually.

5. Discussion of Example 7.5

This example demonstrates that a deep network trained with the greedy deep learning algorithm for digit recognition performs really well in terms of out-of-sample error, giving 99.8% accuracy. In the example, supervised deep network and gradient descent iterations are used. Also, 6 hidden units learnt in the first hidden layer are presented. These features do not ideally represent human intuition; however, the result is automatic, purely data-driven, and gives impressive out-of-sample performance.