Artificial Neural Networks
Lecture Notes

Prof. Dr. Sen Cheng

Winter Semester 2019/20

9 Training multilayer networks

9.1 Multilayer perceptrons (MLP)

Multilayer perceptrons (MLPs) can solve problems that are not linearly separable. For instance, an MLP can solve the XOR problem (Fig. 1). The network consists of two perceptrons in the second layer that both assign the off-diagonal elements to class +1, but also misclassifies one of the diagonal elements as +1 (Fig. 1b).

$$w^1_{1.} = \begin{bmatrix} 1.5 & -1 & -1 \end{bmatrix} \tag{1a}$$

$$w^1_{2.} = \begin{bmatrix} -0.5 & 1 & 1 \end{bmatrix} \tag{1b}$$

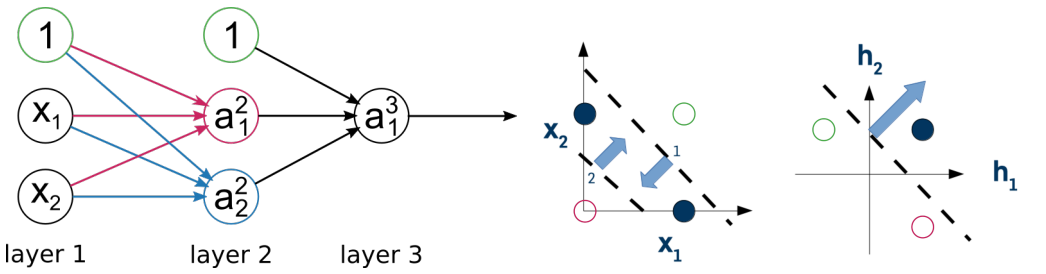
By adding an output perceptron in the third layer, we can combine the outputs of the two perceptrons in the second layer. Only if both perceptrons in the second layer return +1, the output perceptron should return +1, otherwise it should return -1 (Fig. 1c).

$$w^2_{1.} = \begin{bmatrix} -1 & 1 & 1 \end{bmatrix} \tag{1c}$$

Even though solutions to the XOR problem, such as the one above, were known since the 1940s, nobody knew how to train these networks until the 1970s and, even then, the solutions weren't widely publicized until Rumelhart et al. (1986). For pratical problems, it is not feasible to construct a solution by hand, as we did for the very simple XOR-function.

9.2 Backpropagation of errors

Data: Inputs and outputs are quite flexible and can be continuous, categorical or binary. Inputs are generally of much higher dimensionality than outputs. To simplify the math, we break with our previous notation here and use x and y without the indices to denote inputs and outputs, respectively. Nevertheless, x, y are implied to take on different values as we iterate through the data.



(a) 3-layer network.

(b) Activations of first layer.

(c) Activations of second layer.

Figure 1: A solution of XOR problem using MLP.

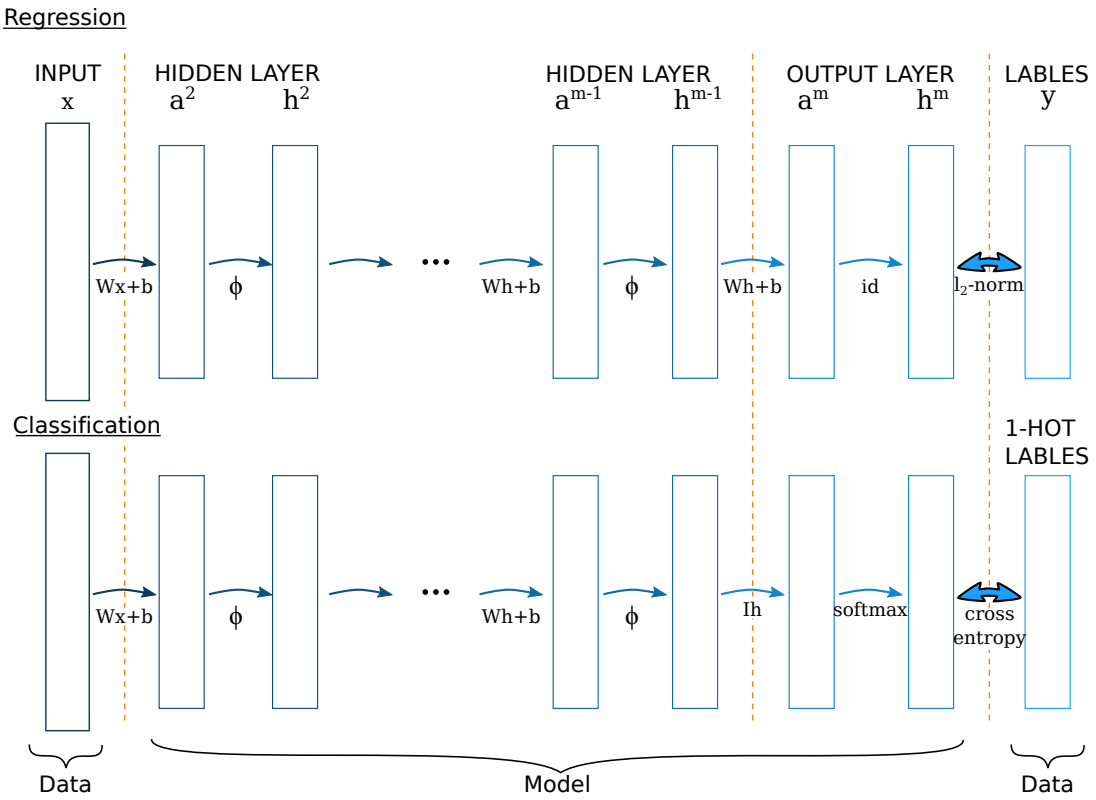


Figure 2: Multilayer networks for regression and classification problems. Note that the two components of each layer are shown separately: the weighted summation leading to a^l and the activation function leading to h^l.

There are several ways to represent categorical variables: string labels, numerical labels and *one-hot encoding*. String labels are difficult to handle in algorithms, numerical labels lead to unwanted potential correlations. So often the preferred method is one-hot encoding. The class membership in every class c_j is represented by one feature of x, i.e., $x_i \in \{0,1\}$. So, if the item is an element in class c_j:

$$x_i = \begin{cases} 1 & ,\text{if } i = j \\ 0 & ,\text{if } i \neq j \end{cases} \tag{2}$$

Model class: multilayer neural network (Fig. 2)

$$h_i^{l+1} = \phi^{l+1}\left(\sum_j w_{ij}^l h_j^l + b_i^l\right) = \phi^{l+1}\left(a_i^{l+1}\right) \tag{3}$$

where

$$a_i^{l+1} = \sum_j w_{ij}^l h_j^l + b_i^l \tag{4}$$

The activation functions can differ between the layers. In practice, they are the same for the hidden layers, and a different activation function is used in the last layer depending on the task. For regression, the identity function $\phi^m(a) = a$ is used (Fig. 2, top). For multiclass classification, the *softmax* function is used (Fig. 2, bottom).

$$\phi_i^m(a^m) = \frac{e^{a_i^m}}{\sum_{j=1}^n e^{a_j^m}} \tag{5}$$

In addition, the last layer does not sum over its inputs, instead it simply applies the activation function to each of the inputs, i.e., $a^m = h^{m-1}$. Since the outputs are bounded between 0 and 1, and sum to 1, the outputs can be interpreted as a probability distribution. If the last layer has n^m units that each represent a different class, $\phi_i^m(a^m)$ gives the

probability that the input x is a representative of class c_i. The class predicted by the network is simply the class for which the output value is the highest, i.e., $\hat{y} = \arg\max_i \phi_i^m(a)$.

The parameters of the model are all the weights and biases. The hyperparameters of the model include the number of layers m and the number of neurons n^l in each layer l, as well as the activation functions ϕ^l. Additional hyperparameters will be added by the optimization procedure.

Loss function: $L(\theta, x, y)$ depends on the task. l_2-norm for regression, cross entropy for classification. To simplify the math, we break with our previous notation here and use L to denote the loss for a single data pair x, y.

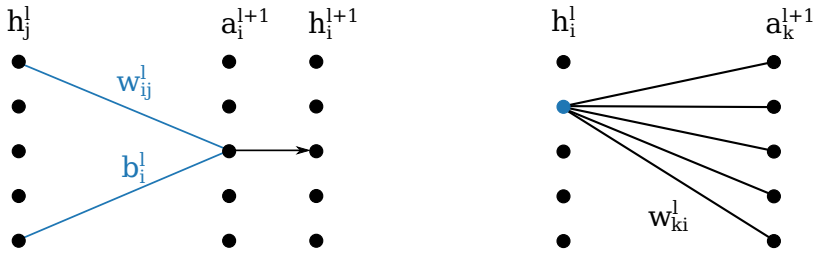


Figure 3: Application of the chainrule to derive the backpropagation algorithm.

Optimization: Gradient descent on the loss functions. We'll derive the algorithm in general for any differentiable loss function. To update the weights, we need the derivative of the loss function w.r.t. to the weights: $\frac{\partial L}{\partial w_{ij}^l}$. The weight w_{ij}^l affects the loss function only through the summed activity of the downstream unit a_i^{l+1} (Fig. 3, left). That's why, using the chain rule, the gradient w.r.t. the weights can be written as

$$\frac{\partial L}{\partial w_{ij}^l} = \underbrace{\frac{\partial L}{\partial a_i^{l+1}}}_{\delta_i^{l+1}} \underbrace{\frac{\partial a_i^{l+1}}{\partial w_{ij}^l}}_{h_j^l} = \delta_i^{l+1} h_j^l \tag{6a}$$

The gradient w.r.t. the biases is calculated similarly (Fig. 3, left):

$$\frac{\partial L}{\partial b_i^l} = \frac{\partial L}{\partial a_i^{l+1}} \underbrace{\frac{\partial a_i^{l+1}}{\partial b_i^l}}_{1} = \delta_i^{l+1} \tag{6b}$$

The *error* term δ_i^l is common to both gradients. It affects the loss function only through the activation h_i^l (Fig. 3, left).

$$\delta_i^l = \frac{\partial L}{\partial a_i^l} = \frac{\partial L}{\partial h_i^l} \underbrace{\frac{\partial h_i^l}{\partial a_i^l}}_{(\phi^l)'(a_i^l)} = \left(\phi^l\right)'\left(a_i^l\right) \frac{\partial L}{\partial h_i^l} \tag{6c}$$

At the output layer $l = m$, $y = h^m$, the loss function is defined in terms of the y's. If the loss function is differentiable w.r.t. y, then the gradient $\frac{\partial L}{\partial h_i^m}$ is known, and the error can be calculated directly:

$$\delta_i^m = (\phi^m)'(a_i^m) \frac{\partial L}{\partial h_i^m} \tag{6d}$$

For the hidden layers, it is more complicated, since their activations do not enter the loss function directly. But we can derive how the gradient $\frac{\partial L}{\partial h_i^m}$ depends on the errors of the next layer, and derive a *recursive equation*. The key is to note that the activation h_i^l affects all summed activity a_k^{l+1} in the next layer (Fig. 3, right)

$$\frac{\partial L}{\partial h_i^l} = \frac{\partial L(\ldots, a_k^{l+1}, \ldots)}{\partial h_i^l} = \sum_k \frac{\partial L}{\partial a_k^{l+1}} \underbrace{\frac{\partial a_k^{l+1}}{\partial h_i^l}}_{w_{ki}^l} = \sum_k \delta_i^{l+1} w_{ki}^l \tag{6e}$$

So, the recursive equation for the error is:

$$\delta_i^l = \left(\phi^l\right)' \left(a_i^l\right) \sum_k \delta_k^{l+1} w_{ki}^l \tag{6f}$$

To calculate all the errors, one starts in the output layer $l = m$, and computes the error according to Eq. 6d. Then one traverses the layers of the network in reverse order from $l = m-1$ to $l = 2$ and calculates the error iteratively according to Eq. 6f. Once the errors are known, the gradients can be computed.

Since the error terms are passed from the output layer back through the layers of the network, this algorithm is called *backward propagation of errors* or *backpropagation* for short – or simply *backprop*. Activations h_i^l are propagated forward through the network, and errors are passed backwards through the network.

9.2.1 Backpropagation algorithm

Algorithm 9.1 Backpropagation algorithm – stochastic gradient descent

Require: X, Y

1: Initialize weights w_{ij}^l
2: **repeat**
3: Randomly select a data pair (x, y)
4: *Forward pass*: calculate the network activations h_i^l
5: *Backward pass*: calculate the errors δ_i^l
6: Update all weights $w_{ij}^l \leftarrow w_{ij}^l - \eta \delta_i^{l+1} h_j^l$
7: **until** some stopping criterion

9.2.2 Vector/matrix notation and summary

The gradient for the bias term (Eq. 6b) is same as if we had a weight w_{i0}^l that connects a unit that outputs $h_0^l = 1$ in one layer to unit i in the next layer.
Forward pass

$$h^1 = \begin{bmatrix} 1 \\ x \end{bmatrix} \tag{7a}$$

$$a^2 = w^1 h^1 \tag{7b}$$

$$h^2 = \begin{bmatrix} 1 \\ \phi^2(a^2) \end{bmatrix} \tag{7c}$$

$$\vdots \tag{7d}$$

$$\tag{7e}$$

$$a^{m-1} = w^{m-2} h^{m-2} \tag{7f}$$

$$h^{m-1} = \begin{bmatrix} 1 \\ \phi^{m-1}(a^{m-1}) \end{bmatrix} \tag{7g}$$

$$a^m = w^{m-1} h^{m-1} \tag{7h}$$

$$h^m = \phi^m(a^m) \rightarrow \hat{y} \tag{7i}$$

Backward pass - errors:

$$\delta^m = (\phi^m)'(a^m) \odot \nabla L(h_m) \tag{8a}$$

$$\delta^{m-1} = \left(\phi^{m-1}\right)'\left(a^{m-1}\right) \odot \left[\left(w^{m-1}\right)^T \delta^m\right] \tag{8b}$$

$$\vdots \tag{8c}$$

$$\delta^2 = \left(\phi^2\right)'\left(a^2\right) \odot \left[\left(w^2\right)^T \delta^3\right] \tag{8d}$$

$$\tag{8e}$$

	function	**derivative**
logistic	$\sigma(a) = \dfrac{1}{1+e^{-a}}$	$\sigma(a)\left(1-\sigma(a)\right)$
hyperbolic tangent	$\tanh(a) = \dfrac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$	$\tanh'(a) = 1-\tanh^2(a)$
ReLU	$\phi(a) = \begin{cases} 0 & a \leq 0 \\ a & a > 0 \end{cases}$	$\phi'(a) = \begin{cases} 0 & a \leq 0 \\ 1 & a > 0 \end{cases}$
leaky ReLU	$\max\left(\beta a, a\right); 0 < \beta < 1$	$\phi'(a) = \begin{cases} \beta & a \leq 0 \\ 1 & a > 0 \end{cases}$
ELU	$\phi(a) = \begin{cases} \beta\left(e^{a}-1\right) & a \leq 0 \\ a & a > 0 \end{cases}$	$\phi'(a) = \begin{cases} \phi(a)+\beta & a \leq 0 \\ 1 & a > 0 \end{cases}$
maxout	$\phi(a) = \max_i\left(a_i\right)$	$\dfrac{\partial\phi(a)}{\partial a_i} = \delta_{ik}$
softmax	$\phi_i(a) = \dfrac{e^{a_i}}{\sum_{j=1}^{n} e^{a_j}}$	$\dfrac{\partial\phi_i(a)}{\partial a_j} = \phi_i(a)\left(\delta_{ij}-\phi_j(a)\right)$

Table 1: Common activation functions and their derivatives. δ_{ij} is the Kronecker delta function. In the derivative of the Maxout activation function, k is the index of the largest element, i.e., $k = \arg\max_i x_i$.

where we have used the Hadamard product $\odot$ to multiply the two vectors component-wise.

Backward pass - weight updates:

$$\Delta w^{m-1} = -\eta\delta^{m}\left(h^{m-1}\right)^{T} \tag{8f}$$

$$\vdots \tag{8g}$$

$$\Delta w^{1} = -\eta\delta^{2}\left(h^{1}\right)^{T} \tag{8h}$$

If using stochastic gradient descent (SGD), the forward and backward pass are iterated for each data pair, and the weights are updated for each data pair. If using batch gradient descent (BGD), the weight updates are summed for the entire data set before the weights are changed.

$$w^{l} \leftarrow w^{l} + \Delta w^{l} \tag{9}$$

The difference between the two is that in BGD the activations and errors are calculated with the same network parameters for all data pairs, whereas in the former case the network parameters are different each time because they are constantly updated.

9.2.3 Activation function

The backpropagation algorithm derived above works for any loss function as long as it is differentiable. The logistic function $\sigma(t)$ was popular for a long time because its derivative is particularly simple $\sigma' = \sigma(1-\sigma)$. So, the activations calculated during the forward pass $h^m = \sigma(a^m)$ can be saved and used to compute the errors. However, many other activation functions have been, and are being, used (Table 1). Depending on the application, they all have advantages and disadvantages. For instance, sigmoid activation functions are based on the exponential function, which is computationally expensive. More severely, they suffer from the vanishing gradient problem (see below).

9.3 Vanishing/Exploding gradient problems

As the recursive algorithm Eq. 6f propagates the errors down the layers, the error tends to become smaller and smaller. If the network has many layers, the *gradient vanishes*. As a result, the weights in the lower layers are not changed

activation function	uniform distribution $[-r, r]$	normal distribution	name
tanh	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$	Xavier
logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$	Xavier
ReLU	$r = \sqrt{\frac{6}{n_{\text{inputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}}}}$	He et al. (2015)

Table 2: Different weight initialization schemes for different activation function.

and learning does not converge to a good solution. In some other cases, the errors might increase with every iteration through the layers. As a result, the *gradients explode* and the algorithm diverges. Both scenarios are undesirable.

The source of this problem is a combination of using the logistic activation function and initialzing weights with random normal values with zero mean and standard deviation of 1 (Glorot & Bengio, 2010). Activation keeps increasing with every layer. The logistic function saturates for very large negative or positive values and, therefore, the derivative is nearly zero. Once the network reaches this point, the weights cannot be reduced any more.

9.3.1 Xavier and He initialization

One solution is to make sure that the activations and the gradients do not die out or explode when passing through the network layers. This could be achieved by making sure that the output magnitude is roughly equal to the input magnitude, which in turn can be achieved by carefully initializing the weights. When using the logistic activation function, Glorot & Bengio (2010) suggested to initialize weights randomly drawn from a:

1. normal distribution with zero mean and standard deviation $\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$, or
2. uniform distribution between $\pm\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$

The parameters n_{inputs} and n_{outputs} are also known as *fan-in* and *fan-out* factors. This method is named after the first name of the first author *Xavier initialization*. The optimal weight initialization depends on the activation function (Table 2).

9.3.2 Nonsaturating activation functions

The ReLu function does not suffer as much from vanishing gradients, because it doesn't saturate for positive input values. A disadvatage is that the ReLU function is not differentiable at zero. In practice, the derivative is simply set to zero, i.e., $\phi'(0) = 0$, which works remarkably well in practical applications. It also generates many zero values in the activations, leading to *representational sparsity*. However, the ReLu function saturates for negative inputs, in which case both the activation and the derivative are zero. Because of the zero derivative, the incoming weights will not be updated and therefore the inputs to the unit are not going to become stronger. If this happens for all inputs, the unit is effectively removed from the network. This effect is called the *dying ReLU problem*.

To avoid both the vanishing gradient and the dying ReLU problems, an alternative is to use activation functions that do not saturate. The leaky ReLU or the ELU activation function use the same activation function when the input is positive. However, for negative inputs, they take on small negative values and the derivatives are non-zero. For $\beta = 1$, ELU is differentiable at zero and therefore avoids large jumps in the gradient around zero. The disadvantage of ELU is that is uses the costly exponential function.

9.3.3 Gradient clipping

Using careful weight initialization avoids the vanishing/exploding gradient problem at the beginning of the training, however, it might reappear later during training. To avoid these issues during training one can use *gradient clipping*.

In this method, the gradients are limited to a certain range $[-g, g]$. If any gradient exceeds g, the gradient is set to g. Likewise, if the gradient is smaller than $-g$, it is set to $-g$.

References

Glorot, X. & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9, (pp. 249–256)., Sardinia, Italy. Proceedings of Machine Learning Research.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *ArXiv150201852 Cs*.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536.