Lectures on Machine Learning (Fall 2017)

Hyeong In Choi Seoul National University

Lecture 13: Feedforward Neural Networks (Draft: version 0.9.1)

Topics to be covered:

- Brief history of neural network
- XOR problem and neural network with a hidden layer
- Universal approximation theorem
- Multilayer neural network
  - Formalism
  - Errors
  - Backpropagation algorithm
  - Backpropagation in matrix form
- Mathematical supplement: automatic differentiation

1 Brief history of neural networks

Neural network has a very interesting history. Its popularity comes in ebb and flow like a drama. Neural network first got started when F. Rosenblatt proposed the so-called Perceptron [4, 5]. Architecturally it is just like a neural network with no hidden layer, although its learning rule uses the

so-called perceptron learning algorithm, which is different from the present-day backpropagation algorithm. It nonetheless is the first one that can learn from data without explicit programming by human experts. This feature created a lot of excitement, and with the perceptron convergence theorem, perceptron seem to be able to meet many interesting challenges. However, mathematically what Perceptron does is finding *linear* classification boundary.

This fact drew attention from Minsky and Papert, who wrote a book criticizing that Perceptron cannot even distinguish between the very simple XOR configurations of four data points on a plane [3]. This book almost instantly killed the then-flourishing many perceptron research projects, and this marked the closing of the first wave of neural network.

Since then, MIT (and others) led the artificial intelligence research along the path of logical analysis and rule-based programming. They contrived many interesting methods and approaches to embody human intelligence through the rule-like programming paradigm. Although this approach had many successes in various areas where logical thinking is the key, it failed pretty badly in handling the perception problem, which is what Perceptron was intended to solve. This lack of *common sense* became the glaring deficiency of this type of artificial intelligence methods, and it is the reason why logic-based rule-like artificial intelligence gradually withered away.

The second wave of neural network came when the PDP group led by Rumelhart, McClelland and others discovered the magic of adding a hidden layer to the neural network [6]. In particular, they showed that the XOR problem, which had vexed Perceptron so much, can be easily overcome. (See Section 2.2 below.) This and many other interesting applications rekindled interest on neural networks. Many other models of neural networks were proposed and many interesting problems solved. Indeed it seemed that the secret of human intelligence was about to be revealed. However, it soon dawned on people that complex neural networks with many hidden layers are very hard to train. They frequently overfit and, as a consequence, a small change in the dataset leads to drastic changes in the neural network itself; thereby earning the reputation of being "hard-to-train". So around the mid-1990s and on, the PDP-style neural network projects were mostly abandoned, and people turned to other machine learning methods like Support Vector Machine and kernel methods; ensemble methods like boosting and random forests; and so on.

Such was the state of affairs until suddenly G. Hinton and his colleagues

came bursting onto the scene with fresh new ideas of pre-training neural networks in the mid-2000s. (See for example [2] and our subsequent lectures.) With the drastic increase in computational capability with the use of GPU machines, people began to create very complex neural networks with tens of hidden layers and train them reasonably well, as long as there are enough data to feed to the network.

Currently we are witnessing the third wave of the neural network under the banner of **deep learning** coupled with *big data*. Its success is impressive. Overnight, it has almost completely replaced voice recognition technology that has a long history of its own; vision research is now mostly based on deep learning; and with the use deep learning, complicated natural language problems like machine translation are at a different level of competency. Many commercial applications developed with deep learning are now routinely in use.

However, deep learning is not a panacea. Although it is capable of solving many, if not all, complex perception problems, it is not as adept at reasoning tasks. Also, its black-box nature makes it harder for humans to accept the outcome if it conflicts with one's expert knowledge or judgment. Another aspect of deep learning is that it requires a huge amount of data, which contrasts with the way humans learn. All these objections and questions will be presented as some of the central themes in the next phase of artificial intelligence research, and it is interesting to see how one can mesh deep learning with this new coming trend.

2 Neural network with hidden layer

2.1 Neural network formalism of logistic regression

In Lecture 3, we showed that logistic regression can be recast in neural network formalism. It goes as follows: Let $x = [x_1, \cdots, x_d]^T$ be an input (vector). Define

$$\begin{aligned} z_k &= w_k \cdot x + b_k \\ &= \sum_{j=1}^{d} w_{kj} x_j + b_k \end{aligned} \tag{1}$$

and let

$$z = [z_1, \cdots, z_K]^T$$

as a K dimensional column vector. Then in matrix notation we have

$$z = Wx + b.$$

The matrix W and the vector b are to be estimated with data.

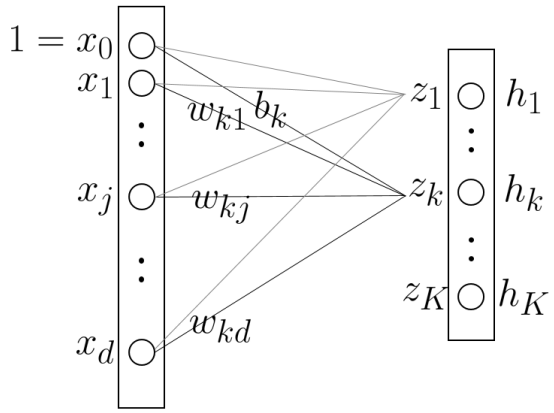

Figure 1: Neural network view of logistic regression

The output is the softmax function:

$$\text{softmax}(z_1, \cdots, z_K) = \left(\frac{e^{z_1}}{\sum_{j=1}^K e^{z_j}}, \cdots, \frac{e^{z_K}}{\sum_{j=1}^K e^{z_j}} \right),$$

and its k-th element of the output is denoted by

$$\text{softmax}_k(z_1, \cdots, z_K) = \frac{e^{z_k}}{\sum_{j=1}^K e^{z_j}},$$

In vector notation, we write

$$\text{softmax}(z) = \text{softmax}(z_1, \cdots, z_K).$$

This softmax defines the probability $P(y = k \,|\, x)$ as the output $h = [h_1, \cdots, h_K]^T$, which is the conditional probability given by:

$$h_k = h_k(x) = P(Y = k \,|\, X = x) = \text{softmax}_k(z_1, \cdots, z_K) = \frac{e^{z_k}}{\sum_{j=1}^K e^{z_j}}. \quad (2)$$

Its neural network formalism is as in Figure 1. The circles in the left box represent the input variables $x_1, \cdots, x_d$ and the ones in the right the response variables $h_1, \cdots, h_K$. The internal state of the k-th output neuron is the variable z_k whose value is as described in (1). In neural network parlance, z_k is gotten by summing over all j the multiples of w_{kj} and x_j, and then adding the value b_k, called the bias. Once the values of $z_1, \cdots, z_K$ are given, the output $h_1, \cdots, h_K$ can be found by applying the softmax function (2).

2.2 XOR problem and neural network with hidden layer

Let $\mathfrak{D} = \{(0,0), (0,1), (1,0), (1,1)\}$ be a dataset consisting of 4 points in $\mathbb{R}^2$. They belong to one of the two classes as shown in Figure 2. Note that there is no line that separates these two classes. Let us see how this problem

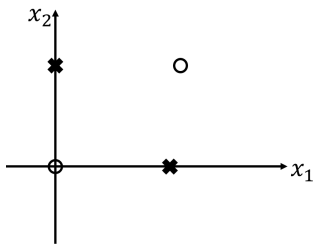

Figure 2: XOR Problem

can be solved if we add a hidden layer. Treating x_1 and x_2 as Boolean variables, we can write

$$\mathrm{XOR}(x_1, x_2) = x_1\bar{x_2} + \bar{x_1}x_2.$$

Thus if we give label $x_1\bar{x_2}$ to (x_1, x_2) as in Figure 3, the line $x_1 - x_2 - 1/2 = 0$ becomes the separating line. Note that if $a >> 0$, $\sigma(at)$ becomes very close to the step function as Figure 4 shows. Now let $z_1 = a(x_1 - x_2 - 1/2)$ and let $h_1 = \sigma(z_1)$. Then the value of h_1 becomes (very close to) 1 at the bottom right side of the line $x_1 - x_2 - 1/2 = 0$, and 0 at the top left side of line $x_1 - x_2 - 1/2 = 0$. This situation is depicted in Figure 3. In neural network notation, this variable relation is drawn as in Figure 5.

x_1	x_2	$x_1\bar{x}_2$
1	1	0
1	0	1
0	1	0
0	0	0

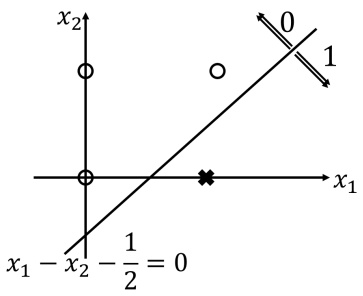

Figure 3: Value of h_1 and the separating line

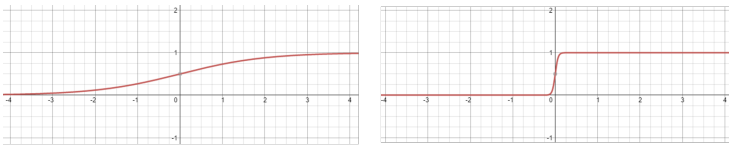

Figure 4: For large a, sigmoid becomes very close to step function

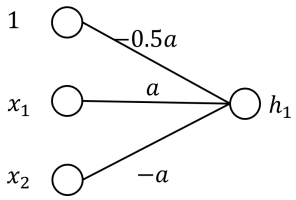

Figure 5: Neural network for h_1

x_1	x_2	$\bar{x}_1x_2$
1	1	0
1	0	0
0	1	1
0	0	0

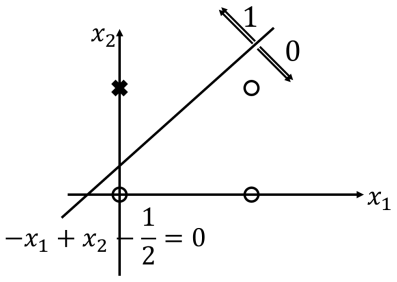

Figure 6: Value of h_2 and the separating line

Similarly, for the label $\bar{x_1}x_2$, the Boolean value of $\bar{x_1}x_2$ and the separating line $-x_1+x_2-1/2=0$ is shown in Figure 6. Now let $z_2 = a(-x_1+x_2-1/2)$ and let $h_2 = \sigma(z_2)$. Then by the same argument, the value of h_2 becomes (very close to) 0 at the bottom right side of the line $-x_1+x_2-1/2=0$, and 1 at the top left side of line $-x_1+x_2-1/2=0$. This situation is depicted in Figure 6. In neural network notation, this variable relation is drawn as in Figure 7. To combine h_1 and h_2, let $h_3 = \text{XOR}(x_1, x_2) = x_1\bar{x_2} + \bar{x_1}x_2$.

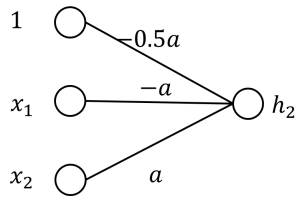

Figure 7: Neural network for h_2

The values of all the variables are shown in Figure 8. Note that the range

x_1	x_2	$h_1 = x_1\bar{x_2}$	$h_2 = \bar{x_1}x_2$	$h_3 = x_1\bar{x_2} + \bar{x_1}x_2$
1	1	0	0	0
1	0	1	0	1
0	1	0	1	1
0	0	0	0	0

Figure 8: Value of variables

of values (h_1, h_2) can take becomes restricted, i.e. they are $(0,0)$, $(0,1)$, and $(1,0)$. However $(1,1)$ does not now show up. This situation is depicted in Figure 9 and thus $h_1+h_2-1/2$ is a separating line for these data. Define $z_3 = b(h_1+h_2-\frac{1}{2})$, for large $b >> 0$, and $h_3 = \sigma(z_3)$. And the value of h_3 on the plane is also shown in Figure 9. If we combine everything we have done in this subsection, it can be written in the neural network as in Figure 10. Clearly this neural network with one hidden layer separates the original XOR data.

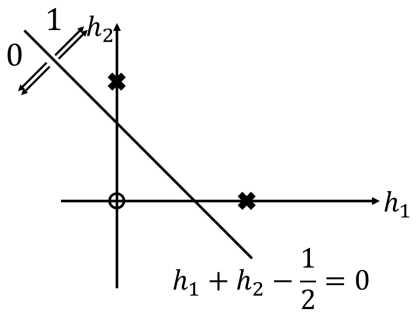

Figure 9: Value of h_3 and the separating line

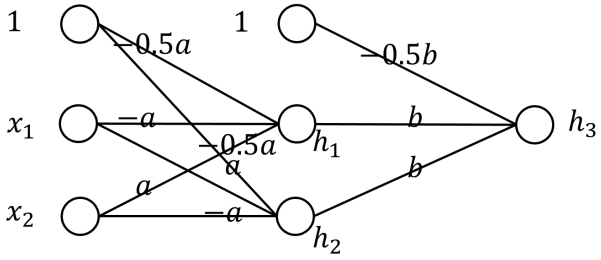

Figure 10: Final neural network

2.3 Universal approximation theorem

Let us rewrite part of the above neural network construction as in Figure 11. If we take the OR Boolean operation of Boolean variables h_1 and h_2, we

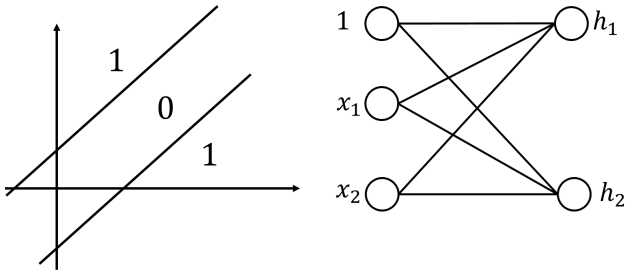

Figure 11: Neural network for h_1 and h_2

have the table in Figure 12. Since the label of the OR operation is 1 except at $(0, 0)$, where the label is 1, it is easy to find a separating line for this OR data. Similarly, we can construct another neural network as in Figure

h_1	h_2	$h_1 \vee h_2$
1	1	1
1	0	1
0	1	1
0	0	0

Figure 12: Boolean OR of h_1 and h_2

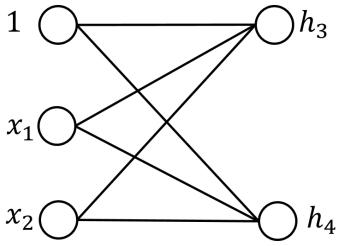

Figure 13: Neural network for h_3 and h_4

13. Its OR value on the plane is depicted in Figure 14. If we combine all

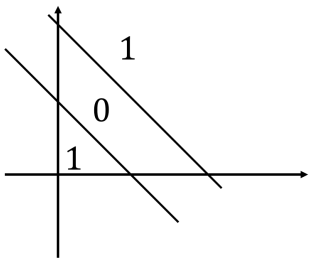

Figure 14: OR value of h_3 and h_4

four variables h_1, h_2, h_3 and h_4 by OR operation, its values are shown in the table in Figure 15. Note that all points (h_1, h_2, h_3, h_4) in $\mathbb{R}^4$ have the label 1, except $(0, 0, 0, 0)$ which has label 0. It is very easy to separate this with a hyperplane in $\mathbb{R}^4$ so that the neural network in Figure 16 separates all points in the table in Figure 15 with the h_5 label. Its neural network is depiected in Figure 16. The value of h_5 as a function of x_1 and x_2 is shown in Figure

h_1	h_2	h_3	h_4	$h_5 = h_1 \curlyvee h_2 \curlyvee h_3 \curlyvee h_4$
1	1	1	1	1
1	1	1	0	1
⋮	⋮	⋮	⋮	⋮
0	0	0	1	1
0	0	0	0	0

Figure 15: Combined OR value table of h_1, h_2, h_3 and h_4

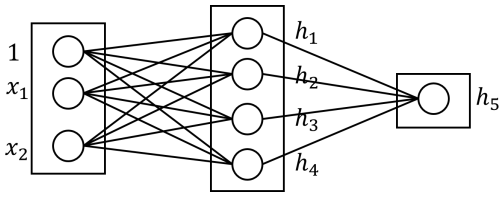

Figure 16: Neural network for OR of h_1, h_2, h_3 and h_4

17.

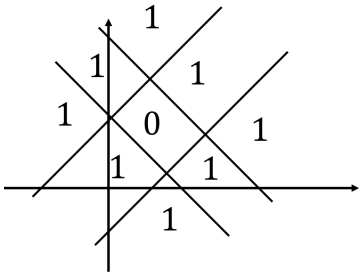

Figure 17: Value of h_5

Continuing this way, one can construct any approximate bump function as an output of a neural network with one hidden layer. Furthermore combining these bump functions, one can approximate any continuous function. Namely, a neural network with one hidden layer can do any task, at least in principle. This heuristic argument can be made rigorous using a Stone-Weienstrass theorem-type argument to get the following theorem.

Theorem 1. *(Cybenko-Hornik-Funabashi Theorem) Let $K = [0,1]^d$ be a d-dimensional hypercube in $\mathbb{R}^d$. Then the sum of the form*

$$f(x) = \sum_i c_i \text{sigmoid}(\text{b}_\text{i} + \sum_{\text{j}=1}^{\text{d}} \text{w}_\text{ij}\text{x}_\text{j})$$

can approximate any continuous function on K to any degree of accuracy.

Although this theorem is stated for the hypercube, it actually is valid on any compact set K.

3 Multilayer neural network

Logistic regression is an example of the simplest neural network consisting only of the input and output layers. However, the solution to the XOR problem with a hidden layer presented in the previous section shows the advantage of having a hidden layer. In general, a neural network with many hidden layers is called a **multilayer neural network** or **multilayer perceptron**.

3.1 Formalism of multilayer neural network

In Figure 18 is depicted the layer structure of a multilayer neural network. (Connection links are not shown.) The left-most group of neurons, called Layer 0, is the input layer. The ℓ-th layer to the right of the input layer is called Layer ℓ, and the right-most layer, Layer L, is called the output layer. (So there are $L-1$ hidden layers here.) The boxes enclosing the neurons in each layer there only for the convenience of communication. They are sometimes drawn and sometimes not.

Figure 19 shows Layers $\ell - 1$ and ℓ and the connection links between them. Moreover, every neuron in Layer $\ell - 1$ is connected to every neuron in Layer ℓ, but there is no connection between neurons in the same layer. The pre-activation value of (input to) neuron i in Layer ℓ is z_i^ℓ, which is gotten as a weighted sum of the previous layer's outputs by

$$z_i^\ell = \sum_j w_{ij}^\ell h_j^{\ell-1} + b_i^\ell, \tag{3}$$

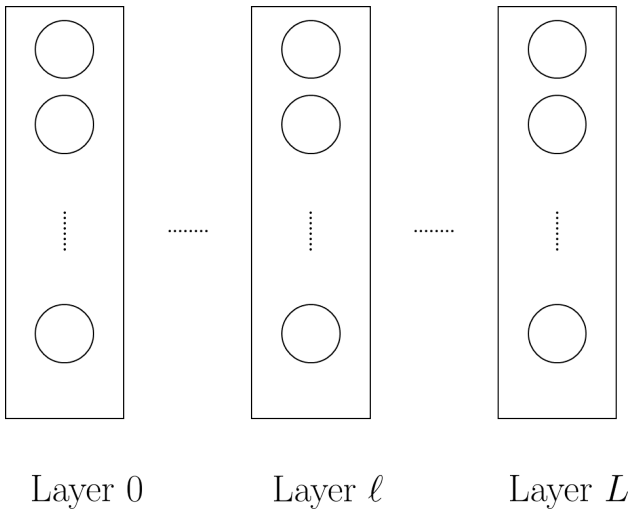

Figure 18: MLP Layers

for $\ell = 1, \cdots, L$. In vector-matrix notation, we have

$$z^\ell = W^\ell h^{\ell-1} + b^\ell. \tag{4}$$

(Here, the superscript denotes the layer number.)

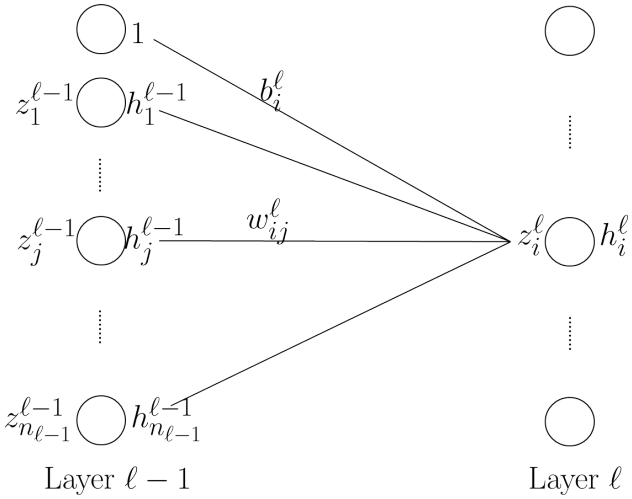

Figure 19: Connection weights

The activation value of (output of) neuron i in Layer ℓ is h_i^ℓ. The pre-activation and activation values are related by the activation function $\varphi(t)$

so that

$$h_i^\ell = \varphi_\ell(z_i^\ell), \tag{5}$$

for $\ell = 1, \cdots, L-1$. Some of the most popular activation functions are

$$\begin{aligned} \sigma(t) &= \frac{1}{1+e^{-t}}, && \text{sigmoid function} \\ \mathrm{ReLU}(t) &= \max(0,t), && \text{rectified linear unit} \\ \tanh(t) &= \frac{e^t - e^{-t}}{e^t + e^{-t}}, && \text{hyperbolic tangent function.} \end{aligned}$$

For $\ell = 0$, h_j^0 is set to be the input x_j, and we do not use z_j^0. For the output layer, i.e. $\ell = L$, the form of output h_i^L changes depending on the problem. For regression, the output is as usual

$$h_i^L = \varphi_L(z_i^L).$$

However, for classification, it is the softmax function of the pre-activation values z_i^L. Thus

$$h_k^L = \mathrm{softmax}_k(z_1, \cdots, z_K) = \frac{e^{z_k^L}}{\sum_{j=1}^K e^{z_j^L}}.$$

In this case, the output has the usual probabilistic interpretation

$$h_k^L = h_k^L(x) = P(Y = k \,|\, X = x).$$

3.2 Errors

The error of classification problems is essentially the same as that of logistic regression. As before, the formalism goes as follows. For a generic input-output pair (x, y), where $y \in \{1, \cdots, K\}$, use the one-hot encoding to define

$$y_k = \mathbb{I}(y = k).$$

So y can be identified with $(y_1, \cdots, y_K)$ in which $y_k \in \{0, 1\}$ and $y_1 + \cdots + y_K = 1$. Then the cross entropy error is

$$E = -\sum_k y_k \log h_k^L(x) = -\sum_k y_k \log P(Y = k \,|\, X = x).$$

Let $\mathfrak{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^N$ be a given dataset. For each data point $(x^{(i)}, y^{(i)})$, its cross entropy error is

$$E = -\sum_k y_k^{(i)} \log h_k^L(x^{(i)}) = -\sum_k y_k^{(i)} \log P(Y = k \,|\, X = x^{(i)}),$$

where $y_k^{(i)} = \mathbb{I}(y^{(i)} = k)$. For a mini-batch $\{(x^{(i)}, y^{(i)})\}_{i=1}^B$, its cross entropy error is

$$E = -\sum_{i=1}^B \sum_k y_k^{(i)} \log h_k^L(x^{(i)}) = -\sum_{i=1}^B \sum_k y_k^{(i)} \log P(Y = k \,|\, X = x^{(i)}),$$

The training of a neural network is to find *good* parameters w_{ij}^ℓ and b_i^ℓ to minimize the error, which is the topic we will be covering in the next subsection and subsequent lecture on *Training Deeping Neural Network*.

For the regression problem, one usually uses the L^2 error so that

$$E = -\sum_{i=1}^B |y^{(i)} - h^L(x^{(i)})|^2,$$

for the mini-batch $\{(x^{(i)}, y^{(i)})\}_{i=1}^B$, for instance. However, other forms of error can also be used. For instance, the L^1 error $E = -\sum_{i=1}^B |y^{(i)} - h^L(x^{(i)})|$ can be used, and as we have seen in Lecture 12 on Gradient Boost, errors like Huber error can also be used.

Neural networks are prone to overfitting. So to alleviate the problem, one usually resorts to regularization, which is to add the regularizing term $\Omega(\theta)$ to the error terms above. (Here, θ is a generic notation for all parameters like w_{ij}^ℓ and b_i^ℓ.) Typically, $\Omega(\theta)$ is of the form $\lambda|\theta|^2 = \lambda(\sum |w_{ij}^\ell|^2 + \sum |b_i^\ell|^2)$, $\mu|\theta| = \mu(\sum |w_{ij}^\ell| + \sum |b_i^\ell|)$, or $\lambda|\theta|^2 + \mu|\theta| = \lambda(\sum |w_{ij}^\ell|^2 + \sum |b_i^\ell|^2) + \mu(\sum |w_{ij}^\ell| + \sum |b_i^\ell|)$. However, many other forms can be used.

For classification, the typical error term with regularizer becomes

$$E = -\sum_{i=1}^B \sum_k y_k^{(i)} \log h_k^L(x^{(i)}) + \Omega(\theta) = -\sum_{i=1}^B \sum_k y_k^{(i)} \log P(Y = k \,|\, X = x^{(i)}) + \Omega(\theta),$$

and for regression, it is of the form

$$E = -\sum_{i=1}^B |y^{(i)} - h^L(x^{(i)})|^2 + \Omega(\theta) = -\sum_{i=1}^B \sum_{j=1}^d |y_j^{(i)} - h_j^L(x^{(i)})|^2 + \Omega(\theta).$$

3.3 Backpropagation algorithm

Training of a neural network, in a nutshell, is a gradient descent algorithm which is basically of the form:

$$\begin{aligned} \Delta w_{ij}^{\ell} &= -\lambda \frac{\partial E}{\partial w_{ij}^{\ell}} \\ \Delta b_{i}^{\ell} &= -\lambda \frac{\partial E}{\partial b_{i}^{\ell}}, \end{aligned}$$

where

$$\begin{aligned} \Delta w_{ij}^{\ell} &= w_{ij}^{\ell}(\text{new}) - w_{ij}^{\ell}(\text{old}) \\ \Delta b_{i}^{\ell} &= b_{i}^{\ell}(\text{new}) - b_{i}^{\ell}(\text{old}). \end{aligned}$$

In fact, deep learning training methods are not this simplistic, but all of them are variations of one form or another of this basic idea of gradient descent. For more details, the reader is referred to the forthcoming lecture on "Training Deep Neural Network."

So to do the training it is necessary to compute $\frac{\partial E}{\partial w_{ij}^{\ell}}$ and $\frac{\partial E}{\partial b_{i}^{\ell}}$ efficiently.

At the output layer, Layer L, it is easy to determine $\frac{\partial E}{\partial h_{i}^{L}}$, as we know the expression of the error E. By the chain rule, we can compute

$$\frac{\partial E}{\partial z_{j}^{L}} = \sum_{i} \frac{\partial h_{i}^{L}}{\partial z_{j}^{L}} \frac{\partial E}{\partial h_{i}^{L}}. \tag{6}$$

The variable dependencies are depicted in Figure 20. From this we can easily

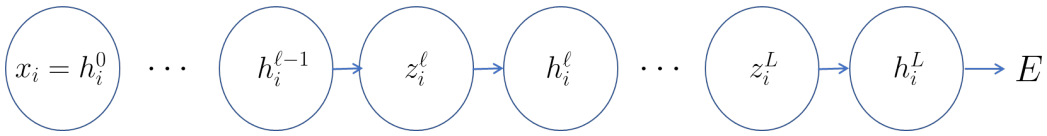

Figure 20: Variable dependency

write by the chain rule that for $\ell = 1, \cdots, L-1$,

$$\frac{\partial E}{\partial z_{i}^{\ell}} = \frac{dh_{i}^{\ell}}{dz_{i}^{\ell}} \frac{\partial E}{\partial h_{i}^{\ell}},$$

where

$$\frac{dh_i^\ell}{dz_i^\ell} = \begin{cases} h_i^\ell(1-h_i^\ell) & \text{if } \varphi_\ell \text{ is } \sigma \\ \mathbb{I}(z_i^\ell \geq 0) & \text{if } \varphi_\ell \text{ is ReLU} \\ \operatorname{sech}^2 z_i^\ell & \text{if } \varphi_\ell \text{ is } \tanh z \end{cases}$$

Applying the chain rule again, we get

$$\frac{\partial E}{\partial h_j^{\ell-1}} = \sum_i \frac{\partial z_i^\ell}{\partial h_j^{\ell-1}} \frac{\partial E}{\partial z_i^\ell}.$$

Using (3), we have

$$\begin{aligned} \frac{\partial z_i^\ell}{\partial h_j^{\ell-1}} &= \omega_{ij}^\ell \\ \frac{\partial z_i^\ell}{\partial \omega_{ij}^\ell} &= h_j^{\ell-1}. \end{aligned}$$

Therefore, we get

$$\frac{\partial E}{\partial h_j^{\ell-1}} = \sum_i \omega_{ij}^\ell \frac{\partial E}{\partial z_i^\ell}, \tag{7}$$

and

$$\frac{\partial E}{\partial \omega_{ij}^\ell} = \frac{\partial z_i^\ell}{\partial \omega_{ij}^\ell} \frac{\partial E}{\partial z_i^\ell} = h_j^{\ell-1} \frac{\partial E}{\partial z_i^\ell}. \tag{8}$$

Note that (7) says that the error $\dfrac{\partial z_i^\ell}{\partial h_j^{\ell-1}}$ is the weighted sum, with weights ω_{ij}^ℓ, of errors $\dfrac{\partial E}{\partial z_i^\ell}$, which are in the next layer. This means the error propagates backward from layer ℓ to layer $\ell - 1$, hence the name *backpropagation*.

Let us now derive similar formulas with respect to bias, i.e. b_i^ℓ. Note from (3),

$$\frac{\partial z_i^\ell}{\partial b_i^\ell} = 1.$$

Thus we have

$$\frac{\partial E}{\partial b_i^\ell} = \frac{\partial z_i^\ell}{\partial b_i^\ell} \frac{\partial E}{\partial z_i^\ell} = \frac{\partial E}{\partial z_i^\ell} \tag{9}$$

The whole thing is summed up as the following summary.

Summary

- By (3) and (5), the data flows forward, i.e. from Layer $\ell - 1$ to Layer ℓ, hence the name *feedforward network*
- By (6) and (7), the error *derivative*, not the actual error, can be computed backward, i.e.

$$\frac{\partial E}{\partial z_i^{\ell-1}} \overset{by(6)}{\leftarrow} \frac{\partial E}{\partial h_i^{\ell-1}} \overset{by(7)}{\leftarrow} \frac{\partial E}{\partial z_i^{\ell}} \overset{by(6)}{\leftarrow} \frac{\partial E}{\partial h_i^{\ell}} \leftarrow,$$

  hence the name *backpropagation*.
- Equations (8) and (9) are the basic equations to be used for the gradient descent algorithm for learning.

3.4 Backpropagation in matrix form

Recall that the variable dependency in Figure 20. We want to write the chain rule in matrix form. Using the vector notation, we denote by $\left(\frac{\partial E}{\partial z^L}\right)$ the column vector whose j-th entry is $\frac{\partial E}{\partial z_j^L}$. Similarly let

$$\left(\frac{\partial h^\ell}{\partial z^\ell}\right)$$

be the Jacobian matrix whose (i, j)-th entry is $\frac{\partial h_i^\ell}{\partial z_j^\ell}$. Then in matrix form (6) can be written as

$$\left(\frac{\partial E}{\partial z^L}\right) = \left(\frac{\partial h^L}{\partial z^L}\right)\left(\frac{\partial E}{\partial h^L}\right).$$

Note that the chain rule

$$\frac{\partial h_i^\ell}{\partial h_j^{\ell-1}} = \sum_k \frac{\partial z_k^\ell}{\partial h_j^{\ell-1}} \frac{\partial h_i^\ell}{\partial z_k^\ell}$$

can be written in matrix form as

$$\left(\frac{\partial h^{\ell}}{\partial h^{\ell-1}}\right) = \left(\frac{\partial z^{\ell}}{\partial h^{\ell-1}}\right)\left(\frac{\partial h^{\ell}}{\partial z^{\ell}}\right)$$

Thus the vector used in the backpropagation rule is nothing but the matrix products of the following form:

$$\left(\frac{\partial E}{\partial h^{\ell}}\right) = \left(\frac{\partial z^{\ell+1}}{\partial h^{\ell}}\right)\left(\frac{\partial h^{\ell+1}}{\partial z^{\ell+1}}\right)\cdots\left(\frac{\partial h^{L}}{\partial z^{L}}\right)\left(\frac{\partial E}{\partial h^{L}}\right), \tag{10}$$

and

$$\left(\frac{\partial E}{\partial z^{\ell}}\right) = \left(\frac{\partial h^{\ell}}{\partial z^{\ell}}\right)\left(\frac{\partial z^{\ell+1}}{\partial h^{\ell}}\right)\left(\frac{\partial h^{\ell+1}}{\partial z^{\ell+1}}\right)\cdots\left(\frac{\partial h^{L}}{\partial z^{L}}\right)\left(\frac{\partial E}{\partial h^{L}}\right). \tag{11}$$

4 Mathematical supplement: automatic differentiation

Backpropagation algorithm depends heavily on the derivative calculations. Mathematically, calculating derivatives is no big deal. However, if a function is not given explicitly – for example, if it is defined in terms of a certain procedure – calculating derivatives may become no small matter.

There are three way to calculate the derivatives. The first way is to use symbolic differentiation. This method requires the symbolic mathematics engine and it is not applicable if the function is not given explicitly. The second is to use numerical method like finite difference. However, numerical differentiation often entails numerical error and it is rather difficult to control in general. The third way, which by now has become the default in machine learning, is to use the so-called **automatic differentiation** or **algorithmic differentiation**.

It is a numerical way of calculating the *exact* value of derivatives without any aid of the symbolic mathematics engine. It relies on the dual number invented by Clifford in 1873. Dual number is a pair of real numbers $< x, x' >$ which is usually written as $x + x'\epsilon$. The arithmetic operation of dual numbers

are as follows:

$$\begin{aligned}
\lambda(a+a'\epsilon) &= \lambda a + \lambda a'\epsilon \\
(a+a'\epsilon)+(b+b'\epsilon) &= (a+b)+(a'+b')\epsilon \\
(a+a'\epsilon)-(b+b'\epsilon) &= (a-b)+(a'-b')\epsilon \\
(a+a'\epsilon)(b+b'\epsilon) &= ab+(a'b+ab')\epsilon \\
\frac{1}{a+a'\epsilon} &= \frac{1}{a}-\frac{a'}{a^2}\epsilon
\end{aligned}$$

Note that if we put $a = b = 0$ and $a' = b' = 1$ in the fourth line above, we get

$$\epsilon^2 = 0,$$

from which it is easy to prove by induction that

$$(x+x'\epsilon)^n = x^n + nx^{n-1}x'\epsilon.$$

In fact, for any function $f(x)$, $f(x+x'\epsilon)$ is defined as

$$f(x+x'\epsilon) = f(x) + f'(x)x'\epsilon,$$

which is basically the first order Taylor series expansion. This mysterious ϵ can be envisioned as being sort of like the smallest floating point number representable in a computer so that $\epsilon^2 = 0$. This is also reminiscent of the complex number i satisfying $i^2 = -1$.

Let us see how this dual number can be used to compute partial derivatives. Suppose $f(x,y) = xy^2 + y + 5$. Since $\dfrac{\partial f}{\partial y}(x,y) = 2xy + 1$, we can directly compute $f(2,3) = 26$ and $\dfrac{\partial f}{\partial y}(2,3) = 13$.

To see how automatic differentiation can be used to do the same, we look at

$$\begin{aligned}
f(2,3+\epsilon) &= 2(3+\epsilon)^2 + (3+\epsilon) + 5 \\
&= 2(9+6\epsilon) + (3+\epsilon) + 5 \\
&= 26 + 13\epsilon,
\end{aligned}$$

from which $f(2,3) = 26$ and $\dfrac{\partial f}{\partial y}(2,3) = 13$ can be read off. This procedure is systematized by constructing the computational graph which is a parsing

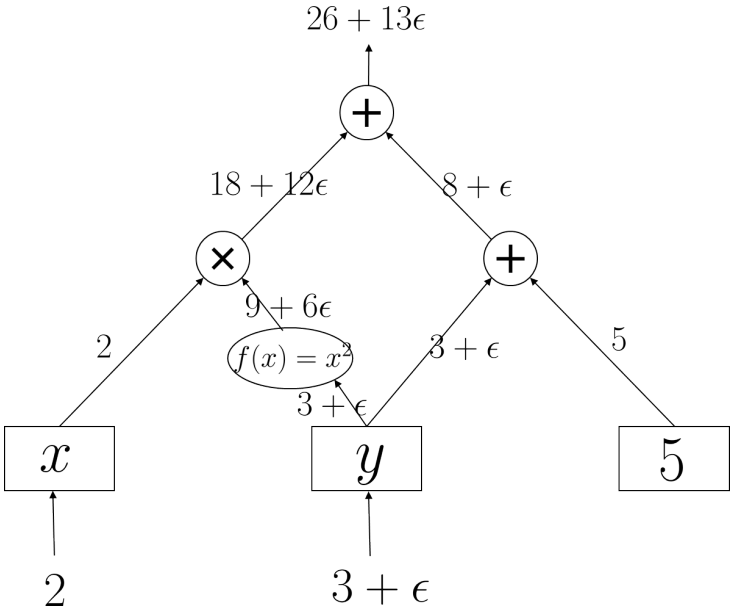

Figure 21: AutoDiff in Forward Mode

tree for the computation of $f(x, y)$. An example of the computational graph for this computation is given in Figure 21. In here, to compute $\dfrac{\partial f}{\partial y}(2, 3)$, one feeds the values $x = 2$ and $y = 3+\epsilon$ at the bottom variables. One then follows the arrows of the computational graph and performs appropriate operations whose results are recorded on the corresponding links. From the value at the top node, one can read off $f(2, 3) = 26$ and $\dfrac{\partial f}{\partial y}(2, 3) = 13$. Note that to compute $\dfrac{\partial f}{\partial x}(2, 3)$, we must redo the similar computation $f(2 + \epsilon, 3)$ on the same computational graph all over again by feeding $x = 2+\epsilon$ and $y = 3$. This may be quite burdensome if there are millions of variables to deal with as is the case in deep learning. This way of directly computing partial derivatives is called the **forward mode automatic differentiation**.

An alternative method called the **reverse mode automatic differentiation** is devised to alleviate this problem. It is basically a repeated application of chain rule. It goes as follows. First, use the forward computational graph to compute values of each node in the forward pass, i.e. from the input values at the bottom nodes through all nodes upward. The number

inside each node in Figure 22 is the value of that node. The symbol n_i at the left of each node is the node number, which one can think of as a variable representing the value of the corresponding node. The idea of reverse mode automatic differentiation is *to trace the variable dependencies along the paths of the computational graph and apply the appropriate chain rules.* Note $f = n_1$ and $n_1 = n_2 + n_3$. Thus $\dfrac{\partial n_1}{\partial n_2} = 1$ and this number is recorded on the link from n_1 to n_2. The number on the link from n_1 to n_3 is gotten similarly. Now $n_2 = n_4 n_5$, from which we get $\dfrac{\partial n_2}{\partial n_5} = n_4 = 9$. Therefore we have

$$\frac{\partial n_1}{\partial n_5} = \frac{\partial n_1}{\partial n_2}\frac{\partial n_2}{\partial n_5} = 9,$$

which is recorded on the link from n_2 to n_5. Note that this is the chain rule (along the path from n_1 tp n_5.). The reason why no other variables enter into this chain rule calculation can be easily seen: namely, a change in n_5 only affects n_2, which again affects n_1, and no other variables are affected. One can read off this dependency from the graph structure, i.e. there is only one path from n_1 to n_5. Similarly, $\dfrac{\partial n_2}{\partial n_4} = n_5 = 2$. Thus

$$\frac{\partial n_1}{\partial n_4} = \frac{\partial n_1}{\partial n_2}\frac{\partial n_2}{\partial n_4} = 2,$$

which is recorded on the link from n_2 to n_4. Now $n_4 = n_6^2$. Thus

$$\frac{\partial n_4}{\partial n_6} = 2n_6 = 6. \tag{12}$$

Thus

$$\frac{\partial n_1}{\partial n_6} = \frac{\partial n_1}{\partial n_4}\frac{\partial n_4}{\partial n_6} = 12,$$

which is recorded on the link from n_4 to n_6. Note that this does not yet give the value of $\dfrac{\partial f}{\partial y}(2,3)$, although in terms of values of variables, $f = n_1$ and $y = n_6$. The reason is that there is another path from n_1 to n_6 via n_3, and this calculation only reflects the dependency of n_1 on n_6 through the path via n_4. This number 12 recorded on the link from n_4 to n_6 represents only the part of this dependency. Another path of dependency of n_1 on n_6 is via the path through n_3. Computing similarly, we can get the number 1 on the

link from n_3 to n_6. Since n_1 depends on n_6 through these two paths, we have to add these two numbers to get

$$\frac{\partial f}{\partial y}(2,3) = 12 + 1 = 13.$$

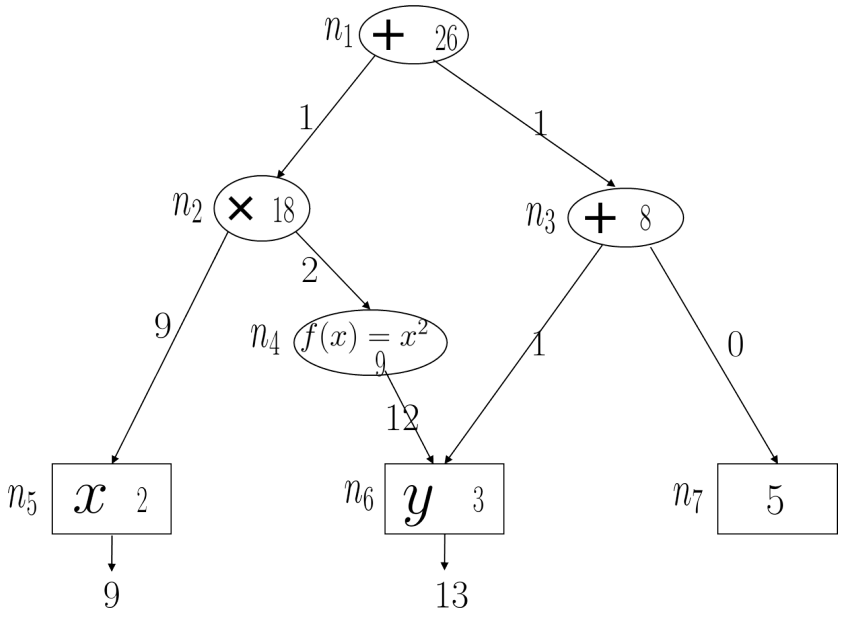

Figure 22: AutoDiff in Reverse Mode

This reverse mode automatic differentiation is heavily dependent on derivative calculation at each node. This individual differentiation is where the forward mode automatic differentiation enters. For instance (12) is gotten by computing $(3+\epsilon)^2 = 9 + 6\epsilon$ and noting that this gives $\dfrac{\partial n_4}{\partial n_6} = 6$.

The reverse mode automatic differentiation is a way of organizing derivative calculations to get all partial derivatives at once with one single computational graph, which is a big advantage in the case like deep learning where there are millions of input variables while there are only a few output variables. In contrast, one has to repeat the similar computations millions of times if one relies entirely on the forward mode automatic differentiation. This is the reason why most deep learning software implements reverse mode automatic differentiation.

References

[1] Community Portal for Automatic Differentiation
http://www.autodiff.org/

[2] Goodfellow, I., Bengio, Y., Courville, A., *Deep Learning*, MIT Press (2016)

[3] Minsky M. L. and Papert S. A., *Perceptrons: An Introduction to Computational Geometry*, MIT Press (1969)

[4] Rosenblatt, F., *A Probabilistic Model for Information Storage and Organization in the Brain*, Cornell Aeronautical Laboratory, Psychological Review, v65, No. 6, 386408 (1958)

[5] Rosenblatt, F., *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Spartan Books (1962)

[6] Rumelhart, D., McClelland, J., PDP Research Group, *Parallel Distributed Processing, vol 1 & 2*, Bradford Book (1987)