Neural Networks

October 27, 2020

Abstract

Topics: feedforward neural networks, activation functions, forward and backward propagations, regularization.
Readings: Chapter 5 of PRML.

1 Feedforward neural networks

We study a family of neural networks call "feedforward neural networks". The term "feedforward" means that the network is layered and an input will be fed into the first layer, whose output will be fed into the second, and so on, until the output layer is reached. There is no information flowing backward from a later layer to an earlier layer.

In particular, we will use logistic regression as a building-block to construct a type of neural network called "multi-layered perceptron" (MLP for short). An MLP consists of one input layer, one output layer, and one or more hidden layers. The input layer is the given features of an example $\mathbf{x} = [x_1, \ldots, x_n]^\top \in \mathbb{R}^n$. The output layer is the prediction of the target value y, which is assumed to be binary for the moment. A hidden layer is a layer between the input and output layer. It is called hidden since it is not observed like the input feature vector $\mathbf{x}$ and the target y. Let L be the total number of layers excluding the input layer (since the input layer does not do any computation). Sometimes L is called the "depth" of the neural network. We index the layers by the integer $\ell \in \{0, \ldots, L\}$, where $\ell = 0$ represents the input layer and $\ell = L$ represents the output layer. Figure 1 shows two example MLPs.

A layer consists of a set of "neurons", each of which represents some computation unit that has an input and an output (like a function). A neuron is represented as a circle in Figure 1. In a feedforward neural network, a neuron will take, as input, the output of the neurons from the previous layer and computes a single output, which can be used in the next layer. We assume that a neuron takes the following form

$$a = g(z) = g(\mathbf{w}^\top \mathbf{a} + b), \tag{1}$$

where $\mathbf{a}$ is some output from the previous layer. The linear function maps $\mathbf{a}$ to $z = \mathbf{w}^\top \mathbf{a} + b$, followed by a non-linear function g, which maps from $\mathbb{R}$ to $\mathbb{R}$. The function g is called "activation function" and we will discuss different activation functions later.

Example 1.1. *A typical neuron is a logistic regression model with the sigmoid activation function*

$$a = g(z) = \sigma(z) = \frac{1}{1 + \exp\{-(\mathbf{w}^\top \mathbf{a} + b)\}}. \tag{2}$$

The neural network is shown in the left panel of Figure 1.

1.1 Notation

Since there are multiple layers, each with multiple neurons, we need a notation system to denote these elements.

- L: the number of layers of an MLP. The input layer is not counted.
- $n[\ell]$: the number of neurons at the ℓ-th layer, $\ell = 0, \ldots, L$.

- $g^{[\ell]}$: the activation function at layer ℓ. We add the index $[\ell]$ to the activation function g, since different layer can take different activation functions. The activation function applies to each element of the matrix or vector in its argument.

  In the output layer, there is just one neuron that outputs the probability of classifying the input $\mathbf{x}$ into the positive class for binary classification problems. Therefore, the activation function at the last layer $g^{[L]}$ is chosen to be the sigmoid function. We will discuss MLP for multi-class classification problems in Section 4.

- $\mathbf{z}^{[\ell]} \in \mathbb{R}^{n[\ell]}$: the term that will be fed into the activation function $g^{[\ell]}$ to obtain $\mathbf{a}^{[\ell]}$.

- $\mathbf{a}^{[\ell]} \in \mathbb{R}^{n[\ell]}$: the output of the neurons of the ℓ-th layer. Then $a_j^{[\ell]}$, $j = 1, \ldots, n[\ell]$ represents the output of the j-th neuron at the ℓ-th layer. In the input layer, we denote $\mathbf{x}$ by $\mathbf{a}^{[0]}$. The square bracket in the superscript is to be distinguished from the superscript (i) that indexes training examples $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$. If we need to distinguish different training examples, we denote $\mathbf{x}^{(i)}$ by $\mathbf{a}^{[0](i)}$.

- $W_j^{[\ell]} \in \mathbb{R}^{n[\ell-1]}$: the weights for the j-th neuron at the ℓ-th layer. By stacking these $n[\ell]$ vectors $\{W_j^{[\ell]} : j = 1, \ldots, n[\ell]\}$, we obtain the weight matrix $W^{[\ell]} \in \mathbb{R}^{n[\ell] \times n[\ell-1]}$ for layer ℓ. In MLP, each neuron from layer $\ell - 1$ is connected to all neurons in layer ℓ, so the matrix W is of size $n[\ell] \times n[\ell - 1]$.

- $b_j^{[\ell]} \in \mathbb{R}$: the bias term for the j-th neuron of the ℓ-th layer. We can concatenate these $n[\ell]$ bias terms to construct the bias vector $\mathbf{b}^{[\ell]} \in \mathbb{R}^{n[\ell]}$.

In the above definitions, the W matrices and $\mathbf{b}$ vectors are parameters of the network and will be trained using training data. The $\mathbf{a}$ and $\mathbf{z}$ vectors are intermediate results. The choices of the number of layers L, the activation function $g^{[\ell]}$, the number of neurons at each layer $n[\ell]$, collectively determine the *architecture* of the neural network.

Example 1.2. *The MLP in the left panel of Figure 1 is a logistic regression model with a single layer (the output layer) and $L = 1$. The input layer has 3 features so $n[0] = 3$. The weight matrix for layer 1 is $W^{[1]}$ is a 1×3 matrix (row vector). The bias for layer 1 is $b^{[1]}$ is a scalar. The activation function for the output neuron is $a^{[1]} = \sigma(z^{[1]}) = \sigma(W^{[1]}\mathbf{x} + b^{[1]})$.*

Example 1.3. *The MLP in the right panel of Figure 1 is more complex than logistic regression, with one hidden layer between the input and output layer. There are $L = 2$ layers. At the input layer, $\mathbf{a}^{[0]} = \mathbf{x} \in \mathbb{R}^3$. Since the input has 3 features and there are 5 neurons in the first layer, $n[0] = 3$ and $n[1] = 5$. As a result, $W^{[1]}$ is a 5×3 matrix and $\mathbf{b}^{[1]}$ is a 5×1 column vector. Since the output has just one neuron, $n[2] = 1$ and therefore $W^{[2]}$ is a 1×3 matrix (row vector) and $\mathbf{b}^{[2]}$ is a 1×1 vector (scalar).*

We can see that, if the activation functions $g^{[1]}$ and $g^{[2]}$ are chosen to be the sigmoid function, then this neural network is a composition of 6 logistic regression models.

2 Forward propagation

Forward propagation is to take the input feature vector $\mathbf{x}$ and compute the activations of each neuron layer-by-layer in a neural network, as follows,

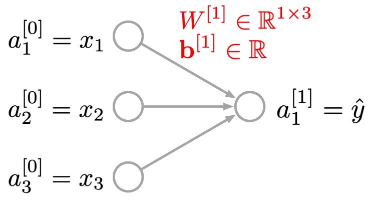
$$a_j^{[\ell]} = g^{[\ell]}(W_j^{[\ell]}\mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]}), \quad j = 1, \ldots, n[\ell], \quad \ell = 1, \ldots, L. \tag{3}$$

Take the neural network shown in the right panel of Figure 1 as an example. Given the input feature $\mathbf{x} \in \mathbb{R}^2$, we can first compute the activations at layer 1:

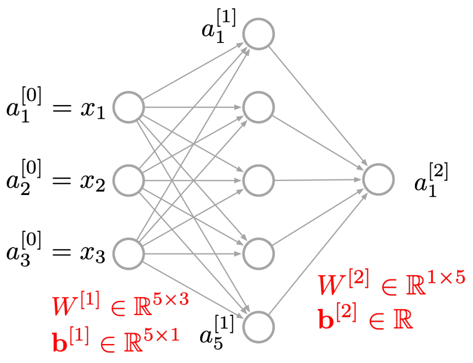
$$a_j^{[1]} = g^{[1]}(W_1^{[1]}\mathbf{a}^{[0]} + b_1^{[1]}), \quad \mathbf{a}^{[0]} = \mathbf{x}, \quad j = 1, \ldots, n[1]. \tag{4}$$

After obtaining all activations at layer 1, collectively denoted by $\mathbf{a}^{[1]} = [a_1^{[1]}, \ldots, a_5^{[1]}]^\top$, we compute the activation at layer 2 as

$$a_1^{[2]} = g^{[2]}(W_1^{[2]}\mathbf{a}^{[1]} + b_1^{[2]}). \tag{5}$$



(a) Logistic regression as a neural network: there are three input features, without the appended constant 1. The output $a_1^{[1]} = \sigma(W^{[1]}\mathbf{a}^{[0]} + \mathbf{b}^{[1]})$ is the predicted probability of $\mathbf{x}$ belonging to class 1. The parameters of this neural network are $W^{[1]}$ and $\mathbf{b}^{[1]}$.



$a_1^{[1]}$
$a_1^{[0]} = x_1$
$a_2^{[0]} = x_2$
$a_3^{[0]} = x_3$
$a_1^{[2]}$
$W^{[1]} \in \mathbb{R}^{5\times 3}$
$\mathbf{b}^{[1]} \in \mathbb{R}^{5\times 1}$
$a_5^{[1]}$
$W^{[2]} \in \mathbb{R}^{1\times 5}$
$\mathbf{b}^{[2]} \in \mathbb{R}$

(b) A two layer MLP: layer 0 is the input, layer 1 is a hidden layer, and layer 2 is the output layer. The superscript $[\ell]$ in $a_j^{[\ell]}$ indexes the layers, while the subscript indexes the input features or neurons in a layer. The parameters of this neural network are $W^{[1]}$, $W^{[2]}$, $\mathbf{b}^{[1]}$ and $\mathbf{b}^{[2]}$.

Figure 1: Two examples of MLPs.

2.1 Vectorization

The above computation needs two for loops: the outer for loop will go through the layers and the inner for loop will go through the neurons within a layer. To speed up the computation, the inner loop can be replaced by a matrix-vector multiplications and an vector addition as follows.

$$\mathbf{z}^{[\ell]} = W^{(\ell)}\mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]}, \tag{6}$$
$$\mathbf{a}^{[\ell]} = g^{[\ell]}(\mathbf{z}^{\ell}), \tag{7}$$

where the individual operations are:

- $W^{(\ell)}\mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]}$ is the multiplication of the $n[\ell] \times n[\ell-1]$ matrix $W^{(\ell)}$ and the $n[\ell-1] \times 1$ column vector $\mathbf{a}^{[\ell-1]}$, added to the $n[\ell] \times 1$ column bias vector $\mathbf{b}^{[\ell]}$, resulting in a $n[\ell] \times 1$ column vector $\mathbf{z}^{[\ell]}$.
- The activation function $g^{[\ell]}$ is applied to each element of the vector $\mathbf{z}^{[\ell]}$, resulting in the activation vector $\mathbf{a}^{[\ell]}$ for all neurons at layer ℓ.

For feedforward neural networks, there is no way to eliminate the output loop for further speedup.

When dealing with multiple examples, such as in mini-batch training (Algorithm 1), the activations can be computed for all examples at once. Let $X^\top$ be the data matrix with the i-th column being the feature vector $\mathbf{x}^{(i)}$. In the first layer,

$$A^{[1]} = g^{[1]}(W^{[1]}X^\top + \mathbf{b}^{[1]}), \tag{8}$$

where $A^{[1]} = [\mathbf{a}^{[1](1)}, \ldots, \mathbf{a}^{[1](m)}] \in \mathbb{R}^{n[1]\times m}$ and each column is the activation for an example. The multiplication $W^{[1]}X^\top$ is to map all examples to a $n[1] \times m$ matrix. The addition of the $n[1] \times 1$ column vector $\mathbf{b}^{[\ell]}$ to that matrix should be interpreted as "replicating the column vector to m columns and then adding the resulting matrix" (this is called "broadcasting" in numpy). We denote $W^{[1]}X^\top + \mathbf{b}^{[1]}$ by $Z^{[1]}$. Lastly, the activation function $g^{[1]}$ is applied to each element of $Z^{[1]}$. In general, for each layer $\ell = 1, \ldots, L$, we can compute

$$Z^{[\ell]} = W^{(\ell)}\mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]} \qquad (\mathbf{a}^{[0]} = X^\top), \tag{9}$$
$$A^{[\ell]} = g^{[\ell]}(Z^{[\ell]}). \tag{10}$$

You can verify that the sizes of $Z^{[\ell]}$ and $A^{[\ell]}$ are both $n[\ell] \times m$.

2.2 Activation functions

Sigmoid function, as an activation function, is in fact not a good option for the hidden layers but is only used for the output layer for binary classification problems. To see why, we can take the

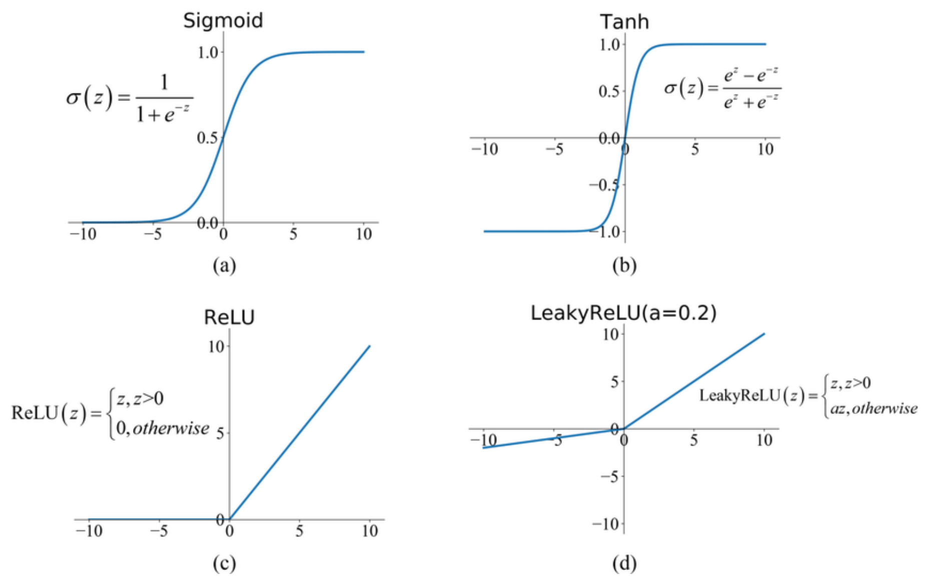


Figure 2: Four activation functions

derivative of the sigmoid function

$$\sigma(z) = \frac{1}{1+\exp\{-z\}} \tag{11}$$

with respect to z and obtain

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z)). \tag{12}$$

We can see that when $z \to \infty$ or $z \to -\infty$, $\sigma(z)(1-\sigma(z)) \to 0$ exponentially fast. During the training of MLP using backward propagation, the gradients will be close to zero and it is hard to make progress in updating the network parameters. The problem is called "*gradient vanishing*", which is more severe when the networks becomes deeper.

There are other options for activation functions beyond the sigmoid function (see Figure 2).

- Hyperbolic tangent:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{13}$$

  This function has the range $(-1, 1)$, compared to the range $(0, 1)$ of the sigmoid function. The tanh function also has the vanishing gradient issues, since its derivative with respect to z is

$$\frac{d\tanh(z)}{dz} = 1 - \tanh(z)^2. \tag{14}$$

- ReLU (Rectified Linear Unit):

$$\mathrm{ReLU}(z) = \max\{0, z\}. \tag{15}$$

  There is no gradient vanishing problem when $z > 0$ since the gradient will be 1. Theoretically, there activation function is not differentiable at $z = 0$. In practice, this can be addressed by setting the gradient to either 0 or 1 at $z = 0$.

- Leaky ReLU:

$$\mathrm{ReLU}(z) = \max\{az, z\}, a \in (0, 1). \tag{16}$$

  The parameter a can be a fixed constant or be learned during network training.

Remark: All these activation functions are non-linear in z. It is important to use non-linear activation functions in neural networks. To see the reason, let's use the identity mapping as the activation function in all layers, then a two layer neural network is computing the following

$$\mathbf{a}^{[2]} = W^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]} = W^{[2]}(W^{[1]}\mathbf{a}^{[0]} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]} = W^{[2]}W^{[1]}\mathbf{a}^{[0]} + W^{[2]}\mathbf{b}^{[1]} + \mathbf{b}^{[2]} = \tilde{W}\mathbf{x} + \tilde{\mathbf{b}}. \tag{17}$$

That is, the two-layered network is equivalent to a linear regression model.

3 Backward Propagation

Remark: this section is based on Chapter 6.5 of the book "Deep Learning" by Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Warning: Our notation is rather different from theirs and the book may contain typos.

Backward propagation, or backprop, computes the gradient of a loss function with respect to the network parameters. The algorithm, combined with an optimization algorithm, such as mini-batch stochastic or gradient descent, is widely used to train neural networks, ranging from simple MLP and more complicated deep networks. The algorithm is sometimes misunderstood to be *the* training method, but it is really just used for computing the gradients needed by a numerical optimization algorithms to update the network parameters.

Technically, backprop implements the chain rule in calculus over functions that are composed of multiple elementary operations. Different from the forward propagation that uses a neural network to transform the input $\mathbf{x}$ to a prediction, backprop starts from the loss based on the network output, and compute the gradients of the loss with respect to the network parameters backwards and layer-by-layer, using the gradients that have been computed in a higher layer to compute the gradients in a lower layer.

3.1 Chain rule

The chain rule computes the gradient of a composite function with respect to the input. For example, for the composite function $(f \circ g)(x) = f(g(x)) = f(z)$, where $z = g(x)$, the chain rule is

$$\frac{d(f \circ g)(x)}{dx} = \frac{df(z)}{dz}\frac{dz}{dx} = f'(g(x))g'(x). \tag{18}$$

If the function $f : \mathbb{R}^m \to \mathbb{R}$ while $g : \mathbb{R} \to \mathbb{R}^m$, that is, the output of g and the input of f becomes a vector $\mathbf{z} = g(x)$, then the chain rule becomes

$$\frac{d(f \circ g)}{dx} = \sum_{j=1}^{m} \frac{df(\mathbf{z})}{dz_j}\frac{dz_j}{dx} = \sum_{j=1}^{m} \frac{df(\mathbf{z})}{dz_j}\frac{dg_j(x)}{dx} = \left(\frac{\partial g}{\partial x}\right)^{\top} \frac{\partial f}{\partial \mathbf{z}}, \tag{19}$$

where g_j is the j-th output of the vector function g and $\partial g/\partial x = [dg_1/dx, \ldots, dg_m/dx]^{\top}$ is a vector of the same shape as $\mathbf{z}$. The vector $\partial f/\partial \mathbf{z}$ is the gradient of the scalar function f with respect to the vector $\mathbf{z}$ and is of the same size of $\mathbf{z}$.

Chain rule can also handle vector inputs[1]. In particular, let the input $\mathbf{x} \in \mathbb{R}^n$ and the composite function $(f \circ g)(\mathbf{x}) = f(g(\mathbf{x})) = f(\mathbf{z})$, where $g : \mathbb{R}^n \to \mathbb{R}^m$ and $f : \mathbb{R}^m \to \mathbb{R}$. One example of such a g is the linear mapping $g(\mathbf{x}) = A\mathbf{x}$ where $A \in \mathbb{R}^{m\times n}$. Then the chain rule becomes

$$\frac{\partial(f \circ g)}{\partial \mathbf{x}} = \left(\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}\right)^{\top} \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \tag{20}$$

where $\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}$ is the so-called Jacobian matrix for the multiple-output function g:

$$\frac{\partial g(\mathbf{x}))}{\partial \mathbf{x}} = \begin{bmatrix} \frac{dg_1}{dx_1} & \cdots & \frac{dg_1}{dx_n} \\ \vdots & \ddots & \vdots \\ \frac{dg_m}{dx_1} & \cdots & \frac{dg_m}{dx_n} \end{bmatrix} \tag{21}$$

We can see that the Jacobian matrix is a generalization of the gradient vector $\partial g/\partial x$ when x is a scalar and g is a multiple-output function. Eq. (21) is the Jacobian-gradient product and is used in the backprop algorithm.

3.2 Backprop for MLP

A feedforward neural network can be seen as a composition of multiple layers that processes the input $\mathbf{x}$ and produces the prediction $\hat{y}$. Take the simple MLP in the right panel of Figure 1 as an example. In the forward computation, we have the following two steps for each layer $\ell = 1, \ldots, L$

$$\begin{aligned} \mathbf{z}^{[\ell]} &= W^{(\ell)}\mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]}, & (22) \\ \mathbf{a}^{[\ell]} &= g^{[\ell]}(\mathbf{z}^{\ell}), & (23) \end{aligned}$$

[1]For simplicity, we assume the final output is a scalar

We assume that the activation function at the last layer $g^{[L]}$ is the sigmoid function for predicting the probability $\Pr(y = 1|\mathbf{x})$. Since our goal is to train the network to predict binary target labels for input examples, we further compute the cross-entropy loss on each training example $(\mathbf{x}, y)$

$$J(y, \hat{y}) = J(y, a^{[L]}) = -y \log a^{[L]} - (1 - y) \log(1 - a^{[L]}). \tag{24}$$

Training the network can be done by minimizing the loss function overall all training examples. For simplicity, the following steps of backward propagation are computed on a single training example.

- Find the gradient of J with respect to $a^{[L]}$:

$$\frac{dJ(y, a^{[L]})}{da^{[L]}} = -\frac{y}{a^{[L]}} + \frac{1-y}{1-a^{[L]}}. \tag{25}$$

- Find the gradient of J with respect to $z^{[L]}$:

$$\frac{dJ(y, a^{[L]})}{dz^{[L]}} = \frac{dJ(y, a^{[L]})}{da^{[L]}} \times \frac{da^{[L]}}{dz^{[L]}} = -y(1 - a^{[L]}) + (1 - y)a^{[L]} = a^{[L]} - y \tag{26}$$

  since $\frac{da^{[L]}}{dz^{[L]}} = a^{[L]}(1 - a^{[L]})$ using Eq. (11).

- Find the gradient of J with respect to $W^{[L]}$ and $b^{[L]}$

$$\frac{dJ(y, a^{[L]})}{dW^{[L]}} = \frac{dJ(y, a^{[L]})}{dz^{[L]}} \times \frac{dz^{[L]}}{dW^{[L]}} = (a^{[L]} - y)(a^{[L-1]})^\top, \tag{27}$$

  since $z^{[L]} = W^{[L]}a^{[L-1]} + b^{[L]}$ and the row vector $(a^{[L-1]})^\top$ is the Jacobian matrix of $z^{[L]}$ with respect to $W^{[L]}$. Similarly,

$$\frac{dJ(y, a^{[L]})}{db^{[L]}} = a^{[L]} - y \tag{28}$$

  since $b^{[L]}$ is a scalar.

- In general, to find the gradient of J with respect to $W^{[\ell]}$ and $b^{[\ell]}$ for any $\ell < L$, we need to find $\frac{dJ(y,a^{[L]})}{d\mathbf{z}^{[\ell]}}$. Backprop algorithm does this by computing $\frac{dJ(y,a^{[L]})}{d\mathbf{z}^{[\ell]}}$ for $\ell = L - 1, \dots, 1$ in that order. Based on the foward propagation, the mapping from $\mathbf{a}^{[\ell]}$ to $\mathbf{z}^{[\ell+1]}$ is a linear function $\mathbf{z}^{[\ell+1]} = W^{[\ell+1]}\mathbf{a}^{[\ell]}$, with vector input and output. Therefore we can use Eq. (21), the Jacobian-gradient product, to find

$$\frac{dJ(y, a^{[L]})}{d\mathbf{a}^{[\ell]}} = (W^{[\ell+1]})^\top \frac{dJ(y, a^{[L]})}{d\mathbf{z}^{[\ell+1]}} \tag{29}$$

  Assume that $\frac{dJ(y,a^{[L]})}{d\mathbf{z}^{[\ell+1]}}$ has been calculated, which is true for $\ell = L - 1$.

  The $\mathbf{a}^{[\ell]}$ vector is obtained by applying the activation function $g^{[\ell]}$ to the vector $\mathbf{z}^{[\ell]}$ element-wisely, and

$$\frac{dJ(y, a^{[L]})}{d\mathbf{z}^{[\ell]}} = \left[(W^{[\ell+1]})^\top \frac{dJ(y, a^{[L]})}{d\mathbf{z}^{[\ell+1]}}\right] \otimes (g^{[\ell]})'(\mathbf{z}^{[\ell]}), \tag{30}$$

  where $\otimes$ means element-wise multiplication of two vectors and $(g^{[\ell]})'(\mathbf{z}^{[\ell]})$ is the element-wise derivative of the activation function $g^{[\ell]}$ at the input $\mathbf{a}^{[\ell]}$.

  When $g^{[\ell]}(\mathbf{a}^{[\ell]}) = \sigma(\mathbf{z}^{[\ell]})$, $(g^{[\ell]})'(\mathbf{z}^{[\ell]})$ is the vector $(\mathbf{z}^{[\ell]}) \otimes (1 - \mathbf{a}^{[\ell]})$; when $g^{[\ell]}(\mathbf{a}^{[\ell]}) = \mathrm{ReLU}(\mathbf{a}^{[\ell]})$, $(g^{[\ell]})'(\mathbf{z}^{[\ell]})$ is the vector $\mathbb{1}[\mathbf{a}^{[\ell]} \geq \mathbf{0}]$.

  Starting from $\frac{dJ(y,a^{[L]})}{d\mathbf{z}^{[L]}}$, by applying Eq. (30) iteratively for $\ell = L - 1, \dots, 1$, all $\frac{dJ(y,a^{[L]})}{d\mathbf{z}^{[\ell]}}$ can be computed. Based on $\frac{dJ(y,a^{[L]})}{d\mathbf{z}^{[\ell]}}$, obtaining $\frac{dJ(y,a^{[L]})}{dW^{[\ell]}}$ and $\frac{dJ(y,a^{[L]})}{db^{[\ell]}}$ is similar to Eq. (27) and Eq. (28),

$$\frac{dJ(y, a^{[L]})}{dW^{[\ell]}} = \frac{dJ(y, a^{[L]})}{d\mathbf{z}^{[\ell]}} \times \frac{d\mathbf{z}^{[\ell]}}{dW^{[\ell]}} = \frac{dJ(y, a^{[L]})}{d\mathbf{z}^{[\ell]}} \times (a^{[\ell-1]})^\top, \tag{31}$$

$$\frac{dJ(y, a^{[L]})}{db^{[\ell]}} = \frac{dJ(y, a^{[L]})}{d\mathbf{z}^{[\ell]}} \times \frac{d\mathbf{z}^{[\ell]}}{db^{[\ell]}} = \frac{dJ(y, a^{[L]})}{d\mathbf{z}^{[\ell]}}, \tag{32}$$

The above steps are demonstrated in Figure 3.

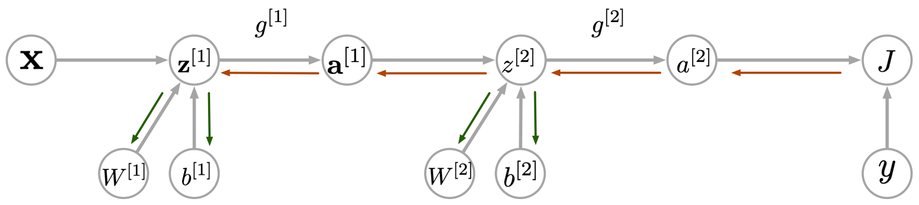


Figure 3: Backward propagation of the MLP in Figure 1. The gray-colored arrows denote the forward propagation computations, which must be executed to find the activations $\mathbf{a}^{[\ell]}$ for all layers $\ell = 1, \dots, L$, that backprop will use. The red-colored arrows denote the backprop steps that aim at computing $\frac{dJ(y,a^{[L]})}{d\mathbf{z}^{[\ell]}}$. The gree-colored arrows are for computing the gradients with respect to the network parameters $W^{[\ell]}$ and $b^{[\ell]}$. There is no backprop towards the given training example $\mathbf{x}$ and y since they are constants and will not be updated.

3.3 Regularization

To avoid overfitting of neural network training, we can regularize the loss function using the Frobenius norm of the W matrices (the matrix version of ℓ-2 norm of vectors), so that the loss function becomes

$$J + \lambda \sum_{\ell=1}^{L} \|W^{[\ell]}\|_F^2, \tag{33}$$

where

$$\|W^{[\ell]}\|_F^2 = \sum_{i=1}^{n[\ell-1]} \sum_{j=1}^{n[\ell]} (W_{ji}^{[\ell]})^2. \tag{34}$$

This does not add much complexity to the backprop algorithm. Rather, the new gradient with respect to $W^{[\ell]}$ becomes

$$\frac{dJ(y, a^{[L]})}{dW^{[\ell]}} + \lambda W^{[\ell]}. \tag{35}$$

3.4 Mini-batch training of MLP

Algorithm 1 trains an MLP using the mini-batch stochastic gradient descent algorithm. It runs in multiple iterations until the parameters converge. In each iteration, a small subset of size B is sampled from the training data $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$. The forward propagation is run to obtain activations $\mathbf{a}^{[\ell](i)}$ of neurons at all layers $\ell = 1, \dots, L$ and for all training example $(\mathbf{x}^{(i)}, y^{(i)})$ in the mini-batch. Then the backprop computes the gradient of the parameters at each training example and then averages the gradients over the examples in the mini-batch. Lastly, before entering the next iteration, the network parameters are updated according to

$$W^{[\ell]} \leftarrow W^{[\ell]} - \eta \frac{1}{B} \sum_{i=1}^{B} \left[\frac{\partial J^{(i)}}{\partial W^{[\ell]}} \right] - \lambda W^{[\ell]}, \tag{36}$$

$$b^{[\ell]} \leftarrow b^{[\ell]} - \eta \frac{1}{B} \sum_{i=1}^{B} \frac{\partial J^{(i)}}{\partial b^{[\ell]}}. \tag{37}$$

To vectorize the backprop, after the vectorized forward propagation using Eq. (9) and (10), we can use the following equations,

$$\frac{\partial J}{\partial Z^{[L]}} = A^{[L]} - Y \in \mathbb{R}^{n[L] \times B}, \tag{38}$$

$$\frac{\partial J}{\partial W^{[L]}} = \frac{1}{B} (A^{[L]} - Y)(A^{[L-1]})^\top \in \mathbb{R}^{n[L] \times n[L-1]}, \tag{39}$$

$$\frac{\partial J}{\partial \mathbf{b}^{[L]}} = \frac{1}{B} (A^{[L]} - Y) \mathbf{1}_{B \times 1} \in \mathbb{R}^{n[L]}, \tag{40}$$

$$\frac{\partial J}{\partial Z^{[\ell]}} = (W^{[\ell+1]})^\top \frac{\partial J}{\partial Z^{[\ell+1]}} \otimes (g^{[\ell]})'(Z^{[\ell]}) \in \mathbb{R}^{n[\ell] \times B}, \tag{41}$$

$$\frac{\partial J}{\partial W^{[\ell]}} = \frac{1}{B}\frac{\partial J}{\partial Z^{[\ell]}}(A^{[\ell-1]})^\top \in \mathbb{R}^{n[\ell]\times n[\ell-1]}, \tag{42}$$

$$\frac{\partial J}{\partial \mathbf{b}^{[\ell]}} = \frac{1}{B}\frac{\partial J}{\partial Z^{[\ell]}}\mathbf{1}_{B\times 1} \in \mathbb{R}^{n[\ell]\times 1}, \tag{43}$$

Using mini-batch, rather than a single training example or the full training set, is a trade-off between training stability and computational efficiency. If a single training example is used to compute the gradient, the gradient is too much depend on the particular training example being sampled and may not stable. Using the full training set can compute more stable gradients but since the activations of the neurons of all training examples need to be stored before backprop finishes, there may not be sufficient memory for the storage. This is particularly true for deep networks for images: both the sizes of the training data and the neurons are large and even modern GPUs can't process all training images at once.

Algorithm 1: Training MLP using mini-batch stochastic gradient descent

```
Given: training data $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, an MLP, $T$ (max iterations);
Initialization: randomly initialize parameters $W^{[\ell]}$ and $b^{[\ell]}$, learning rate $\eta$, mini-batch size
  $B$;
for $t = 1$ to $T$ or the parameters converge do
    Randomly sample a mini-batch of size $B$ from $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$;
    Use forward propagation to compute the activations and the loss;
    Use backward propagation to compute the gradient of the loss with respect to the
      parameters;
    Update the parameters using Eqs. (36) and (37).
end
return The trained MLP.
```

4 MLP for multi-class prediction

We have been assuming the target label $y \in \{0, 1\}$. In the most successful application of deep learning, such as object recognition, there are usually thousands of classes to be predicted. When using neural network in reinforcement learning as a policy, there can be more than two actions under a state.

Assume that there are K classes, $K > 2$. The only change we need to make to the MLP is the output layer: instead to using the sigmoid function to predict the probability of $\mathbf{x}$ being positive, the softmax function with K input entries is used to predict the probabilities that $\mathbf{x}$ belongs to each of the K classes. In particular, given the activation of the second last layer $\mathbf{a}^{[L-1]}$, the activation $\mathbf{a}^{[L]}$ is computed as

$$\mathbf{z}^{[L]} = W^{[L]}\mathbf{a}^{[L-1]} = \left[z_1^{[L]}, \dots, z_K^{[L]}\right], \tag{44}$$

$$\hat{\mathbf{y}} = \mathbf{a}^{[L]} = \left[\frac{\exp(z_1^{[L]})}{Z}, \dots, \frac{\exp(z_K^{[L]})}{Z}\right], \tag{45}$$

where $Z = \sum_{k=1}^K \exp(z_k^{[L]})$. That is, $\mathbf{a}^{[L]} = \text{softmax}(\mathbf{z}^{[L]})$.

According to the lecture note of multi-class logistic regression, we know that the partial derivative of the multi-class cross-entropy (or negative log-likelihood) loss function with respect to $\mathbf{z}^{[L]}$ is

$$\frac{\partial J(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{[L]}} = \mathbf{a}^{[L]} - \mathbf{y}, \tag{46}$$

where $\mathbf{y}$ is a K-dimension one-hot vector, with the k-th element being one indicating that the corresponding training example belongs to the k-th class. The remaining parts of the backprop algorithm remain the same as the binary classification situation.