ECE586 MDPs and Reinforcement Learning **University of Illinois at Urbana-Champaign**
Spring 2019 **Acknowledgment: R. Srikant's Notes and Discussions**

Lecture 7: Neural Networks

Instructor: Dimitrios Katselis *Scribe: Alireza Moradzadeh, Cathy Shih*

7.1 Networks: Neurons and Layers

Neural networks are a computing structure inspired by the biological neural networks in brains. The neural network itself is not an algorithm, but rather a framework that can process complex data inputs by changing the importance, weights, at each neurons and by inducing more non-linearity with layers. Such systems "learn" to perform tasks by considering labeled data, and they are able to automatically generate identifying characteristics from the learning material that they process.

7.1.1 Neurons

A neural network is a collection of connected units or nodes called artificial neurons. For short, we just call them neurons here. Each connection, like the synapses in a brain, can transmit a signal from one artificial neuron to another. An artificial neuron that receives a signal can process it and then signal additional artificial neurons connected to it.

Definition 7.1. *Neuron: A neuron (perceptron) is a nonlinear function, which takes $x \in \mathbb{R}$ as input and produces $\sigma(x) \in \mathbb{R}$ as output.*

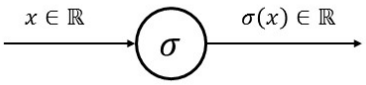

Figure 7.1: A neuron, take $x \in \mathbb{R}$ as input and produces $\sigma(x) \in \mathbb{R}$ as output.

In implementations, the input to a neurons is in real space, and the output of each neuron is computed by activation function of the sum of its inputs. These activation functions σ are usually chosen to be non-linear functions. Few examples are shown in Fig. 7.2, each function is explained blow,

Sign function: Sign function is one of the nonlinearity used in a neuron, which has value of 1 for possitive x and value of -1 for negative x. Mathematically it can be expressed as,

$$\sigma(x) = \begin{cases} +1, & \text{if } x \geq 0 \\ -1, & \text{otherwise} \end{cases} \tag{7.1}$$

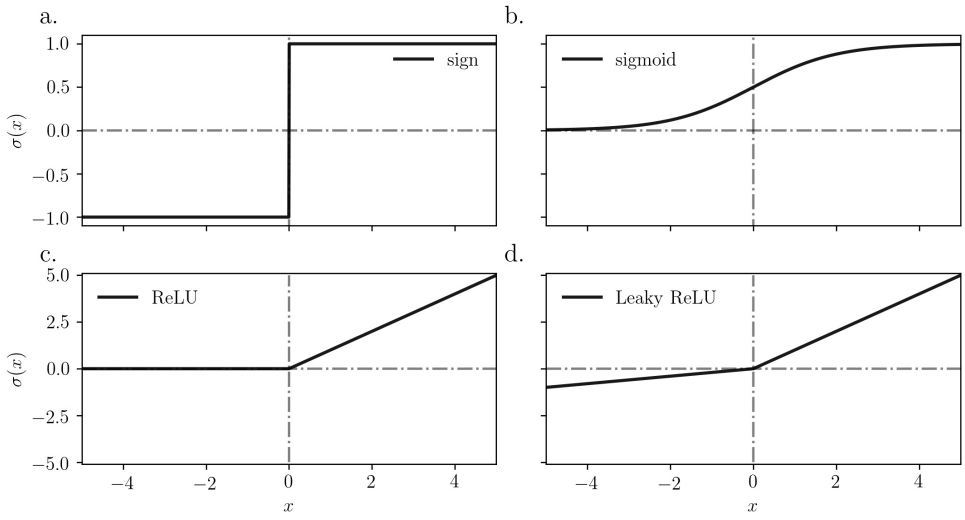

Figure 7.2: Various activation functions used in a neuron. a. sign function b. sigmoid c. Rectifier linear unit (ReLU) d. Leaky ReLU

Sigmoid function: Sigmoid function, also known as logistic function, has as "S" shape curve. For $x \to \infty$, sigmoid function approaches 1, while for $x \to -\infty$, it approaches to zero. Mathematically sigmoid funciton can be expressed as,

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{7.2}$$

ReLU function: The rectifier linear unit (ReLU) is an activation function defined as the positive part of its argument.

$$\sigma(x) = \max(0, x) = [x]^+ \tag{7.3}$$

Note that ReLU has a zero gradient for negative arguments.

Leaky ReLU: Leaky ReLUs a modification of ReLU allow a small, positive gradient when the unit is not active,

$$\sigma(x) = \max(\alpha x, x) \text{ where } 0 < \alpha < 1 \tag{7.4}$$

In practice, the states are usually more than one. When a vector of states is put into a neuron, the output will become a weighted sum of inputs with a bias term. The weight increases or decreases the "strength" of the states at the neuron. As shown in Fig. 7.3, a weighted sum of each component of input vector and bias ($\sigma(\boldsymbol{W^T x} + \boldsymbol{b})$) is fed into a neuron. (To be concise, summation is hidden for input.)

For example, if $\sigma(\boldsymbol{x}) = \text{sign}(x)$, then,

$$\sigma(\boldsymbol{W^T x} + \boldsymbol{b}) = \begin{cases} +1, & \text{if } \boldsymbol{W^T x} \geq -\boldsymbol{b} \\ -1, & \text{otherwise} \end{cases} \tag{7.5}$$

The equation $\boldsymbol{W^T x} + \boldsymbol{b} = 0$ defines a hyperplane in $\mathbb{R}^n$, and divides the space into the two half-spaces (Fig.7.4). Therefore, a single neuron can separate patterns to two opposite sides. Also, the bias term serves as a threshold to shift the activation function.

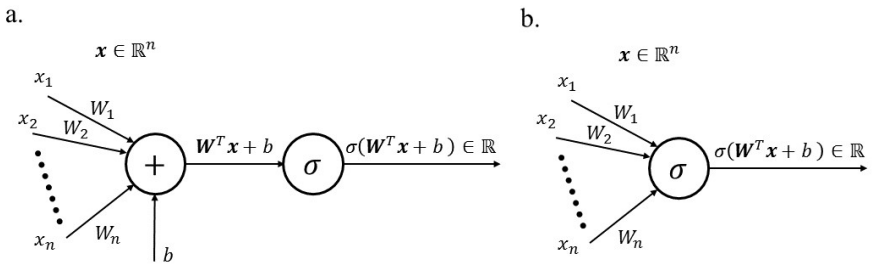

Figure 7.3: A weighted sum of vector input and bias are taken as input for neorun. a. schematic representation of vector input to neuron. b. concise schematic representation of vector input, where summation is hidden.

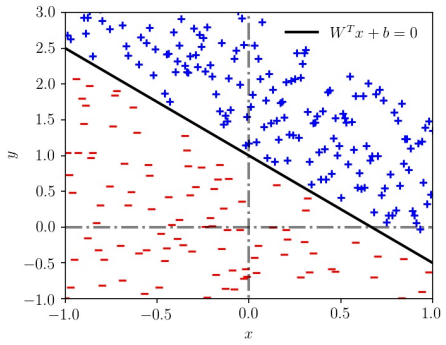

Figure 7.4: A hyperplane in 2 dimension which divides space into two distinct parts. The line seprates positive and negative points from each other.

7.1.2 Layers

Typically, neurons are aggregated into layers (Fig. 7.5). The leftmost layer of the network is called the input layer, and the rightmost layer the output layer (which, in this example, has only one neuron). Layers between them are called hidden layers because their values are not observed in the training set.

Connecting hidden layers together result in a multi-layer perceptron (MLP), which can be used to approximate many continuous functions. Note that the number of neurons in each layer can be different from other layers. Furthermore, each layer can have different nonlinear activation functions. A network is usually called shallow if the number of layer is less than three, otherwise, it's deep.

7.2 Function Approximation with a Neural Network

Let's see how we can approximate a function using a neural network.

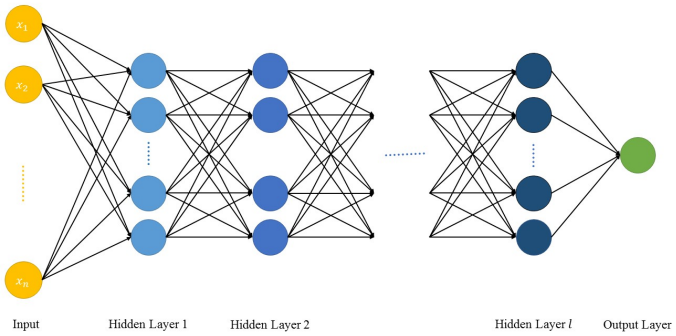

Figure 7.5: A neural network with l hidden layers and a scalar output (one node).

7.2.1 Estimators

If there is a variable called x, and let the estimate be $\hat{x}$, we can try to pick a function $\hat{f}(x)$ out of all possible functions such that it is close to real function $f(x)$,

$$\min_{f} E[\,\|x - \hat{x}\|^2\,] = E[\,\left\|f(x) - \hat{f}(x)\right\|^2\,] \tag{7.6}$$

There are too many options to choose from, Therefore, we can reduce the function set to have only $f_\theta = \theta^T x + d$, where θ and d are parameters and bias. The problem thus becomes

$$\min_{\theta} E[\,\|y - f_\theta(x)\|^2\,] \tag{7.7}$$

Here the functions in the reduced function set is exactly what a neuron does. In addition, because neural networks are just doing linear operations on neurons, we can see that they can be used to do function approximations. Furthermore, we can always increase numbers of layers and/ or neurons at each layers to get better approximation. According to universal approximation theorem, a neural network with enough layers and units can approximate any Borel measurable function from one fnite-dimensional space to another one with a small nonzero error.

7.2.2 Function Approximation

The goal here is to choose parameters of the neural network $\tilde{f}(x)$, including weights and biases for each neurons at each layers, such that the neural network approximate a given function $f(x)$ for any input.

Define weights and biases of layer l as,

1. $\boldsymbol{W}_l = [w_{l,ij}]$, where $w_{l,ij}$ is weight from x_j to neuron i.

2. $\boldsymbol{b}_l = [b_{l,i}]$, where $b_{l,i}$ is bias to neuron i.

Let's see how everything works mathematically. The output vector $\boldsymbol{y}_1$ from the first layer is just

$$\boldsymbol{y}_1 = \sigma(\boldsymbol{x}_1) \text{ where } \boldsymbol{x}_1 = \boldsymbol{W_1 y}_0 + \boldsymbol{b}_1 \tag{7.8}$$

While in the second layer, output vector $\boldsymbol{y}_1$ from the first layer influences the output vector $\boldsymbol{y}_3$ by going through linear transformation and an activation function σ element-wisely.

$$\boldsymbol{y}_2 = \sigma(x_2) \text{ where } x_2 = \boldsymbol{W_2 y}_1 + \boldsymbol{b}_2 \tag{7.9}$$

In general, we can write for layer l that

$$\boldsymbol{y}_l = \sigma(\boldsymbol{x}_l) \text{ where } \boldsymbol{x}_l = \boldsymbol{W_l y}_{l-1} + \boldsymbol{b}_l \tag{7.10}$$

Each layer is in fact a mapping from layer $l-1$ to l with weights $\boldsymbol{W}_l$ (matrix) and biases $\boldsymbol{b}_l$ (vector), with $\boldsymbol{W}_l \in \mathbb{R}^{d_{l-1} \times d_l}$ and $\boldsymbol{b}_l \in \mathbb{R}_l^d$. Where d_l is number of neurons in layer l.

In conclusion, a neural network can be written with the following equations

$$\boldsymbol{x}_k = \boldsymbol{W_k y}_{k-1} + \boldsymbol{b}_k \tag{7.11}$$

$$\boldsymbol{y}_k = \sigma(\boldsymbol{x}_k) \tag{7.12}$$

$$\text{Define } \boldsymbol{y}_0 = \boldsymbol{x}, \text{ where } \boldsymbol{x} \text{ is input data} \tag{7.13}$$

and the structure can be represented as in Fig 7.6.

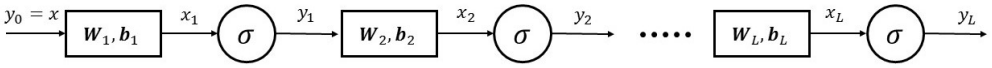

Figure 7.6: Another way to represent a neural network.

7.2.3 Solving the Problem with Gradient Descent

Again the goal is to find weights $\boldsymbol{W}_1, \boldsymbol{W}_2, ...\boldsymbol{W}_L$ and biases $\boldsymbol{b}_1, \boldsymbol{b}_2, ..., \boldsymbol{b}_L$ such that the function $f(x)$ is approximated with smallest error. To achieve this goal, we first need some data from function $f(x)$ like $(x^{(1)}, f(x^{(1)}))$, $(x^{(2)}, f(x^{(2)}))$, ..., $(x^{(N)}, f(x^{(N)}))$, then use these to solve the following minimization equation,

$$\min_{\boldsymbol{W}_l \boldsymbol{b}_l \forall l \in \{1,2,...,L\}} J = \sum_{n=1}^{N} \tilde{L}(f(x^{(n)}), y_L^{(n)}) \tag{7.14}$$

where $\tilde{L}$ is loss function, which can be defined as measure of error in approximation of function $f(x)$. For scalar function, mean squared error is used as loss function.

$$\tilde{L}(f(x^{(n)}), y_L^{(n)}) = \frac{1}{2}(f(x^{(n)}) - y_L^{(n)})^2 \tag{7.15}$$

To minimize the objective function, we can use what we learned before, gradient descent, for updating parameters in the neural network. To be more specific, we can take derivatives of J with respect to $\boldsymbol{W}_l$ and $\boldsymbol{b}_l$, and update the parameters with a learning rate ϵ (step size)

$$\boldsymbol{W}_l(t+1) = \boldsymbol{W}_l(t) - \epsilon_t \nabla_{\boldsymbol{W}_l} J(\boldsymbol{W}(t), \boldsymbol{b}(t) \tag{7.16}$$

$$\boldsymbol{b}_l(t+1) = \boldsymbol{b}_l(t) - \epsilon_t \nabla_{\boldsymbol{b}_l} J(\boldsymbol{W}(t), \boldsymbol{b}(t) \tag{7.17}$$

With updated parameters, decreasing loss function indicates more accurate approximation of our neural network. One catch here is that in each later we need to calculate derivatives, and this can be a lot of work! Luckily, there is an efficient way to do this, and it is backpropagation.

7.2.4 Backpropagation

Backpropagation is nothing but a smart way to calculate the derivatives in a neural network.

Let's try to find the relations for the derivatives at each layer. First, we can start with the last layer L,

$$\begin{aligned}\frac{\partial \tilde{L}}{\partial w_{L,ij}} &= \frac{\partial \tilde{L}}{\partial y_{L,i}}\frac{\partial y_{L,i}}{\partial w_{L,ij}} = \frac{\partial \tilde{L}}{\partial y_{L,i}}\frac{\partial y_{L,i}}{\partial x_{L,i}}\frac{\partial x_{L,i}}{\partial w_{L,ij}} \\ &= \frac{\partial \tilde{L}}{\partial y_{L,i}}\sigma' x_{L,i}\frac{\partial x_{L,i}}{\partial w_{L,ij}} = \frac{\partial \tilde{L}}{\partial y_{L,i}}\,\sigma'\, x_{L,i}\, y_{L-1,i}\end{aligned}$$

$$\begin{aligned}\frac{\partial \tilde{L}}{\partial b_{L,i}} &= \frac{\partial \tilde{L}}{\partial y_{L,i}}\frac{\partial y_{L,i}}{\partial b_{L,i}} = \frac{\partial \tilde{L}}{\partial y_{L,i}}\frac{\partial y_{L,i}}{\partial x_{L,i}}\frac{\partial x_{L,i}}{\partial b_{L,i}} \\ &= \frac{\partial \tilde{L}}{\partial y_{L,i}}\sigma' x_{L,i}\frac{\partial x_{L,i}}{\partial b_{L,i}} = \frac{\partial \tilde{L}}{\partial y_{L,i}}\,\sigma'\, x_{L,i}\end{aligned}$$

In general, we can write the derivatives in layers $1 \le l < L$:

$$\frac{\partial \tilde{L}}{\partial w_{l,ij}} = \frac{\partial \tilde{L}}{\partial y_{l,i}}\,\sigma'\, x_{l,i}\, y_{l-1,i} \tag{7.18}$$

$$\frac{\partial \tilde{L}}{\partial b_{l,i}} = \frac{\partial \tilde{L}}{\partial y_{l,i}}\,\sigma'\, x_{l,i} \tag{7.19}$$

where $\frac{\partial \tilde{L}}{\partial y_{l,i}}$ actually has connection with the next layer $l+1$ and can be derived as following

$$\begin{aligned}\frac{\partial \tilde{L}}{\partial y_{l,i}} &= \sum_k \frac{\partial \tilde{L}}{\partial x_{l+1,k}}\frac{\partial x_{l+1,k}}{\partial y_{l,i}} = \sum_k \frac{\partial \tilde{L}}{\partial y_{l+1,k}}\frac{\partial y_{l+1,k}}{\partial x_{l+1,k}}\frac{\partial x_{l+1,k}}{\partial y_{l,i}} \\ &= \sum_k \frac{\partial \tilde{L}}{\partial y_{l+1,k}}\,\sigma'\, x_{l+1,k} w_{l+1,ki}\end{aligned} \tag{7.20}$$

7.2.5 Summary of Training a Neural Network for Function Approximation

Forwardpass : using data to evaluate the network prediction values

$$x_k = \boldsymbol{W_k} y_{k-1} + \boldsymbol{b}_k \tag{7.21}$$

$$y_k = \sigma(x_k) \tag{7.22}$$

$$\text{Define } y_0 = x, \text{ where } x \text{ is input data} \tag{7.23}$$

Backwardpass : using backpropagation algorithm to calculate the derivatives efficiently

Compute $\frac{\partial \tilde{L}}{\partial y_{L,i}}$

for $l = L, L-1, ..., 1:$ do the following

$$\frac{\partial \tilde{L}}{\partial w_{l,ij}} = \frac{\partial \tilde{L}}{\partial y_{l,i}} \sigma' \, x_{l,i} \, y_{l-1,i} \tag{7.24}$$

$$\frac{\partial \tilde{L}}{\partial b_{l,i}} = \frac{\partial \tilde{L}}{\partial y_{l,i}} \sigma' \, x_{l,i} \tag{7.25}$$

where

$$\frac{\partial \tilde{L}}{\partial y_{l,i}} = \sum_k \frac{\partial \tilde{L}}{\partial y_{l+1,k}} \sigma' \, x_{l+1,k} w_{l+1,ki}$$

Update paramters : using gradient descent to update neural network parameters

$$\boldsymbol{W}_l(t+1) = \boldsymbol{W}_l(t) - \epsilon_t \nabla_{\boldsymbol{W}_l} J(\boldsymbol{W}(t), \boldsymbol{b}(t) \tag{7.26}$$

$$\boldsymbol{b}_l(t+1) = \boldsymbol{b}_l(t) - \epsilon_t \nabla_{\boldsymbol{b}_l} J(\boldsymbol{W}(t), \boldsymbol{b}(t) \tag{7.27}$$

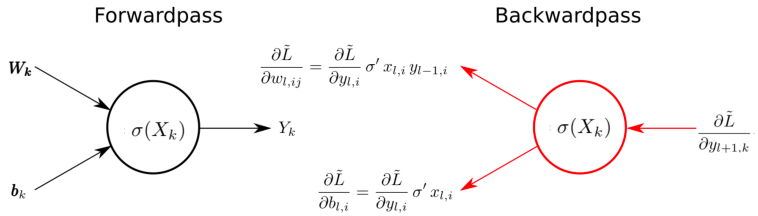

Figure 7.7: A neural network with l hidden layers and a scalar output (one node).

Fig. 7.7 shows both forward and backward passes during training of a neural network.

7.2.6 Stochastic Gradient Descent

Recall Eqn 7.26-27, we compute the gradient descent with a batch of labeled data. If we update the parameters every time we evaluate a data point, then we get update rules for stochastic gradient descent,

$$w_{l,ij}(t+1) = w_{l,ij}(t) - \epsilon_t \frac{\partial \tilde{L}(f(x^t), y_L^t)}{\partial w_{l,ij}} \tag{7.28}$$

$$b_{l,i}(t+1) = b_{l,i}(t) - \epsilon_t \frac{\partial \tilde{L}(f(x^t), y_L^t)}{\partial b_{l,i}} \tag{7.29}$$

Stochastic gradient descent converges to a local minimum under some conditions, and it is also a common way to update neural networks parameters. One application is to approximate policy in reinforcement learning which we will see soon.