Foundations of AI **Fall 2024**
Professors Greenwald and Ewing **Perceptrons and Neural Networks**

Thus far, our discussion of supervised learning has been limited to techniques designed for classification. With the introduction of artificial neural networks (ANNs), we expand the capabilities of supervised learning systems to include continuous-valued function approximation (i.e., regression). As is usual with supervised learning, the goal of learning via neural networks is function approximation. Thus, a neural network represents a function, which takes as input a vector $\vec{x} \in \mathbb{R}^n$, and output a vector $\vec{z} \in \mathbb{R}^m$. Given a data set consisting of training examples, the intent is to train the network by adjusting its weights until its approximation error is sufficiently small. In this lecture, we present ANNs as classifiers, based on threshold and sigmoidal units, as well as ANNs as general function approximators, via combinations of linear and non-linear units.

1 Neural Units

Motivated by the design of biological learning systems, artificial neural networks are complex webs of simple *neural units* interconnected in parallel structures. The connections among the units are typically described using a directed graph, with edges that are labeled by weights. Each processing unit is equipped with a very simple program, which (i) computes a weighted sum of the input data it receives from those units which feed into it, and (ii) outputs a single value, which is some function of the weighted sum of its inputs. (See Figure 1.)

Figure 1: Neural Unit.

In 1943, McCulloch and Pitts proved that ANNs are universal machines. The networks they analyzed were composed of binary threshold units, which take as input binary vector $\vec{x} = (x_1, \ldots, x_n)$ and output binary vector $\vec{z} = (z_1, \ldots, z_m)$ *s.t.*:

$$z_k = g\left(\sum_{i=1}^{n} w_{ik} x_i - \theta_k\right) \tag{1}$$

where the *transfer function* g is the step function, defined as follows:

$$g(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

In Equation 1, w_{ik} denotes the strength of the connection from the ith input to the kth output; if $w_{ik} > 0$, the connection is excitatory, if $w_{ik} < 0$, the connection is inhibitory, and if $w_{ik} = 0$, there is no connection at all.

Figure 2: McCulloch and Pitts' ANNs.

The parameter θ_k is the threshold value. This parameter can be absorbed into the summation by assuming an additional input whose value is fixed at either -1 or $+1$. In the former case, this yields $\theta_k = w_{0k}$, whereas in the latter, this yields $\theta_k = -w_{0k}$. Now

$$z_k = g\left(\sum_{i=0}^{n} w_{ik} x_i\right) \tag{3}$$

NB: We often write $z_k = g(h_k)$, where

$$h_k = \sum_{i=0}^{n} w_{ik} x_i = \vec{w}_k \cdot \vec{x} \tag{4}$$

In this lecture, we study neural units that generalize the simple binary threshold unit by (i) allowing for discrete-valued or continuous-valued inputs and outputs, and (ii) considering additional forms of the transfer function, specifically the sign, identity, and sigmoidal functions.

2 Perceptron Rule

Perceptrons are feed-forward, layered networks. Feed-forward implies that such networks form directed acyclic graphs (DAGs). An N-layered network has N layers of connections; specifically, such a network has a single input layer, and N additional layers of nodes, where the final layer is called the output layer, and the $N-1$ internal layers are called hidden layers. There are N sets of weights in an N-layered perceptron. An l-layered perceptron is depicted in Figure 3.

The simple perceptron has no hidden layers; it is a 1-layer network with only an output layer. We study a model of a simple perceptron that is composed of threshold units based on the sign function. This transfer function implies binary outputs: i.e., classification. As there is no dependence among the outputs in a simple perceptron, it suffices to consider them one at a time; notationally, we drop the dependence on k of output z, weights w_i, and weighted sum h. Now

$$g(h) = \operatorname{sgn}(h) \tag{5}$$

i.e.,

$$g(h) = \begin{cases} +1 & \text{if } h \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{6}$$

Given data set D consisting of examples of the form $\langle \vec{x}, y \rangle$, with $\vec{x} \in \mathbb{R}^n$ and $y \in \{+1, -1\}$, the perceptron rule updates weight w_i as follows:

$$\Delta w_i = \alpha (y - z) x_i \tag{7}$$

Figure 3: Perceptron.

```
PERCEPTRON($NN, D, \epsilon, \alpha$)
  Inputs      $D = (D_1, D_2)$ data set
              $\epsilon$ convergence condition
              $\alpha$ decaying learning rate
  Output      function approximation
  Initialize  weights $w_i$ randomly

while ($\Delta$ERROR($D_2$) $> \epsilon$) do
    1. for all training examples $\langle \vec{x}, y \rangle \in D_1$
        (a) $z = \mathrm{sgn}(\vec{w} \cdot \vec{x})$
        (b) if $y = z$, SKIP
        (c) else for all weights $w_i$
            i. $\Delta w_i = \alpha(y - z)x_i$
            ii. $w_i \leftarrow w_i + \Delta w_i$
    2. compute $\Delta$ERROR($D_2$)
return weights $w_i$
```

Table 1: Perceptron Rule.

given learning rate α.

The idea that underlies the perceptron learning rule is as follows: if the output of the perceptron is correct, continue; if the output of the perceptron is $z = -1$ but the target output is $y = +1$, increase the weights on all positive inputs, and decrease the weights on all negative inputs; on the other hand, if the output of the perceptron is$z = +1$ but target output is $y = -1$, decrease the weights on all positive inputs, and increase the weights on all negative inputs.

Let us argue that the prescribed update rule achieves its intended purpose. Assume input $x_i > 0$. If $y = +1$ but $z = -1$, then $y - z = 2$ is positive, so that $\Delta w_i > 0$; and if $y = -1$ but $z = +1$, then $y - z = -2$ is negative, so that $\Delta w_i < 0$. On the other hand, assume input $x_i < 0$. If $y = +1$ and $z = -1$, then again $y - z = 2$ is positive, so that $\Delta w_i < 0$; and if $y = -1$ and $z = +1$, again $y - z = -2$ is negative, so that $\Delta w_i > 0$. Finally, if $x_i = 0$, then $\Delta w_i = 0$.

Theorem [Minsky and Papert, 1969] The perceptron rule converges to weights that correctly classify all training examples, provided the given data set can be separated by a linear function.

According to this theorem, the perceptron rule is limited to learning linearly separable functions. The set of vectors $\{\vec{x} \mid \vec{w} \cdot \vec{x} = 0\}$ determines a hyperplane, a generalization of a line to multiple dimensions. Recall that $g(h) = +1$ iff $h \geq 0$ iff $\vec{w} \cdot \vec{x} \geq 0$. In other words, the network output is positive for all $\vec{x}$ that lie on the non-negative side of the hyperplane; and negative, otherwise. A function is said to be *linearly separable* if it can be separated in this way by a hyperplane.

The Boolean function AND is linearly separable, and can be represented by a simple perceptron with 2 primary inputs x_1 and x_2 with weights $w_1 = w_2 = 1$, and an additional input $x_0 = -1$ with weight $w_0 = 1.5$, for example. Similarly, the function OR is linearly separable. (**Exercise** Represent OR as a simple perceptron.) The function XOR, however, is not linearly separable, and hence cannot be represented by a simple perceptron. This realization stagnated neural network research for about a decade in the 1970s (although it was known that XOR can be represented by a multi-layer perceptron). We now proceed to consider alternative, more expressive, neural network models.

3 Gradient Descent

In this section, we study a simple perceptron composed of linear units, which learns via gradient descent. Linear units output real or discrete values, rather than simply binary values, implying that such networks are capable of general function approximation including "linear" regression. The transfer function is taken to be linear; indeed, it suffices to choose the identity function $g(h) = h$.

The goal of supervised learning in neural networks is to determine a weighting scheme that minimizes the error between network output and training examples. In classification problems, the error measure is generally taken to be the number of misclassifications. In regression, the natural measure of error is the residual sum of squares, which we denote by $E(\vec{w})$ and define as follows:

$$E(\vec{w}) = \frac{1}{2} \sum_{D} (y - z)^2 \tag{8}$$

Given data set D consisting of examples of the form $\langle \vec{x}, y \rangle$, with $\vec{x} \in \mathbb{R}^n$ and $y \in \mathbb{R}$, regression is the problem of finding the weight vector $\vec{w}$ that minimizes $E(\vec{w})$ in Equation 8. Despite the existence of a closed-form solution that solves this problem, we nonetheless describe learning via gradient descent methods for perceptrons composed of linear units in preparation for the natural extensions of this class of algorithms to multi-layer networks.

The idea underlying gradient descent is to repeatedly modify weights by taking small steps in the direction that produces the steepest descent in error, until some local minimum is reached. Mathematically, the steepest *ascent* in error is given by the gradient $\nabla E(\vec{w})$. Thus, the gradient descent update rule modifies weights in direct proportion to the *negative* of the gradient.

Given $E(\vec{w})$, the gradient $\nabla E(\vec{w})$ is the vector of partial derivatives, namely:

$$\nabla E(\vec{w}) = \left(\frac{\partial E}{\partial w_0}, \ldots, \frac{\partial E}{\partial w_n} \right) \tag{9}$$

In terms of partial derivatives, the gradient descent rule for calculating Δw_i is:

$$\Delta w_i = -\alpha \frac{\partial E}{\partial w_i} \tag{10}$$

given learning rate α. This rule simplifies as follows:

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_D (y-z)^2 \\ &= \frac{1}{2}\sum_D \frac{\partial}{\partial w_i}(y-z)^2 \\ &= \sum_D (y-z)\frac{\partial}{\partial w_i}(y-z) \\ &= \sum_D (y-z)(-x_i) \qquad (11) \end{aligned}$$

The final step in the above derivation follows from the fact that $z = g(h) = h$, and

$$\frac{\partial h}{\partial w_i} = \frac{\partial}{\partial w_i}\sum_{j=1}^{n} w_j x_j = x_i. \qquad (12)$$

Now substituting Equation 11 back into Equation 10, we obtain:

$$\Delta w_i = \alpha \sum_D (y-z)x_i \qquad (13)$$

Equation 13 is the batch learning form of an update rule known under various guises, including the Δ-rule, the Widrow-Hoff rule, and the least mean squares (LMS) rule. A batch version of gradient descent is presented in Table 2. The distinguishing feature of a batch learning algorithm is that each update to the weights depends on all examples.

An alternative to this batch variant of gradient descent is stochastic gradient descent, which uses the following update rule to approximate Equation 13:

$$\Delta w_i = \alpha(y-z)x_i \qquad (14)$$

Equation 14 adjusts weights according to the gradient of the error function on each individual training example d, namely $E_d(\vec{w}) = \frac{1}{2}(y-z)^2$.

Gradient descent converges to the global minimum in simple perceptrons of linear units, since $E(\vec{w})$ in such networks is convex. In cases in which the function $E(\vec{w})$ has multiple local minima, stochastic gradient descent sometimes avoids being trapped by non-global minima.

4 Back-Propagation

Given that simple perceptrons are limited in their expressivity, we now turn our attention to multi-layer networks. In particular, we consider multi-layer networks of (non-linear) sigmoidal units: i.e.,

$$g(h) = \frac{1}{1+e^{-h}} \qquad (15)$$

The function $g(h)$ approximates a step function in that $g(h) \sim 0$ for $h \ll 0$ and $g(h) \sim 1$ for $h \gg 0$. In addition, the sigmoidal has the nice properties of being continuous and differentiable,[1] making it amenable to gradient descent-like learning procedures, including back-propagation, which is an algorithm by which multi-layer neural networks learn weights for classification and regression.

[1] The derivative of the sigmoidal function $\sigma'(x) = \sigma(x)(1-\sigma(x))$.

```
Batch_Gradient_Descent($NN, D, \epsilon, \alpha$)
  Inputs      $D = (D_1, D_2)$ data set
              $\epsilon$ convergence condition
              $\alpha$ decaying learning rate
  Output      function approximation
  Initialize  weights $w_i$ randomly

while ($\Delta$Error($D_2$) $> \epsilon$) do
    1. for all weights $w_i$, initialize $\Delta w_i$ to 0
    2. for all training examples $\langle \vec{x}, y \rangle \in D_1$
        (a) let $z = \vec{w} \cdot \vec{x}$
        (b) if $y = z$, skip
        (c) for all weights $w_i$, increment $\Delta w_i$ by $\alpha(y - z)x_i$
    3. for all weights $w_i$, $w_i \leftarrow w_i + \Delta w_i$
    4. compute $\Delta$Error($D_2$)
return weights $w_i$
```

Table 2: Batch Gradient Descent.

```
Stochastic_Gradient_Descent($NN, D, \epsilon, \alpha$)
  Inputs      $D = (D_1, D_2)$ data set
              $\epsilon$ convergence condition
              $\alpha$ decaying learning rate
  Output      function approximation
  Initialize  weights $w_i$ randomly

while ($\Delta$Error($D_2$) $> \epsilon$) do
    1. for all training examples $\langle \vec{x}, y \rangle \in D_1$
        (a) let $z = \vec{w} \cdot \vec{x}$
        (b) if $y = z$, skip
        (c) for all weights $w_i$
             i. $\Delta w_i = \alpha(y - z)x_i$
            ii. $w_i \leftarrow w_i + \Delta w_i$
    2. compute $\Delta$Error($D_2$)
return weights $w_i$
```

Table 3: Stochastic Gradient Descent.

Using sigmoidal functions, neural networks are fully representational. Although multi-layer networks of threshold units are capable of expressing any boolean function (**Exercise** Why?), there is no analog of the perceptron rule in such networks. Hence, it is necessary to rely on the back-propagation algorithm, which depends on continuous and differentiable sigmoidal transfer functions as approximations of threshold units. Alternatively, if the sigmoidal units comprise the hidden units, but linear units comprise the output layer, such a multi-layer neural network is capable of learning any continuous and differentiable non-linear function.

Consider a multi-layer neural network with multiple outputs indexed by k. The error experienced by the network on training example d is computed as follows:

$$E_d(\vec{w}) = \frac{1}{2}\sum_k (y_k - z_k)^2 \tag{16}$$

where y_k denotes the target value of output unit k and z_k denotes the actual output. The back-propagation update rule extends Equation 10 to the case of multiple layers. Let x_{ij} denote the ith input to unit j, and let w_{ij} denote the strength of the connection between the ith input to unit j and unit j. Now

$$\Delta w_{ij} = -\alpha \frac{\partial E_d}{\partial w_{ij}} \tag{17}$$

This rule simplifies via the chain rule:

$$\begin{aligned} \frac{\partial E_d}{\partial w_{ij}} &= \frac{\partial E_d}{\partial h_j}\frac{\partial h_j}{\partial w_{ij}} \\ &= \frac{\partial E_d}{\partial h_j} x_{ij} \end{aligned}$$

since $h_j = \sum_i w_{ij} x_{ij}$. Letting

$$\delta_j \equiv \frac{\partial E_d}{\partial h_j} \tag{18}$$

yields $\Delta w_{ij} = -\alpha \delta_j x_{ij}$. It remains to compute δ_j. This computation consists of two cases, that in which j is an output unit and that in which it is not. Let us consider the former case.

Rewriting the definition of δ_j via the chain rule yields:

$$\delta_j \equiv \frac{\partial E_d}{\partial h_j} = \frac{\partial E_d}{\partial z_j}\frac{\partial z_j}{\partial h_j} \tag{19}$$

The second term in this equation is simply:

$$\begin{aligned} \frac{\partial z_j}{\partial h_j} &= \frac{\partial g(h_j)}{\partial h_j} \\ &= g(h_j)(1 - g(h_j)) \\ &= z_j(1 - z_j) \end{aligned}$$

The first term in Equation 19 reduces as follows:

$$\begin{aligned} \frac{\partial E_d}{\partial z_j} &= \frac{\partial}{\partial z_j}\frac{1}{2}\sum_k (y_k - z_k)^2 \\ &= \frac{\partial}{\partial z_j}\frac{1}{2}(y_j - z_j)^2 \\ &= (y_j - z_j)\frac{\partial}{\partial z_j}(y_j - z_j) \\ &= -(y_j - z_j) \end{aligned}$$

Finally, $\delta_j = -z_j(1 - z_j)(y_j - z_j)$ and $\Delta w_{ij} = \alpha z_j(1 - z_j)(y_j - z_j)x_{ij}$. (**Exercise** Derive $\delta_j = -y_j(1 - y_j)\sum_{j'} w_{jj'}\delta_{j'}$ where j is a hidden unit that feeds into j'.)

The back-propagation algorithm utilizes these update rules as follows:

```
BACK_PROPAGATION($NN, D, \epsilon$)
 Inputs       $D = (D_1, D_2)$ data set
              $\epsilon$ convergence condition
 Output       function approximation
 Initialize   weights $w_{ij}$ randomly

while ($\Delta$ERROR($D_2$) $> \epsilon$) do
   1. for all training examples $\langle \vec{x}, y \rangle \in D_1$
      (a) for all output units $k$
            i. compute $z_k$
           ii. $\delta_k \leftarrow z_k(1 - z_k)(y_k - z_k)$
      (b) for all hidden units $j$ that feed into units $j'$
            i. compute $z_j$
           ii. $\delta_j \leftarrow z_j(1 - z_j)\sum_{j'} w_{jj'}\delta_{j'}$
      (c) for all weights $w_{ij}$
            i. $\Delta w_{ij} \leftarrow \alpha \delta_j x_{ij}$
           ii. $w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$
   2. compute $\Delta$ERROR($D_2$)
return weights $w_{ij}$
```

Table 4: Back Propagation Algorithm.

Problems

#1 Overfitting in neural networks often results from overtraining the weights until they grow too large. One way to avoid this outcome is to train networks using an error function that penalizes large weights. For example, consider the following error function $E_d(\vec{w})$, which computes the error corresponding to training example $d = (\vec{x}, y)$, as a function of network output z:

$$E_d(\vec{w}) = \frac{1}{2}\left[(y - z)^2 + \lambda \sum_{i=1}^{n} w_i^2\right]$$

Derive the stochastic gradient descent update rule that corresponds to the given error function for a simple perceptron with a linear output unit.

#2 Consider the following data set consisting of n training examples:

$$\{(1, y_1), \ldots, (1, y_n)\}$$

All examples have attribute value 1, but there are n possible output values. To train a simple perceptron on this data set requires only one input x and one weight w. Assume such a simple perceptron with a single linear output unit.

(a) What is the sum of squared error function $E(w)$ over the given training set?

(b) Given $E(w)$ as defined in part (a), derive the gradient descent update rule.

(c) The optimal value of w (i.e., that which minimizes $E(w)$) is simply the mean of the y_i's. What value of α ensures convergence to the optimal w after just one update, regardless of the initial value of w?