EE543 - Neurocomputers and Deep Learning

Ahmet Taha Albayrak

December 2021

Chapter 1

From Biological Neuron To Artificial Neuorn Model

Chapter 2

Recurrent Neural Networks and Neurodynamis

Chapter 3

Neural Networks as Associative Memory

Chapter 4

Learning in Feed Forward Neural Networks

4.1 Perceptron Convergence Procedure

Perceptron was introduced by Frank Rosenblatt in the late 1950's with a learning algorithm. Perceptron may have continuous valued inputs. It works in the the same way as the formal artificial neuron defined previously Its activation is determined by equation:

$$\alpha = w^T u + \theta$$

Now, consider such a perceptron in N dimensional space,t he equation

$$w^T u + \theta = 0$$

that

$$w_1 u_1 + w_2 u_2 + ... + w_n u_n = 0$$

defines a hyperplane. This hyperplane devides the input space into two parts such that at one side, the perceptron has output value +1, and in the other side, it is -1.

A perceptron can be used to decide whether an input vector belongs to one of two classes say classes A and B. The decision rule may be set as to respond as class A if the output is +1 and as class B if the output is -1.The perceptron forms two decision regions seperated by the hyperplane. The equation of the boundary hyperplane depends on the connection weights and threshold

Connection weights and the threshold in a perceptron can be fixed or adapted using a number of different algorithms. The original convergence procedure developed by Rosenblatt for adjusting weights is provided in the following

1. Initialize weight and threshold
2. Present New Input and Desired output
3. Calculate actual output
4. Adapt weights
5. Repeat steps 2-4 no error occurs

In [Rosenblatt, 1959] it is proved that if the inputs presented from the two classes are linearly separable, that is they fall on opposite side of some hyperplane, then the perceptron convergence procedure always convergence in time. Furthermore, it positions the final decision hyperplane such that it separates the sample of class A from those of class B

One problem with the perceptron convergence procedure is that decision boundary may oscillate continuously when the distribution overlap or the classes are not linearly separable.

4.2 LMS Learning Rule

A modification to the perceptron convergence procedure forms the Least Mean Square (LMS) solution for the case that the classes are not separable. This solution minimizes the mean square error between the desired output and the acutal output of the processing element The LMS algorithm was first proposed for Adaline (Adative Linear Element) in [Widrow and Hoff 60].

The output function of the Adaline can be represented by the identity function as $f(a) = a$ So the output can be written in terms of input and weights as

$$x = f(a) = \sum_{j=0}^{N} w_j u_j$$

where the bias is implemented via a connection to a constant input u_0 which means the input vector and the weight vector are of space $R^{(N+1)}$ instead of R^N The output equation of Adaline can be written as: $x = w^T u$ where w and u are weight and input vectors respectively having dimension $N + 1$

Suppose that we have a set of input vectors $u^k, k = 1, 2..K$,each having its own desired output value y^k The performance of the Adaline for a given input value u^k can be defined by considering the difference between the desired

output y^k and the actual output x^k, which is called error and denoted as ε^k Therefore, the error for the input u^k is as follows:

$$\varepsilon^k = y^k - x^k = y^k - w^T u^k$$

The aim of LMS learning is to adjust the weights through a training set u^k,y^k $k = 1, 2..K$ such that the mean of square of the errors is minimum. The mean square error is defined as

$$< (\varepsilon^k)^2 >= \lim_{k\to\infty} 1/K \sum_{n=1}^{K} (\varepsilon^k)^2$$

where the notation $< . >$ denotes the mean value.

The mean square error can be written as $< (\varepsilon^k)^2 >=< (y^k - w^T u^k)^2 >=< (y^k)^2 > + w^T < u^k \times u^k > w - 2 < y^k u^{(k^T)} > w$ where T denotes transpose and $\times$ denotes outer vector product. Defining input correlation matrix R and a vector P as $R =< u^k \times u^k >=< u^k u^{(k^T)} >$ and $P =< y^k u^k >$ results in $e(w) =< (\varepsilon^k)^2 >=< (y^k)^2 > + w^T R w - 2P^T w$. The optimum value w^* for weight vector corresponding to the minimum of the mean square error can be obtained by evaluating the gradient of $e(w)$.

The point which makes gradient zeros gives us the value of w^*. That is

$$\nabla e(w)|_{w=w^*} = \frac{de(w)}{dw}|_{w=w^*} = 2Rw^* - 2P = 0$$

. Here the gradient is $\nabla e(w) = [\frac{de}{dw_1} \frac{de}{dw_2} ... \frac{de}{dw_n}]^T$ and it is a vector exteding in the direction of the greatest rate of change The gradient of a function evaluated at some point is zero if the function has a maximum or minimum at that point. The error function is of the second degree. So it is a paraboloid and it has a single minimum at point w^*.

4.3 Stepest Decent Algorithm

The analytical calculation of the optimum weight vector for a problem is rather difficult in general. Not only does the matrix manipulation get cumbersome for the large dimensions, but also each component of R and P require knowledge of the statistics of the input signal [Freeman 91]. A better approach would be to let the Adaline Linear Combiner to find the optimum weights by itself a search over the error surface.

Instead of having a purely random search, some intelligence is added to procedure such that weight vector is changed by considering the gradient of $e(w)$ iteratively [Widrow 60], according to formula know as **delta rule**. $w(t+1) = w(t) + \Delta w(t)$ where $\Delta w(t) = -\eta \nabla e(w(t))$. In the formula η is a small positive constant

For the real valued scalar function $e(w)$ on a vector space $w \in R^N$, the gradient $e(w)$ gives the direction of the steepest upward slope, so the negative of the gradient is the direction of the steepest descent. We have considered the linear output function in the derivation of the optimum weight w^* for the minimum error. However in the general case, we should consider any nonlinearity $f(.)$ at the output of the neuron. It should be noted that in such a case the error surface is no more paraboloid, so it may have several local minima. For an input u^k applied at the time t, $(e^k(t))^2$ can be used as an approximation to $< \varepsilon(k)^2 >$ where $\varepsilon^k(t)) = y^k - f(a^k) = y^k - f(w(t)^T u^k)$. Therefore, we obtain, $\nabla < (\varepsilon^k)^2 > \simeq \nabla(\varepsilon^k(t))^2 = \nabla(y^k - f(a^k)^2$ with differtiable function $f(.)$ having derivate $f'(.)$, it becomes $\nabla(y^k - f(a))^2 = -2\varepsilon^k(t) f'(a) \nabla a$

Since $\nabla a^k = \nabla w(t)^T u^k = u^k$ the weight update formula becomes $w(t+1) = w(t) + 2\eta\varepsilon^k(t) f'(a) u^k$

1. Apply an input vector u^k with the desired output value y^k to the neuron's input.
2. By considering u^k and using the current value of the weight vector determine the value of the activation vector a^k where $a^k = w(t)^T u^k$
3. Determine the value of the derivation of the output function using the current value of activation a^k that is $f'(a^k) = \frac{f(a)}{a}|_{a=a^k}$.
4. Determine the value of error $\varepsilon^k(t)$ as $\varepsilon^k(t) = y^k - f(a^k)$
5. Update the weight vector wtih respect to following update formula $w(t+1) = w(t) + 2\eta f'(a^k)\varepsilon^k(t) u^k$.
6. Repeat steps 1-5 until $< \varepsilon^k(t) >$ reduces to an acceptable level.

Notice that the iterative weight update by the **delta rule** is derived by assumingh constant u^k. Therefore, it tends to minimize the error wtih respect to applied u^k In fact, we require the average error to be minimized. This implies that

$$\frac{de}{dw_j} = \frac{1}{K} \sum_{k=1}^{K} \frac{d(\varepsilon^k)^2}{dw_j} = \frac{1}{K} \sum_{k=1}^{K} \frac{2\varepsilon^k d\varepsilon^k}{dw_j}$$

Therefore, the net change in w_j after one complete cycle of pattern presentation is expected to be

$$w_j(t+K) = w_j(t) - \eta \frac{1}{K} \sum_{k=1}^{K} \frac{2\varepsilon^k d\varepsilon^k}{w_j}$$

However, this would be true that if the weights are not updated along a cycle but only at the end. By changing the weights as each pattern is presented, we depart to some extend from gradient descent in e Nevertheless, provided the learning rate is sufficiently small, this departure will be negligible and **delta rule** will implement a very close approximation to gradient descent in mean squared error [Freeman 91].

4.4 The Backpropagation Algorithm: Single Layer Network

Consider a single layer multiple output network. Here, we stil have N inputs denoted u_j, $j = 1...N$ but M processing elements whose activation and outputs denoted as a_i and x_i, $i = 1...M$ respectively w_{ji} used to denote the strength of the connection from the j^{th} input to the i^{th} processing element. In vector notation w_{ji} is the j^{th} component of weight vector w_i, while u_j is the j^{th} component of the input vector u

Let u^k and y^k to represent the k^{th} input sample and the corresponding desired output vector respectively. Let the error observed at the output i, when u^k is applied at the input, be $\varepsilon_i^k = y_i^k - x_i^k$. If the error is to be written in terms of the input vector u^k and the weights w_j, we obtain $\varepsilon_i^k = y_i^k - f(w_i^T u^k)$.

If we take the partial derivative with respect to $w_j i$ by applying the chain rule

$$\frac{d\varepsilon_i^k}{dw_{ji}} = \frac{d\varepsilon_i^k}{dx_i^k} \frac{dx_i^k}{dw_{ji}}$$

where

$$\frac{d\varepsilon_i^k}{dx_i^k} = -1 \; and \; \frac{dx_i^k}{dw_{ji}} = f'(a_i^k) u_j^k$$

We obtain

$$\frac{d\varepsilon_i^k}{dw_{ji}} = -f'(a^k) u_j^k$$

If we define the total output error for input u^k as the sum of the square of errors at each neuron output, then partial derivate of the total error with

respect to w_{ji}, when u^k is applied at the input can be written as

$$\frac{de^k}{dw_{ji}} = \frac{de^k}{d\varepsilon_i^k}\frac{d\varepsilon_i^k}{dw_{ji}}$$

which is

$$\frac{de^k}{dw_{ji}} = -\varepsilon_i^k f'(a^k)u_j$$

by defining

$$\delta_i^k = \varepsilon_i^k f'(a^k)$$

it can be formulated as

$$\frac{de^k}{dw_{ji}} = -\delta_i^k u_i^k$$

For the error to be minimum the gradient of the total error with respect to weights should be 0. In order to reach the minimum of the total error, without solving the above equation, we can apply **delta rule** in the same way explained for the steepest descent algorithm

$$w_{ji}(t+1) = w_{ji}(t) + \eta\delta_i^k u_j^k \; for j = 1..N, i = 1..M$$

4.5 The Backpropagation Algorithm: Multi Layer Network

Now, assume that another layer of neurons is connected to the input side of the output layer. Therefore we have input, hidden and the output layer. In order to discriminate between the elements of the hidden and output layers We will use the subscripts L and o respectively. Furthermore, we will use h as the index on the hidden layer elements, while still using index j and i for the input and output layers respectively

In such a network, the output value of i^{th} neuron of the output layer can be written as

$$x_{i_o}^k = f_o(w_{i_o}^T x_L^k)$$

where x_L^k being the vector of the output values at hidden layer that is connected as input to the output layer. The value of the h^{th} element x_L^k is determined by the equation

$$x_{h_L}^k = f_L(w_{h_L}^T u^k)$$

The partial derivate of the output of a neuron i_o of output layer wtih respect the hidden layer weight w_{jhL} can be determined by applying chain rule

$$\frac{dx_{i_o}^k}{dw_{jh_L}} = \frac{dx_{i_o}^k}{dx_{h_L}^k}\frac{dx_{h_L}^k}{dw_{jh_L}}$$

We can rewrite the equation as

$$\frac{dx_{i_o}^k}{dw_{jh_L}} = (f_o'(a_{i_o}^k)w_{h_L i_o})(f_L'(a_{h_L}^k u_j^k))$$

Then the partial derivate of the total error with respect to hidden layer w_{jh_L} can be written as

$$\frac{de^k}{dw_{jh_L}} = -\sum_{i_0=1}^{M} \varepsilon_{i_o}^k f_o'(a_{i_o}^k) w_{h_L i_o} f_L'(a_{h_L}^k) u_j^k$$

Therefore, the weight update rule for the hidden layer

$$w_{jh_L}(t+1) = w_{jh_L}(t) - \eta\frac{de^k}{dw_{jh_L}}$$

can be reformulated by using the same analogy in the single layer network section

$$w_{jh_L}(t+1) = w_{jh_L}(t) + \eta\delta_{h_L}^k u_j$$

1. **Initialize weights:** to small random values
2. **Apply a sample:** apply to the input a sample vector u^k having desired output vector y^k
3. **Forward Phase:** Starting from the first hidden layer and propagating towards the output layer
   - 3.1. **Calculate the activation values** for the units at layer L
   - 3.2. **Calculate the output values** for the units at layer L
4. **Output erros:** Calculate the error at the output layer as
$$\delta_{i_o}^k = (y_{i_o}^k - x_{i_o}^k) f_o'(a_{i_o}^k)$$
5. **Backward Phase** Propagate error backward to the input layer through each layer L using the error term
$$\delta_{h_L}^k = f_L'(a_{h_L}^k) \sum_{i_{L+1}=1}^{N_{L+1}} \delta_{i_{(L+1)}}^k w_{h_L i_{(L+1)}}^k$$

6. **Weight update:** Update the weights according to the formula

$$w_{j_{(L-1)h_L}}(t+1) = w_{j_{(L-1)h_L}}(t) + \eta \delta^k_{h_L} x^k_{j_{(L-1)}}$$

7. **Repeat** steps 1-6 until the stop criterion is satisfied.

Chapter 5

Deep Learning

5.1 Deep Learning Motivation

Patter recognition task contains 3 steps such as **Feature Extraction**, **Dimension Reduction** and **Classification**. In Feature extraction steps, it requires expertise to decide on which feature to use. In Dimension reduction step, learning is harder if the dimension of the feature vector is high. In classification step, we can use a simple fully connected network (shallow). We can skip feature extraction and dimension reduction steps by using Fully Connected Networks. Also, we can use Deep Neural Networks for all steps.

Motivation of Deep Learning is optimization problem, shallow networks capacity, unsupervised learning.

5.1.1 Problems In Deep MLP (Fully Connected Deep Network)

1. Training takes long time due to number of parameters.
   Solution: Convolutional Neural Network(CNN): local connection and weight sharing

2. Need lots of labeled data
   Solution: Auto Encoder, Restricted Boltzman Machine

3. Partial Derivatives at lower layer fades away in backpropagation phase.
   Solution: Stacked Auto Encoder, Deep RBM

5.2 Deep Learning Overview

Multiple layer works to build improved feature space such that first layer learns 1^{st} order features (e.g edges, lines ..) Second layer learns higher order features like combination of first layer (e.g square, rectangle), further layer can combine those filters to get more sophisticated features like face, hand. Usually Deep learning results best when input space is locally structured (spatial or temporal) (e.g image, language)

In current models layer often learns in an unsupervised mode and discover general features of the input spcae - serving multiple tasks related to the unsupervised instances Then final layer features are fed into supervised layers and entire network is often subsequently tuned using supervised training of the entire net, using the initial weightings learned in the unsupervised phase. Also, you can do fully supervised version.

5.3 Convolutional Neural Networks (CNN)

CNN is firstly proposed by Fukushima(1980) (Neo-Cognitron) and LeCun (1998) (Convolutional Neural Networks). The common characteristics of CNN are:

1. A special kind of multi-layer neural networks.
2. Implicityly extract relevant features.
3. A feed-Forward network that can extract topological properties from a structured data (e.g image)
4. Like almost every othe neural networks, CNN are trained with a version of the backpropagation algorithm.

Convolution layer uses filters (i.e locally connected and shared weight structure resulting in a much smaller number of parameters due to local connection and weight sharing in filters Locally connected shared weights results in translation equvariance, if you shift input m units output shifts accordingly.

Convolution (1D):

$$s(t) = (x * w)(t)$$

continuous case:

$$s(t) \int x(a)w(t-a)\,da$$

discrete case:

$$s[t] \sum_{a=-\infty}^{\infty} x[a]w[t-a]$$

5.3.1 Weight update for shared weights

Let $w_{(i+k,j+k)k=0,1..M}$ be shared weights the n to constrain $w_{(i+k_1,j+k_1)} = w_{(i+k_2,j+k_2)}$ where $k_1, k_2 = 0, 1..M-1$. We need that $\Delta w_{(i+k_1,j+k_1)} = \Delta w_{(i+k_2,j+k_2)}$ in the training. For this purpose, we will use

$$\Delta w_{(i+k_1,j+k_1)} = \Delta w_{(i+k_2,j+k_2)} = \frac{1}{m} \sum_{k} \frac{de}{dw_{i+k,j+k}}$$

Remarks:

1. CNN may be used either for classification or regression similar to MLP. For classification, using softmax at the last layer increases the performance.
2. Instead of using convolutional layer with large sized filter, use many filter with small filters.
3. Zero padding may be used in order to feature map to have the same size as the previous layer after filtering
4. Due to memory limitations, instead the whole training set, use minibatch selected randomly from training set, then change the minibatch used after updating the weights
5. Weight initization is important. (e.g Xavizer initiliser)
6. Use Adam optimizer insted of classical gradient descent
7. Over-fit occurs if the dataset is not large enough. In order to increase **generalisation ability**, you may reduce the depth of the network in order to decrease the number of parameters but there are some other solution
   - 7.1. **Data augmentation:** in order to increase training set, produce artificial sample (e.g mirror symmetry)
   - 7.2. **Drop-out:** At each iteration, arbitrarily select some neuron and set their output to 0,
   - 7.3. **Transfer Learning:** use a CNN trained on a large dataset as initial weights

5.4 Stacked Auto-Encoders

Stacked Auto-Encoders is proposed by Bengio (2007) after Deep Belief Networks (Hinton,2006). We can stack many (sparse) auto-encoders in succession and them using greedy layer-wise training and each iteration drop the decode output layer.

Auto-Encoders will often do a dimensionality reduction like PCA or non-linear dimensionality reduction. This leads to a "dense" repesentation which is nice terms of use of resource (i.e number of nodes). All features typically have non-zero values for any input and the combination of values contains compressed information.

However, this distributed repesentation can often make it more difficult for successive layers to pick out the salient featuer Sparse repesentation usees more feature where at any given time a significant number of the features will have a 0 value. Thisd leads to more localist veriable length encodings where a particular node with value 1 signifies the presence of a feature. This is easier for subseqeunt layers to use for learning For sparse encodings, use more hidden nodes in the encoder and use regularization techniques which encourge sparseness

There is another auto-encoders type which is de-noising auto-encoder. De-noising Auto-Encoder stochastically corrupt training instance each time, but still train auto-encoder to decode the uncorrupted instance, forcing it to learn conditional dependencies within the instance. De-noising Auto-Encoder yield better empirical result and handles missing values well