UNIVERSITY OF MARY
SCHOOL OF ENGINEERING
ENR 338
Advanced Engineering Mathematics

Lecture AI: Artificial Intelligence

Introduction

In this lecture we would like to write our own Neural Network. We will learn the mathematics of neural networks and back propagation and then codify the math into a computer program, written in the C programming language, and test it out to see if it works. The goal is a sort of "hello world" for neural networks and artificial intelligence. We will not be doing anything sophisticated but, instead, we will show in detail how it all works and the steps to writing the code. This will give our engineering students the advantage of knowing "what's under the hood" in every artificial intelligence application they find in the world at large. The main difference between the ones you find out there and the one you will write yourself here is scale. The industrial applications of neural networks have way more layers, way more neurons per layer, and are trained on way more data. But otherwise they are pretty much the same.

The history of artificial intelligence (AI) dates back to the 1940s. The term "artificial intelligence" was coined in 1956 during a workshop at Dartmouth College. Early AI research focused on symbolic AI, where computer programs used rules and logic to simulate human reasoning.

In the 1960s and 1970s, AI research saw significant progress in areas like problem-solving, natural language processing, and machine learning. However, funding decreased in the late 1970s, leading to what became known as the "AI winter" – a period of reduced interest and investment in AI research.

The field regained momentum in the 1980s with advancements in expert systems and knowledge-based systems. The 1990s saw the rise of statistical methods and machine learning algorithms, paving the way for practical applications like speech recognition and data analysis.

In the 2000s, with the growth of the internet and access to vast amounts of data, AI applications expanded into areas like recommender systems, virtual personal assistants, and more. Then, in the 2010s, deep learning, a subset of machine learning, emerged as a dominant approach, achieving breakthroughs in image recognition, natural language processing, and game playing.

Today, AI is ubiquitous in our lives, powering technologies like self-driving cars, personalized recommendations, and medical diagnosis systems, among others. The field continues to evolve rapidly, driven by ongoing research, improved algorithms, and increasing computational power.

Here's a list of some of the important uses for artificial intelligence that are actively being used today:

- Natural Language Processing (NLP): AI is used in NLP to enable machines to understand, interpret, and generate human language. Applications include chatbots, virtual assistants, sentiment analysis, and language translation.
- Image and Video Analysis: AI-powered computer vision systems can analyze images and videos, enabling applications like facial recognition, object detection, and autonomous vehicles.
- Recommendation Systems: AI algorithms power recommendation engines in online platforms, suggesting products, movies, music, and content tailored to individual preferences.
- Healthcare Diagnosis and Imaging: AI is utilized in medical imaging and diagnosis, assisting healthcare professionals in detecting diseases and interpreting medical images more accurately.
- Financial Services: AI is applied in fraud detection, algorithmic trading, credit scoring, and customer service within the financial industry.
- Autonomous Vehicles: AI plays a crucial role in self-driving cars, helping them perceive the environment, make decisions, and navigate safely.
- Gaming and Entertainment: AI is used in creating intelligent non-player characters (NPCs), adaptive game difficulty, and game testing.
- Virtual Assistants: AI-powered virtual assistants like Siri, Google Assistant, and Alexa help users with tasks, queries, and scheduling.
- Smart Home Devices: AI is used in smart home devices to automate tasks and enhance energy efficiency.
- Social Media: AI algorithms analyze user behavior and content to personalize feeds and target advertisements.
- Cybersecurity: AI assists in identifying and mitigating cybersecurity threats, detecting anomalies, and preventing attacks.
- Language Translation: AI-powered translation tools offer real-time language translation for communication across language barriers.
- E-commerce: AI enhances customer experience through personalized recommendations, chatbots, and supply chain optimization.
- Drug Discovery: AI accelerates drug discovery by simulating molecular interactions and predicting potential drug candidates.

- Speech Recognition: AI enables accurate speech recognition in voice assistants, transcription services, and voice-controlled devices.

These are just a few examples, and AI's applications continue to expand into various industries and domains, transforming the way we live and work.

Neural Networks

Neural networks play a crucial role in artificial intelligence and are a subset of machine learning algorithms. They are a computational model inspired by the structure and functioning of the human brain. In artificial intelligence, neural networks are used to process complex data and learn patterns from it, enabling the system to make decisions, recognize patterns, and perform tasks without being explicitly programmed for each step.

Neural networks are designed to learn from examples and adapt their internal parameters to make predictions or decisions based on new, unseen data. This learning process is fundamental in AI applications to improve performance over time. Neural networks also excel at recognizing patterns in data. They can identify objects in images, understand speech, translate languages, and perform various other tasks where pattern recognition is essential.

Deep learning is a specialized form of neural networks with multiple layers. These deep neural networks have achieved remarkable success in image and speech recognition, natural language processing, and game playing.

Mathematics of Neural Networks

Let's now turn to the gruesome details instead of just the advertising slogans. We want to write a neural network in the C programming language but, in order to do this, we first have to understand in detail the mathematics involved and then we can convert the math equations into computer code.

A neural network consists of layers, neurons, transition functions, and bias functions defined as follows:

- $(T+1)$ layers indexed by $t \in \{0, 1, \cdots, T\}$. In other words, we start the 'time' t at zero and end at time T in steps of 1. We start at $t = 0$, rather than 1, to make the transition to the C code easier later. The value of each neuron in the layer is denoted by $X^\mu(t)$. So for example, the 7th neuron in the 4th layer would have the value $X^6(3)$ since $\mu = 6$ is the 7th neuron (they start at $\mu = 0$) and $t = 3$ is the 4th layer (they start at $t = 0$). We will call these 'neuron 6' and 'layer 3' so that 'layer 3' (when $t = 3$) is actually the 4th layer but indexed by the number 3. Similarly 'neuron 6' means the 7th neuron, following neuron 0, neuron 1, neuron 2, neuron 3, neuron 4, and neuron 5. Note that, oftentimes, the zeroth layer, is called the 'input layer' and the rest are called 'hidden layers'. Hence a network with 5 hidden layers would have layers indexed by $t = 0, 1, 2, 3, 4, 5$.
- $N(t) + 1$ neurons in each layer indexed by $\mu \in \{0, 1, \cdots, N(t)\}$. The reason we let N be a function of t is so that we can choose to have a different number of neurons in each layer if we like. Note that the index μ also starts at zero, just like t did, to make it easier to code later.
- The transition functions are denoted $R^\mu_\nu(t+1, t)$ indexed by $\mu \in \{0, 1, \cdots, N(t+1)\}$ and $\nu \in \{0, 1, \cdots, N(t)\}$ and are functions of the layer t and the next layer $t+1$ where $t \in \{0, 1, \cdots, T-1\}$. Note that since the transition functions are functions of the current layer and the next layer, and the variable t only goes up to T, they can only go up to $t = T-1$ so that the final transition function will be $R^\mu_\nu(T, T-1)$.
- The bias functions $B^\mu(t)$ are indexed by $\mu \in \{0, 1, \cdots, N(t)\}$ and $t \in \{1, 2, \cdots, T\}$. In other words, they start with $B^\mu(1)$ in layer one (the layer zero is $t = 0$) and end with $B^\mu(T)$ in the last layer.

We note that 'layer 0', at $t = 0$, does not have bias functions associated with it. This is the input layer. The entire network is a machine that takes inputs and produces outputs. We have defined the inputs as layer $t = 0$ and the outputs as layer $t = T$.

When you are 'training' the neural network you have a desired output that you would like the network to produce. For example, you may have a collection of handwritten numbers, each labeled by the number they are supposed to show. You would put the pictures in as the input and the network would generate numbers as the output. You would then compare the numbers that it generated with the correct answers from the labels to see how well the network did. We will use the special label Y^μ to denote the 'correct answer' when we are training the network. We will compare the output of the network: $X^\mu(T)$ with the correct answer Y^μ for each value of μ to see how accurate our network is.

The transition and bias functions $R^\mu_\nu(t+1, t)$ and $B^\mu(t)$ are the adjustable parameters of the network. You adjust them with a set of training data until your network is getting the answers correct to the desired accuracy and then your network is 'trained'. You can then feed it data for which the answer is unknown and have a certain degree of trust that the answers that come out are the correct ones even though you no longer have a known correct answer to check it with.

Exercises

1. How many adjustable parameters are there in a neural network with 5 hidden layers and 4 neurons per layer?
2. What is the general formula for a 'square' network with T hidden layers and N neurons per layer?
3. What is the general formula, using a Riemann summation, for a general neural network with T hidden layers and $N(t)$ neurons in layer t? Hint: find an equation for the number of transition functions, an equation for the number of bias functions, and then add them together.
4. GPT-4 has 120 layers and 100 billion neurons. How many parameters does it have?
5. The human brain has about 86 billion neurons, which is less that that of GPT-4. One might assume that GPT-4 should be smarter than we are. However, in most ways it is not. What are some of the most significant differences between the human brain and GPT-4 that could account for the fact that we are conscious and GPT-4 isn't? Defend your answers to be convincing since there is no consensus on what are the "right" answers here.

Forward Propagation

Starting with a given input and given network parameters and then computing what comes out the other side is called forward propagation. The way we compute it is by first setting layer zero ($t = 0$) to the input values and then we compute $X^\mu(t)$ for each neuron and all layers in the network, sequentially, until we reach the output layer. The following is the formula that is used for forward propagation

$$X^\mu(t+1) = \sigma\left(R^\mu_\nu(t+1,t)X^\nu(t) + B^\mu(t+1)\right)$$

where $\sigma(z)$ is a particular non-linear function of its argument. Note that we are using the Einstein summation convention here. The convention is that, whenever an index appears twice in any single term in an expression – once upper and once lower, that index is summed over all of it's possible values. So, since the above expression has the index ν repeated once upper and once lower, the above expression is merely short form for the following

$$X^\mu(t+1) = \sigma\left(\sum_{\nu=0}^{N(t)} R^\mu_\nu(t+1,t)X^\nu(t) + B^\mu(t+1)\right)$$

in other words:

$$X^\mu(t+1) = \sigma\Big(R^\mu_0(t+1,t)X^0(t) + R^\mu_1(t+1,t)X^1(t) + \cdots + R^\mu_{N(t)}(t+1,t)X^{N(t)}(t) + B^\mu(t+1)\Big)$$

As you can see, the value of the μth neuron in layer $t+1$ is found by summing up contributions from all of the neurons in layer t followed by the bias function of layer $t+1$ and then sticking the total sum into the nonlinear function σ.

Although this expression may look complicated it is quite easy to implement on a computer. Let us make it a bit more clear by computing a simple example.

Example: Forward Propagation

Suppose we have a neural network with 2 hidden layers ($t = 0, 1, 2$), 2 neurons in each layer ($\mu, \nu = 0, 1$), transition and bias functions given by

$$\begin{aligned}
R^0_0(1,0) &= 1\\
R^0_1(1,0) &= 2\\
R^1_0(1,0) &= 3\\
R^1_1(1,0) &= 4\\
R^0_0(2,1) &= 5\\
R^0_1(2,1) &= 6\\
R^1_0(2,1) &= 7\\
R^1_1(2,1) &= 8\\
B^0(1) &= 1\\
B^1(1) &= 2\\
B^0(2) &= 3\\
B^1(2) &= 4
\end{aligned}$$

and initial values given by

$$\begin{aligned}
X^0(0) &= 10\\
X^1(0) &= 10
\end{aligned}$$

We can calculation the values of layer 1 from those of layer 0 as follows:

$$X^\mu(1) = \sigma\left[R^\mu_0(1,0)X^0(0) + R^\mu_1(1,0)X^1(0) + B^\mu(1)\right]$$

which gives

$$\begin{aligned}
X^0(1) &= \sigma\left[R^0_0(1,0)X^0(0) + R^0_1(1,0)X^1(0) + B^0(1)\right]\\
X^1(1) &= \sigma\left[R^1_0(1,0)X^0(0) + R^1_1(1,0)X^1(0) + B^1(1)\right]
\end{aligned}$$

and, putting in the numbers, we find

$$\begin{aligned}
X^0(1) &= \sigma\left[10 + 20 + 1\right] = \sigma(31)\\
X^1(1) &= \sigma\left[30 + 40 + 2\right] = \sigma(72)
\end{aligned}$$

A common non-linear function that is often used is the logistic, or sigmoid, function: $\sigma(z) = 1/(1+e^{-z})$. If we assume this function in our answer for layer 1 is:

$$X^0(1) = \sigma(31) = \frac{1}{1+e^{-31}} = 1$$
$$X^1(1) = \sigma(72) = \frac{1}{1+e^{-72}} = 1$$

Notice that all of the neurons in layer 1 end up with values of 1. This is simply because the exponential in the logistic function is vanishingly small for the initial values that we have chosen. Let us continue and use these values for layer 1 to find the values for layer 2:

$$X^0(2) = \sigma\left[R_0^0(2,1)X^0(1) + R_1^0(2,1)X^1(1) + B^0(2)\right]$$
$$X^1(2) = \sigma\left[R_0^1(2,1)X^0(1) + R_1^1(2,1)X^1(1) + B^1(2)\right]$$

and, putting in the numbers, we find

$$X^0(2) = \sigma\left[5\cdot 1 + 6\cdot 1 + 3\right] = \sigma(14)$$
$$X^1(2) = \sigma\left[7\cdot 1 + 8\cdot 1 + 4\right] = \sigma(19)$$

which is

$$X^0(2) = \frac{1}{1+e^{-14}} = 1$$
$$X^1(2) = \frac{1}{1+e^{-19}} = 1$$

Even values of 14 and 19 are still too large for the logistic function to return anything significantly different from 1. In any case this example should show how these computations are accomplished. With a computer program it is a simple collection of `for` loops.

Now that we have seen how forward propagation works we can already write a perfectly working neural network in that it can take input data and produce output data. Here is the C code corresponding to the forward propagation example that we gave above. You should type it in, compile it, and verify that the output is what we computed by hand in our example.

forward.c

```
#include <stdio.h>
#include <math.h>

#define N 2 // num neurons
#define T 2 // num hidden layers

double sigma(double x);
int main (void){
 double X[N][T+1],Z[N][T+1];
 double R[N][N][T+1],B[N][T+1];
 int mu, nu;

 R[0][0][0] = 1;
 R[0][1][0] = 2;
 R[1][0][0] = 3;
 R[1][1][0] = 4;
 R[0][0][1] = 5;
 R[0][1][1] = 6;
 R[1][0][1] = 7;
 R[1][1][1] = 8;
 B[0][1] = 1;
 B[1][1] = 2;
 B[0][2] = 3;
 B[1][2] = 4;
 X[0][0] = 10;
 X[1][0] = 10;

 // forward propagate

 for (int t = 0; t < T; t++){
  for (mu = 0; mu < N; mu++){
   Z[mu][t+1] = B[mu][t+1];
   for (nu = 0; nu < N; nu++){
    Z[mu][t+1] = Z[mu][t+1] + R[mu][nu][t
        ]*X[nu][t];
   }
   X[mu][t+1] = sigma(Z[mu][t+1]);
   fprintf(stderr,"X[%d][%d] = %lf\n", mu
        , t+1, X[mu][t+1]);
  }
 }
 return 0;
}
double sigma(double x){
 return 1/(1+exp(-x));
}
```

Notice that, while this network works, it is pretty useless. We simply chose arbitrary transition and bias functions and so the result is a network that returns all 1's. We could use it do decide whether the input is really big or really small I suppose. However, that misses the point. We don't just create networks arbitrarily and then figure out what they are good for. Instead, the way that we create a useful neural network is by *training it* to solve a particular problem that we are interested in. We will now discuss this training procedure.

Training the Network

In order to find the correct parameters (i.e. transition functions and bias functions) which will allow us to solve specific problems we need to train the network. We use a set of known inputs for which the answers are already known. These could be images along with the descriptions of what they are – such as handwritten numerals and their digital values, or pictures which may or may not contain fire hydrants, etc. We take this data set of solved problems and we feed them into our network. Then we compare the output with the desired answers and, if they are not the same to within a specified tolerance, we adjust the parameters of the network until they are.

Once the network has been adjusted, or trained, to get the correct answers for all of the training data we can then use it to provide answers for data for which the answer is not known.

To accomplish this training we need a way of comparing the output of the network to the desired output. Suppose, in our example in the previous section, we desired our output to be

$$\begin{aligned} Y^0 &= 1 \\ Y^1 &= 0.5 \end{aligned}$$

The answer we got was

$$\begin{aligned} X^0(2) &= 1 \\ X^1(2) &= 1 \end{aligned}$$

Is this close to the correct answer? One of the outputs is the same as the corresponding correct output but the other is not. Is it close enough? In order to define the tolerance of the network – i.e. whether or not the output is "close enough" to the desired output, we define a *Cost Function.* We then consider the network to be trained when the value of the cost function is minimized across the training data.

This is similar to the way that least squares curve fitting works. You have a set of data points and you try to fit them to a function. The way you do the fit is to find minimize the function that adds up the differences squared. That is why it is called a least squares fit. Here is an example of a least squares cost function:

$$C = \frac{1}{2} \sum_{\mu=0}^{N(T)} \left[X^\mu(T) - Y^\mu\right]\left[X_\mu(T) - Y_\mu\right]$$

This is a function of the last layer in the network and, for each neuron in the layer, it adds the squared difference with the corresponding value in the desired answer. For example, our example in the previous section would give the following cost function:

$$\begin{aligned} C &= \frac{1}{2}\Big(\left[X^0(2) - Y^0\right]\left[X_0(2) - Y_0\right] \\ &\quad + \left[X^1(2) - Y^1\right]\left[X_1(2) - Y_1\right]\Big) \\ &= \frac{1}{2}\Big([1-1]\,[1-1] + [1-0.5]\,[1-0.5]\Big) \\ &= \frac{1}{2}\Big(0 - [0.5]\,[0.5]\Big) \\ &= \frac{1}{2}\Big(0.25\Big) \\ &= 0.125 \end{aligned}$$

If we then changed our parameters to a new set of values and ran our network again, we would get a different value for the cost function. The next one may be smaller than this and, in that case, we would say that the new parameters are better than the old ones for solving the problem.

So the cost function will allow us to test whether one set of parameters is better or worse than another but how do we adjust them to get better ones? Just picking random parameters and checking the cost function each time would be a time-consuming and possibly even fruitless task. We would much rather have a way of adjusting the parameters to get smaller and smaller values of the cost function until we find a set of parameters that minimizes it. Then we would know that, since we are at a minimum value of C, any further changes in the parameters will only increase C and so we have found the best set of parameters possible without changing the structure of the network. Of course, if the minimum value is still not good enough to solve the problems we would like it to solve we are free to add more neurons, more layers, or both and then try again with the new network structure.

The way to train the network in tiny increments to achieve smaller and smaller values of the cost function until we reach it's minimum value is called the method of steepest descent, or *gradient descent.* This method adjusts each of the parameters across the network according to its effect on the cost function and the adjustment is made in the direction of the negative rate of change of the cost function with respect to that parameter. The parameter is adjusted by a tiny fixed amount that we define as ϵ. We call this the *learning rate.* Note that this is exactly the same thing that we do in normal calculus when we are finding the minimum of a function. We travel in the direction of the negative gradient of the function with respect to position. The gradient of a function points in the direction of fastest increase of the function and so the negative gradient will point in the direction of fastest decrease. If we continue to travel by small amounts in the direction of fastest decrease we will eventually arrive at a local minimum of the function. Note that we say "local minimum" since there may be several minima of a given function and this method will only find one of them. The one that it finds may not be the global or absolute minimum of the function but it will be a local minimum.

We perform the adjustments to the parameters in the following way:

$$\begin{aligned} \delta B^\mu(t) &= -\epsilon \frac{\partial C}{\partial B_\mu(t)} \\ \delta R^\mu_\nu(t+1,t) &= -\epsilon \frac{\partial C}{\partial R^\nu_\mu(t+1,t)} \end{aligned}$$

Notice the order of the indices in the quantities on the right. We do this so that upper indices and lower indices are consistent on both the left and right hand sides of these equations. An upper index in the denominator of an expression is equivalent to a lower index in the numerator. Recall that our summation convention requires an upper index be matched with a lower index to trigger a summation and therefore we need to be careful that our expressions preserve the distinction between upper and lower.

The above expressions say that we adjust our parameters via

$$\begin{aligned} B^\mu(t) &\longrightarrow B^\mu(t) + \delta B^\mu(t) \\ R^\mu_\nu(t+1,t) &\longrightarrow R^\mu_\nu(t+1,t) + \delta R^\mu_\nu(t+1,t) \end{aligned}$$

where the change is in the direction of the negative gradient of the cost function with respect to the parameter and scaled by the value of the learning rate ϵ.

We perform this adjustment over and over until the cost function has fallen below some predefined minimum threshold.

The question now is this: How do we find this negative gradient for each of the parameters in the network? Recall that the cost function is given by

$$C = \frac{1}{2}\sum_{\mu=0}^{N(T)} \left[X^\mu(T) - Y^\mu\right]\left[X_\mu(T) - Y_\mu\right]$$

Notice that none of the parameters of the network appear in this formula! There are no transition functions and no bias functions in C and hence, taking the derivative with respect to them would seem to give zero.

Well, in fact, the transition functions and biases **are** contained in this formula. They are just hidden inside the $X^\mu(T)$. Recall that the forward propagation formula shows how to get $X^\mu(T)$ from the values of $X^\mu(T-1)$ using a combination involving the transition functions and biases. So we can compute these derivatives by using the chain rule of calculus. This is called *back propagation.*

Back Propagation

Back propagation is the method that we use to find the negative gradient of the cost function for each of the parameters in the network. It uses the fact the the output values of the neurons are actually functions of all of the parameters in the network and so one can use the chain rule to take the derivatives.

Let's use our simple network as an example. Recall the formula for doing forward propagation.

$$X^\mu(t+1) = \sigma\left(R^\mu_\nu(t+1,t)X^\nu(t) + B^\mu(t+1)\right)$$

In the case were $t = T-1$ we get

$$X^\mu(T) = \sigma\left(R^\mu_\nu(T,T-1)X^\nu(T-1) + B^\mu(T)\right)$$

and so our formula for the cost function becomes

$$\begin{aligned} C = \frac{1}{2}\sum_{\mu=0}^{N(T)} &\left[\sigma\left(R^\mu_\lambda(T,T-1)X^\lambda(T-1) + B^\mu(T)\right) - Y^\mu\right] \\ &\times\left[\sigma\left(R^\beta_\mu(T,T-1)X_\beta(T-1) + B_\mu(T)\right) - Y_\mu\right] \end{aligned}$$

This expression *does* contain the transition functions and the bias functions of the last layer in the network and so now we could perform the derivative with respect to them to get

$$\begin{aligned} \delta B^\mu(T) &= -\epsilon\frac{\partial C}{\partial B_\mu(T)} \\ \delta R^\mu_\nu(T,T-1) &= -\epsilon\frac{\partial C}{\partial R^\nu_\mu(T,T-1)} \end{aligned}$$

Instead of directly substituting the values in to the original equation we can use the chain rule as follows:

$$\begin{aligned} \delta B^\mu(T) &= -\epsilon\frac{\partial C}{\partial B_\mu(T)} \\ &= -\epsilon\frac{\partial C}{\partial X_a(T)}\frac{\partial X_a(T)}{\partial B_\mu(T)} \\ &= -\epsilon\frac{\partial C}{\partial X_a(T)}\frac{\partial X_a(T)}{\partial \sigma(z_a)}\frac{\partial \sigma(z_a)}{\partial B_\mu(T)} \\ &= -\epsilon\frac{\partial C}{\partial X_a(T)}\frac{\partial X_a(T)}{\partial \sigma(z_a)}\frac{\partial \sigma(z_a)}{\partial z_a}\frac{\partial z_a}{\partial B_\mu(T)} \end{aligned}$$

where we have defined

$$z_a = R^\nu_a(T,T-1)X_\nu(T-1) + B_a(T)$$

which is the argument to the $\sigma(z_a)$ function. So the chain rule has given us

$$\delta B^\mu(T) = -\epsilon\frac{\partial C}{\partial X_a(T)}\frac{\partial X_a(T)}{\partial \sigma(z_a)}\frac{\partial \sigma(z_a)}{\partial z_a}\frac{\partial z_a}{\partial B_\mu(T)}$$

Now let's see if we can solve each of these factors.

$$\begin{aligned} \frac{\partial C}{\partial X_a(T)} &= \frac{\partial}{\partial X_a(T)}\frac{1}{2}\sum_{\mu=0}^{N(T)}\left[X^\mu(T) - Y^\mu\right]\left[X_\mu(T) - Y_\mu\right] \\ &= \frac{1}{2}\sum_{\mu=0}^{N(T)}\Big(\delta^{\mu a}\left[X_\mu(T) - Y_\mu\right] + \left[X^\mu(T) - Y^\mu\right]\delta^a_\mu\Big) \\ &= \frac{1}{2}\Big(\left[X^a(T) - Y^a\right] + \left[X^a(T) - Y^a\right]\Big) \\ &= X^a(T) - Y^a \end{aligned}$$

where we have introduced the Kronecker delta

$$\delta_{ab} = \delta^{ab} = \delta^a_b = \begin{cases} 0 & \text{if } a \neq b \\ 1 & \text{if } a = b \end{cases}$$

and we have used the identities

$$\begin{aligned} \frac{\partial}{\partial X_a(T)}X^\mu(T) &= \delta^{\mu a} \\ \frac{\partial}{\partial X_a(T)}X_\mu(T) &= \delta^a_\mu \end{aligned}$$

we also have

$$\frac{\partial}{\partial X^a(T)} X^\mu(T) = \delta^\mu_a$$
$$\frac{\partial}{\partial X^a(T)} X_\mu(T) = \delta_{a\mu}$$

Exercise

Show, by writing it out and using the properties of the delta function, that

$$\sum_{b=0}^{N} \delta^a_b X^b(T) = X^a(T)$$

where $a \in \{0, \dots, N\}$ and $b \in \{0, \dots, N\}$. Note that we could have written this as

$$\delta^a_b X^b(T) = X^a(T)$$

since the repeated index b in an upper and lower position implies a sum over its values.

We have found that

$$\delta B^\mu(T) = -\epsilon \left[X^a(T) - Y^a\right] \frac{\partial X_a(T)}{\partial \sigma(z_a)} \frac{\partial \sigma(z_a)}{\partial z_a} \frac{\partial z_a}{\partial B_\mu(T)}$$

The next factor is easy

$$\frac{\partial X_a(T)}{\partial \sigma(z_a)} = 1$$

and the next is

$$\frac{\partial \sigma(z_a)}{\partial z_a} = \frac{\partial}{\partial z_a}\left(\frac{1}{1+e^{-z_a}}\right) = \sigma(z^a)\left[1 - \sigma(z^a)\right]$$

Exercise

Show that

$$\frac{\partial \sigma(z_a)}{\partial z_a} = \sigma(z^a)\left[1 - \sigma(z^a)\right]$$

So we have

$$\delta B^\mu(T) = -\epsilon \left[X^a(T) - Y^a\right] \sigma(z^a)\left[1 - \sigma(z^a)\right] \frac{\partial z_a}{\partial B_\mu(T)}$$

The final factor is

$$\begin{aligned}\frac{\partial z_a}{\partial B_\mu(T)} &= \frac{\partial}{\partial B_\mu(T)} \left(R^\nu_a(T, T-1) X_\nu(T-1) + B_a(T)\right) \\ &= \delta^\mu_a\end{aligned}$$

since there is no $B_\nu(T)$ inside $X_\nu(T-1)$. Our result is therefore

$$\delta B^\mu(T) = -\epsilon \left[X^a(T) - Y^a\right] \sigma(z^a)\left[1 - \sigma(z^a)\right] \delta^\mu_a$$

However, notice that our forward propagation rule says that

$$X^a(T) = \sigma\left(R^a_\nu(T, T-1) X^\nu(T-1) + B^a(T)\right) \equiv \sigma(z^a)$$

and so our result is

$$\delta B^\mu(T) = -\epsilon \left[X^a(T) - Y^a\right] X^a(T)\left[1 - X^a(T)\right] \delta^\mu_a$$

Since the index a appears in both upper and lower positions so we can sum over it

$$\delta B^\mu(T) = -\epsilon \left[X^\mu(T) - Y^\mu\right] X^\mu(T)\left[1 - X^\mu(T)\right]$$

The μ only appears in the upper position so there is no summation over that index.

In a similar fashion we can show that

$$\begin{aligned}&\delta R^\mu_\nu(T, T-1) = \\ &\quad -\epsilon \left[X^\mu(T) - Y^\mu\right] X^\mu(T)\left[1 - X^\mu(T)\right] X_\nu(T-1)\end{aligned}$$

We have shown how to find the changes in the parameters in the last layer of the network. We next have to find the changes in the parameters of the second last layer.

We could do this by simply going back to our equation for the changes and put in $t = T - 1$ and then use the chain rule as before

$$\begin{aligned}\delta B^\mu(T-1) &= -\epsilon \frac{\partial C}{\partial B_\mu(T-1)} \\ &= -\epsilon \frac{\partial C}{\partial X_a(T)} \frac{\partial X_a(T)}{\partial X_b(T-1)} \frac{\partial X_b(T-1)}{\partial B_\mu(T-1)}\end{aligned}$$

since $X(T)$ contains $X(T-1)$ inside it and $X(T-1)$ contains $B(t-1)$ inside it.

Let us do this completely generally so that we can write a computer program that will back propagate through the entire network to find the changes to all the parameters.

Algorithm for Back Propagation

Our equation for the changes to any of the parameters was given by

$$\delta B^\mu(t) = -\epsilon \frac{\partial C}{\partial B_\mu(t)}$$
$$\delta R^\mu_\nu(t+1, t) = -\epsilon \frac{\partial C}{\partial R^\nu_\mu(t+1, t)}$$

We would now like to find the changes to the parameters in an arbitrary layer at $t = T - k$ for some k. The case $k = 0$ is the output layer at $t = T$, the case $k = 1$ is the second last layer. etc., with the case $k = T - 1$ being the first hidden layer of the network.

For arbitrary $k \in \{0, \dots, T-1\}$ we have

$$\delta B^\mu(T-k) = -\epsilon \frac{\partial C}{\partial B_\mu(T-k)}$$
$$\delta R^\mu_\nu(T-k, T-k-1) = -\epsilon \frac{\partial C}{\partial R^\nu_\mu(T-k, T-k-1)}$$

where we see that $k = 0$ gives

$$\delta B^\mu(T) = -\epsilon \frac{\partial C}{\partial B_\mu(T)}$$
$$\delta R^\mu_\nu(T, T-1) = -\epsilon \frac{\partial C}{\partial R^\nu_\mu(T, T-1)}$$

which are the parameters of the output layer and $k = T - 1$ gives

$$\delta B^\mu(1) = -\epsilon \frac{\partial C}{\partial B_\mu(1)}$$
$$\delta R^\mu_\nu(1, 0) = -\epsilon \frac{\partial C}{\partial R^\nu_\mu(1, 0)}$$

which are the parameters of the first hidden layer. Hence, if we find these quantities for all k we will have found the changes to all of the parameters in the network.

In order to find the changes in parameters for an arbitrary layer k, we begin at the output layer where we have a formula for C and we use the chain rule to work our way back as follows.

$$\delta B^\mu(T-k) = -\epsilon \frac{\partial C}{\partial B_\mu(T-k)} = -\epsilon \frac{\partial C}{\partial X^{a_0}(T)} \times \frac{\partial X^{a_0}(T)}{\partial X^{a_1}(T-1)} \frac{\partial X^{a_1}(T-1)}{\partial X^{a_2}(T-2)} \cdots \frac{\partial X^{a_k}(T-k)}{\partial B_\mu(T-k)}$$

and

$$\delta R^\mu_\nu(T-k, T-k-1) = -\epsilon \frac{\partial C}{\partial R^\nu_\mu(T-k, T-k-1)}$$
$$= -\epsilon \frac{\partial C}{\partial X^{a_0}(T)} \times \frac{\partial X^{a_0}(T)}{\partial X^{a_1}(T-1)} \frac{\partial X^{a_1}(T-1)}{\partial X^{a_2}(T-2)} \cdots \frac{\partial X^{a_k}(T-k)}{\partial R^\nu_\mu(T-k, T-k-1)}$$

Notice from the above expressions that the only difference is in the last factor of each. This means that if we figure out the following

$$A_{a_k}(T-k) \equiv -\epsilon \frac{\partial C}{\partial X^{a_0}(T)} \times \frac{\partial X^{a_0}(T)}{\partial X^{a_1}(T-1)} \frac{\partial X^{a_1}(T-1)}{\partial X^{a_2}(T-2)} \cdots \frac{\partial X^{a_{k-1}}(T-[k-1])}{\partial X^{a_k}(T-k)}$$

then we can use it in both expressions.

$$\delta B^\mu(T-k) = A_{a_k}(T-k) \frac{\partial X^{a_k}(T-k)}{\partial B_\mu(T-k)}$$
$$\delta R^\mu_\nu(T-k, T-k-1) = A_{a_k}(T-k) \frac{\partial X^{a_k}(T-k)}{\partial R^\nu_\mu(T-k, T-k-1)}$$

Let us begin with the factor of $A_{a_k}(T-k)$. It is clear from the formula that, letting $k \to [k-1]$ we have

$$A_{a_{k-1}}(T-[k-1]) = -\epsilon \frac{\partial C}{\partial X^{a_0}(T)} \times \frac{\partial X^{a_0}(T)}{\partial X^{a_1}(T-1)} \frac{\partial X^{a_1}(T-1)}{\partial X^{a_2}(T-2)} \cdots \frac{\partial X^{a_{k-2}}(T-[k-2])}{\partial X^{a_{k-1}}(T-[k-1])}$$

and so

$$A_{a_k}(T-k) = A_{a_{k-1}}(T-[k-1]) \frac{\partial X^{a_{k-1}}(T-[k-1])}{\partial X^{a_k}(T-k)}$$

In other words, we can use the result from the previous layer inductively to find the next layer as we move back through the network. For example, for $k = 0$ we have the output layer and so

$$\delta B^\mu(T) = -\epsilon \frac{\partial C}{\partial B_\mu(T)} = -\epsilon \frac{\partial C}{\partial X^{a_0}(T)} \frac{\partial X^{a_0}(T)}{\partial B_\mu(T)}$$

and so

$$A_{a_0}(T) = -\epsilon \frac{\partial C}{\partial X^{a_0}(T)} = -\epsilon \left[X_{a_0}(T) - Y_{a_0}\right]$$

For $k = 1$ we have

$$A_{a_1}(T-1) = A_{a_0}(T) \frac{\partial X^{a_0}(T)}{\partial X^{a_1}(T-1)}$$

For $k = 2$ we have

$$A_{a_2}(T-2) = A_{a_1}(T-1) \frac{\partial X^{a_1}(T-1)}{\partial X^{a_2}(T-2)}$$

and we can just keep going, using the result from the previous calculation to compute the next. This is the back propagation algorithm. The general formula is

$$A_{a_{k+1}}(T-[k+1]) = A_{a_k}(T-k) \frac{\partial X^{a_k}(T-k)}{\partial X^{a_{k+1}}(T-[k+1])}$$

The partial derivative is easily found to be

$$\frac{\partial X^{a_k}(T-k)}{\partial X^{a_{k+1}}(T-[k+1])} = X^{a_k}(T-k)\left[1 - X^{a_k}(T-k)\right] R^{a_k}_{a_{k+1}}(T-k, T-[k+1])$$

We can combine this with the parts we have solved before to get

$$\delta B^\mu(T-k) = A_{a_k}(T-k) \frac{\partial X^{a_k}(T-k)}{\partial B_\mu(T-k)} = A^\mu(T-k) X^\mu(T-k)\left[1 - X^\mu(T-k)\right]$$
$$\delta R^\mu_\nu(T-k, T-k-1) = A_{a_k}(T-k) \frac{\partial X^{a_k}(T-k)}{\partial R^\nu_\mu(T-k, T-k-1)} = A^\mu(T-k) X^\mu(T-k)\left[1 - X^\mu(T-k)\right] X_\nu(T-k-1)$$

and now we can start at $k = 0$ for the output layer, and increment to $k = [T - 1]$ always using the results from the previous computation to get the next until all of the parameters are updated.

After we have updated all of the parameters, we then run the forward propagation again with the new parameters, compare the result with our training data, and if the result is still too far away from the training data we do another backpropagation to update the parameters again. We continue in this fashion, over and over, until the output of the network matches the desired output to within our specified error.

It the next section we will start with the C program that we wrote for forward propagation and we will add the back propagation algorithm to it.

Writing a Neural Network in C

The C program given below produces the following output:

```
After 6000 backpropagations the neural net
is now trained with the following parameters:

B[mu](0)  R[mu][nu](0)

|0.0098|  | 0.1978 0.2978 |
|0.0030|  | 0.3300 0.4300 |

B[mu](1)  R[mu][nu](1)

|1.7533|  | 2.1244 2.2510 |
|-0.7756|  | -0.2261 -0.1744 |

Final output and desired values are:

X[0][2] = 0.9907   Y[0] = 1.0000
X[1][2] = 0.5000   Y[1] = 0.5000
```

Here is the code that we used to produce the output:

backward.c

```
#include <stdio.h>
#include <math.h>

#define N 2 // num neurons per layer
#define T 2 // num hidden layers
#define EPS 0.5 // learning rate
#define EPOCHS 16000 // number of
    backpropagations

double sigma(double x);
int main (void){
 double X[N][T+1], Z[N][T+1];
 double R[N][N][T+1], B[N][T+1];
 double Y[N], A[N][T+1];
 int mu, nu, beta, t;

 R[0][0][0] = 0.1;
 R[0][1][0] = 0.2;
 R[1][0][0] = 0.3;
 R[1][1][0] = 0.4;
 R[0][0][1] = 0.5;
 R[0][1][1] = 0.6;
 R[1][0][1] = 0.7;
 R[1][1][1] = 0.8;
 B[0][1] = 0.1;
 B[1][1] = 0.2;
 B[0][2] = 0.3;
 B[1][2] = 0.4;
 X[0][0] = 10;
 X[1][0] = 10;
 Y[0] = 1;
 Y[1] = 0.5;

for(int epoch = 0; epoch < EPOCHS; epoch
    ++){
 // forward propagate
 for (t = 0; t < T; t++){
  for (mu = 0; mu < N; mu++){
   Z[mu][t+1] = B[mu][t+1];
   for (nu = 0; nu < N; nu++){
    Z[mu][t+1] = Z[mu][t+1] + R[mu][nu][t
        ]*X[nu][t];
   }
   X[mu][t+1] = sigma(Z[mu][t+1]);
  }
 }
 // back propagate
 // output layer first
 for (mu = 0; mu < N; mu++){
  A[mu][T-1] = (X[mu][T]-Y[mu])*X[mu][T
      ]*(1-X[mu][T]);
  B[mu][T-1] = B[mu][T-1]-EPS*A[mu][T-1];
  for (nu = 0; nu < N; nu++){
   R[mu][nu][T-1] = R[mu][nu][T-1]-EPS*A[
       mu][T-1]*X[nu][T-1];
  }
 }
 // back through remaining layers
 for (t = T-1; t > 0; t--){
  for (mu = 0; mu < N; mu++){
   A[mu][t-1] = 0;
   for (beta = 0; beta < N; beta++){
    A[mu][t-1] = A[mu][t-1] + A[beta][t]*
        R[beta][mu][t]*X[mu][t]*(1-X[mu][t
        ]);
   }
   B[mu][t-1] = B[mu][t-1]-EPS*A[mu][t
       -1];
   for (nu = 0; nu < N; nu++){
    R[mu][nu][t-1] = R[mu][nu][t-1]-EPS*A
        [mu][t-1]*X[nu][t-1];
   }
  }
 }
}
 // print out final values of R and B
 fprintf(stderr,"After %d
     backpropagations the neural net\n",
     EPOCHS);
 fprintf(stderr,"is now trained with the
     following parameters:\n");
 for (t = 0; t < T; t++){
  fprintf(stderr,"\nB[mu](%d)\tR[mu][nu
      ](%d)\n", t, t);
  for (mu = 0; mu < N; mu++){
   fprintf(stderr,"\n|%0.4lf|\t|", B[mu][
       t]);
   for (nu = 0; nu < N; nu++){
    fprintf(stderr," %0.4lf", R[mu][nu][t
        ]);
   }
   fprintf(stderr," |");
  }
  fprintf(stderr,"\n");
 }
 fprintf(stderr,"\nFinal output and
     desired values are:\n\n");
 for (mu = 0; mu < N; mu++){
  fprintf(stderr,"X[%d][%d] = %0.4lf\t Y
      [%d] = %0.4lf\n", mu, T, X[mu][T],
      mu, Y[mu]);
 }
 fprintf(stderr,"\n");
 return 0;
}
double sigma(double x){
 return 1/(1+exp(-x));
}
```

We would now like to make some improvements to this program. We would first like to simply choose the initial parameters randomly rather than input them by hand, we would like to have the cost function determine how many epochs we run rather than putting that in by hand. I.e. we will keep doing back propagations until our error is minimized under a certain level. Finally we would like to be able to input how many layers and how many neurons per layer as well as both the input data and the training data. Here is some code that achieves these goals. You would be advised to study the code and compare with the formulae to ensure that you understand how we are translating mathematical formulas into computer code.

neural.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define N 2       // num neurons per layer
#define EPS 0.5     // learning rate
#define ERROR 0.00001 // end when error
    reaches this

double sigma(double x){return 1/(1+exp(-x
    ));}

int main (int argc, char **argv){
 if (argc != 2*N+2){
  fprintf(stderr,"need %d inputs, %d
      correct answers\n", N, N);
  fprintf(stderr,"followed by the number
      of hidden layers\n");
  fprintf(stderr,"EX: %s 0.05 0.10 0.01
      0.99 2\n", argv[0]);
  return 1;
 }
 srand(time(NULL));
 int T = atoi(argv[argc-1]);
 double Y[N], cost=ERROR+1;
 double X[N][T+1],Z[N][T+1],dB[N][T+1];
 double R[N][N][T+1],B[N][T+1];
 int mu, nu, t=0, cycles=0;
 for (mu = 0; mu < N; mu++){
  X[mu][0] = atof(argv[mu+1]);
  Y[mu] = atof(argv[mu+N+1]);
 }
 // randomly initialize parameters
 for (t = 1; t <= T; t++){
  for (mu = 0; mu < N; mu++){
   B[mu][t] = (double) (rand()%1000)
       /1000.0;
   for (nu = 0; nu < N; nu++){
    R[mu][nu][t] = (double) (rand()%1000)
        /1000.0;
   }
  }
 }
 while( cost > ERROR ){
  // forward propagate
  for (t = 1; t <= T; t++){
   for (mu = 0; mu < N; mu++){
    Z[mu][t] = B[mu][t];
    for (nu = 0; nu < N; nu++){
     Z[mu][t] = Z[mu][t] + R[mu][nu][t]*X
         [nu][t-1];
    }
    X[mu][t] = sigma(Z[mu][t]);
   }
  }
  // back propagate
  // k=0 layer
  for (mu = 0; mu < N; mu++){
   dB[mu][T] = -EPS*(X[mu][T]-Y[mu])*X[mu
       ][T]*(1-X[mu][T]);
   B[mu][T] = B[mu][T] + dB[mu][T];
   for (nu = 0; nu < N; nu++){
    R[mu][nu][T] = R[mu][nu][T] + dB[mu][
        T]*X[nu][T-1];
   }
  }
  // k = 1...T-1 layers
  for (int k = 1; k < T; k++){
   for (mu = 0; mu < N; mu++){
    dB[mu][T-k] = 0;
    for (int a = 0; a < N; a++){
     dB[mu][T-k] = dB[mu][T-k] + dB[a][T-
         k+1]*R[a][mu][T-k+1]*X[mu][T-k
         ]*(1-X[mu][T-k]);
    }
    B[mu][T-k] = B[mu][T-k] + dB[mu][T-k
        ];
    for (nu = 0; nu < N; nu++){
     R[mu][nu][T-k] = R[mu][nu][T-k] + dB
         [mu][T-k]*X[nu][T-k-1];
    }
   }
  }
  // calculate cost function
  cost = 0;
  for (mu = 0; mu < N; mu++){
   cost = cost + (X[mu][T]-Y[mu])*(X[mu][
       T]-Y[mu]);
  }
  cost = 0.5*cost;
```

```
    // increment number of backpropagations
    cycles++;
   }
   // print out final values of R and B
   fprintf(stdout,"\nThe trained network
       parameters:\n");
   for (t = 1; t <= T; t++){
    fprintf(stdout,"\nB[mu](%d)\t R[mu][nu
        ](%d,%d)\n", t, t, t-1);
    for (mu = 0; mu < N; mu++){
     fprintf(stdout,"\n|%6.3lf |\t|", B[mu
         ][t]);
     for (nu = 0; nu < N; nu++){
      fprintf(stdout," %6.3lf ", R[mu][nu][
          t]);
     }
     fprintf(stdout," |");
    }
    fprintf(stdout,"\n");
   }
   fprintf(stdout,"After %d
       backpropagations:\n\n",cycles);
   // print out final values X
   fprintf(stdout,"OUTPUT\t\t\tCORRECT
       OUTPUT\n");
   fprintf(stdout,"------\t\t\t
       --------------\n");
   for (mu = 0; mu < N; mu++){
    fprintf(stdout,"X[%d][%d] = %0.2lf \t\
        tY[%d] = %0.2lf\n", mu, T, X[mu][T],
         mu, Y[mu]);
   }
   return 0;
  }
```

Here is the output

```
./a.out 10 10 1 0.5 2

 The trained network parameters:

 B[mu](1)    R[mu][nu](1,0)

 | 0.029 |  |  0.728   0.725  |
 | 0.798 |  |  0.243   0.305  |

 B[mu](2)    R[mu][nu](2,1)

 | 1.457 |  |  1.861   2.091  |
 | 0.061 |  | -0.116   0.056  |

 After 16967 backpropagations:

 OUTPUT      CORRECT OUTPUT
 ------      --------------
 X[0][2] = 1.00     Y[0] = 1.00
 X[1][2] = 0.50     Y[1] = 0.50
```

Training Data

In our example we have only used a single training example. In real neural networks there is an entire collection of examples that are used as training data. This is because, rather than designing a neural network to simply solve a single problem, we would like it to solve any problem of a given type. We feed it many training examples of the type we are interested in with the hope that it will learn how to solve problems of the same type but for which the answer is not known.

In order to train the network on a larger collection of training data we need to figure out how to design a cost function that will take into account multiple training examples at the same time. In other words, we have a collection of input-answer pairs which contain the input to the network and the desired answer that you would like it to produce from that input.

You feed these pairs in one at a time, modifying the parameters of the network for each one with the hope that, if you have enough layers and enough neurons, the parameters will converge to a solution of the 'meta problem' that you are trying to solve. For example suppose we would like to design a neural network that acts like an AND gate. This means that when we input a $(1,0)$,$(0,1)$ or $(0,0)$ the output should be 0, but if we input $(1,1)$ the output should be 1. If we train the network simply using the $(1,0)$ input with the correct answer being zero it is not likely that the network will also get the correct answer for the larger problem that we are trying to solve. We will have trained it for the 'problem' of returning 0 for a $(1,0)$ input but we have not trained it for the 'meta problem' of acting like an AND gate.

Instead, we attempt to create an AND gate by modifying our code so that it cycles through all of the solutions as we train it. We perform a single gradient descent to modify the parameters between each attempt with a different input-output pair. The result is to, hopefully, build a network solution to the meta problem of being and AND gate. We then test our solution by feeding all of the inputs back in again using only forward propagation to see if our trained network can still solve them all after the training has been completed. If it can then it can be called an AND gate network since it is then a neural network that acts exactly like an AND gate with no further training required.

and.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define EPS 0.2     // learning rate
#define ERROR 0.00001 // end when error
    reaches this

double sigma(double x){return 1/(1+exp(-x
    ));}

int main (int argc, char **argv){
 if (argc != 3){
  fprintf(stderr,"need the number of
      neurons per layer\n");
 fprintf(stderr,"and the number of
     hidden layers\n");
 fprintf(stderr,"EX: %s 4 2\n", argv[0])
     ;
 return 1;
}
srand(time(NULL));
int N = atoi(argv[1]);
int T = atoi(argv[2]);
int mu, nu, t=0, cycles=0;
double Y[N],cost=ERROR+1;
double R[N][N][T+1],B[N][T+1];
double X[N][T+1],Z[N][T+1],dB[N][T+1];
double IN[4][N],OUT[4][N];

// initialize all inputs and outputs
for(int i = 0; i < 4; i++){
 for (mu = 0; mu < N; mu++){
  IN[i][N] = 0.001;
  OUT[i][N] = 0.001;
 }
}
// inputs and desired outputs for AND
    gate
// Note that we only care about the
    first
// two neurons, all others remain zero
// For the output, all 1's is a 1, all
   0's
// is a 0
IN[0][0] = 1;
IN[0][1] = 1;
OUT[0][0] = 1;
IN[1][0] = 1;
IN[1][1] = 0;
OUT[1][0] = 0;
IN[2][0] = 0;
IN[2][1] = 1;
OUT[2][0] = 0;
IN[3][0] = 0;
IN[3][1] = 0;
OUT[3][0] = 0;

// randomly initialize network
    parameters
for (t = 1; t <= T; t++){
 for (mu = 0; mu < N; mu++){
  B[mu][t] = (double) (rand()%1000)
      /1000.0;
  for (nu = 0; nu < N; nu++){
   R[mu][nu][t] = (double) (rand()%1000)
       /1000.0;
  }
 }
}

while( cost > ERROR ){
 // loop over training data
 for(int i = 0; i < 4; i++){
  for (mu = 0; mu < N; mu++){
   X[mu][0] = IN[i][mu];
   Y[mu] = OUT[i][0];
  }
  // forward propagate
  for (t = 1; t <= T; t++){
   for (mu = 0; mu < N; mu++){
    Z[mu][t] = B[mu][t];
    for (nu = 0; nu < N; nu++){
     Z[mu][t] = Z[mu][t] + R[mu][nu][t]*
         X[nu][t-1];
    }
    X[mu][t] = sigma(Z[mu][t]);
   }
  }
  // back propagate
  // k=0 layer
  for (mu = 0; mu < N; mu++){
   dB[mu][T] = -EPS*(X[mu][T]-Y[mu])*X[
       mu][T]*(1-X[mu][T]);
   B[mu][T] = B[mu][T] + dB[mu][T];
   for (nu = 0; nu < N; nu++){
    R[mu][nu][T] = R[mu][nu][T] + dB[mu
        ][T]*X[nu][T-1];
   }
  }
  // k = 1...T-1 layers
  for (int k = 1; k < T; k++){
   for (mu = 0; mu < N; mu++){
    dB[mu][T-k] = 0;
    for (int a = 0; a < N; a++){
     dB[mu][T-k] = dB[mu][T-k] + dB[a][T
         -k+1]*R[a][mu][T-k+1]*X[mu][T-k
         ]*(1-X[mu][T-k]);
    }
    B[mu][T-k] = B[mu][T-k] + dB[mu][T-k
        ];
    for (nu = 0; nu < N; nu++){
     R[mu][nu][T-k] = R[mu][nu][T-k] +
         dB[mu][T-k]*X[nu][T-k-1];
    }
   }
  }
  // calculate cost function
  cost = 0;
  for (mu = 0; mu < N; mu++){
   cost = cost + (X[mu][T]-Y[mu])*(X[mu
       ][T]-Y[mu]);
  }
  cost = 0.5*cost;
  // increment number of
      backpropagations
  cycles++;
 } // end training data loop
} // end cost while loop

// Now the network should be trained.
// print out final values of R and B
fprintf(stdout,"\nThe trained network
    parameters\n");
fprintf(stdout,"after %d
    backpropagations:\n", cycles);
for (t = 1; t <= T; t++){
 fprintf(stdout,"\nB[mu](%d)\t R[mu][nu
     ](%d,%d)\n", t, t, t-1);
 for (mu = 0; mu < N; mu++){
  fprintf(stdout,"\n|%6.3lf |\t|", B[mu
      ][t]);
  for (nu = 0; nu < N; nu++){
   fprintf(stdout," %6.3lf", R[mu][nu][t
       ]);
  } fprintf(stdout," |");
 } fprintf(stdout,"\n");
}

// Now test the trained network
fprintf(stdout,"\nTesting the trained
      network...\n");
  for(int i = 0; i < 4; i++){
   for (mu = 0; mu < N; mu++){
    X[mu][0] = IN[i][mu];
    Y[mu] = OUT[i][0];
   }
   for (t = 1; t <= T; t++){
    for (mu = 0; mu < N; mu++){
     Z[mu][t] = B[mu][t];
     for (nu = 0; nu < N; nu++){
      Z[mu][t] = Z[mu][t] + R[mu][nu][t]*X
          [nu][t-1];
     }
     X[mu][t] = sigma(Z[mu][t]);
    }
   }
   // print out results from this input
   fprintf(stdout,"\nINPUT\t\t\t\t\t\
       tOUTPUT\n");
   fprintf(stdout,"-----\t\t\t\t\t\t
       ------\n");
   for (mu = 0; mu < N; mu++){
    fprintf(stdout,"X[%d][0] = %0.2lf\tX[%
        d][%d] = %0.2lf\n", mu, X[mu][0],
        mu, T, X[mu][T]);
   }
  }
  return 0;
 }
```

Here is what I get with 3 layers of 4 neurons each:

```
./a.out 4 3

 The trained network parameters
 after 708108 backpropagations:

 B[mu](1)    R[mu][nu](1,0)

 |-5.357 |  |  3.706   3.624   0.963   0.507  |
 | 1.541 |  |  0.613   0.548   0.105   0.901  |
 | 1.332 |  |  0.729   0.815   0.061   0.295  |
 |-4.649 |  |  3.163   3.248   0.201   0.921  |

 B[mu](2)    R[mu][nu](2,1)

 |-1.589 |  |  3.082  -0.719  -0.951   3.092  |
 | 2.100 |  |  0.987   1.472   1.474   1.293  |
 |-4.942 |  |  7.313  -1.218  -0.770   5.919  |
 | 1.902 |  |  0.869   1.457   1.614   0.996  |

 B[mu](3)    R[mu][nu](3,2)

 |-2.357 |  |  2.492  -2.329   9.628  -1.598  |
 |-2.482 |  |  2.328  -1.479   9.747  -2.300  |
 |-2.834 |  |  2.714  -1.763   9.463  -1.715  |
 |-2.237 |  |  2.798  -2.385   9.407  -1.706  |

 Testing the trained network...

 INPUT            OUTPUT
 -----            ------
 X[0][0] = 1.00  X[0][3] = 1.00
 X[1][0] = 1.00  X[1][3] = 1.00
 X[2][0] = 0.00  X[2][3] = 1.00
 X[3][0] = 0.00  X[3][3] = 1.00

 INPUT            OUTPUT
 -----            ------
 X[0][0] = 1.00  X[0][3] = 0.00
 X[1][0] = 0.00  X[1][3] = 0.00
 X[2][0] = 0.00  X[2][3] = 0.00
 X[3][0] = 0.00  X[3][3] = 0.00

 INPUT            OUTPUT
 -----            ------
 X[0][0] = 0.00  X[0][3] = 0.00
 X[1][0] = 1.00  X[1][3] = 0.00
 X[2][0] = 0.00  X[2][3] = 0.00
 X[3][0] = 0.00  X[3][3] = 0.00

 INPUT            OUTPUT
 -----            ------
 X[0][0] = 0.00  X[0][3] = 0.00
 X[1][0] = 0.00  X[1][3] = 0.00
 X[2][0] = 0.00  X[2][3] = 0.00
 X[3][0] = 0.00  X[3][3] = 0.00
```

Notice that, rather than having different numbers of neurons in different layers, I have just let all layers have four neurons. It made the `for` loops easier to code. However, this means that we have four input neurons when an AND gate only takes two inputs, and we have four outputs when an AND gate only produces a single output. We have chosen to simply pad the input layer with zeros so that only the first two neurons act as the AND gate input, and we have chosen to represent a zero output by all four output neurons being 0 while a 1 output is represented by all four output neurons being 1. This makes the program code much simpler while not adding a significant amount of computational overhead.

Exercises

1. Test out different values of the learning rate ϵ to see what affect it has on both the results and the time it takes to complete.
2. Notice that the AND gate will still work quite well with only 2 neurons per layer and only 1 hidden layer (try it!). Test out different numbers of layers and neurons and see if there is a minimum number to get the correct answer and what are the effects of adding more of each. Does it take significantly longer to run as you add more neurons and layers? Use the Bash `time` command to time the execution time versus number of neurons per layer and graph it. Then do the same for number of layers. Can you see why we need giant supercomputing data centers to run large neural networks with billions of neurons? Why do typical networks have a large number of neurons but a relatively small number of layers? Is it the execution time? or the accuracy? or a combination/optimization of both? or something else entirely?
3. Use our code as a template to write neu-

ral networks for OR, NOR, NOT, XOR, and NAND gates. Note that a NAND and NOR gates are universal. This means that any logical function can be built entirely from NAND gates or entirely from NOR gates. So if you have a method of constructing NAND gates you can build any computing machine with it. Here is what I get for an XOR gate:

```
./a.out 2 2

The trained network parameters
after 3037360 backpropagations:

B[mu](1)          R[mu][nu](1,0)

|-3.973  |       |  8.507  8.506 |
|-10.292 |       |  6.747  6.747 |

B[mu](2)          R[mu][nu](2,1)

|-5.982 |        | 12.419 -12.876 |
|-5.982 |        | 12.419 -12.875 |

Testing the trained network...

INPUT            OUTPUT
-----            ------
X[0][0] = 1.00  X[0][2] = 0.00
X[1][0] = 1.00  X[1][2] = 0.00

INPUT            OUTPUT
-----            ------
X[0][0] = 1.00  X[0][2] = 1.00
X[1][0] = 0.00  X[1][2] = 1.00

INPUT            OUTPUT
-----            ------
X[0][0] = 0.00  X[0][2] = 1.00
X[1][0] = 1.00  X[1][2] = 1.00

INPUT            OUTPUT
-----            ------
X[0][0] = 0.00  X[0][2] = 0.00
X[1][0] = 0.00  X[1][2] = 0.00
```

Project

Modify the neural network training code so that it will output the network parameters of the trained network to files. One file for the transition functions and one file for the bias functions. Then write a neural network that reads in these parameters along with the user-supplied input data to produce output. If the network parameters were generated by correct training, this will now be a network that should be able to answer questions correctly.

Advanced Project

Classifying hand-written digits is often called the 'hello world' program for neural networks. It is used in mobile deposits for banks as well as a multitude of other applications. There is a free data set, called MNIST, that can be found online containing 50,000 images of handwritten numerical digits and they are all labeled by the number that the image is showing.
Write a neural network that takes the sequence of greyscale pixels from the images – i.e. each pixel has a number assigned to it depending on how light or dark it is, these numbers can be lined up into a long vector and then used as input to the neural network. The output of the neural network is a vector Y^μ with ten components. So, for example, if the digit is a 0 then the output will have a 1 in the zeroth component and zeros everywhere else, if the digit in the picture is a 1 then the output should have a 0 in the zeroth component, a 1 in the first component, and zeros everywhere else. Note that there are fewer output neurons than input neurons so you will have to modify your code to allow different number of neurons in different layers (at least the last layer) or else simply pad the output vector with zeros after the first ten.
You should be able to train your network with the first 40,000 images in the data set and then test it with the remaining 10,000 to see if it gives the correct answers – or at least decide what percent accuracy it has. You should then try different numbers of layers and neurons per layer to see if you can improve your networks accuracy.
If you can complete this project successfully, congratulations! You are prepared to write more complicated neural networks and you could even be selected for a position in a company which designs neural networks.

Language Models

A large language model refers to a type of artificial intelligence model designed to understand and generate human language. It is trained on vast amounts of text data to learn the statistical patterns and relationships within language.

These models utilize deep learning techniques, particularly neural networks with numerous layers, to process and analyze text data. They are called "large" because they consist of a massive number of parameters, enabling them to capture intricate patterns and nuances in language.

Large language models, like GPT-4 (Generative Pretrained Transformer 4), are capable of various natural language processing tasks, such as text generation, language translation, sentiment analysis, question-answering, and more. They can comprehend context, contextually generate coherent responses, and even mimic human-like conversation in certain situations.

The size and complexity of these models contribute to their remarkable performance, making them state-of-the-art in many language-related tasks. However, training and deploying such large models require significant computational resources, making them less accessible to individual developers but more prevalent in research and industry applications.

The Future of AI

AI holds great potential and is likely to lead us to new frontiers of technological advancement. AI research and development will continue to progress rapidly, leading to more sophisticated and capable AI systems. Breakthroughs in algorithms, hardware, and data availability will drive innovation in the field. AI is already making significant impacts in healthcare, finance, transportation, and manufacturing. As AI technologies become more integrated into society, there will be increased focus on ethical considerations. Ensuring fairness, transparency, and accountability in AI systems will be crucial to building trust and minimizing potential negative consequences.

The combination of AI and robotics will lead to the development of more advanced autonomous systems, enhancing capabilities in fields like space exploration, disaster response, and personal assistance.

AI will play a central role in delivering personalized experiences across various platforms, from recommendation systems to personalized education and healthcare.

AI systems will continue to demonstrate creativity in various domains, including art, music, and storytelling, challenging traditional notions of human creativity. AI-powered educational tools and adaptive learning systems will provide personalized learning experiences, catering to individual needs and improving educational outcomes.

While the future of AI holds tremendous promise, there are also potential challenges, such as privacy concerns, job displacement, and ensuring AI's alignment with human values. Responsible development and thoughtful deployment will be key to shaping a positive and beneficial future for artificial intelligence.

I'M TRYING TO CREATE AN AI THAT CAN UNDERSTAND AND USE SARCASM.
HOW'S IT GOING?
IT'S GOING GREAT. THIS GUY IS A REAL GENIUS.
WAS THAT SARCASM?
I DON'T KNOW.

References

[1] Understanding Backpropagation Algorithm, Simeon Kostadinov,Towards Data Science, 2019.

[2] Learning representations by back-propagation errors, David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams, Nature, 1986.

[3] Deep Learning, Ian Goodfellow, Yoshua Bengio, Aaron Courville, MIT Press, 2016.

[4] Foundations of Machine Learning, Mehryar Mohri, Afshin Rostamizadeh, Ameet Talwalkar, MIT Press, 2018.

[5] Perceptrons, Marvin L. Minsky, Seymour A. Paperi, MIT Press, 1988.