CIS 522

Lecture 1: History to Deep Learning

1/16/2020

Lecturer: Konrad Kording *Scribe: Sadat Shaik, Kushagra Goel*

1 Welcome to CIS 522!

Welcome to CIS 522 - Deep Learning for Data Science, we hope you had a great winter break! The instructors and TA's have worked hard to make this course a reality, and so from all the staff, we are excited to welcome you to the course!

Note: These lecture notes are not meant as a replacement for lecture but rather a supplement. They are also a work in progress, so if you spot anything that may be incorrect, please let us know and we're more than happy to fix it ! :)

1.1 Course Overview

The goal of this course is to provide an intuitive understanding of Deep Learning, it's theories, techniques, and applications. As such, the course is broken down into the following coarse time-line:

Timeline

1. Fundamentals of Deep Learning (Weeks 1-4)
2. Computer Vision (Weeks 5-6)
3. NLP (Weeks 7-8)
4. Reinforcement Learning (Week 9-10)
5. Special Topics (Weeks 11-15)

It's perfectly fine (and in fact, expected!) to not know what these different domains are, that's what the purpose of this class is! By the end we hope that you have an inuitive understanding of the domain and it's techniques, such that you would be able to have a solid foundation for a future project in any of these domains.

For more about the logistics of the course, we highly recommend reading through all the provided information on our website: https://www.seas.upenn.edu/~cis522/

1.2 Lecture Overview

Deep learning has a rich and interesting history, starting with it's neurological origins in 1888 to the revolutionary framework that it has now become. Understanding the origins and development of this discipline is not only fascinating, but also provides an intuition for deep learning, providing a foundation for the concepts that we will later learn in the course. As such, the goal of this lecture will be to provide a chronological history of neural networks and it's major developments.

2 History of The Neuron

2.1 Background

The nervous system was discovered as early as the 3rd century BC by the ancient Greeks, discovering that the nervous system was responsible for receiving and processing signals from the various sensory organs (source).

In 1838, Theodore Schwann published a landmark work, translated in English to "Microscopic investigations on the similarity of structure and growth of animals and plants" where he postulated that "All living things are composed of cells and cell products". However, even with this discovery, the nervous system was thought to be composed of a single thread-like cell. In fact, until 1888, it was assumed that the nervous system was made of thread-like cells (Syncytium).

2.2 Santiago Ramón y Cajal

Santiago Ramón y Cajal is one of the fathers of modern neuroscience. In 1888, Cajal published one of his pivotal works, "Revista Trimestral de Histología Normal y Patológica" which contained many discoveries about the structure of the nervous system, laying the foundation for modern neuroscience.

Neurons in the context of Deep learning
There are four main tenets:

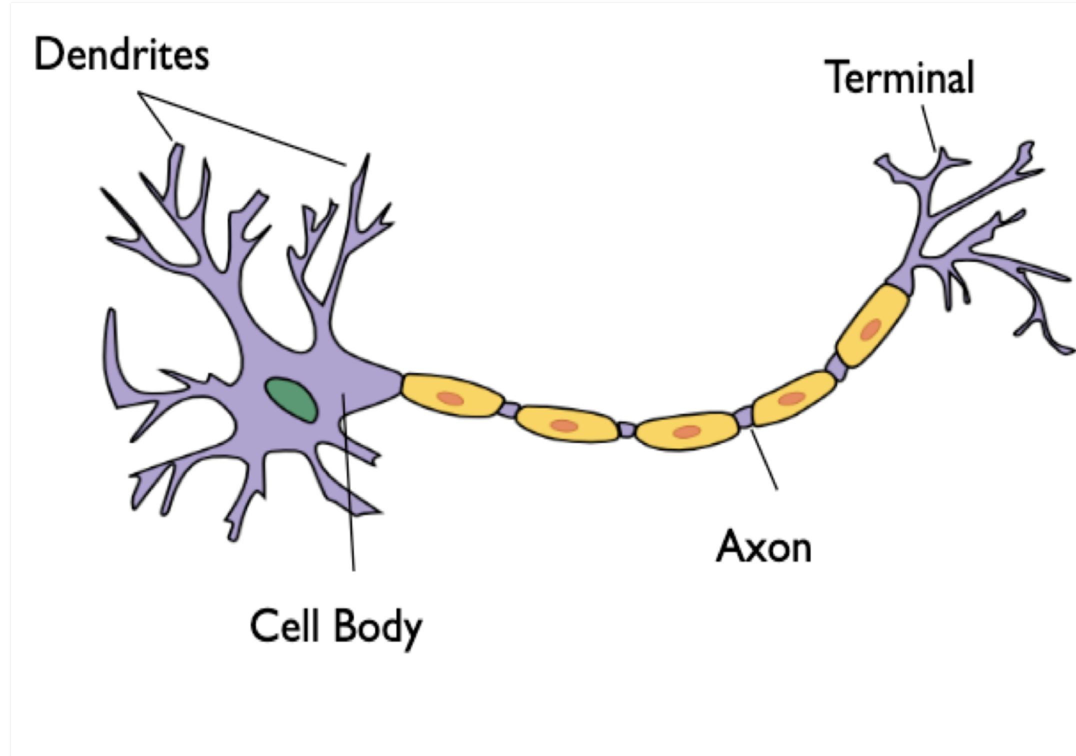
1. The nervous system is composed of numerous individual cells, called "neurons" (this term was coined by anatomist H. Waldeyer Hartz in 1891). These "neurons" are the functional unit of the nervous system. This idea is also called **the Neuron doctrine**.

2. Neurons are comprised of three parts, the dendrites, cell body (soma) and axon, of which the axon sends signals to the dendrites of another cell (see image below).

3. Although neurons are capable of transferring signals in both directions, neurons in a tissue primarily conduct a signal in a single preferred direction, this law is called **The Law of Dynamic Polarization**.

3 Structure of a Neuron

There are many components to a neuron, however, there are four components that are specifically relevant to Neural Networks: the dendrites, cell body, axon, and terminal/synapse.



Dendrites: Dendrites are a structure that integrates information from pre-synaptic neurons. Many synapses impinge on a typical dendrite. Each of these synapses mediates the influence of one of the pre-synaptic cells. The structure of the dendrite integrates all these electrical influences. It also contains nonlinear electrical channels that allow a dendrite to potentially compute a very nonlinear function. Within neural networks a dendrite is typically assumed to just be a linear integrator. source

Cell Body: The cell body often accumulates the signals from multiple dendrites. Close to the cell body (but in the Axon hillock see below) is the site of action potential integration. If the integrated electrical signal there exceeds a certain threshold level than an all or none spike is produced and propagated into the axon and often, to a lesser extent, into the dendrite.

Axon: The axon hillock is where spikes are initiated. The axon is an active electrical substrate that propagates the electrical signal that then gets transmitted to it's synapses. Hodgkin and Huxley were among the neuroscientists that helped us understand how spikes are transported without loss along axons.

Synapse Along the axon of a neuron, there are typically many synapses. When the electrical

signal traveling along the axon arrives at the synapse, a synpase converts this electrical signal into a chemical signal. This chemical signal then binds to receptors on the post-synaptic side of the synapse where it produces electrical currents, which are, in turn, integrated by the postsynaptic dendrite.

4 Perceptron Models

With the physiology of the neuron defined, we beg the question: can we model this computationally? Here we explore the development of two such models.

4.1 McCullochs - Pitts Linear Threshold Unit

McCulloch, a neuroscientist and Walter Pitts, a logician, published "A logical calculus of the ideas immanent in nervous activity" in 1943, where they jointly created a basic computational model of the neuron, which they called a "Linear Threshold Unit". The model can be defined as follows:

Definition 4.1. Consider an input to the function $x \in \{0,1\}^n$, and manually tuned parameter $\theta \in \mathbb{Z}^*$. The model's function f is defined as:

$$f(x) = \mathbb{1}\big(\sum_{i=1}^{n} x_i \geq \theta\big)$$

The summation of the features in the feature vector x is the linear function, which is then thresholded, hence the name "Linear Threshold Unit". However, this model has some clear limitations (what are they)?

4.2 Rosenblatt's Perceptron

Rosenblatt's Perceptron model (published in 1958) improves on the previous model in two key ways: the input is now expanded to the set of real numbers, and the parameters are learned rather than manually set. This improved model is defined as follows

Definition 4.2. Consider an input to the model $x \in \mathbb{R}^n$, and learned weights $w \in \mathbb{R}^{n+1}$. We denote an augmented input $x' = [1, x]$. The function f_w learned by this model is:

$$f_w(x') = \mathbb{1}\big(x' \cdot w \geq 0\big)$$

First we note the similarities between this and McCullochs - Pitts Linear Threshold Unit. In fact, you can see closely that the functions defined by McCulloch's Pitts model are a subset of the models learned by Rosenblatt's Perceptron by setting $w = [-\theta, 1^n]$. Let us in fact show this.

Claim 4.3. *McCullochs - Pitts Linear Threshold unit is a special case of Rosenblatt's Perceptron when* $w = [-\theta, 1^n]$

$$f_w(x') = \mathbb{1}\big(x' \cdot w \geq 0\big)$$
$$f_w(x') = \mathbb{1}\big((-\theta + \sum_{i=2}^{n+1} x'_i \cdot w_i) \geq 0\big)$$
$$f_w(x') = \mathbb{1}\big((-\theta + \sum_{i=1}^{n} x_i \cdot w_{i+1}) \geq 0\big)$$
$$f_w(x') = \mathbb{1}\big(\sum_{i=1}^{n} x_i \geq \theta\big)$$

Now that we have defined the model, we must still define how it learns the weight vector w!

Definition 4.4. Rosenblatt's Perceptron Algorithm

Algorithm 1: Rosenblatt's Perceptron Update Algorithm

```
Function Perceptron(x, y, max, α):
    w_t ← Random(n + 1);
    x' ← [1, x];
    for t ← 1 to max do
        ŷ ← 𝟙(w_t · x' ≥ 0);
        w_{t+1} ← w_t + α(y − ŷ)x;
    end
    return;
;
```

Intuition: Intuitively, we can think of the update rule as follows. Consider a sample (x, y) where x is the input to the perceptron and y is the label from our training set, and the prediction is $\hat{y}$. $(y - \hat{y})$ is either $-1, 0, 1$, which denotes whether the prediction $\hat{y}$ of the perceptron was too high, correct, or too low.

To illustrate, let us consider the case where the $\hat{y}$ is too low (i.e. $y = 1$ but $\hat{y} = 0$). In this case, $(y - \hat{y}) = 1$. Therefore by setting $w_{t+1} = w_t + \alpha(y - \hat{y})x$, we are increasing the weight of w_{t+1} by αx, causing the prediction to be higher next time the sample is passed through the prediction, making it more likely the perceptron predicts correctly.

What is the purpose of the learning rate α? This is a nuanced discussion and will be covered in more detail later in the course, for now think of it as just the magnitude for which we correct on a mistaken prediction.

This intuition begs the question, what are the bounds of what a Perceptron can learn? We will show that a perceptron can learn any linearly separable function.

4.3 Proof of Convergence of Rosenblatt's Perceptron algorithm

Proof. First, assume that there exists a parameter vector w^*, and $\gamma > 0$ such that $||w^*|| = 1$ where $\forall i \in [1...m]$ where m is the number of samples in the dataset. For this proof, we define $y \in \{-1, 1\}$ instead of $y \in \{0, 1\}$.

$$y_i(x_i \cdot w^*) \geq \gamma$$

Note that here we are simply claiming that the function our dataset represents is **linearly separable**, where the equation for the line separating the classes is the unit vector w^*. Note that w^* being a unit vector is simply a convenience for the proof and does not limit generality. Any linearly separable function can simply have it's parameter vector normalized to achieve the same result.

Lemma 4.5. *Define an error as when $y_i \neq \hat{y}_i$ on iteration i of the update algorithm. We let $\alpha = 1$, and initialize $w_0 = [0]^{n+1}$, where n is the length of input feature vector x. Then on the kth error*

$$w_{k+1} \cdot w^* \geq k\gamma$$

First, note that the the choice of initialization is up to us. In practice, we randomly initialize the weight vector w as it leads to faster convergence results, but for ease of the proof, we initialize w_0 to the zero vector. We prove this lemma through induction, inducting on k.

Induction Hypothesis: Assume for an arbitrary but particular k that $w_k \cdot w^* \geq k\gamma$.
Base Case: $k = 0$
Clearly, w_0 is the zero vector therefore $0 \geq 0$, therefore this claim holds.

Induction Step: Let t be the iteration where the kth error occurs.

$$\begin{aligned}
w_{k+1} &= w_k + y_t x_t && \text{By the definition of the update rule} \\
w_{k+1} \cdot w^* &= (w_k + y_t x_t) \cdot w^* \\
w_{k+1} \cdot w^* &= (w_k \cdot w^*) + (y_t x_t \cdot w^*) \\
w_{k+1} \cdot w^* &\geq (w_k \cdot w^*) + \gamma && \text{By definition of linearly separable} \\
w_{k+1} \cdot w^* &\geq k\gamma + \gamma && \text{By Induction Hypothesis} \\
w_{k+1} \cdot w^* &\geq (k+1)\gamma
\end{aligned}$$

□

We've now shown that $w_k \cdot w^* \geq k\gamma$. We note the following:

$$\begin{aligned}
w_k \cdot w^* &\geq k\gamma \\
||w_k|| * ||w^*|| \geq w_k \cdot w^* &\geq k\gamma && \text{By Cauchy Schwarz Inequality} \\
||w_k|| * ||w^*|| &\geq k\gamma \\
||w_k|| &\geq k\gamma && \text{Since } w^* \text{ is a unit vector}
\end{aligned}$$

Intuition: Why does this make sense? We assumed that for a choice of w^* that linearly separates the dataset, $y_i(x_i * w^*) \geq \gamma, \forall i \in [1...m]$. By the update rule, when we have an error, we add a minimum (when y_i = 1) of $\gamma = x_t * w^*$. Therefore, of course after k errors we have that $||w_k|| \geq k\gamma$. You can think of γ as the margin between the linearly separable hyperplane and the closest point in our dataset to the separating hyperplane, so on every error, our weight vector increases by a minimum of that margin γ.

Lemma 4.6. *Let* $R \geq ||x_i||, \forall i \in [1..m]$. *Then* $||w_k||^2 \leq kR^2$

We prove this claim using induction on the number of errors k.

Induction Hypothesis: Assume for an arbitrary but particular k that $||w_k||^2 \leq kR^2$.
Base Case: k = 0
Since w_0 is the zero vector, and k and R are both non-negative, this claim holds.

Induction Step: Let t denote the iteration of the kth error.

$$
\begin{aligned}
||w_{k+1}||^2 &= ||w_k + y_t x_t||^2 && \text{By definition of the update rule} \\
||w_{k+1}||^2 &= ||w_k|| + y_t^2||x_t||^2 + 2(y_t x_t) \cdot w_t \\
||w_{k+1}||^2 &\leq ||w_k|| + y_t^2||x_t||^2 \\
||w_{k+1}||^2 &\leq ||w_k|| + ||x_t||^2 && \text{Because } y_t \text{ is either } -1 \text{ or } 1 \\
||w_{k+1}||^2 &\leq ||w_k|| + R^2 && \text{Because } ||x_t||^2 \leq R \\
||w_{k+1}||^2 &\leq (k+1)R^2 && \text{By Induction Hypothesis}
\end{aligned}
$$

Intuition: Before, we made the claim that on each error, since the margin was γ, $||w_k|| \geq k\gamma$ after k errors because each time a minimum of γ was added to the length of w_k. Here, we find the upper bound, by noting that R is simply the maximum length of any sample in our dataset, therefore after each error we add at most R^2 to $||w_k||^2$.

Corollary 4.7. *Let t^* to be the iteration that $\frac{R^2}{\gamma^2}$ errors have been made. It is guaranteed that $w_{t^*} = w^*$*

Since we have proved a lower and upper bound to the $||w_k||$ after k errors we can construct the following inequality:

$$
\begin{aligned}
k^2\gamma^2 \leq ||w_k||^2 &\leq kR^2 \\
k^2\gamma^2 &\leq kR^2 \\
k &\leq \frac{R^2}{\gamma^2}
\end{aligned}
$$

Since there is an upper bound on the number of errors that can be made, we have shown that the algorithm eventually converges to w^*. source

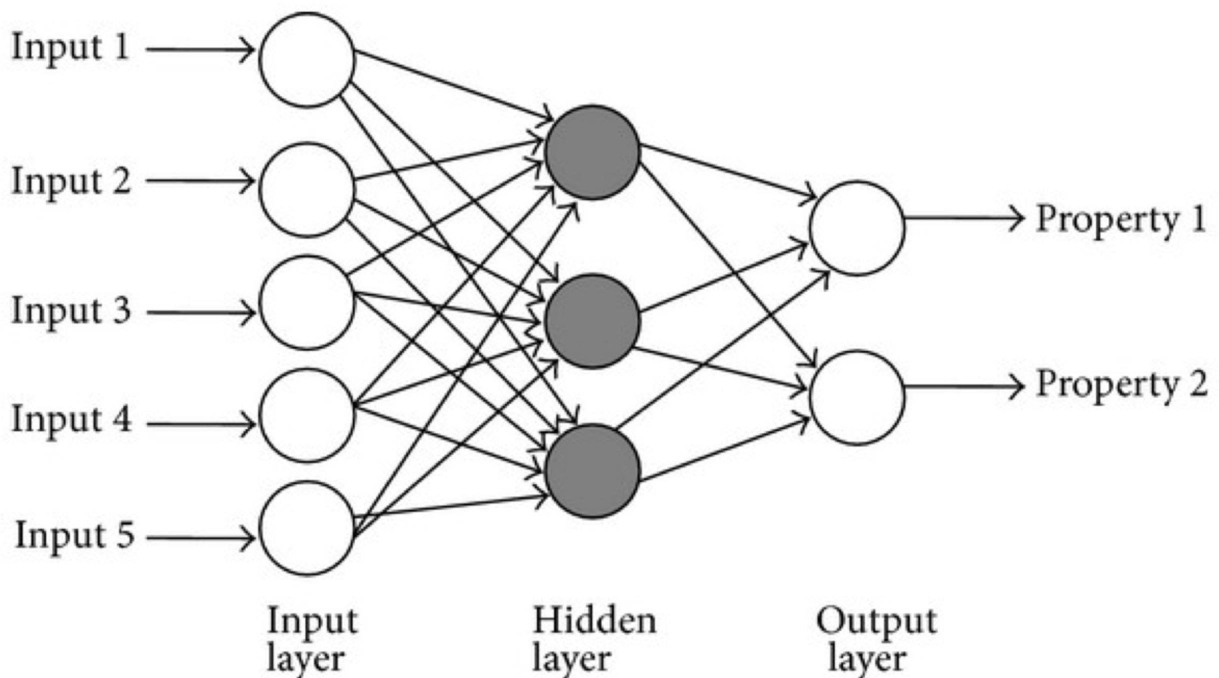
□

5 Multi-Layer Perceptrons

5.1 Neural Network Winter

A key assumption made with the Rosenblatt's Perceptron update algorithm is that the dataset must be linearly separable. This is a major issue with the model, as most functions that are useful to learn in the real world are not linearly separable. For instance, consider the XOR function, which is a very simple function yet not linearly separable! To circumvent this, one can apply a kernel function to $(x \cdot w)$, however, such approaches yielded suboptimal results compared to SVMs, which quickly became the model of choice for solving classification tasks.

So how do we fix this? A paper was published in 1960 that proved that Rosenblatt's perceptron could only learn linearly separable functions, however in this same paper they propose that potentially combining multiple of these perceptron units may overcome this issue.



This was later coined as a **Multi-Layer Perceptron**, however, no one knew how to train such a model! This was fixed in 1986, when David Rumelhart, Geoffrey Hinton, and Ronald Williams discovered how to train such multi-layer perceptrons with a technique they called **Backpropagation**.

5.1.1 Backpropagation

Training multi-layer perceptron is mathematically an optimization problem of the form

$$\min_{\theta} \sum_{i=1}^{n} l(y_i, \widehat{y_i}; \theta)$$

Where θ are the trainable parameters in the multi-layer perceptron (which in this case would be the weights and biases) and l is the loss function of choice. One common way of solving this optimization problem is by taking steps along the gradient to a minima. It helps to realise that a multi-layer perceptron can be written as one long function in terms of its input. For example, the multi-layer perceptron in the above figure (assume 1 hidden layer and no biases with sigmoid activation σ) can be written as

$$\widehat{y_i} = f(x_i; \theta) = \mathrm{Softmax}(w_2^T \sigma(w_1^T x_i))$$

where $\mathrm{Softmax}(h_i) = \frac{e^{h_i}}{\sum_{i=1}^{k} e^{h_j}}$.
Let η represent the size of the step (also called the learning rate), then the value of the parameters at time $t+1$ in terms of the value at time t is given by :

$$w^{t+1} = w^t - \eta \frac{\partial(\sum_{i=1}^{n} l(y_i, \widehat{y_i}; \theta^t))}{\partial(w^t)}$$

The gradient of any parameter in the network can be found using the chain rule and updated according to the equation presented above.

One step of backpropagation includes 2 high level operations

- Forward Pass
  We calculate the outputs (prediction) of the neural network by traversing through all perceptrons from first layer to the last layer.

- Backward Pass
  We start at the loss and successively calculate the gradients using chain rule and update the weights starting at the last layer and going back to first layer. It is important to note that the gradients for a previous layer are computed using the original values of the network parameters (θ^t).

For more details refer here.

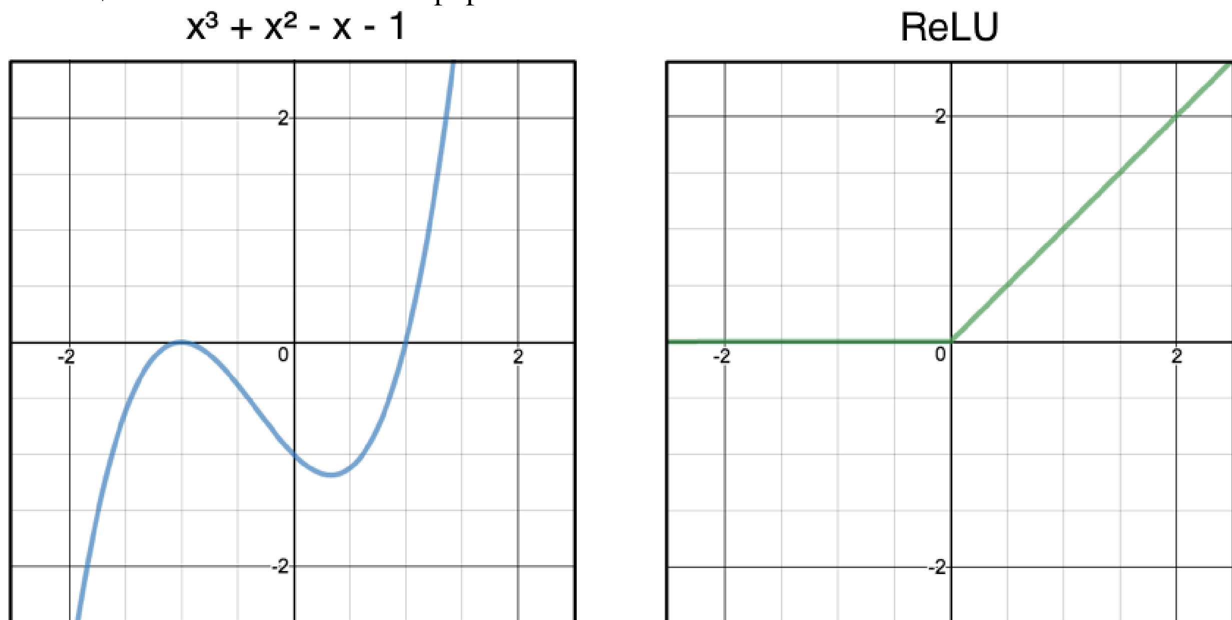
Since these are matrix operations, they can be accelerated significantly by parallel processors like GPUs, multi-core CPUs, etc.

The main benefit of backpropagation is the ease of implementation as you'll learn about gradient calculation using computation graphs in subsequent lectures.

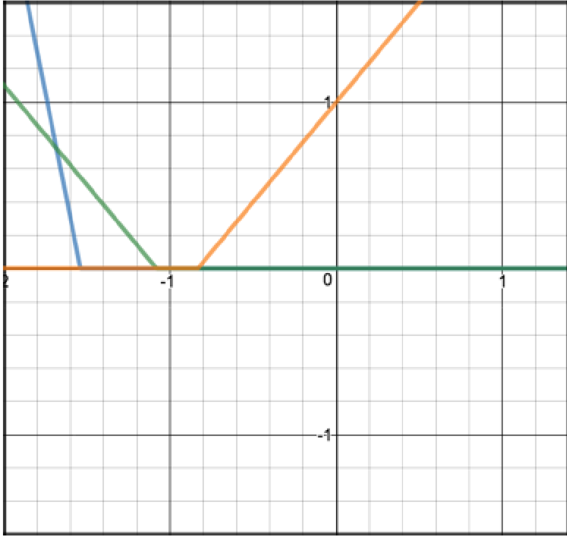
6 Universal Approximation Theorem

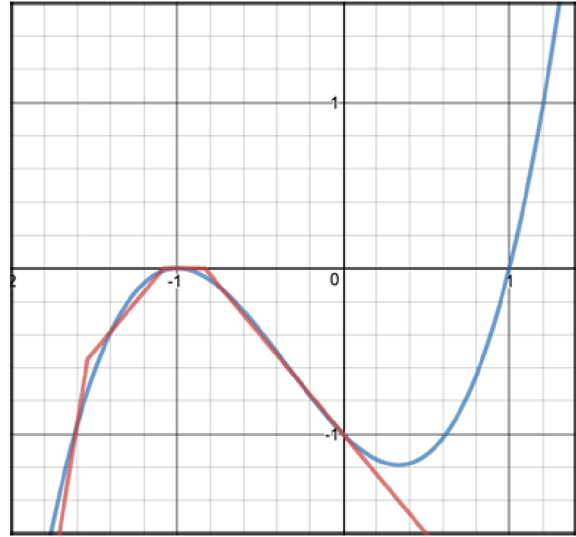
The Universal Approximation Theorem for artificial neural networks states that a feed forward network with a single hidden layer containing a finite number of neurons can *approximate* continuous functions on closed and bounded subsets of Euclidean space (provided we do not use a polynomial activation function). With that said, its important to note that such a single layered neural network might be impractical to use as it can be shown that the width of such a network will be exponentially large. Since we are talking about approximating the function, the width is inversely proportional to tolerance in the approximation.

Now let us try to understand Universal Approximation Theorem intuitively. Consider the function $x^3 + x^2 - x - 1$ and the ever popular ReLU activation function. Source

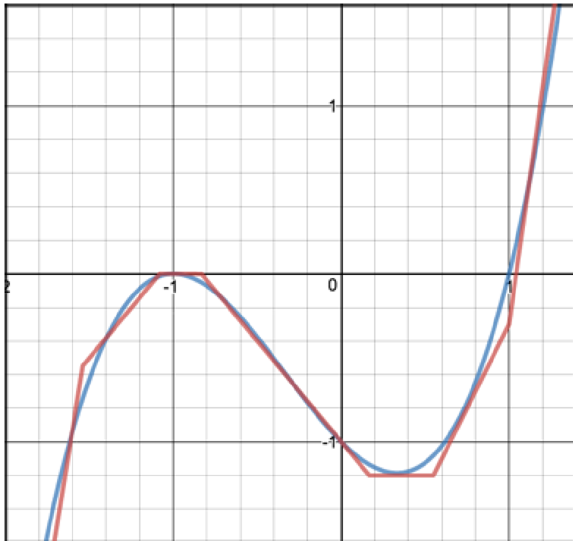


For a multi-layer perceptron with just 1 layer and several neurons, each neuron (before activation) is a linear combination of its input. This means each neuron (before activation) is a hyperplane (in our case a line). After the application of ReLU activation, we can see that the points at which pre-activated value was negative now becomes 0. We can use this observation to turn on or off the contribution of different linear combinations of the inputs.





As shown in the above image, using 3 neurons in the hidden layer, we get three different neuron activation patterns. We can then linearly combine them to approximate a part of our target function. By increasing the number of neurons to 6 in the hidden layer, we can better approximate the target function as shown below :



$$n_1(x) = Relu(-5x - 7.7)$$
$$n_2(x) = Relu(-1.2x - 1.3)$$
$$n_3(x) = Relu(1.2x + 1)$$
$$n_4(x) = Relu(1.2x - .2)$$
$$n_5(x) = Relu(2x - 1.1)$$
$$n_6(x) = Relu(5x - 5)$$

$$Z(x) = -n_1(x) - n_2(x) - n_3(x) + n_4(x) + n_5(x) + n_6(x)$$

For more details refer here or here.

7 The Explosion of Deep Learning

7.1 Neural Network Winter # 2

So multi-layer perceptrons are now trainable and can approximate any function! Sounds perfect, right? However, there were still major issues with neural networks that stalled their proliferation. Namely, although backpropagation would eventually converge to the optimal weight parameters, this convergence would be much too slow, requiring too much computing resources and data to train with, both of which were simply not accessible enough to make training multi-layer perceptrons feasible enough to be widely adopted.

7.2 Accelerating Training with GPUs

In 1994, the term "GPU" or Graphics Processing Unit, was coined. The GPU is a specialized hardware device that allows for massive parallelization of a specific set of operations, specifically floating point operations. In 1999 Nvidia released the GeForce 256, which was landmark device, allowing for efficient transformations, rasterization and rendering in graphics applications. This revolutionized the world of gaming and by 2009 GPUs had gotten much better than in 1999, significantly increasing in FLOPS (floating operations per second) and memory bandwidth (data transferred to the GPU per second), allowing gaming processes to be sped up significantly. This trend in fact, is still continuing with the increase in FLOPS and memory bandwidth per year nearly being exponential.

In 2009 Nvidia, one of the largest GPU manufacturers, explored further applications of the GPU and realized that it could be used to speed up training of neural networks! Google Brain quickly followed, and soon interest in deep learning immediately resurged, as this technology suddenly became feasible.

Why do GPUs speed up training? GPUs are optimized for floating point operations, and in 2004, researchers at Stanford discovered how to use GPUs to speed up matrix multiplication. As described in the backpropagation section, both the forward and backward passes in a deep neural network can be represented as a series of matrix multiplications. Therefore, can be optimized significantly!

CUDA or Compute Unified Device Architecture is a parallel computing platform and programming model that makes using a GPU for general purpose computing simple and elegant. A GPU contains many (cuda) cores which can be directly programmed by using CUDA to utilize massive parallelization capabilities in the GPU. For example an Nvidia RTX 2080 Ti contains 4352 cuda cores which are capable enough to perform simple floating point operations. Compare this to recently unveiled AMD threadripper 3990x which has just 64 cores. Another key factor is the memory. GPUs have limited amounts (11 GBs for RTX 2080 Ti) whereas one can easily have

several 100 GBs of RAM. But the bandwidth at which data can be moved is more important. Comparing CPUs to GPUs here, Intel Core-i9 9990XE has a peak memory bandwidth of 79.47 GB/s when using quad channels whereas RTX 2080 Ti has a peak memory bandwidth of 616 GB/s which easily outperforms the CPUs in simple tasks involving movement of data in batches that are of size comparable to GPU memory capacities (Hint : Datasets).

7.3 Datasets

We mentioned earlier, that there were two issues, computation power and training data. GPUs help mediate the issue of computation power, but there's still not enough data!

In 2010, Fei-Fei Li of Stanford led one of the largest data collection efforts, using Mechanical Turk at a large scale to construct labels for image data. When published, this dataset contained on the order of *1 million* images with annotations! Many institutions and companies have since followed suit, curating large datasets and releasing them for public research. For instance, in 2017 Google AI introduced OpenImages, which contains *9 million* images, with around 2.8 million segmentation masks, 16 million bounding boxes and much more.

In addition, websites have sprouted to allow for easy access to datasets. For instance, Kaggle now allows researchers to access thousands of datasets for exploratory purposes.

At the same time these advances were happening, internet speeds around the world were rapidly expanding, allowing researchers to feasibly download these massive datasets, making data significantly more accessible.

7.4 Luck

As you can see, the rapid proliferation of deep learning today was a conglomeration of discoveries that fortunately occurred nearly simultaneously, creating the perfect conditions for deep learning to become the powerful technique that it has become today. Without any of, GPUs, creation of datasets or increase in internet speeds, advancements in the field may have slowed significantly compared to what it is today.

8 What next?

Deep learning, for all it's applications, still has significant problems. It suffers issues regarding transfer learning (using a learned model for a task it didn't train on), requires too much data, extrapolates poorly, and much more. With how much the world has changed from just the advancements in deep learning in the decade of 2010, it's exciting to see how researchers will tackle some of the remaining issues, and how the world will change as a result. As such, we on CIS 522's staff are excited to teach you about this technology, and hope you will find the information useful for future pursuits as well as to satiate your own curiosity.

9 What you might enjoy reading

On the webpage we will be compiling for each lecture relevant reading materials. Reading it might be fun, and produce a broader understanding yet of the field. It is nowhere near mandatory. This is not mandatory reading, just a set of historically meaningful papers.