Jasper Robbins

R Tillquist

CSCI 411

07 April 2024

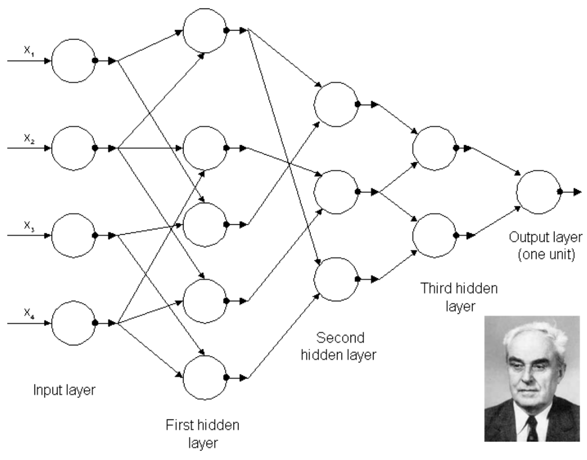
Backpropagation

INTRODUCTION

Artificial Intelligence has become a foundational element across various industries, transcending traditional tech boundaries to find applications among all operations. This widespread integration points to AI's transformative potential, yet the intricate mechanisms driving its core functionalities remain a mystery to many. This report seeks to demystify one of the most pivotal processes underpinning AI's process: backpropagation.

A BRIEF HISTORY

The first AI models had nothing to do with backpropagation. As a matter of fact, the earliest deep learning algorithms I could find were from 1965, where Alexey Ivakhnenko, a ukrainian mathematician, and his associate V.G. Lapa created small yet functional neural networks which used a 'survival of the fittest' type logic, in which each layer of the neural network selected the best features through statistical methods and forwarded them to the next layer. These neural networks did not use backpropagation in training, opting instead for a layer-by-layer least squares fitting. While it did not use backpropagation, Ivakhnenko's work laid foundational concepts for neural network architectures, which was the crucial next step for advancements towards backpropagation. In a more advanced model, according to Nvidia [2], the first convolutional network would be in 1979. This would be done by Japanese computer scientist, Kunihiko Fukashima.

 x_1 x_2 x_3 x_4 Input layer First hidden layer Second hidden layer Third hidden layer Output layer (one unit)	*This illustrates the basic logic of Ivaknenko's neural network, showcasing a 'survival of the fittest' type formula.*

Continuing on throughout the history of deep learning, the ideas of AI continued to grow, yet most of it remained theoretical. For example, backpropagation was first theory crafted in the 1970s by Seppo Linnainmaa, who is credited with the earliest formulation of the basic principles behind the backpropagation algorithm, although his theories and work were not in the context of neural networks. Linnainmaa's method was originally designed to calculate derivatives of functions efficiently and accurately, and his work would lay the groundwork for backpropagation by introducing automatic differentiation, which is a broader concept than backpropagation, but it encompassed it and was essential for its development. As time continued, we continued to see interest in the topic grow, with various contributions such as Paul Werbos in 1974, who titled his Harvard University PhD thesis, 'Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences', but we wouldn't see an actual application of backpropagation in neural networks until the late 1980s. In this time period, Yann LeCun, a French American computer scientist, would develop and apply a variant of the backpropagation algorithms to a neural network for the purposes of recognizing and classifying handwritten digits for zip codes. Yann would publish various papers, with his main publication being in 1989, in a paper titled

‘Backpropagation Applied to Handwritten Zip Code Recognition’, where he demonstrated and discussed the practical utility of backpropagation in training neural networks for real life tasks. Yann’s work was very influential in the field, but was swept under the rug due to an ‘AI Winter’ , which is a term coined to discuss periods of time where AI received global reduced funding majorly due to lack of interest. According to NVIDIA [2], The first AI winter would occur due to researchers making promises of strong AI advancements, leading to heavy AI funding, with little to no results. The lack of results would lead to dramatic funding cuts as well as dramatic drops in interest. Even during these AI winters, interest was not fully dead, fewer researchers continued down the path of research for AI. The two ‘AI Winters’ were approximately from 1974-1980, and 1987-2000. Yann’s work would once again come to light and receive interest in the 2000s, mainly driven by advancements in computing power, availability of datasets, and various other breakthroughs, demonstrating the effectiveness of deep learning techniques.

In 2012, AlexNet, a deep learning convolutional neural network was the next major step in showcasing backpropagation’s effectiveness, as it was an image classification system. AlexNet was a convolutional neural network developed by Alex Krizhevsky, with aid from Ilya Sutskever and Geoffrey Hinton to compete in the ‘ImageNet Large Scale Visual Recognition Challenge’. The team behind the creation of AlexNet would be based from University of Toronto, which is a university that remains at the forefront of AI/ML today. It achieved a 15.3% error rate, which was significantly lower than the second-place error rate of 26.2%. AlexNet was fundamentally different from the other computer vision softwares, basing itself on GPUs for training, as well as using ReLU activation function for non-linearity, which helped fix the vanishing gradient problem, allowing deeper networks to be trained more efficiently. AlexNet’s success is also attributed to the use of backpropagation for efficient training of deep networks. After AlexNet’s success in 2012, deep learning models continued to dominate major AI competitions. Each successive year saw new models more accurate and sophisticated than the previous, such as ZFNet in 2013, GoogLeNet in 2014, and ResNet in 2015. If it wasn't clear before, we were officially out of the ‘AI Winter’, AI was taking off.

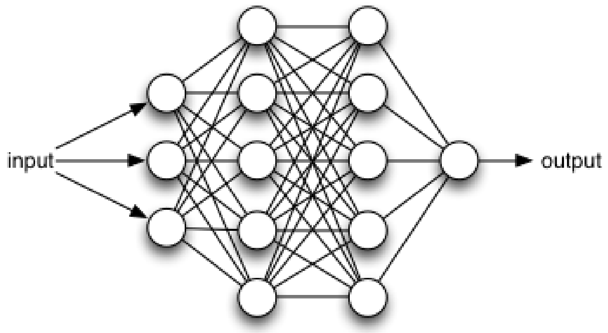
It wasn't until 2015, that TensorFlow was released by Google, and PyTorch was released just a year later, in 2016, by a Facebook AI research team. TensorFlow would fall out of favor for the AI community, but would re-emerge with TensorFlow 2.0. The two libraries for creating models continue to fight over space, but their existence alone rocketed AI’s development in deep learning, as it made the industry as a whole more accessible. While backpropagation is distinct from the broader field of artificial intelligence, it plays a crucial role in its evolution. Next, we will explore backpropagation comprehensively, diving deep into its current state, advancements, and potential areas for future development.

INTUITION

Backpropagation is the cornerstone for neural network training, responsible for fine tuning the weights of the networks based on the error between predicted and actual output. As it is with most machine learning concepts, we can trace backpropagation’s conceptual base to calculus. More specifically, backpropagation uses the chain rule to compute gradients of the loss function with respect to each weight in the network. In slightly more mathematical terms, we are talking about partial derivatives when we talk about gradients. These gradients tell us how loss changes with each according value in weight, guiding the update process to minimize the loss.

Now that we know roughly what backpropagation is aiming to do, let's talk about the ‘how’ in the actual sense of a neural network. The very rough format of training a neural network involves a few broad steps. We will quickly discuss each step, going more in depth for the backwards step.

First, the forward pass, where input data is fed forward through the network, layer by layer, until predictions are made to the output layer. During this step, before the model learns anything, this is a random prediction that will generally yield horrible errors, but this is why we have various other features and strategies such as learning rate, which is a scale of how much weights can change during any given pass. Either way, regardless of the error, we end up in the output layer with a ‘prediction’ at the end of the forward pass.

	*This image shows a rough neural network, illustrating the general process, but more relevantly, visually showing what each layer is called and looks like. With the input layer being the first layer, the output layer being the final layer, and the hidden layer being everything in between.*

At this point, we now need to calculate loss. Loss is calculated through whatever loss function the neural network. In my neural network, I opted to use the huber loss function, in an attempt to be less sensitive to outlier data, although loss functions in general’s purpose is simple, to tell the model how off the prediction was from the expected.

With the loss calculated, the backwards pass begins, which is where backpropagation occurs. This step involves using the loss calculated L, and meticulously calculating the gradients of L with respect to each network weight w, denoted as $\frac{\partial L}{\partial w}$. This gradient delineates how much alterations in a specific weight w influenced the loss L. This overall process involves iteratively using the chain rule in a reversed sequence from the output layer, heading towards the input layer, iterating through weights of the hidden layer. Backpropagation propels the gradients, or error sensitivities, throughout the network. For any given neuron with activation *a* and input *z*, where $z = Wa + b$ where *W* represents weights, and *b* represents biases, the process calculates the gradient of the loss with respect to the neuron's output $\frac{\partial L}{\partial a}$, and then uses this to find $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$. The essence of this operation lies in its ability to parse through the network's structure in reverse, applying $\delta^{(l)} = (W^{(l+1)T}\delta^{(l+1)}) \odot f'^{(l)}(z^{(l)})$ for each layer *l* where $f'^{(l)}$ is the derivative of the activation function. This is further elaborated on in [8]. To simplify this further, this equation is used to compute the error term for each layer of a neural network during the backpropagation process. This is rarely done in 1 line, as this computation generally gets fairly heavy. This process is not itself the chain rule, although it embodies the chain rule's application in the context of neural network backpropagation. In the process, we dissect the aggregate error to attribute specific fractions to individual weights, giving the model insights towards minimizing the loss function *L* through precise weight adjustments.

Now in the process, we are back at the input layer, we need to update weights and biases of our neural network based on the gradients calculated during the backwards pass. This is a process that generally involves some optimization algorithm, although regardless of the algorithm used, the calculations use the gradients computed during the backwards pass. Weights are adjusted in a way to minimally decrease the loss, controlled by the learning rate, which is a scale at which we allow the model to learn.

Finally, to finish the process, we need to commence another forward pass, with the updated weights. At this point, we have finally completed a single loop or training epoch.

PSEUDOCODE

```
X = Input data matrix

Y = Target output data matrix

Z1 = Pre-activation values at hidden layer

A1 = Activation values at hidden layer

Z2 = Pre-Activation values at output layer

output = Predicted output from forward pass

lr = Learning Rate

lambd = Regularization parameter
```

```
Function Backwards_Pass(X,Y,Z1,A1,Z2,output,lr,lambd):

    m = number of examples

    dWeights = {}

    dBiases = {}

    dZ = output - Y

    for i in number of layers to 1:

        dWeights[i] = dot(activations[i-1].T,dZ) / m + (lambd * weights[i]) / m

        dBiases[i] = sum(dZ) / m
```

```
            dA = dot(dZ, weights[i].T)

            dZ = dA * reluDerivative(activations[i-1])

      dWeights[1] = dot(X.T, dZ) / m + (lambd * weights[1]) / m

      dBiases[1] = sum(dZ) / m

      for i from 1 to number of layers:

            weights[i] = weights[i] - lr * dWeights[i]

            biases[i] = biases[i] - lr * dBiases[i]
```

DETAILED DESCRIPTION

First, we will showcase the actual implementation, compare it to the pseudocode, then have a conversation about backpropagation in its current state and where it's headed.

```
    def backward_pass(self, X, Y, Z1, A1, Z2, output, lr, lambd):

        # Get number of examples to normalize gradient

        m = Y.shape[0]


        # Calculate gradient of the loss with respect to Z2

        dZ2 = output - Y



        # Calculate gradients for weights and biases of output layer, including
regularization for weights

        dW2 = (1 / m) * np.dot(A1.T, dZ2) + (lambd / m) * self.W2

        db2 = (1 / m) * np.sum(dZ2, axis=0, keepdims=True)


        # Backpropagate errors from output to hidden layer
        dA1 = np.dot(dZ2, self.W2.T)


        # Apply derivative of ReLU to get gradient of loss with respect to
pre-activation of first layer
        dZ1 = dA1 * self.relu_derivative(Z1)


        # Compute gradient for weights and biases of first layer
        dW1 = (1 / m) * np.dot(X.T, dZ1) + (lambd / m) * self.W1
        db1 = (1 / m) * np.sum(dZ1, axis=0, keepdims=True)


        #  Employ gradient clipping to prevent exploding gradients
        grad_clip_size_neg = -.5
        grad_clip_size_pos = abs(grad_clip_size_neg)
        dW1 = np.clip(dW1, grad_clip_size_neg, grad_clip_size_pos)
        db1 = np.clip(db1, grad_clip_size_neg, grad_clip_size_pos)
        dW2 = np.clip(dW2, grad_clip_size_neg, grad_clip_size_pos)
        db2 = np.clip(db2, grad_clip_size_neg, grad_clip_size_pos)


        # Print gradients
        print("dW1:", dW1)
        print("db1:", db1)
        print("dW2:", dW2)
        print("db2:", db2)


        # Update weights and biases
        self.W1 -= lr * dW1
        self.b1 -= lr * db1
        self.W2 -= lr * dW2
        self.b2 -= lr * db2
```

As we can see, there are some distinct differences. My implementation of a neural network is a concise neural network, with one input layer, one hidden layer, and one output layer, streamlining the backpropagation process without needing to iterate over multiple layers of nodes. This differs from the pseudocode, where we use a generalized approach that accounts for any amount of layers with dictionaries and for loops. In my implementation, we also aim to add stability by implementing a few strategies outside of the normal, mainly gradient clipping, which is a process where we restrict the gradient values to a predefined range. This helps with an issue we experienced during the construction, gradient explosion. This overall neural network architecture was aimed to simplify as much as possible within a neural network without taking away from the underlying features that makes it work.

RUN TIME ANALYSIS

Looking at the code, we will perform a quick sweep and analyze the runtime of backpropagation. This includes anything within the function backwards_pass. To clearly analyze the runtime, I think it actually works out best to just re-display the code, and walk through line by line with comments discussing the runtime before making final statements about its runtime.

```
    def backward_pass(self, X, Y, Z1, A1, Z2, output, lr, lambd):

        m = Y.shape[0] # O(1)
        dZ2 = output - Y # O(n) where n is size of output (a matrix of predicted outputs)

        dW2 = (1 / m) * np.dot(A1.T, dZ2) + (lambd / m) * self.W2 # O(n^2) where n is size
of A1 or dZ2, which are both matrices.

        db2 = (1 / m) * np.sum(dZ2, axis=0, keepdims=True) # O(n) where n is size of dZ2
        dA1 = np.dot(dZ2, self.W2.T) # O(n^2) where n is size of dZ2 or self.W2

        dZ1 = dA1 * self.relu_derivative(Z1) # O(n) where n is size of dA1 or Z1

        dW1 = (1 / m) * np.dot(X.T, dZ1) + (lambd / m) * self.W1 # O(^2) where n is size of
X or dZ1

        db1 = (1 / m) * np.sum(dZ1, axis=0, keepdims=True) # O(n) where n is size of dZ1

        grad_clip_size_neg = -.5 # O(1)

        grad_clip_size_pos = abs(grad_clip_size_neg) # O(1)

        dW1 = np.clip(dW1, grad_clip_size_neg, grad_clip_size_pos) # O(n) where n is size of
dW1

        db1 = np.clip(db1, grad_clip_size_neg, grad_clip_size_pos) # O(n) where n is size of
dW1

        dW2 = np.clip(dW2, grad_clip_size_neg, grad_clip_size_pos) # O(n) where n is size of
dW1

        db2 = np.clip(db2, grad_clip_size_neg, grad_clip_size_pos) # O(n) where n is size of
dW1



        print("dW1:", dW1) # O(1)
        print("db1:", db1) # O(1)
        print("dW2:", dW2) # O(1)
        print("db2:", db2) # O(1)
        self.W1 -= lr * dW1 # O(n) where n is number of elements in dW1

        self.b1 -= lr * db1 # O(n) where n is number of elements in db1

        self.W2 -= lr * dW2 # O(n) where n is number of elements in dW2

        self.b2 -= lr * db2 # O(n) where n is number of elements in db2
```

This overall analysis of runtime of each line of code leaves us with an overall time complexity of O(n^2) due to primarily the dot product operations. As we can see, a lot of matrix multiplication.

FUTURE OF BACKPROPAGATION

Backpropagation remains a foundational technique for training deep neural networks, seemingly, where we are now, there are many kinds of neural networks that exist, but in seemingly all of them, backpropagation remains a stable point in all of these networks. Despite its relevance, backpropagation does face challenges that are yet to be handled, we will discuss the most relevant ones.

EXPLODING/VANISHING GRADIENT

This problem is one I actually ran into, it occurs when derivatives of the activation functions in the network are less than one, during backpropagation, the repeated multiplication of these small numbers may lead to a vanishing gradient, bottlenecking the neural network’s overall ability to learn. Inversely, exploding gradient is the opposite, when values are too high, leading to inaccurate results on the other end of the spectrum. Both phenomena are exposed in deep networks, and fundamentally limit the depth and complexity of trainable models until addressed.

PARALLELIZATION

By nature, backpropagation is a sequential algorithm, where gradients need to be computed layer by layer, in reverse order. Within each calculation, some calculations can be made in parallel, but the overall scheme of layer by layer backpropagation introduces a sequential bottleneck, as each calculation requires knowledge of the previous calculation.

COMPUTATIONAL EFFICIENCY

Especially when working with large networks and datasets, backpropagation generally involves storing a good bit of data that is required for gradient calculations, leading to issues with memory space. This is synonymous to training most neural networks, but it takes a lot of resources. This is why Amazon and Google have services to help people train models through their services such as Microsoft Azure, IBM Cloud, Google Cloud, etc…

WHERE IS BACKPROPAGATION GOING

As discussed, despite being universally used, backpropagation faces challenges mainly in training very deep networks. Currently, research into alternative network architectures that circumnavigate some of backpropagation's limitations are in the works. For example, Capsule Networks (Capnet) is a emerging neural network format that uses backpropagation, but introduces a dynamic routing mechanism and capsule structure to mainly address the 'Picasso problem' in image recognition, where images have all the right parts but that are not in the correct spatial relationship. Capsule neural network exploits the fact that while viewport changes have nonlinear effects at the pixel level, they have linear effects at the part/object level. On the other end, neural networks are being created that don't use backpropagation.On the other hand, Spiking Neural Network (SNN) is a neural network that draws inspiration from biological processes of the human brain, utilizing discrete spikes between neurons, introducing temporal dynamics into neural processing. The adoption of SNN necessitates a departure from traditional backpropagation due to their non-differentiable nature, instead, SNNs rely on biologically inspired rules like Spike-Timing-Dependent Plasticity or innovative adaptations of backpropagation to accommodate for the new structure. In essence, backpropagation is not going anywhere, although the industry is changing rapidly. As it goes with most algorithms, there is always a better implementation, just often we aren't aware of it yet. That fact is the same for backpropagation. It works, it's great, but it could be better, and we are finding out in real time the ceiling of AI.

FINAL NOTE

Further Analysis and Discussion of code found in repository at

github.com/JRobbinsss18/Backpropagation-in-Neural-Networks

Works Cited

[1] Schmidhuber Jurgon (2014), *Who Invented Backpropagation*, IDSIA,

https://people.idsia.ch/~juergen/who-invented-backpropagation.html

[2] Dettmers Tim (2015 Dec 16), *Deep Learning in a Nutshell: History and Training*, NVIDIA,

https://developer.nvidia.com/blog/deep-learning-nutshell-history-training/

[3] AI Winter, Wikipedia, https://en.wikipedia.org/wiki/AI_winter

[4] Yann LeCun, Wikipedia, https://en.wikipedia.org/wiki/Yann_LeCun

[5] Valchanov Iliya (2023 June 15), *Backpropagation - The Math Behind Optimization,* 365, Datascience

https://365datascience.com/trending/backpropagation/

[6] 365 Data Science (2017 Nov 10), *Backpropagation - The Math Behind Optimization,* Youtube,

https://www.youtube.com/watch?v=Z_m4HQYQfU4

[7] Maximinusjoshus (2021 Nov 29), *The Math behind Backpropagation Every Move Finally Explained*,

Medium https://medium.com/featurepreneur/the-mathematics-of-backpropagation-4b114fd64a63

[8] Nielson Michael, *The backpropagation algorithm*, LibreTexts Engineering,

https://eng.libretexts.org/Bookshelves/Computer_Science/Applied_Programming/Book%3A_Neural_Networks_and_Deep_Learning_(Nielsen)/02%3A_How_the_Backpropagation_Algorithm_Works/2.03%3A_The_backpropagation_algorithm