Debounce

Koral Kulacoglu, Ian Patrick Tan

April 8, 2021

Contents

1 Introduction **1**

2 Procedure **2**

3 Observations **4**

4 Conclusion **5**

5 Acknowledgments **5**

1 Introduction

Feed-forward neural networks are computer systems that learn through artificial neural pathways. They have the ability to learn patterns through propagating backwards to adapt to any given input. To do this, they store multi-layered/dimensional lists of numbers called weights and change them based on how they transform the inputs to form outputs. When the output is calculated, the network's error rate, also known as its cost is calculated through taking the square of the difference between the network's outputs and real outputs. Finally, when the cost is calculated, it is used for backpropagation, which involves the adjustment of each weight of the network through a series of gradient calculations, also known as optimizations. This report involves the documentation of a new model of neural network backpropagation optimization with advanced velocity calculations in non-parallel

layers with the purpose of preventing the network from missing the locally minimal error rate. Debounce, named after this effect, is a non-parallel neural network optimization method. It works by using the velocity of the previous weight in the layer to calculate the current weight's new gradient for optimization. This creates stronger weight relevance for the cost of non-parallelizing the network, hence sacrificing speed for accuracy.

2 Procedure

$$\theta_{t+1} = \theta_t - \left(\left(\beta - \alpha\left(tanh(V_t - V_{t-1})\right)\right)V_t + \eta g_t\right)$$

Figure 1: Debounce Weight Update

Figure 1 shows the formula for a weight update used in Debounce. Beta, (β) represents the velocity of the algorithm's decay. The higher it is, the faster the algorithm descends down to a local minimum, vice-versa. The concept behind this formula was to control beta relative to the change in velocity (V_t - V_{t-1}) to slow down when the optimizer is nearing minimal error that weight can produce, and speed up otherwise. The tanh function is set to limit delta velocity between one and negative one, and the range (α) is there to scale it optimally. The rest of the optimizer resembles the update rule of any typical optimizer, forming the new velocity, and adding the learning rate (n) multiplied by the current gradient (g_t) of the optimizer.

```python
def adjustWeight(self, neuron, weight):
      newWeights = deepcopy(self.weights)
      newWeights[neuron][weight] += self.dx
      return (nn.layerCost(self.inputs, newWeights, self.outputs) -
nn.layerCost(self.inputs, self.weights, self.outputs)) / self.dx


gradient = self.adjustWeight(neuron, weight)
```

Figure 2: Gradient Calculation

In figure 2, the gradient of the function is calculated through changing the current weight of the layer by an extremely small value such as 0.001 and taking the difference of produced errors of the weights and dividing it again by 0.001.

```python
velW = (beta - scale * tanh(velW - preVelW)) * velW + gradient * rate
```

Figure 3: Velocity Calculation

In figure 3, the new velocity is calculated with the formula mentioned in figure 1.

```python
newWeights[neuron][weight] -= (beta - scale * tanh(velW - preVelW)) *
velW + rate * gradient

preVelW = velW
```

Figure 4: Weight Adjusting

In figure 4, the individual weight is finally changed based on the new velocity (velW) that is calculated with the formula aforementioned in figure 1. Finally, the previous velocity is set for the next iteration of weights.

3 Observations

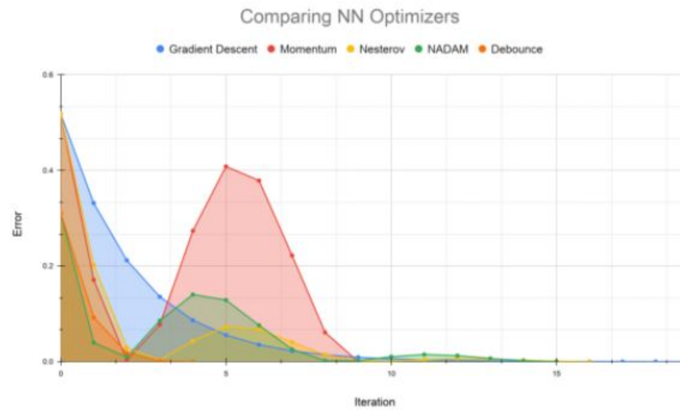

Figure 5: Finding the Local Minimum of y=x^2

Figure 5 shows the iteration speed at which different neural network optimizers find the local minimum of a typical quadratic function. It can clearly be seen that Debounce is the quickest when it comes to finding the local minimum of a simple function.

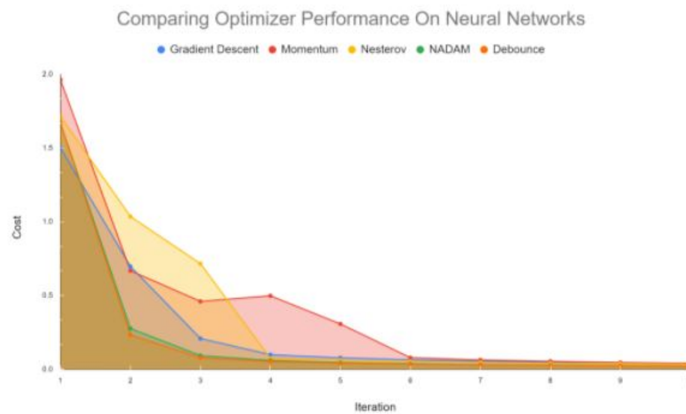

Figure 6: Optimizing a Simple Neural Network

When tested with neural networks given a small dataset, it is only slightly faster than NADAM. This can be seen in figure 6 in which its cost is very close to NADAM throughout.

4 Conclusion

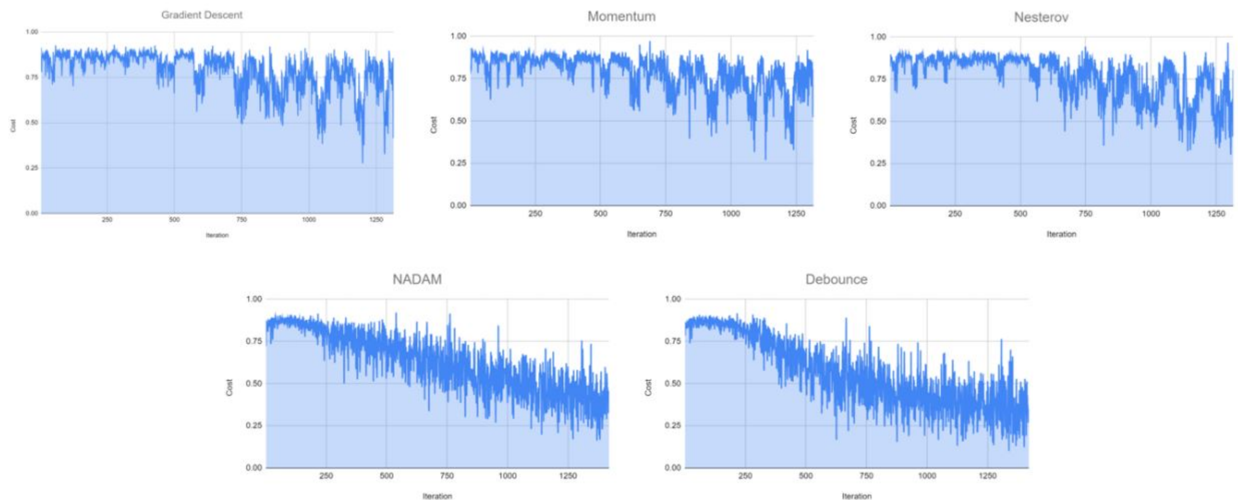

Figure 7: Optimizing a Simple Neural Network

Figure 7 shows the final optimization results of the five optimizers after training on digit recognition for 9 hours. It can clearly be seen that Debounce rapidly allows for the neural network to decrease its cost. From these results, it can be concluded that a Debounce-optimized neural network is not only the quickest to learn, but also the most accurate when it comes to non-parallel neural networks. Although this method is not be too useful for large-resolution images as of now, it can still be effectively applied to simpler classification problems such as digit recognition. Perhaps, an implementation of this optimizer can be added to a neural network library such as TensorFlow for better memory allocation as well as faster speed.

5 Acknowledgments

Special thanks to our math teacher Ilan Tzitrin for supporting us in our endeavours. We would also like to thank 3Blue1Brown for making visually appealing calculus courses and the NumPy library for existing.

References

Doshi, S. (2019, January 13). Various Optimization Algorithms For Training Neural Network. https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6.

Gad, A. (2018, June 27). Beginners Ask "How Many Hidden Layers/Neurons to Use in Artificial Neural Networks?". https://towardsdatascience.com/beginners-ask-how-many-hidden-layers-neurons-to-use-in-artificial-neural-networks-51466afa0d3e.

Gupta, T. (2017, January 5). Deep Learning: Feedforward Neural Network. https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7.

KoralK5, IanPatrickMTan. (n.d.). KoralK5/ScienceFair2021. https://github.com/KoralK5/ScienceFair2021.

KoralK5, IanPatrickMTan. (n.d.). KoralK5/YRSTF2021. https://github.com/KoralK5/YRSTF2021.

McGonagle, J., Shaikouski, G., Williams, C. (n.d.). Backpropagation. https://brilliant.org/wiki/backpropagation/.

Ruder, S. (2016, January 19). An overview of gradient descent optimization algorithms. https://ruder.io/optimizing-gradient-descent/index.html#nesterovacceleratedgradient.

Sharma, S. (2017, September 6). Activation Functions in Neural Networks. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.