18.0851 Project: Machine Learning from Scratch: Stochastic Gradient Descent and Adam Optimizer

James Gabbard and Daniel Miller

December 2018

1 Introduction

Machine learning is an increasingly important field of study with a wide range of applications. From facial recognition in social media to aiding doctors in diagnosing patients, it is quickly spreading throughout daily life. To accomplish these tasks, it is almost always necessary to leverage the learning capabilities of neural networks. Such learning depends on having a means of training these neural networks and, since many AI libraries such as Tensorflow provide prepackaged optimizers, the process of actually updating the parameters that compose a neural network is often overlooked. This project seeks to remedy that by explaining the theory behind two popular optimizers: Stochastic Gradient Descent (SGD) and Adapative Moment Estimation (Adam). First, a brief overview of neural networks will be provided. Then, SGD and Adam will both be discussed, with a detailed subsection on backpropagation. Finally, the performance of both optimizers will be evaluated in a classification problem and in a reinforcement learning (RL) problem. All of the code for this project was written in MATLAB with no pre-existing code being used.

2 Neural Network Overview

At its simplest, a neural network is a collection of matrix calculations that takes an input and produces an output. In a classification problem, a neural network would be provided with labelled data and would be tasked with producing the correct label for a given input. In reinforcement learning, neural networks are used to select a correct action given the state of an agent. These applications will be explained in greater detail in Sections 5 and 6.

The dimensions of a neural network are described in terms of depth and width. Depth refers to the number of hidden layers, or layers between the input layer and the output layer. The width of a neural network refers to the number of neurons in each layer. The weights need to be of a dimension that can take the input of one layer and transform it into the appropriate size of the next layer. The biases are then added to the product of the weights and the input.

As a demonstration, consider the neural network in Fig. 1. It has a depth of 2, an input and output of width 2, and hidden layers with a width of 4. If the input x is

$$\begin{bmatrix} x_{11} & x_{12} \end{bmatrix} = \begin{bmatrix} 1 & 2 \end{bmatrix} \tag{1}$$

and the first set of weights and biases are

$$\begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \\ W_{31} & W_{32} \\ W_{41} & W_{42} \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} \tag{2}$$

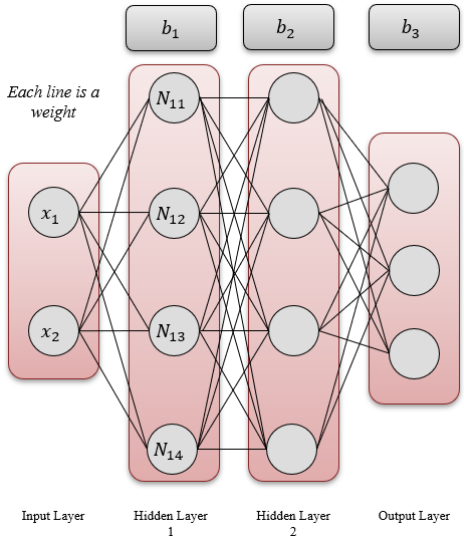


Figure 1: A Neural Network

$$b_1 = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix} \tag{3}$$

then the neurons of the first hidden layer would receive a value of

$$N = Wx^T + B \tag{4}$$

$$\begin{bmatrix} N_{1,1} \\ N_{1,2} \\ N_{1,3} \\ N_{1,4} \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix} \tag{5}$$

$$\begin{bmatrix} N_{1,1} \\ N_{1,2} \\ N_{1,3} \\ N_{1,4} \end{bmatrix} = \begin{bmatrix} 5.50 \\ 11.50 \\ 17.50 \\ 23.50 \end{bmatrix} \tag{6}$$

These values are then sent through an activation function in an element-wise manner in order to add nonlinearities to the network. Common examples of these functions include sigmoid, tanh, and rectified linear unit (ReLU). The output of these functions becomes the input to the next layer in the network, and this process repeats until the output layer is reached.

In order for a network to learn, a method must be devised to shape these weights and biases into producing an accurate output. The solution that has been devised is to define and minimize an objective function. To better understand how this works, consider a common classification problem in which a neural network is being used to differentiate pictures containing a cat from a picture containing a dog. When the data set of images is produced, each image will receive a label – a 0 for a cat and a 1 for a dog, for example. When the neural network receives each image as an input, it will then produce a 1 or a 0 as an output depending on what it believes the image contained. The objective function is then the error between the image labels and the corresponding output of the neural network.

3 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an optimization technique most commonly applied to neural networks. The stochastic part of its name refers to randomly selecting training data from a larger data set. This reduces the computational cost of the learning process. The second portion of the name, gradient descent, refers to how the algorithm goes about updating the weights and biases that compose the neural network. Gradient descent optimizes a network by minimizing an objective function, $J(\theta)$, where θ represents the weights and biases that compose the neural network. The algorithm calculates the gradient of the objective function with respect to the neural network parameters, $\nabla_\theta J(\theta)$, and then updates the parameters in the opposite direction. That is to say, it updates the network with the negative of the gradient, $-\nabla_\theta J(\theta)$

3.1 Preliminary Steps

Before using SGD, an applicable set of training data must be created. A batch of labelled images such as that which was discussed in Section 2 would be acceptable. Next, the neural network's weights W and biases b must be initialized. There is no single way of initializing a network and information on different techniques can be found on the internet.

The process of learning can now begin. First, a minibatch of data is created by randomly selecting data points from the larger pool of training data. In the aforementioned classification problem, this would be a random subset of photos. In a reinforcment learning application, this would be a minibatch of one or more tuples containing the state and reward data that the agent received for its actions (see Section 6). These steps are described in Algorithm 2 lines 2-6. Next, backpropagation is applied.

3.2 Backpropagation

All gradient based optimization strategies (SGD, ADAM, AdaGrad, and RMSprop, and others) require the gradient of the network's objective function with respect to the network's parameters. This can be computed efficiently with the backpropagation algorithm. An excellent in-depth description of the algorithm can be found in the article "How the backpropagation algorithm works" on *neuralnetworksanddeeplearning.com*; the following derivation follows closely the notation and structure of this article.

Backpropagation relies on the judicious application of the chain rule to the equations that define a network. In this derivation a superscript such as b^l denotes a quantity associated with layer l, while a subscript denotes a component of a vector (e.g. b^l_i represents the bias of the i^{th} neuron in layer l). Let the vector a^l be the activation of layer l, and z^l denote the weighted and biased input to a layer in the network, so that

$$z^l_i = \sum_j w^l_{ij} a^{l-1}_j + b^l_i \tag{7}$$

The input and activation are related through the activation function σ, which acts on each component of the input:

$$a^l_i = \sigma(z^l_i) \tag{8}$$

The backpropagation algorithm relies on the concept of "error" in a particular neuron, denoted δ^l_i and defined by $\delta^l_i = \frac{\partial J}{\partial z^l_i}$. If the parameters in a network are near optimal, the gradient of the objective with respect to any parameter should be near zero; heuristically, $\frac{\partial J}{\partial z^l_i}$ is small for any neuron with near optimal parameters, and large where optimization is still needed.

The backpropagation algorithm begins with a forward pass. Here, the network is applied to a set of features x, and the input z^l and activation a^l are recorded at each layer l according to (7). This yields the networks final activation, a^L, which is compared to a set of labels y to calculate the network's objective function $J(x, \theta)$. For simplicity, consider a quadratic objective function

$$J(x, \theta) = \left\| a^L - y \right\|^2 \tag{9}$$

This objective function is convenient because of its simple derivative with respect to any activation:

$$\frac{\partial J}{\partial a_i^L} = (a_i^L - y_i) \tag{10}$$

The error in the final layer, δ^L, is derived from the chain rule:

$$\delta_i^L = \frac{\partial J}{\partial z_i^L} = \frac{\partial J}{\partial a_i^L}\frac{\mathrm{d}a_i^L}{\mathrm{d}z_i^L} \tag{11}$$

Here the total derivative $\frac{\mathrm{d}a_i^L}{\mathrm{d}z_i^L}$ is used, since any neuron's activation function is a single-variable function of its input. Taking derivatives from (8) and (10) gives

$$\delta_i^L = (a_i^L - y_i)\sigma'(z_i^L). \tag{12}$$

This is an element-wise product (Hadamard product) of two vectors, often notated with the "$\odot$" symbol:

$$\delta^L = (a^L - y) \odot \sigma'(z^L) \tag{13}$$

To compute the error in previous layers, error is "backpropagated" from the last layer to the first. Each step of this backward pass begins with δ^l known, and arrives at δ^{l-1} through the chain rule:

$$\delta_i^{l-1} = \frac{\partial J}{\partial z_i^{l-1}} = \sum_j \frac{\partial J}{\partial z_j^l}\frac{\partial z_j^l}{\partial a_i^{l-1}}\frac{\mathrm{d}a_i^{l-1}}{\mathrm{d}z_i^{l-1}} \tag{14}$$

Although the expression is a bit unruly, the only new derivative here is $\frac{\partial z_j^l}{\partial a_i^{l-1}}$. From (7),

$$\frac{\partial z_i^l}{\partial a_j^{l-1}} = w_{ij}^l \tag{15}$$

Substituting this into (14) gives

$$\delta_i^{l-1} = \Big[\sum_j \delta_j^l w_{ji}\Big]\sigma'(z_i^{l-1}) \tag{16}$$

Recognizing the bracketed term as multiplication by $(W^l)^T$, and the remaining multiplication as an element-wise product, the above can be written succinctly in matrix notation as

$$\delta^{l-1} = (W^l)^T\delta^l \odot \sigma'(z^{l-1}) \tag{17}$$

Applying this equation to layer L and continuing backwards to the first layer yields the error for every neuron in the network. Finally, the gradient of the objective function with respect to any parameter is related to the error in a particular neuron. Using the chain rule,

$$\frac{\partial J}{\partial b_i^l} = \frac{\partial J}{\partial z_i^l}\frac{\partial z_i^l}{\partial b_i^l} \tag{18}$$

$$\frac{\partial J}{\partial w_{ij}^l} = \frac{\partial J}{\partial z_i^l}\frac{\partial z_i^l}{\partial w_{ij}^l} \tag{19}$$

Taking derivatives of (7) gives

$$\frac{\partial z_i^l}{\partial b_i^l} = 1 \tag{20}$$

$$\frac{\partial z_i^l}{\partial w_{ij}^l} = a_j^{l-1} \tag{21}$$

After substituting these into (18) and (19), and then vectorizing the result,

$$\frac{\partial J}{\partial b^l} = \delta^l \tag{22}$$

$$\frac{\partial J}{\partial W^l} = \delta^l (a^{l-1})^T \tag{23}$$

The equations of backpropagation are collected in fully-vectorized, algorithmic form below.

Algorithm 1 Backpropagation

procedure BACKPROPAGATION(X, θ)
  $a^0 \leftarrow x$ ▷ Set Input
  for $l \leftarrow [1, L]$ **do** ▷ Forward Pass: Compute weighted inputs and activations
    $z^l \leftarrow W^l a^{l-1} + b^l$
    $a^l \leftarrow \sigma(z^l)$
  end for

  $\delta_L \leftarrow (a^L - y) \odot \sigma'(z^L)$ ▷ Compute Error in Final Layer
  for $l \leftarrow [L-1, 1]$ **do** ▷ Backpropagate Error
    $\delta^l \leftarrow (W^{l+1})^T \delta^{l+1} \odot \sigma'(z^l)$
  end for

  for $l \leftarrow [1, L]$ **do** ▷ Compute gradients w.r.t. network parameters
    $\nabla_{b^l} = \delta^l$
    $\nabla_{W^l} = \delta^l (a^{l-1})^T$
  end for
  return $\nabla_\theta = (\nabla_{W^l}, \nabla_{b^l})$
end procedure

3.3 Network Update

With backpropagation complete, it is time to update the network weights and biases. For each layer of the neural network, the parameters update is as follows:

$$\theta_i \leftarrow \theta_i - \eta \nabla_\theta \tag{24}$$

in which η is the learning rate. This procedure updates network parameters based on a single point from a labeled dataset; if mini-batching is desired, gradients are computed for every data point in the batch, and then averaged before an update step is taken:

$$\theta_i \leftarrow \theta_i - \frac{\eta}{B} \sum_i (\nabla_\theta)_i \tag{25}$$

Here B is the size of the minibatch. The SGD algorithm is now repeated until a stopping criteria is met. This may be some predefined convergence criteria or a limit on the number of updates that can be completed.

4 Adam Optimizer

Adam optimizer is technically not an entirely new algorithm, but rather a modification of stochastic gradient descent. As such, the entire algorithm operates identically to SGD except for a new parameter update step involving two newly introduced variables, m and v. These are known as the first and second moments of inertia, and are at the core of what separates Adam from SGD. As will be familiar to anyone who has taken

Algorithm 2 Stochastic Gradient Descent (SGD)

procedure SGD TRAINING(X)
  initialize $\theta = (W, b)$ to small random numbers ▷ W are NN weights, b are NN biases
  randomize order of training examples in $\mathbf{X}$ ▷ Not applicable in RL
  while not converged **do**
    Create *minibatch* of length B from X ▷ Depends on application
    $a_1 = minibatch_j$ ▷ Input to NN from Batch
    ∇_θ =Backpropagation(X, θ)
    for $i \leftarrow [2, L]$ **do** ▷ Update moments for both W and b
      $\theta_i \leftarrow \theta_i - \frac{\eta}{B} \sum_i (\nabla_\theta)_i$ ▷ Update Network Parameters
    end for
  end while
end procedure

a class in probability and statistics, the first moment of inertia of a distribution refers to its mean, while the second is the uncentered variance. More specifically, m_i is the moving average of the mean of the gradient and v_i is the moving average of the variance. How these improve the performance of the optimizer will soon be clear.

4.1 Network Update

Psuedo-code for the Adam optimizer is given in Algorithm 3. Before updating the network parameters, the moments must be updated and corrected for bias. Recall that the first moment represents a moving average of the gradient, while the second moment is the moving average of the uncentered variance of the gradient. Using parameters β_1 and β_2, these averages are updated. In this report, values of $\beta_1 = 0.9$ and $\beta_2 = 0.999$ were used.

$$m_i \leftarrow \beta_1 m_i + (1 - \beta_1)\nabla_\theta \tag{26}$$

$$v_i \leftarrow \beta_2 v_i + (1 - \beta_2)\nabla_\theta^2 \tag{27}$$

By using these moving averages, Adam builds momentum as it conducts gradient descent. If, after calculating many similar gradients, it comes across a single spurious gradient, the optimizer will mostly disregard it. This smooths the learning process greatly.

However, before using them to update parameters, these moments must be corrected for bias. When learning first begins, Adam initializes m_i and v_i to zero. As a result, the first few updates are all biased towards this value. Therefore, to avoid biasing the update step, it is necessary to conduct updates using the true moments, $\mathbb{E}[\nabla_\theta]$ and $\mathbb{E}[\nabla_\theta^2]$. In order to derive the bias correction equations, first consider the average update equation for v_i, Eq. 27, at some future time step t. At such a point, the value of v_t can be rewritten as a geometric series of discounted gradients.

$$v_t = (1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \cdot \nabla_\theta^2 \tag{28}$$

To find our desired expected value, we then take the expectation of both sides of the equation.

$$\mathbb{E}[v_t] = \mathbb{E}[(1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \cdot \nabla_\theta^2] \tag{29}$$

Since the summation portion of this previous equation is a constant, it can be pulled out of the expectation. This leads the right hand side with a constant and the desired expected value.

$$\mathbb{E}[v_t] = \mathbb{E}[\nabla_\theta^2] \cdot (1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \tag{30}$$

A simpification is then applied

$$\mathbb{E}[v_t] = \mathbb{E}[\nabla_\theta^2] \cdot (1 - \beta_2^t) \tag{31}$$

and the terms are rearranged, thus providing the desired bias correction update equation.

$$\mathbb{E}[\nabla_\theta^2] = \frac{\mathbb{E}[v_t]}{(1 - \beta_2^t)} \tag{32}$$

The derivation for the first moment of inertia is similar. The final form of the unbiased estimators and parameter update equations are shown below.

$$\hat{m}_i = \frac{m_i}{1 - \beta_1^t} \tag{33}$$

$$\hat{v}_i = \frac{v_i}{1 - \beta_2^t} \tag{34}$$

$$\theta_i \leftarrow \theta_i - \frac{\alpha}{\sqrt{\hat{v}_i} + \epsilon} \hat{m}_i \tag{35}$$

Algorithm 3 Adam

procedure ADAM(X)
  initialize $\theta = (W, b)$ to small random numbers ▷ W are NN weights, b are NN biases
  randomize order of training examples in $\mathbf{X}$ ▷ Not applicable in RL
  initialize $m \leftarrow 0$, $v \leftarrow 0$ ▷ 1st and 2nd moments of inertia
  while not converged **do**
    Create *minibatch* of length B from X ▷ Depends on application
    $a_1 = minibatch_j$ ▷ Input to NN from Batch
    ∇_θ =Backpropagation(X, θ)
    for $i \leftarrow [2, L]$ **do** ▷ Update moments for both W and b
      $m_i \leftarrow \beta_1 m_i + (1 - \beta_1)\nabla_\theta$ ▷ Update biased first moment
      $v_i \leftarrow \beta_2 v_i + (1 - \beta_2)\nabla_\theta^2$ ▷ Update biased second moment
      $\hat{m}_i = \frac{m_i}{1-\beta_1^t}$ ▷ Bias-corrected first moment
      $\hat{v}_i = \frac{v_i}{1-\beta_2^t}$ ▷ Bias-corrected second moment
      $\theta_i \leftarrow \theta_i - \frac{\alpha}{\sqrt{\hat{v}_i}+\epsilon}\hat{m}_i$ ▷ Update Network Parameters
    end for
  end while
end procedure

5 Classification Results

To test the performance of the SGD and ADAM algorithms, both were implemented in MATLAB. Theses algorithms were tested on classification datasets inspired by the Google TensorFlow Playground application (*playground.tensorflow.org*), shown in Figure 2. Each datapoint provides the coordinates of a point in 2-dimensional space, as well as a number representing its color (1 for orange, 0 for blue). Clusters of blue or orange points are clustered to form recognizable shapes such as rings, circles, squares, and spirals. Given the coordinates of an arbitrary point, a neural network is tasked with predicting its color.

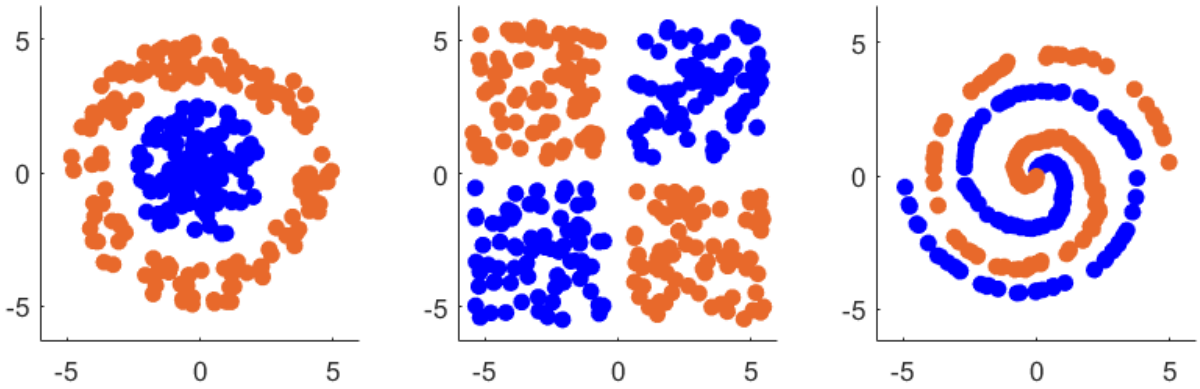
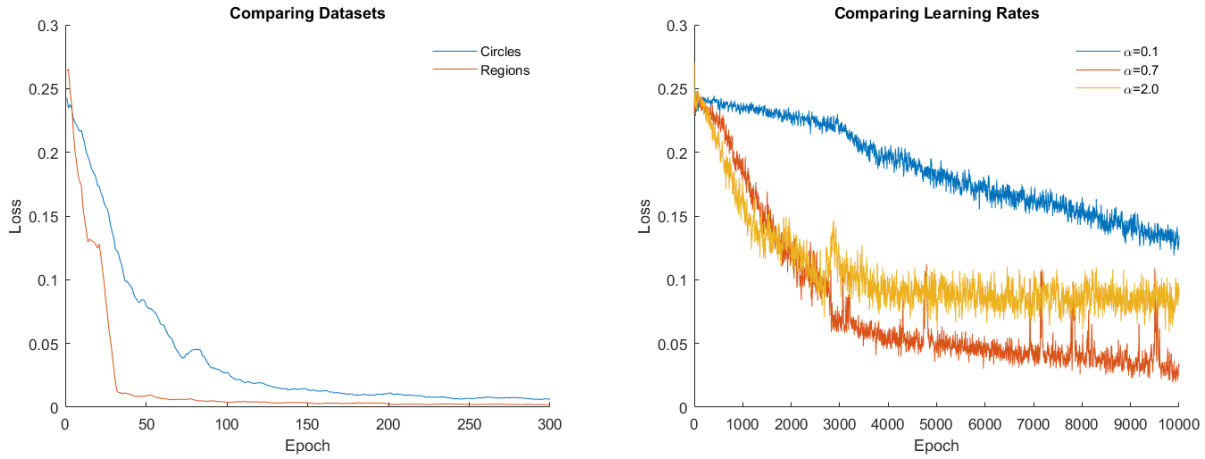


Figure 2: Imitations of Google TensorFlow Playground Datasets: circles, square regions, and spirals

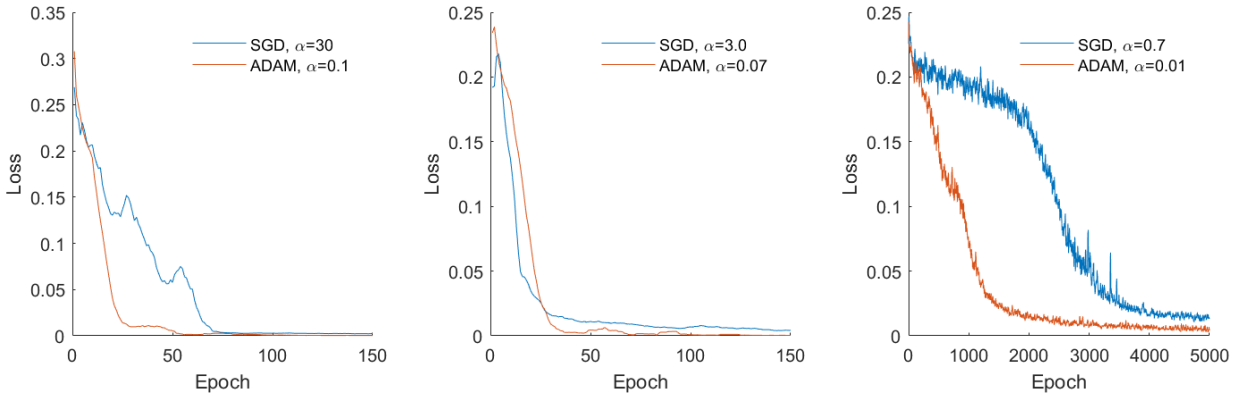
To test the efficacy of SGD, the algorithm was used to train a simple, fully-connected neural network on these three datasets. The network consisted of two inputs (x and y coordinates), two hidden layers of eight neurons each, both using a ReLU activation function, and a single output neuron using a sigmoid activation function. This produces a number between 0 and 1 to represent the color of any point in the plane. During the training, each parameter update was based on a batch of 50 datapoints.



(a) (b)

Figure 3: (a) Training the circle and regions datasets with SGD. (b) Training the spiral dataset with SGD using different learning rates

As seen in Figure 3a, SGD is able to successfully train the network on the simplest two datasets, circles and square regions. These tests used a learning rate of $\alpha = 3$, and arrived at satisfactory solutions within 150 iterations. For the more complex spiral dataset, learning rate has a significant impact on the results. Figure 3 shows the performance of SGD on the spiral dataset at three different learning rates. For $\alpha = 0.1$, the network learns slowly, and is unable to converge to a steady loss within $10,000$ updates. At higher learning rates, the algorithm does converge; $\alpha = 0.7$ reaches a stable average loss of 0.05 per within $5,000$ iterations. However, as shown in Figure 4c, the global minimum loss for a network of this size is several orders of magnitude lower. Even at this stage, the learning rate is too high for the algorithm to settle at this minimum, and instead it skips around near this point. If the learning rate continues to increase to $\alpha = 2.0$, the network reaches it's steady-state faster, but ultimately strays farther from the minimum.



(a) Circles (b) Regions (c) Spirals

Figure 4: Comparing SGD and ADAM optimizers on all three PlayGround datasets

With the ADAM optimizer, the network trains faster on all three datasets. Figure 4 shows typical loss vs. epoch plots for both optimizers, demonstrating that ADAM reaches a satisfactory solution faster and ultimately achieves a lower loss than SGD. For these plots, the base learning rate of each optimizer has been chosen by trial and error to achieve a balance of speed and ultimate accuracy during the optimization. While gains are marginal for the simple dataset, ADAM is able to consistently train the spirals dataset to a high degree of accuracy within several thousand epochs, while SGD takes many more epochs to achieve sub-optimal accuracy.

6 Reinforcement Learning

Consider the following scenario. A machine learning researcher has a robot and a maze in which it needs to navigate. The maze is full of obstacles and the task is to get to the middle of the maze without colliding with any obstacles. At any given moment the robot knows its state, but nothing of its surroundings. It also has the ability to take a single step in any direction per discrete time step. How does the researcher train the robot to successfully complete the course?

One solution is to provide feedback to the robot based upon its actions. Every time the robot takes a step closer to the goal state, it receives a positive reward of +1. Every time it moves away from the goal, it receives a negative reward of -1. Finally, if the robot collides with an obstacle, it receives a larger negative reward of -5 and is placed back to the point in the maze in which it started. Over time, the robot begins to associate certain actions in certain states with positive or negative rewards. If provided with a sufficiently large number of attempts, the robot will learn to navigate to the center of the maze. This strategy of providing a reward for an action in order to train a robot, otherwise known as an agent, is the core of a subcategory of machine learning called Reinforcement Learning (RL).

With this basic concept introduced, the question of how to make an agent understand and learn from its actions quickly arises. Since this report is focused on SGD and Adam, this report shall limit its discussion of RL to a popular, well-proven RL algorithm called Deep Q-Learning (DQN).

Consider an agent acting in a discrete time system. At time point $t \in \{0, 1, ...T\}$, the agent is in state s_t. The agents selects an action a_t from a list of available actions, which takes the agent to state s_{t+1} at time $t + 1$. Upon entering this state, the agent receives a reward R_{t+1}, and the state-action-reward cycle repeats. The sequence ends when the agent enters a "terminal state", usually corresponding to either success or failure. Typically the agent chooses its action based on some deterministic rule at each state. This is a "policy", and the goal of reinforcement learning is to find a policy that maximizes a system's total reward.

The notion of total reward can be ill-defined for systems with no terminal states, so it is helpful to introduce the idea of a discounted future reward. The quantity we seek to maximize is the expected value

of future discounted rewards,

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ..., \tag{36}$$

where γ is a discount factor ($0 < \gamma < 1$). Higher values of γ lead to a farsighted strategy, while lower values of γ prioritize immediate reward over long-term gain. Given a particular policy, define the "quality" of an action to be the expected value of future rewards after taking that action from a given state:

$$Q(a, s) = \mathbb{E}(G_t \mid s_t = s, a_t = a) \tag{37}$$

A this point it is helpful to introduce, without proof, two results from the theory of Markov Decision Processes. First, an optimal policy is one which chooses the highest quality action in any given state. Second, when following an optimal policy, the quality function obeys the Bellman equation:

$$Q(a_t, s_t) = R_{t+1} + \gamma \max_a Q(a_{t+1}, s_{t+1}) \tag{38}$$

Intuitively, the Bellman equation states that the expected future reward of an action comes in two parts: the immediate reward, and expected future reward from the next state's best possible action. Along with the result about optimal policies, this result is enough to construct a successful neural-network based reinforcement learning algorithm.

6.1 Deep-Q Networks

The central idea of DQN is to approximate the quality function Q with a neural network. This network takes the current state as an input, and calculates the quality of each available action, assuming that the agent follows an optimal policy. In turn, the agent chooses the action with the highest possible quality at each time step. If the network accurately approximates Q, then its output will obey the Bellman equation. This can be framed as a cost function, which is minimized only when the Bellman equation is satisfied:

$$C = \left[R_{t+1} + \gamma \max_a Q(a_{t+1}, s_{t+1}) - Q(a_t, s_t)\right]^2 \tag{39}$$

This cost function is used in conjunction with backpropagation and gradient-based optimization to train the neural network. Theoretically, as this cost function decreases, the Q network comes closer to approximating a solution of the Bellman equation, and the agent's choices approach an optimal policy. This basic DQN algorithm is given below.

```
1: procedure NAIVE DQN(X)
2:     Initialize Neural Network to represent action-value function Q_θ
3:     for ep = [1,N] do                          ▷ N is max number of episodes
4:         s_t ← s_0                              ▷ Random Initial State
5:         while s_t is not a terminal state do
6:             Select a_t = argmax_a Q(s_t, a)
7:             Receive reward R_{t+1} and new state s_{t+1}
8:             Train network using Bellman cost function
9:             s_t ← s_{t+1}
10:        end while
11:    end for
12: end procedure
```

6.2 Improving on DQN

In practice, a naive implementation of deep-Q learning is highly unstable. However, with some minor alterations, the algorithm is capable of solving complex tasks.

- **ϵ-greedy policy.** Exploration vs. exploitation is a classic problem in RL: exploring new solutions to problems can lead to important gains, but this strategy usually does not maximize rewards in the short term. A simple way to encourage exploration is to force an agent to act randomly at some timesteps. An ϵ-greedy policy is one in which, at each timestep, the agent chooses a random action with probability ϵ, and otherwise chooses the highest quality action. This parameter ϵ typically starts close to 1, and decreases as training progresses.

- **Experience Replay.** When a network trains only on the results of its most recent action, it can "forget" solutions that were learned in the past. One way to preserve this information is to store the agent's past actions in memory, and sample randomly from this memory when training the network. This strategy is called Experience Replay, and helps stabilize the performance of Deep-Q networks.

- **Target Network.** In traditional learning, changing the parameters of a network moves its output close to a desired output label). In DQN, quality is the network output, and the right hand side of the Bellman equation (38) acts as a label. However, changing the parameters now changes both the output and the label, which causes the training process to be extremely unstable. To remedy this, these labels can be generated by a static "target network" that is updated to match the trained network every few thousand time steps. This approach requires double the memory for network parameters, but greatly increases the stability of the training process.

These improvements to DQN are given below as a revised version of the original algorithm.

---

procedure DQN(X)
  Initialize Neural Network to represent action-value function Q_θ
  Initialize Experience Replay memory, M
  Initialize target action-value function $\hat{Q}_{\theta^-}$
  for ep = [1,N] **do** ▷ N is max number of episodes
    $s_t \leftarrow s_0$ ▷ Random Initial State
    while s_t is not a terminal state **do**
      With probability ϵ select a random action a_t
      Otherwise, select $a_t = argmax_a Q(s_t, a)$
      Receive reward R_{t+1} and new state s_{t+1}
      Store transition $\{s_t, a_t, s_{t+1}, R_{t+1}\}$ in M ▷ Experience Replay
      Train Q_θ using random experiences from M
      $s_t \leftarrow s_{t+1}$
    end while
    $\hat{Q}_{\theta^-} \leftarrow Q_\theta$ ▷ Update Target Network
  end for
end procedure

---

7 RL Results

7.1 Cart Pole

A classic test problem in reinforcement learning is "Cart Pole", illustrated in Figure 5. In this environment, an agent is presented with a dynamic system with two degrees of freedom: a cart that slides back and forth on a track, and a pole that pivots about the center of the cart. Both the cart and the pole have mass, and the dynamic system is integrated numerically using a symplectic Euler scheme. Typical parameters for Cart Pole are given in Table 1.

A single state of the system consists of the position and velocity of both the cart and the pole: $s = \{x, \dot{x}, \theta, \dot{\theta}\}$. At each timestep in the numerical integration, the agent can choose from two actions: apply a force of 10N or -10N to the cart. The system reaches a terminal state when either x or θ exceeds some

$x_{max} = 2.4$	$m_{cart} = 1.0kg$	$\|F\| = 10N$
$\theta_{max} = 12°$	$m_{pole} = 0.1kg$	$L = 0.5m$

Table 1: Parameters used in the CartPole environment

predefined criteria (x_{max}, θ_{max}). The agent receives a reward of 1 at every timestep that the cart is upright, and 0 for terminal states. To maximize reward, an agent must find a way to keep the pole balanced upright, without leaving the screen.

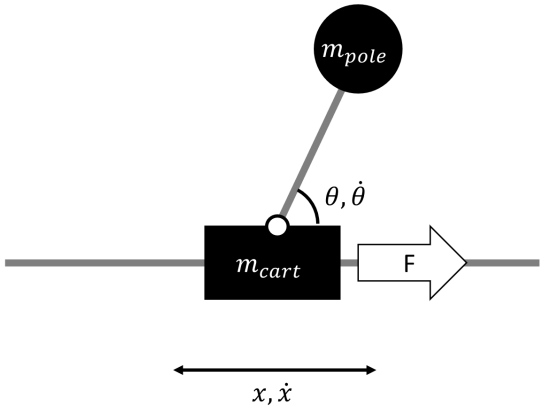


Figure 5: CartPole: a classic reinforcement learning problem

To solve this problem, a Deep-Q network was setup with two hidden layers of 16 neurons each. The hyper-parameters used for the learning are given in Table 2. The Cart-Pole environment was implemented with the same parameters given in Table 1. Figure 6 shows a typical graph of performance vs. time for the Deep-Q network, illustrating that the networks performance does in fact improve as training progresses. Unfortunately the algorithm is still somewhat unstable, despite all of the alterations proposed in the previous section. However, over the course of training the network moves from an average run of 15 time steps to over 100 time steps, and several times reaches a peak performance of over 300 time steps.

If the network is stopped at peak performance, the resulting policy is able to reproduce this peak performance consistently, showing that the instability is due to training updates and not to an a relatively stable network learning an unreliable policy. On the upside, if a trigger is set up to stop the network at a certain performance network, the relatively small network is able to generate policies that can consistently balance a pole for upwards of 2000 time steps.

$\alpha = 0.003$	$\epsilon = 0.05$	$\gamma = 0.95$
Memory: 2000 transitions	Batchsize: 128 transitions	Target net updates: 3000 steps

Table 2: Hyper-parameters used in the Deep Q network

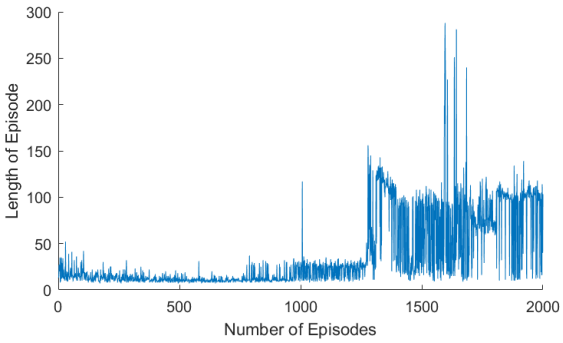


Figure 6: Performance vs. time for a typical Cart Pole training session

8 Conclusion

The purpose of this report was to remove the aura of mystery from neural network learning and to provide a greater understanding of this process that many consider to be a black box. To this end, Backpropagation, Stochastic Gradient Descent, and Adam optimizer were discussed in detail, and the two optimizers were compared on a sample problem. The basic frameworks of Reinforcement Learning and Deep-Q Learning were also presented, to illustrate a fascinating application of gradient-based optimization. Machine learning is a fast-moving field, and it is likely that these algorithms will be (or already have been) altered or replaced in cutting-edge research. However, the basic concepts presented here should provide a thorough background for exploring these new technologies.

9 References

[1] Ruder, Sebastian. "An Overview of Gradient Descent Optimization Algorithms." Sebastian Ruder, Sebastian Ruder, 19 Jan. 2016, ruder.io/optimizing-gradient-descent.

[2] Kingma, Diederik P, and Jimmy Ba. "ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION." *CoRR*, abs/1412.6980, 2014, arxiv.org/abs/1412.6980.

[3] Nielson, Michael A. *Neural Networks and Deep Learning*. Determination Press, 2015, neuralnetworksanddeeplearning.com/chap2.html.

[4] Mnih, Volodymyr, et al. "Human-Level Control through Deep Reinforcement Learning." Nature, vol. 518, 26 Feb. 2015, pp. 529–533., doi:https://doi.org/10.1038/nature14236.

[5] Smilkov, Daniel, and Shan Carter. "Tensorflow - Neural Network Playground." *A Neural Network Playground*, playground.tensorflow.org/.