Evaluation of Optimization Algorithms for Machine Learning Applications

Pyokyeong Son

Abstract

Recent advances in the field of machine learning has profoundly impacted our use of technology. These advances are driven by the discovery of the back-propagation algorithm used to train various types of neural networks—the key to its success, determined by the mathematical optimization algorithm used for minimizing its cost function. In this paper, I discuss the design of neural networks and the methods used for its training, and evaluate the different optimization algorithms used to train a neural network.

Table of Contents

I. Background Knowledge
    1. A Brief History of Machine Learning
    2. Usage of Neural Networks
II. Introduction to Core Concepts
    1. Artificial Neurons
    2. Neural Networks
    3. Usage of Neural Networks
III. Optimization Algorithms for Neural Networks
    1. Gradient Descent and Stochastic Gradient Descent (SGD)
    2. Momentum-Assisted Gradient Descent
    3. Accelerated Gradient Descent
    4. A Need for Evaluation
IV. Evaluation of Optimization Functions for Training Neural Networks
    1. The Sphere Function
    2. The Rastrigin Function
    3. The Styblinsky Tang Function
    4. The Cost function of the Neural Network
V. Conclusion
VI. Bibliography
VII. Appendix

This paper must be viewed with color.

I. Background Knowledge

1. A Brief History of Machine Learning

The motivation to create intelligent beings like ourselves has led to numerous endeavors by computer scientists and mathematicians since the idea was initially conceived. In 1950, Alan Turing, a mathematics professor at Cambridge University—often regarded the Father of computer science—theorized the "Turing Test," a set of conditions that would determine if a computing entity matches human-level intelligence, kickstarting the development of artificial intelligence. The following is a brief history of major discoveries of concepts in A.I. that will be of interest for our purposes:

1950's - Invention of the Perceptron and Neural Networks

1960's - Invention and development of the Back-propagation Algorithm for Neural Networks

1974–1980 - Failure of Machine Language Translation, and cut in government funding, leading to the First A.I. Winter

1987–1993 - Collapse of LISP, a popular language used in the A.I. development, and failure of Expert Systems, leading to the Second A.I. Winter

2000's - Computing power reaches practical levels for use in machine learning

2010's - Successful use of Neural Networks with convolution or reinforcement learning techniques, through research in institutions and the private sector.

2. Usage of Neural Networks

Neural networks are mathematical models that exhibit traits of specific types of intelligence. Combined with designs including recursion or convolution, they can be applied to a wide set of problems including image classification, speech recognition, natural language processing, and more.

Most types of neural networks, however, need to be "trained" on pre-labeled data; e.g., an image classification network would need to be trained to classify specific types of images into categories, on a set of pictures that are already categorized. In this paper, we will focus on the algorithms used to train certain types of neural networks—a topic closely related to the field of mathematical optimization.

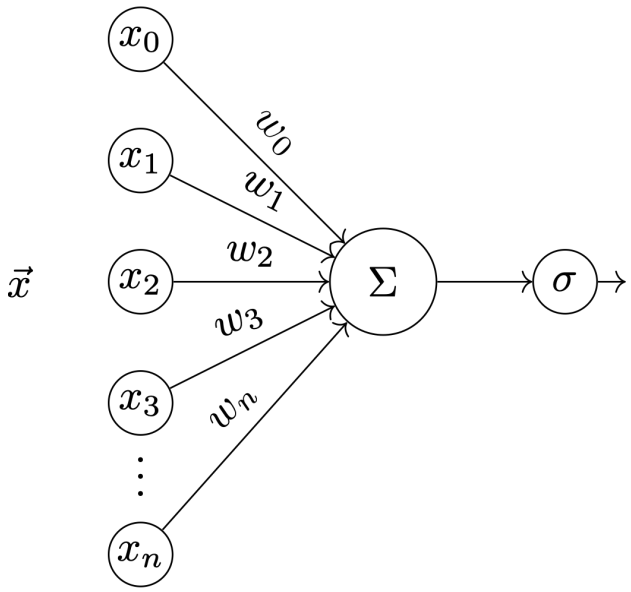
II. Introduction to Core Concepts

1. Artificial Neurons

An **Artificial Neuron** is a mathematical function that takes an input of a column vector of n-dimensions and outputs a single number, with parameters of weight and bias.

Input: $\overrightarrow{x_i} = \{x_1, x_2, \ldots, x_n\} \in R^n$
Output: $y \in R$
Parameters: $\overrightarrow{w_i} = \{w_1, w_2, \ldots, w_n\}, b$

[Diagram 2.1.1: An Artificial Neuron]



Each row of the input vector x_i is multiplied by its respective weight w_i, and summed. The bias is added, and the resultant number is passed through a sigmoid function: $\sigma = \frac{1}{1+e^{-x}}$ [1] as we iterate through multiple neurons. The resulting value is known as the neuron's *activation*.

Mathematically, the function of the neuron can be described as:

$$y = \sigma(\sum_{i=1}^{n} [w_i x_i] + b)$$

We can use dot multiplication to simplify our notation:

$$y = \sigma(\overrightarrow{w} \cdot \overrightarrow{x} + b)$$

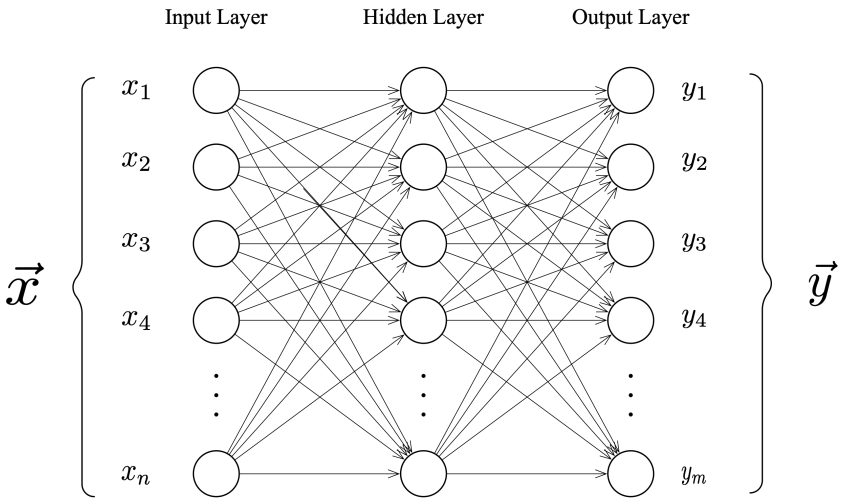
[1] A Sigmoid Function is used as the *Activation Function* in this Neural network, in order to remove fluctuations and keep activation values within a range.

2. Neural Networks

A **Neural Network** (NN) is a mathematical function that is composed of artificial neurons. It takes an input of a column vector of n-dimensions and outputs another column vector of m-dimensions, and is composed of one input layer, many hidden layers, and one output layer.

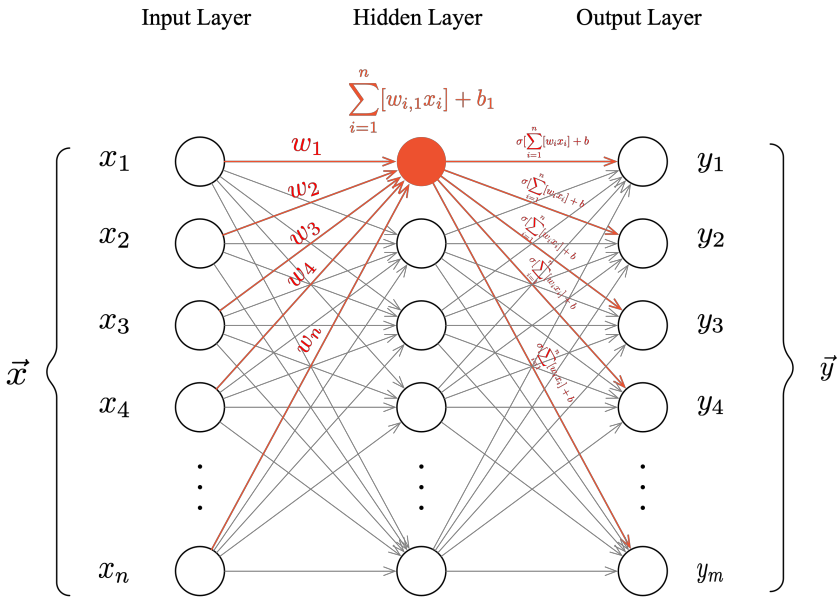
Input: $\vec{x} = \{x_1, x_2, \ldots, x_n\} \in R^n$
Output: $\vec{y} = \{x_1, x_2, \ldots, x_m\} \in R^m$
Parameters: *Shown Later*

[Diagram 2.1.2: A Neural Network with 3



Each number in our input vector, $\vec{x}$, is fed into each respective neuron in the input layer. The output of each layer is passed onto the next layer.

[Diagram 2.1.3: Role of a Single Neuron in a Neural Network]



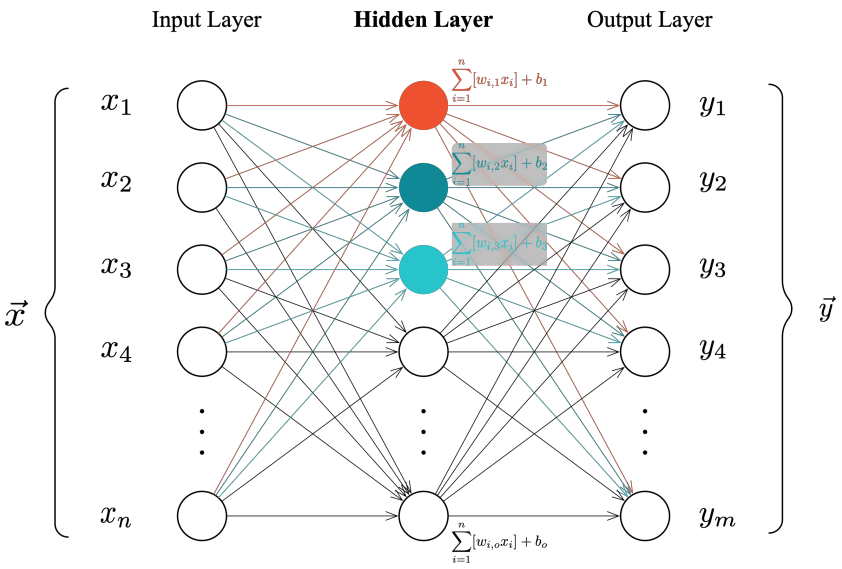
A single *layer* in a Neural Network is a column of neurons. The *input* to each neuron is the output column vector of all neurons from the previous layer, and the *output* is a single number. This number is further passed on as inputs to neurons in the next layer.

Each artificial neuron, as described previously, has the parameters: $\overrightarrow{w_i}$, and b, where i would be determined by the number of neurons in the previous layer. For the hidden layer in our diagram, $\overrightarrow{w}$ would have n rows.

As we have o number of neurons in our hidden layer, if we take the current layer as a whole, it has the parameters $\{\{w_{1,1}, w_{2,1}, \ldots w_{n,1}\}, \{w_{1,2}, w_{2,2}, \ldots w_{n,2}\}, \ldots \{w_{1,o}, w_{2,o}, \ldots w_{n,o}\}, \}$, and $\{b_1, b_2, \ldots, b_o\}$. We can write the former as a single matrix $W_{i,j}$ with shape $o \times n$, and the latter as a column vector B_j with shape $o \times 1$:

$$W = \begin{bmatrix} w_{1,1} & \cdots & w_{n,1} \\ w_{1,2} & \cdots & w_{n,2} \\ \vdots & & \vdots \\ w_{1,o} & \cdots & w_{n,o} \end{bmatrix}, B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_o \end{bmatrix}$$

[Diagram 2.1.4: Role of a Single Layer in a Neural Network]



Therefore, the operation of all neurons in a single layer is equivalent to the following process:

1. Take the input of the previous layer x_i and get the dot product with it and the weight W:
   $W \cdot \overrightarrow{x}$
2. Get the Vector Sum of $W \cdot x$ and B:
   $W \cdot x + B$
3. Pass it as the input of the next layer:
   $x' = W \cdot x + B$
   (If the next layer is the output layer, x' is simply the output of the whole neural network, y.)

If, for example, there are 3 hidden layers in a neural network, each layer will have its own parameters W and B. We use superscripts to identify which layer each *weight matrix* and *bias vector* belongs to: W^1, W^2, W^3, and B^1, B^2, B^3.

Therefore, our neural network is a function that takes in the input of a column vector x with shape $o \times 1$, Calculates:

$$x' = W^1 \cdot x + B^1,$$
$$x'' = W^2 \cdot x' + B^2,$$
$$y = W^3 \cdot x'' + B^3,$$

and outputs a column vector y with shape $n \times 1$. In a very simplified form, a neural network with 2 hidden layers is:

$$NN(x) = W^3 \cdot (W^2 \cdot (W^1 \cdot x + B^1) + B^2) + B^3$$

As we go along, we will use a computer program to implement a neural network, and compare its characteristics with our mathematical model. The following neural networks will modeled using the Python programming language and the NumPy library.

3. Usage of Neural Networks

Despite being interesting mathematical models, neural networks did not serve a practical purpose until the discovery of its usefulness in solving particular real-world problems—problems that could not be solved by traditional algorithms. A good and commonly used introductory example is the problem of recognizing of hand-written digits.

*[Diagram 2.2.1:
The Number "9", written by hand on a 28 by 28 grid of black and white pixels]*

Diagram 2.2.1 is a black-and-white image of the digit *9* on grid of 28x28 pixels, written by a human. We can immediately recognize it as the number "9," but it is very difficult for a traditional algorithm, using, for example, conditionals (*if-else*) and loops (*repeat*), to identify. Multiple attempts of using these programming techniques have failed to reliably identify the number from an image of hand-written digits.[2]

Neural Networks, however, possibly because they were modeled with neurons in the human brain, are capable of solving these problems. The details of *why* they can, however, will not be the focus of this paper[3]. Instead, we will investigate *how* we can use it to solve the problem.

---

[2] This technique of simply using known criteria to make rules is known as *Knowledge Engineering.*

[3] The reason for the effectiveness of neural networks are studied in neuroscience as well as computing, and its implications contemplated in philosophy.

We take the brightness of each pixel in the image and map it to a number. This will produce 784 numbers, which we will take as a column vector and use it as an input for a neural network.

$$X_1 = \begin{bmatrix} x_1 = 0.0 \\ x_2 = 0.1 \\ \vdots \\ x_{784} = 0.0 \end{bmatrix}$$

Using the computer code implementation, we can see what the actual data for the image above looks like:

```
array([
       [0.        ],
       [0.        ],
             …
       [0.        ],
       [0.01171875],
       [0.0703125 ],
       [0.0703125 ],
       [0.0703125 ],
       [0.4921875 ],
       [0.53125   ],
       [0.68359375],
       [0.1015625 ],
       [0.6484375 ],
       [0.99609375],
       [0.96484375],
       [0.49609375],
       [0.        ],
       [0.        ]
             …
])
```

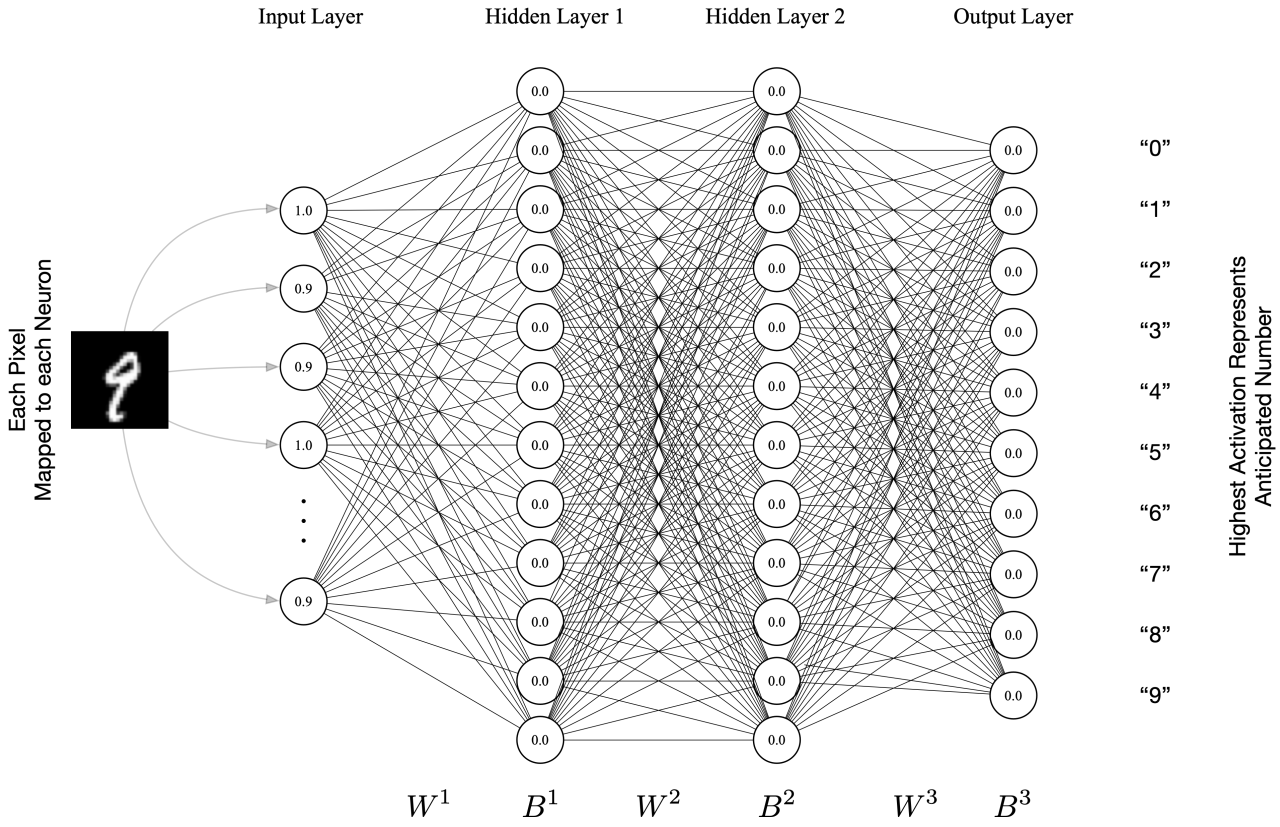
As we can see, the computer does not see an image; it only sees a long array (a vector) of numbers. We have reduced the black and white image into a series of numbers, which will each be fed into the input neuron of the network.

The number of hidden layers in the neural network, and the number of neurons in it, will be arbitrarily, but reasonably chosen. The number of neurons in the input layer, however, must be 784, as it must take the image, or the brightness of each pixel, as its input. The number of neurons in the output layer is also fixed to 10, as the digit written in the diagram must be a digit from zero to nine. We will say that the digit with the highest output value will be the digit the network has determined to the the digit written in the image.

This neural network is arbitrary chosen to have 2 hidden layers, each with 12 neurons. Therefore, the parameters of the whole network will be are:

[Table 2.2.3: Parameters for Neural Network for Digit Recognition]

Parameter Name	Type	Operation	Input	Output
W^1	Matrix, 12×784	$X^1 = W^1 \cdot X + B^1$	X Vector, 784×1	X^1 Vector, 12×1
B^1	Vector, 12×1			
W^2	Matrix, 12×12	$X^2 = W^2 \cdot X^1 + B^2$	X^1 Vector, 12×1	X^2 Vector, 12×1
B^2	Vector, 12×1			
W^3	Matrix, 10×12	$Y = W^3 \cdot X^2 + B^3$	X^2 Vector, 12×1	Y Vector, 10×1
B^3	Vector, 10×1			



[Diagram 2.2.2: A Neural Network for Hand-written Digit Recognition]

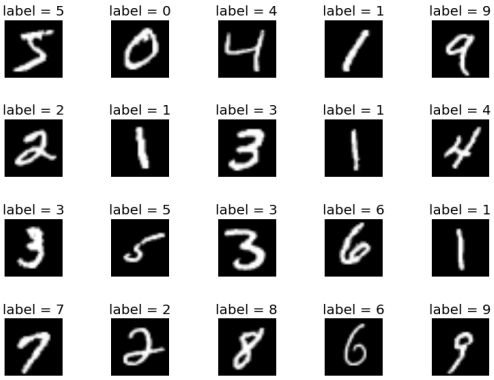
For the sake of simplicity, let us write all weights, W^1, W^2, W^3 as W, and all biases, B^1, B^2, B^3, as B. W and B may not signify a mathematical entity, but will be an aid for conceptual understanding.

In our program, we will implement a network with the above mentioned number of neurons and layers:

```
net = Network([784, 12, 12, 10])
```

3. The Cost Function

We will now attempt to "train" this NN in order to make it determine a digit from 784 numbers. To do this, we will prepare a large number of data (a *dataset*) of, in this case, images of hand-written digits, as well as what number they represent. We will use the MNIST handwritten Digit Database as our training dataset.

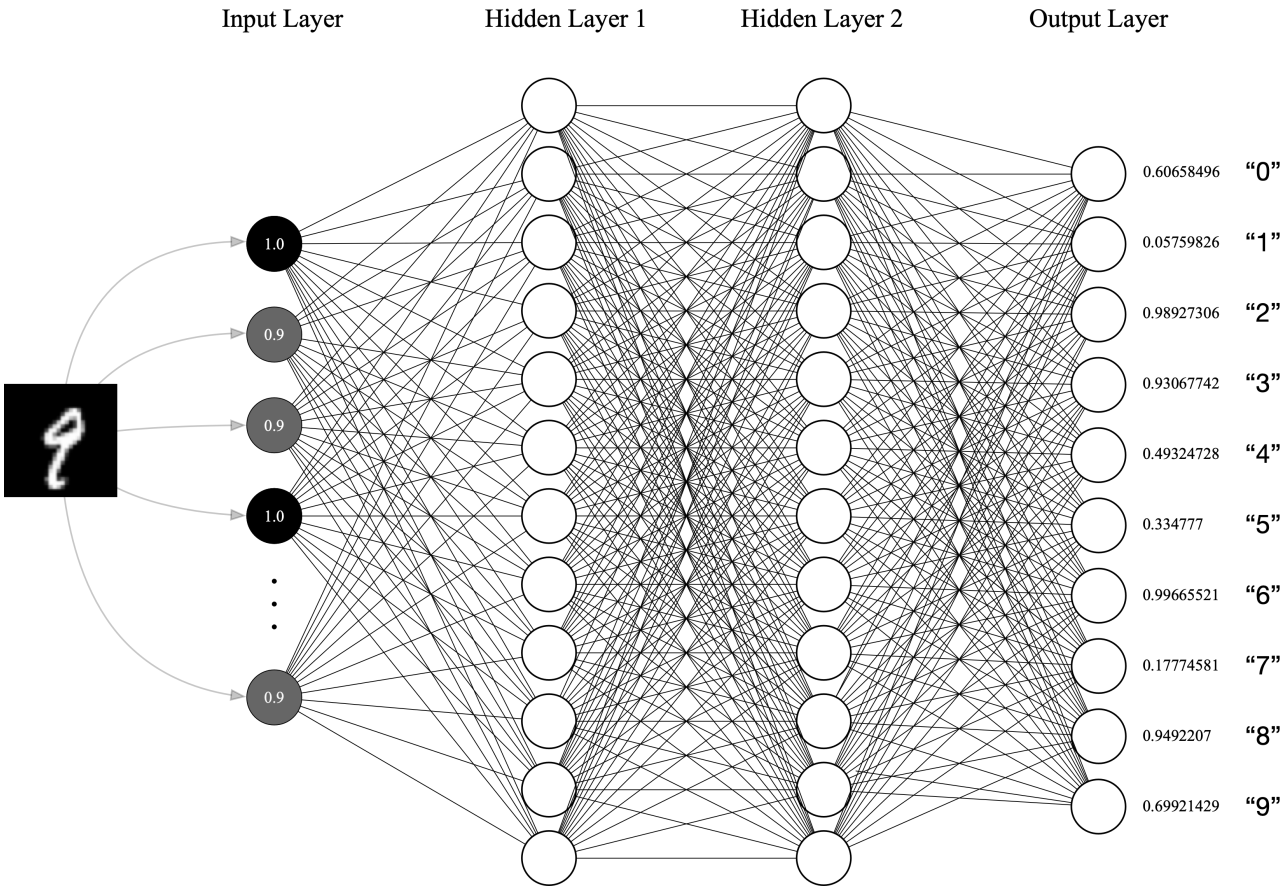


[Diagram 2.3.1: MNIST Handwritten Digit Dataset]

We can make a very bad first attempt, by initializing all the parameters of the network to random values. We take one data point, for example, [], and use it as an input to our neural network.

In our python code, we initialize all the weights and biases to random values. Each element of the array represents the output of each neuron in the output layer. This is illustrated in Diagram 2.3.2.

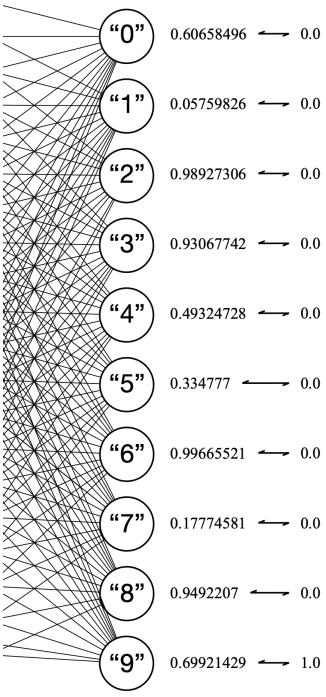
```
print net.test()

[[0.60658496]
 [0.05759826]
 [0.98927306]
 [0.93067742]
 [0.49324728]
 [0.334777  ]
 [0.99665521]
 [0.17774581]
 [0.9492207 ]
 [0.69921429]]
```



[Diagram 2.3.2: A Neural Network with an output]

We can see that our network has performed terribly, and the output values seem to be completely random, quite expectedly, as we initialized the weights and biases to be random. What we can do is evaluate, numerically, how bad the neural network did. We can compare the current output to the ideal value, which would be an output of 1 in the neuron corresponding to "9" and 0 in all other neurons.

[Diagram 2.3.3 (Right Margins): Comparison of the output of a Neural Network with an ideal value]



The mean squared difference between these two vectors, we define as the "cost."

$$\textbf{Cost, for } X_2 = \frac{1}{10}\sum_{i=1}^{10}[(Y_{\textbf{output}} - Y_{\textbf{ideal}})^2]$$

Further on, we will denote the vector of input data for data point i as X_i , and the ideal values for that data point as $X_{i,\textbf{ideal}}$. Therefore:

$$\textbf{Cost, for } X_2 = \frac{1}{10}\sum_{i=1}^{10}(NN(X_2) - X_{2,\textbf{ideal}})^2$$

Depending on the parameters of the neural network, the cost will increase or decrease. Our objective is now encoded, from a vague concept: "Identify the digit in this image," to a specific, mathematical goal: "Minimize the cost of this neural network for these data."—what we call, *training* or *learning*.

Minimizing for a certain variable, as it turns out, has already been studied, in the field of mathematical optimization. We can use the concepts from this field in order the minimize the cost of neural networks.

We define the cost function, J[4], which calculates the *average cost* of all of the data that we currently have, X. We call this function the "cost function" of the neural network. Expressed mathematically, if we were to suppose we had n_{data} number of data points, the cost function would be:

$$J(W,B) = \frac{1}{n_{data}}\sum_{j}^{n_{data}}\frac{1}{10}\sum^{10}(NN(X_j) - X_{j,\textbf{ideal}})^2$$

The value of this cost function will differ depending on the weights and biases, the parameters of the neural network. Our goal is to find the specific combination of weights and biases for our neural network function, $NN()$, that will minimize the cost function, $J(W, B)$—minimizing the differences between our ideal values from the training data.

It would be ideal to calculate the global minimum of the cost function, but this is impractical, as this function will have the number of variables equal to the number of individual weights and biases in the neural network. In our case, with a neural network with number of neurons as $\{784,12,12,10\}$, we will have:

$784 \times 12 + 12$ weights and biases for hidden layer 1

$+12 \times 12 + 12$ weights and biases for hidden layer 2

$+12 \times 10 + 10$ weights and biases for the output layer

$= 9706$ variables; it would be impractical to attempt differentiate this 9706-dimentional function to find its global minima, even assuming that this function is differentiable at all steps of the process[5].

Alternatively, we will use a method known as gradient descent, where we start at a random point in this 9706-dimentioal space (random weights and biases), and calculate the *gradient* or the *slope* of the function at that single point, to find out in which direction we will have to change the weights and biases in order to lower the value of $J(W, B)$.

In the following section, we will evaluate the different heuristics, and the algorithms derived from them, that are used in this process of minimizing this cost function.

---

[4] The "J" is short for "Jacobian Matrix", the matrix of first-order derivatives of a vector-valued function, used in back-propagation *(not covered in this paper)*.

[5] Again, the differentiability of these functions are dealt with in back-propagation.

III. Optimization Algorithms for Neural Networks

We will now explore practical algorithms used in real-life scenarios for optimizing the cost function, or, equivalently, training the neural network. As traditional optimization functions are computationally costly in such multi-variable high-dimensional space, various methodologies have been suggested for reducing its difficulty—mainly, through the investigation of its gradient.

Reducing the cost function $J(W, B)$, or with a simplified notation: $\theta = [W, B]$ and $J(\theta)$, requires a calculation of its gradient $\nabla J(\theta)$, at parameter θ. Then, a "step," is taken in the direction of its gradient, with the "step size" adjusted by multiplying the gradient with the learning rate (η). The new parameters after the update can be represented as:

$$\theta_{new} = \theta - \eta \nabla J(\theta).$$

We can calculate the numerical gradient of a function of arbitrary dimensions *whose explicit form we know*, $f(x)$, where x is a vector, by simply calculating:

$$f'(x) = \frac{f(x_i + \epsilon) - f(x_i)}{\epsilon}$$

However, calculating the gradient of the cost function of a neural network, *whose form is unclear*, requires a special algorithm named *back-propagation*. This, however, is beyond the scope of this paper, and for our purposes, we can treat it as a black box, that if we give it a training data and the current parameters, it will reliably output a gradient at that point.

Reducing the cost function, or simply, *optimizing* or *minimizing* the cost function, using this knowledge of gradients, can be done through several algorithms, each with varying heuristics to improve their performances. There are four main algorithms that we will explore in this section: **Gradient Descent** or **Stochastic Gradient Descent (SGD)**, **Momentum-Assisted Gradient Descent**, and **Accelerated Gradient Descent.**

While our ultimate goal is to minimize the cost function $J(W, B)$, it is useful to get an intuitive grasp of how these minimization algorithms operate, and evaluate its performance in lower dimensions. We will, therefore, use a choice of test functions for optimization, in order to visualize and evaluate the four algorithms explained above.

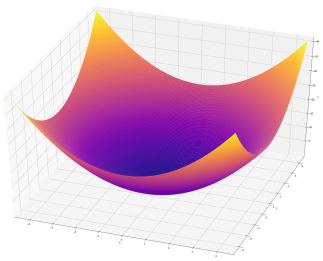
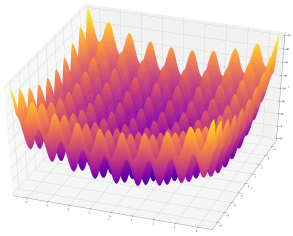
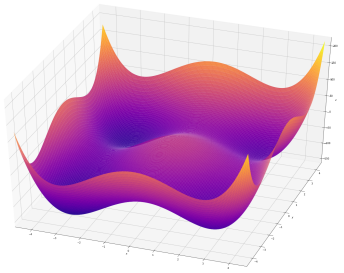
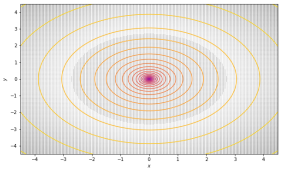
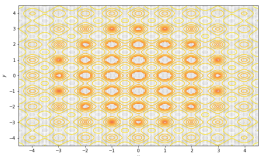
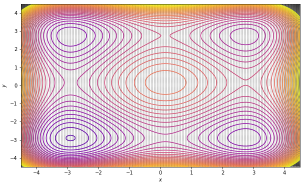
The test functions are chosen loosely according to the following criteria:

- Generalizable to higher dimensions
- Has Multiple Local Minima
- Differentiable in regions neighboring local minima ("smooth")

as this will ensure that the test functions can, though roughly, be indicative of the performance in the real cost function.

The following three functions in *Table 3.0.1* that satisfy each criteria were selected.

[Diagram 3.0.1: Test Functions for Evaluating Optimization Algorithms]

	Sphere Function	Rastrigin Function	Styblinski–Tang Function
Equation	$f(x) = \sum_{i=1}^{n} x_i^2$	$f(x) = 10n + \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i)]$	$f(x) = \frac{\Sigma_{i=1}^{n}[x_i^4 - 16_i^2 + 5x_i]}{2}$
3-D plot			
2-D Contour Plot			
Global Minima	$f(0,\ldots,0) = 0$	$f(0,\ldots,0) = 0$	$-39.16617n < f(\underbrace{-2.903534,\ldots,-2.903534}_{n \text{ times}}) < -39.16616n$

[6]

We will use all three of these functions at varying dimensions, as well as the neural network constructed in the previous section, to get an intuitive understanding of these algorithms, and later benchmark their performances.

(Continues on following page)

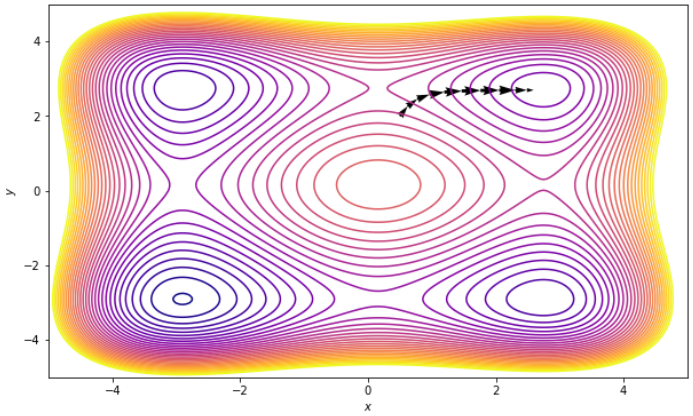
[6] For the code for generating these diagrams, reference *Appendix A*.

1. Gradient Descent and Stochastic Gradient Descent (SGD)

Gradient Descent, as explained earlier, simply takes the numerical gradient of the cost function at that point, and subtracts it from the current parameters, θ. In the n-th step of the gradient descent:

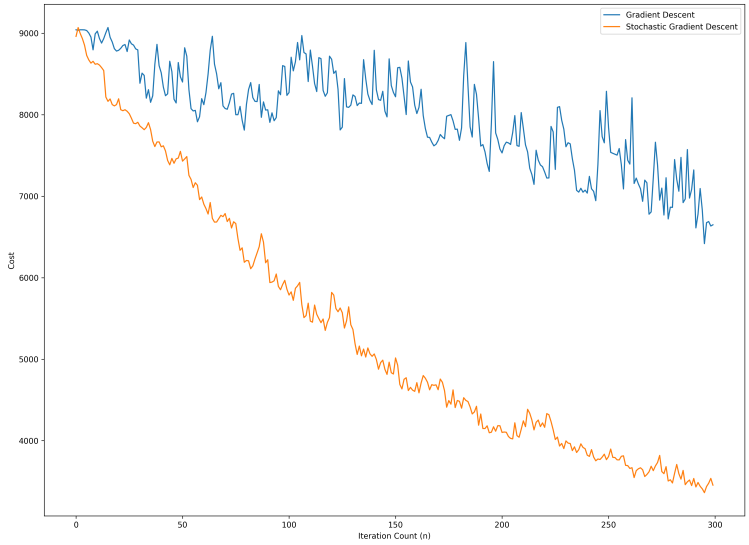
$$\theta_{n+1} = \theta_n - \eta \nabla J(\theta_n)$$

would calculate the next parameters θ_{n+1}. Executed iteratively, θ will gradually move to a local minima on a smooth surface. Within the Styblinski-Tang Function in three dimensions, starting at the point *(0.5, 2.0)*, 10 steps of gradient descent brings us to the minimum at *(2.59,2.70)*, close to the local minimum at *(2.90, 2.90),* as shown in *Diagram 3.1.1.* Each arrow indicates a "step" of gradient descent.



[Diagram 3.1.1: Optimizing the Styblinski-Tang Function using Gradient Descent]

However, for the real cost function of a neural network, calculating the gradient at every instant, or, *every training data*, is computationally intensive. Therefore, the gradient is calculated for a *set*, or a *mini-batch* of training data to speed up the process. This is known as *stochastic gradient descent*. (This concept is not present in our test functions, as their explicit form is shown and their gradient easily calculable.)



[Diagram 3.1.2: Decrease in the Cost of the Neural Network with Gradient Descent vs. Stochastic Gradient Descent

We can experimentally see that this indeed is the case; plotting the error rate of gradient descent and stochastic gradient descent at every step n, we can see that the cost of the neural network decreases more rapidly in the latter, as shown in *Diagram 3.1.2*.

2. Momentum-Assisted Gradient Descent

Simple gradient descent, or stochastic gradient descent, suffer from the fact that the size of the step, known as the "learning rate" η, cannot be changed—the size of the step is simply proportional to the magnitude of the gradient vector at the point. If the knowledge about the direction and magnitude of previous steps could be encoded into the current step, possibly, we could reduce the number of steps needed to reach an acceptable minimum.

Instead of directly modifying the parameters, θ, we introduce an "update vector," v, to store information about the previous step. We calculate the current update vector by summing, as usual, the step, with the gradient at the current location—but also the value of the previous update vector.

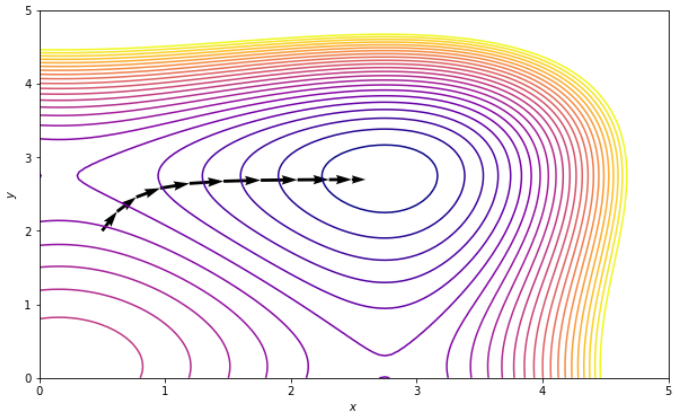
$$v_n = \eta \nabla J(\theta) + \underbrace{\gamma v_{n-1}}_{\textbf{Momentum}}$$

This *term,* of the previous vector, multiplied with the momentum constant γ to fine-tune its impact, is known as "momentum." The magnitude and direction of the previous step is encoded in the current term, and influences the magnitude and direction of the current step.

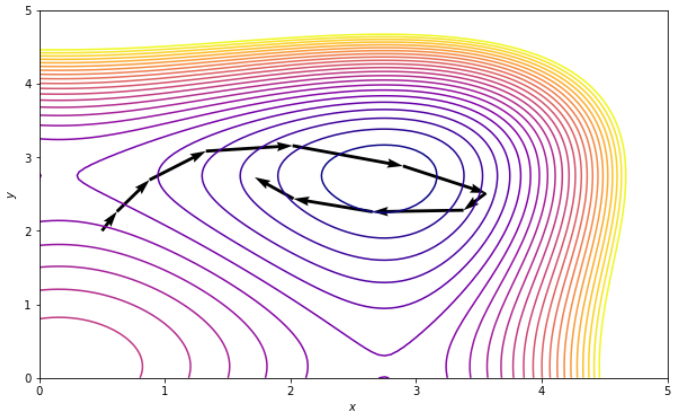
We can then calculate the new parameters of the cost function as usual:

$$\theta_{n+1} = \theta_n - v_n$$

Visually, we can see that as the steps are taken, the direction and the magnitude of the steps increase with regards to the previous steps, and therefore the minimum can be reached with a smaller number of steps.



[Diagram 3.2.1: Simple Gradient Descent]



[Diagram 3.2.2: Gradient Descent with Momentum]

Do note, however, that the increase in momentum can cause an "overshoot," i.e. passing through the local minimum, as evident in *Diagram 3.2.2.* This is a problem that can be addressed with the following concept.

3. Accelerated Gradient Descent

As we saw in *Diagram 3.2.2*, Momentum-Assisted SDG will often over-shoot and pass through the minima, due to the accumulated "momentum" from the previous steps. To solve this, we would want the algorithm to be aware of not only its previous steps and the gradient, but also the anticipated gradient for its next step, in order to be aware of the shape or curvature[7] in its neighborhood.

Looking at the equation for the update vector for momentum again:

$$v_n = \eta \nabla J(\theta_n) + \gamma v_{n-1}$$

Instead of calculating the gradient for the current parameters θ_n, as know the general direction of the step from the momentum term, γv_{n-1}, we can anticipate the location of the parameters in the next step: $\theta_n - \gamma v_{n-1}$; and therefore, also the gradient at that location:

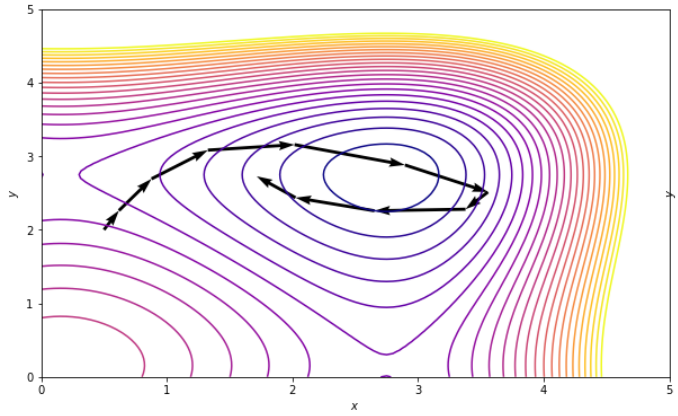
$$v_n = \eta \nabla J(\underbrace{\theta_n - \gamma v_{n-1}}_{\textbf{Aniticipated Next Gradient}}) + \underbrace{\gamma v_{n-1}}_{\textbf{Momentum}}$$

and calculate the new parameters of the cost function:

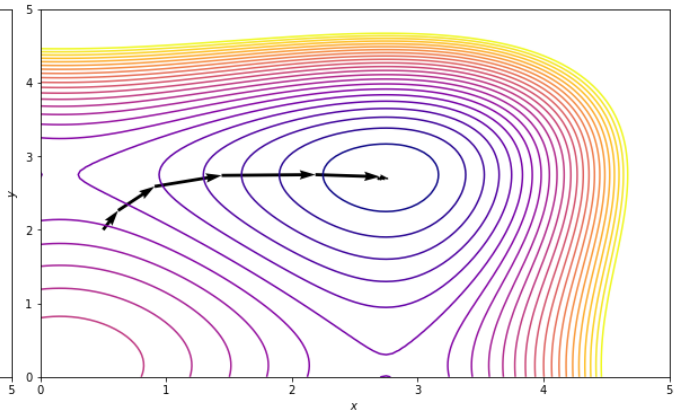
$$\theta_{n+1} = \theta_n - v_n$$

As this incorporates the "acceleration" of the position of the parameters, as if it were caught by a gravitational field, this concept is aptly named, "acceleration."

Visually comparing its movement with both gradient descent and momentum-assisted gradient descent, we can see that using acceleration dramatically reduces the number of steps to the minimum, without any over-shooting:



[Diagram 3.2.2: Gradient Descent with Momentum]

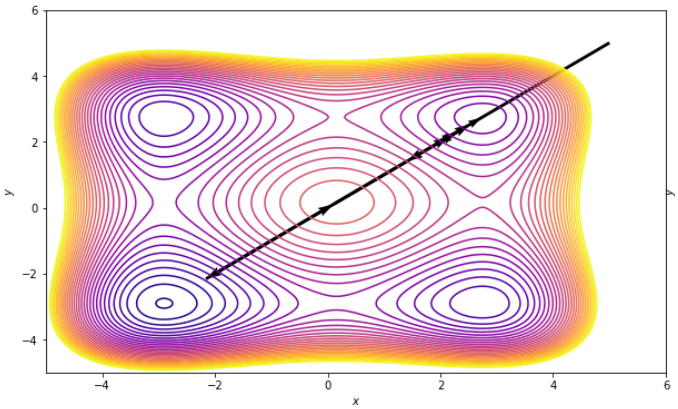


[Diagram 3.2.3: Gradient Descent with Acceleration]

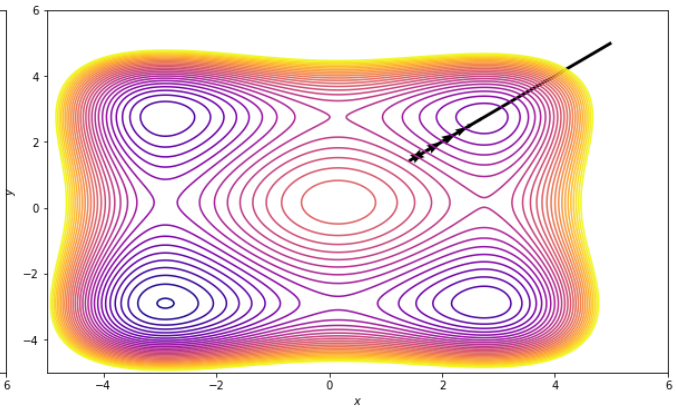
[7] These terms are used very loosely in this context.

4. A Need for Evaluation

In the Styblinsky-Tang function in three dimensions, using acceleration seemed to yield the best results; and this indeed is the case for many variations of the function. However, we must note that this is not always the case in higher dimensions, or even in cases where we start from a different initial position. For example, at domains outside of $-5 < x < 5$ and $-5 < y < 5$, acceleration will cause the parameters to oscillate significantly in the Styblinsky-Tang function; in this case, simple gradient descent appears to reach the minimum with fewer steps:



[Diagram 3.2.4: Styblinski-Tang, with Acceleration]



[Diagram 3.2.5: Styblinski-Tang, Simple Gradient Descent]

Therefore, we cannot always assume that acceleration will yield the best results for minimizing the cost function. At higher dimensions—like the cost function for our original neural network—this behavior cannot be visualized nor anticipated. Therefore, there is a need to evaluate our algorithms with various constraints and functions, to see how they may behave with certain cost functions.

IV. Evaluation of Optimization Functions for Training Neural Networks

As we have seen previously, different optimization algorithms have different characteristics that may be beneficial or harmful in different scenarios. Though our test functions and neural network is simple, for practical use we must be able to predict the rate of optimization more complex, high-dimensional functions, which we may anticipate by observing the behavior of algorithms with simpler functions.

We will, therefore, evaluate each algorithm using all three test functions to optimize at three or higher dimensions. Then, we will use the algorithm to optimize the neural network itself, and compare the results to our evaluation from test functions.

For each optimization algorithm:

A. Gradient Descent (GD) or Stochastic Gradient Descent (SGD)
B. Momentum Assisted SDG
C. Accelerated SDG.

we will evaluate their respective performances in:

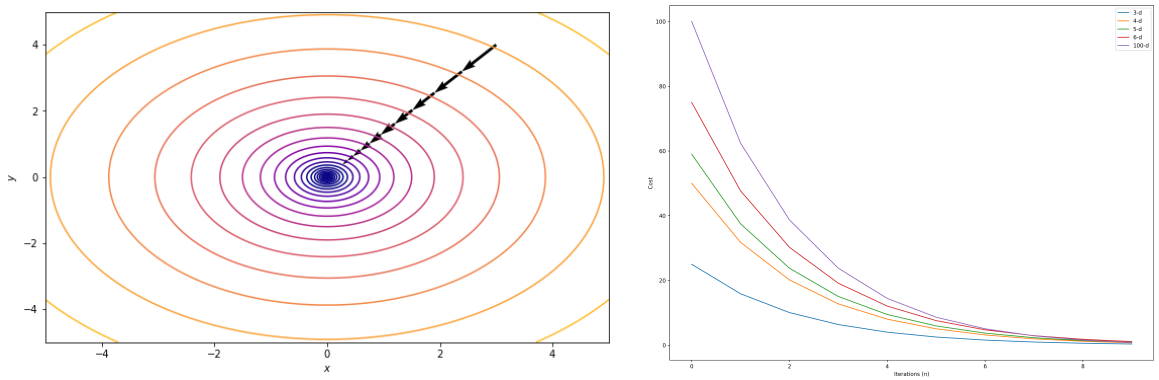
1. The Sphere Function in 3+ dimensions
2. The Rastrigin Function in 3+ dimensions
3. The Styblinski-Tang Function in 3+ dimensions
4. The Cost function of the neural network

in order to compare their respective performances. Both a quantitative analysis, seeing the decreasing cost; and a qualitative analysis, visualizing the movement in 3 dimensions; will be provided.

1. The Sphere Function

A sphere function has a gradient vector proportional in magnitude against the square of the distance from the global minimum, at any dimensions. Therefore, we can anticipate that it would be one of the easiest to optimize using gradient-based algorithms.

A. Gradient Descent

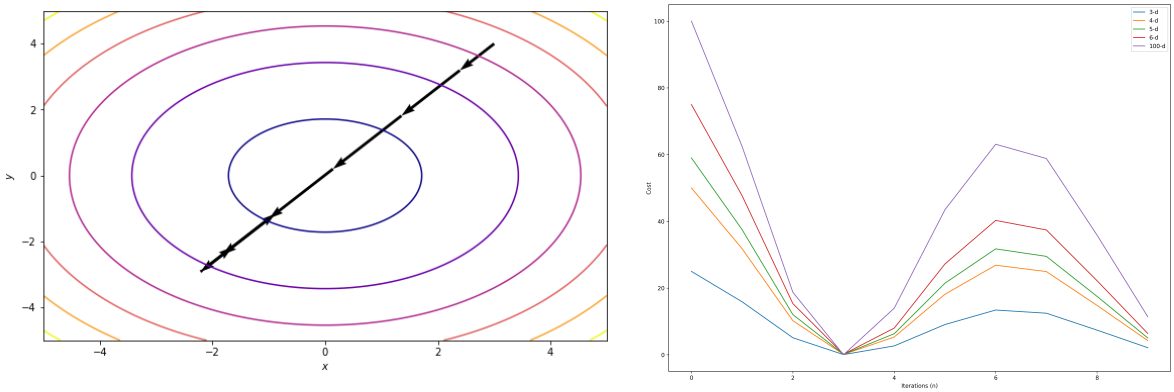


[Diagram 4.1.1: Optimizing the Sphere Function using Gradient Descent]

As anticipated, even with a simple gradient descent, only 10 steps are needed to approach the minimum reasonably closely. We can also see that the step sizes vary according to the gradient, and therefore, the farther away from the minimum, the bigger the step size.

The squared cost[8] from the global minimum of (0,…,0) for each iteration, at the sphere function at higher dimensions can be calculated and plotted, as seen in the graph.

B. Momentum



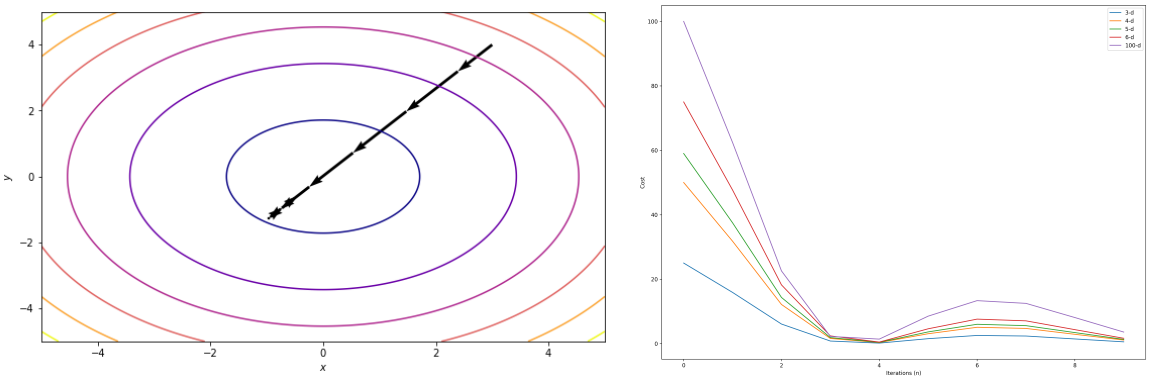
[Diagram 4.1.2: Optimizing the Sphere Function using Momentum-Assisted Gradient Descent]

With momentum, however, there is an "overshoot," passing through the gradient, and finishing the 10 iterations with a higher cost than simple gradient descent. This behavior is also evident in higher dimensions, as shown in the graph.

After 3 iterations of momentum assisted gradient descent, all dimensions of sphere functions demonstrate overshoot, resulting in an overall higher cost.

C. Acceleration

This overshoot is less pronounced when acceleration is introduced, also evident in the graph:



[Diagram 4.1.3: Optimizing the Sphere Function using Accelerated Gradient Descent]

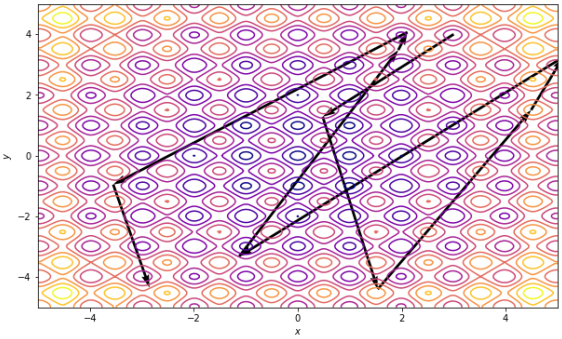
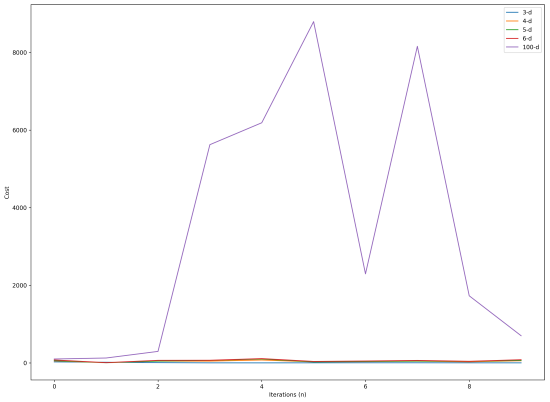
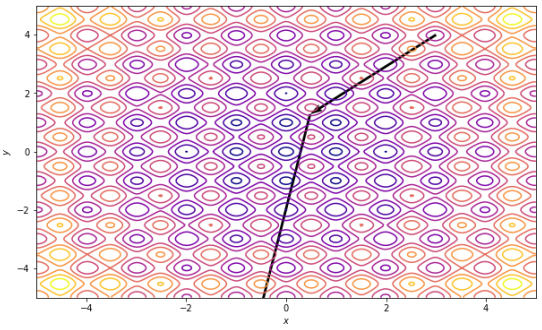
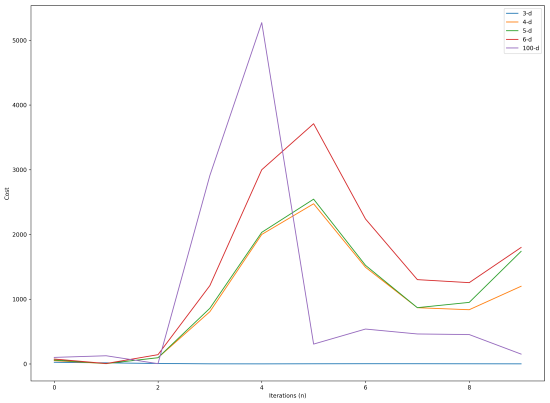
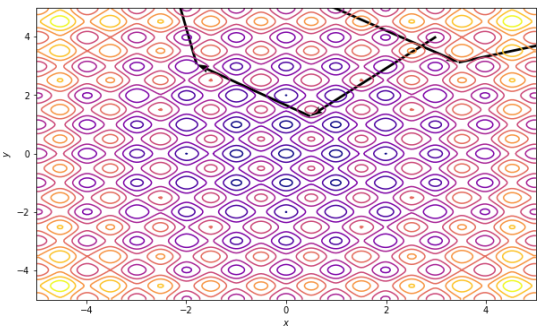
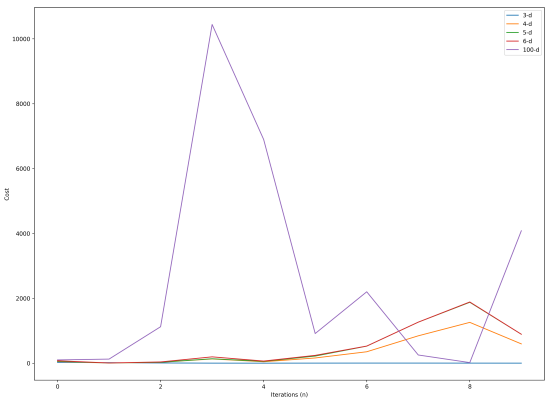
(Functions following the sphere function will be presented in a simpler table and their analysis in bullet form for easier understanding)

[8] Sum of squared cost.

2. The Rastrigin Function

The Rastrigin function is a comparatively difficult function to optimize to a global minima, due to the high magnitude of its gradient values. Though we will still assess the algorithms in a similar fashion, we can consider that even converging into a small, local minimum, would be enough of a success for our algorithm. The learning rate and iterations was decreased to 0.01, to account for the steep gradients to make the steps smaller.

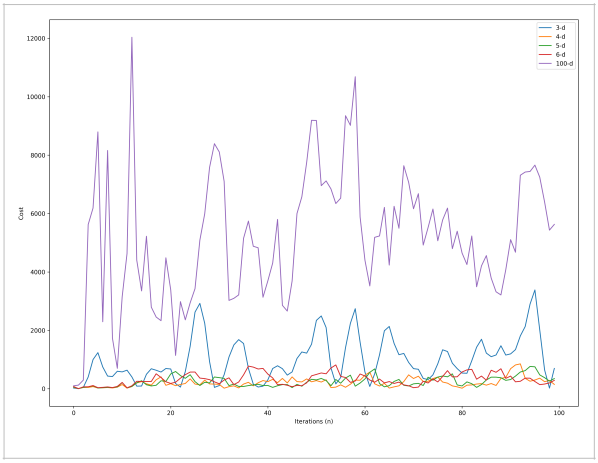
[Table 4.2.1: Optimizing the Rastrigin Function]

	Contour Plot, with Path	Cost, including Higher Dimensions
Gradient Descent		
Momentum-Assisted		
Accelerated		

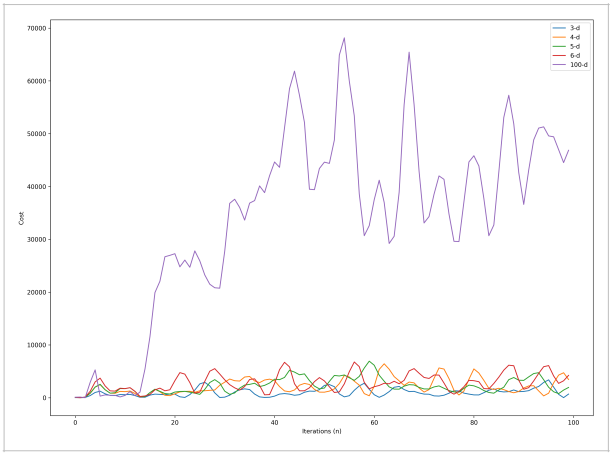
From the behavior in the contour plot and the graphs, we can conclude that:

- Gradient descent unexpected performs best and stays within a smaller neighborhood, possibly due to the fact that the gradient of the function fluctuates greatly based on even small changes in location
- Momentum assistance is more consistent in higher dimensions than acceleration, but finishes at a high cost for all functions; acceleration allows for acceptable minimization in lower dimensions but is seemingly random in high dimensions
- Overshooting due to momentum is ever-present, but is mitigable by acceleration somewhat in lower dimensions.

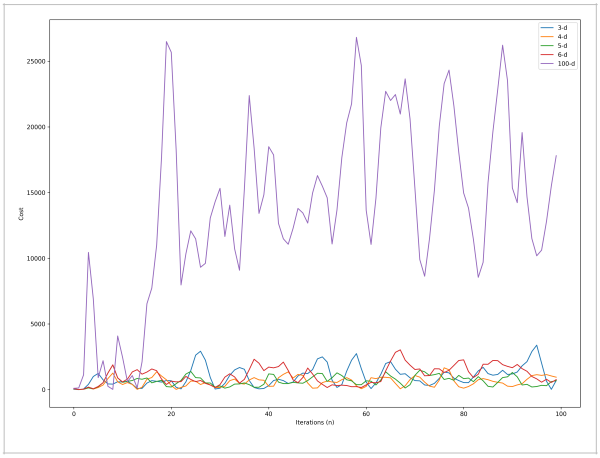
To assess the behavior of the algorithms in high dimensions and with more iterations, the same test was repeated with the dimensions and iterations adjusted. Taking into account the order of magnitude of the scale of the graphs in *Diagrams 4.2.2 - 4.2.4*, we can see that gradient descent deviated the least and maintained a low cost; momentum was more consistent across higher dimensions, although having a higher cost; and acceleration seemed to achieve a balance between the two.



[Diagram 4.2.2: With Gradient Descent]



[Diagram 4.2.3: With Momentum-assisted Gradient Descent]

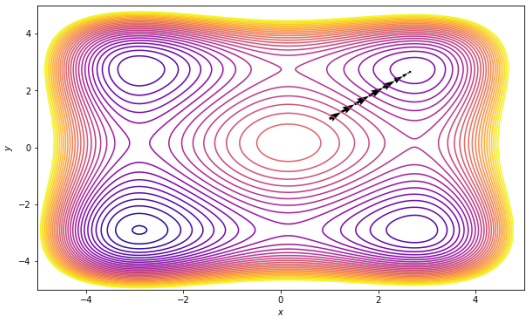
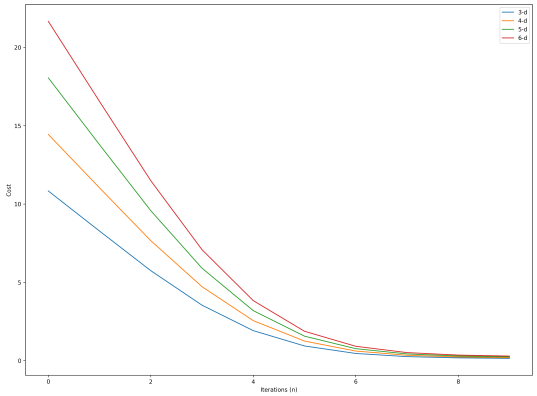
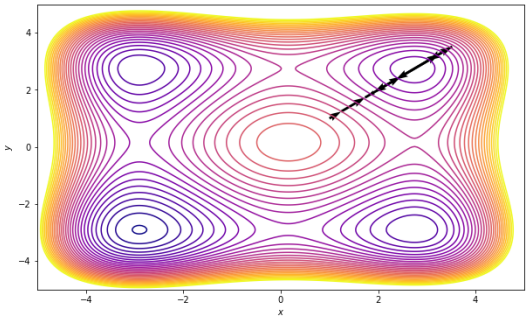
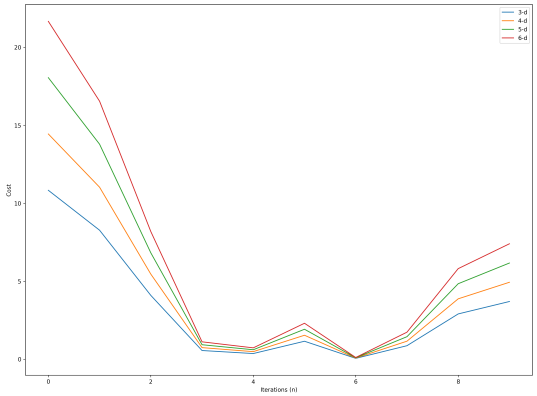
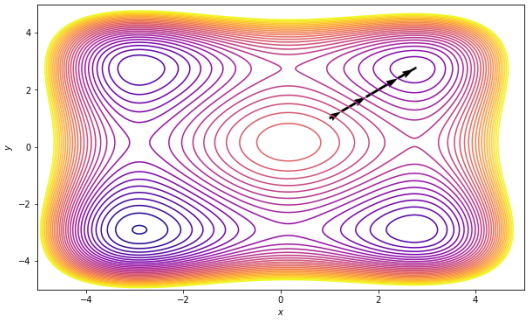
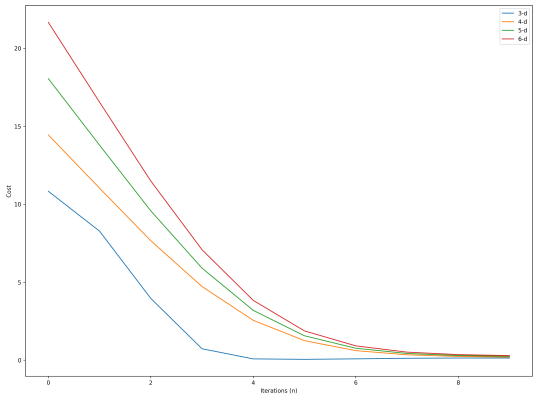


[Diagram 4.2.4: With Accelerated Gradient Descent]

3. The Styblinsky Tang Function

The Styblinsky-Tang Function offers a different challenge to our algorithm, as it has multiple local minima surrounded by smoothly curved surfaces—likely most similar to the cost function in our neural network.

[Table 4.3.1: Optimizing the Styblinsky Tang Function]

	Contour Plot, with Path	Cost, including Higher Dimensions
Gradient Descent		
Momentum-Assisted		
Accelerated		

- Gradient Descent with Acceleration performed the best, followed by plain gradient descent, and momentum.
- All algorithms generalized equally well into higher dimensions, with their cost converging rapidly.
- Overshooting continues to be an issue with momentum-based approaches; as the styblinski-tang function has multiple local minima, we can see the momentum algorithm struggles to move to a

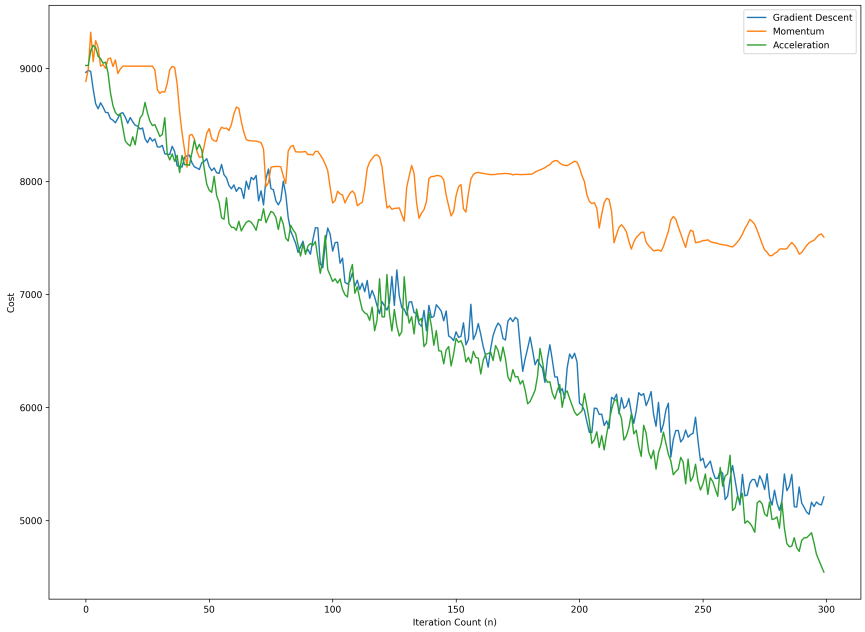
single minimum, but passes through it, reaching another local minimum, while passing through it as well.

Through our assessment of test functions, we can expect the behavior of the real cost function of the neural network. As evidenced, momentum-assisted descent consistently performed worse than plain gradient descent and accelerated gradient descent, due to the issue of momentum accumulating greatly and moving past the minimum. We also found that accelerated gradient descent mitigates these fluctuations, due to the fact that future trajectories are estimated.

4. The Cost function of the Neural Network

The final test for our algorithm would be a test based on the cost function of the original neural network (introduced in *Section II*). All three algorithms with functionally equivalent code was tested on the 9706-dimentional cost function of the neural network aimed at recognizing hand-written digits from monochrome images. *Diagram 4.3.2* reveals the rate of minimization of the three algorithms:

[Table 4.3.2: Optimizing the Neural Network]



Expectedly, the cost function of the neural network was minimized most efficiently with accelerated gradient descent, closely followed by plain gradient descent, and then momentum-assistance. When the network was left to train until above 90% accuracy (or, a cost of less than 1000) momentum was not able to meet the optimization goal after 30 minutes of runtime.

Despite the fact that our test functions were of lower dimension and were simpler, the qualitative behavior inherent in our algorithms were also present in the final test: the over-shooting present in moment-assistance, and the mitigation of fluctuations by incorporating the concept of acceleration. However, the quantitative values of the results of the cost functions were not particularly helpful in anticipating the algorithm's behavior; this can be attributed to the heuristic nature of optimization of neural networks; algorithms are developed not due to a numerical analysis of their efficiency, but rather the concepts they incorporate.

V. Conclusion

Neural networks continue to be a topic of fascination; integrating the strict rigor of mathematical optimization, to solving problems in real life. Optimizing such networks are inherently limited by computation, and algorithms are developed for practicality, rather than theoretical elegance. Despite this seeming deviation from the core ideologies of pure mathematics—the pursuit of simple elegance—, the fascination that many hold about neural networks lie in this complexity; the applicability of complex, applied mathematics—optimization—to solve seemingly illogical problems of the real world.

This unbelievable connection might be a glimpse into the mathematical nature of our world—while our minds seem to operate on intuition rather than strict reason, neural networks—which model our own brains—operate only with strict mathematical rigor. The algorithms that accelerate this computation, in turn, are developed heuristically, or, *intuitively*, which the network appears to be able to mimic. This fact—that optimization algorithms, developed with intuitive concepts, used to train a mathematical model, that can itself exhibit intuition—is another extraordinary evidence for the unreasonable effectiveness of mathematics in describing our universe.

> *Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.*
>
> *— Alan Turing*

VI. Bibliography

Imran, Abdullah. “Intuition of Gradient Descent for Machine Learning.” Medium. 13 Nov. 2017. medium.com/abdullah-al-imran/intuition-of-gradient-descent-for-machine-learning-49e1b6b89c8b. Accessed 20 Nov. 2018.

Tiao, Louis. “Visualizing and Animating Optimization Algorithms with Matplotlib.” 26 Apr. 2016. louistiao.me/notes/visualizing-and-animating-optimization-algorithms-with-matplotlib/. Accessed 20 Nov. 2018.

Bennett, Kristin et. al. “The Interplay of Optimization and Machine Learning Research.” 2006. www.jmlr.org/papers/volume7/MLOPT-intro06a/MLOPT-intro06a.pdf. Accessed 20 Nov. 2018.

Bennett, Defazio. “New Optimization Methods for Machine Learning.” 19 Mar. 2016. arxiv.org/pdf/1510.02533.pdf. Accessed 20 Nov. 2018.

Bottou, Leon et. al. “Optimization Methods for Machine Learning Part II – The theory of SG” 2016. icml.cc/2016/tutorials/part-2.pdf. Accessed 20 Nov. 2018.

Nielsen, Michael. “Neural Networks and Deep Learning.” 2015. neuralnetworksanddeeplearning.com/chap1.html. Determination Press. Accessed 20 Nov. 2018.

Surjanovic, Sonja. “Virtual Library of Simulation Experiments: Test Functions and Datasets.” Aug. 2017. www.sfu.ca/~ssurjano/optimization.html. Accessed 20 Nov. 2018.

VII. APPENDIX

Appendix A. Python code for generating function renders and optimizer performance and higher-resolution images. Code includes example diagrams and animations.

Appendix B. Python code for modeled neural networks and optimization functions. Optimizers functionally equivalent to those found in *Appendix A*.

The two appendices are formatted as an iPython notebook, and interactive application used for various purposes. The program jupyter was used to develop and run this notebook. The program is described in its homepage *jupyter.org*, likewise:

> "The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more."

The code was executed on macOS 10.14.1 with python 2.7 and other required packages installed via homebrew.

Appendix A. Python code for generating function renders and optimizer performance and higher-resolution images. Code includes example diagrams and animations.

In [1]:
```
%matplotlib inline
```

In [2]:
```
import matplotlib.pyplot as plt
import autograd.numpy as np
import math
import pickle

from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LogNorm
from matplotlib import animation
from IPython.display import HTML

from autograd import elementwise_grad, value_and_grad
from scipy.optimize import minimize
from scipy.optimize import optimize
from collections import defaultdict
from itertools import izip_longest
from functools import partial
```

In [3]:
```
'''
Test functions written by Pyokyeong Son
'''

def sphere(*args):
    k = 0
    for i in args: k += i**2
    return k
logSpace = np.logspace(-5, 2, 35)
logNorm = LogNorm()

def rastrigin(*args):
    A = 10
    return A + sum([(i**2 - A * np.cos(2 * math.pi * i)) for i in a
rgs])

def styblinskiTang(*args):
    return sum([(i**4 - 16 * i**2 + 5 * i) for i in args])
stSpace = np.linspace(-200, 100, 35)
stNorm = None

# f = rastrigin
# name = "rastrigin"
# fspace = logSpace
# fnorm = logNorm
```

Initialize Parameters for Drawing Plot Diagrams

In [4]:
```
# x_lims formatted as [xmin, xmax, xstep] as a list
def get_mesh_for_plot(f, x_lims, y_lims):
    xmin, xmax, xstep = tuple(x_lims)
    ymin, ymax, ystep = tuple(y_lims)
    x, y = np.meshgrid(np.arange(xmin, xmax + xstep, xstep),
                       np.arange(ymin, ymax + ystep, ystep))
    z = f(x, y)
    return x, y, z
```

In [3]:
```
'''
Following functions written by Pyokyeong Son
following tutorial by:
http://louistiao.me/notes/visualizing-and-
animating-optimization-algorithms-with-matplotlib/
'''
```

Out[3]:
```
'\nFollowing functions written by Pyokyeong Son\nfollowing tutoria
l by:\nhttp://louistiao.me/notes/visualizing-and-\nanimating-optim
ization-algorithms-with-matplotlib/\n'
```

3-D Plot Function

In [5]:
```
def three_dimentional_plot(f, x_lims, y_lims):
    x, y, z = get_mesh_for_plot(f, x_lims, y_lims)
    xmin, xmax, xstep = tuple(x_lims)
    ymin, ymax, ystep = tuple(y_lims)

    fig = plt.figure(figsize=(32, 22))
    ax = plt.axes(projection='3d', elev=40, azim=-70)

    ax.plot_surface(x, y, z, rstride=1, cstride=1,
                    cmap=plt.cm.plasma, linewidth=1, antialiased=Tr
ue)
    # ax.plot(*minima_, f(*minima), 'r*', markersize=20, color="r")

    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.set_zlabel('$z$')

    ax.set_xlim((xmin, xmax))
    ax.set_ylim((ymin, ymax))

    plt.show()

#three_dimentional_plot(sphere, [-5, 5, 0.05], [-5, 5, 0.05])
#three_dimentional_plot(rastrigin, [-5, 5, 0.05], [-5, 5, 0.05])
#three_dimentional_plot(styblinskiTang, [-5, 5, 0.05], [-5, 5, 0.05
])
```

Plot Contour Map

In [6]:
```
def contour_plot(f, x_lims, y_lims):
    x, y, z = get_mesh_for_plot(f, x_lims, y_lims)
    xmin, xmax, xstep = tuple(x_lims)
    ymin, ymax, ystep = tuple(y_lims)

    dz_dx = elementwise_grad(f, argnum=0)(x, y)
    dz_dy = elementwise_grad(f, argnum=1)(x, y)

    fig, ax = plt.subplots(figsize=(10, 6))

    ax.contour(x, y, z, cmap=plt.cm.plasma) # took out levels=fspac
e, norm=fnorm,
    ax.quiver(x, y, x - dz_dx, y - dz_dy, alpha=.5)
    #ax.plot(*minima_, 'r*', markersize=18)

    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')

    ax.set_xlim((xmin, xmax))
    ax.set_ylim((ymin, ymax))

    plt.show()

#contour_plot(sphere, [-5, 5, 0.05], [-5, 5, 0.05])
#contour_plot(rastrigin, [-5, 5, 0.05], [-5, 5, 0.05])
#contour_plot(styblinskiTang, [-5, 5, 0.05], [-5, 5, 0.05])
```

Optimization

In [27]:
```
def sphere2(x):
    return sum([i**2 for i in x])

def rastrigin2(x):
    A = 10
    return A + sum([(i**2 - A * np.cos(2 * math.pi * i)) for i in x
])

def styblinskiTang2(x):
    return sum([(i**4 - 16 * i**2 + 5 * i) for i in x])

'''
Optimizer_test class written by Pyokyeong Son
'''
class Optimizer:

    def __init__(self, f, eta, grad_step = 0.1):
        self.f = f
        self.eta = eta
        self.grad_step = grad_step

    def gradient_descent(self, param, count):
        path = [param]
        for i in range(count):
            param = self.update(param, self.f)
```

```
            path.append(param)

        path = np.array(path).T
        return path

    def gd_momentum(self, param, count, momentum=0.9):
        path = [param]
        update_vector = 0
        for i in range(count):
            (param, update_vector) = self.update_momentum(param,
                                                          self.f, m
omentum, update_vector)
            path.append(param)

        path = np.array(path).T
        return path

    def gd_acceleration(self, param, count, momentum=0.9):
        path = [param]
        update_vector = 0
        for i in range(count):
            (param, update_vector) = self.update_accel(param,
                                                       self.f, mome
ntum, update_vector)
            path.append(param)

        path = np.array(path).T
        return path

    def update(self, params, f):

        delta_params = self.eta * optimize.approx_fprime(params, f,
self.grad_step)
        #print "gradient: {0}".format(delta_params)
        params = [p - dp for p, dp in zip(params, delta_params)]

        return params

    def update_momentum(self, params, f, momentum, prev_update_vect
or):

        update_vector = (momentum * prev_update_vector + self.eta *
                         optimize.approx_fprime(params, f, self.gra
d_step))
        #print "gradient: {0}".format(optimize.approx_fprime(params
, f, self.eta))
        params = [p - dp for p, dp in zip(params, update_vector)]

        return (params, update_vector)

    def update_accel(self, params, f, momentum, prev_update_vector)
:

        update_vector = (momentum * prev_update_vector + self.eta *
                         optimize.approx_fprime(params - prev_updat
e_vector, f, self.grad_step))
        #print "gradient: {0}".format(optimize.approx_fprime(params
```

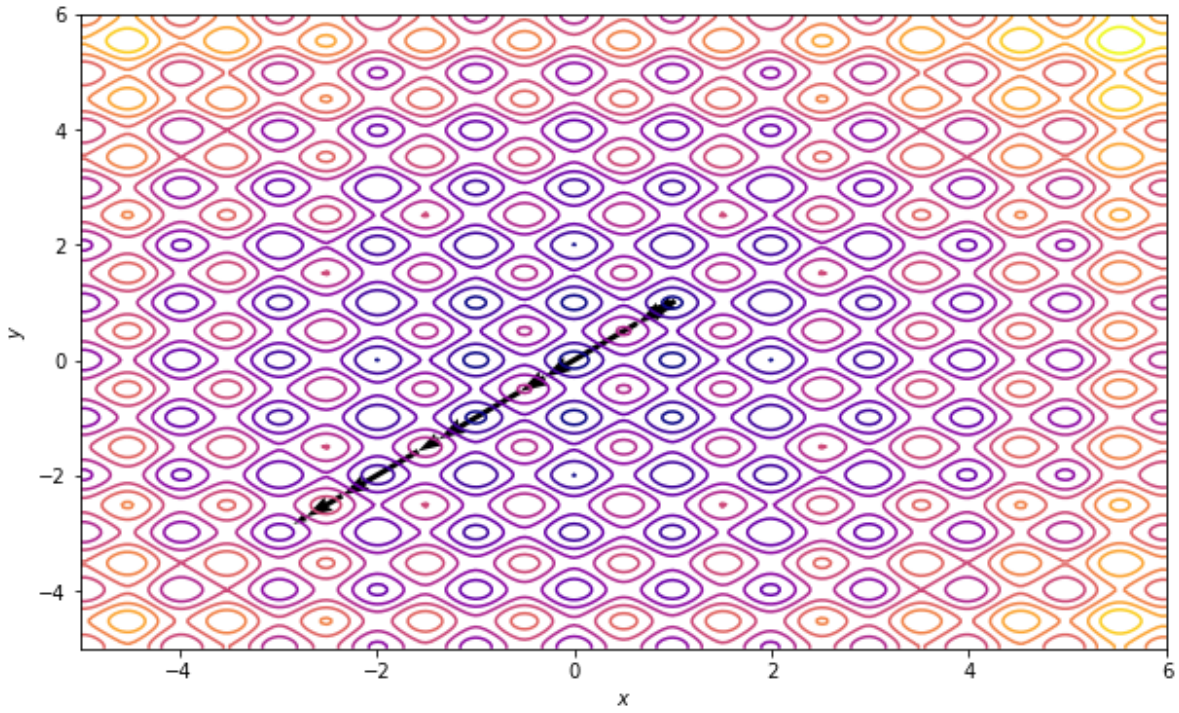
```
, f, self.eta))
        params = [p - dp for p, dp in zip(params, update_vector)]

        return (params, update_vector)

def optimization_plot(f, x_lims, y_lims, path):
    x, y, z = get_mesh_for_plot(f, x_lims, y_lims)
    xmin, xmax, xstep = tuple(x_lims)
    ymin, ymax, ystep = tuple(y_lims)

    fig, ax = plt.subplots(figsize=(10, 6))

    ax.contour(x, y, z, levels=stSpace,
               norm=stNorm, cmap=plt.cm.plasma) # took out levels=f
space, norm=fnorm,
    ax.quiver(path[0,:-1], path[1,:-1],
              path[0,1:]-path[0,:-1],
              path[1,1:]-path[1,:-1],
              scale_units='xy', angles='xy', scale=1, color='k', wi
dth=0.005)
    #ax.plot(minima_, 'r*', markersize=10)

    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')

    ax.set_xlim((xmin, xmax))
    ax.set_ylim((ymin, ymax))

    plt.show()

def animate_plot(f, x_lims, y_lims, path):
    x, y, z = get_mesh_for_plot(f, x_lims, y_lims)
    xmin, xmax, xstep = tuple(x_lims)
    ymin, ymax, ystep = tuple(y_lims)

    fig, ax = plt.subplots(figsize=(10, 6))

    ax.contour(x, y, z,
               levels=stSpace, norm=stNorm, cmap=plt.cm.plasma) # t
ook out levels=fspace, norm=fnorm,
    # ax.plot(minima_, 'r*', markersize=18)

    line, = ax.plot([], [], 'b', label='Trajectory of Theta', lw=2)
    point, = ax.plot([], [], 'bo')

    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')

    ax.set_xlim((xmin, xmax))
    ax.set_ylim((ymin, ymax))

    ax.legend(loc='upper left')

    def init():
        line.set_data([], [])
        point.set_data([], [])
        return line, point
```

```
    def animate(i):
        line.set_data(*path[::,:i]) # comment out to not show the t
rajectory
        point.set_data(*path[::,i-1:i])
        return line, point

    return animation.FuncAnimation(fig, animate, init_func=init,
                                   frames=path.shape[1], interval=1
00,
                                   repeat_delay=5, blit=True)

def calc_cost(path):
    costs = []
    path = np.array(path).T
    for x in path:
        costs.append(sum([(i - 2.9)**2 for i in x]))
    return costs
```

Contour Path and Animation

In [8]:
```
opt = Optimizer(rastrigin2, 0.01)
path = opt.gd_acceleration(np.array([1., 1.]), 10)
#path = opt.gradient_descent(np.array([5., 5.]), 10)
print "final opt: {0},{1}".format(path[0][-1],path[1][-1])

x_lim = [-5, 6, 0.05]
y_lim = [-5, 6, 0.05]

optimization_plot(rastrigin, x_lim, y_lim, path)

#anim = animate_plot(styblinskiTang, x_lim, y_lim, path)
#HTML(anim.to_html5_video())
```

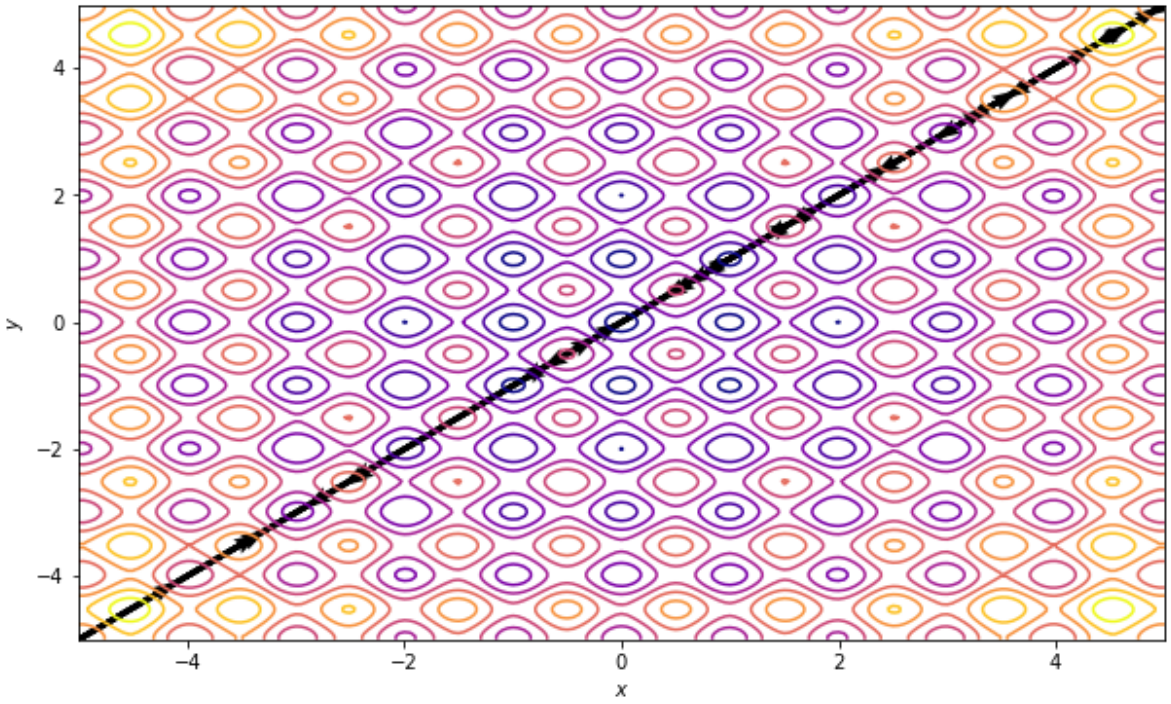
```
final opt: -2.84371132456,-2.84371132456
```



Example

In [22]:
```
opt_sphere = Optimizer(rastrigin2, 0.02)
path_sphere = opt_sphere.gd_momentum(np.ones(3), 99)
print "final opt: {0},{1}".format(path_sphere[0][-1],path_sphere[1]
[-1])

x_lim = [-5, 5, 0.05]
y_lim = [-5, 5, 0.05]

optimization_plot(rastrigin, x_lim, y_lim, path_sphere)

anim = animate_plot(styblinskiTang, x_lim, y_lim, path_sphere)
HTML(anim.to_html5_video())

cost_sphere = calc_cost(path_sphere)
```

```
final opt: -0.950772260287,-0.950772260287
```

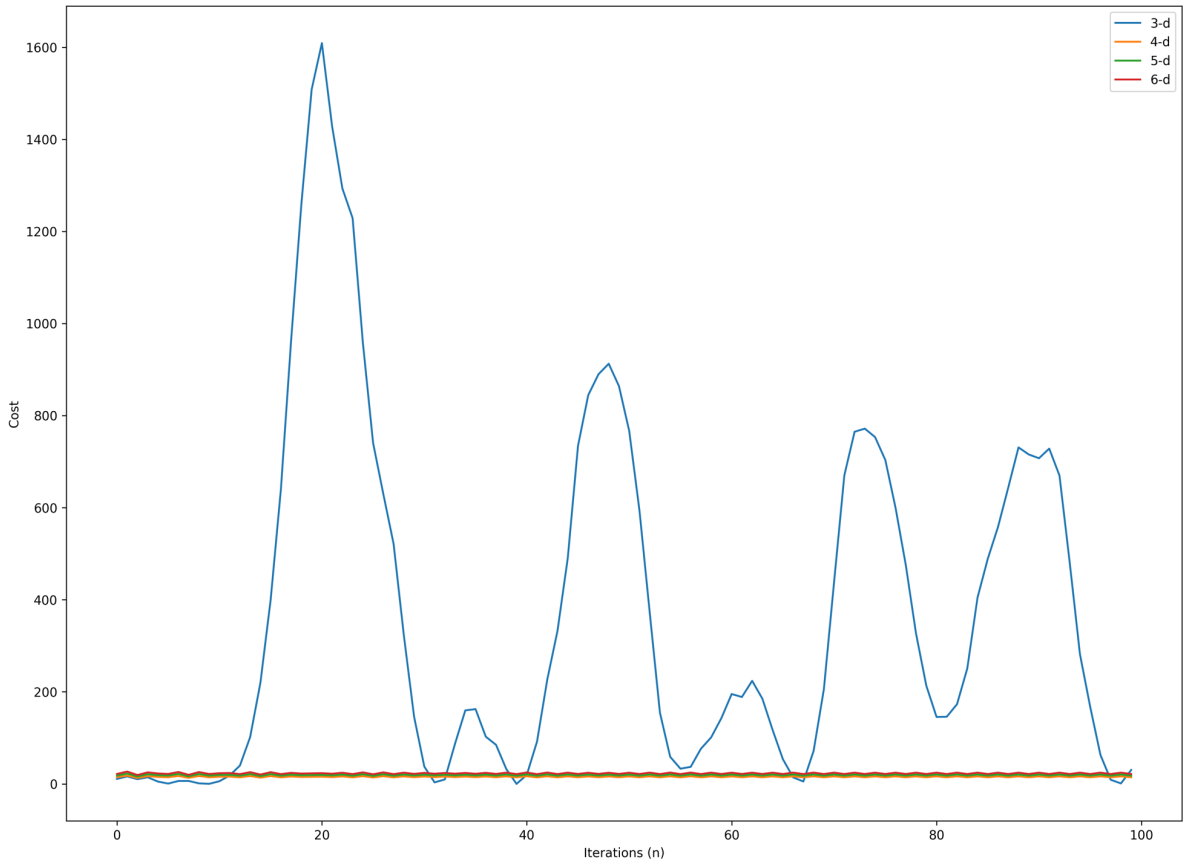


In [23]:
```
opt_sphere = Optimizer(rastrigin2, 0.01)
path_sphere4 = opt_sphere.gd_momentum(np.ones(4), 99)
cost_sphere4 = calc_cost(path_sphere4)

path_sphere5 = opt_sphere.gd_momentum(np.ones(5), 99)
cost_sphere5 = calc_cost(path_sphere5)

path_sphere6 = opt_sphere.gd_momentum(np.ones(6), 99)
cost_sphere6 = calc_cost(path_sphere6)

path_sphere100 = opt_sphere.gd_momentum(np.ones(100), 99)
cost_sphere100 = calc_cost(path_sphere100)

print len(cost_sphere)
```

```
100
```

In [24]:
```
plt.figure(figsize=(16,12), dpi=300)
plt.plot(range(100), cost_sphere, label='3-d')
plt.plot(range(100), cost_sphere4, label='4-d')
plt.plot(range(100), cost_sphere5, label='5-d')
plt.plot(range(100), cost_sphere6, label='6-d')
#plt.plot(range(10), cost_sphere100, label='100-d')

plt.xlabel("Iterations (n)")
plt.ylabel("Cost")
plt.legend()
```

Out[24]:
```
<matplotlib.legend.Legend at 0x115c60510>
```



Appendix B. Python code for modeled neural networks and optimization functions. Optimizers functionally equivalent to those found in *Appendix A*.

In [1]:
```
%matplotlib inline
```

In [3]:
```
import numpy as np
import random
import cPickle as pickle
import gzip
from prettytable import PrettyTable
import matplotlib.pyplot as plt
```

In [4]:
```
'''
Loading functions derived from: Michael Nielsen - Neural Networks a
nd Deep Learning.
https://github.com/mnielsen/neural-networks-and-deep-learning
'''

def load_data():
    f = gzip.open('mnist.pkl.gz', 'rb')
    training_data, validation_data, test_data = pickle.load(f)
    f.close()
    return (training_data, validation_data, test_data)

def load_data_wrapper():
    tr_d, va_d, te_d = load_data()
    training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]
    training_results = [vectorized_result(y) for y in tr_d[1]]
    training_data = zip(training_inputs, training_results)
    validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]
    validation_data = zip(validation_inputs, va_d[1])
    test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
    test_data = zip(test_inputs, te_d[1])
    return (training_data, validation_data, test_data)

def vectorized_result(j):
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e
```

In [5]:
```
'''
Network class written by Pyokyeong Son
Backpropagation function taken from: Michael Nielsen - Neural Netwo
rks and Deep Learning.
'''

class Network(object):

    ### Misc Functions ###

    def __init__(self, sizes, lim=300):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes
[:-1], sizes[1:])]
```

```
        self.lim = lim

    # Prints all the weights and biases of the network
    def print_weights_biases(self):
        with np.printoptions(precision=3, suppress=True):
            print("weights: ")
            for w in self.weights: print(w)
            print("biases: ")
            for b in self.biases: print(b)

    # Prints the output of neural network when given the input `inp
ut`
    def print_feedforward(self, input):
        print(self.feedforward(input))

    # Returns the output of neural network when given the input `in
put`
    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a



    ### Optimizer Functions ###

    # Optimizes the cost function using the gradient descent algori
thm, with the number
    def gradient_descent(self, training_data, data_count, test_data
, eta=0.1):

        n_test = len(test_data)
        history = []
        random.shuffle(training_data)
        training_data = training_data[0:data_count]

        # a single gradient descent step for every training_data
        for data in training_data:
            self.update_single_data(data, eta)
            if test_data:
                print "Accuracy: {0} / {1}".format(self.evaluate(te
st_data), n_test)
                history.append(self.evaluate(test_data))
        print "Training complete"

        return history

    def batch_gradient_descent(self, training_data, batch_size, tes
t_data, eta=0.1):

        n_test = len(test_data)

        n = len(training_data)
        random.shuffle(training_data)

        training_data = training_data[0:batch_size]
        self.update_batch(training_data, eta)
```

```
        print "Accuracy: {0} / {1}".format(
            self.evaluate(test_data), n_test)
        print "Training complete"

    def stochastic_gradient_descent(self, training_data, mini_batch
_size, test_data=None, eta=0.1):

        n_test = len(test_data)
        n_train = len(training_data)
        history = []

        random.shuffle(training_data)

        # split training data into mini batches
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n_train, mini_batch_size)]

        i = 0 # Counter for limiting number of batch trains
        print "Number of mini_batches: {0}".format(len(mini_batches
))

        # a single gradient descent step for every mini batch
        for mini_batch in mini_batches:
            self.update_batch(mini_batch, eta)
            if test_data:
                print "Batch {0}: {1} / {2}".format(i, self.evaluat
e(test_data), n_test)
                history.append(self.evaluate(test_data))
            i += 1
            if i == self.lim: break

        print "Training complete"

        return history

    def sgd_momentum(self, training_data, mini_batch_size, test_dat
a, momentum=0.9, eta=0.1):

        n_test = len(test_data)
        n_train = len(training_data)
        history = []

        random.shuffle(training_data)

        # split training data into mini batches
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n_train, mini_batch_size)]

        i = 0 # Counter for limiting number of batch trains
        print "Number of mini_batches: {0}".format(len(mini_batches
))

        # variables to store previous update vectors for momentum c
alculuation
```

```
        update_vector_weights = [ np.zeros(w.shape) for w in self.w
eights ]
        update_vector_biases = [ np.zeros(b.shape) for b in self.bi
ases ]

        # a single gradient descent step for every mini batch
        for mini_batch in mini_batches:
            (update_vector_weights, update_vector_biases) = self.up
date_batch_momentum(mini_batch, (update_vector_weights, update_vect
or_biases), momentum, eta)
            if test_data:
                print "Batch {0}: {1} / {2}".format(i, self.evaluat
e(test_data), n_test)
                history.append(self.evaluate(test_data))
            i += 1
            if i == self.lim: break

        print "Training complete"

        return history

    def sgd_acceleration(self, training_data, mini_batch_size, test
_data, momentum=0.9, eta=0.1):

        n_test = len(test_data)
        n_train = len(training_data)
        history = []

        random.shuffle(training_data)

        # split training data into mini batches
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n_train, mini_batch_size)]

        i = 0
        print "Number of mini_batches: {0}".format(len(mini_batches
))

        # variables to store previous update vectors for momentum c
alculuation
        update_vector_weights = [ np.zeros(w.shape) for w in self.w
eights ]
        update_vector_biases = [ np.zeros(b.shape) for b in self.bi
ases ]

        # a single gradient descent step for every mini batch
        for mini_batch in mini_batches:
            (update_vector_weights, update_vector_biases) = self.up
date_batch_accel(mini_batch, (update_vector_weights, update_vector_
biases), momentum, eta)
            if test_data:
                print "Batch {0}: {1} / {2}".format(i, self.evaluat
e(test_data), n_test)
                history.append(self.evaluate(test_data))
            i += 1
            if i == self.lim: break
```

```
        print "Training complete"

        return history

    def sgd_epoch(self, training_data, mini_batch_size, test_data,
epochs=3):

        n_test = len(test_data)
        n_train = len(training_data)

        # Train with mini batches for every epoch
        for j in xrange(epochs):
            random.shuffle(training_data)

            # split training data into mini batches
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n_train, mini_batch_size)]

            # a single gradient descent step for every mini batch
            for mini_batch in mini_batches:
                self.update_batch(mini_batch)

            # if the user requests a test
            if test_data:
                print "Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test)
            else:
                print "Epoch {0} complete".format(j)

        print "Training complete"



    ### Optimizer Helper Functions ###

    def update_batch(self, mini_batch, eta):

        delta_weights = [ np.zeros(w.shape) for w in self.weights ]
        delta_biases = [ np.zeros(b.shape) for b in self.biases ]

        for x, y in mini_batch:
            delta_biases_temp, delta_weights_temp = self.backProp(x
, y)
            delta_weights = [ dwp + dwtp for dwp, dwtp in zip(delta
_weights, delta_weights_temp) ]
            delta_biases = [ dbp + dbtp for dbp, dbtp in zip(delta_
biases, delta_biases_temp) ]

        self.weights = [w - eta * dw for w, dw in zip(self.weights,
delta_weights)]
        self.biases = [b - eta * db for b, db in zip(self.biases, d
elta_biases)]

    def update_batch_momentum(self, mini_batch, prev_update_vectors
, momentum, eta):
```

```
        # Instantaize change needed to parameters
        delta_weights = [ np.zeros(w.shape) for w in self.weights ]
        delta_biases = [ np.zeros(b.shape) for b in self.biases ]

        # unpack previous update vectors
        prev_update_vector_weights, prev_update_vector_biases = pre
v_update_vectors

        # sum the totoal deltas
        for x, y in mini_batch:
            delta_biases_temp, delta_weights_temp = self.backProp(x
, y)
            delta_weights = [ dwp + dwtp for dwp, dwtp in zip(delta
_weights, delta_weights_temp) ]
            delta_biases = [ dbp + dbtp for dbp, dbtp in zip(delta_
biases, delta_biases_temp) ]

        # sum the momentums from previous update vector
        update_vector_weights = [ momentum * puvw + eta * dw for pu
vw, dw in zip(prev_update_vector_weights, delta_weights)]
        update_vector_biases = [ momentum * puvb + eta * db for puv
b, db in zip(prev_update_vector_biases, delta_biases)]

        # update the weights and biases accordingly
        self.weights = [w - dw for w, dw in zip(self.weights, updat
e_vector_weights)]
        self.biases = [b - db for b, db in zip(self.biases, update_
vector_biases)]

        return (update_vector_weights, update_vector_biases)

    def update_batch_accel(self, mini_batch, prev_update_vectors, m
omentum, eta):

        # Instantaize change needed to parameters
        delta_weights = [ np.zeros(w.shape) for w in self.weights ]
        delta_biases = [ np.zeros(b.shape) for b in self.biases ]

        # unpack previous update vectors
        prev_update_vector_weights, prev_update_vector_biases = pre
v_update_vectors

        # sum the totoal deltas, but accounting for acceleration
        for x, y in mini_batch:
            delta_biases_temp, delta_weights_temp = self.backProp_a
ccel(x, y, prev_update_vector_weights, prev_update_vector_biases)
            delta_weights = [ dwp + dwtp for dwp, dwtp in zip(delta
_weights, delta_weights_temp) ]
            delta_biases = [ dbp + dbtp for dbp, dbtp in zip(delta_
biases, delta_biases_temp) ]

        # sum the momentums from previous update vector
        update_vector_weights = [ momentum * puvw + eta * dw for pu
vw, dw in zip(prev_update_vector_weights, delta_weights)]
        update_vector_biases = [ momentum * puvb + eta * db for puv
b, db in zip(prev_update_vector_biases, delta_biases)]
```

```
        # update the weights and biases accordingly
        self.weights = [w - dw for w, dw in zip(self.weights, updat
e_vector_weights)]
        self.biases = [b - db for b, db in zip(self.biases, update_
vector_biases)]

        return (update_vector_weights, update_vector_biases)

    def update_single_data(self, single_data, eta):

        (x, y) = single_data

        delta_biases, delta_weights = self.backProp(x, y)

        self.weights = [w - eta * dw for w, dw in zip(self.weights,
delta_weights)]
        self.biases = [b - eta * db for b, db in zip(self.biases, d
elta_biases)]



    ### Backpropagation Implementations ###

    def backProp(self, x, y):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]

        # feedforward
        activation = x
        activations = [x] # list to store all the activations, laye
r by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)

        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())

        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) *
sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose
())
        return (nabla_b, nabla_w)

    def backProp_accel(self, x, y, puvw, puvb):
```

```
        biases_temp = [ b - db for b, db in zip(self.biases, puvb)
]
        weights_temp = [ w - dw for w, dw in zip(self.weights, puvw
) ]

        nabla_b = [np.zeros(b.shape) for b in biases_temp]
        nabla_w = [np.zeros(w.shape) for w in weights_temp]

        # feedforward
        activation = x
        activations = [x] # list to store all the activations, laye
r by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(biases_temp, weights_temp):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)

        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())

        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(weights_temp[-l+1].transpose(), delta) *
sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose
())
        return (nabla_b, nabla_w)

    def evaluate(self, test_data):
        test_results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)

    def cost_derivative(self, output_activations, y):
        return (output_activations-y)



### Miscellaneous Functions ###

def sigmoid(z):
    # The sigmoid function
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    # Derivative of the sigmoid function, used in backpropagation
    return sigmoid(z)*(1-sigmoid(z))
```

In [6]:
```
training_data, validation_data, test_data = load_data_wrapper()
```

In [15]:
```
lim = 10

net = Network([784, 12, 12, 10], lim)
history_stochastic = net.stochastic_gradient_descent(training_data,
30, test_data=test_data, eta=0.1)

net = Network([784, 12, 12, 10], lim)
history_momentum_point_three = net.stochastic_gradient_descent_mome
ntum(training_data, 30, test_data=test_data, momentum=0.3, eta=0.1)

net = Network([784, 12, 12, 10], lim)
history_momentum_point_nine = net.stochastic_gradient_descent_momen
tum(training_data, 30, test_data=test_data, momentum=0.9, eta=0.1)

net = Network([784, 12, 12, 10], lim)
history_accel_point_three = net.stochastic_gradient_descent_acceler
ation(training_data, 30, test_data=test_data, momentum=0.3, eta=0.1
)

net = Network([784, 12, 12, 10], lim)
history_accel_point_nine = net.stochastic_gradient_descent_accelera
tion(training_data, 30, test_data=test_data, momentum=0.9, eta=0.1)
```

```
Number of mini_batches: 1667
Batch 0: 872 / 10000
Batch 1: 876 / 10000
Batch 2: 880 / 10000
Batch 3: 912 / 10000
Batch 4: 937 / 10000
Batch 5: 974 / 10000
Batch 6: 979 / 10000
Batch 7: 1037 / 10000
Batch 8: 1094 / 10000
Batch 9: 1035 / 10000
Training complete
```

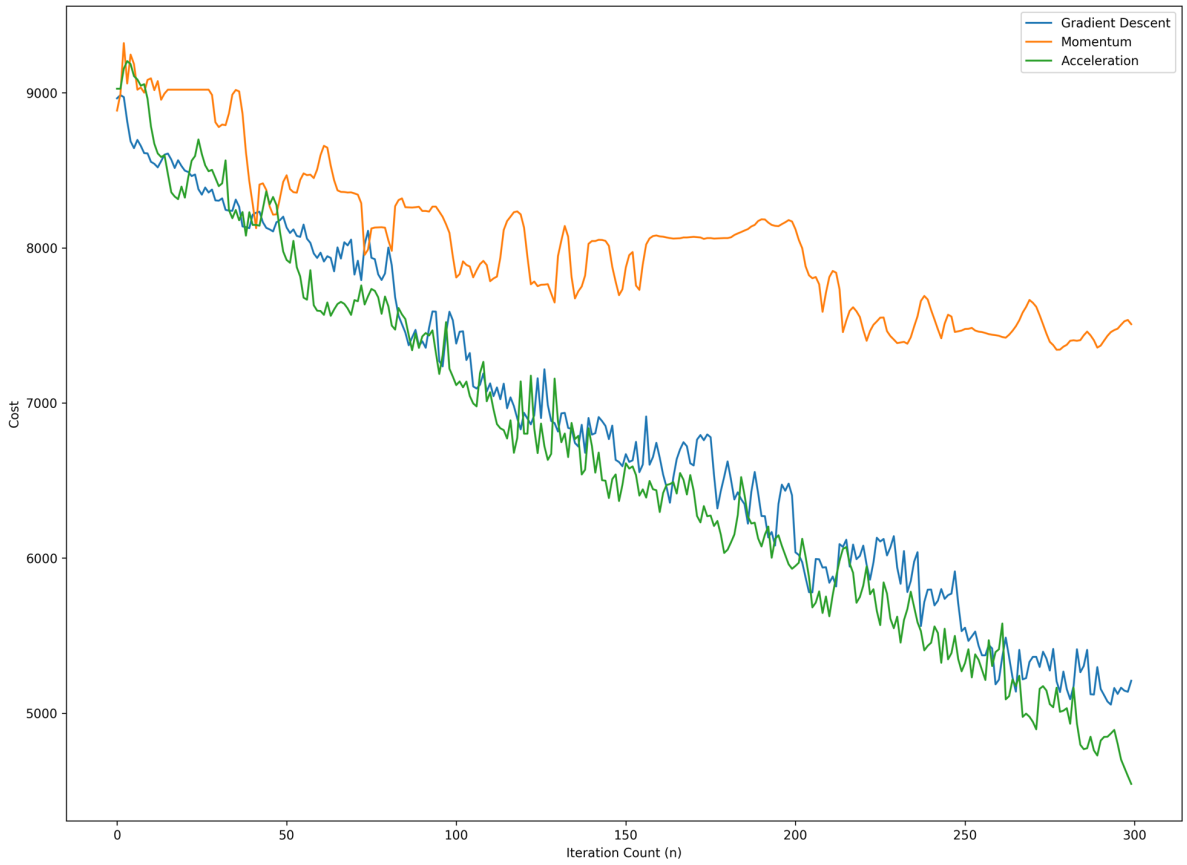
In [91]:
```
plt.figure(figsize=(16,12), dpi=300)
plt.plot(range(lim), history_stochastic, label='stochastic')
plt.plot(range(lim), history_momentum_point_three, label='momentum
0.3')
plt.plot(range(lim), history_momentum_point_nine, label='momentum 0
.9')
plt.plot(range(lim), history_accel_point_three, label='acceleration
0.3')
plt.plot(range(lim), history_accel_point_nine, label='acceleration
0.9')

plt.xlabel("Batch Count")
plt.ylabel("Accuracy")
plt.legend()
```

Out[91]:
```
<matplotlib.legend.Legend at 0x10f359750>
```

In [94]:
```
histories = [history_stochastic,
             history_momentum_point_three,
             history_momentum_point_nine,
             history_accel_point_three,
             history_accel_point_nine]

pickle.dump( histories, open( "histories.p", "wb" ) )
```

In [12]:
```
lim = 300
plt.figure(figsize=(16,12), dpi=300)
costs = []
for history in histories:
    costs.append([10000 - value for value in history])
plt.plot(range(lim), costs[0], label='Gradient Descent')
plt.plot(range(lim), costs[2], label='Momentum')
plt.plot(range(lim), costs[3], label='Acceleration')
plt.xlabel("Iteration Count (n)")
plt.ylabel("Cost")
plt.legend()
```

Out[12]:
```
<matplotlib.legend.Legend at 0x1168c4710>
```



In [8]:
```
histories = pickle.load( open( "histories.p", "rb" ) )
```

In [ ]:
```
```