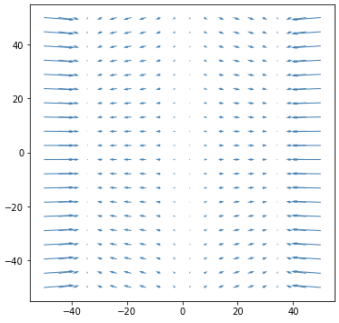
Utilizing The iRProp⁻ Algorithm in Mathematical Optimization

Jason Bonner

1 Introduction

Optimization algorithms are programs that, given a certain input or set of inputs, can produce the maximum, or *optimized* values of a function. In the development of this project for example, given the function $f(x, y) = -.005(x^4 - 2x^3 - 2450x^2 + 7350x) - y^2$ an optimization algorithm should be able to—given any starting point within the field of the function— find a local or global maximum or minimum within the field of the function. The method which will be used in this project utilizes the function's vector field to trace a path along vector arrows from the starting position to reach a maximum relative to that point.

An ultimate objective of an optimization algorithm, as stated above, is to find the maximum or minimum of a given function; one method of doing so is called *gradient ascent/descent*. Gradient ascent and descent use iteration to follow the vectors of a function over a fixed amount of time to reach that maximum or minimum. At both, the arrows become increasingly smaller, pointing upwards to a maximum until it can get no higher, and downwards to a minimum until it can get no lower:



the vector field for the equation above. the three vertical lines with vectors that are essentially dots represent approaches to maxima and minimum; the maxima are on the outsides and the minimum is in the middle.

In the context of artificial intelligence and machine learning, optimization is important in approximating the function that can find those values, as described in [2]. Using optimization in these cases allows for the creation of a general algorithm that can take a vast number of inputs and even a non-specific function and produce an optimized function to describe them. To use a real-world example, applications of machine learning in the field of astrophysics include generating bias-free models of interactions, reducing and expanding the dimensionality of inputs, and more effectively transferring information (V. Acquaviva, personal communication, February 16, 2022). In each of these cases, different methods of machine learning can be implemented, and for each method, optimization algorithms can help computers learn better how to make models, compress information, and help scientists work with data more effectively.

2 The Algorithm

The iRProp⁻ algorithm is from [1] and is an improvement on another algorithm (RProp) originally described in [3]. It relies on an update of the "weight" of movement during ascent for each step of the algorithm. By doing so, the algorithm can minimize oscillations to converge to the maximum/minimum without making large "jumps" across vectors to do so as well as maximize its step size. Both of these result in a faster convergence during ascent or descent.

The algorithm includes four methods. One is used to initialize the variables *update_signs* and *alphas*, the former of which will help ensure the code is moving in the right direction, the latter will act as the "weight" of movement. Each of these variables is a list containing elements set to an initial value and expanded to the size of whatever input the algorithm is faced with to account for the correct number of dimensions:

```
def __init__(self, init_pos, **kwargs):
    self.update_signs = [0 for i in init_pos]
    self.alphas = [10 for i in init_pos]
```

The second method takes *update_sign* and *alpha* (single numbers from each of the lists above) as well as *grad*, a number from a generated list of the vertical and horizontal vectors at any given point on the function. With these values, it is able to compute whether the signs of *update_sign* and *grad* are the same by multiplying the two together. If the values are in the same direction (their product is greater than zero), then *alpha* is increased to bias the movement more in that direction. If the two are opposite (their product is negative), then *alpha* is decreased and *update_sign* is set to 0 to reset the direction of motion. If their product is 0 (which occurs when *update_signs* has just been set to 0), then the sign of *update_sign* is set to -1, 0, or 1, corresponding to the sign of *grad*, and alpha is kept the same. This method then returns the modified *update_sign* and *alpha* for each step the algorithm runs:

```
def modifier(update_sign, alpha, grad):

  sign_update = update_sign * grad

  if sign_update > 0:
    alpha *= 1.2
  elif sign_update == 0:
    update_sign = np.sign(grad)
  else:
    update_sign = 0
    alpha = alpha * 0.5

  return update_sign, alpha
```

The third method is what communicates with the given function. It vectorizes the previous method, uses it to create a list with the elements *update_sign* at index 0 and *alpha* at index 1, sets the lists *update_signs* and *alphas* equal to their corresponding values, then returns a final list called *updates*, the elements of which are the two values multiplied together:

```
def get_updates(optimizer_status, grads, **kwargs):

  vector_modifier = np.vectorize(modifier)
  vectors = [vector_modifier(update_sign, alpha, grad) for update_sign, alpha, grad in
zip(optimizer_status.update_signs, optimizer_status.alphas, grads)]
  optimizer_status.update_signs = [vector[0] for vector in vectors]
  optimizer_status.alphas = [vector[1] for vector in vectors]

  updates = [update_sign * alpha for update_sign, alpha in zip(optimizer_status.update_signs,
optimizer_status.alphas)]

  return updates
```

The final method is made to return *true* when the function has only made very small improvements for more than a set number of iterations defined in the given code:

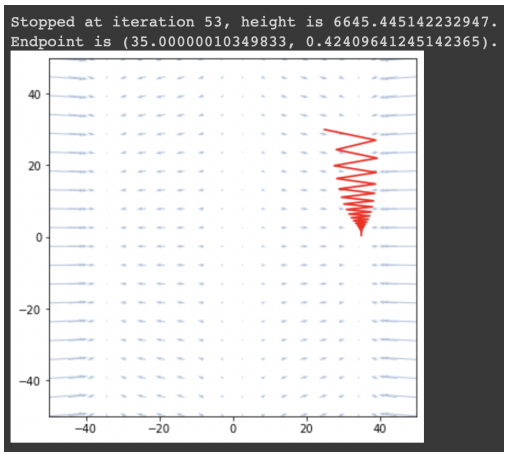
```
def trigger_stopping(optimizer_status, **kwargs):
  return True
```

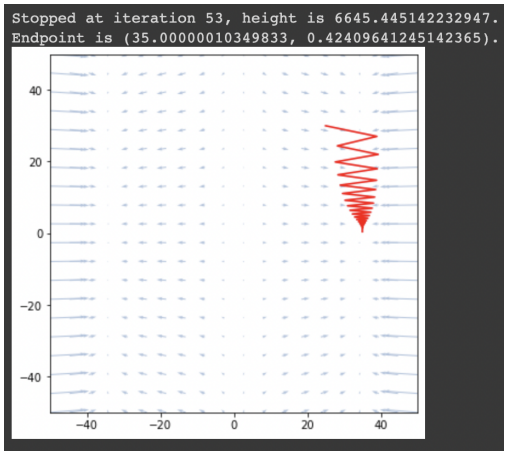
3 Performance & Comparison

In this section, I will compare the performance of the modified algorithm with that of the file *inverse_step_size* for 3 different pre-programmed Jupyter notebooks. I chose *inverse_step_size* because its trigger_stopping method is the same as that of iRProp. In doing so, I hope to eliminate the difference in step size updates that the trigger_stopping method of *adaptize_step_size* would create. Note that for each of the tests in this section, the alpha value used was 2, rather than 10.

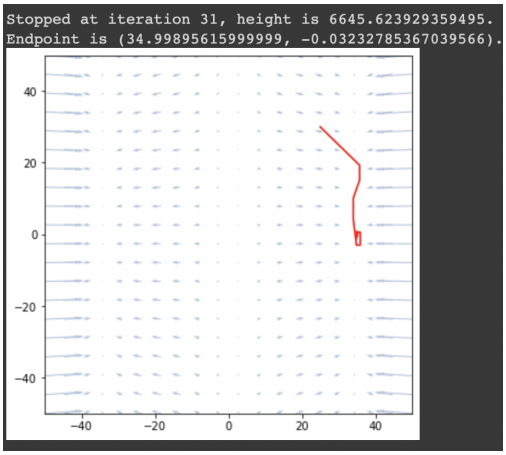
3.1 Simple Tests

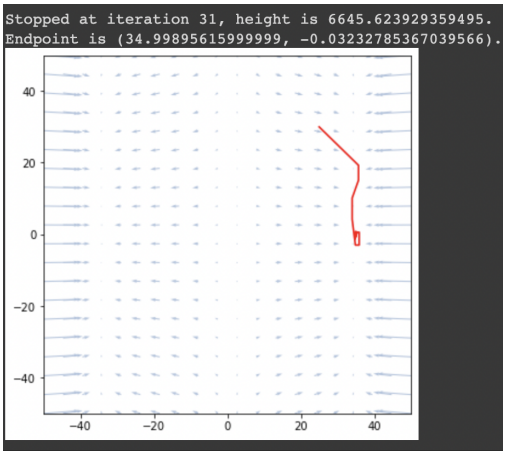
This notebook performs the same test as in the development notebook. These first two images are the two test notebooks run with the *inverse_step_size* algorithm:





while these second two utilize the iRProp algorithm:

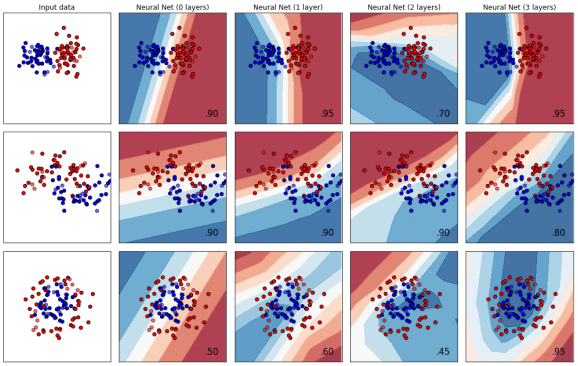




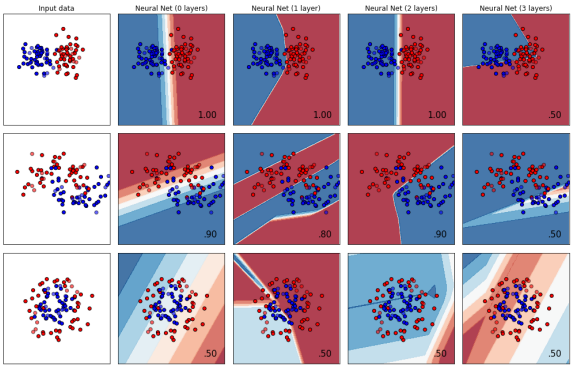
Clearly, iRProp outperforms *inverse_step_size* in both the number of iterations (31 to 53) and the closeness to the maximum (6645.624 to 6645.445).

3.2 Neural Networks

This second notebook begins to experiment with the training of a neural network. The network is expected, given a dataset of 100 points, to "learn" to best predict where the points are distributed. Three different datasets are each fed to four different neural networks to compare which generates the best predictions. This first image is the result of neural networks trained on *inverse_step_size*:



While this second one is the result of the networks being trained on iRProp:



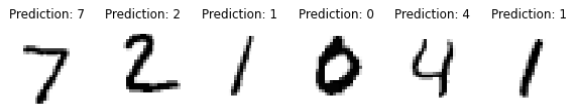
The data seems to be more nuanced here, though one observation is that, for the first and second datasets, the networks seemed more "confident" in their predictions, producing more defined areas with less gradient between them. This was especially successful for the first dataset (all but the 3-layer neural net seem to have approximated the regions very well). For the final dataset, which appears to be the most difficult for a neural network to predict, the *inverse_step_size*-trained networks seem to be more successful, though truly it's hard to say. However, the first images seem to better approximate the curve of the circle, while the images in the second collection seem more "jagged."

3.3 Neural Networks Again

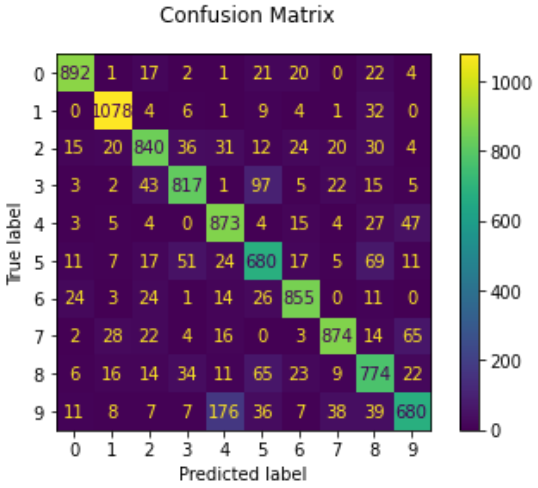
This final neural network is designed to guess which numbers (0-9) handwritten characters represent. The neural network is trained to increase its accuracy over a certain number of iterations. It then guesses the characters, creating a visual output for comparison, and creates a confusion matrix that can further help illuminate its confidence in predicting each character. For the neural network trained on *inverse_step_size*, this took 300 iterations and resulted in a "confidence" of 0.837 (meaning the computer will accurately predict the character about 84% of the time; on random chance, a computer would predict the correct character 10% of the time, as there are 10 digits, 0-9):

```
Iteration 295, loss = 0.752104257329305
Iteration 300, loss = 0.7508502462069194
The accuracy of prediction on the test set is 0.837.
```

It correctly predicted all of the characters:



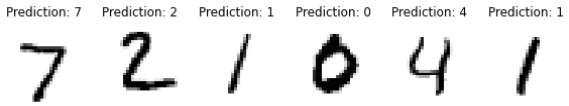
And had its highest confidence in the digits 1, 0, 7, 4, and 6:



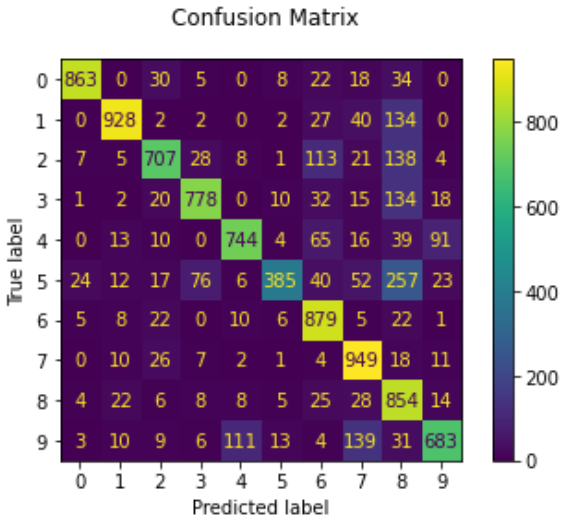
The neural network trained on iRProp seems to have taken only 10 iterations, but had a lower prediction accuracy (though only by about 6%):

```
Iteration 5, loss = 14.100665690104167
Iteration 10, loss = 9.56636238606771
The accuracy of prediction on the test set is 0.7794.
```

It also correctly predicted all of the characters:



And the values for its confusion matrix were generally lower than those of the other neural network, meaning it had lower confidence for some of them. However, its confidence for 7 was its highest, followed by 1, 6, 0, and 8:



There seemed to be more strange “quirks” with this test than with the other two, in that it required much less training time, but resulted in lower accuracy, and that its confidence in predicting certain digits was higher, but for others was considerably low. It would be interesting to explore what exactly makes the training time so low when trained with iRProp in this specific case and see whether training it for longer would make it more accurate.

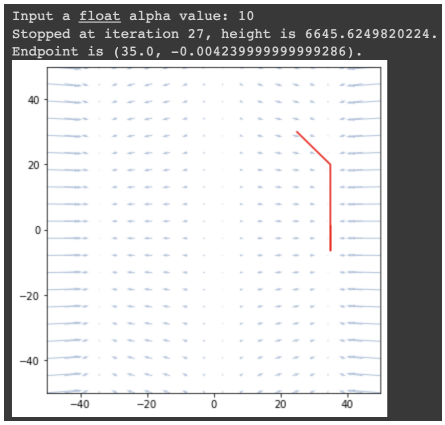
4 Refinements

One refinement I suggest to the program above is to allow more experimentation with the alpha value. This will allow the algorithm to find which *initial* alpha value most quickly converges to the desired minimum or maximum. I began implementing this by allowing the user to input the initial alpha value for the first test:

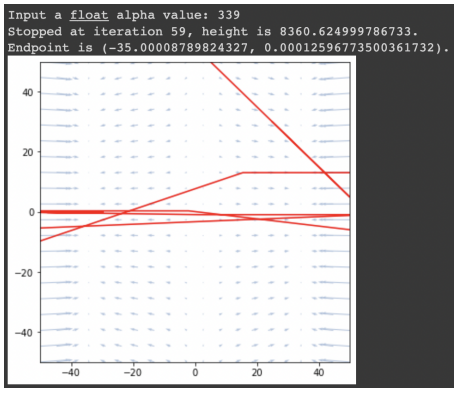
```
alpha_input = input("Input a " + "\u0332".join("float ") + "alpha value:
self.alphas = [float(alpha_input) for i in init_pos]
```

prompts user with "Input a float alpha value:" upon running the test code. This float value will then be used as the initial alpha.

Another way this experiment could be extended is by taking a sampling of different alpha values and seeing which of them converge most quickly to the absolute minimum. I found that inputting 10 makes the function converge very quickly:



Interestingly, inputs greater than 338 make the function converge to the absolute maximum on *the other side of the saddle*, though with much more movement:



By taking a sampling of alpha values, the function may be able to more quickly reach each of the two maxima. One caveat of this is that the alpha values may be arbitrary, as it's hard to know how large or small the range to test should be. Moving forward, the method of using a sampling of alpha values could also be implemented in each of the other test notebooks to continue to improve the tests in each notebook.

References

[1] C. Igel and Michael Hüsken. Improving the Rprop learning algorithm. In Hans-Heinrich Bothe and Raúl Rojas, editors, *Proceedings of the second ICSC Symposium on Neural Computation: May 23-26, 2000, Berlin, Germany*, pages 115–121. ICSC Academic Press, International Computer Science Conventions, Canada / Switzerland, 2000.

[2] Brownlee, J. (2021, October 11). *Why Optimization is Important in Machine Learning*. Machine Learning Mastery. Retrieved March 9, 2022, from https://machinelearningmastery.com/why-optimization-is-important-in-machine-learning/

[3] Riedmiller, M.A., & Braun, H. (1993). A direct adaptive method for faster backpropagation learning: the RPROP algorithm. *IEEE International Conference on Neural Networks*, 586-591 vol.1.