Mathematical Techniques for Tuning the Hyperparameters of Recurrent Neural Networks

Laker Newhouse

Abstract

We compare algorithms for minimizing continuous, multivariable functions in the context of optimizing hyperparameters for recurrent neural networks. The algorithms tested are gradient descent (first order), Newton's method (second order), and Quasi-Newton methods (second order). Our discrete objective function is the cross-entropy loss of the confidence of an RNN replicating the complete works of Shakespeare after α steps, averaged over β iterations. α and β are constant for any specific computation but vary throughout the study. We found that Gradient Descent performs best, though not well; the second order methods are unstable because the objective function is ill-defined with a small β. This work translates into models that learn more with fewer computational resources. For mathematics, this confirms that second order methods generally perform poorly on ill-defined functions.

Table of Contents

Background

Despite its explosive success in recent years, machine learning is a nascent field. The concept of a neural network itself is only about fifty years old, and it has been used effectively for less than a decade. There is still, therefore, much left to learn about the mechanics and interactions of neural networks under different types of data, different architectures, and, the focus of our study, different hyperparameters.

Introduction

The class of neural network studied in our investigation is the recurrent neural network, or RNN. The architecture of RNNs factor in the previous output as input, giving it a kind of memory. RNNs are, because of this, well suited to predicting sequences of data like text and numbers. In our study, we vary the hyperparameters of RNNs learning to generate Shakespeare to find the optimal configuration for fast learning.

Hyperparameters are arbitrary numbers that control how neural networks learn. The simplest example of a hyperparameter is the learning rate, a constant multiplied into the gradient of the cost function before applying it to the weights. We also use two other hyperparameters: the size of the hidden layer and how far back the RNN sees (sequence length).

Our investigation compares the efficacy of three optimization algorithms when applied to tuning the hyperparameters of our RNNs. The ideal set of hyperparameters, what these algorithms are searching for, is one which makes the RNN learn to write Shakespeare well after less training than another RNN learning under random hyperparameters. The algorithms we tested are Gradient Descent, Newton's method, and Quasi-Newton methods.

Procedure:

We applied Gradient Descent, Newton's method, and Quasi-Newton methods to minimize our objective function, the loss averaged β times of an RNN with randomly initialized weights trained on Shakespeare for α steps. The RNNs had three hyperparameters which the algorithms could change: the number of hidden neurons, the number of characters back it could see (previous hidden layer states), and the constant multiplied into the negative gradient during training, the learning rate. The algorithms could not, for example, deviate the number of hidden layers (1), the training method used (Adagrad), or the loss function (cross-entropy loss of confidence). Future investigation may uncover the effects of these untested hyperparameters.

Two of the hyperparameters tested were discrete, the number of hidden neurons (henceforth referred to as hidden size) and the number of characters back the RNN could see (henceforth referred to as sequence length). To work around this issue, we defined a new continuous objective function as the weighted average of the losses given each discrete parameter rounded up and down. An example will illustrate best.

Let the algorithm request the loss if the hidden size is 40.25, the sequence length is 12.8, and the learning rate is 0.1. The hidden size of 40.25 can be thought of as ¾ of the loss given hidden size 40 and ¼ of the loss given hidden size 41. The sequence length of 12.8, similarly, can be thought of as ⅕ of the loss given sequence length 12 and ⅘ of the loss given sequence length 13. The learning rate is continuous and plugged in directly. Naming the discrete objective function referenced above $f(h, s, l)$, the continuous objective function can then return

$$0.75 \cdot (0.2 \cdot f(40, 12, 0.1) + 0.8 \cdot f(40, 13, 0.1)) + 0.25 \cdot (0.2 \cdot f(41, 12, 0.1) + 0.8 \cdot f(41, 13, 0.1)) .$$

Upon the continuous objective function, we layered another condition. If any requested hyperparameter in the objective function is verifiably absurd, return a high and undesirable number (10^6) to discourage further exploration. To this undesirable number, we add an additional number representing how far the hyperparameter is from its reasonable range: $(h - 410)^2$ for hidden size, $(s - 25)^2$ for sequence length, and $(l - 0.5)^2$ for learning rate. This additional term is quadratic so second order methods can detect it.

The range for each hyperparameter is as follows. Hidden size must be between 20 and 800; sequence length must be between 5 and 40; and learning rate must be between 0.0001 and 1. These ranges were chosen roughly such that any RNN with a hyperparameter outside of its range performs badly. Thus, around the plains of minimization lie quadratic arches of undesirability meant to direct the search back to sanity.

We ran a number of trials for each optimization algorithm. Each algorithm initialized the hyperparameters to a random number between 0 and 10 and terminated the search if the magnitude of the gradient at its current guess was less than 0.0001 (though this never happened). The arbitrary multivariable function they were given to minimize was the continuous objective function defined above.

For Gradient Descent, we ran five high-level tests. Each high-level test included three low-level tests, which varied the step size. Each low-level test optimized for 10000 steps and used a different constant to multiply into the negative normalized gradient (step size). The first used a step size of 0.1, the second 0.01, and the third 0.5.

The first three high-level tests of Gradient Descent used the continuous objective function given $\alpha = 100$ and $\beta = 1$ for the discrete objective function. The fourth high-level test used $\alpha = 10$ and $\beta = 1$. The fifth high-level test used $\alpha = 1$ and $\beta = 1$ and reported its guess every 100 steps, along with a more intensive calculation of the gradient at the guess every 1000 steps ($\alpha = 1000$ and $\beta = 10$). Although the more accurate gradient wasn't applied in the algorithm, it was a useful proxy for the success of the less accurate gradients the algorithm used at each step.

For Newton's method, we ran two identical tests with $\alpha = 10$ and $\beta = 1$, recording predictions every 10 steps. To avoid non-invertible matrix errors, prior to using the inverse Hessian, the determinant was calculated. If it equaled 0, Newton's method reverted to Gradient Descent. The step size used was 10 in an effort to get the hyperparameters back to sanity, where the Hessian would be invertible, as quickly as possible.

If the determinant was nonzero, the Hessian invertible, then the standard Newton's method was used. The step was calculated as the negative inverse Hessian times the gradient. As Newton's method generally predicts step size accurately, its steps weren't scaled.

For Quasi-Newton methods, we ran two tests, one per inverse Hessian update formula, recording predictions every step. Each test used the objective function defined by $\alpha = 10$ and $\beta = 10$. The first test used the BFGS update formula; the second test used the DFP update formula.

As in Newton's method, the step at every iteration was defined to equal the negative inverse Hessian times the gradient. Quasi-Newton methods, however, only track an approximation for the inverse Hessian, so the step is the negative of this approximate inverse Hessian times the gradient. Unlike Newton's method, though, the magnitude of the calculated steps isn't trusted. We therefore used a backtracking line search to find the optimal magnitude for the calculated step to minimize the objective function. We used a lessening factor of 0.5 and a control of 0.99. With the notation of Armijo's original paper, we used c = 0.99 and $\tau = 0.5$.

The addition of a backup Gradient Descent was also kept from Newton's Method, though the scaling factor was changed from 10 to 1. Lacking the Hessian, however, the 0 determinant heuristic for being stuck couldn't be used. Instead, we observed that absurd hyperparameters caused the Quasi-Newton methods to freeze, not moving at all. Therefore, the Quasi-Newton heuristic we used for reverting to Gradient Descent was if the magnitude of the algorithm's proposed step was smaller than 10^{-6} or if any hyperparameters were measurably absurd (hidden size < 20 or > 800, sequence length < 5 or > 40, learning rate < 10^{-6} or > 1).

The gradient was numerically approximated with the limit definition of partial derivatives and $h = 10^{-8}$. The Hessian was numerically approximated with a recursive implementation of the limit definition of mixed partial derivatives. See the code in section 9, **Code**.

Results:

Hyperparameters were converted into three dimensional coordinates so the algorithms could use them. The first dimension is the hidden size, the second is the sequence length, and the third is the learning rate. All coordinates are rounded to either four or six digits of accuracy depending on the required precision. Results are sectioned under their corresponding algorithm.

Gradient Descent:

Trials 1 - 3 ($\alpha = 100$, $\beta = 1$, 10000 iterations) (Fig 1A)

Step Size	Trial 1	Trial 2	Trial 3
0.1	[24.7835, 7.3241, 0.9308]	[20.6382, 7.6258, 0.7576]	(not recorded)
0.01	[20.2441, 9.8831, 0.3094]	[19.9035, 9.0293, 0.9868]	(not recorded)
0.5	[36.7481, 35.9527, 0.2646]	[66.4636, 39.7850, 0.9208]	[74.5392, 18.9831, 0.2774]

Trial 4 ($\alpha = 10$, $\beta = 1$): (Fig 1B)

Step Size	Trial 4
0.1	[20.6689, 5.4620, 0.3758]
0.01	[20.0701, 5.2175, 0.7085]
0.5	[28.0586, 27.1379, 0.7804]

Trial 5 ($\alpha = 1$, $\beta = 1$): (Fig 1C)

Step Size	Trial 5
0.1	[22.4090, 11.7156, 0.2462]

0.01	[20.3657, 5.0541, 0.8160]
0.5	[31.4826, 32.9537, 1.1683]

Intensive gradients and their magnitudes (α = 100 and β = 100) were calculated for each result of Fig 1A. For the reader's convenience, all components are multiplied by 10^{-8}, the value used for h in the limit definition of the partial derivative. Lower gradient magnitudes mean the guess is better, because the gradient serves as a proxy for how close the final guesses are to a minimum. The reason all gradient magnitudes are higher than might be expected is discussed in section 6, **Analysis**, under Gradient Descent.

Intensive Gradients (α = 100, β = 100, divided by 10^8 for convenience): (Fig 1D)

Step Size	Trial 1 Gradients	Trial 2 Gradients	Trial 3 Gradients
0.1	[-0.66361387, 9.48057384, -2.37193118]	[-3.44347236, 0.16334927, 2.35988607]	N/A
0.01	[-0.30589072, 0.15317452, 0.08640280]	[-4.79179805, -4.06369183, -2.06258140]	N/A
0.5	[-1.20603242, -0.28046133, -11.54847267]	[-623.41250076, 569.28816667, 257.37610905]	[32.55462667, -10.97172678, -24.75223905]

(Fig 1E)

Step Size	Trial 1 Magnitudes	Trial 2 Magnitudes	Trial 3 Magnitudes
0.1	9.79529077227	4.1777083648	N/A
0.01	0.352841351	6.61280288613	N/A
0.5	11.6146628823	882.595391071	42.3421284812

Every 100 steps, trial 5 recorded its guess. The following are graphs of the predictions over time of trial 5. Each datum represents 100 steps of optimization after the previous, except on multiples of 100 when the test restarts with a new step size (0.1, then 0.01, then 0.5). First (Fig 2A) is all three hyperparameters overlaid. Second (Fig 2B), third (Fig 2C), and fourth (Fig 2D) are graphs of each hyperparameter separately.

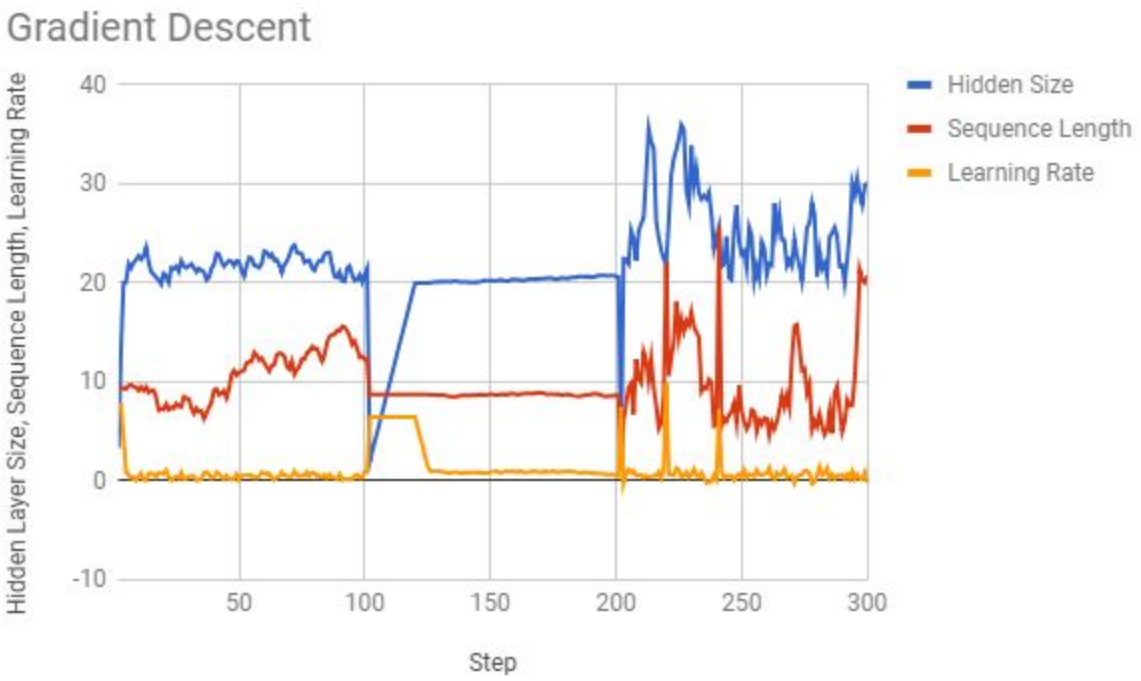

Fig 2A: all Gradient Descent hyperparameters

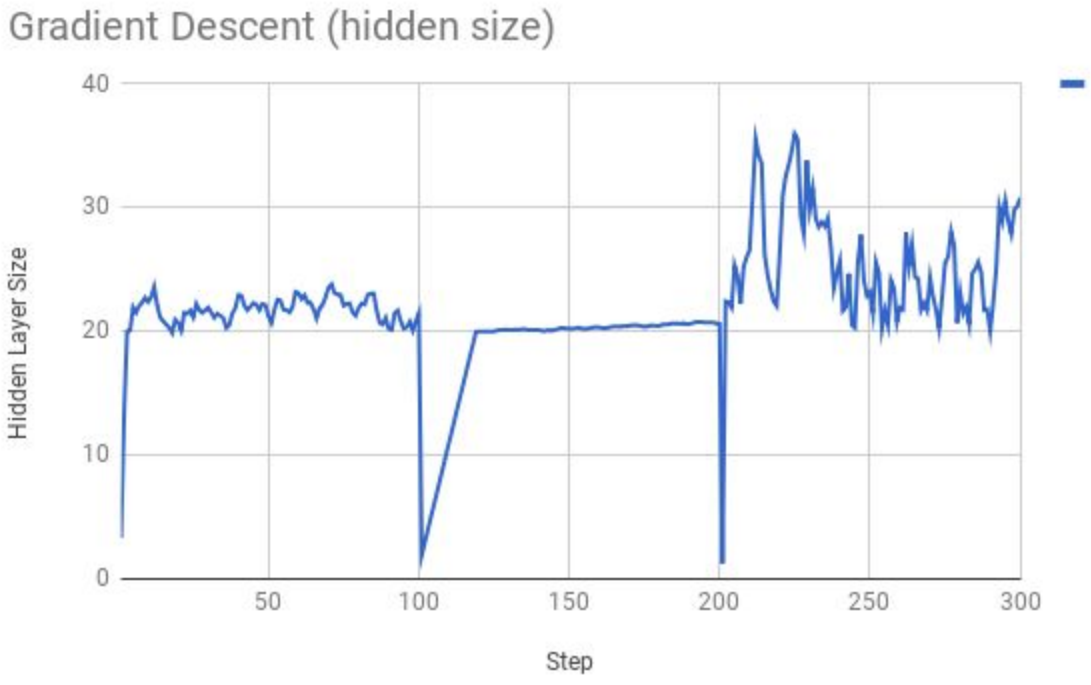

Fig 2B: hidden size over time

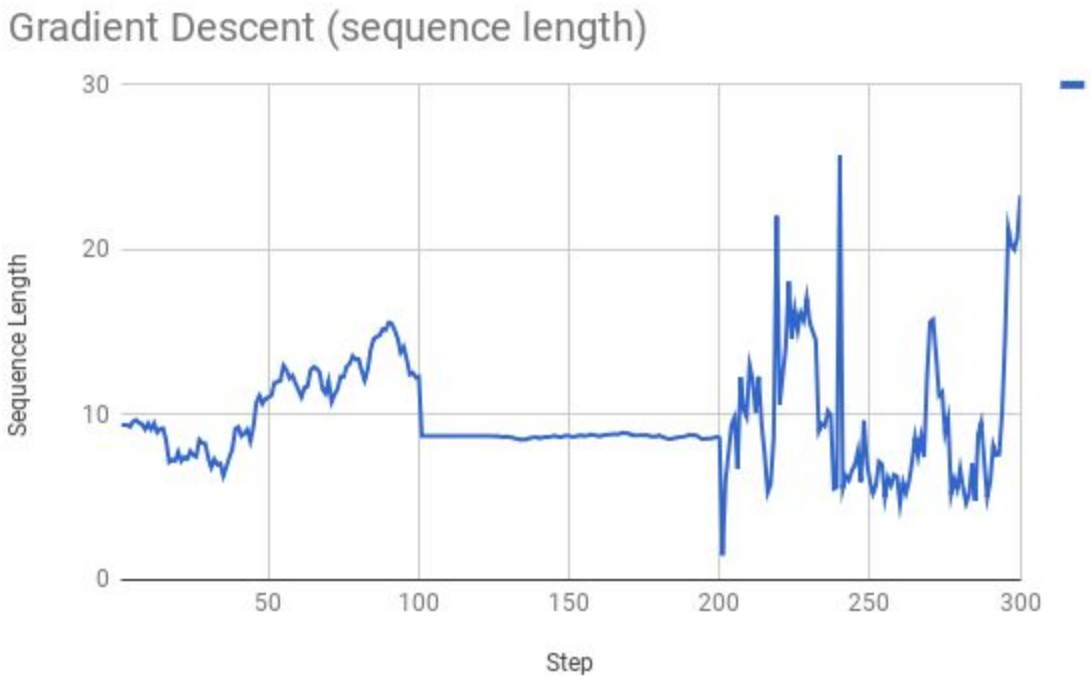

Fig 2C: sequence length over time

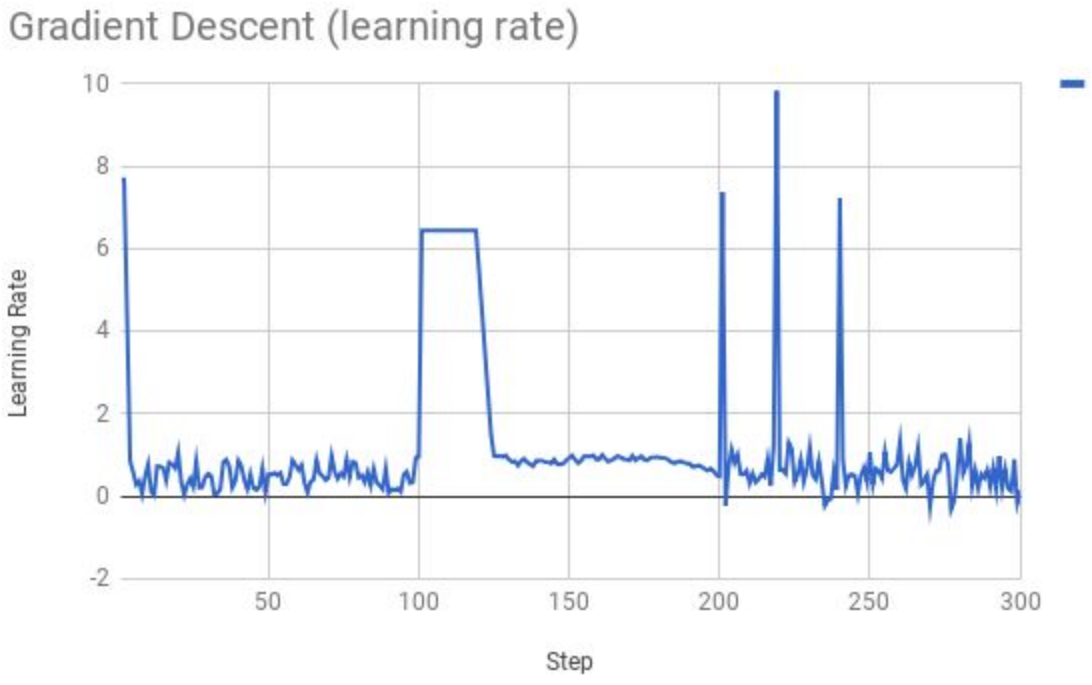

Fig 3D: learning rate over time

Newton's Method:

Trial 6 (α = 10, β = 1, step multiplier = 1): (Fig 3A)

Step	Trial 6
0	[2.56**3309**, 9.97**5360**, 7.96**5497**]
10	[12.56**1629**, 9.97**5360**, 7.78**2208**]
20	[20.56**1268**, 9.97**5360**, 5.65**3969**]
30	[20.56**4417**, 9.97**5330**, 0.65**4158**]
40	[20.56**7070**, 9.97**6808**, 0.65**4354**]
50	[20.56**6863**, 9.97**6869**, 0.65**4446**]
60	[20.56**6834**, 9.97**6864**, 0.65**4429**]
70	[20.56**6844**, 9.97**6744**, 0.65**4288**]
80	[20.56**6858**, 9.97**7455**, 0.65**4483**]
90	[20.56**6887**, 9.97**7575**, 0.65**4435**]

As is clear from the boldface final digits of trial 6, Newton's method is taking very small steps. The reason for this is discussed in section 6, **Analysis**, under Newton's Method. Regardless, we consequently ran a trial in which the gradient descent backup is the same but each Newton's method step is scaled by a factor of 1000.

Trial 7 (α = 10, β = 1, step multiplier = 1000): (Fig 3B)

Step	Trial 7
0	[7.4375, 5.3365, 7.5200]
10	[17.4360, 5.3365, 7.3455]
20	[22.8466, 5.3394, 0.9031]
30	[23.3507, 5.6824, 0.7586]
40	[23.2082, 5.6415, 0.8135]
50	[22.9045, 5.7803, 0.7647]

60	[22.7354, 5.8058, 0.7951]
70	[22.3963, 5.7339, 0.3012]
80	[22.3760, 5.7867, 0.3734]
90	[22.3629, 5.7534, 0.3744]

Intensive Gradients (α = 100, β = 100, divided by 10^8 for convenience): (Fig 3C)

Trial 6 Gradient	Trial 7 Gradient
[-2.01919592, 0.66754092, -2.96560209]	[1.07868134, -0.52292994, -0.10280518]

(Fig 3D)

Trial 6 Magnitude	Trial 7 Magnitude
3.64932306	1.20315338

Every 10 steps, trial 7 recorded its guess. Fig 3E is all three hyperparameters overlaid. Each datum represents 10 steps of optimization after the previous, except on multiples of 100 when the test restarts.

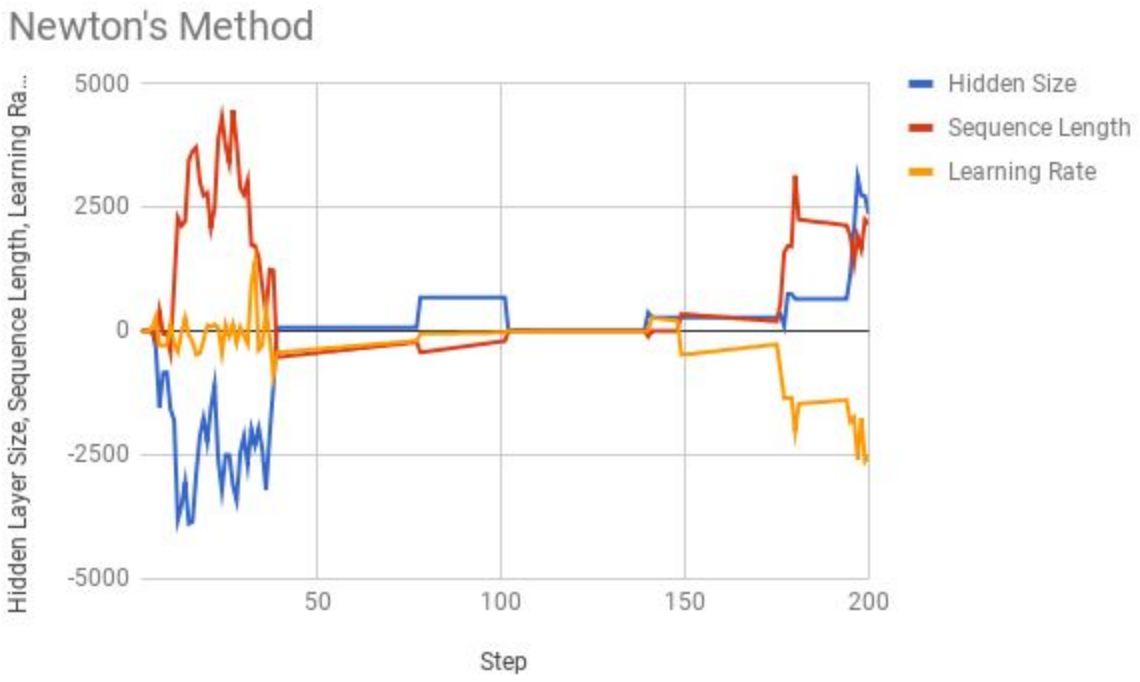

Fig 3E: all Newton's Method hyperparameters

Quasi-Newton Methods:

Trial 8 (α = 10, β = 10, BFGS): (Fig 4A)

Step	Trial 6
0	[2.681075, 8.746468, 6.987274]
10	[11.679926, 8.746468, 6.84345]
20	[20.678913, 8.746468, 5.716134]
30	[46.401594, 22.644959, -4.414161]
40	[92.842201, -159.625745, -148.2645]
50	[92.842281, -151.839258, -141.989903]
60	[92.842529, -144.053257, -135.714704]
70	[92.842619, -136.266045, -129.441009]
80	[92.842619, -128.478271, -123.168012]
90	[92.842619, -120.690901, -116.894512]

Trial 9 (α = 10, β = 10, DFP): (Fig 4B)

Step	Trial 6
0	[9.592970, 5.979446, 1.928359]
10	[19.592908, 5.979446, 1.893177]
20	[43.503087, -1.290651, -0.430466]
30	[66.303606, 40.219715, 11.952225]
40	[66.303606, 39.219715, 2.952202]
50	[-27.087827, -220.533407, 966.260186]
60	[-23.070797 -218.277099, 957.384797]
70	[-19.053640, -216.020328, 948.509583]
80	[-15.036408, -213.763940, 939.634306]
90	[-15.036408, -213.763940, 939.634306]

Intensive Gradients (α = 100, β = 100, divided by 10^8 for convenience): (Fig 4C)

Trial 8 Gradient	Trial 9 Gradient
[0.0, -0.0002910, -0.00023433]	[-0.00085091, -0.00047695, 0.00187790]

(Fig 4D)

Trial 8 Magnitude	Trial 9 Magnitude
0.00037363.8401447	0.00211614

Graphing every step of the Quasi-Newton method for BFGS and DFP gives the following.

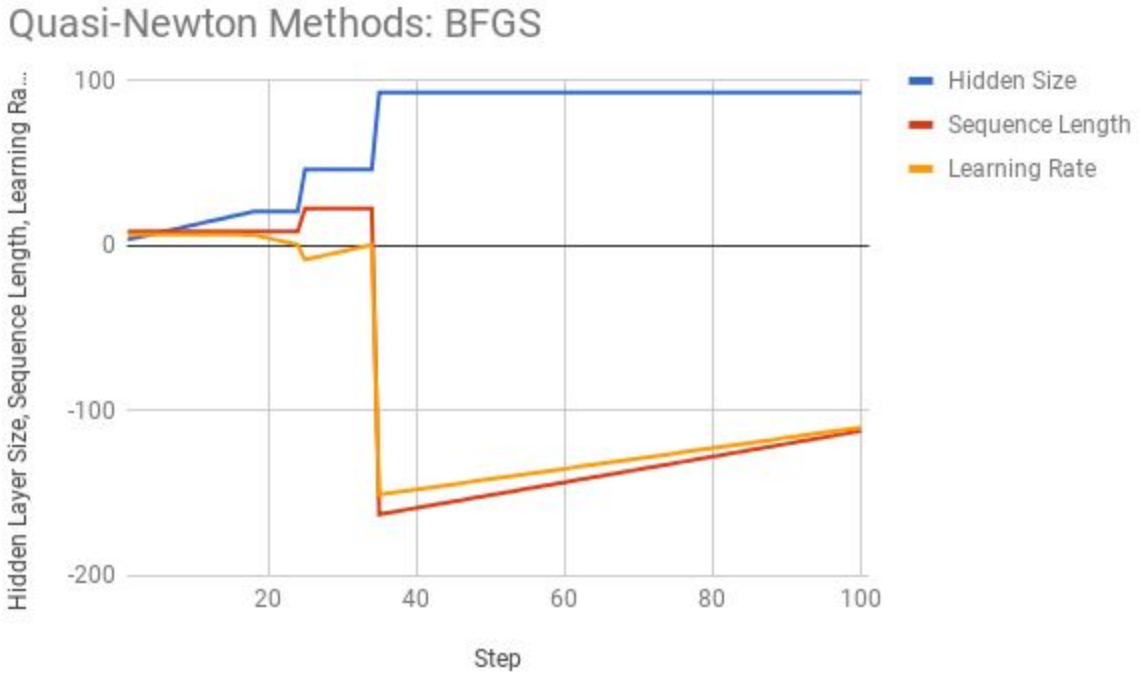

Fig 4E: all Quasi-Newton (BFGS) hyperparameters overlaid

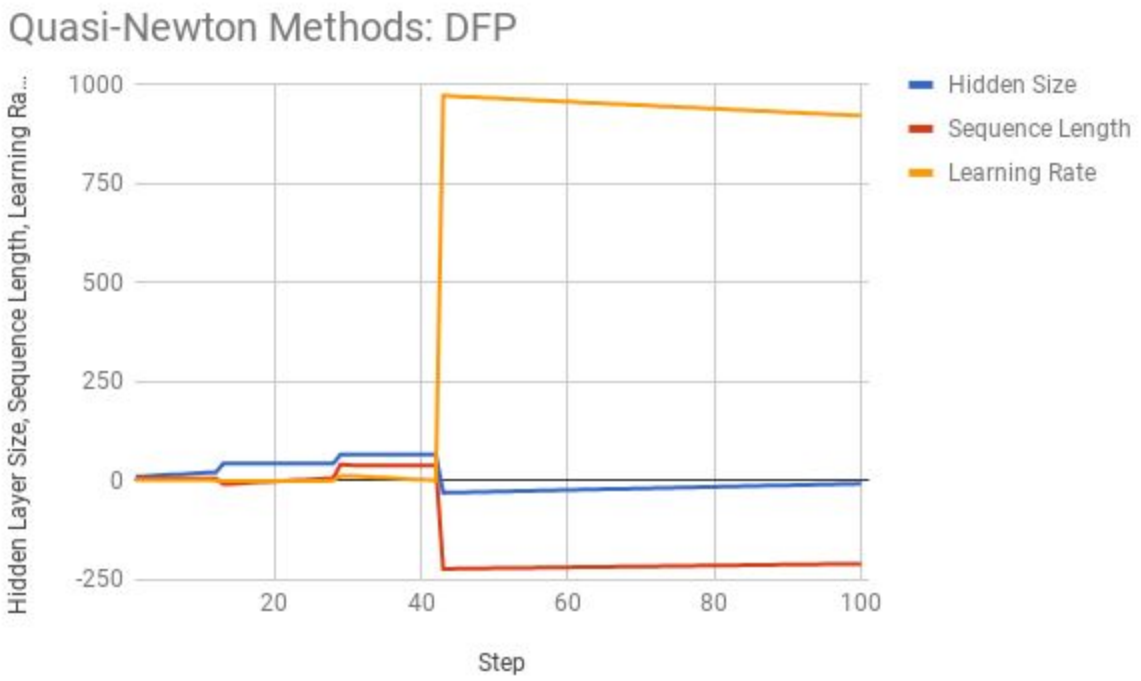

Fig 4F: all Quasi-Newton (DFP) hyperparameters overlaid

Analysis:

Gradient Descent

Gradient Descent was the most extensively studied algorithm and was also the most stable. Its outputs suffered most from the lack of computation available. With α = 100 and β =1, running a single trial with step sizes 0.1, 0.01, and 0.5 took over 40 hours on our machines.

As discussed in **issues**, the continuous objective function is ill-defined for β as a finite number. Although each algorithm was hurt by inconsistent derivatives this caused, Gradient Descent fared best. This is probably because it is a first order approximation. An error in the gradient, therefore, would stay as it is, while an error in a second order approximation would get squared.

Though not perfect, the results of gradient descent can be tested. Note the intensive gradients of Fig 1D (calculated with α = 100 and β = 100) and their magnitudes from Fig 1E. A random gradient would have a magnitude close to 10^8, as derivatives were calculated with $h = 10^{-8}$ and the objective function is ill-defined with respect to small changes in inputs ($\lim_{h \to 0} \frac{c}{h} = \infty$). The change in the objective function, then, when divided by h, would be

artificially large. Fig 1E shows that gradients generally had a magnitude around 10^8, some greater, some lesser.

It can therefore be inferred that Gradient Descent is a valid optimization method for hyperparameters but is inefficient. This returns to the issue of an ill-defined objective function; for large β s, the algorithm would likely not become confused and oscillate as it did in Fig 2A. Additionally, a small α diminishes the objective function's ability to gauge the long-term benefits of a higher hidden size or lower learning rate. This may be why, for example, the gradient of trial 3's 0.5 step size was higher than trial 1's 0.01 step size. Even with α = 100 and β = 100, the objective function may have failed to detect the advantages of more hidden neurons.

For the astute observer, the short plateau in learning rate directly following the resets every 100 steps was caused by an extreme difference in the perceived importance of minimizing learning rate versus minimizing sequence length and especially hidden size. Recall the sanity check from section 4, **Procedure**. When hyperparameters are absurd, the sum of the squares of the differences between the absurd values and accepted reasonable values plus a high and undesirable number is returned as the loss. Because the hidden size and sequence length must be optimized on a significantly larger scale than learning rate, from the normalized gradient's perspective, optimizing learning rate is of less importance until hidden size and sequence length are within range. This is why the plateau ends once the hidden size and sequence length are within reasonable range.

Newton's Method

Newton's Method wasn't effective. It took steps that were either tiny or huge, never practical. It spent the majority of its thousand steps in Fig 3A, for example, using Gradient Descent to pick itself out of the early miscalculation that made the hidden size negative (around step 5). We theorize that these two modes, tiny steps and huge steps, are a result of the ill-defined objective confusing the second order approximations. Specifically, the mixed partial derivatives of the Hessian are either lined up into an artificially steep paraboloid or into a flat paraboloid.

When the mixed partial derivatives align to a flat paraboloid, a huge step will be taken. The virtue of second order approximations is that, given a paraboloid, unlike first order approximations, they will jump to its critical point in a single step. In the case of a nearly flat paraboloid, the minimum will be very far from the starting point. In minimizing this flat paraboloid, Newton's method will bring its hyperparameters far out of the reasonable bounds. Conversely, when the mixed partial derivatives align to a steep paraboloid, a tiny step will be taken. This is because steep paraboloids have minima close to the starting point.

We answer why the majority of steps are tiny with this explanation. Given an ill-defined objective function (small β), numerically approximated derivatives are mostly random. Additionally, the approximated derivatives will be artificially large. (See Gradient Descent from **Analysis** for why derivatives are overestimated.) Therefore, with nine elements in a Hessian of three dimensions, the chance of steep alignment far outweighs the chance of flat alignment. This translates to a much higher chance for tiny steps than huge steps.

Quasi-Newton Methods

The Quasi-Newton methods performed the worst of the trio tested, undistinguished by update formula. They weren't capable of keeping the hyperparameters within the reasonable range. Once its Gradient Descent mode had taken the hyperparameters out of absurdity, like Newton's method, the Quasi-Newton methods took unreasonably long steps. Unlike Newton's method, however, the Quasi-Newton methods never took small steps. We have two theories on the matter.

The first involves the backtracking line search. Using the Armijo-Goldstein condition, the condition which determines whether a proposed step length is satisfactory, (see section 8, **Glossary**) requires assuming that the direction given for the line search points downward. This cannot be guaranteed with a low β's ill-defined objective function. The direction which minimizes loss for one RNN initialized with random weights may maximize the loss for another in the short term. As a positive slope in the line search is equivalent to requesting a maximizing step, this is a possible cause for the long steps the Quasi-Newton methods took.

The second theory is that the ill-defined objective function led to incorrect and large approximate inverse Hessians. The failure to predict accurate inverse Hessians would cause mostly random steps, as observed in Fig 4E and Fig 4F. Additionally, the huge gradients (see Gradient Descent from **Analysis** for why derivatives are overestimated) would cause huge inverse Hessians. Generally, a matrix with high values will have an inverse with low values, implying that a large approximate inverse Hessian would translate to a small approximate Hessian. Small Hessians, in turn, imply nearly flat paraboloids. As discussed in Newton's method from **Analysis**, second order approximations attempt to minimize the paraboloid defined by the Hessian in a single step. When the paraboloid defined is nearly flat, as would be the case in the Quasi-Newton methods, the step taken to minimize will be long.

Issues

With infinite computational power, the number of averaging steps, β, could be arbitrarily high. While the limit of $f(h, s, l)$ as β approaches positive infinity is a

well-defined function, at any finite whole number, the function is ill-defined. This is inherent to the random initialization of the RNNs. A certain random configuration may be better suited to learning Shakespeare than another. This variation is mostly independent to small changes in hyperparameters, meaning that numerical approximations to derivatives of the objective function are plagued by both inaccuracy and artificial magnitude.

As such, it is expected when calculated gradients seem random. The hope is twofold: either, with a low α and β and many short steps, the computer will stagger its way to an optimum, the net effect of the random initializations cancelling over time; or, with a high α and β and a few calculated steps, the computer will converge to an optimum, cancelling the random initializations in the averaging step.

Due to severe limitations in computer power, however, α and β were kept low. β, for this reason, was unused outside of the Quasi-Newton methods and calculating more accurate gradients. Further experimentation with a high α and β may fill the void left by our investigation.

Conclusion:

The most viable algorithm we tested was Gradient Descent. Newton's method and the Quasi-Newton methods, being second order approximations, dealt poorly with the uncertainty of an ill-defined objective function. Newton's method took steps either much too short or much too long. Regardless of the update formula used, the Quasi-Newton methods only took steps too long.

Still, Gradient Descent is not an ideal solution. 10000 steps of its tuning with various step sizes failed to converge to a measurable optimum. Its oscillation, as seen in Fig 2A, may be due to the ill-defined objective function; a high β might reduce the variability in loss that plagued our calculations. The low α and β is the central shortcoming of our study.

Unfortunately, raising α and β would require more computer power. Each evaluation of the continuous objective function runs $4\alpha\beta$ training steps. The cloud is a good option in this respect: it's cheap and powerful. Future experimentation in optimizing hyperparameters with continuous algorithms will need to explore these options.

Alternatives include discrete optimization methods. A grid search, analogous to a binary search but in multiple dimensions, is a good example. The search space (sane hyperparameters) is divided in half once per dimension. The function is evaluated at each box's vertices, and the most promising box becomes the new space. The method continues as such, exploring the most promising solution at each step. Grid search is powerful in low dimensions and would take a calculable amount of time to reach a certain accuracy. Due to

time constraints, we didn't test grid search. It may, however, prove to be a fruitful avenue for the aspiring hyperparameter tuner.

Glossary:

Neural Network: a set of layers of artificial neurons which each hold a number and connect to all neurons in the next layer with unique weights (connection strength). Neural networks have two or more layers: the input layer, the hidden layers (optional), and the output layer.

RNN: recurrent neural network; neural networks that are fed the state of their hidden layer(s) from the previous use as input. RNNs function with a sort of memory.

Hyperparameters: variables that control how a given neural network functions and learns. The most basic hyperparameter is the learning rate, a scalar multiplied into the negative gradient before it is applied to the weights.

Architecture: the structure of layers and the properties of recurrence in a neural network.

Gradient: the direction of fastest ascent in a multivariable function, used in machine learning to optimize neural network weights; an n long column matrix of a function's partial derivatives evaluated at the input.

Gradient Descent: a first order minimization algorithm for continuous and differentiable functions that, at every step, moves to the previous location minus a constant times the gradient of the function at that location.

Newton's Method: a second order minimization algorithm that subtracts the inverse Hessian of the function at the current location times the gradient of the function at the current location from the guess at each step.

Quasi-Newton Methods: a family of second order minimization algorithms that aim to capture the speed of Newton's Method without the inefficiency of both calculating and inverting the Hessian. Quasi-Newton Methods approximate the inverse Hessian at every step with varying update formulas based on previous first order information. The magnitude of the step taken in the direction of the approximate inverse Hessian times the gradient is determined by a line search.

Line Search: an algorithm which determines the optimal magnitude for a step in a continuous function which minimizes the function in the step's direction. Line searches are usually not exhaustive, however, because any unnecessary computational resources spent on them could be more effectively diverted to improving the accuracy of the direction. The backtracking line search is a simple line search algorithm.

Backtracking Line Search: an algorithm which limits the absolute minimization of an ideal line search to a finite number of integer choices. Integers, k, between 0 and n are tested to determine whether $x + p\tau^k$ minimizes to a satisfactory extent, where x is the original location, p is the direction vector, and τ is the lessening factor between 0 and 1. The satisfactory extent is defined by the Armijo-Goldstein condition, which states that, for a chosen control factor between 0 and 1 called c and the gradient of the function at x called g, a point is satisfactory if $f(x + p\tau^k) \leq f(x) + pgc\tau^k$. Any large enough k and thus small enough τ^k will satisfy this condition, so the backtracking line search, earning its name, begins with k at 0 and increments it until the Armijo-Goldstein condition is met or k > n.

Code:

Selected excerpts of the code used are provided below. The complete code may be accessed at github.com/Arongil/Optimization. All code is written in Python 3.5.

Mixed Partial Derivative (recursive implementation, multivariable_calculus.py):

```
def mixedPartialDerivative(f, v, order):
    # order is an array of dimensions by which to take partial derivatives.
    # order = [0, 1]    ==> partial of f with respect to x then y.
    # order = [1, 2, 2] ==> partial of f with respect to y then z then z.
    h = 1e-6 # The limiting variable in the limit definition of the partial derivative.
    if len(order) == 1:
        # If order is 1, numerically approximate the partial derivative as usual.
        step = v[:]
        step[order[0]] += h
        return (f(step) - f(v)) / h
    # Peel the onion of layers: first, the outermost, then work inwards.
    step = v[:]
    step[order[len(order) - 1]] += h # Final derivative step done first.
    # Recursively calculate partial derivatives for the final derivative.
    return (mixedPartialDerivative(f, step, order[0:len(order) - 1]) - mixedPartialDerivative(f, v, order[0:len(order) - 1])) / h
```

Gradient (multivariable_calculus.py):

```
# gradient computes the gradient of a function of n dimensions, f, at position v.
def gradient(f, v):
    grad = [0 for i in range(len(v))] # This is the gradient array of partial derivatives.
    h = 1e-12 # The limiting variable in the limit definition of the partial derivative is approximated with 1e-8.
    for i in range(len(v)):
        step = v[:]
        step[i] += h
        grad[i] = (f(step) - f(v)) / h # limit definition of the partial derivative
    return grad
```

Continuous Objective Function (tuner.py):

```
def continuousNetworkLoss(v):
  # To make the discrete inputs of hidden_size and seq_length continuous for the loss function,
  # take their weighted averages between the two closest integers.
  lowHiddenWeight  = 1 - (v[0] - np.floor(v[0]))
  highHiddenWeight = v[0] - np.floor(v[0])
  lowSeqWeight  = 1 - (v[1] - np.floor(v[1]))
  highSeqWeight = v[1] - np.floor(v[1])
  discreteLowHidden  = lowHiddenWeight * (lowSeqWeight*networkLoss([int(np.floor(v[0])), int(np.floor(v[1])), v[2]]) +
                                          highSeqWeight*networkLoss([int(np.floor(v[0])), int(np.floor(v[1])) + 1, v[2]]))
  discreteHighHidden = highHiddenWeight * (lowSeqWeight*networkLoss([int(np.floor(v[0])) + 1, int(np.floor(v[1])), v[2]]) +
                                           highSeqWeight*networkLoss([int(np.floor(v[0])) + 1, int(np.floor(v[1])) + 1, v[2]]))
  return discreteLowHidden + discreteHighHidden
```

Hyperparameter Sanity Check (RNN.py):

```
def sanityChecks(self):
      ### sanity checks ###
      # Square how mistaken the hyperparameters     #
      # are so second order expansions can operate. #
      # Shift error by the bound so optimizers know #
      # in which direction to guess.                #
      loss = 0
      if self.hidden_size < 20:
          loss += (self.hidden_size - 410)**2 + 1e6
      if self.hidden_size > 800:
          loss += (self.hidden_size - 410)**2 + 1e6
      if self.seq_length < 5:
          loss += (self.seq_length - 25)**2 + 1e6
      if self.seq_length > 40:
          loss += (self.seq_length - 25)**2 + 1e6
      if self.learning_rate < 1e-4:
          loss += (self.learning_rate - 0.5)**2 + 1e6
      if self.learning_rate > 1:
          loss += (self.learning_rate - 0.5)**2 + 1e6
      if loss == 0: # All hyperparameters are OK.
          return True
      return loss
```

Newton's Method Gradient Descent Fallback (newton.py)

```
if np.linalg.det(H) == 0: # The Hessian is noninvertible (singular). Abandon it and revert to gradient descent.
           step = -grad.dot(10 / np.sqrt(grad.dot(grad))) # Subtract 10 times the unit vector in the direction of the gradient.
else: # The Hessian is invertible: continue as normal.
           step = -(np.linalg.inv(H)).dot(grad).dot(1000) # step to the minimum
```

Backtracking Line Search (quasi_newton.py):

```
def lineSearch(f, n, grad, xk, pk):
    # Given a function and a in which direction to travel,
    # lineSearch solves for the optimal distance to travel to not under- or over-shoot.
    # Backtracking line search will initialize alpha, the distance to travel, as a high number.
    # alpha will be iteratively lessened until the Armijo-Goldstein condition is satisfied.
    alpha = 1
    control = 0.99 # 0 < control < 1 is a control parameter for the Armijo-Goldstein condition. See
https://en.wikipedia.org/wiki/Backtracking_line_search.
    lesseningFactor = 0.5 # 0 < lesseningFactor < 1 is multiplied into alpha at each iteration to lessen it.
    m = pk.dot(grad.T).tolist()[0] # local slope in direction pk
    t = control * m # Store this value for later access in the condition.
    fxk = f(xk.tolist()[0])
    # Armijo set control and lesseningFactor to 1/2 in his original paper, as done here.
    # Now, lessen alpha until the condition is satisfied. Break after 40 steps in case something went wrong.
    for i in range(20):
        # If the Armijo-Goldstein condition is met, terminate. Otherwise, lessen alpha.
        if f((xk[0] + alpha*pk).tolist()) <= fxk + alpha*t:
            break
        alpha = alpha * lesseningFactor
    return alpha
```

BFGS and DFP Inverse Hessian Update Formulae (quasi_newton.py):

```
def BFGS(V, sk, yk):
    ykskT = yk.dot(sk.T)
    return (1 + yk.dot(V).dot(yk.T)/ykskT)*(sk.T).dot(sk)/ykskT - (V.dot(yk.T).dot(sk) + (sk.T).dot(yk).dot(V))/ykskT #(1 +
(yk.T).dot(V).dot(yk)/skTyk)*(sk.dot(sk.T))/skTyk - (sk.dot(yk.T) * V+ V.dot(yk).dot(sk.T))/skTyk

def DFP(V, sk, yk):
    VykT = V.dot(yk.T)
    return (sk.T).dot(sk)/yk.dot(sk.T) - (VykT.dot(yk).dot(V))/yk.dot(VykT)
```

Quasi-Newton Step and Gradient Descent Fallback (quasi_newton.py):

```
grad = np.array([mvc.gradient(f, xk.tolist()[0])])
# Calculate direction by Newton's Method with an approximated inverse Hessian.
pk = -V.dot(grad[0])
# Perform line search to calculate step size, alpha.
alpha = lineSearch(f, n, grad, xk, pk) # Calculate to find next point after step.
sk = alpha*pk # step at iteration k
if sk.dot(sk) < convergenceSquared or not sanityCheck(xk.tolist()[0]): # Hyperparameters are probably wacky. Revert to Gradient Descent.
    step = -grad[0].dot(1 / np.sqrt(grad[0].dot(grad[0]))) # Subtract 0.1 times the unit vector in the direction of the gradient.
    xk = xk + step
    continue
sk = np.array([sk.tolist()])
xk_next = xk + sk
# Now, for the new xk, update the approximate inverse Hessian with BFGS.
yk = np.array([mvc.gradient(f, xk_next.tolist()[0])]) - grad
xk = xk_next
#### Update the inverse Hessian with a combination of DFP and BFGS. For more details, see
en.wikipedia.org/wiki/Broyden–Fletcher–Goldfarb–Shanno_algorithm.
V = V + DFP(V, sk, yk) # + BFGS(V, sk, yk)
```

Samples:

Below is a 2000 character sample of Shakespeare generated by an RNN trained for 100000 steps with hyperparameters from Fig 1A (Gradient Descent). The following link has samples from all optimized hyperparameters from Fig 1A, Fig 3A, and Fig 3B. The Quasi-Newton methods returned unusable hyperparameters, so their results weren't sampled.

https://docs.google.com/document/d/1TEWN6zrRn-GlkaOtPP6IJODOwUpv8Wwu4M8iXa5KfQ4/edit?usp=sharing

Sampling 2000 characters from [36.7481, 35.9527, 0.2646] after 100000 training steps gives

nkigh.

MEDRIU:
Hour fay, it lighs im eak it upan exoll griend.
Dhe I day a le, thosoprerss.

Fordant ho
W'll? shan
Mysroudr's 'trick o's macltar you: yes:
to proke in gexen:
Will swan us this afore'd, his corvole esmorlingiss I and youles thly wo my by me, pun toicked. I yaur'd meat of carded shald in's ull of ubocriof heacks no radaret in in and weert,
If shilk.

MENENIUS:
Sion mich,
Maved,
And beake,
Thap your:
Thim ins!
Lerve wilene, extey
To grome.

CODYOLINIA:
The whe couiwies esterut thee
All. I weal you to must icy To mamed hace, we pigh the lockes.

CRUDININIUM:UFoDINIUS:
Think to shy have comells---eal, let eyen!
sit espellsroud worle
That frowrres
The say cund when, be the as hesid-. I the ink,
Didse hicher alere allssmellar my caur ang
The extarse sone that ward outt, comast.

MENENIUS:
Whour laad.

MORTIUSINILA:
Weme.

CUCINIUS:
Haje's eden's'n.

VIMISS:
Honds
I'rrey pare owre, heil:
first louge,
Will you be thin'clss, goud his halr; Toome fills jepore
Wifl, say mud gread yike whou the that hamrman,--
I, his nengom Corte dorduifite'd thinh to oul our quave bes haver, cho the sor you as mall have gon him:
God you chaurse, thime fore this thincite.
Time hime.

PEUCUSeUVUSIUS:
As inss. I you you:
The hir in verr'd munt, coursall roy, by imear dist;
As locd
Arrer.

COLINIUS:
I'lr! Col have in revaak
Thee she that hids: the fill tourl mile cerse deeage his emery Po heedy cour sirind.
Huscupry som why caRave
fore theadn pooly,
Thidh:
And sive thous llokd deare, Musious.

COMPEYOLUMANIUS:
Wo I wo, suciking your te efel gtenss,
Now, it so goes ouy,
Seeth
And linges graod itt langss finv,
Do do you.

Fohe housofund's cod of to, sirvor from fing you bour hall have poof morywite ho him your Seak sow? lord thes llot.

MENEUS:
A them to ruve!
Whle lebr,
And froye.

Mied life grave!

VAAUDILIS:
A deald ousnand belith genvers
The whinney losen
To'es ond, to whenise sirspy him, or blet!

VOLUS:
All of to heers, lorded
Pill,
For-hid allshem:
Murby hese
Themand where y.

References:

Armijo, Larry. “Minimization of Functions having Lipschitz Continuous First Partial Derivatives.” Pacific Journal of Mathematics, Volume 16, No. 1, 1966. projecteuclid.org/download/pdf_1/euclid.pjm/1102995080

Fletcher, Roger (1987), Practical methods of optimization (2nd ed.), New York: John Wiley & Sons, ISBN 978-0-471-91547-8.

C. G. BROYDEN; The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations, IMA Journal of Applied Mathematics, Volume 6, Issue 1, 1 March 1970, Pages 76–90, doi.org/10.1093/imamat/6.1.76

Shanno, D F. “Conditioning of Quasi-Newton Methods for Function Minimization.” Mathematics of Computation, vol. 24, no. 111, 1970, doi:10.1090/mcom/1970-24-111.

Shakespeare, William. “The Complete Works of Shakespeare.” Project Gutenberg, Project Gutenberg, gutenberg.org/files/100/100-0.txt