Neural Network Quine

Oscar Chang[1] and Hod Lipson[1]

[1]Data Science Institute, Columbia University, New York, NY 10027
oscar.chang, hod.lipson@columbia.edu

Abstract

Self-replication is a key aspect of biological life that has been largely overlooked in Artificial Intelligence systems. Here we describe how to build and train self-replicating neural networks. The network replicates itself by learning to output its own weights. The network is designed using a loss function that can be optimized with either gradient-based or non-gradient-based methods. We also describe a method we call *regeneration* to train the network without explicit optimization, by injecting the network with predictions of its own parameters. The best solution for a self-replicating network was found by alternating between regeneration and optimization steps. Finally, we describe a design for a self-replicating neural network that can solve an auxiliary task such as MNIST image classification. We observe that there is a trade-off between the network's ability to classify images and its ability to replicate, but training is biased towards increasing its specialization at image classification at the expense of replication. This is analogous to the trade-off between reproduction and other tasks observed in nature. We suggest that a self-replication mechanism for artificial intelligence is useful because it introduces the possibility of continual improvement through natural selection.

Introduction

The concept of an artificial self-replicating machine was first proposed by John von Neumann in the 1940s prior to the discovery of DNA's role as the physical mechanism for biological replication. Specifically, Von Neumann demonstrated a configuration of initial states and transformation rules for a cellular automaton that produces copies of the initial cell states after running for a fixed number of steps (Von Neumann and Burks, 1966). Hofstadter (1980) later coined the term 'quine' in *Gödel, Escher, Bach: an Eternal Golden Braid* after the philosopher Willard Van Orman Quine, to describe self-replicating expressions such as: 'is a sentence fragment' is a sentence fragment.

There has also been work done in making physical self-replicators. Notable examples include tiles (Penrose, 1959), molecules (Wang et al., 2011), polymers (Breivik, 2001), and robots (Zykov et al., 2005).

In the context of programming language theory, quines are computer programs that print their own source code. A trivial example of a quine is the empty string, which in most languages, the compiler transforms into the empty string. The following code snippet is an example of a non-trivial Python quine written in two lines.

```
s = 's = %r\nprint(s%%s)'
print(s%s)
```

While self-replication has been studied in many automata, it is notably absent in neural network research, despite the fact that neural networks appear to be the most powerful form of AI known to date.

In this paper, we identify and attempt to solve the challenges involved in building and training a self-replicating neural network. Specifically, we propose to view a neural network as a differentiable computer program composed of a sequence of tensor operations. Our objective then is to construct a neural network quine that outputs its own weights.

We tested our approach using three distinct classes of methods: gradient-based optimization methods, non-gradient-based optimization methods, and a novel method called regeneration. We further designed a neural network quine which has an auxiliary objective in addition to the job of self-replication. In this paper, the chosen auxiliary task is MNIST image classification (LeCun and Cortes, 1998), which involves classifying images of digits from 0 to 9, and is commonly used as a 'hello world' example for machine learning.

We observed a trade-off between the network's ability to self-replicate and its ability to solve the auxiliary task. This is analogous to the trade-off between reproduction and other auxiliary survival tasks observed in nature. The two objectives are usually aligned, but for example, when an animal has been put in starving conditions, its sex hormones are usually down-regulated to optimize for survival at the expense of reproduction. The opposite occurs as well: for example, in male dark fishing spiders, the act of copulation results in a sudden irreversible change to its blood pressure, immobilizing it and leaving it vulnerable to cannibalization by the female spider (Drake, 2013).

Motivations

Modern artificial intelligence is primarily powered by deep neural networks for applications as diverse as detecting diabetic retinopathy (Gulshan et al., 2016), synthesizing human-like speech (Shen et al., 2017), and executing strategic decisions in Starcraft (Vinyals et al., 2017). In line with ALIFE 2018's theme of going *beyond AI*, we list several motivations for studying self-replicating neural networks.

- Biological life began with the first self-replicator (Marshall, 2011), and natural selection kicked in to favor organisms that are better at replication, resulting in a self-improving mechanism. Analogously, we can construct a self-improving mechanism for artificial intelligence via natural selection if AI agents had the ability to replicate and improve themselves without additional machinery.

- Neural networks are capable of learning powerful representations across many different domains of data (Bengio et al., 2013). But can a neural network learn a good representation of itself? Self-replication involves a degree of introspection and self-awareness, which is helpful for lifelong learning and potential discovery of new neural network architectures.

- Learning how to enhance or diminish the ability for AI programs to self-replicate is useful for computer security. For example, we might want an AI to be able to execute its source code without being able to read or reverse-engineer it, either through its own volition or interaction with an adversary.

- Self-replication functions as the ultimate mechanism for self-repair in damaged physical systems (Zykov et al., 2005). The same may apply to AI, where a self-replication mechanism can serve as the last resort for detecting damage, or returning a damaged or out-of-control AI system back to normal.

Related Work

Quines have been written for a variety of programming languages. The Quine Page (Thompson, 1999) contains code contributions of quines written in 55 different languages. An Ouroboros set of programs extends the concept of a quine by having a program in language A generate the source code for a program in language B, which then generates the source code for a program in a language C, and so on, until it finally generates back the source code for the initial program in language A. Endoh (2017) made an Ouroboros with 128 programming languages in it.

Our work focuses on building a self-replication mechanism via weight prediction. Denil et al. (2013) demonstrated the presence of redundancy in neural networks by using a portion of the weights to predict the rest. There are also neural networks that can modify the weights of other neural networks (Schmidhuber, 1992; Ha et al., 2017), which have been shown to be useful in meta-learning an optimizer (Ravi and Larochelle, 2016; Andrychowicz et al., 2016). Schmidhuber (1993) proposed an architecture and a training algorithm for a self-referential recurrent neural network, which is philosophically very similar to our work in that the network refers to itself rather than another network. To our knowledge, our work is the first to attempt the task of self-replication in neural networks.

Building the Network

How can a neural network refer to itself?

Problem with Direct Reference A neural network is parametrized by a set of parameters Θ, and our goal is to build a network that outputs Θ itself. This is difficult to do directly. Suppose the last layer of a feed-forward network has A inputs and B outputs. Already, the size of the weight matrix in a linear transformation is the product AB which is greater than B for any $A > 1$.

We also looked at open-source implementations of two popular generative models for images, DCGAN (Radford et al., 2016) and DRAW (Gregor et al., 2015). They use 12 million and 1 million parameters respectively to generate MNIST images with 784 pixels.

In general, the set of parameters Θ is a lot larger than the size of the output. To circumvent this, we need an indirect way of referring to Θ.

Indirect Reference HyperNEAT (Stanley et al., 2009) is a neuro-evolution method that describes a neural network by identifying every topological connection with a coordinate and a weight. We pursue the same strategy in building a quine. Instead of having the quine output its weights directly, we shall set it up so that it inputs a coordinate (in a one-hot encoding) and outputs the weight at that coordinate.

This overcomes the problem of Θ being larger than the output, since we are only outputting a scalar Θ_c for each coordinate c.

Vanilla Quine

We define the *vanilla quine* as a feed-forward neural network whose only job is to output its own weights.

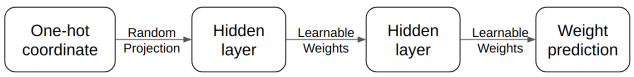

Figure 1: Structure of a vanilla quine

Suppose the number of inputs is A, and the number of units in the first hidden layer is B, then the size of the projection matrix would be the product AB which is greater

than A for any $B > 1$. Hence, we cannot have the projection itself be a parameter of the network due to the one-hot representation. We thus decide to use a fixed random projection to connect the one-hot encoding of the coordinate to the hidden layer. All other connections, namely the connections between the hidden layers as well as the connections between the last hidden layer and the output layer, are variable parameters of the neural network.

Von Neumann argued that a non-trivial self-replicator necessarily includes three components that by themselves do not suffice to be self-replicators: (1) a description of the replicator, (2) a copying mechanism that can clone descriptions, and (3) a mechanism that can embed the copying mechanism within the replicator itself (Von Neumann and Burks, 1966). In this case, the coordinate system that assigns each of the weights a point in the one-hot space corresponds to (1). The function computed by the neural network corresponds to (2). The fixed random projection corresponds to (3). We explain below reasons for our choices of (1), (2), and (3), while keeping in mind that alternatives to these choices are interesting future research directions.

(1) One-hot Input Encoding A one-hot encoding is a vector that contains exactly a single '1', and is 0 everywhere else. If we directly input the coordinate instead of using a one-hot encoding, then the network will not be sufficiently expressive. This is because for any coordinate c, the difference between $f(c)$ and $f(c+1)$ is constrained by the network's Lipschitz bound, hence the network cannot accurately output the weights at c and $c+1$ if their difference is sufficiently big. We demonstrate a visualization of this in Figure 2: contiguous weights might be very different, but contiguous outputs cannot be very different.

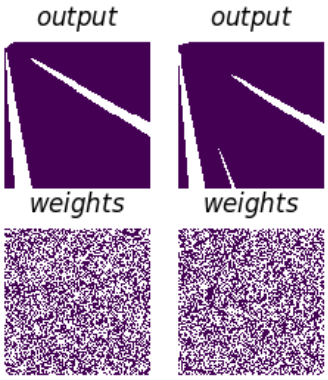

Figure 2: Log-normalized illustration of the weights and weight predictions of a quine without one-hot encoding

(2) Multi-Layered Perceptrons

$$y_i = \sigma_i(W_i x_i + b_i) \tag{1}$$

Multi-layered perceptrons (MLPs) are feed-forward neural networks that consist of repeated applications of Equation 1, where at the ith layer of the network, σ_i is an activation function, W_i a weight matrix, b_i a bias vector, x_i the input vector, and y_i the output vector. MLPs are known to be good function approximators, specifically a feedforward neural network with at least one hidden layer forms a class of functions that is dense in the space of continuous functions under a compact domain (Hornik, 1991; Cybenko, 1989). While not precluding other kinds of generative neural network architectures, this makes an MLP seem like a suitable candidate for a neural network quine, because we think it is expressive enough to derive and store a representation of itself.

(3) Random Projections We believe that random projections are a good choice as an embedding layer to connect a one-hot representation into the network because of their distance-preserving property (Johnson and Lindenstrauss, 1984) and the fact that random features have been shown to work well both in theory and practice (Rahimi and Recht, 2008). Indeed, they form a key component of Extreme Learning Machines (Huang et al., 2006) which are feedforward neural networks that have proven useful in classification and regression problems.

Auxiliary Quine

We define the *auxiliary quine* to be a vanilla quine that solves some auxiliary task in addition to self-replication. It is responsible for taking in an auxiliary input and returning an auxiliary output.

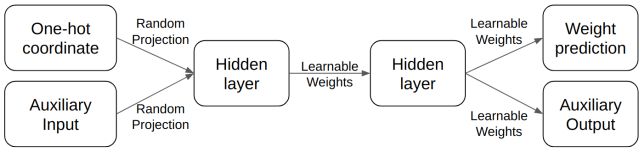

Figure 3: Structure of an auxiliary quine

In this paper, we chose image classification as the auxiliary task. The MNIST dataset (LeCun and Cortes, 1998) contains square images (28 pixels by 28 pixels) of handwritten digits from 0 to 9. These images are to be fed in as the auxiliary input. It is possible to make the connection from the auxiliary input to the network a parameter rather than a random projection, but in this paper, we only report results for the latter.

The auxiliary output is a probability distribution over the ten classes, where the class with the maximum probability will be chosen as the predicted classification. 60000 images are used for training and 10000 images are used for testing; we have no need for a validation set since we are not

strictly trying to optimize for the performance of the classifier. Our primary aim in this paper is to demonstrate a proof of concept for a neural network quine, which makes MNIST a suitable auxiliary task as it is considered an easy problem for modern machine learning algorithms.

Training the Network

Network Architecture

Before describing how the neural network quines are trained, we specify the exact network architecture used in our experiments below for both the vanilla quine and the auxiliary quine. In both cases, they are MLPs composed of two hidden layers with 100 hidden units each where every layer is followed by a SeLU (Klambauer et al., 2017) activation function. In the case of the auxiliary quine, the one-hot coordinate is projected to the first 50 hidden units, while the MNIST input is projected to the next 50 hidden units. The auxiliary output is a vector of size 10 (number of classes) computed by a softmax.

The total number of parameters is 20,100 for the vanilla quine, and 21,100 for the auxiliary quine. The nature of the quine problem and our choice of the one-hot encoding means that the input vector will be of the same size as the number of parameters. These are small networks by modern deep learning standards where millions of parameters are the norm, but it is a challenge to handle input vectors with dimensions much larger than 20,000.

How do we train a neural network quine?

Self-Replicating Loss We define the *self-replicating loss* to be the sum of the squared difference between the actual weight and its predicted value. A vanilla quine is achieved when this loss is exactly zero. Because of numerical imprecision errors, we can expect that in practice, optimizing this loss will nonetheless result in a number slightly above zero, except for the trivial *zero quine* where all the weights are exactly zero to begin with.

$$L_{SR} = \sum_{c \in C} \left\| f_\Theta(c) - \Theta_c \right\|_2^2 \tag{2}$$

Auxiliary Loss It is possible to jointly optimize an existing loss function with the self-replicating loss so that a neural network gains the ability to self-replicate in addition to an auxiliary task it specializes in. We define the *auxiliary loss* to be the sum of the self-replicating loss L_{SR} and the loss from the auxiliary task L_{Task}, with a hyperparameter λ to scale both losses to a similar magnitude. An auxiliary quine can be trained by optimizing on the auxiliary loss, but we do not expect to see a near-zero loss, unless it is also perfect at the auxiliary task. In our MNIST experiment, L_{Task} is the cross-entropy loss, which is commonly used for classification problems.

$$L_{Aux} = L_{SR} + \lambda L_{Task} \tag{3}$$

Training Methods There are three distinct classes of methods that we can use to train our neural network quines.

- **Gradient-based methods** Stochastic gradient descent (SGD) and its variants are the workhorse algorithm for training deep neural networks today. In our case, the loss function is a moving target, since Θ_c changes after each gradient update. Updating the loss function after every mini-batch update is expensive. To avoid that, we split the set of possible coordinates into random mini-batches of size 10, and update the loss function after every training epoch. In other words, each training epoch will consist of running through the set of all possible coordinates. We do not use a validation set for our experiments, while the test loss is computed at the end of every training epoch after updating the loss function. Below is pseudo-code for training a vanilla quine. A similar procedure is used to train an auxiliary quine with L_{SR} replaced with L_{Aux} to account for the auxiliary task.

  Pseudo-code for training a vanilla quine via optimization:

  ```
  begin
    Initialize set of parameters Θ_C
    Initialize number of training epochs T
    for t := 0 to T do
      Θ_t := Θ_C
      Divide Θ_t into random mini-batches
      for each mini-batch do
        Compute L_SR
        Θ_C := optimize(Θ_C, L_SR)
      end
    end
  end
  ```

- **Non-gradient-based methods** Optimization methods that do not make use of gradient information can also be used to train neural networks. For example, evolutionary algorithms have been used successfully to train deep reinforcement learning agents with over four million parameters (Such et al., 2017; Salimans et al., 2017). For the same reasons of computational efficiency as mentioned above, we shall choose to execute non-gradient-based optimization in mini-batches. The training algorithm is identical to the pseudo-code shown above, except with `optimize` being non-gradient-based. We only consider hill-climbing in this paper.

- **Regeneration method** Perhaps somewhat surprisingly, it is also possible to train a vanilla quine without explicitly optimizing for the self-replicating loss. We do so by replacing the current set of parameters with the weight predictions made by the quine. Each such replacement is called a *regeneration*. We then alternate between executing regeneration and a round of optimization to achieve a low but non-trivial self-replicating loss. We note that re-

generation is sensitive to choices of weight initialization and activation function.

Pseudo-code for Regeneration:

```
begin
  Initialize set of parameters Θ_C
  Initialize number of generation epochs G
  Initialize number of optimization epochs T
  for g := 0 to G do
    // Optimization
    for t := 0 to T do
      Θ_t := Θ_C
      Divide Θ_t into random mini-batches
      for each mini-batch do
        Compute L_SR
        Θ_C := optimize(Θ_C, L_SR)
      end
    end
    // Regeneration
    for c ∈ C do
      Θ_c := f_{Θ_C}(c)
    end
  end
end
```

Results and Discussion

In the experimental results produced below, we used a mini-batch of size 10 for training. λ in L_{Aux} and the temperature for the softmax in the auxiliary output are set to 0.01.

Vanilla Quine

We trained a vanilla quine with classical SGD ($lr = 0.01$), SGD with momentum ($lr = 0.01$, $\rho = 0.9$), ADAM (Kingma and Ba, 2014), Adagrad (Duchi et al., 2011), Adamax (Kingma and Ba, 2014), and RMSprop (Tieleman and Hinton, 2012) with default hyperparameter settings on the self-replicating loss for 30 epochs. The quine was initialized with the same procedure as in He et al. (2015), and the initial loss L_{SR} prior to any training was 90.16. We observe in Figure 4 that Adamax performed the best, while Adagrad exhibited increasing loss rather than plateauing. RMSprop (not plotted) was found to explode the loss right from the start of training. We carried on training the quine on Adamax for 100 epochs, achieving a best test loss of 32.10 by the end of training, which is a third of its pre-trained value.

Is this a quine?

It is hard to quantify how significant it is to reduce the self-replicating loss to a third of its pre-trained value. After all, our goal was to produce a self-replicator, but if the loss we achieved is not close to zero, then it seems that we have not reached our goal. On the other hand, replication mechanisms are rarely perfect. Even in nature, replication mechanisms often contain high levels of noise, sometimes referred to as 'mutation' or replication error.

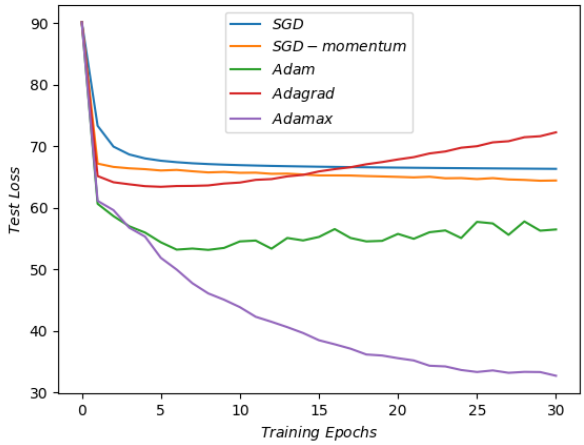

Figure 4: Comparison of gradient-based optimization methods used to train a vanilla quine

Adams and Lipson (2009) constructed a mathematical framework to calculate the *self-replicating quotient* of a replicator, which measures the log likelihood ratio of a perfect self-replication happening via the replicator's noisy replication mechanism compared to it happening by chance. For example, Zykov et al. (2005) estimate the self-replicating quotient of Penrose Tiling (Penrose, 1959) to be below 0.69 and that of animals to be at least 46.05. This framework is useful for distinguishing between trivial and non-trivial replicators. To compute this metric for our network, we need to compute the chance that a random neural network would produce a copy of itself within the same ϵ-ball as achieved by our vanilla quine. Assuming that a random network has a uniform distribution of outputs from $[-0.5, 0.5]$ (a big assumption), then the self-replicating quotient for the vanilla quine is 6.44, which implies a certain amount of non-triviality.

Another measure we can look at is the *average weight prediction margin*, which is defined as the average absolute difference between the weights and the weight predictions. The pre-training loss of 90.16 corresponds to an average weight prediction margin of 0.067, while the post-training loss of 32.10 corresponds to an average weight prediction margin of 0.040. This suggests we still have significant room for improvement. However, it is worth pointing out that the relatively small pre-training weight prediction margin reflects the fact that modern best practices for the choice of weight initialization and activation function keep the output in the same order of magnitude as the input.

Hill-climbing

Next, we use a hill-climbing algorithm to train the vanilla quine. The algorithm works by iteratively perturbing the parameters of the network with diagonal Gaussian noise and keeping the perturbation if it results in an improvement. This is equivalent to an evolutionary algorithm with a population size of 1. In this case, we do not need the gradients, hence the training process only requires the forward and not the backward pass, which makes each training epoch computationally cheaper. Nonetheless, it takes around 5000 epochs to find a solution that is on par with that found by classical SGD after 10 epochs. We found that doing hill-climbing on the solution that SGD converged to improves it significantly, but the same does not hold true for the solution that Adamax converged to. This suggests that the solution found by Adamax is already a local optima.

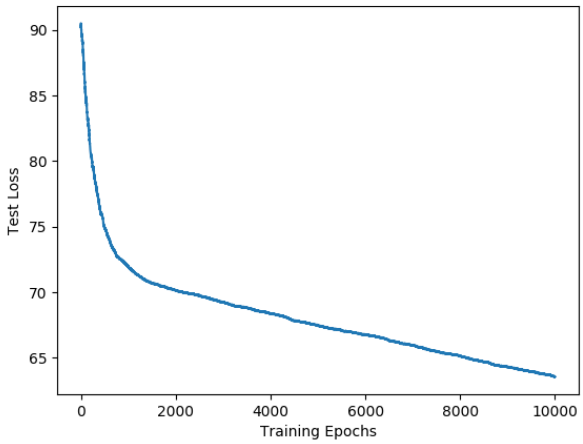

Figure 5: Training a vanilla quine via hill-climbing

Regeneration

Finally, we use regeneration to train the vanilla quine, setting $T = 1$ with Adamax as the optimizer. Each generation epoch is very computationally expensive as it involves as many forward passes as there are parameters in the network to replace its actual weights with its outputs. However, one epoch suffices to reduce the test loss substantially, with the best self-replicating loss of 0.86 found after ten generation epochs. This corresponds to a self-replicating quotient of 10.06 and an average weight prediction margin of 0.0065, which is an order of magnitude better than the best solution found previously.

One might wonder if the solution we found via regeneration might simply be the zero quine, i.e. all the weights are zero. Indeed, we find that iteratively injecting the network with its predicted weights has a similar effect as statistical shrinkage. It effectively learns to reduce the self-replicating

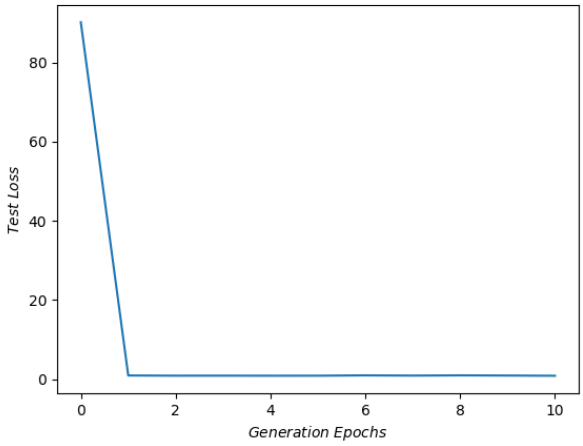

Figure 6: Training a vanilla quine via regeneration

loss by shrinking the order of magnitudes of the weights, thus creating a small weight prediction margin. Without the optimization step (when $T = 0$), a visual inspection of the network reveals that it rapidly converges to the zero quine. However, with the optimization step, the solution found appears to be non-trivial: the order of magnitude of the weights are in line with what we would observe in a normal neural network. The strong attraction of the zero quine also underscores the importance of having an additional auxiliary task.

Figure 7 shows a visualization of the solution found by regeneration.

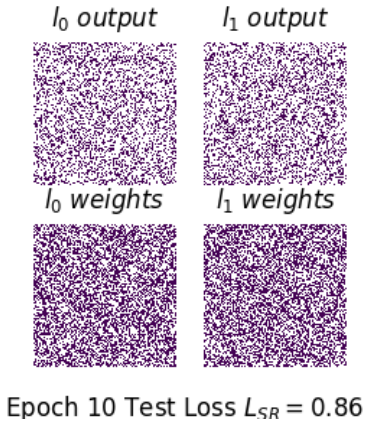

Figure 7: Log-normalized illustration of the weights and weight predictions of two hidden layers in a vanilla quine that has been trained with regeneration

Auxiliary Quine

We trained an auxiliary quine on the MNIST image classification task with Adamax using the default hyperparameter settings on 30 epochs. The quine was also initialized with He init, and the initial loss L_{Aux} prior to any training was 1072.05. We observe in Figure 8 that somewhat counter-intuitively, after the initial drop, the auxiliary loss actually increases over time instead of converging. This is due to the network prioritizing the task loss L_{Task} over the self-replicating loss L_{SR} despite the fact that it is being optimized on their sum. The same trend is observed when we repeat the experiment on other gradient-based optimization methods besides Adamax. After 30 epochs, the network achieved an accuracy of 90.41% on the held-out test set, which is comparable to the 96.33% achieved by an identical network whose only objective is MNIST image classification. This shows that self-replication occupies a significant portion of the neural network's capacity, but it is heartening nonetheless that joint optimization of the objectives is possible. If we leave the auxiliary quine running, the task loss eventually converges, while ignoring the exploding self-replicating loss.

This is an interesting finding: it is more difficult for a network that has increased its specialization at a particular task to self-replicate. This suggests that the two objectives are at odds with each other, but that the gradient-based optimization procedure prefers to maximize the network's specialization at solving the MNIST task, even at the expense of a reduction in its ability to self-replicate. (It is not immediately obvious from Figure 8, but the first few training epochs reduce the self-replicating loss too.)

There are parallels to be drawn between self-replication in the case of a neural network quine and biological reproduction in nature, as well as specialization at the auxiliary task and survival in nature. The mechanisms for survival are usually aligned with the mechanisms for reproduction, however when they come into conflict with each other, the survival mechanism usually is prioritized at the expense of the reproduction mechanism (except in rare cases like that of the male dark fishing spider). We hypothesize that in nature, self-replication might rapidly become trivial without the presence of an auxiliary task, just as we have observed here in the case of a neural network quine.

Hill-climbing progressed too slowly for us to observe anything meaningful, but we do not expect to observe the same behavior because the algorithm, by definition, does not allow for harmful changes to the overall loss to be made. The regeneration techique cannot be used in this case, because we require the auxiliary input for each generation and random inputs do not work well.

Conclusion

In this paper, we have described how to build and train a self-replicating neural network. Specifically, we proposed to treat the problem of self-replication in a neural network as a problem of weight prediction, and devised various encoding and training schemes to solve this problem. This allowed us to create a neural network quine, which akin to a computer program quine, outputs its own source code (weights in this case).

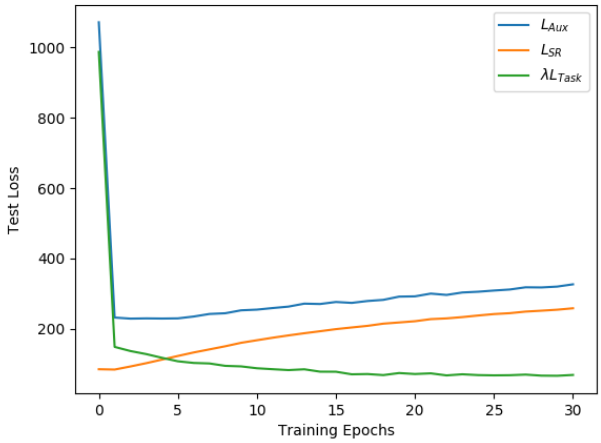

Figure 8: Training an auxiliary quine with Adamax

We have identified three interesting future directions for research. Firstly, we can seek to improve weight prediction by assuming a low-rank matrix factorization for the network's weights as in Denil et al. (2013). Secondly, we can attempt to build neural network quines using more sophisticated models and representations, for example a convolutional neural network quine might be interesting. Thirdly, we can extend the concept of self-replication to universal replication: a neural network that can replicate other neural networks.

Acknowledgements

We would like to thank Peter Duraliev for helpful comments. This research was supported in part by the US Defense Advanced Research Project Agency (DARPA) *Lifelong Learning Machines* Program, grant HR0011-18-2-0020.

References

Adams, B. and Lipson, H. (2009). A universal framework for analysis of self-replication phenomena. *Entropy*, 11:295–325.

Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., and de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989.

Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35, no. 8:1798–1828.

Breivik, J. (2001). Self-organization of template-replicating polymers and the spontaneous rise of genetic information. *Entropy*, 3(4):273–279.

contributors, W. (2017). Self-replication — wikipedia, the free encyclopedia. [Online; accessed 5-March-2018].

contributors, W. (2018). Quine (computing) — wikipedia, the free encyclopedia. [Online; accessed 5-February-2018].

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

Denil, M., Shakibi, B., Dinh, L., De Freitas, N., et al. (2013). Predicting parameters in deep learning. In *Advances in neural information processing systems*, pages 2148–2156.

Drake, N. (2013). Why male dark fishing spiders die spontaneously after sex. *Wired*.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Endoh, Y. (2017). Quine relay.

Gregor, K., Danihelka, I., Graves, A., Rezende, D. J., and Wierstra, D. (2015). Draw: A recurrent neural network for image generation. *Proceedings of the 32nd International Conference on Machine Learning*.

Gulshan, V., Peng, L., Coram, M., Stumpe, M. C., Wu, D., Narayanaswamy, A., Venugopalan, S., Widner, K., Madams, T., Cuadros, J., Kim, R., Raman, R., Nelson, P. C., Mega, J. L., and Webster, D. R. (2016). Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *JAMA*, 316(22):2402–2410.

Ha, D., Dai, A. M., and Le, Q. V. (2017). Hypernetworks. *International Conference on Learning Representations*.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034.

Hofstadter, D. (1980). *Gödel, Escher, Bach: an Eternal Golden Braid*. New York: Vintage Books.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257.

Huang, G.-B., Zhu, Q.-Y., and Siew, C.-K. (2006). Extreme learning machine: theory and applications. *Neurocomputing*, 70(1-3):489–501.

Johnson, W. B. and Lindenstrauss, J. (1984). Extensions of lipschitz mappings into a hilbert space. *Contemporary mathematics*, 26(189-206):1.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Klambauer, G., Unterthiner, T., Mayr, A., and Hochreiter, S. (2017). Self-normalizing neural networks. In *Advances in Neural Information Processing Systems*, pages 972–981.

LeCun, Y. and Cortes, C. (1998). The mnist database of handwritten digits.

Marshall, M. (2011). First life: The search for the first replicator. *New Scientist*, Issue 2825.

Penrose, L. S. (1959). Self-reproducing machines. *Scientific American*, 200:105–112.

Radford, A., Metz, L., and Chintala, S. (2016). Unsupervised representation learning with deep convolutional generative adversarial networks. *International Conference on Learning Representations*.

Rahimi, A. and Recht, B. (2008). Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184.

Ravi, S. and Larochelle, H. (2016). Optimization as a model for few-shot learning.

Salimans, T., Ho, J., Chen, X., and Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Schmidhuber, J. (1992). Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139.

Schmidhuber, J. (1993). A self-referentialweight matrix. In *ICANN93*, pages 446–450. Springer.

Shen, J., Pang, R., Weiss, R. J., Schuster, M., Jaitly, N., Yang, Z., Chen, Z., Zhang, Y., Wang, Y., Skerry-Ryan, R., et al. (2017). Natural tts synthesis by conditioning wavenet on mel spectrogram predictions. *arXiv preprint arXiv:1712.05884*.

Stanley, K. O., DAmbrosio, D., and Gauci, J. (2009). A hypercube-based indirect encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212.

Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., and Clune, J. (2017). Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*.

Thompson, G. P. (1999). The quine page.

Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31.

Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J., et al. (2017). Starcraft ii: a new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*.

Von Neumann, J. and Burks, A. W. (1966). Theory of self-reproducing automata. page 8. Urbana: University of Illinois Press.

Wang, T., Sha, R., Dreyfus, R., Leunissen, M. E., Maass, C., Pine, D. J., Chaikin, P. M., and Seeman, N. C. (2011). Self-replication of information-bearing nanoscale patterns. *Nature*, 478(7368):225.

Zykov, V., Mytilinaios, E., Adams, B., and Lipson, H. (2005). Robotics: Self-reproducing machines. *Nature*, 435:163–164.