HYPERNETWORKS ARE GENERATIVE MODELS FOR WEIGHTS OF INTERPRETABLE NETWORKS

Isaac Liao
iliao2345.github.io
iliao2345@gmail.com

ABSTRACT

We use a hypernetwork as a generative model for neural network weights, and we build in an inductive bias for generating weights that are structured, symmetric, patterned, sparse, and otherwise interpretable. We explore several interpretations for generated models that compute L1 norms. We find a diversity of strategies which can be automatically classified, we analyze how these strategic phases develop during training, and we study the method by which the hypernetwork constructs these strategic phases. We show that the hypernetwork can construct models that compute L1 norms in many dimensions despite only being trained for dimension 16. We finally observe that none of this structure emerges when we train the network using Adam instead of generating the weights using a hypernetwork. This paper is a half-baked and is constructed from some preliminary results. We decided to take about an hour or so to clean our work up to make what you see here.

1 INTRODUCTION

Although large language models have demonstrated a number of surprising mathematical and algorithmic capabilities (Yuan et al., 2023), it remains unknown whether they rediscover algorithms familiar to humans, or they create more alien forms of mathematics and algorithms that appear counterintuitive to humans (at least at first sight). This question can be partially answered by recent efforts to mechanistically interpret neural networks.

This paper aims to answer this question: what is the minimal example where "alien math" can emerge? We find that an extremely simple setup, i.e., computing the L_1 norm of a vector with a two-layer MLP, is already able to demonstrate very complicated behavior. A conventionally trained network appears to be highly uninterpretable, with no clear patterns in weights or activations. We hence resort to hypernetworks (Chauhan et al., 2023) to produce weights for our networks, to facilitate mechnanistic interpretation. In particular, we identify in our neural networks three types of algorithms for computing the L_1 norm: the double-sided ReLU algorithm, the pudding algorithm, and the convexity algorithm (see Figure 1), though the authors only expected the double-sided ReLU algorithm to be learned before experiments revealed otherwise. We further define order parameters to auto-classify these algorithms and find intriguing phase transitions between their occurrences, either in training, or when model complexity is varied. By ablating our hypernetwork, we also find that hypernetworks produce the pudding algorithm in two main ways, only one of which is disrupted by the ablation. Through all this analysis, we finally arrive at a hypothesis for how a conventionally trained L1 norm network works, which is much more complicated than any of the strategies found by the hypernetwork.

Our results highlight the complexity of mechanistic description even in models trained to perform extremely simple mathematical tasks. They call for full exploration of algorithmic phase spaces, not just single algorithmic solutions based on human intuition.

2 L1 NORM NETWORK EXPERIMENT

In this section, we explain how all of the networks we trained can compute the L1 norm. All of our networks are two-layer MLPs with 16 inputs, 48 hidden neurons, and 1 output, with swish activation

(Ramachandran et al., 2017) (Hendrycks & Gimpel, 2016). The training data is randomly generated online by sampling the inputs from iid standard normal distribution, and by shifting and rescaling the target L1 norm outputs so that they have zero mean and unit variance.

We train a hypernetwork to generate good weights for this network. Our hypernetwork has a hyperparameter β which controls the balance between the objectives of lower loss and lower model complexity (which we measure using a KL divergence). A higher β values simplicity over loss, and a lower β values loss over simplicity. This is by no means a complete explanation nor even a working model of how the hypernetwork works; Appendix A.2 contains a complete description. As a result of this training, we have a large number of models saved at various points in training for many different β values. Moreover, we repeated this experiment 33 times so we have 33 independently sampled copies of all of these models.

2.1 Interpretation of Generated Networks

In this section, we deconstruct the strategies performed by the networks generated by our hypernetworks. We determined these strategies by looking at force-directed graph drawings of the learned networks (Kobourov, 2012). Force-directed graph drawings are a way to visualize graphs of computations by organizing the nodes on the plane of a drawing, to make diagrams of these graphs more intuitive to read. Our force-directed graph drawings assign a position on a drawing to every neuron to minimize an energy consisting of mutual repulsion, connection strength weighted attraction, and central attraction (Bannister et al., 2013). Full details of how we generate these drawings can be found in Appendix A.1.

By looking at these drawings, we found that networks generally compute the L1 norm using one of three main strategies:

- **The Double-sided ReLU Strategy** Most humans, if given a two-layer neural network with swish activation and told to pick weights to compute an L1 norm, would pick the same strategy. They would realize that a zoomed out plot of a swish function looks like a ReLU, then they would add the output of two oppositely-oriented ReLUs together to make an absolute value function, and then they would compute a sum of absolute value functions to create the L1 norm:

$$||\boldsymbol{x}||_1 = \sum_{i=1}^{16} \text{ReLU}(x_i) + \sum_{i=1}^{16} \text{ReLU}(-x_i) \tag{1}$$

  Indeed, this is one possible strategy that the hypernetwork produces.

- **The Signed Pudding Strategy** It turns out that there is another method of computing the L1 norm using ReLUs. The hypernetworks of the pudding strategy have learned to take advantage of the following fact:

$$||\boldsymbol{x}||_1 = 2\sum_{i=1}^{16} \text{ReLU}(\mp x_i) \pm \sum_{i=1}^{16} x_i \tag{2}$$

$$\approx \lim_{c\to\infty} 2\sum_{j=1}^{16} \text{ReLU}(\mp x_j \pm \sum_{i=1}^{16} x_i) + \sum_{j=1}^{32} \left(\text{ReLU}(c \pm \sum_{i=1}^{16} x_i) - c \right) \tag{3}$$

  which holds for both signs $\pm$ (hence the name "signed pudding"), where i iterates through input neurons and j through hidden neurons. The signed pudding strategy assigns one hidden neuron to each input neuron i to compute the first summation, and uses the leftover hidden neurons all compute the second summation term. Note that only one neuron is actually needed to compute the second term, and so this strategy can actually be implemented with $n + 1$ hidden neurons, which is more efficient than the $2n$ needed for the double-sided ReLU strategy. The pudding strategy is almost always implemented imperfectly, as in Equation 3. The hypernetwork uses a hardcoded assignment of pairs of hidden neurons to input neurons; changing the generation seed does not change the order of the hidden neurons. This is the most common strategy found in our experiments.

- **The Convexity Strategy** This is an imperfect random strategy that is easy for the hypernetwork to produce. This strategy notices that the L1 norm is a convex function and it tries to match the convexity using randomly oriented swish functions:

$$||\boldsymbol{x}||_1 \approx \alpha \sum_{j=1}^{48} \text{swish}\left(\sum_{i=1}^{16} W_{ij}^{(0)} x_i\right), \quad W_{ij}^{(0)} \sim D \tag{4}$$

with α some constant and D some distribution that is typically symmetric. Sometimes the distribution of D is unimodal, and sometimes it is bimodal.

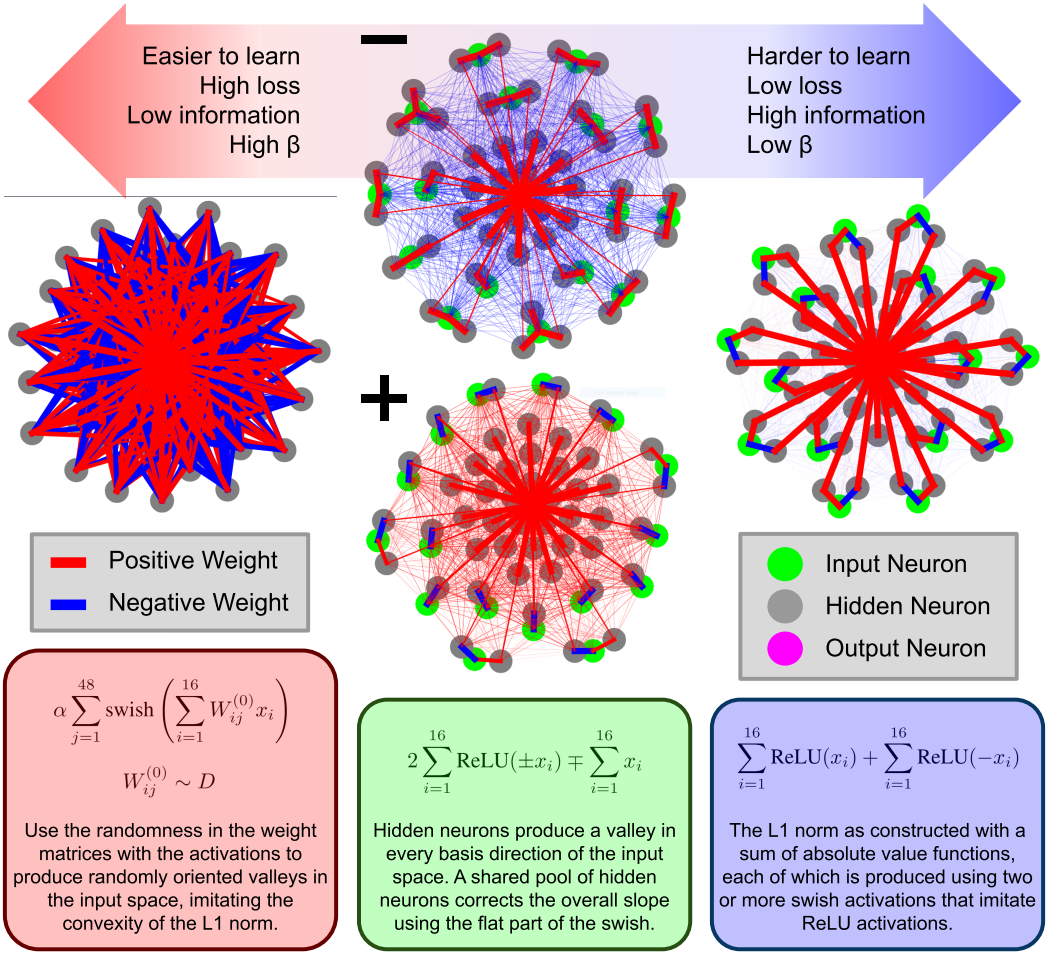


Figure 1: Three strategies used to compute the L1 norm. **Left:** the convexity strategy. **Center top:** the negative pudding strategy. **Center bottom:** the positive pudding strategy. **Right:** the double-sided ReLU strategy.

2.2 Order Parameters

Having identified that our experiments mainly consist of three strategies, we construct two order parameters which we use to distinguish the three strategies apart from one another.

Double Sidedness: Given the weight matrix $W \in \mathbb{R}^{n_0 \times n_1}$ for the first linear layer (bias not included) where n_0 is the number of input neurons and n_1 is the number of hidden neurons, the double sidedness order parameter α_1 is defined as:

$$\alpha_1 = \frac{\min_i \min(-\min_j W_{ij}, \max_j W_{ij})}{\underset{i,j}{\text{median}}\, \text{abs}(W_{ij})} \tag{5}$$

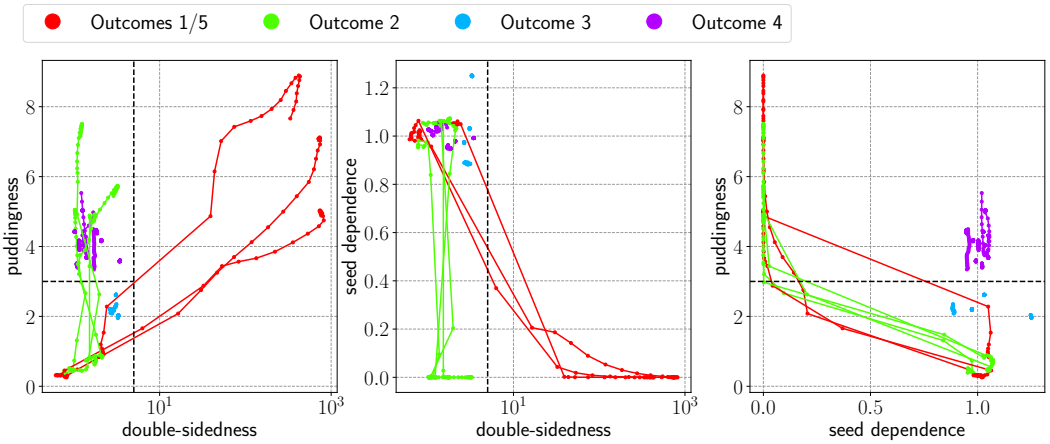


Figure 2: Networks that try to compute the L1 norm cluster into three general strategies in order parameter space. Lines are generated by sweeping β from 10^{-12} to 1 in 30 increments. The dotted lines represent hand picked boundaries which determine when phase transitions between strategies occur. Lines are grouped and colored by the phases where they start and end at.

The double sidedness measures the degree to which the solution uses the double-sided ReLU strategy.

Strongest Connection: The strongest connection order parameter α_2 is defined as:

$$\alpha_2 = \max_{i,j} \text{abs}(W_{ij}) \tag{6}$$

A weak strongest connection is indicative of the convexity solution.

Seed Dependence: The seed dependence order parameter α_3 is defined as:

$$\frac{||W - V||_F^2}{||W||_F^2 + ||V||_F^2} \tag{7}$$

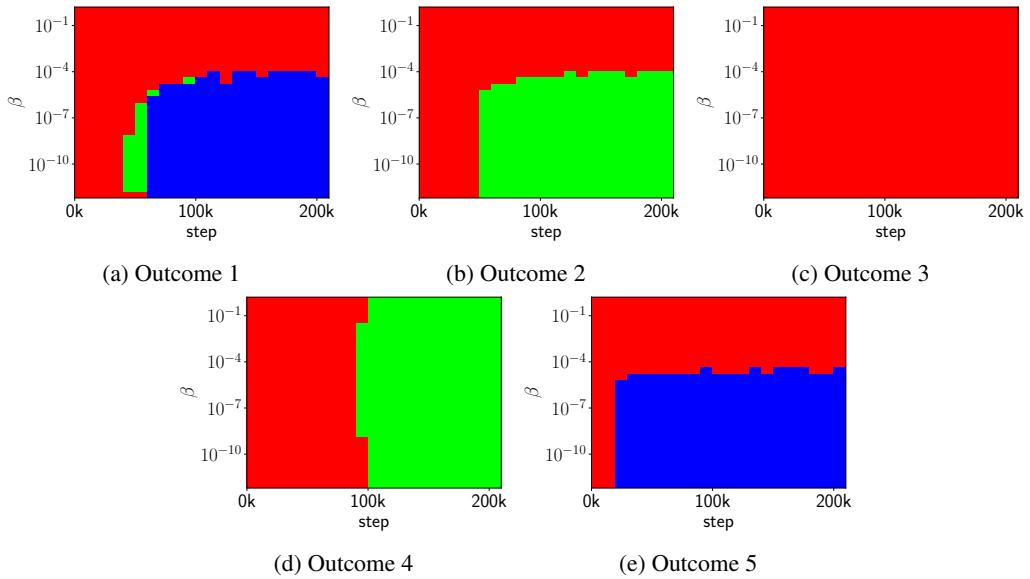
where W and V are weight matrices for the first linear layer (bias not included) generated with different randomization seeds using the same hypernetwork. A low seed dependence indicates that the hypernetwork is using memorized information about configurations of weights rather than generating this information randomly.

The networks divide themselves roughly into three clusters in order parameter space, each corresponding to one strategy, as shown in Figure 2. While the seed dependence is not immediately relevant since it does not differentiate between strategies, we will explain later that it can be used to determine how the hypernetwork constructs the pudding strategy.

2.3 Development of Strategies Throughout Training

Using the order parameters, we automatically classified the strategies which developed at various points during training for various β values, as in Figure 3. We find that the convexity strategy always develops first, and that other strategies differentiate away from there. The convexity strategy can evolve into either the pudding or double-sided ReLU, and the pudding sometimes also transitions into the double-sided ReLU. Transitions to the pudding strategy can happen either for low β or for all β. Oftentimes, the convexity strategy remains in the high β regime as the low β regime evolves, and the transition boundary's β value increases over time and then stabilizes.

It is worth noting that in Figure (3d) the hypernetwork becomes insensitive to β. In this case, the hypernetwork's accumulated KL divergence sums up to nearly zero for all β, causing the pareto frontier between loss and simplicity to collapse to a single point. This is not the case for the other cases that develop the pudding strategy; the KL divergence usually increases considerably for lower β. We believe that in case (3d), the hypernetwork randomly generataes an assignment of hidden neurons to input neurons, while in the other cases, the hypernetwork stores a memorized assignment



(a) Outcome 1 (b) Outcome 2 (c) Outcome 3

(d) Outcome 4 (e) Outcome 5

Figure 3: Six ways to develop the three strategies through training. The horizontal axis is the step number while the vertical axis is the β parameter used to generate the network. **Red:** convexity strategy. **Green:** pudding strategy. **Blue:** double sided ReLU strategy.

on the decoder side and passes it through to the encoder for output. Thus the hypernetwork accumulates KL in all cases except (3d). To test this hypothesis, we generated another network using only the decoder side of the hypernetwork without the encoder side[1], and evaluated its loss on the L1 problem again. Removing the encoder should prevent the hypernetwork from using any memorized assignments without affecting its ability to randomly generate an assignment. Indeed, Figure 4 shows that when we remove the encoder, hypernetworks in case (3d) are still able to construct working L1 networks that implement the pudding strategy but hypernetworks in other cases are not.

2.4 Generalization Capabilities

A key feature of the assignment generation hypothesis is that implies hypernetworks in case (3d) can generate L1 networks of different layer sizes, because all the configurations of weights are automatically randomly generated instead of being memorized specifically for the $(16, 48, 1)$ layer size structure. We can therefore use an existing hypernetwork to generate networks that compute the L1 norm in a wide range of dimensions, including dimensions larger than 16 which is what the hypernetwork was trained for. The hidden layer size can also be modified to be larger or smaller in the same way. Figure 5 shows that a large number of these L1 networks perform similarly to the original $(16, 48, 1)$ network. We find that there is a region of low loss which extends in the direction of increasing input dimension and hidden dimension, leading us to believe that the hypernetwork has found a general algorithm for computing L1 norms of vectors of any arbitrarily large size.

2.5 Adam's Strategy

We used Adam (Kingma & Ba, 2014) to train the same L1 norm network instead of generating the weights using a hypernetwork. The resulting network has a lower loss than the networks generated via hypernetwork, but it is much more difficult to interpret. This is already visible in the cleanliness of the visualization for the hypernetwork in comparison to Adam, as shown in Figure 6.

Since the Adam-trained L1 norm network is much harder to interpret, we only have a hypothesis about how it computes the L1 norm, which is as follows. In the Adam-trained L1 norm network, hidden neurons are split into two groups: some that are assigned to a single input neuron, and the leftover hidden neurons that are attached to all input neurons. Each input neuron then uses any of

[1] by drawing latents from the distribution defined by the decoder side instead of the encoder side

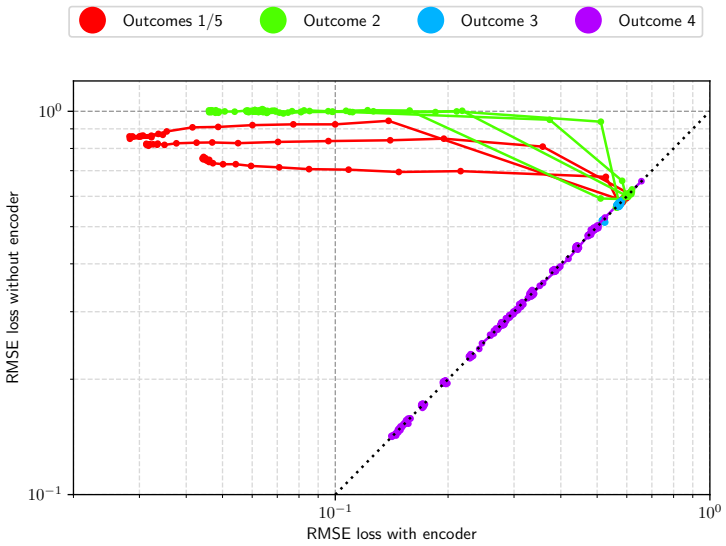


Figure 4: Loss of neural network generated when the encoder is either used or ignored. The black dotted line indicates when the performance is unaffected by the presence of the encoder, ie. when the encoder is unused during the weight generation process. Lines are generated by sweeping β from 10^{-12} to 1 in 30 increments.

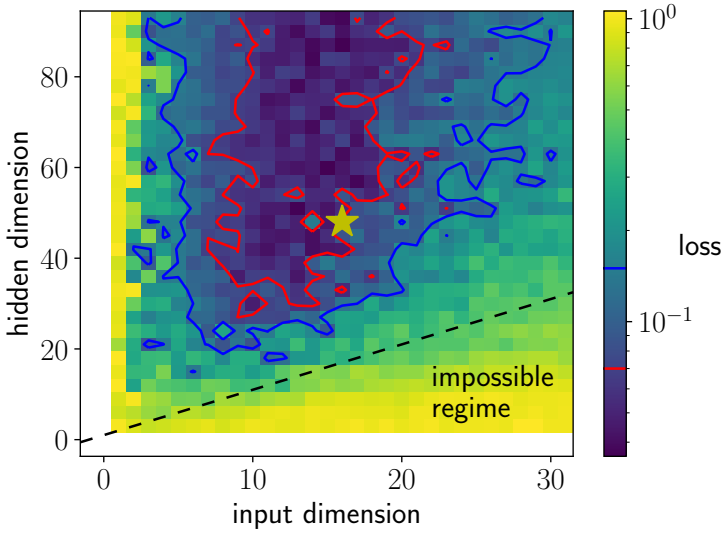
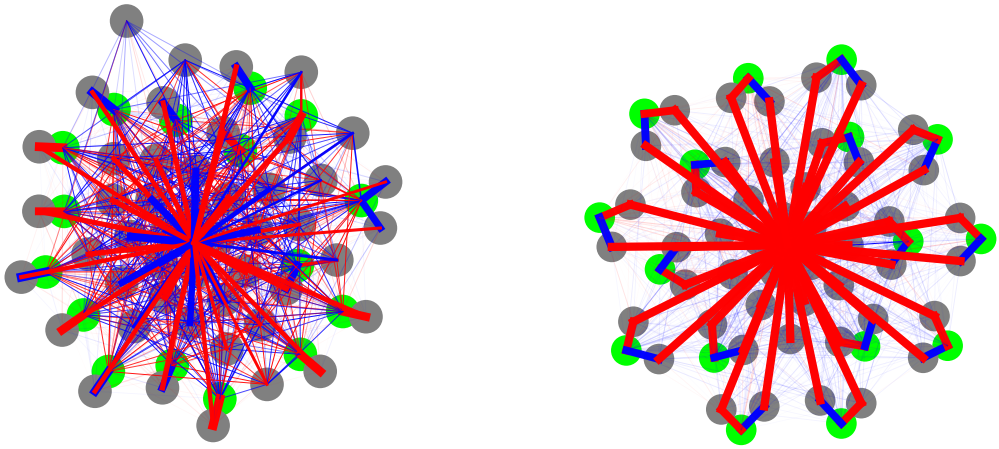


Figure 5: Loss of neural network generated with various input dimensions and hidden dimensions using a hypernetwork. The red contour denotes a loss of 0.07 while the blue denotes a loss of 0.15. The hypernetwork was only trained to generate networks with input dimension 16 and hidden dimension 48 (yellow star), yet it can produce networks of many different shapes which all compute the L1 norm with reasonable accuracy.



(a) Trained by Adam.

(b) Generated via hypernetwork.

Figure 6: Visualization of a $(16, 48, 1)$ neural networks trained to compute the L1 norm of a vector. Input neurons in green, output neurons in magenta. Positive weights in red, and negative weights in blue. The Adam-trained network is messy whereas the hypernetwork-generated network is much easier to interpret.

its assigned hidden neuron(s) to create a kink of appropriate difference in slope between the positive and negative sides, and the leftover hidden neurons are used to correct the average slope, forming an absolute value function for that input neuron. The absolute value functions for each input neuron are then added together. This is like the pudding strategy except that there can be many hidden neurons assigned to one input neuron (eg. 5 hidden neurons, $+++--$) and that the corresponding weights can differ in their sign. Signs for all weights connecting to any given leftover hidden neuron are randomized, and each input neuron connects to every leftover hidden neuron with a different random strength, provided that all the weights satisfy the constraints above.

The network trained by Adam is unnecessarily complex and variable due to the randomness involved, whereas a much simpler and more interpretable solution exists that can be found via hypernetworks.

3 METHOD

In this section, we seek to provide a rough understanding of how we use our hypernetwork to generate weights and why we expect the resulting networks to be interpretable. Most important is the fact that the hypernetwork's outputs are used as the weights of the neural network (Chauhan et al., 2023). Altogether, our hypernetwork architecture consists of two graph transformers (Ying et al., 2021) which operate on the computation graph of the target network, which serve as the encoder and decoder side of a hierarchical variational autoencoder (HVAE) (Vahdat & Kautz, 2020; Sønderby et al., 2016; Child, 2020). The weights of the hypernetwork are generated by a Pareto hyperhypernetwork (Navon et al., 2020) which receives the HVAE β hyperparameter (Higgins et al., 2017) as input. The exact details of our hypernetwork and hyperhypernetwork architectures can be found in Appendix A.2. Since our hypernetwork is a merge between a graph transformer and a HVAE, there are multiple lenses through which we understand our hypernetwork design:

The Hypernetwork Interpretation. The hypernetwork is a neural network whose output is used as the weights for a neural network (Chauhan et al., 2023). For input, the hypernetwork is given information about every weight it needs to generate, such as the layer number and indices to identify the input and output neurons that it is connected to. The hypernetwork can then learn a general process for configuring each individual weight depending on it's location within the network. Simpler configurations are easier to learn and thus the resulting networks tend to be simpler and thus more interpretable.

The GNN Interpretation. A neural network is a computation graph, so the most natural way to manipulate weight data is through a graph neural network (GNN) (Wu et al., 2022; Scarselli et al., 2009; Sanchez-Lengeling et al., 2021; Daigavane et al., 2021). The hypernetwork is a graph transformer, whereby information is stored at the edges and is sent to and from adjacent nodes where attention heads operate. This allows the hypernetwork to compute based on how weights are mutually related to one another within the architecture (which itself does not need to be fixed either). This allows the hypernetwork to form structures of weights which are connected to the way the architecture is structured and are thus more likely to be interpretable.

The MDL/Compression Interpretation. Our hypernetwork an application of the minimum description length (MDL) principle, which treats learning as a data compression procedure described by a mathematical version of Occam's razor (Grünwald, 2007; Grünwald & Roos, 2019; Rissanen, 1978; Solomonoff, 2009). The MDL principle treats a neural network as a compressor, so that we can speak of the "Occam simplicity" of a model through KL divergences, which measure how well that model compresses a dataset in an information theoretic sense. (Chaitin, 1975; Polyanskiy & Wu) For our case the dataset consists of the neural network weights and the compressor is the hypernetwork; the hypernetwork is an encoding/decoding system for compressing neural network weights into as simple a latent representation as possible, and networks derived from simpler representations are more interpretable.

The Generative Model Interpretation. Our hypernetwork is a generative model for data in the form of neural network weights. In the past, generative models were developed for and have been successful in generating human-interpretable forms of data, such as natural language (OpenAI, 2023; Touvron et al., 2023) and images (Ho et al., 2020). Thus we may expect that a generative model for data in the form of neural network weights would naturally have an inductive bias for human-interpretable weights.

4 Related Work

Our work is primarily an application of the minimum description length (MDL) principle, which treats learning as a data compression procedure described by a mathematical version of Occam's razor (Grünwald, 2007; Grünwald & Roos, 2019; Rissanen, 1978; Solomonoff, 2009). The MDL principle treats a neural network as a compressor, so that we can speak of the "Occam simplicity" of a model through KL divergences, which measure how well that model compresses a dataset in an information theoretic sense. (Chaitin, 1975; Polyanskiy & Wu)

While we were building a neural network compressor, we noticed that most machine learning systems compress neural network weights with techniques (Tibshirani, 1996; Frankle & Carbin, 2018; Tan & Le, 2019; White et al., 2023; Redmon & Farhadi, 2018) that are different to how they compress data from a dataset (Kingma & Welling, 2013; Kobyzev et al., 2020; Vahdat & Kautz, 2020; Sønderby et al., 2016; Child, 2020; Vaswani et al., 2017; OpenAI, 2023; Touvron et al., 2023). When we tried to use techniques for data compression to compress weights instead, it turned out we were constructing systems that are commonly known as hypernetworks (Chauhan et al., 2023).

Because of the MDL principle, this means our hypernetwork is trying to find weights which are as simple as possible for solving the task. An artifact of the simplicity of the weights is that they should be easier for humans to pick apart: the resulting model should be very interpretable, and this fact is the main driving force of this paper.

Research on interpretability helps us explain how neural networks operate at the individual neuron level, so that we can understand why they produce certain outputs. This lets us build safer models that we can have better trust in for applications that need this trust. Landmark works in interpretability have included the discovery of polysemantic neurons (Scherlis et al., 2022; O'Mahony et al., 2023), high-low frequency detectors (Schubert et al., 2021), edge detectors, circular representations of numbers (Power et al., 2022), and tokenizers and detokenizers.

Right now, most hypernetwork research focuses on how we can use them to predict different things about a model in a particular setting without having to train it in that setting, like it's loss, accuracy, or trained parameters (Zhang et al., 2018) (Knyazev et al., 2021). This information can be used to quickly design a faster, smaller, more performant architecture to train, skip some training steps (Knyazev et al., 2023), adjust the parameters based on different loss functions (Navon et al., 2020),

and more. Our work instead uses hypernetworks to compress and generate data coming in the form of neural network weights.

Training hypernetworks typically involves computing hypergradients (Baydin et al., 2017), and our work is no exception to this.

Our hypernetwork's architecture is partially based on Graph Neural Networks, self-attention, and deep hierarchical VAEs, all in combination.

5 DISCUSSION

In this paper, we have introduced a novel hypernetwork-based method for constructing neural network weights which makes them mechanistically interpretable. We then used this method to construct networks which compute the L1 norm and mechanistically interpreted them.

We found that the hypernetwork-generated L1 norm networks implement three main algorithms for computing the L1 norm, and they represent different tradeoffs between their errors and the model simplicity. This tradeoff can be manipulated by the β hyperparameter, which controls the relative weight of error vs simplicity. The algorithm most expected by humans is the double-sided ReLU algorithm, which is the hardest to learn and the most accurate. We constructed three order parameters, two of which we use to automatically classify neural networks according to the three strategies. We find that the three strategies develop in different β regimes and at different times during training. Namely, the convexity strategy is the easiest to learn, simplest, and the one that develops at greatest β, followed by the pudding strategy, followed by the double-sided ReLU strategy. The pudding strategy is even more simple than the expected double-sided ReLU in that it can be used to compute the L1 norm with fewer hidden neurons than what the authors originally thought was possible.

There is also great value in developing explanations for how the hypernetworks themselves learn to build neural networks. We find that the hypernetworks develop two main methods for constructing networks which operate via pudding strategy: one which constructs a random assignment of hidden neurons to input neurons and another which uses a memorized assignment whose information content is penalized. We demonstrate that hypernetworks which construct random assignments can be used to generate working L1 networks to operate on different, sometimes even larger input sizes and hidden dimensions. This works completely in inference time without any retraining. The hypernetwork's ability to generalize to other input dimensions signals that the hypernetwork has learned an algorithm for computing L1 norms in general for any input vector size, rather than just a circuit that computes an L1 norm with a fixed input size that cannot generalize to other sizes. We hope that analysis of hypernetwork behavior of this type can reveal similarly general algorithm-learning behaviors in other problems.

6 CONCLUSION

In our search for the simplest algorithmic problem with an unintuitive learned solution, we have found that the task of computing the L1 norm creates a diversity of interesting, unique, and interpretable neural solutions and behaviors which emerge in different ways. We show the hypernetwork to be a valuable tool in the training of interpretable networks, as networks trained by Adam are much more difficult to deconstruct. The resulting mechanistic descriptions of L1 norm networks exemplify the way in which neural networks can find seemingly alien solutions to even the most simple of problems, begging us to develop a more complete understanding of the development of algorithmic phases to complement the algorithmic solutions themselves.

7 ACKNOWLEDGEMENTS

I would like to extend special thanks to Ziming Liu and Max Tegmark for the excellent help they have given on the technical aspects of this work, as well as the support and training they have given me through their mentorship and advising.

REFERENCES

Michael J Bannister, David Eppstein, Michael T Goodrich, and Lowell Trott. Force-directed graph drawing using social gravity and scaling. In *Graph Drawing: 20th International Symposium, GD 2012, Redmond, WA, USA, September 19-21, 2012, Revised Selected Papers 20*, pp. 414–425. Springer, 2013.

Atilim Gunes Baydin, Robert Cornish, David Martinez Rubio, Mark Schmidt, and Frank Wood. Online learning rate adaptation with hypergradient descent. *arXiv preprint arXiv:1703.04782*, 2017.

Gregory J. Chaitin. A theory of program size formally identical to information theory. *J. ACM*, 22 (3):329–340, jul 1975. ISSN 0004-5411. doi: 10.1145/321892.321894. URL https://doi.org/10.1145/321892.321894.

Vinod Kumar Chauhan, Jiandong Zhou, Ping Lu, Soheila Molaei, and David A Clifton. A brief review of hypernetworks in deep learning. *arXiv preprint arXiv:2306.06955*, 2023.

Rewon Child. Very deep vaes generalize autoregressive models and can outperform them on images. *arXiv preprint arXiv:2011.10650*, 2020.

Ameya Daigavane, Balaraman Ravindran, and Gaurav Aggarwal. Understanding convolutions on graphs. *Distill*, 2021. doi: 10.23915/distill.00032. https://distill.pub/2021/understanding-gnns.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

Peter Grünwald and Teemu Roos. Minimum description length revisited. *International journal of mathematics for industry*, 11(01):1930001, 2019.

Peter Grünwald. *The Minimum Description Length Principle*. 01 2007. ISBN 9780262256292. doi: 10.7551/mitpress/4643.001.0001.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-VAE: Learning basic visual concepts with a constrained variational framework. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=Sy2fzU9gl.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Boris Knyazev, Michal Drozdzal, Graham W Taylor, and Adriana Romero Soriano. Parameter prediction for unseen deep architectures. *Advances in Neural Information Processing Systems*, 34:29433–29448, 2021.

Boris Knyazev, Doha Hwang, and Simon Lacoste-Julien. Can we scale transformers to predict parameters of diverse imagenet models? *arXiv preprint arXiv:2303.04143*, 2023.

Stephen G Kobourov. Spring embedders and force directed graph drawing algorithms. *arXiv preprint arXiv:1201.3011*, 2012.

Ivan Kobyzev, Simon JD Prince, and Marcus A Brubaker. Normalizing flows: An introduction and review of current methods. *IEEE transactions on pattern analysis and machine intelligence*, 43 (11):3964–3979, 2020.

Aviv Navon, Aviv Shamsian, Gal Chechik, and Ethan Fetaya. Learning the pareto front with hypernetworks. *arXiv preprint arXiv:2010.04104*, 2020.

Laura O'Mahony, Vincent Andrearczyk, Henning Müller, and Mara Graziani. Disentangling neuron representations with concept vectors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 3769–3774, 2023.

OpenAI. Gpt-4 technical report, 2023.

Yury Polyanskiy and Yihong Wu. Information theory: From coding to learning. preprint on webpage at `https://people.lids.mit.edu/yp/homepage/papers.html`.

Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. Grokking: Generalization beyond overfitting on small algorithmic datasets. *arXiv preprint arXiv:2201.02177*, 2022.

Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978. ISSN 0005-1098. doi: https://doi.org/10.1016/0005-1098(78)90005-5. URL `https://www.sciencedirect.com/science/article/pii/0005109878900055`.

Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. A gentle introduction to graph neural networks. *Distill*, 2021. doi: 10.23915/distill.00033. https://distill.pub/2021/gnn-intro.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

Adam Scherlis, Kshitij Sachan, Adam S Jermyn, Joe Benton, and Buck Shlegeris. Polysemanticity and capacity in neural networks. *arXiv preprint arXiv:2210.01892*, 2022.

Ludwig Schubert, Chelsea Voss, Nick Cammarata, Gabriel Goh, and Chris Olah. High-low frequency detectors. *Distill*, 2021. doi: 10.23915/distill.00024.005. https://distill.pub/2020/circuits/frequency-edges.

Ray J. Solomonoff. *Algorithmic Probability: Theory and Applications*, pp. 1–23. Springer US, Boston, MA, 2009. ISBN 978-0-387-84816-7. doi: 10.1007/978-0-387-84816-7_1. URL `https://doi.org/10.1007/978-0-387-84816-7_1`.

Casper Kaae Sønderby, Tapani Raiko, Lars Maaløe, Søren Kaae Sønderby, and Ole Winther. Ladder variational autoencoders. *Advances in neural information processing systems*, 29, 2016.

Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pp. 6105–6114. PMLR, 2019.

Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996. ISSN 00359246. URL `http://www.jstor.org/stable/2346178`.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Arash Vahdat and Jan Kautz. Nvae: A deep hierarchical variational autoencoder. *Advances in neural information processing systems*, 33:19667–19679, 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. Neural architecture search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727*, 2023.

Lingfei Wu, Peng Cui, Jian Pei, and Liang Zhao. *Graph Neural Networks: Foundations, Frontiers, and Applications*. Springer Singapore, Singapore, 2022.

Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform badly for graph representation? *Advances in Neural Information Processing Systems*, 34:28877–28888, 2021.

Zheng Yuan, Hongyi Yuan, Chuanqi Tan, Wei Wang, and Songfang Huang. How well do large language models perform in arithmetic tasks? *arXiv preprint arXiv:2304.02015*, 2023.

Chris Zhang, Mengye Ren, and Raquel Urtasun. Graph hypernetworks for neural architecture search. *arXiv preprint arXiv:1810.05749*, 2018.

A METHOD

A.1 FORCE-DIRECTED GRAPH DRAWINGS

Throughout this paper we use a force-directed graph drawing algorithm to visualize neural networks as computation graphs. Force-directed graph drawing algorithms try to position every node in the plane to minimize clutter and account for several visual quality measures, such as edge overlaps, drawing area, and symmetry. Typically, such an algorithm defines an energy consisting of a sum of several components which each depend on the node positions, and the node positions are adjusted to minimize this energy via gradient descent. In our case, we apply three components: $1/r$ pairwise repulsion between all neurons, kr^2 pairwise attraction for weights of absolute value k, and r^2 attraction pulling all neurons towards the origin.

This graph drawing algorithm will help us to observe the structure of weights in the neural networks which we train in this paper, so that we may more easily understand how they function. Most importantly it allows us to observe modularity, which is when individual parts of a learned neural network compute their operations individually without interaction. This is because the modules may form separate connected components which stay self-connected but repel each other in the drawn graph, making components easy to identify.

One of the main issues with force-directed graph drawing is that the gradient descent often falls into local minima of the energy, since two separate modules in the network can become tangled up, after which point the modules can no longer slide past one another and separate properly. To remedy this, we position the nodes in four dimensions instead of two (this provides mode connectivity), and we apply an increasingly strong decay the two excess dimensions during gradient descent until they disappear, leaving a fully two-dimensional arrangement. This arrangement can be rescaled as needed for the visualization.

A.2 ATTENTIONAL HYPERNETWORKS

In this section, we explain in more detail how we use a hypernetwork to generate the weights of the MLP which we would like to train. Instead of learning the MLP weights directly, we learn the "hyperweights" of this hypernetwork. We generate the MLP weights every iteration as part of the forward pass when doing gradient descent.

We design the hypernetwork in a way so that it has an inductive bias to create certain structures and formations of weights with more ease than others. For example, we may want the hypernetwork to tell the weights how to self-organize into many duplicates of a specific circuit, which are then connected together in a formation, rather than many unrelated circuits connected in a more complex manner. Given that these structures are organized, with a small number of hyperweights it might be easier to learn a method of constructing such structures rather than the structures themselves. We believe that and inductive bias for learning such methods should best arise from clever forms of parameter reuse, just like how convolutional filters are best for translationally equivariant computations. As such, our hypernetwork operates much like a graph transformer, whose computations are duplicated across all nodes and edges.

For every component of the MLP, there is an analogous component for our hypernetwork, which itself can somewhat be thought of like an MLP. For example, the hypernetwork has hyperfeatures, hyperlayers, hyperwidth, hyperdepth, and hyperactivations in the same way that the MLP (or "network") has features, layers, width, depth, and activations.

For our network architecture, we restrict ourselves to a neural network consisting of weights $W^{(\ell)} \in \mathbb{R}^{n_{i+1} \times n_i}$ and biases $b^{(\ell)} \in \mathbb{R}^{n_{i+1}}$ with N layers:

$$a_j^{(\ell+1)} = \sigma(b_j^{(\ell)} + \sum_i W_{ij}^{(\ell)} a_i^{(\ell)})$$

with $a_i^{(1)}$ the input vector and $b_j^{(N)} + \sum_i W_{ij}^{(N)} a_i^{(N)}$ the output vector.

We will now describe our hypernetwork architecture. The fundamental unit of data processed by the hypernetwork is a high-dimensional ragged tensor of "hyperactivations" containing information about every weight in the network. Each "hyperlayer" operates by applying many operations which

serve to perform computations that only conduct information along individual axes of the tensor. For example, if we have a hyperactivation tensor pertaining to a 4 dimensional CNN filter tensor with an x axis, we might apply a blur filter along the corresponding x axis of the hyperactivation tensor while treating all other dimensions as batch dimensions, duplicating this operation for all indices.

Returning from the CNN example to our MLP network, we designate the last axis as special and we call the "hyperfeature" dimension, analogous to the feature dimension in the MLP. The hyperactivation tensor is sliced into many blocks along the hyperfeature axis, and each block undergoes its own operation along its own designated axis (all other axes treated as batch dimesions), then the results are concatenated back together in the hyperfeature dimension. A linear operation in the hyperfeature axis can then be used to control the movement of data between these individual blocks, allowing for movement of data in every direction of the tensor. A good analogy is the way that trains can be shunted onto different tracks where they may travel in different directions or undergo different procedures. This all can then be repeated multiple times by stacking together multiple of these hyperlayers. We will index hyperlayers using the variable ℓ' since we are indexing layers with ℓ, and let us call the number of hyperlayers the "hyperdepth" N', which we set to 4.

We will now roughly describe how we construct a hyperactivation tensor from the MLP weights. We can summarize the MLP weights using a ragged tensor $W_{ij}^{(\ell)}$ indexed by input neuron i, output neuron j, and layer ℓ. The hyperactivation tensor then has the same shape, except that it has an additional hyperfeature axis indexed by a variable i'.

The operations performed on blocks of the hyperactivation tensor are treated like activation functions for the hypernetwork. For a hyperactivation tensor with indices i, j, ℓ, and i', a block is processed with each of the following operations:

- $x \to \sigma(x)$. Elementwise activation function. Used on a block of 20 hyperfeatures.
- Nothing $\to$ Positional encoding of i if $\ell = 1$, else a zero tensor.
- Nothing $\to$ Positional encoding of j if $\ell = N$, else a zero tensor.
- Nothing $\to$ Positional encoding of ℓ.
- Nothing $\to$ 5 hyperfeatures of random i.i.d. samples from a standard normal distribution.
- A self-attention head for every neuron in the network. Each edge feeds 3 sub-blocks of the hyperactivation tensor—the keys, queries, and values—to the neuron in front and another 3 sub-blocks to the neuron behind, and concatenates the results received from both sides. The keys, queries, and values are all 5 hyperfeatures in size.

The output of the final hyperlayer has two hyperfeatures—one is used as the weight tensor and the other is averaged along the input neuron axis i and is used as the bias tensor.

Notice that this hypernetwork architecture as it stands does not contain that many learned parameters, compared to how many MLP parameters it can generate. Even with so few parameters, it still takes a huge amount of computation, which is necessary for the amount of parameter reuse we want. Remember that parameter reuse is useful for the hypernetwork to produce highly patterned structures in the network's weights.

A.2.1 KL Attentional Hypernetworks

Plain attentional hypernetworks like the one described above have a certain defect: there might be some networks that are learned more easily by gradient descent but which are hard for the hypernetwork to capture because they are not structured in any obvious fashion. To allow the attentional hypernetwork to capture these circuits, we introduce another axis to the hyperactivation tensor, which we call the KL axis.

The KL axis has two indices, representing the encoder and decoder side of an information channel. Information is passed through this axis via two operations:

- Nothing $\to$ 5 hyperfeatures of learned variables which go along with the hyperweights during training. These hyperfeatures are intended to capture information that can be learned by standard gradient descent, but they are only given on the encoder side and are set to zero for the decoder side.

- Start with two blocks of 4 hyperfeatures representing μ and σ values for normal distributions. The total KL divergence $D_{\text{KL}}(q||p)$ between the encoder-side and decoder-side distributions q and p is computed and added to an accumulator variable. The output is 4 hyperfeatures of samples from q; a copy of these samples for each the encoder and decoder sides to use.

The weights and biases are constructed using only the output from the decoder side.[2]

In this additional axis, learning via standard gradient descent is allowed to take place, since the hypernetwork can pass the learned variables provided to the encoder through the KL channel to the decoder, where the variables can be output to be treated as weights and biases. But notice that any information passing through the KL channel has its information content measured and accounted for in an accumulator variable. This means we can suppress the usage of gradient descent strategies by regularizing the quantity of raw weight information which passes through. [3] Tuning the suppression factor β allows us to control the balance between a hypernetwork which only designs very structured patterns of weights and a hypernetwork which regurgitates unstructured patterns of learned and memorized weights. The balance between loss and KL implicitly encourages the neural network to develop structures like modules and duplicated circuits of neurons, as these are structures that the hypernetwork can encode in a more compressed manner and will be penalized less for.

A.2.2 HYPERHYPERNETWORKS FOR MULTI-OBJECTIVE OPTIMIZATION

We are now left with a multi-objective optimization problem where we would like to jointly optimize for the network's loss and hypernetwork's total accumulated KL. To solve this, we use a hyperhypernetwork to generate the hyperweights in every step during the forward pass, using something like a Pareto Hypernetwork. This hyperhypernetwork takes $\log \beta$ (rescaled and shifted) as input, has two hidden layers of size 100 and 10 with swish activation, and outputs two vectors a and b where $a\sigma(b)$ is treated as the flattened vector of hyperweights. At every iteration, we sample β from a distribution, use the hyperhypernetwork to generate the hyperweights, use the hypernetwork to generate the weights, and use $\log(L + \beta D_{\text{KL}})$ as the objective, where L is the network's loss and D_{KL} is the hypernetwork's accumulated KL.

[2] By introducing the KL axis, we have essentially turned our hypernetwork into a kind of conditional hierarchical VAE.

[3] Note that the feedback in the channel lets the hypernetwork perform simpler computations without destroying the ELBO bound in the VAE interpretation of our hypernetwork.

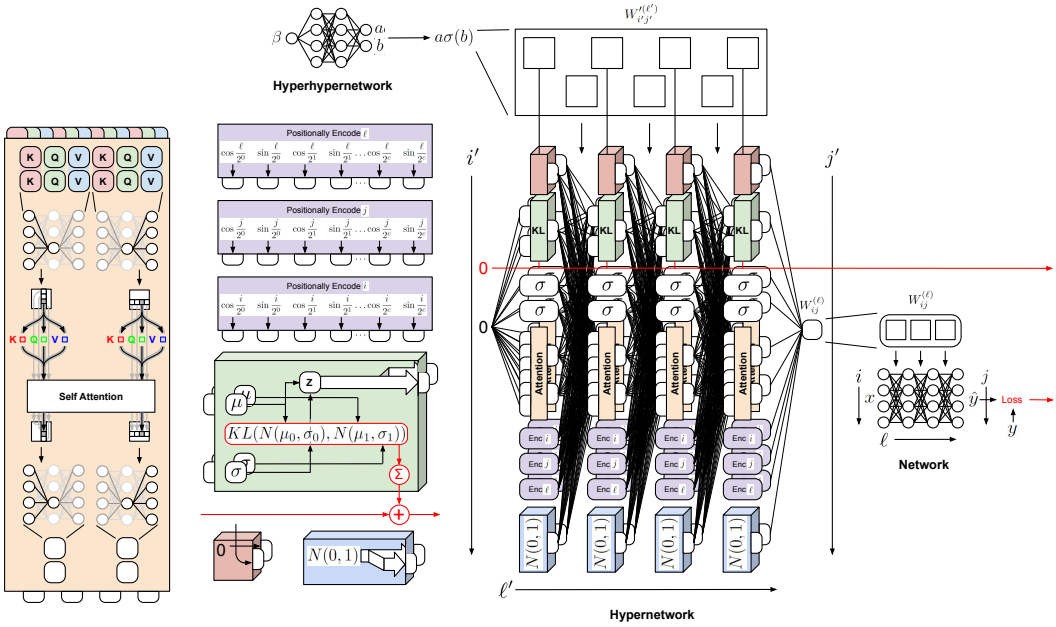


Figure 7: Architecture