Generalisation in Bitwise XOR and the Effect of Weight Decay

Ethan Tan

September 29, 2022

1 Introduction

Inspired by this paper and this blog post, I trained a 1-layer MLP to learn bitwise XOR, with the purpose of investigating generalisation, interpretation of simple models, and the effect of regularisation. After repeatedly training models using AdamW optimisation and weight decay, I found that the MLP consistently learned the same algorithm (involving learning rows of a Hadamard matrix), which I was able to reverse-engineer.

The XOR problem and 1L MLP architecture were chosen to make analysis and interpretation of the model simple. Both models that generalise successfully as well as models trained on the entire dataset find the same algorithm. Due to the nature of this algorithm, it is easy to test to what extent a model has learned this — for instance, unregularised models perform worse on valuation data after training than at initialisation, but do in fact learn some identifiable generalising patterns. This may make XOR 1L MLP a useful framework for other experiments.

2 Method

2.1 Data

A model is trained to predict the bitwise XOR of two integers in $\{0, \ldots, 2^n - 1\}$. All numbers are converted to vectors of length 2^n via one-hot encoding, and the inputs are concatenated, so that the model has $2 \cdot 2^n$ inputs and 2^n outputs.

For the results shown in this document, I used $n = 5$, with 350 of the 1024 possible pairs used as training data, and the remaining 674 used as validation. 350 was chosen manually so that training loss would decay to 0 much faster than validation loss. Similar results to those in this document can be reproduced with larger n.

2.2 Model Architecture

I use a 1L softmax regression MLP with 48 hidden neurons, AdamW optimisation, and weight decay. I used a learning rate of 2×10^{-3} and a weight decay of 1 for most of my results, though I also train a model with no weight decay for comparison.

3 Results

3.1 Learned Algorithm

I now describe the model's algorithm for predicting XOR, which I reverse-engineered using the model weights (explained further in 3.2).

The general outline is as follows: for each hidden neuron, the model defines a linear map $\phi : \mathbb{F}_2^n \to \mathbb{F}_2$, then evaluates ϕ on both inputs $\mathbf{c}, \mathbf{d} \in \mathbb{F}_2^n$ (treating $\{0, \dots, 2^n - 1\}$ as $\mathbb{F}_2^n$). The neuron chooses an element $(z_1, z_2) \in \mathbb{F}_2^2$, and then outputs a positive nonzero value if and only if $(\phi(\mathbf{c}), \phi(\mathbf{d})) = (z_1, z_2)$.

Since XOR $\oplus$ is simply addition in $\mathbb{F}_2^n$, we have $\phi(\mathbf{c} + \mathbf{d}) = \phi(\mathbf{c}) + \phi(\mathbf{d})$, and so if the neuron outputs 1, we know that $\phi(\mathbf{c} + \mathbf{d}) = z_1 + z_2$. This eliminates half the possible outputs, so the neuron sends a positive value to all output neurons corresponding to $\mathbf{y} \in \mathbb{F}_2^n$ for which $\phi(\mathbf{y}) = z_1 + z_2$, and a negative value to the other output neurons. (These values are approximately constant in magnitude, both across output neurons and between hidden neurons.) Observe that a positive value is always sent to the neuron corresponding to the correct answer.

With enough hidden neurons, the XOR of the two inputs receives a positive value from every hidden neuron, whereas all other outputs receive some negative values. Upon summation, the correct answer has a higher value than all other outputs, and is predicted when taking softmax.

More precisely, the process is as follows:

Consider a pair $(a, b) \in \{0, \dots, 2^n - 1\}^2$. When represented in one-hot encoding, we obtain the vector $\mathbf{x} = \mathbf{e}_a + \mathbf{e}_{2^n + b}$.

Let $\mathbf{w} \in \mathbb{R}^{2^{n+1}}$ be a row of the $(n+1)^{\text{th}}$ Hadamard matrix (specifically, the Walsh matrix) H_{n+1}. We write $\mathbf{w} = (\mathbf{s}, \mathbf{t}) \in \mathbb{R}^{2^n} \times \mathbb{R}^{2^n}$; note that we either have $\mathbf{s} = \mathbf{t}$ or $\mathbf{s} = \mathbf{1} - \mathbf{t}$.

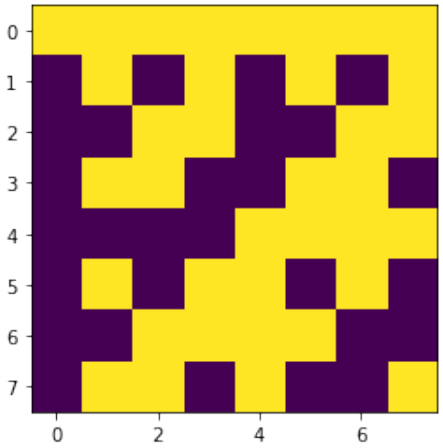

Figure 1: The matrix H_3. Yellow corresponds to 1, purple to -1.

Computing $\mathbf{w} \cdot \mathbf{x}$, we have $\mathbf{e}_a \cdot \mathbf{s} + \mathbf{e}_b \cdot \mathbf{t}$. Let us index the 2^n coordinates of $\mathbb{R}^{2^n}$ by elements of $\mathbb{F}_2^n$, by mapping $m \in \{0, \ldots, 2^{n-1}\}$ to $\psi(m) \in \mathbb{F}_2^n$, its length n binary representation. Since $\mathbf{s}$ is a row of H_n, there exists a vector $\mathbf{v}_s \in \mathbb{F}_2^n$ such that the i^{th} coordinate of $\mathbf{s}$ is 1 if $\psi(i) \cdot \mathbf{v}_s = 1 \in \mathbb{F}_2$, and -1 otherwise.

Let $\mathbf{c} = \psi(\mathbf{a}), \mathbf{d} = \psi(b)$. We then have that $\mathbf{e}_a \cdot s = 1 \iff \mathbf{c} \cdot \mathbf{v}_s = 1$. Suppose that $\mathbf{s} = \mathbf{t}$. Then $\mathbf{w} \cdot \mathbf{x} = \mathbf{e}_a \cdot \mathbf{s} + \mathbf{e}_b \cdot \mathbf{s}$. Now

$$\mathbf{w} \cdot \mathbf{x} = \begin{cases} 2 & \mathbf{c} \cdot \mathbf{v}_s = \mathbf{d} \cdot \mathbf{v}_s = 1, \\ -2 & \mathbf{c} \cdot \mathbf{v}_s = \mathbf{d} \cdot \mathbf{v}_s = -1, \\ 0 & \text{otherwise.} \end{cases}$$

Using Iverson bracket notation (i.e. $[\cdot]$ is an indicator function), we have the equations

$$\begin{aligned} \text{ReLU}(\mathbf{w} \cdot \mathbf{x}) &= 2[\mathbf{c} \cdot \mathbf{v}_s = \mathbf{d} \cdot \mathbf{v}_s = 1] \\ &= 2[(\mathbf{c} \cdot \mathbf{v}_s) \wedge (\mathbf{d} \cdot \mathbf{v}_s)] \qquad (1) \\ \text{ReLU}(-\mathbf{w} \cdot \mathbf{x}) &= 2[\mathbf{c} \cdot \mathbf{v}_s = \mathbf{d} \cdot \mathbf{v}_s = -1] \\ &= 2[(\neg(\mathbf{c} \cdot \mathbf{v}_s)) \wedge (\neg(\mathbf{d} \cdot \mathbf{v}_s))] \qquad (2) \end{aligned}$$

Similarly, if $\mathbf{s} = \mathbf{1} - \mathbf{t}$, we obtain

$$\begin{aligned} \text{ReLU}(\mathbf{w} \cdot \mathbf{x}) &= 2[\mathbf{c} \cdot \mathbf{v}_s = 1, \mathbf{d} \cdot \mathbf{v}_s = -1] \\ &= 2[(\mathbf{c} \cdot \mathbf{v}_s) \wedge (\neg(\mathbf{d} \cdot \mathbf{v}_s))] \qquad (3) \\ \text{ReLU}(-\mathbf{w} \cdot \mathbf{x}) &= 2[\mathbf{c} \cdot \mathbf{v}_s = \mathbf{d} \cdot \mathbf{v}_s = -1] \\ &= 2[(\neg(\mathbf{c} \cdot \mathbf{v}_s)) \wedge (\mathbf{d} \cdot \mathbf{v}_s)] \qquad (4) \end{aligned}$$

Each hidden neuron tends to learn (a multiple of) either a row of H_{n+1}, or the negation thereof, so its output corresponds to one of the four equations given above. We consider the case when the first of these equations hold, since the other three are similar.

For some $\alpha > 0$, a hidden neuron with weight $\beta\mathbf{w}$ (with output denoted o) sends αo to every output corresponding to $\mathbf{y} \in \mathbb{F}_2$ where $\mathbf{y} \cdot v_s = 1 \in \mathbb{F}_2$, and $-\alpha o$ for outputs where $\mathbf{y} \cdot v_s = 0$. Thus, when o is nonzero, the model sends $2\alpha\beta$ to every $\mathbf{y}$ for which $\mathbf{y} \cdot \mathbf{v}_s = (\mathbf{c} + \mathbf{d}) \cdot \mathbf{v}_s$, and $-2\alpha\beta$ to the other outputs. Exactly half of the $\mathbf{y} \in \mathbb{F}_2^n$ satisfy this, as long as $\mathbf{v}_s \neq \mathbf{0}$.

Provided that enough different $\mathbf{w}$ are learned to separate any $\mathbf{x}$ from $\mathbf{x}'$ using equations $(1), (2), (3), (4)$, this model achieves perfect accuracy, since the correct answer will receive $2c$ from every hidden neuron whose output is nonzero.

It is worth noting that this algorithm is not too different to the more human approach of simply computing each bit. Indeed, if $\mathbf{w}$ has ones in every second place, an associated neuron essentially compares the two inputs in a single bit. If we believe that this is natural for the model to learn, then upon observing that the network should be agnostic to the choice of basis in $\mathbb{F}_2^n$, we must believe that the model is equally willing to learn any $\mathbf{w}$ in the Hadamard basis. With the weight decay constraint, it is not unsurprising that the model learns many such elements of this basis. A model could theoretically approach zero loss with $2n$ hidden neurons (two for each basis element of $\mathbb{F}_2^n$) but doing so requires large weights.

3.2 Learning Curves and Weight Evolution

Models are trained on every data point in the training set in each epoch — all 350 points are fed to the model in a single batch. In the model with weight decay, training loss rapidly decays to zero, while validation loss takes significantly longer to do so. In the model without weight decay, validation loss increases continuously as the model is trained further, indicating overfitting.

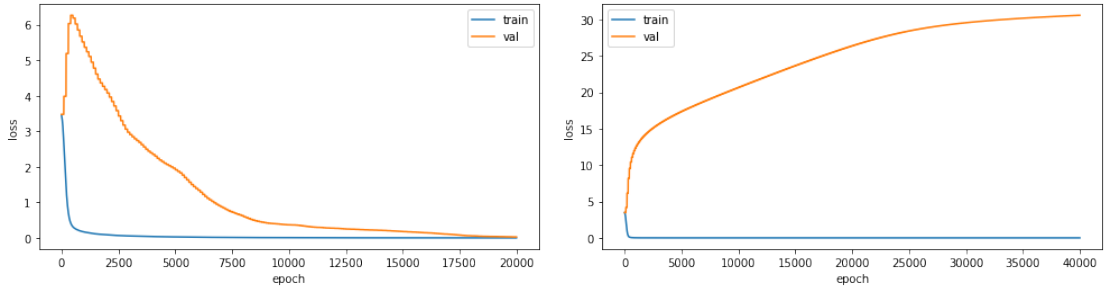

Figure 2: Training and validation loss on models with and without weight decay, respectively.

We first study the model with weight decay.

Visualising the weights of the 48 hidden neurons using a heatmap reveals regular patterns in each row - almost all rows correspond to a row of the Walsh matrix described before.

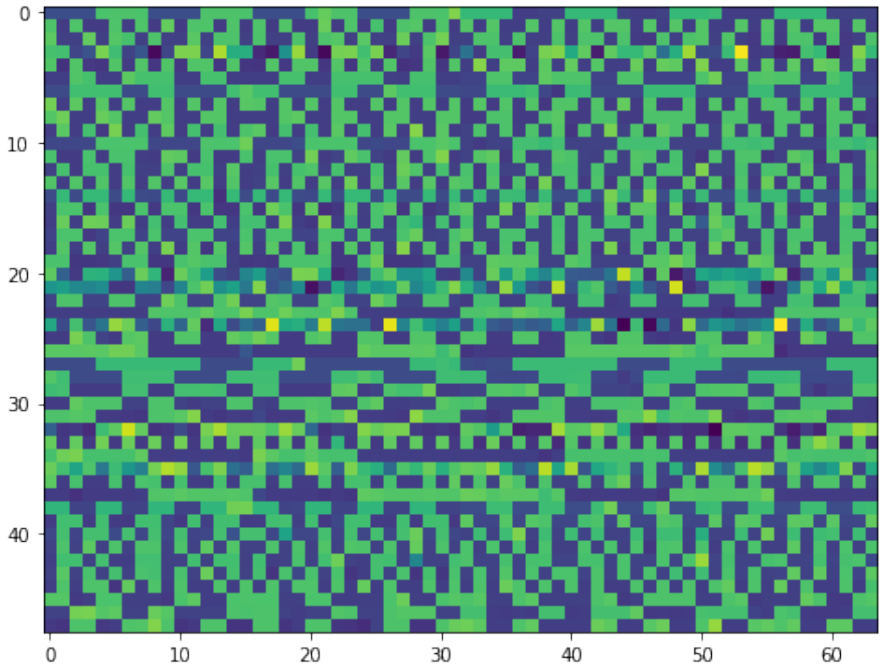

Figure 3: Input-layer weights for all 48 hidden neurons and all 64 input neurons.

To analyse the evolution of this model, we save weights periodically throughout training. For each hidden neuron, we record all 64 input weights over time, and also record the weights from this neuron to all outputs.

Since the Walsh matrix has orthogonal rows, normalising rows of the matrix gives an orthonormal basis of $\mathbb{R}^{64}$. We can express the normalised weight vector of each hidden neuron in this basis (i.e. we take a Hadamard transform), and report the squares of the coefficients over time. These always sum to one, and if a coefficient grows close to one then the associated neuron has essentially learned a row of the Walsh matrix.

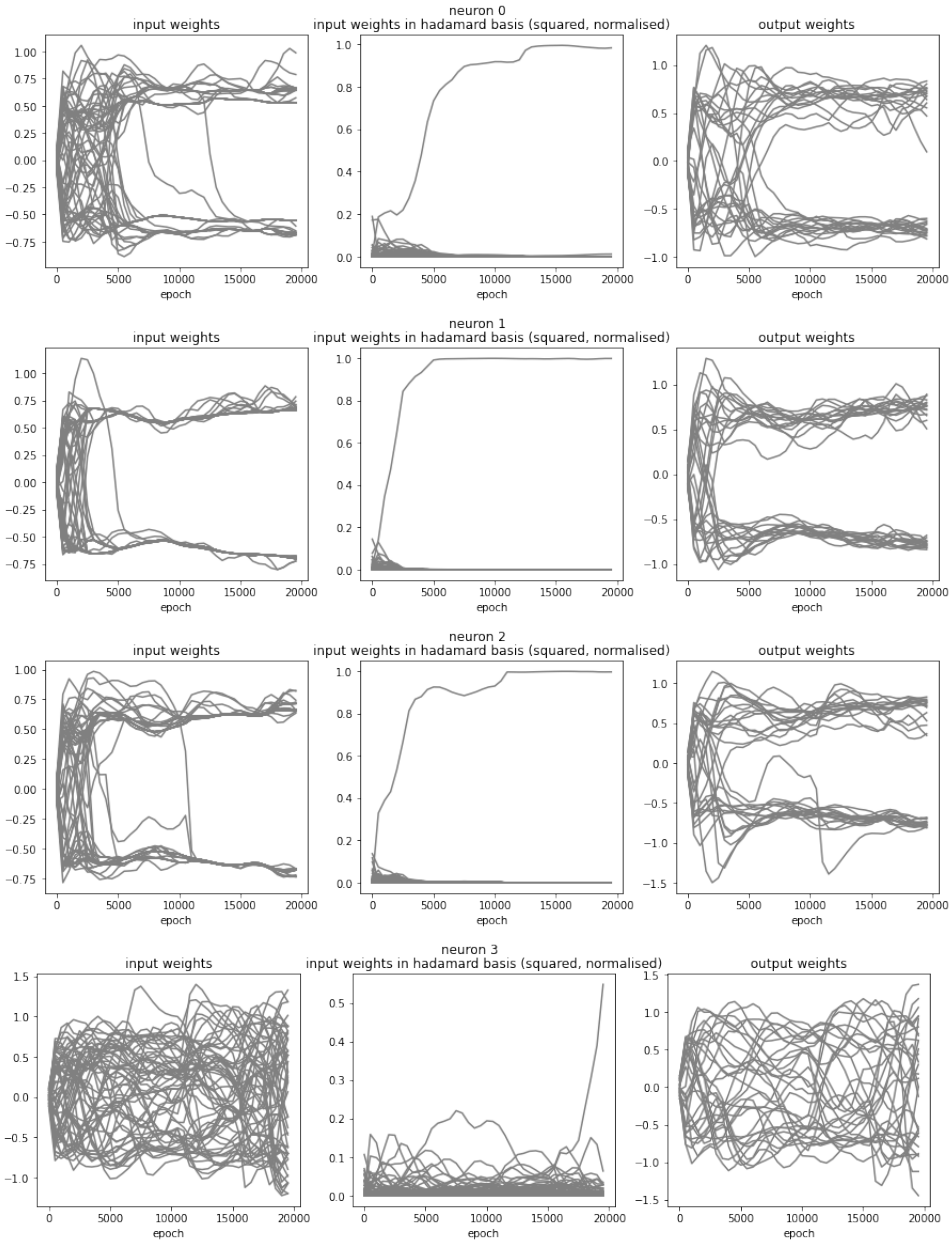

Figure 4: Input and output weights for four hidden neurons over time.

In many of the neurons, a row of the Walsh matrix is learned quickly, with the input weights and output weights both splitting into two equally sized groups as per the algorithm discussed earlier. A single element of the Hadamard basis begins to dominate quickly, while the other orthogonal directions rapidly decay to zero. Inputs are occasionally "caught on the wrong side" of a hidden neuron, and may take a long time to cross to the right side.

Some neurons take significantly longer to learn a row, and in some cases all coefficients stay small for the first 5000 or 10000 epochs, until at some point one coefficient rapidly grows. There are also neurons where a coefficient begins dominating, but then decays as another coefficient replaces it, though it is not yet clear why this happens.

Input biases are also small; most biases are < 0.05, with a few significantly larger exceptions (above > 0.5) corresponding exactly to neurons which have not yet converged to an element of the Hadamard basis.

Turning to the model without weight decay, we see markedly different results. The weights as visualised on a heatmap appear much more random and cannot easily be identified with rows of a Walsh matrix.

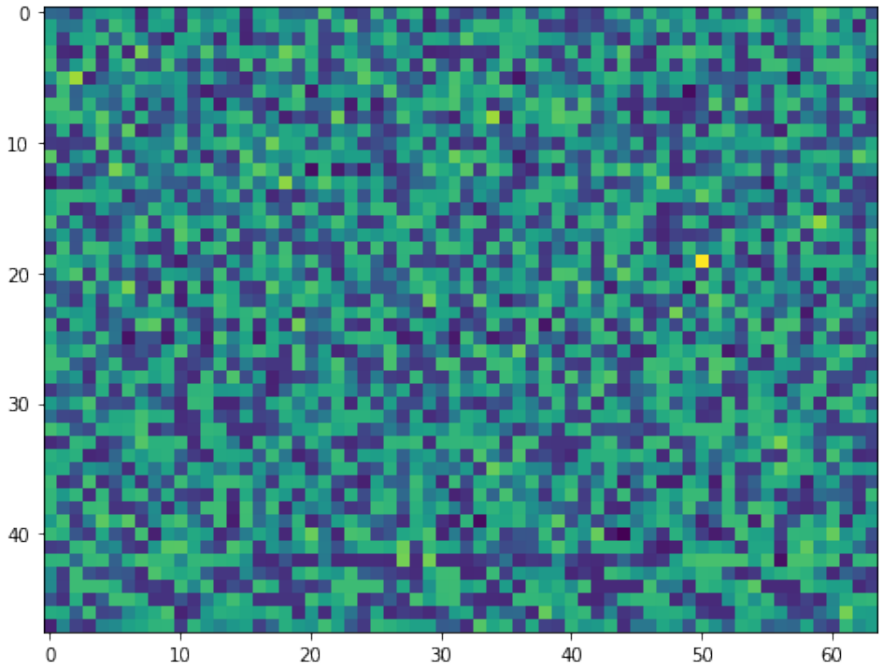

Figure 5: Input-layer weights for all 48 hidden neurons and all 64 input neurons, in a model with no weight decay.

If we study change in parameter values over time, we see that for individual hidden neurons, it is often the case that some Hadamard coefficient becomes significantly larger than the others (although by a much smaller margin).

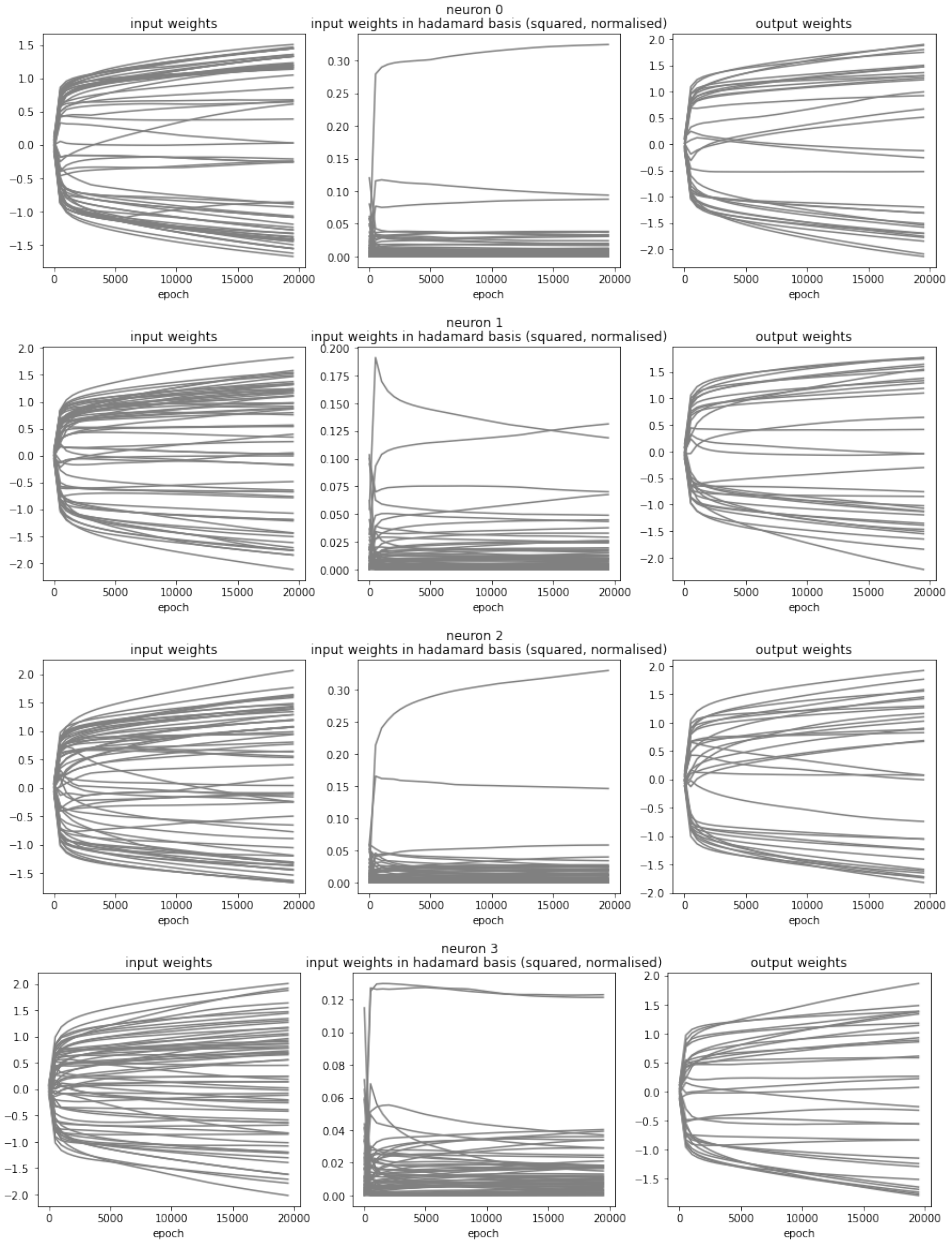

Figure 6: Input and output weights for four hidden neurons over time, in a model with no weight decay.

Weights seem to change more slowly after the first thousand steps, and without weight decay, orthogonal directions to the dominant one in the Hadamard basis do not decay easily. A handful of neurons demonstrate dominant squared Hadamard coefficient exceeding 0.5. Training for many more epochs causes all weights to eventually asymptote, without improvement in valuation loss.

I also ran a model in which weight decay was turned on after 10000 epochs. Neurons whose

dominant coefficient before regularisation is large tend to have the same dominant coefficient after regularisation (and the coefficient rapidly tends to 1).

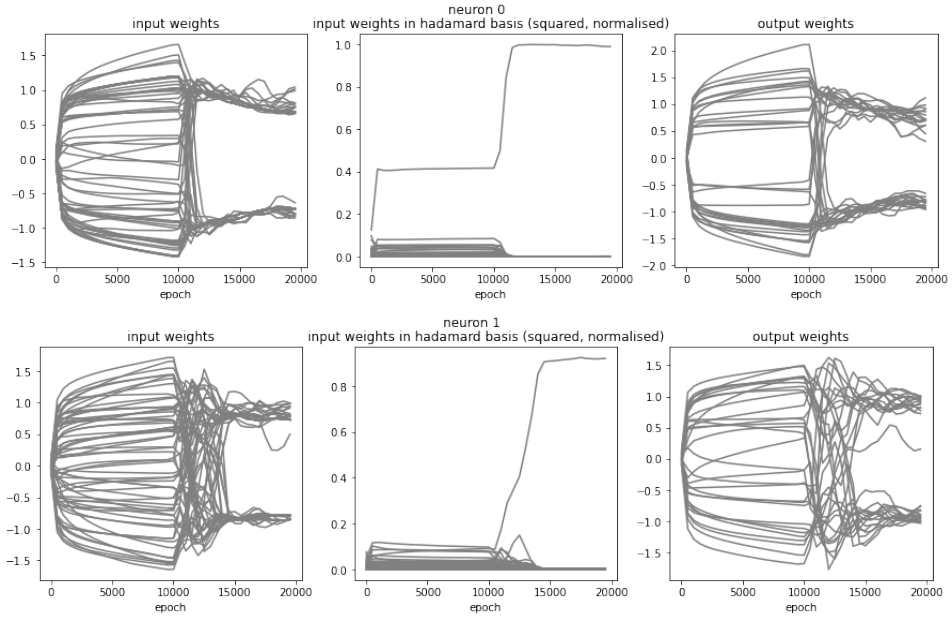

Figure 7: Input and output weights for two hidden neurons over time. Weight decay is turned on at epoch 10000.

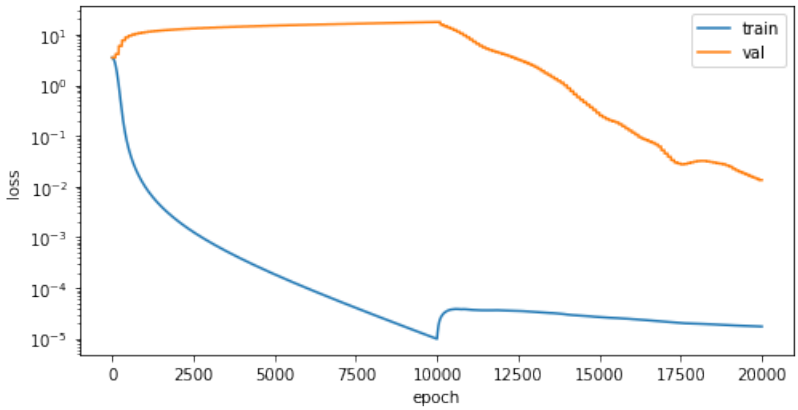

Figure 8: Training and validation loss on the aforementioned model. Loss is on a log scale.

Finally, training a model on all 1024 data points without weight decay yields a net where most neurons learn a single element of the Hadamard basis.

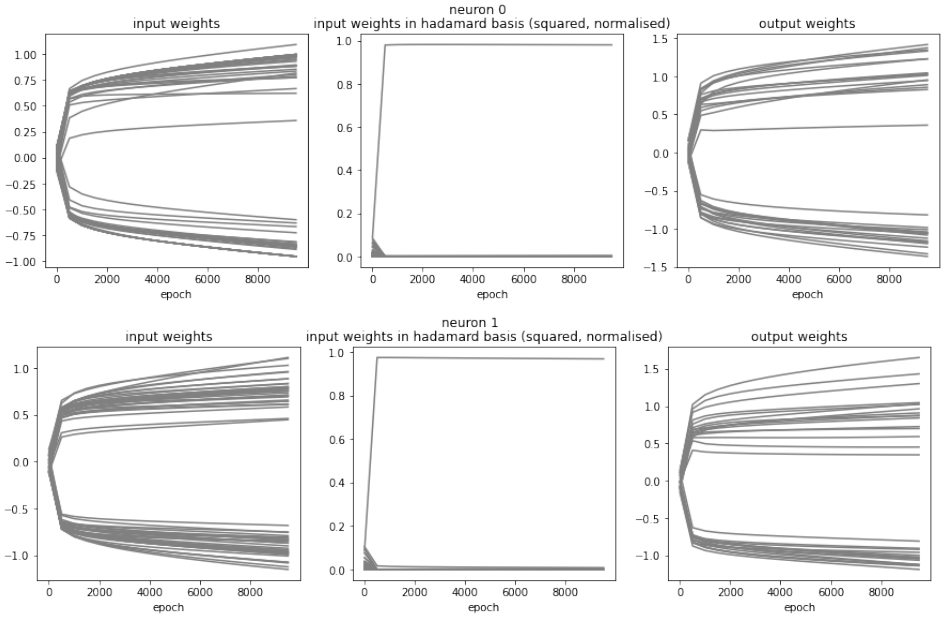

Figure 9: Input and output weights for two hidden neurons over time. The model is trained on all 1024 pairs, without weight decay.

4 Discussion

The effect of weight decay on generalisation (see Figure 2) is clear, but the underlying reasons for this are unclear.

We can treat the size of the largest squared Hadamard coefficient of a neuron's weights as a proxy for how much a model has learned about the ground truth, as opposed to memorisation/overfitting. This may be a better measure than the validation loss, which is more blunt and can be damaged heavily by a small number of neurons.

The model trained without weight decay still performs better with respect to this metric than random chance (the largest coefficient is consistently larger after training than when the weights are initialised), which suggests that the model has learned real patterns despite overfitting elsewhere.

A commonly quoted intuition for the effectiveness of weight decay is that solutions obtained via generalisation are "simpler" in some sense those which memorise, and that they have smaller weight sizes. We do see that weights in the model without weight decay are somewhat larger in magnitude than in the model with weight decay. This may explain the difference in generalisation ability, but does not offer much explanation as to why the model without weight decay learns some generalising patterns quickly before essentially freezing up.

A more speculative explanation is that generalising solutions are stronger attractors in feature space than memorisation solutions, which are much more common but break easily under small perturbations. One could imagine a well whose centre is a generalising solution, while there are deep, thin "troughs" cut randomly in the surrounding space, representing memori-

sation solutions. If this were the case, one might expect that a model without weight decay will generally move towards the generalising solution while optimising loss, until it falls into a trough — yielding a partially memorised, partially generalised solution. This is somewhat evidenced by the Hadamard coefficients of the model without weight decay. Weight decay would constantly perturb solutions, allowing the model to move towards the generalisation solution without getting stuck. In further research, it would be interesting to test this hypothesis: one could replace weight decay with constant addition of small noise to weights, and see if it leads to generalisation.