Understanding Polysemanticity and Improved Interpretability in Neural Networks with Coding Theory - Supplementary Material

1 Experimental details

A zip file is provided in this supplementary material with the exact files used to create the results showcased in this experiment. We also provide brief descriptions of each file here to allow those looking to recreate our work or to find specific experiment details quicker access. After the anonymous peer review process is over these will be provided on a public github linked in the main text.

ResNetVAE/modules.py This code defines a neural network model called ResNet_VAE that is capable of encoding and decoding image data using a variational autoencoder architecture. The model uses a pre-trained ResNet18 model for encoding, and includes several fully connected layers and convolutional layers for decoding. The model is capable of generating a reconstructed image from an input image, as well as outputting a latent representation of the input image that can be used for further downstream analysis.

ResNetVAE/ResNetVAE_cifar10.py The ResNetVAE_cifar10.py script contains the code to train a variant of the Variational AutoEncoder using the ResNet architecture as an encoder to encode an image into its latent representation. The script uses the CIFAR10 dataset to train the model and record the loss metrics throughout the training process. The code handles saving and loading of model and optimizer states, and uses weights and biases for logging and visualization. Finally, the trained models are stored in local disk space as well. The script has an adjustable dropout probability and can be trained for an adjustable number of epochs.

ResNetVAE/make_plots.py This code is used to analyze the properties of a ResNet-VAE neural network model trained on the CIFAR-10 dataset. The code first loads the model and uses hooks to extract activations at specified layers. The activations are then used to perform PCA analysis and the variance explained by each component is plotted. The code then generates activations for a set of dot images and project them onto a low-dimensional space obtained through a random projection matrix. The energy of the projection and the correlation between inputs and the projection are computed and plotted. Finally, the code computes a decay coefficient for each layer of the model and plots it against the energy of the projection and the correlation between inputs and the projection. The results are saved to disk as figures.

ResNetVAE/helpers.py This code contains several utility functions for computer vision and machine learning tasks. It includes a function to generate a sequence of dot images moving in a circular path, and preprocesses the images using the default transform for CIFAR images. Another function computes the total energy of a set of points in n-dimensional space by calculating the squared magnitude of the differences between successive points. Additionally, there is a function that returns a dictionary of random projection matrices with specified dimensions for different layers in a neural network. These functions can be used to generate image data, calculate energy, and create random projection matrices for machine learning tasks.

automated-interpretability/neuron-explainer/generate_trajectory.py This code generates random trajectories for a pre-trained language model. It captures activations of certain layers during a forward pass and computes the total energy of a set of points in n-dimensional space. It then finds the minimum trajectory and project it to a 1D plot. The process is repeated multiple times to visualize different trajectories in each iteration. The final result is a set of PDF files and NPY files showing the 1D projections of different minimum trajectories. The generated plots can be used to gain insights into how the language model processes various sequences of text inputs.

automated-interpretability/neuron-explainer/generate_explanations.py This script generates explanations for the neuron activation patterns of a pre-trained GPT-2 XL language model using a calibrated simulator.

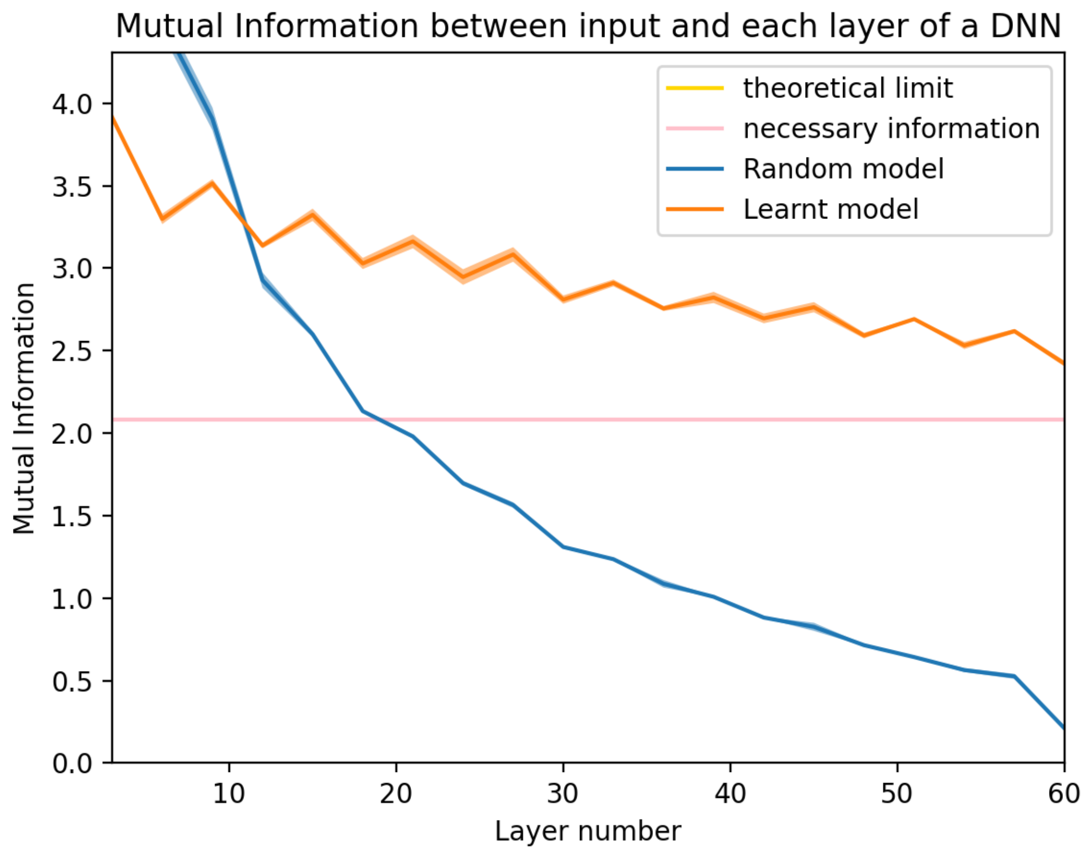

Figure 1: MINE is deployed to estimate the continuous mutual information between the input and a given layer of a neural network. One neural network is trained in the presence of dropout to preserve a subset of the information in the input while the other isn't. This demonstrates that in some settings a neural network is capable of learning an error correcting scheme, the random model provides a demonstration of what happens to information in this scenario that isn't intentionally preserved

The explanations are generated for the top 10 activation patterns of each layer in the neural network based on their mean maximum activation value compared to all activation patterns. The TokenActivationPairExplainer is used to generate explanations for the neuron activations, followed by simulating and scoring the explanation using an UncalibratedNeuronSimulator. The generated explanations and scores are saved for later use.

Estimate-MI-of-NN-Layers.py This script trains a very deep neural network on a simple task (preserving information) in a regime where error correction is needed to preserve any information. The results of this program are shown in figure 1

2 Theorems and Proofs

We provide formal statements and proofs behind the results claimed in the main text.

2.1 Optimal sparse input ratio

This subsection derives the predicted optimal ratio # dimensions-per-feature to sparsity.

First, we formally define the sparse input distribution. It is based on a uniform random distribution with S features set to 0.

Definition 1 (Sparsification). *Let S-sparsification be a process that takes a vector random variable v and returns a random variable v' with each element of v' independently set to 0 with probability S. If X is any distribution with i.i.d. elements then the S-sparsification of X is the distribution of X after S-sparsification, denote it as X^S*

The entropy of such a distribution is

Lemma 2 (Entropy of sparse distribution). *Let X be some distribution. If X^S is the distribution after S-sparsification then the entropy of X^S is:*

$$H(X^S) = -nS\log(S) - n(1-S)\log(1-S) + n(1-S)H_{elem}(X)$$

Where $H_{elem}(X)$ is the elementwise entropy of X

Proof. Express the joint entropy as the sum of the individual entropies (chain rule of entropy)

$$H(X^S) = H(X_0^S, X_1^S, \dots, X_n^S) = \sum_i (H_i^S | X_0^S, \dots, X_{i-1}^S, X_{i+1}^S, \dots X^S)$$

as elements are i.i.d. (a condition of sparsification).

$$H(X^S) = \sum_i (H_i^S) = nH(X_0^S)$$

W.L.O.G. we have set all entropies equal to that of the first element of the vector.

If we assume smoothness of X the probability that any un-sparcified element of a vector is exactly 0 is vanishingly small $P(X_0 = 0) \approx 0$, allowing us to assume that if X_0^S is 0, it has been set this way by sparsification. This implies that a single element of the sparcified vector has the distribution:

$$P(X_0^S = Y) = \begin{cases} (1-S)P(X_0^S = Y), & \text{if } Y \neq 0 \\ S, & \text{if } Y = 0 \end{cases}$$

Directly calculating the entropy of this distribution:

$$H(X_0^S) = -\sum_{X_0^S} P(X_0^S)\log\big(P(X_0^S)\big) = -\sum_{X_0^S \neq 0} P(X_0^S)\log\big(P(X_0^S)\big) + P(X_0^S = 0)\log\big(P(X_0^S = 0)\big)$$

$$H(X_0^S) \approx -\sum_{X_0^S \neq 0} (1-S)P(X_0)\log((1-S)P(X_0)) - S\log(S)$$

$$H(X_0^S) \approx -(1-S)\sum_{X_0^S \neq 0} P(X_0)\log(P(X_0)) - (1-S)\sum_{X_0^S \neq 0} P(X_0)\log(1-S) - S\log(S)$$

$$H(X_0^S) \approx -(1-S)H(X_0) - (1-S)\log(1-S) - S\log(S)$$

Which proves the result of the lemma. □

The above statement is approximate, taking the assumption that $P(X_0 = 0)$ is arbitrarily small. This can be turned into a strongly rigorous statement by taking smoothness or other assumptions which amount to bounding $P(X_0 = 0) < \varepsilon$. In this case the stated entropy result is within additive ϵ. For the case we are interested in (Elhage et al.) X is a uniform distribution of high precision floats, making $P(X_0 = 0) \approx 0$ sufficiently accurate.

The stated ratio simply follows from this lemma by assuming the precision (and hence entropy) of the internal state/hidden layer/linear algebra is equal to that of the unsparcified input distribution, X: H_elem. The channel capacity of m hidden units is $m \times H_elem$. The ratio, $m : n$, is found by divding one by the other.

2.2 Hardness of decoding a certain problem

As codes become more efficient (approach the Shannon limit given the entropy) it seems likely that they become less interpretable. To provide weight to this conjecture we outline a learning problem such that any attempt at decoding [1] the optimal code in the final layer in polynomial time must fail, assuming cryptographic primitives.

The existence of such a task is relatively trivial, we can see that a neural network is capable of learning to implement a discrete exponential, an function which is easy one-way but complicated to find the inverse of. If any algorithm were able to decode the input from a given output we would be able to use the same algorithm to find the inverse, i.e. to solve the discrete logarithm problem. The optimality of code constraint forces the model to keep only the bits it needs for the problem, leaving just the exponential and no "leaks" as to what the input was.

Formally the problem gives 3 values as input: a prime, p, a primitive element, g of $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$, and $x \in \mathbb{Z}_p^*$. The tasks is then to output $g^x (\text{mod} p)$. This is an easy function to implement and very likely learnable up to large p, g, and x (only existence, not learnability, is strictly needed to show the result). It is now obvious how any algorithm that could take the output of this circuit and calculate x could solve DLP.

DLP is thought to be average-case hard for chosen groups (chosen p), restricting our problem case to these instances completes our result. Many other cryptographic one-way functions could be used in the place of DLP here, strengthening this result to rely on any one of a number of possible assumed hardness results. Indeed, given a public-private key encryption scheme which is average case hard for every bit of the input, we can modify the learning problem such that the neural network implements the public key encryption, again any decoder that breaks even a single bit of the encrypted activations in polynomial time could be used to recover the original message without access to the private key.

2.3 Channel capacity of a dropout layer

This subsection proves the channel capacity result claimed in the main text, namely that a neural network with n dropout layers, with dropout rate D has a channel capacity of $C_{dropout} \sim C(1-D)$, where C is the capacity of the dropout-less layer.

Theorem 3. *The channel capacity of m independent neurons, each with capacity C_{base} when subject to an uncorrelated dropout layer with rate D is bounded by:*

$$(1-\varepsilon) \times (1-D) \times C_{base} \times m \leq C^m_{dropout} \leq (1-D) \times C_{base} \times m,$$

where $\varepsilon = -\log\left(1 - 2^{-C_{base}}\right)$ tends to zero asymptotically for large C_{base} (high precision).

Proof. We first note that C^m_{dropout} is equivalent to $m \times C^1_{\text{dropout}}$ due to the chain rule of entropy and the independence of channels.

We then establish the lower and upper bounds separately for C^1_{dropout}, both rely on the similarity of this channel to the standard erasure channel.

Lower bound(proof by construction) Create a uniform distribution on all but the 0 symbol, call this X. The entropy of this distribution is equal to the full uniform distribution sans 1 state:

$$H(X) = \log\left(2^{C^1_{base}} - 1\right) = \log\left(1 - 2^{-C^1_{base}}\right) + C^1_{base}$$

Let Y be the distribution given by putting X through the dropout layer. The mutual information between the two is:

$$I(X;Y) = H(X) - H(X|Y) = H(X) - \sum P(Y=y)H(X|Y=y)$$

If $y = 0$, $H(X|Y=y) = H(X)$ as dropout applies to all inputs equally and the 0 symbol in X has no probability mass, if $y \neq 0$ then $y = x$ and $H(X|Y=y) = 0$. Thus:

$$I(X;Y) = H(X) - H(X|Y) = H(X) - P(Y=0)H(X) = (1-D)H(X)$$

[1] i.e. to take the code of an input and resolve the original input, or what information about the input has been conserved

Implying the lower bound on the channel capacity

$$C^1_{\text{dropout}} \geq (1-D) \times (\log\left(1 - 2^{-C^1_{base}}\right) + C^1_{base})$$

Upper bound(proof by contradiction) Assume toward contradiction that $C^1_{\text{dropout}} = (1-D)C^1_{base} + \varepsilon$ for some fixed $\varepsilon > 0$. Let the n-bit erasure channel be the extension of the binary erasure channel to an input of n bits, i.e. on input $x \in \mathcal{X} = \{0,1\}^n$ the channel produces output $y \in \mathcal{Y} = \{0,1\}^n + e$, where e is an error symbol, with probability:

$$P(Y=y|X=x) = \begin{cases} \alpha, & y = e \\ 1-\alpha, & y = x \end{cases}$$

The proof of the channel capacity of this channel is a trivial extension of the binary case: $I(X;Y) = H(X) + H(X|Y) = H(X) - \sum P(Y=y)H(X|Y=y) = (1-\alpha)H(X)$

Which is maximized when X is uniform, giving a capacity of $(1-\alpha)n = (1-\alpha)C_{base}$.

However, given access to a n-bit erasure channel with $\alpha = D$ we can recreate a dropout channel by setting all e symbols to 0. If The channel capacity of our dropout layer was as assumed we could use the encoding procedure for the dropout layer, pass it through the n-bit erasure channel, perform the $e = 0$ operation and achieve the dropout layer capacity. As $\varepsilon > 0$ this implies it is possible to transmit more information than the channel capacity through the erasure channel, causing a contradiction. □

2.4 Information loss from power law distribution

The entropy of a distribution with variance following power law distribution is highly limited, constraining the channel capacity ($C = H(X) - H(X|Y) < H(X)$). This section calculates how much information is not transmitted by the power law distribution compared to a uniform (flat distribution).

Theorem 4. *The entropy of a distribution of m neurons to precision n following a power law distribution with exponent α, X, is upper bounded by:*

$$H(X) \lessapprox mn - \alpha m \log(m/e)$$

if the distributions of each number follow a normal distribution.

Proof. W.L.O.G. Assume the variances, σ_i, are normalised such that $\sigma_1 = 1$. The power law distribution then implies $\sigma_i = i^\alpha$

Entropy is upper bounded by the case where all m channels are independent:

$$H(X) \leq \sum_i^m H(X_i)$$

Assuming the first element optimally uses every bit it would uniformly distribute over all values, the subsequent variances would take up a smaller and smaller section (as the variance must decrease, the entropy implied by each variance is approximately:

$$H(X_i) \approx \log(2^n \sigma_i) \tag{1}$$

Substituting in variance gives:

$$H(X) \leq \sum_i^m \log(2^n \sigma_i) = \sum_i^m \log(2^n i^\alpha) = mn - \alpha \log\left(\prod_i^m i\right)$$

Applying Stirlings lemma gives the final result:

$$H(X) \lessapprox mn - \alpha m \log(m/e)$$

□

Considering the Gaussian distribution has the highest entropy for a given variance the choice to assume a normal distribution in the previous answer may appear strange. Normal was chosen so that the first variance could be assumed to be an optimally dense code, if Gaussian is chosen edge effects (i.e. running out of numbers for a true Gaussian) must be considered, making it unclear what the initial value of σ_0 is. However, a reader that wishes to use a Gaussian may simply modify the above proof to replace the entropy in the equation 1 with that of a Gaussian. Any reasonable values of the variance result in an additive term of at most $\frac{m}{2}log(2\pi e)$, making the normal approximation close to the Gaussian.

2.5 Connection between power law exponent and energy of a random projection

This section connects the action with the power series decay parameters, α.,

We're given a vector of hidden activations, $A \in \mathbb{R}^m$, centered around zero, $A \sim \mathcal{N}(0, \Sigma)$, and random projections, $r \in \mathbb{R}^{m\times 3}$ centered at 0 with normalised variance $r_i \sim \mathcal{N}(0, \frac{1}{m}I)$. Expand definition of the action:

$$\langle E[z] \rangle = \frac{1}{2}\langle ||Ar||^2 \rangle = \frac{1}{2}\langle r^T A^T A r \rangle.$$

The action is a scalar, thus it is equal to its trace, applying the cyclic and linear properties of the trace we rearrange and pull the expected value into the trace (the trace trick):

$$\langle E[z] \rangle = \frac{1}{2}\langle \text{tr}(r^T r) \rangle \text{tr}(\langle A^T A \rangle) = \frac{1}{2}\text{tr}(\Sigma)$$

As the trace is equal to the sum of the eigenvalues, and the eigenspectrum of Σ is the set of variances, we rewrite the expected action as a sum over the power law,

$$\langle E[z] \rangle = \frac{1}{2}\sum_i \lambda_i = \frac{1}{2}\sum_i i^\alpha \approx \frac{1}{2}\sum_{i=1}^{\infty} i^\alpha = \frac{1}{2}\sum_{i=1}^{\infty}\frac{1}{i^{-\alpha}} = \frac{1}{2}\zeta(-\alpha).$$

As the later terms of the sum rapidly approach zero, we can approximate the sum with the infinite series. Rewriting the expression in terms of $-\alpha$ then results in the connection with the ζ function,

$$\langle E[z] \rangle \approx \frac{1}{2}\sum_{i=1}^{\infty} i^\alpha = \frac{1}{2}\sum_{i=1}^{\infty}\frac{1}{i^{-\alpha}} = \frac{1}{2}\zeta(-\alpha).$$