Model stitching to investigate the location of simplicity bias in neural networks

Candidate number: 1064144

Project number: TP11

Supervisor: Professor A Louis

1 Abstract

We draw on *model stitching*, a technique developed to investigate the internal representation of neural networks, and use it to analyse trait localisation. We take two trained models A and B, freeze their layers, and connect the top layers of A with the bottom layers of B via a stitching layer. Through a number of experiments we verify that stitched models inherit "traits" exhibited by the initial models, a trait cannot be inherited if it is located in a layer not present in the stitched model, and a trait distributed throughout many layers is exhibited increasingly as the number of layers containing that trait increases. Consequently, we argue that stitching has a number of under-recognized capabilities including trait locating, creating reductionist descriptions for distributed traits, and creating multi-specialised networks. In particular, our core finding is preliminary evidence suggesting stitching can be used to locate "simplicity" within models.

2 Introduction

2.1 Supervised training for classification tasks

In this paper we are training models to perform image categorisation, meaning we want to be able to input an image from a dataset to the model and have it correctly output the category that the image belongs to (i.e. identify what the image is). We are using supervised learning, meaning that for each input we know the correct output i.e. each image in the database has a label identifying what category it belongs to (car, plane, etc). The metric by which we judge the performance of the the model is the loss function - all training requires some loss function, we chose cross-entropy loss as it is standard for categorisation tasks:

Cross-entropy loss
$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik}\log\left(\hat{y}_{ik}\right) \tag{1}$$

where:

- $y_{ik} = 1$ (the label) if example i belongs to category k, and 0 otherwise.
- $\hat{y}_{ik}$ (the model's output) is the predicted probability that example i belongs to class k.
- n is the number of inputs
- K is the number of categories

The output of the model is a set of numbers ranging from 0 to 1 based on how strongly the model 'believes' that the input belongs to each category. The loss function is larger when the model's output is further from the true label (which would be 1 in the correct category, and a 0 in all others). The act of training is simply giving the model a training set of images and trying to minimise the loss of the model's outputs on those inputs. We chose the standard method of doing this, stochastic gradient descent. What this entails is taking a small batch of inputs (we used 4 at a time), and using an algorithm called back propagation we calculate what 'direction' in the loss landscape (meaning what change of weights and biases of the model) would produce the largest decrease in the loss. Then we take a small step in that direction (the size of the step is set by the learning rate, we used a default of 10^{-3}, which is standard). We then take a new batch and repeat the process until the loss stops meaningfully decreasing at each step (the loss plateaus), at which point our model is trained. Precise training details are included in the methods section.

2.2 The unreasonable performance of over-parametrised networks

Modern neural networks are increasingly being shown to have revolutionary potential in a wide variety of tasks, this is especially seen with deep neural networks containing many hidden layers [1]. This is due to the surprising result that neural networks work best in the over-parametrised regime, with vastly more parameters than data points (with a possible exception for large language models [2]). Classical analysis would tell us that (as in the case of fitting a curve) this would lead to over-fitting of noise rather than capturing the underlying trend and hence lead to poor generalisation, but this is not what we observe.

To introduce this more precisely, we consider the models as representing functions which take some input x_i and then outputs y_i'; and for each x_i there is a correct output y_i. On some training set of m pairs of input-output S = $(x_i, y_i)_{i=1}^m$ we consider some loss function $\mathcal{L}(y_i', y_i)$ which measures the difference between the outputs given by the model and the correct outputs. Because the models are so over parametrised they have a very large capacity, and because training methods are generally quite competent, we can assume that the

function reaches complete accuracy (zero loss) on the training set. Then we test the functions on a test set with n input-output pairs $\mathrm{T} = (x_i, y_i)_{i=1}^{n}$. In the simplest case of binary classification there are $\mathrm{N} = 2^n$ functions that have zero training error. Since our networks are so high capacity, they will be able to express many (or even all) of these functions. Why then, of the huge number of possible functions $\mathrm{N} = 2^n$, do models converge on the small fraction that perform well on the test set - why do they converge on the small fraction that generalise?

A frequently proposed solution is that neural networks have a bias for representing simpler functions, which coupled with the structures in our data, encourages the generalisation we observe [1]. There are many theories as to how this simplicity bias emerges, including describing it as a by product of the specific optimisers used in training [3], attributing it the initialisation of the weights and biases [4], or as an intrinsic inductive bias [1]. However, there is currently no conclusive theoretical or experimental confirmation of any theory. Many of the theories would produce different predictions for whether all layers are equally affected, or if some layers are particularly susceptible to the simplicity bias. Therefore, we believe that being able to locate the "complexity" in a model layer by layer would provide meaningful insight and be a useful tool for analysing these theories - which in turn would shed light on the exceptional ability of neural networks to generalise.

2.3 Measures of simplicity

Throughout this paper we take high simplicity as equivalent to low complexity and vice versa. The primary metric for simplicity that we use in this paper is the flatness of the loss landscape, as this is a frequently made connection in the literature [5]. It should be noted that the connection is not undisputed [6], but that applies to most measures of simplicity for neural networks, and it is suitable for our purposes as it is defendable and easy to use. We used two quantitative measures for flatness: gradient norm (the magnitude of the gradient vector of the loss) and loss sharpness (the magnitude of change in the loss under a small perturbation of the parameters):

Gradient norm
$$\|\nabla_\theta \mathcal{L}(\theta)\|_2 = \sqrt{\sum_i \left(\frac{\partial \mathcal{L}}{\partial \theta_i}\right)^2} \tag{2}$$

Loss sharpness
$$\mathrm{Sharpness}(\theta) = \max_{\|\Delta\theta\| \leq \epsilon} \mathcal{L}(\theta + \Delta\theta) \tag{3}$$

Where $\mathcal{L}$ is our loss function, and θ is the parameters of the model, i.e. all the weights and biases of all neurons. Practically, one cannot calculate the change in loss for every $\Delta\theta$, so we use an assortment of random perturbations and take the average. We also chose to use a separate, independent measure: the compressibility of the model using gzip. This was chosen because, as argued by Jiang et al [7], it can be used as an approximation of Kolmogorov complexity, which has been argued by Mingard et al [1] to be the most relevant and meaningful definition of complexity. Additionally it is used for being intuitive; the more a model can be compressed, the more simple it must be. The code for all three simplicity measures is in Appendix 1.

2.4 Model stitching

Consider two neural networks A and B, both with some architecture κ consisting of L layers. We define a new model with the first x layers from model A, followed by a low capacity stitching layer, then the remaining $L - x$ layers from model B. This is our stitched model C. If we consider each model as a list of layers where $A = [l_{A1}, l_{A2}, ..., l_{An}]$ and $B = [l_{B1}, l_{B2}, ..., l_{Bn}]$ then $C = [l_{C1}, l_{C2}, ..., l_{Cn+1}] = [l_{A1}, ..., l_{Ax}, S, l_{Bx+1}, ..., l_{Bn}]$ where S is the stitching layer. The layers from A and B have their weights and biases frozen while the stitching layer is optimised on the training data. We refer to the model which provides the layers before the stitching layer as the "top" model, and the model which provides the layers after as the "bottom" model. An example Section of code for stitching is provided in Appendix 2.

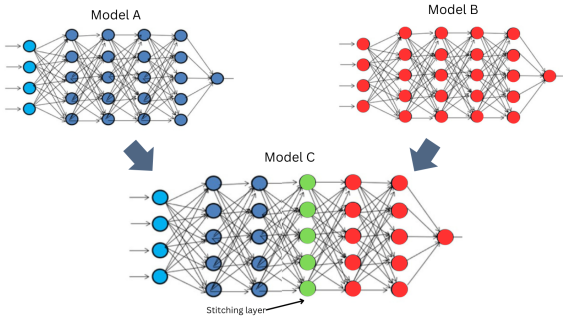


Figure 1: Picture representing the first 3 layers of model A being stitched to the last three layers of model B

Model stitching was first introduced in a 2015 paper by Lenc and Vedaldi [8] and then developed further in 2021 by Bansal, Nakkiran, and Barak [9]. It has primarily been used as a tool to investigate representations

held within neural networks ("representation" is a broad term used to qualitatively describe how the network understands the data, for example an early layer might identify edges and colours, whereas a later layer might look for features like whiskers or doors - these are the representations of those layers). The general concept being that networks which share representations can be easily stitched together whereas networks with different representations will suffer a sharp drop in performance as their layers will not be compatible. In this paper we focus on a related but distinct concept of "traits", simply meaning a measurable feature of the outputs of a model. Some examples included if a model has a particularly low or high performance on a certain subset of the data, if the model confuses two categories, or if a model can only uses certain parts of the input

As the stitched model C shares the representations of the models A and B, it is reasonable to assume that model C might also inherit the traits of models A and B. The contribution of this paper is the assumption that model C cannot inherit a trait which is localised to a layer in A or B that is not included in model C. Furthermore, we hypothesize that if the traits of C can be viewed as the net result of its layers, then by stitching a model A to a test model B and observing how C's traits change as the stitching layer moves, we can get a measure for how much each of A's layer contribute to that trait. The case of interest is measuring the trait of simplicity/complexity layer by layer. In Section 4 we will present results supporting each of these concepts in turn.

3 Methods

In this project we have employed stitching as described in Section 2.4 across a number of different model architectures to test its capabilities as an analytic tool. Specifically, the architectures we have used are: fully connected models (called MLP for multi-layer perceptron) with layers of uniform width (64 neurons wide) and a varying number of hidden layers (4-10); convolutional neural networks (CNN) with two initial convolutional layers (with pooling layers) and 3-5 fully connected (Fc) layers; and the ResNet18 architecture [10] (an architecture with 18 convolutional layers). A variety of model architectures is necessary, as we want to demonstrate that stitching can be applied in a wide number of cases. ResNet18 is only used when a larger network is needed, as it is much larger than the other architectures, making training and stitching more time consuming.

All models are tested and trained using the MNIST [11] and CIFAR10 [12] datasets, which are commonly used datasets containing images belonging to one of ten possible labels (for MNIST it is the digits 0-9, for CIFAR it is ten common objects/animals such as truck, cat, car, frog, etc.). We chose these data sets as they are standard in the field, easy to train models to good performance on, and because they differ in complexity, which is a property we use in Section 4.4. We use vision based tasks as this is the domain where stitching has been shown to work, and testing stitching in a new domain (e.g. a language model) is beyond the scope of this experiment.

For training the models (as mentioned in Section 2.1) the criterion used is the cross-entropy loss and the training method is stochastic gradient descent (SGD). We generally used a batch size of 4, an initial learning rate of 10^{-3}, and a momentum of 0.9 - these values were chosen as they are standard. Additionally, for some specific goals we add additional features such as weight decay, pruning, and weight regularisation, these are explained later when they are used. All training is done until the loss plateaus, rather than a fixed number of epochs.

The specific stitching layer used depends on the architecture: for MLPs we used a fully connected layer; for the CNNs we use a 1x1 convolution layer if placed between convolutional layers, a fully connected layer if placed within the fully connected layers, or both a 1x1 convolutional and a fully connected layer when connecting a convolutional to a fully connected layer; for ResNet18 we use a 1x1 convolutional layer, but add a BatchNorm layer either side (this layer solely normalises the outputs of a layer which is useful for optimisation, it does not affect the representational capacity). Care was taken to ensure the stitching layer is not too powerful; it is important for our results that the stitched layer is able to connect the models without having the representational capacity to learn any traits. To ensure this, for each experiment we first stitch two untrained and randomly initialised models to ensure the stitching layer alone cannot learn the traits.

All computational work was performed on a MacBook Air (2024) equipped with an Apple M3 system-on-chip architecture (8-core CPU), 16 GB of unified memory, and running macOS Ventura version 14.6. All code used was written specifically for this project, and is available at:
https://github.com/oxfordphysics55/Stitching-project-code

4 Results and Analysis

In this Section we will detail the results that have been taken from the experiments on stitching, specifying how the methods deviate from the general outline listed above as necessary. The first three serve as the foundation for the fourth, which is the main conclusion of this paper. We will cover the following results:

1. Traits can be inherited using stitching
2. Stitching can be used to locate a trait within a network
3. Stitching can be used to give a layer by layer description of a holistic trait
4. Stitching as a means of locating simplicity

4.1 Traits can be inherited using stitching

To begin with we aim to confirm the concept that model C does inherit traits from models A and B. We train two MLP networks with 9 hidden layers of width 64 on CIFAR, model B is trained normally, model A is trained with data where all cars are labelled as cats (cats are still correctly labelled). Stitching after the 5th hidden layer, we find model C is unable to correctly identify cars (independent of which model is top/bottom) but can perform comparably to A and B on the other 9 image categories.

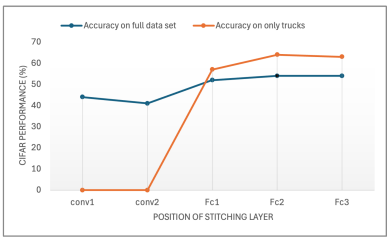
To ensure this applies over multiple architectures we use CNN models and again train model A with the altered labels and model B with the correctly labelled inputs, then connect the models by inserting a stitching layer between the convolution layers and the fully connected layers. When model B is the front model, model C is unable to correctly identify cars, regardless of training. When model A is the front model, model C is able to correctly identify cars, after sufficient training. This seems to imply that C does inherit traits, but that the trait 'car-cat' confusion is located in the fully connected layers, not the convolutional layers.

We began with the MLP architecture as it is conceptually the simplest, however its performance is significantly worse than a CNN architecture, so we decided to stop using MLPs for the rest of the experiments.

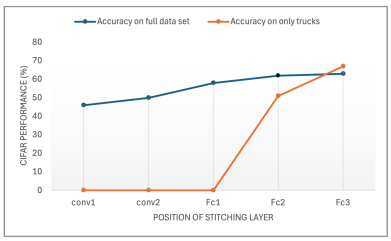
4.2 Stitching can be used to locate a trait within a network

Given that traits can be inherited, we make the assumption that if model A has a trait, and that trait is entirely located in a single layer, then C can only inherit that trait from A if that special layer is included in C. If this is true, then by moving the position of the stitching layer and measuring the expression of that trait, we would be able to locate which layer is the special layer, and hence find where the trait was located in A.

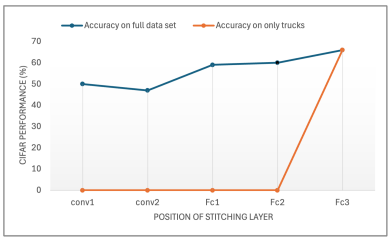
To test the assumptions, we use model A and B as standard CNN's with four fully connected layers. Model A and B are both trained on a modified CIFAR dataset where all images of trucks are removed, then all layers of A except the special layer are frozen and this layer is trained on a complete CIFAR dataset. Model A is then tested on images of trucks and is found to be able to identify them to a level comparable to the other 9 categories. B, as expected, has no ability to recognize trucks (i.e. is 'truck-blind').



(a) Performance jump with special layer Fc1



(b) Performance jump with special layer Fc2



(c) Performance jump with special layer Fc3

Figure 2: This shows the stitched model's performance on the whole data set (blue) and just images of trucks (orange). The x-axis shows the layer that the stitching layer immediately follows. Note that the performance jumps up once the special layer from A is included

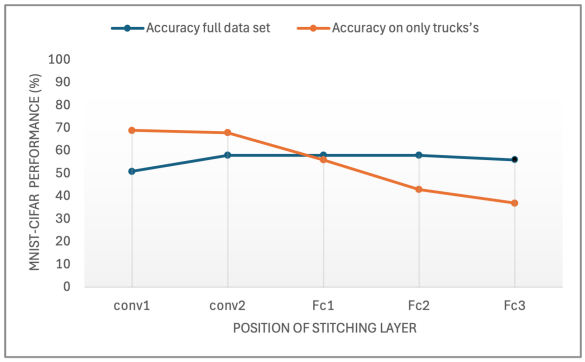
We test C's ability to detect trucks as a function of the position of the stitching layer, this is shown on Figure 2. For clarity, each point along the x-axis on the graph is a distinct stitched model; if the position of the stitching layer is Fc1 that means that all layers up to and including Fc1 come from model A (Conv1, Conv2, Fc1), then

the stitching layer, then all layers after come from model B (Fc2, Fc3, Fc4). Each graph corresponds to stitching with a distinct model A (A, A', A") which have been trained to have their stitching layers in the layers Fc1, Fc2, and Fc3 respectively; the same model B is used on all three graphs. On all three graphs we can see that model C is unable to recognise trucks until the special layer l_{Ai} is included in C, after which point it has comparable performance on trucks to the other 9 categories. Hence, if we knew there was a special layer where this trait was located, but did not know which one it was, we could scan the layers until we see this spike to identify it.

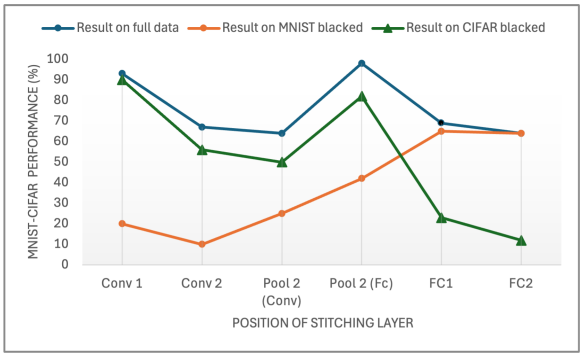
We have shown this applies for all three of the fully connected layers we can stitch with, we have not done this for the two convolutional layers as - due to their role as being general feature extractors - if all other layers are not trained on trucks, just training the convolutional layers cannot produce a model capable of recognising trucks.

4.3 Stitching can give a layer by layer description of a distributed trait

In this Section we want to establish that stitching can still meaningfully carry over traits that are not localised to a single layer. Instead of having a single special layer, model A is wholly trained on the complete data set and model B is entirely trained on data lacking trucks. Again we plot model C's performance as the stitching layer moves down the model, shown in Figure 3a. The stitched model gets progressively worse at recognising trucks as the number of layers from A decreases, seeming to imply that A's ability to recognise trucks is roughly equally spread throughout its fully connected layers.



(a) Truck-blindness increasing (performance on trucks decreasing) as the number of truck-blind layers increases



(b) MNIST performance increasing and CIFAR performance decreasing as MNIST model layers are added CIFAR model layers are removed.

Figure 3: Graphs of the performance of stitched models at a task. The blue line is the competency on the full data set, the red and green lines show how the stitched model exhibits the traits increasingly as layers from the model possessing that trait increases. "Full data" means neither side is blacked out.

Figure 4: A example of an input for the MNIST-CIFAR models

To confirm this concept we repeat with a different set of traits. Consider images made of a complete MNIST digit and a suitably rescaled CIFAR picture concatenated together, called an MNIST-CIFAR image (see Figure 4). The dataset is ordered and paired such that all 0's are with a plane, and all planes are with a 0 (and so on), hence a model could learn either the MNIST or CIFAR side as a sufficient method for categorising all images. Model A is trained such that it learns only the MNIST side, only generic training is required for this to occur, as MNIST is a far easier dataset to learn than CIFAR. Model B is trained on the CIFAR side by decorrelating the MNIST digit from the label. This means the two traits we consider are performance on the

MNIST side, and performance on the CIFAR side. We can see from Figure 3b that as the stitching layer moves, the first trait decreases steadily while the second trait increases steadily. The interpretation we give for this is that the respective traits are roughly evenly distributed throughout the models, and as more layers from a model are added, the better the stitched model becomes at that corresponding task. Note also the clear spike in performance on the right hand graph at the point labelled 'Pool2 (Fc)'. This occurs because instead of the usual low capacity convolutional stitching layer (the adjacent data point) we use a high capacity, fully connected layer which is powerful enough to increase performance across the board. This is not relevant to the conclusions of this section, but it is included as it will be commented on in Section 6.

4.4 Stitched as a means of locating simplicity

We build on the result from the previous section's results that if a model has a trait distributed throughout its layers, then as more layers from that model are included in the stitched model, the more strongly we see that trait being expressed in the stitched model. Using this, we move to the crux of the paper about whether this can be used to examine the complexity of a model layer by layer. We use the three quantitative measures of simplicity (gradient norm, loss sharpness, and compression) introduced in Section 2.3.

We began with using the MNIST-CIFAR setup from the previous Section with the same CNN's. The argument being that MNIST is a far simpler dataset to learn than CIFAR, this is a well established result, and intuitive as MNIST is black and white numbers, whereas CIFAR is coloured and has categories which are not simply identified by curves and lines. Therefore, model A would presumably be far simpler than model B (an intuition we confirmed with all three metrics).

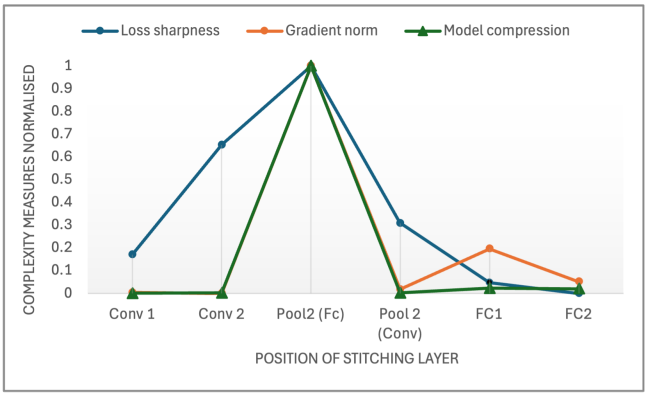


Figure 5: This shows the complexity of the stitched model by all three metrics (normalised). Model compression is in units of bits, the other two are unit-less.

Recall that for this graph each point along the x-axis is a new stitched model, with each one having more layers from model B and less from model A as we move from left to right. When we performed this experiment (Figure 5) we did not see the expected result of a steady increase in complexity as layers from model B were added. Instead, the complexity peaks sharply in the middle, and most tellingly the complexity varies strongly between the two models stitched after the layer Pool2, based on the type of stitching layer used. This implies that the complexity of the stitched model is dominated by the fact that it is a stitched model (and how it is stitched), not from the complexity of the two models used to create it.

To resolve this, we hypothesised that if the two models being stitched were far larger, and hence far more complex, then the added complexity of the stitching layer could be rendered negligible. We switched to using ResNet-18 architecture as a compromise between being significantly larger than our previous CNN models, but still small enough to be feasible for training and stitching. One point of note is that ResNet uses residual connections between layers (meaning that the output of layers is used as part of the input for layers other than the one directly after it), so we chose to stitch in such a way to avoid residual connections passing over the stitching layer (limiting us to 4 stitching locations: after the 5th, 9th, 13th, and 17th layers) as this could complicate the concept of stitching beyond the scope of this paper.

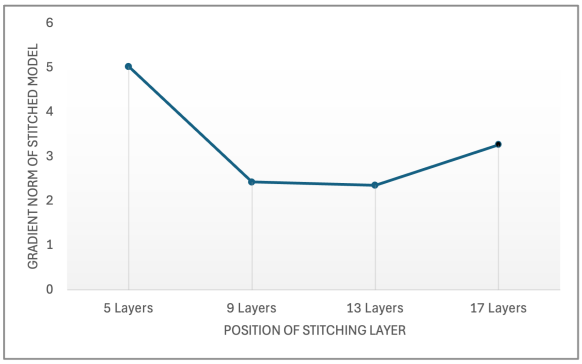
Due to the magnitude of ResNet18 it requires more compute to train the models, so we switch back to solely MNIST data (easier to learn), and use models partially pre-trained on ImageNet. As both models now use the same data, we altered the training methods to ensure the difference in complexity. Model A we train with L1 regularisation and intermittent pruning (setting random weights to zero), model B we train with L2

regularisation and no pruning.

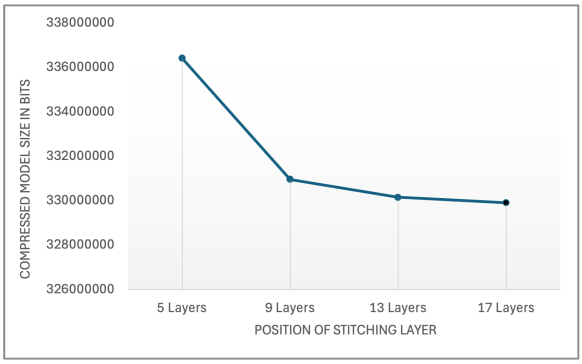
L1 Regularisation
$$\mathcal{L}_{\text{L1}} = \mathcal{L}_0 + \lambda \sum_{j=1}^{p} |w_j| \tag{4}$$

L2 Regularisation
$$\mathcal{L}_{\text{L2}} = \mathcal{L}_0 + \lambda \sum_{j=1}^{p} w_j^2 \tag{5}$$

Where $\mathcal{L}_0$ is the default cross-entropy loss function, w_j are the weights of the models (p being the total number of weights), and λ being a parameter to moderate the strength of regularisation. Both methods add a penalty for having larger weights, but L2 particularly punishes the largest weights while L1 punishes all weights equally. The effect this has on the model is that model B will generally smooth out the size of the weights and biases to prevent any from being too big, but will still use all its neurons [13]. Model A will drive many weights to zero, effectively turning off the neurons for certain features that are not necessary (pruning helps to encourage this behaviour) [14]. This loss of features should make model A simpler, which is what we observe as these models have gradient norm of 1.57 and 5.53, and model compression of 330000000 bits and 344000000 bits (rounded to 3 significant figures).



(a) Complexity of stitched model measured as gradient norm (in arbitrary units) as the number of layers from the simpler model increases



(b) Complexity of stitched model measured via the number of bits once compressed using gzip

Figure 6: Complexity of stitched model as the stitching layer moves, increasing layers from the simple model and decreasing layers from the complex model

In Figure 6a we show the results of the experiment using these two models using the measure of gradient norm (loss sharpness produces the same curve, so is not shown). We see broadly that as we add more layers of the simpler model, the complexity of the stitched model decreases. Additionally the jump from the first to second layer has the largest change, implying that the first layer may have a dominant role in determining the complexity of the model, compared to the other layers. In Figure 6b we show the results of the same experiment of stitching a more compressible model into a less compressible model, again seeing the same overall trend. It is worth noting that the 4th data point of Figure 6a does not fit the expected trend, and the limited data points generally mean that these graphs, while indicative of the expected trends, are far from conclusive.

5 Conclusion

The purpose of this paper was to establish the premises that: traits can be inherited using stitching; stitching can be used to locate a trait within a network; and stitching can give a layer by layer description of a distributed trait. This set-up for the ultimate claim of the paper that stitching enables us to study the simplicity of networks on the level of individual layers and hence gain insight into how the simplicity bias comes about. We believe that this paper has shown reasonable evidence for the first three claims, and that Figure 6 shows sufficient preliminary evidence to the final claim as to justify further thought and experimentation.

The findings of this paper must be considered with regards to its limitations. One concern is that, although care was taken to ensure that the stitching layers were not high enough capacity to learn the traits the models possessed, they could potentially aid the expression of an existing trait. This could be an issue in models where the width of layers (and hence the capacity of the stitching layer) varies, particularly if precise measurements

are being made; this would be best resolved by using larger networks such that the capacity of the stitching layer becomes negligible. Furthermore, the use of only two metrics, while both are suitable, is a limitation of this paper due to the available computation. Stitching is compatible with most measures of simplicity, and further research could benefit by using more measures, in particular the Hessian [15] and VC dimension [16]. The final concern has to be that the culminating experiment of the paper relies on 4 data points to establish a trend.

Hopefully this can be viewed in the context that the proceeding points had experiments employing multiple models, traits, and architectures to demonstrate the conclusions robustly. The justification for further research would be that this is a novel approach, and could produce an insightful diagnostic tool for research in simplicity bias.

The next step would be to repeat the simple-complex stitching (Section 4.4) with a larger architecture; ResNet50 has 16 positions suitable for stitching (points avoiding the residual 'pass-forward'), and 16 points would be far more statistically significant. Future work is also needed to test the results of stitching in another domain of tasks (e.g. language recognition), as the usefulness of stitching is constrained until it is shown to apply more broadly for neural networks. Finally, in the outlook Section we outline an alternative way stitching could be brought into the machine learning toolkit - some preliminary work on very simple multi-specialised networks could be useful to test if the concept has any potential.

6 Outlook

For the remainder of this paper we will discuss additional properties and conclusions of stitching we have found worthy of note, but which stand separate from the main branch of the argument. In particular, we consider stitching not as a diagnostic tool but as a method of creating powerful, useful models by combining existing models with useful traits. These points are included for the sake of prompting discussion only; we do not put forward any evidence confirming these conclusions as they were not included in the scope of the paper, merely noted as a part of the other experiments.

6.1 Stitching for robustness

It is often seen that the data in a test set given to a model have multiple pieces of information, each of which is sufficient to identify it. For example, consider a data set which only includes swans or ducks, the model might simply learn to identify them by colour and ignore all other features, which makes the model far more susceptible to data corruption changing colour. Robustness can be ensured during training by altering some images before training, e.g. making some swan pictures brown; stitching is capable of ensuring robustness after models are fully trained.

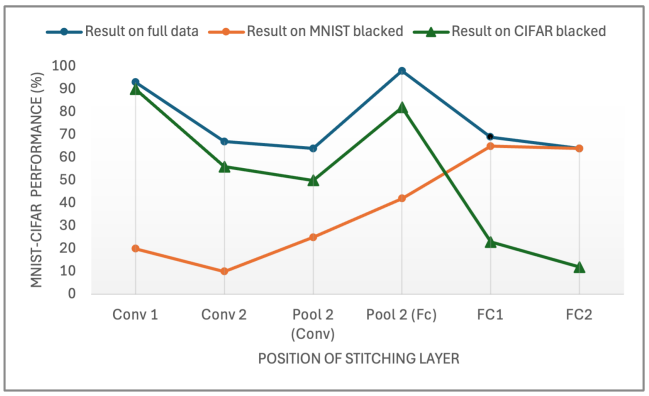


Figure 7: Performance on all three measures as a function of stitching layer position (repeat of Figure 3b)

Recall the earlier experiment on the MNIST-CIFAR data where model A learns MNIST and model B learns CIFAR, shown in Figure 7. This we focus on the result with the very high capacity fully connected layer. Before stitching, model A has 97% performance and model B has 63% performance on the full data set (each has the same performance on just their side of the data, and 0% on the other side of the data). When stitched with the high capacity layer, model C has 98% performance on the full data, 82% on solely the MNIST side and 42% on solely the CIFAR side. Therefore through stitching we have produced a model with equivalent total

performance to our best initial model while having reasonable performance on each of the sides individually, ensuring a robust performance.

6.2 Stitching can be used to rapidly produce multi-specialised networks

Consider an input (I) which consists of M pictures (P) concatenated together. Each image has a label, which belong to a unique set of ten possible categories. Hence we write the task as $I = P^1_{i_1} P^2_{i_2} P^3_{i_3} ... P^M_{i_M} = \{P^j_{i_j}\}_{j=1}^{M}$ where each index i_x takes a value from 1 to 10. To correctly identify the input the model must accurately identify all M labels.

Consider a task where the model is presented with, say, 10000 inputs, and the model earns 1 point for each input fully and correctly identified. Additionally, we are told that some time (T) before the task begins, one specific type of input I^* will be marked as important (e.g. $I^* = P^1_3 P^2_7 P^3_8 ... P^M_4$), and correctly identifying this input is worth 1000 points. Before T we know that one type of input will be marked as important, but not which. For an example of such a task, consider a model which is trained to identify people in CCTV footage. If we know that we are looking for a particular individual who has a certain height, build, hair colour, skin colour, clothing, etc then we might want to quickly produce an expert model for recognising people with all those characteristics.

To optimise this task, we need a model which can perform well at identifying most inputs I, but is an expert at recognising I^*. For a reasonably deep model we can assume T is a far too short time to fully train a model with an emphasis on I^*. This leaves three reasonable strategies:

1. Have 1 pre-trained model. Use the time T to fine tune the model on inputs I^* or very similar inputs
2. Have 10^M different pre-trained models which are specialists for each possible special input I^*. Use the time T to simply choose the correct specialist.
3. Have $10 \cdot M$ pre-trained specialist models, one for each $P^j_{i_x}$, and use time T to stitch the required M models together to create a stitched specialist at I^*

Clearly method 2 is far more demanding than method 3, and it is not a priori clear that method 1 would outperform method 3. If method 3 outperforms method 1, then stitching could provide a sweet spot between performance and computational demand.

References

[1] Chris Mingard, Henry Rees, Guillermo Valle-Pérez, and Ard Louis. Do deep neural networks have an inbuilt occam's razor? *arXiv:2304.06670v1 [cs.LG]*, Apr 2023.

[2] Jing Luo, Huiyuan Wang, and Weiran Huang. Investigating the impact of model complexity in large language models. *arXiv preprint arXiv:2410.00699*, 2024.

[3] Yakir Berchenko. Simplicity bias in overparameterised machine learning. In *The Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI-24)*, 2024.

[4] Thao Nguyen, Maithra, and Simon Kornblith. Do wide and deep networks learn the same things? *arXiv:2010.15327v2 [cs.LG]*, Apr 2021.

[5] Nishant Keskar, Dheevatsa Mudigere, Ping Tak Peter Tang, Ping Tak Peter Tang, Huan Wang, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2017.

[6] Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. Sharp minima can generalize for deep nets. *arXiv preprint arXiv:1703.04933v2*, 2017.

[7] Zhiying Jiang, Matthew Y.R. Yang, Mikhail Tsirlin, Rapael Tang, Yiqin Dai, and Jimmy Lin. Low-resource text classification: A parameter-free classification method with compressors. 2024.

[8] Karel Lenc and Andrea Vedaldi. Understanding image representations by measuring their equivariance and equivalence. *arXiv:1411.5908v2 [cs.CV]*, Jun 2015.

[9] Yamini Bansal, Preetum Nakkiran, and Boaz Barak. Revisiting model stitching to compare neural representations. *arXiv:2106.07682v1 [cs.LG]*, Jun 2021.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[11] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[12] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[13] Aitor Lewkowycz and Guy Gur-Ari. On the training dynamics of deep networks with l2 regularization. In *Advances in Neural Information Processing Systems*, volume 33, pages 21232–21243, 2020.

[14] Rodolphe Jenatton, Jean-Yves Audibert, and Francis Bach. Structured variable selection with sparsity-inducing norms. *Journal of Machine Learning Research*, 12:2777–2824, 2011.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Flat minima. *Neural Computation*, 9(1):1–42, 1997.

[16] Vladimir Naumovich Vapnik and Alexey Yakovlevich Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971.

7 Appendix 1

Code for the three simplicity measures

```
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import numpy as np
from PIL import Image
import random
import gzip
import pickle
import copy


def compute_loss_sharpness(model, loss_fn, testloader, epsilon=1e-3):
    """Computes the sharpness measure by perturbing model parameters."""
    model.eval()
    total_sharpness = 0.0
    num_batches = 0

    original_params = {name: p.clone() for name, p in model.named_parameters()}

    for data, target in testloader:
        data, target = data.to(next(model.parameters()).device),
        target.to(next(model.parameters()).device)

        model.zero_grad()
        output = model(data)
        original_loss = loss_fn(output, target).item()

        # Perturb parameters
        for p in model.parameters():
            p.data += epsilon * torch.randn_like(p)

        perturbed_output = model(data)
        perturbed_loss = loss_fn(perturbed_output, target).item()

        # Restore original parameters
        for name, p in model.named_parameters():
            p.data = original_params[name]

        total_sharpness += perturbed_loss - original_loss
        num_batches += 1

    return total_sharpness / num_batches if num_batches > 0 else 0.0


def compress_model_numpy(model):
    weights = np.concatenate([p.cpu().detach().numpy().flatten() for p in model.parameters()])
    model_bytes = pickle.dumps(weights)
    compressed = gzip.compress(model_bytes)
    return len(compressed) * 8  # Size in bits


def compute_gradient_norm(model, loss_fn, testloader):
    """Computes the average gradient norm over the test set."""
    model.eval()
    total_norm = 0.0
    num_batches = 0
```

```
for data, target in testloader:
    data, target = data.to(next(model.parameters()).device),
    target.to(next(model.parameters()).device)

    model.zero_grad()
    output = model(data)
    loss = loss_fn(output, target)
    loss.backward()

    batch_norm = torch.sqrt(sum(p.grad.norm()**2 for p in model.parameters() if p.grad is not None)
    total_norm += batch_norm.item()
    num_batches += 1

return total_norm / num_batches if num_batches > 0 else 0.0
```

8 Appendix 2

This is example code for stitching two CNN models together and training the stitching layer on the MNIST-CIFAR dataset, with the stitching layer after the first pooling layer. This is does not include the code required for uploading the data nor code for evaluating the stitched model afterwards - the full file can be found on the GitHub page.

```
net1 = torch.load('model_cifar_mnist_2b_full.pth')
net2 = torch.load('model_cifar_mnist_2c_full.pth')

    #Stitching network
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.stitch = nn.Conv2d(6, 6, 1)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(16 *13*5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.stitch(x)
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 13 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net3 = Net()

# Get the state_dict for each model
state_dict_1 = net1.state_dict()
state_dict_2 = net2.state_dict()
state_dict_3 = net3.state_dict()

# Iterate over the state_dict keys and copy weights
keys = list(state_dict_3.keys())
for i, key in enumerate(keys):
    if i < 2:
        # First half from model_A
        state_dict_3[key] = state_dict_1[key]
    elif i > 3:
        # Second half from model_B
        state_dict_3[key] = state_dict_2[key]

# Load the modified state_dict into model_C
net3.load_state_dict(state_dict_3)
net = net3

import torch.optim as optim

criterion = nn.CrossEntropyLoss()
```

```
# Set up the optimizer for the unfrozen parameters
optimizer = torch.optim.SGD(filter(lambda p: p.requires_grad, net.parameters()), lr=0.001, momentum=0.9


    epochs = 7
    # Freeze all layers initially
    for param in net.parameters():
        param.requires_grad = False

    #Unfreeze stitching layer
    for param in net.stitch.parameters():
        param.requires_grad = True

    for epoch in range(epochs):  # loop over the dataset multiple times

        running_loss = 0.0
        for i, data in enumerate(combined_loader, 0):
            # get the inputs
            inputs, labels = data

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()
            if i % 1000 == 999:    # print every 1000 mini-batches
                print('[%d, %5d] loss: %.3f' %
                      (epoch + 1, i + 1, running_loss / 2000))
                running_loss = 0.0
    print('Finished Training.')
```