Neural Network Surgery with Sets

Jonathan Raiman[*]
Dali
jonathan@dali.ml

Susan Zhang[*]
OpenAI
susan@openai.com

Christy Dennison
OpenAI
christy@openai.com

Abstract

The cost to train machine learning models has been increasing exponentially [1], making exploration and research into the correct features and architecture a costly or intractable endeavor at scale. However, using a technique named "surgery" OpenAI Five was continuously trained to play the game DotA 2 over the course of 10 months through 20 major changes in features and architecture. Surgery transfers trained weights from one network to another after a selection process to determine which sections of the model are unchanged and which must be re-initialized. In the past, the selection process relied on heuristics, manual labor, or pre-existing boundaries in the structure of the model, limiting the ability to salvage experiments after modifications of the feature set or input reorderings.

We propose a solution to automatically determine which components of a neural network model should be salvaged and which require retraining. We achieve this by allowing the model to operate over discrete sets of features and use set-based operations to determine the exact relationship between inputs and outputs, and how they change across tweaks in model architecture. In this paper, we introduce the methodology for enabling neural networks to operate on sets, derive two methods for detecting feature-parameter interaction maps, and show their equivalence. We empirically validate that we can surgery weights across feature and architecture changes to the OpenAI Five model.

1 Introduction

The computational cost of training neural networks has been shown to be growing at an exponential rate [1]. The ability to repurpose or fine-tune machine learning models for different tasks has in parallel been touted as an approach to reap the benefits of a costly investment in a high capacity model. Starting with word vectors [2], base layers of computer vision models, and lately with generative models, Transformers, and reinforcement learning agents, larger and larger components from a trained model can be used on a new task or modified to satisfy new requirements, and attempts have been made to understand how best to achieve this. For instance, [3] used linear classifier probes to try and determine the utility of layers within a network, while [4] looked at which portions of a model can be transferred. In simpler cases, it has been found that the output of a network is sufficiently similar to an existing task, and thus the majority of the network can be kept as is [5, 6, 7, 8].

While some transfer cases have natural boundaries along which parameters can be transferred, we find that domains with semantically rich and complex input features pose an important difficulty to overcome when attempting to gradually incorporate them into a growing model. In the development of OpenAI Five[2] [9], the input domain would periodically change to incorporate new heroes or additional observable features that were deemed necessary to master the game across changes in game

[*] equal contribution

[2] OpenAI Five is a deep reinforcement learning agent trained using selfplay that defeated the world champions OG at the game DotA 2.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

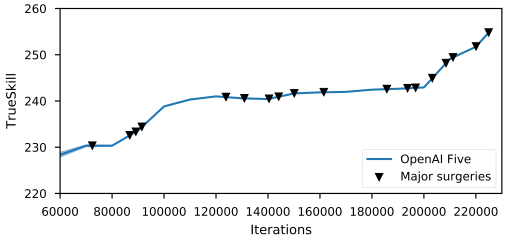
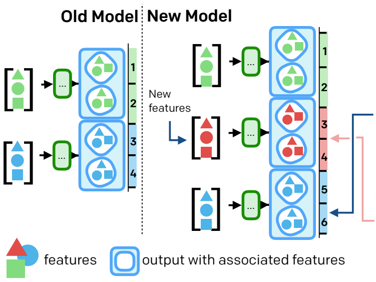


Figure 1: Over the course of training, OpenAI Five went through 18 major surgeries. Surgery events are marked on the skill curve. We measure progress using matches between the training model and reference opponents to compute a TrueSkill [10] rating.



Using feature groups as a key, we can notice that parameters previously interacting with inputs at location 3 and 4 **have shifted** to 5, 6, and we should copy over those parameters from the predecessor model. The feature groups for **input 3 and 4 have no precedent**, and thus should be initialized without copy.

Figure 2: When comparing feature-parameter interaction maps across an old and new model, we can notice the introduction of new features and whether features that exist across both generations have shifted places.

versions, as depicted in Figure 1. These changes would pose significant challenges for continued training of the model, motivating the development and formalization of a methodology named "surgery" for automatically discovering which parameters to port over.

The main contributions of this work are a formalization of the discovery of surgery steps for porting the parameters from one network to another across feature or architecture changes, two implementations using gradients or boolean logic operations, and empirical validation of surgery applied to feature changes in OpenAI Five.

2 Approach

2.1 Surgery Overview

For a change to a model F_{old} with parameters Θ_{old}, we would like to define a mapping $\mathcal{M} : \Theta_{old} \rightarrow \Theta_{new}$ of model parameters for a new model F_{new} such that

$$F_{old}\left(X_{old}^{in}(s); \Theta_{old}\right) = F_{new}\left(X_{new}^{in}(s); \Theta_{new}\right) \qquad \forall\ s \in \mathcal{S} \tag{1}$$

where s is a state of the world $\mathcal{S}$, $X_{old}^{in}(s)$ and $X_{new}^{in}(s)$ are the corresponding featurized inputs to each model at a state s. In order to define this mapping $\mathcal{M}$, we require the following:

1. An ordered and named schema for all input features in $X_{old}^{in}(\cdot)$ and $X_{new}^{in}(\cdot)$.
2. A mapping Φ_{old} from each input feature $x_i^{in}(\cdot) \in X_{old}^{in}(\cdot)$ to each parameter $\theta_j \in \Theta_{old}$ indicating the interaction between the two within the network. A separate mapping Φ_{new} is obtained for features in X_{new}^{in} and parameters in Θ_{new}.
3. The difference between Φ_{old} and Φ_{new} is computed. Changes indicate areas where new parameters need to be inserted to preserve output equivalence between F_{old} and F_{new}.

2.2 Input feature schema

Let $X^{in}(s)$ denote the input features for a given state s that is fed into some model F, which takes the form $X^{in}(s) = [x_{\text{max health}}(s), x_{\text{attack damage}}(s), \ldots]$. We would then record an index mapping of: $\{\text{max health} \rightarrow 0,\ \text{attack damage} \rightarrow 1, \ldots\}$. We call this snapshot of feature identifiers to their corresponding indices within the model input X^{in} an **input feature schema**. For notational simplicity we will omit s from X^{in} past this point.

2.3 Input feature to parameter mapping

Let $\Theta = \{\theta_i\}_{i \in \mathcal{L}}$ be the set of learnable parameters within a model F, where $\mathcal{L}$ is the set of layers within a given model. We need to define a mapping $\Phi : x_i^{in} \rightarrow \theta_j \quad \forall\ x_i^{in} \in X^{in}, \theta_j \in \Theta$ which connects input features to the parameters in the model for which these features interact with. Binary operations between inputs[3] and parameters are defined as "interactions", and we can derive this interaction mapping using either gradients or boolean logic.

2.3.1 Gradient mapping

One method of obtaining Φ is to look for cases where parameters receive a nonzero gradient. For example, in the case of a single fully-connected (FC) layer with an element-wise differentiable activation function f, let us consider the $N \times M$ matrix of weights W, a $1 \times M$ vector of biases b, and a $1 \times N$ input vector X. We now have $Y = f(X \cdot W + b)$, where Y is the output of this layer. If we have a cost function C that takes Y as its input: $C(Y) = C\left(y_1, y_2, \ldots, y_M\right)$, then the gradient of the cost function with respect to the weights of this layer is:

$$\frac{\partial\, C(Y)}{\partial W} = \sum_{k=1}^{M} \left(\frac{\partial C}{\partial y_k} \frac{\partial y_k}{\partial W} \right) \tag{2}$$

and since we have

$$\frac{\partial y_k}{\partial W} = \begin{bmatrix} \frac{\partial y_k}{\partial w_{1,1}} & \cdots & \frac{\partial y_k}{\partial w_{1,M}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_k}{\partial w_{N,1}} & \cdots & \frac{\partial y_k}{\partial w_{N,M}} \end{bmatrix} \tag{3}$$

where $y_k = f\left(\sum_{i=1}^{N} x_i \cdot w_{i,k}\right)$, the matrix of partials $\frac{\partial y_k}{\partial W}$ reduces down to all zeros except for the k-th column. Furthermore, if we were to set the input vector X to be all zeroes except for a 1 at index t, the matrix of partials $\frac{\partial y_k}{\partial W}$ would consist of simply a single nonzero value at index (t, k). Thus from feeding in inputs X_t that is nonzero only at index t, and computing the gradient, we can back out all the elements in W that need to be modified if an element of X was changed, since the nonzero values would all reside on row t of $\frac{\partial C(Y)}{\partial W}$.

Using gradients to define Φ is practical as it relies on existing auto-differentiation tools [11] to collect information. However, poor initialization and saturating functions can mask interactions. We resolve these issues in practice by initializing (non-bias) weights to be strictly positive, normalizing the inputs into sigmoid/tanh to avoid saturation, and replacing $max\text{-}pool$ by $avg\text{-}pool$.

If we were to have more than two layers, we would not be able to distinguish changes to the weight matrix in the second layer under the assumptions given in our surgery setup. That is, given positive weight matrices in our network and the assumption that we do not saturate any of our activation functions, we cannot construct inputs to the second layer that are nonzero at a single index without solving for complex systems of equations. As a result, the gradient of the cost function at the output of the second layer with respect to the weight matrix in the second layer would be nonzero throughout, and we would not be able isolate changes to the second layer parameters as a function of input changes.

While the limitation to a single FC layer may appear to be somewhat restrictive by design, we can still sufficiently cover all cases where input features to the model are added or removed. All input features into the model will be fed through some layer after which the output dimension is changed either through a matmul or a concat with another feature vector. In the case of matmuls, we have illustrated above that we can isolate the impact of individual inputs through the index of nonzero gradients. In the concat case, say with the output of two FC layers processed over two sets of inputs, we can similarly isolate changes in the ordering of the concat arguments by looking at the position of feature groups as shown in Figure 4.

2.3.2 Boolean logic mapping

Another method for defining Φ relies on tracking explicitly when and where a particular input feature is involved in a binary operation with a model parameter. This interaction is stored through a list of boolean flags [4] corresponding to each input feature. Every model parameter will maintain its own list

[3] Note that we distinguish between **inputs** and **input features**. **Inputs** are used to reference inputs to any layer in the network, including hidden states, whereas **input features** refers solely to observations directly taken from the world $\mathcal{S}$ and fed into the network initially.

[4] We use bitfield [12] to efficiently store feature sets.

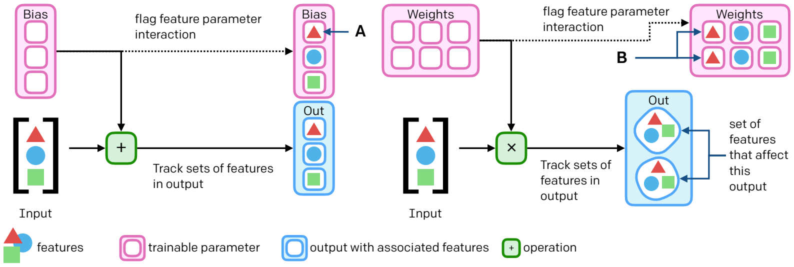

Bias
flag feature parameter interaction
Bias
A
Out
+
Track sets of features in output
Input
Weights
flag feature parameter interaction
Weights
B
×
Track sets of features in output
Out
set of features that affect this output
Input
features
trainable parameter
output with associated features
+
operation

Figure 3: Neural network is evaluated using boolean operations that track the propagation of features (shown as triangles, circles, and squares) using sets.

Interaction	Side effects	Output features
$w_1 \odot w_2$	-	$\emptyset$
$w \odot x$ or $x \odot w$	$\mathcal{F}_w \rightarrow \mathcal{F}_w \cup \mathcal{F}_x$	$\mathcal{F}_x$
$x_i \odot x_j$	-	$\mathcal{F}_{x_i} \cup \mathcal{F}_{x_j}$

Table 1: Feature propagation rules in a neural network made of parameters w and inputs x. $\odot$ represents an additive or multiplicative interaction between two items.

of boolean flags, where a true value indicates the parameter's involvement in a binary operation with a given input feature. An illustration of the propagation and update of these feature flags is shown in Figure 3.

To create these per-parameter feature flags, we create a copy of the computational graph where the original network operations (matmul, addition, sigmoid, etc...) are implemented using boolean logic rules to track feature interactions between inputs and parameters. The rules are defined as follows. Let $\mathcal{F}_w$ denote the set of input features $x_i^{in} \in X^{in}$ that interact with a scalar parameter w, and let $\mathcal{F}_x$ denote the set of input features that feed directly into an input x. Using the rules shown in Table 1, we can combine feature sets whenever parameters and inputs interact through addition or multiplication. At the end of this process, we obtain a parameter to feature mapping, which we can then do a reverse lookup to create Φ. This process is also illustrated in Figure 3.

2.3.3 Gradient and boolean logic mapping equivalence

We can show that both gradient and boolean logic mappings generate the same Φ by noticing that the existence of a nonzero gradient for a parameter (when a single input is nonzero) relies solely on the outputs of the forward pass up to this point being positive. From (2), we can see that:

$$\left[\frac{\partial C(Y)}{\partial W}\right]_{a,b} = \frac{\partial C}{\partial y_b}\left[\frac{\partial y_b}{\partial W}\right]_{a,b} = \frac{\partial C}{\partial y_b} \cdot \frac{\partial}{\partial w_{a,b}} f\left(\sum_{i=1}^{N} x_i \cdot w_{i,b}\right) = \frac{\partial C}{\partial y_b} \cdot f' \cdot x_a \tag{4}$$

Since $\frac{\partial C}{\partial y_b}$ is assumed to be nonzero for all y_b, and f' is similarly designed to be nonzero, we get:

$$\left[\frac{\partial C(Y)}{\partial W}\right]_{a,b} \neq 0 \implies x_a \neq 0 \tag{5}$$

if there exists $w_{a,b} \in W$ that is multiplied with x_a. It follows that if we were to do a forward pass with an input vector X that is zero except for a 1 at index t, any nonzero outputs of $y_k \in Y$ would indicate an interaction between x_t and $w_{t,k}$, which is equivalent to the location of nonzero gradients within $\frac{\partial C(Y)}{\partial W}$.

In order for x_a to be positive it must be either an input feature that was set to 1, or the output of some other layer in the network. In Table 2 we describe how a positive output relies either on any input being positive (in the case of matmuls), or a positive input element at the same index for element-wise operations. We can also observe that the truth table under the positive output conditional is identical to the truth table from conditional yielding the presence of a feature x_i^{in} in the output[5], and thus

[5] Except in the case of the product of two inputs x_1 and x_2 which gradient mapping cannot trace if x_1 and x_2 are inputs with mutually exclusive features. However, we did not encounter this case in practice.

Interaction	Positive output	Input feature x_i^{in} present in output
$x_1 \cdot x_2$	$x_1 > 0 \wedge x_2 > 0$	$(x_i^{in} \in \mathcal{F}_{x_1}) \vee (x_i^{in} \in \mathcal{F}_{x_2})$
$x \cdot w$	$x > 0$	$x_i^{in} \in \mathcal{F}_x$
$x_1 + x_2$	$x_1 > 0 \vee x_2 > 0$	$(x_i^{in} \in \mathcal{F}_{x_1}) \vee (x_i^{in} \in \mathcal{F}_{x_2})$
$x + w$	$x > 0$	$x_i^{in} \in \mathcal{F}_x$

Table 2: Boolean logic condition rules for propagating features when evaluating a neural network made of parameters w and inputs x.

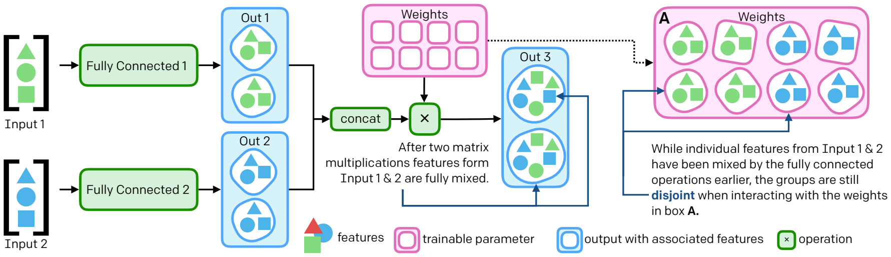

Input 1
Fully Connected 1
Out 1
Out 2
Input 2
Fully Connected 2
Weights
concat
×
After two matrix multiplications features form Input 1 & 2 are fully mixed.
Out 3
A
Weights
While individual features from Input 1 & 2 have been mixed by the fully connected operations earlier, the groups are still disjoint when interacting with the weights in box A.
features
trainable parameter
output with associated features
×
operation

Figure 4: The interaction between features and parameters can be tracked across one matmul or FC layer, but individual features will then be fully mixed. However, when concatenating groups of non-overlapping features as input to a deeper FC layer, the interaction between feature groups can still be distinguished (see box A).

propagation of positivity and the propagation of features operate identically. Gradient and boolean logic mappings are therefore equivalent, since they both only rely on detecting a positive value at x_a, and the operations that determine whether x_a is positive are identical.

2.4 Mapping Differences

The final stage for obtaining surgery steps is to compare Φ_{old} with Φ_{new}. We construct a lookup table using Φ that is keyed by feature groups and maps to parameters that interact with them[6]. Whenever new features are introduced, they will alter the feature groups so that Φ_{old} and Φ_{new} will not share this key in their lookup tables. We can use this mismatch to identify parameters that need to be reinitialized.

When new sets of features are introduced and concatenated with an existing input, the existing feature group keys will make it obvious that features have shifted location; this location shift information is useful for finding the parameters that should be copied from the old model. Figure 2 is an illustration new feature introduction and shift detection.

However, after one FC layer or matmul, features can become mixed (see Out 1 and 2 in Figure 4). making it impossible to track propagation past this point using purely input features. Nonetheless, it is possible to track the movement of feature groups that have been processed in isolation. For instance, in the case of OpenAI Five, information about allied and enemy units was processed by separate network components, and only later concatenated and fed to the main LSTM [9]. This architectural pattern is described pictorially in Figure 4, where Input 1 and 2 go through separate FC layers. Their outputs are later concatenated and multiplied by the same matrix (box A), and we can see how feature group movement, insertion, or deletion is visible in the interaction with this deeper weight matrix.

3 Results

To test whether we can use gradient or boolean logic mapping to automatically obtain surgery steps for transferrings parameters from one model version to the next, we add a group of 12 per-hero features and verify that we are able to preserve most of the trained weights when we use proper initialization and avoidance of zero-gradient functions.

[6] See boxes A and B in Figure 3, where the triangle feature refers to specific parameters in the bias or weights matrix

Mapping	Interactions Found	Params Transferred	Time (secs, $\mu \pm \sigma$)
Gradient (random + init.)	0.52%	90.13%	-
Gradient (random init.)	49.42%	94.91%	-
Gradient (random + init., max → avg)	**100%**	**98.68%**	907.92 ± 195.31
Boolean logic	**100%**	**98.68%**	37.88 ± 0.76

Table 3: Surgery correctness is sensitive to initialization and elimination of zero-gradient functions. Boolean logic and gradient mapping find the same interactions and transfer an equal number of parameters across feature changes. Surgery time comparison on a 2.9 GHz Intel Core i7 computer.

We conduct our experiments on the model that was used during the OG-OpenAI Five match. In Table 3 we report the percentage of total feature-parameter interactions detected, as well as the percentage of the original model's parameters that can be transferred to the new model under different initializations and zero-gradient function replacements. We find that random positive initialization and zero-gradient function replacement achieves the highest number of transferred parameters, and detects all feature-parameter interactions. Random initialization on its own masks interactions (49.42%) and prevents the same degree of transfer.

Unlike gradient mapping, boolean logic mapping has the desirable trait that it is agnostic to initialization or the presence of zero-gradient functions. In Section 2.3.3 we established equivalence between gradient and boolean logic mapping, and in Table 3 we empirically verify this holds and observe the the same number of parameters are transferred by both techniques.

A further advantage of boolean logic mapping is the ability to trace all feature-parameter interactions simultaneously, which we hypothesize should accelerate the map generation process; we measure time taken by both techniques and notice boolean logic mapping is ≈ 24 times faster (Table 3).

The underlying motivation for surgery is the ability to amortize training time while still being able to introduce new features. We find that surgery does indeed reduce the expected overall time to attain professional level play at DotA 2. Specifically, training from scratch (Rerun in [13]) took approximately 2 months, therefore retraining for each of the 20 major surgeries shown in Figure 1 would have taken 40 months, while training with surgery completed in only 10 months.

4 Conclusion

The exponential rise in computation found in machine learning experiments motivates the creation of tools that amortize the cost to train new models. Surgery has emerged as a powerful method for transferring trained weights across model changes and warm starting experiments without having to pay the price of starting from scratch. Knowing which parts of a model should be kept remains a complicated task that requires human labor and expertise.

In this paper, we introduce a methodology for automatically suggesting surgical steps, and present how it was successfully used to continuously train OpenAI Five across 20 surgeries over the course of 10 months. We show how we can evaluate a neural network using sets to trace the interaction between features and parameters deep inside the network. Building upon this technique, we present two different methods to generate these interaction maps and show their equivalence. We empirically validate that we can detect 100% of the feature-parameter interactions present in the model when we use a positive initialization and remove zero-gradient functions. By observing the difference in the interaction maps across a feature change we can transfer 98.68% of the parameters automatically.

We believe neural network surgery with sets can become a powerful tool to modify and reuse parameters in long running machine learning experiments. An important area for future work is determining how to robustly continue training surgeried models without scrambling transferred weights.

Acknowledgement

We wish to thank the anonymous reviewers for their valuable feedback and remarks, along with Jakub Pachocki and Szymon Sidor for their work on gradient-based surgery.

References

[1] Dario Amodei and Danny Hernandez. AI and compute. `https://openai.com/blog/ai-and-compute/`, 2018.

[2] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[3] Guillaume Alain and Yoshua Bengio. Understanding intermediate layers using linear classifier probes. *arXiv preprint arXiv:1610.01644*, 2016.

[4] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

[5] Justin Johnson, Andrej Karpathy, and Li Fei-Fei. Densecap: Fully convolutional localization networks for dense captioning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4565–4574, 2016.

[6] Nikita Kitaev and Dan Klein. Multilingual constituency parsing with self-attention and pre-training. *arXiv preprint arXiv:1812.11760*, 2018.

[7] Jonathan Raphael Raiman and Olivier Michel Raiman. Deeptype: multilingual entity linking by neural type system evolution. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[8] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. *arXiv preprint arXiv:1902.00751*, 2019.

[9] OpenAI. Openai Five. 2018.

[10] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill$^{\mathrm{TM}}$: a bayesian skill rating system. In *Advances in neural information processing systems*, pages 569–576, 2007.

[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[12] Steve Stagg. bitfield: Python fast integer set implementation. `https://github.com/stestagg/bitfield`, 2013.

[13] OpenAI. Playing Dota 2 with Large Scale Deep Reinforcement Learning, 2019.