Deep Neuroevolution with Shared Parameters: Ensembling a Better Solution

Michael Painter
Stanford University
mp703@stanford.edu

Abstract

In this work, we first introduce two new function preserving transformations, which change the architecture of a network, while preserving its input-output characteristics. Our transforms are more general than previous work, and therefore have greater flexibility in how they can be used. One property of the transforms introduced is that they do not alter any existing weights, allowing the old network coexist with the new network. This is advantageous in the context of an efficient architecture search, as parameters can be shared to massively reduce computational and memory requirements.

1. Introduction

In the normal work flow of using neural networks for a machine learning task, many architectures are often tried and tested to select the best one for a desired application. Typically, neural networks are initialized with random weights, and each architecture tried is trained from scratch, which can be a time consuming process. To alleviate this problem, Chen *et al*. [2] propose the `Net2Net` family of operations. These operations are *function preserving*, and so can effectively transfer the knowledge from one network to another, allowing the resulting network to be quickly evaluated and speeding up the data-driven design process.

Additionally, in this work, we observe that even for moderately sized networks, convergence can be reached quicker by first using a smaller network for the initial training steps. The moderately sized (or large) network can then be initialized through a function preserving transform, requires considerably fewer updates to fine tune.

The first half of this work concerns the definition of two new function preserving transformations, we collectively name `Resnet2Resnet`, or `R2R` for short. In line with the convention set by `Net2Net`, we will name our two operations `R2WiderR` and `R2DeeperR` . `R2R` maintains some properties that `Net2Net` does not, such as including arbitrary randomness, and not altering any existing weights. We will discuss these properties in detail later, as well as their advantages.

When evaluating multiple settings for neural network architectures, the concept of *architecture search* comes into play, which describes an automated process. There is plenty existing literature on architecture searches, such as [15, 23, 24], and often they tend to lead to state of the art performance. Later, we explain that this could be because an architecture search can be considered as an optimization algorithm with respect to an architecture space. Typically, these algorithms have a huge computational cost, and are run using hundreds of GPUs. Therefore, as with any optimization problem, further literature has investigated more efficient algorithms, as we will discuss in section 2.3.

Finally, we will utilize the transformations defined in the first half of this paper to implement a architecture search algorithm involving parameter sharing and knowledge transfer, both via `R2R` transforms. By sharing the parameters, it becomes feasible to have many models in memory at once, and so we decide to implement an *evolutionary* algorithm over the architecture space.

2. Related Work

2.1. Inception Architectures

A commonly used architecture are the Inception architectures. Inception Architectures are built from Inception modules, which consist of multiple sized convolutions, and their outputs are concatenated together. In the design of the Inception architectures Szegedy *et al*. [19, 20, 18] were particularly concerned with avoiding "computational bottlenecks", that is, they wanted to make their network efficient to use. Naively implemented, the concatenated output has a very large number of filters, and so Szegedy *et al*. introduce 1×1 convolutions for dimensionality reduction, and helped to keep output volumes from each layer small.

A number of versions of the Inception network have been defined, the most modern two are Inception-v4 and InceptionResNet-v2 [18]. In InceptionResNet-v2, residual connections are added to the network, which means that rather than a network learning some function $h(x)$, it instead learns $\mathcal{F}(x)$ and outputs the function $h(x) =$

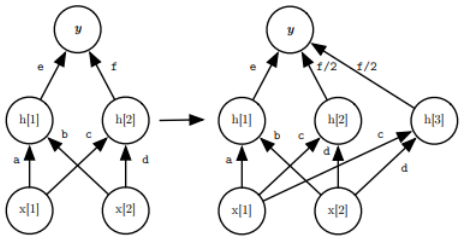

Figure 1. Example of a `net2WiderNet` transformation, where $h[2]$ is duplicated to $h[2]$ and $h[3]$. Reproduced from [2].

$x + \mathcal{F}(x)$. Residual connections were introduced by He *et al*. [6, 7] and are shown by Szegedy *et al*. to reduce training time when introduces to Inception type architectures [18].

2.2. `Net2Net`

When designing a network, we can think of transforming one function into another. For example, if we decided that our network did not have a sufficient representational capacity, we may decide to 'transform' the network into one with more filters per layer. However, when doing then need to perform training from scratch, and this can be a costly procedure.

The concept of *function preserving transforms* helps us to deal with this problem. If we have two networks f_1 and f_2, which we can consider to be parameterized functions, parameterized by say θ_1 and θ_2. If we, for some reason, had optimized f_1, but then decide that f_2 is a more suitable architecture to use, then it would be nice if we could make use of all the computation time spent optimizing f_1. To do this, we need to be able to find θ_2, such that $f_1(\cdot;\theta_1) = f_2(\cdot;\theta_2)$.

Chen *et al*. do exactly this, to define the `net2net` transformations [2], over the Inception-v4 architecture. Conceptually, the `net2WiderNet` transform works by replicating say one node into n, and then multiplying all of the outputs of each of the replicated nodes by $\frac{1}{n}$, as can be seen in figure 1. The `net2DeeperNet` simply adds a new layer to the network, initialized such that the layer is an identity function. These transforms are then generalized for convolutional layers. Chen *et al*. in their experiments train a smaller network, which they refer to as the *teacher network*, apply one of the `net2WiderNet` or `net2DeeperNet` transformations, to produce a *student network*.

2.3. Neural Architecture Search

A lot of work has already been done for neural architecture search [15, 23, 24, 1, 14].

In particular, an architecture search can be thought of as an optimization problem, as described in section 4.1, over some *architecture space*, such as [23].

One important consideration is that the majority of the work involving architecture searches tends to involve large companies, that have access to hundreds and thousands of GPUs. Other researches try to be a little smarter about how they perform the architecture search.

Cai *et al*. [1] utilize the `Net2Net` transforms that we have already seen, utilizing knowledge transfer to quickly evaluate architectures. A related, but different approach is taken by Pham *et al*. [14], who defined a computational graph, with shared weights. Each subgraph of the computational graph is a potential network to be used, and indeed, many of the subgraphs are searched over. Intuitively, this speeds up the evaluation of each model by also not having to train the model from scratch.

3. Function preserving transforms

In this section we will define our `R2R` transformations, `R2WiderR` and `R2DeeperR` . In general, both of these operations will construct a *student network* f_s from a *teacher network* f_t, specifying parameters θ_s and θ_t such that for all input volumes x we have $f_t(x;\theta_t) = f_s(x;\theta_s)$. Given this, we can think of the `R2R` transformations as functions the produce a student network (f_s, θ_s), given a teacher network (f_t, θ_t).

We will see in sections 3.2 and 3.3 that it is relatively simple to widen and/or deepen a convolutional neural network (CNN) when we know how to usefully construct networks that are identically zero.

3.1. Zero function initializations

To begin our derivation, we will consider how to construct a small CNN $z(\cdot;\theta_z)$ such that for all input volumes x we have $z(x;\theta_z) = 0$. We will call such a network *zero initialized*. This task turns out to be trivial, and can be achieved simply by multiplying zero, or by convolving with a zero kernel. We therefore will also specify that we want a parameterized function, maintaining as many degrees of freedom in introduced parameters.

Let $V \in \mathbb{R}^{C_o \times C_i \times k \times k}$ and $x \in \mathbb{R}^{C_i \times W \times H}$. We denote a two dimensional convolution as $V_{ij} * x_j \in \mathbb{R}^{W \times H}$, with $V_{ij} \in \mathbb{R}^{k \times k}$ and $x_j \in \mathbb{R}^{W \times H}$. Zero padding is applied appropriately to preserve the spatial dimensions. The convolutions implemented in CNNs are similarly denoted $V * x \in \mathbb{R}^{C_o \times W \times H}$. For notational convenience, we will

write out convolutions, using block matrices:

$$V * x = \begin{bmatrix} V_{11} & V_{12} & \dots & V_{1C_i} \\ V_{21} & V_{22} & \dots & V_{2C_i} \\ \vdots & \vdots & \ddots & \vdots \\ V_{C_o1} & V_{C_o2} & \dots & V_{C_oC_i} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{C_i} \end{bmatrix}$$
$$= \begin{bmatrix} \sum_\ell V_{1\ell} * x_\ell \\ \sum_\ell V_{2\ell} * x_\ell \\ \vdots \\ \sum_\ell V_{C_o\ell} * x_\ell \end{bmatrix}.$$

Now let $W \in \mathbb{R}^{\frac{C_o}{2} \times C_i \times k \times k}$, $\beta \in \mathbb{R}^{C'_o \times \frac{C_o}{2} \times 1 \times 1}$, keep $x \in \mathbb{R}^{C_i \times W \times H}$, and let σ be an arbitrary activation function. Now, we can define a function in terms of W, β, σ which is identically zero, using block matrix convolutions as follows:

$$\begin{aligned} z(x; \theta_z) &= \begin{bmatrix} \beta & -\beta \end{bmatrix} * \sigma \left(\begin{bmatrix} W \\ W \end{bmatrix} * x \right) \\ &\overset{(a)}{=} \begin{bmatrix} \beta & -\beta \end{bmatrix} * \begin{bmatrix} \sigma(W * x) \\ \sigma(W * x) \end{bmatrix} \\ &= \beta * \sigma(W * x) - \beta * \sigma(W * x) \\ &= 0. \end{aligned}$$

For brevity and simplicity, we have ignored any biases (which can just be initialized to zero), and we set $\theta_z = \{W, \beta, \rho(\sigma)\}$ where $\rho(\sigma)$ denotes any parameters associated with σ. We note that σ is fairly arbitrary, and can be any deterministic function that does not have inter channel dependencies (otherwise equation (a) does not hold). This means that batch normalization [8] can form part of the composition of σ, and that we allow σ to have trainable parameters.

We call an implementation of the above function z a *R2R block*. A schematic of a R2R block can be seen in figure 2, along with the parameter initializations required to make it an identically zero function. Finally, we emphasize that the repeated parameters are just initializations, so are trained *independently*.

3.2. R2WiderR

We can now very simply define the R2WiderR transformation in terms of a zero initialized R2R block z. Given a residual block $f_t(x; \theta_t) = x + F(x; \theta_t)$, we add z to apply the transformation. That is, we set $\theta_s = \theta_t \cup \theta_z$ and $f_s(x; \theta_s) = x + F(x; \theta_t) + z(x; \theta_z)$.

We can visualize what this looks like schematically in figure 3.

3.3. R2DeeperR

The R2DeeperR transform can also be easily defined in terms of a zero initialized R2R block z. Given a teacher network $f_t(x; \theta_t)$ we define the student network as follows $\theta_s = \theta_t \cup \theta_z$ and $f_s(x; \theta_s) = f_t(x; \theta_t) + z(f_t(x; \theta_t); \theta_z)$.

A schema for the transformation can be seen in figure 4. Note, here we defined this in terms of a single R2R block, at

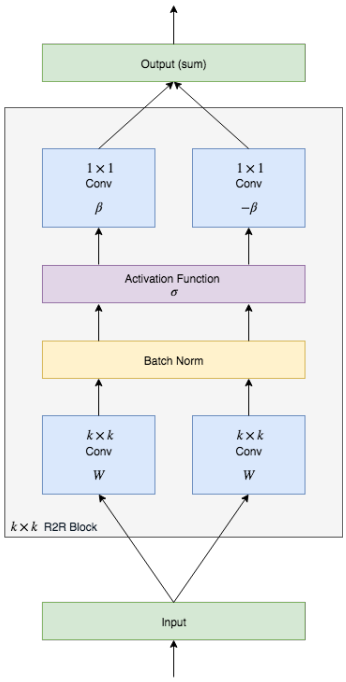

Figure 2. A schematic for a $k \times k$ R2R block, indicating (arbitrary) parameter initializations W, β needed to produce an identically zero output. For clarity in the schema, we show two parallel convolutional computations, however, it is equivalent to concatenate each pair of convolutions into a single convolution.

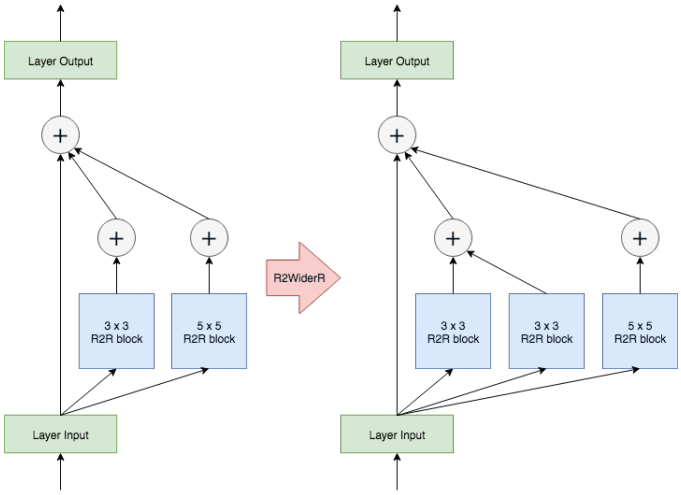

Figure 3. Example schematic for a R2WiderR transformation. It can be implemented by adding a R2R block in parallel, in an existing residual block.

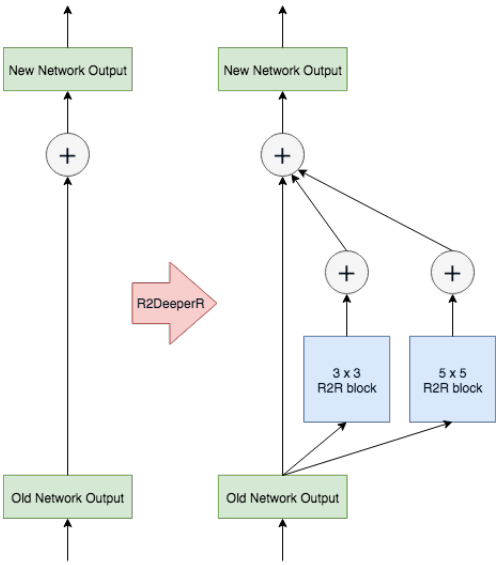

Figure 4. Example schematic for a R2DeeperR transformation. It can be implemented by adding a residual connection and zero initialized R2R blocks at the output of a network. Note that the output must have the same dimensions as the input.

the output of the network. It is simple to extend the idea to be able to arbitrarily add layers at any depth of the network, and also to add multiple R2R blocks in parallel.

3.4. Discussion

Now that we have defined the R2R transformations, we provide a discussion of how these transformations compare to the Net2Net transformations defined by Chen *et al*. [2].

Firstly, we believe that R2R transformations are easier to understand conceptually, and easier to implement that Net2Net transformations. In particular, to widen multiple layers in Net2Net , a *remapping inference* is needed, as widening one layer will change the input to the next. This is not the case in R2R , and no remapping inference is required.

Secondly, R2R transformations operate explicitly on networks with residual connections, whereas, Net2Net transformations can be adapted to work on networks with or without residual connections. This does not represent a problem, as Szegedy *et al*. [18] observe that residual connections empirically lead to faster training, without degraded performance.

One major difference between the R2R and Net2Net transformations are the degrees of freedom in new parameters. If $2K$ new parameters are added be a transform, then R2R transformations have K degrees of freedom, whereas, Net2Net has zero degrees of freedom. We believe that this could be a major advantage of R2R , as it is not obvious that the initializations of new parameters in Net2Net will be conducive to learning. With R2R we believe that novel initializations could be learned for W and β from section 3.1, and the situations may exist where R2R transforms are able to generalize and Net2Net transforms are not.

Finally, we note that the 1×1 convolutions implemented by β will produce a fixed output size, which may create *information bottlenecks*, preventing sufficient information being passed from lower layers to higher layers. In practice this did not represent problem, as we can set C'_o to be sufficiently large at the beginning of training, without too much overhead.

4. Architecture Search

4.1. Architecture search as optimization

We make the observation that previous work on architecture searches often leads to state of the art performance, such as the work of Zoph *et al*. [24]. We can make sense of results if we consider an architecture search formally as an optimization problem.

Consider an *architecture space*, or a set of functions $\mathcal{H}$. For each *architecture*, or parameterized function $h \in \mathcal{H}$, let Θ_h be the set of possible weights (i.e. if h has k parameters, then we likely set $\Theta_h = \mathbb{R}^k$). So an architecture is defined as a function of the form $h : \mathcal{X} \times \Theta_h \to \mathcal{Y}$.

Now, we can consider an optimizaton problem over the function $J : \mathcal{H} \times \Theta \to \mathbb{R}$, where we let $\Theta = \bigcup_{h \in \mathcal{H}} \Theta_h$. Note, that J is not differentiable with respect to $\mathcal{H}$, and therefore we lose the ability to perform purely gradient based optimization over J.

A naive architecture search can be thought of as an exhaustive search over the architecture space $\mathcal{H}$, and optimizing over Θ for each $h \in \mathcal{H}$ using Stochastic Gradient Descent (SGD). An exhaustive search is typically computationally inefficient, often requiring many GPUs and weeks of training.

As mentioned in 2.3, a number of methods can be used to make the search over $\mathcal{H}$ more efficient. Firstly, we can try to make the evaluation of each h more efficient, and reduce amount of computation needed to perform SGD each time. An example of this approach is parameter sharing, as in [14]. We say that two architectures *share parameters* if for $h, h' \in \mathcal{H}$, they satisfy $\Theta_h \cap \Theta_{h'} \neq \emptyset$. A second approach to evaluating architectures efficiently is to use function preserving transforms, such as Net2Net, to utilize knowledge transfer in evaluating new models [1].

Finally, we can improve upon an exhaustive search over $\mathcal{H}$ by using topological neuroevolution, as in [15]. This essentially performs a local search, if we were to appropriately define some distance metric between architectures. In this sense, it provides a more elegant way to search over $\mathcal{H}$.

Algorithm 1 NE(select, mutate, N_i, N_j, K, M, h_i)

let model_set = $\{h_i\}$
joint_train(model_set, N_i) ▷ Trains just h_i.
for 1 **to** num_iters **do**
  model_set = select(model_set, K)
  model_set = mutate(model_set, M)
  joint_train(model_set, N_j)
end for
return select(model_set, K)

Algorithm 2 select(model_set, K)

let selection = $\emptyset$
while $|\text{selection}|_c < K$ and $|\text{model\_set}|_c > 0$ **do**
  let h = select_best_model(model_set)
  remove h **from** model_set
  add h **to** selection
end while
return selection

Algorithm 3 mutate(model_set, M)

while $|\text{model\_set}|_c < M$ **do**
  randomly select h **from** model_set
  add $m(h)$ **to** model_set
end while
return model_set

Algorithm 4 joint_train(model_set, N_j)

for 1 **to** N_j **do**
  randomly select h **from** model_set
  perform training step over h
end for

4.2. Topological neuroevolution with parameter sharing

Prior to this work, the ideas presented in 4.1 could not be implemented in unison. However, given the R2R transformations defined in section 3, this is now possible.

Let m denote a stochastic operator, called a *mutation*, such that if $h \in \mathcal{H}$ then $m(h) \in \mathcal{H}$, and h and $m(h)$ share parameters. For our architecture search, we will implement m using R2R transformations, where the layer, operation and kernel size is randomly chosen. We can also include a threshold parameter τ, to assure some amount of 'locality' in the mutations, and m is limited so that $\frac{|\Theta_h|_c}{|\Theta_{m(h)}|_c} > \tau$, where $|\cdot|_c$ denotes cardinality.

The neuroevolution algorithm (NE) is outlined in algorithm 1, and is parameterized by the select and mutate functions. We begin by initially training a single, base network h_i, for N_i steps. Each iteration of the algorithm will consist of a selection phase (algorithm 2), where the 'best' K models are chosen, a mutation phase (algorithm 3) where the current set of models is mutated using m to generate M different models, with shared parameters. Finally, at the end of each iteration, we perform joint training (algorithm 4) for N_j steps, which on each iteration randomly selects a model and performs an optimization step on it.

There is still freedom as to how algorithm can be implemented, and how to implement "select_best_model" in algorithm 2. For now, we simply select the model with best performance on some validation set.

4.3. Co-evolutionary selection

One concept that we played around with, was an idea of 'co-evolution'. The idea is that a diverse population is maintained, which evolves collectively. This could be implemented through the following selection strategy: once a validation example has been used to select one network, it can no longer be used to select further networks.

Due to space, a full algorithm is omitted, as it involved a softer re-weighting of the validation set for making decisions. However, it is hopefully clear that this will hopefully lead to selecting networks that for ever validation example, at least one network can classify it correctly.

The top networks selected through this process can then be used as feature extractors to a new network, to form a *hierarchical ensemble*. This can be thought of as training a network to learn which of the co-evolved networks are correct.

5. Evaluation

During the implementation of sections 3 and 4 we made a number of practical observations. These are discussed in appendix C in more detail, so as not to clutter the main report.

5.1. Mnist and Cifar-10

During the development of this work, we used the Mnist dataset of handwritten digits [10], and the Cifar-10 [9] dataset.

The monochrome Mnist dataset consists of 60000 training images and 10000 test images, with a resolution of 28×28. As there are 10 different digits, there number of classes is 10. This is a simple dataset, with state of the are work often achieving results better than 99% classification accuracy [16].

Cifar-10 is slightly more challenging, with a more diverse set of 32×32 RGB images. The dataset again consists of 10 classes of objects, and is made up of 50000 training images and 10000 test images.

To achieve the best results on either of these datasets, data augmentation, such as [22] is required to both regularize the network and provide 'more' training examples.

We did not implement any data augmentation, to focus more on the results that could be achieved just by use of R2R transformations.

5.2. Testing R2R

To evaluate R2R we construct set of tests based around a small three layer network with residual connections. We will call this network the *base network*. A number of tests were performed both using Cifar-10, and using Mnist. Only the results and figures for the Cifar-10 tests are described, for brevity, however the results on Mnist are nearly identical (except for a different scale on the y-axis).

5.2.1 R2WiderR

To evaluate R2WiderR we construct a teacher network, with exactly half the number of filters at every layer compared to the base network.

First, we demonstrate that it is faster to train the base network by first training the teacher network, and then applying a R2WiderR transform, which is demonstrated in figure 5. Each model was trained for 6000 updates, and the R2WiderR transform was applied before updates 0, 1500, 3000 and 4500. Unsurprisingly, widening at step 4500 had the fastest training time, finishing in 136 seconds. Training the model from scratch (widen at step 0) took 44 seconds longer, or over 30% longer. Staggeringly, both networks converge to a validation accuracy of around 0.7. Results on the training set are similar, but omitted for space.

Secondly, we also reproduce the tests that Chen *et al.* perform, training the teacher network to convergence, and then fine tuning the base network after a R2WiderR transformation, which can be seen in figure 6. For comparison, we include training curves for if we had (completely) randomly chosen all *new* parameters when widening the network, and if we trained the base network from scratch. Regardless of how each network was initialized, each was trained for 7000 updates. It is easy to see that using random padding and R2WiderR reach convergence significantly faster than training the model from scratch. Precisely, R2WiderR converges at around 3000 updates, whereas the randomly initialized model has only just reached the accuracy of the teacher network at 7000 updates.

We do not need to count floating point operations in this test, as all updates are being performed over the same architecture.

5.2.2 R2DeeperR

To evaluate R2DeeperR , one can perform similar tests as in section 5.2.1. We decide to use the same base network, and for a teacher network, instead use a two layer network.

The first test can be seen in figure 7, with similar properties, but less exaggerated. Similarly to before all models are

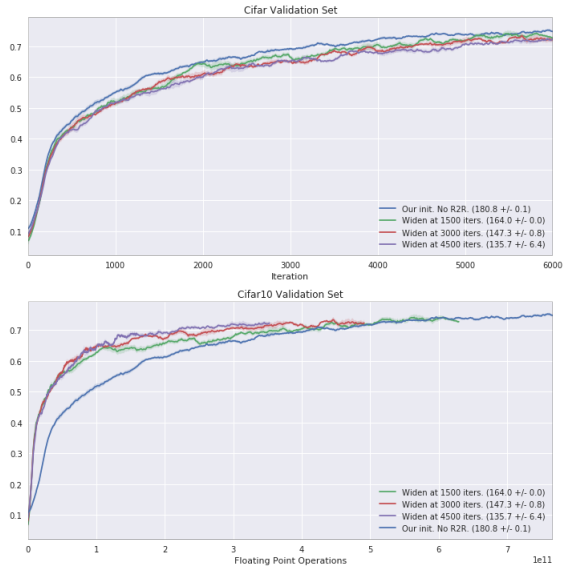

Figure 5. Four accuracy curves on the Cifar-10 validation set, one for applying R2WiderR at updates 0, 1500, 3000 and 4500. Top: Accuracy with respect to the number of *updates* performed. Bottom: Accuracy with respect to the number of *floating point operations* performed, so the later the R2WiderR transformation was applied the 'shorter' the line is.

updated 6000 times. By deepening after 4500 updates, the fastest network to be trained was 142 seconds, rather than the full 180 seconds to train the model from scratch. We can see that the accuracy curves hug each other far more tightly this time, but there is still some benefit to using R2DeeperR for the initial training steps.

Again, in the second test the teacher network is trained to convergence. Three networks are obtained by performing the R2DeeperR transform on the teacher network, by extending it with a new, randomly padded layer, and the last is formed by completely initializing the network from scratch. We see similar performance in figures 6 and 8.

Unfortunately, there seems to be little difference between the performance of a two layer network and a three layer network. So we see that the networks transformed from the trainer network seem to maintain a constant accuracy.

5.3. Neuroevolution tests

For neuroevolution, we evaluate based on the test performance at the end of training. To be as harsh as possible on the algorithms defined in section 4, base network is trained (the same three layer network as in section 5.2), for 10000 steps, and evaluate it on the test set frequently (every 500 updates). Our methodology is that we want the algorithms defined in section 4 to be able to outperform a single model that's even allowed to query the test set, without being able to query the test set itself.

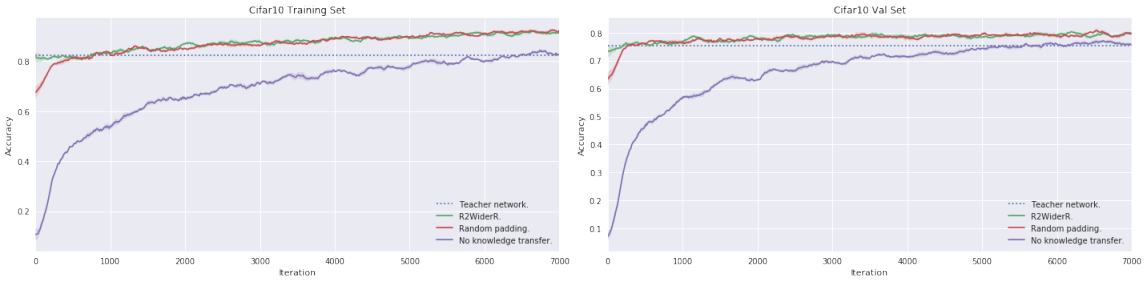

Figure 6. Reproducing the experiments for `Net2Widernet` in [2]. The blue dashed lines represents the teacher network accuracy, the green lines is initialized using the `R2WiderR` transformation, the red lines uses random padding, and the purple lines represent networks trained from scratch. Left: Cifar-10 training set performance. Right: Cifar-10 validation set performance.

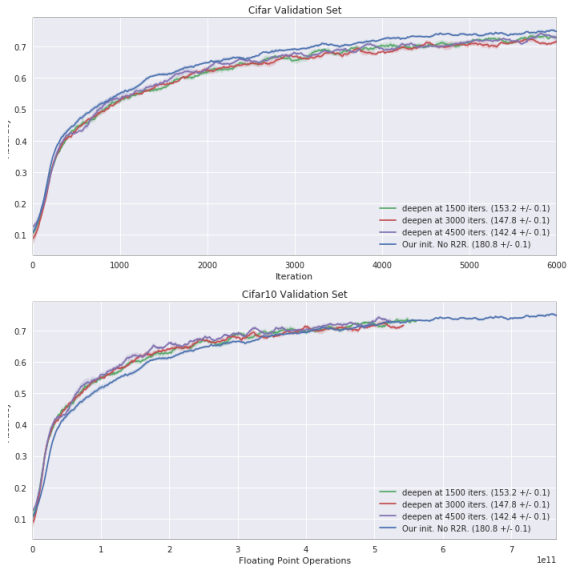

Figure 7. Similar to figure 5, but applying `R2DeeperR` rather than `R2WiderR` at updates 0, 1500, 3000 and 4500. Top: Accuracy with respect to the number of *updates* performed. Bottom: Accuracy with respect to the number of *floating point operations* performed, so the later the `R2WiderR` transformation was applied the 'shorter' the line is.

The algorithms followed are exactly as described in section 4, and training parameters and hyperparameters are as defined in appendix D.

6. Conclusions and future work

6.1. Conclusion

We first defined two new function preserving transforms that can be applied to networks containing residual connections. We believe that these transformations have some properties that could be desirable, such as preserving the old network.

Then, we defined a novel use of our new transformations,

Algorithm/Network	Test Accuracy
Base network (sampled test set)	0.70
NE (best model)	0.74
NE (Co-evolution selection, Best model)	0.72
NE (Co-evolution selection, Ensemble)	0.75

Table 1. Results on the test set for Cifar-10. Note that the test set was held out until the very end (including through the experiments in section 5.2), except for the Base Network. NE is short for Neuroevolution, and are the best performing network architectures found using the algorithms defined in section 4.

in the form of an architecture search, utilizing knowledge transfer from function preserving transforms, and

Finally, we ran a significant number of tests, illustrating the performance benefits of our two main contributions, via a classification task on the Cifar-10 dataset.

6.2. Future Work

There are many possibilities for future work, some of which are as follows.

Firstly, to complete this work, we really need to run the same set of experiments as in section 5 on the ImageNet dataset [4] and other larger datasets. We understand that Cifar-10 is a good dataset to quickly develop algorithms on, however, to seriously show significance, we need to evaluate on larger datasets.

Secondly, both `Net2Net` and our work has considered only feed-forward and convolutional neural networks. It would be interesting to see if any similar results could be found in other neural network architectures, such as recurrent neural networks (RNNs). One important thing to note if this is pursued, is that it is more common to use layer normalization [3] in RNNs, however, this would invalidate the assumptions made in equation (a) in section 3.1. To fix this, group, or instance normalization, as described by Wu and He [21] could be successfully used.

Thirdly, our work introduces some degree of randomness and/or freedom in new parameters added by function

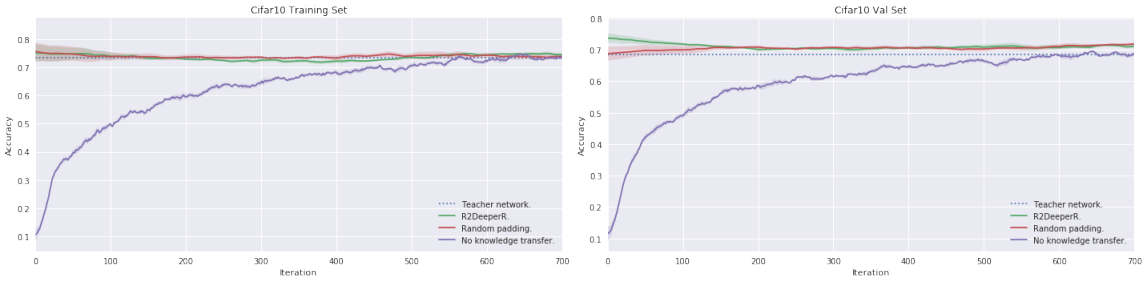

Figure 8. Similar to figure 6, however, the `R2DeeperR` transform has been applied as opposed to widening the trainer network using `R2WiderR` . Left: Cifar-10 training set performance. Right: Cifar-10 validation set performance.

preserving transforms. We observed that further training can be sensitive to magnitude of weights introduced. If the weights are initialized to maintain the statistical variance of parameters in each layer, then we obtained good results. However, we have not ruled out the possibility that better initialization may exist. It would be interested in seeing further work into theory behind how to best to initialize new parameters introduced, within the degrees of freedom permitted, and how the learning rate should be adapted at the time of network transformation.

Finally, recently there has been work in metalearning [5, 12], which is best described as follows: given a set of tasks, it learns "how to learn quickly". Nichol *et al*. propose the Reptile algorithm [12], which learns an initialization to learn quickly in new tasks. We had hoped to train an agent to enter OpenAI's retro contest [11]. Our plan began with training a PPO agent [17] jointly over Sonic levels, and use the Reptile algorithm over the full suite of games available in the Retro environment.Our hope was then to use the initialization learned (from Reptile) with `R2R` transformations on the PPO agent, to create an agent that began with better than random performance and could also adapt to new levels quickly. Unfortunately, training this agent would have required far more computational time and power than we reasonably had access to, and therefore is beyond the scope of this project.

References

[1] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang. Reinforcement learning for architecture search by network transformation. *arXiv preprint arXiv:1707.04873*, 2017.

[2] T. Chen, I. Goodfellow, and J. Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

[3] J. Chung, S. Ahn, and Y. Bengio. Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*, 2016.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[5] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

[6] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[7] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016.

[8] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[9] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. 2009.

[10] Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2, 2010.

[11] A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman. Gotta learn fast: A new benchmark for generalization in rl. *arXiv preprint arXiv:1804.03720*, 2018.

[12] A. Nichol and J. Schulman. Reptile: a scalable metalearning algorithm. *arXiv preprint arXiv:1803.02999*, 2018.

[13] D. Pakhomov, V. Premachandran, M. Allan, M. Azizian, and N. Navab. Deep residual learning for instrument segmentation in robotic surgery. *arXiv preprint arXiv:1703.08580*, 2017.

[14] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[15] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, Q. Le, and A. Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[16] S. Sabour, N. Frosst, and G. E. Hinton. Dynamic routing between capsules. In *Advances in Neural Information Processing Systems*, pages 3859–3869, 2017.

[17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[18] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12, 2017.

[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, et al. Going deeper with convolutions. Cvpr, 2015.

[20] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[21] Y. Wu and K. He. Group normalization. *arXiv preprint arXiv:1803.08494*, 2018.

[22] Z. Zhong, L. Zheng, G. Kang, S. Li, and Y. Yang. Random erasing data augmentation. *arXiv preprint arXiv:1708.04896*, 2017.

[23] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[24] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2017.

A. Code

Our implementation of R2R and the experiments described in this paper can be found in the GitHub repository: https://github.com/MWPainter/Deep-Neuroevolution-With-SharedWeights—Ensembling-A-Better-Solution.

B. Author Contributions

We made use of the FLOPs counting code written by Pakhomov *et al*. [13], which can be publicly found at https://github.com/warmspringwinds/pytorch-segmentation-detection.

Other than using a single file from the aforementioned repository, all work in this report is original.

C. Implementation Details

In this paper we have described the theory behind R2R transformations and defined our neuroevolution architecture search. During the practical implementation of we made a number of practical observations.

C.1. Implementing R2WiderR

In the R2WiderR transformation we can concatenate the new filters from R2R blocks added to old filters. This is a more efficient implementation (with respect to inference) of the R2WiderR transform, as it allows for greater parallelism in both the forward and backward passes during training. However, our experiments also implement the R2WiderR transformation as it is described, as it leads to a simpler implementation in 4.

Unless there is a benefit from the modularity of having many R2R blocks, we found the concatenation option preferable, as it seemed to be computationally more efficient.

C.2. Symmetry breaking

One important consideration for initializations is the idea of *symmetry breaking*. Using a constant initialization for the network or having many symmetries can lead to very poor performance in training, due to correlated updates. There was no observable difference by adding noise for breaking symmetry. This is likely explained by variance in the input being sufficient to break the one degree of symmetry that we introduce in an R2R transform.

C.3. Adaptive weight initializations and learning rates

In our experiments the performance was sensitive to the initialization of new weights introduced. Moreover, we similarly observed sensitivity to the setting of the learning rate.

We found that one particular initialization may lead to poor performance when a R2R operation is performed early in training, whereas, another may lead to poor performance when a R2R operation is performed later.

We reached a conclusion that both the learning rate and weight initializations need to be adapted through the training procedure.

As these need to be adapted through the training proceedure, it is not possible to just search for a good hyperparemters that happen to work, and we instead need to come up with some rules.

For the learning rate, we experimentally found that altering the learning rate so that updates maintain a roughly constant magnitude seemed to work well. If we apply an R2R transformation on $f_1(x;\theta_1)$ to produce $f_2(x;\theta_2)$ at time t, with the learning rate currently set as α, then in our experiments we set the new learning rate as:

$$\alpha' = \frac{|\theta_1|_c}{|\theta_2|_c}\alpha,$$

where $|\cdot|_c$ denotes cardinality.

We also tried directly utilizing the ratio $\frac{|\sum_{x\in\mathcal{M}} \nabla_{\theta_1} f_1(x;\theta_1)|}{|\sum_{x\in\mathcal{M}} \nabla_{\theta_2} f_2(x;\theta_2)|}$ of gradients over a minibatch $\mathcal{M}$ in our update, rather than the ratio of numbers of parameters. However, this either required a very large minibatch, which was very slow, or, added a lot of variance. On some training runs the learning rate was set close to zero due to noise, and killed the training process.

Finally, we remark that design choices made in this section are purely due to experimental results, and that investigating rules for initializations and learning rate schedules could provide interesting future work.

C.4. Dynamic computation graph

For our experiments, we used PyTorch to utilize its dynamic computation graph. This turns out to be important in section 4 where nodes in the computation graph for any neural network are often scrapped, and it was necessary to use a library where nodes could be easily removed from memory. When initially using TensorFlow we ran out of memory even in our R2R experiments, where there is no easy way to remove nodes from the computation graph.

D. Base architecture and hyperparameters

Training hyperparameters and neuroevolution training parameters used in all experiments are defined in table 2. The initial model used in all experiments is visualized in figure 9.

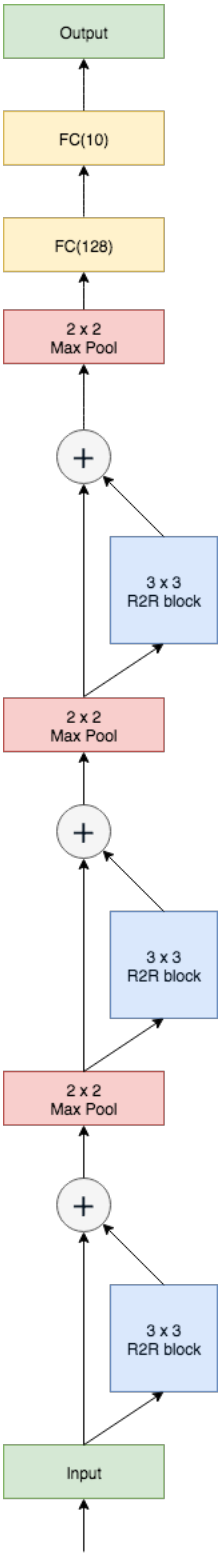

Figure 9. Initial network used in all tests

Parameter	Value
Optimizer	Adam
Learning Rate α	0.003
Weight Decay	10^{-6}
N_i	5000
N_j	5000
K	3
M	10
`num_iters`	15

Table 2. The training and hyperparameters used in all experiments.