Simple and Efficient Architecture Search for CNNs

Thomas Elsken
Bosch Center for Artificial Intelligence
& University of Freiburg
Thomas.Elsken@de.bosch.com

Jan-Hendrik Metzen
Bosch Center for Artificial Intelligence
JanHendrik.Metzen@de.bosch.com

Frank Hutter
University of Freiburg
fh@cs.uni-freiburg.de

Abstract

CNNs have recently had a lot of success. However, CNN architectures that perform well are still typically designed manually by experts in a cumbersome trial-and-error process. We propose a new method to automatically search for well-performing CNN architectures based on a simple hill climbing procedure whose operators apply network morphisms, followed by short optimization runs by cosine annealing. Surprisingly, this simple method yields competitive results, despite only requiring resources in the same order of magnitude as training a single network.

1 Introduction

Neural networks are often still designed by hand, which is an exhausting, time-consuming process. Therefore, a natural goal is to design optimization algorithms that automate this neural architecture search. However, most classic optimization algorithms do not apply to this problem, since the architecture search space is discrete (e.g., number of layers, layer types) and conditional (e.g., the number of parameters defining a layer depends on the layer type). Thus, methods that rely on, e.g., differentiability are not applicable. This led to a growing interest in using evolutionary algorithms (Real et al., 2017; Suganuma et al., 2017) and reinforcement learning (Cai et al., 2017; Zoph & Le, 2017) for automatically designing CNN architectures. Unfortunately, most proposed methods are either very costly (requiring hundreds or thousands of GPU days) or yield non-competitive performance. In this work, we aim to dramatically reduce these computational costs while still achieving competitive performance. Specifically, our contributions are as follows:

We propose a baseline method that *randomly* constructs networks and trains them with SGDR (Loshchilov & Hutter, 2017). We demonstrate that this simple baseline achieves 6%-7% test error on CIFAR-10, which already rivals several existing methods for neural architecture search. Due to its simplicity, we hope that this baseline provides a valuable starting point for the development of more sophisticated methods in the future.

We extend the work on network morphisms (Chen et al., 2015; Wei et al., 2016; Cai et al., 2017) in order to provide popular network building blocks, such as skip connections and batch normalization.

We propose Neural Architecture Search by Hillclimbing (NASH), a simple iterative approach that, at each step, applies a set of alternative network morphisms to the current network, trains the resulting child networks with short optimization runs of cosine annealing (Loshchilov & Hutter, 2017), and moves to the most promising child network. NASH finds and trains competitive architectures at a computational cost of the same order of magnitude as training a single network; e.g., on CIFAR-10, NASH finds and trains CNNs with an error rate close to 5% in roughly one day on a single GPU. Models from different stages of our algorithm can be combined to achieve an error of 4.7 % within

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

two days on a single GPU. On CIFAR-100, we achieve an error below 24% in one day and get close to 20% after two days.

We discuss related work in Section 2, formalize the concept of network morphisms in Section 3 and introduce our method in Section 4, which is then evaluated in Section 5. We conclude in Section 6.

2 Related work

Automated architecture search. In recent years, the research focus has shifted from optimizing hyperparameters to optimizing architectures. While architectural choices can be treated as categorical hyperparameters and be optimized with standard hyperparameter optimization methods (Bergstra et al., 2011; Mendoza et al., 2016), the current focus is on the development of special techniques for architectural optimization. One very popular approach is to train a reinforcement learning agent with the objective of designing well-performing CNNs (Cai et al., 2017; Baker et al., 2016; Zoph & Le, 2017). The latter two train their generated networks from scratch and evaluate their performance on a validation set, which represents a very costly step. Both trained over 10.000 networks, requiring hundreds to thousands of GPU days. To overcome this drawback, Cai et al. (2017) proposed to apply the concept of network morphisms within RL. As in our (independent, parallel) work, the basic idea is to use the these transformation to generate new pre-trained architectures to avoid the large cost of training all networks from scratch. Compared to this work, our approach is much simpler and 15 times faster while obtaining better performance. Real et al. (2017) and Suganuma et al. (2017) utilized evolutionary algorithms to iteratively generate powerful networks from a small network. Operations like inserting a layer, or modifying the parameters of a layer serve as "mutations" in their framework. Real et al. (2017) also used enormous computational resources. Very recently, Brock et al. (2017) used hypernetworks (Ha et al., 2017) to generate the weights for a randomly sampled network architecture with the goal of eliminating the costly process of training a vast amount of networks.

Network morphism/ transformation. Network transformations were first introduced by Chen et al. (2015) in the context of transfer learning. The authors described a function preserving operation to make a network deeper (dubbed "Net2Deeper") or wider ("Net2Wider") with the goal of speeding up training and exploring network architectures. Wei et al. (2016) proposed additional operations, e.g., for handling non-idempotent activation functions or altering the kernel size and introduced the term network morphism. As mentioned above, Cai et al. (2017) used network morphisms for architecture search, though they just employ the Net2Deeper and Net2Wider operators from Chen et al. (2015), i.e., they limit their search space to simple architectures without, e.g., skip connections.

3 Network morphism

A network morphism is a mapping from a neural network f^w with parameters w to another neural network $g^{\tilde{w}}$ so that $f^w(x) = g^{\tilde{w}}(x)$ for every x (network morphism equation, NME).

We now show how standard operations for building neural networks (e.g., adding a skip connection) can be expressed as a network morphism. Appendix A reviews already existing network morphisms.

Network morphism Type I. Let $f_i^{w_i}(x)$ be some part of a NN $f^w(x)$, e.g., a layer or a subnetwork. We replace $f_i^{w_i}$ by $\tilde{f}_i^{\tilde{w}_i}(x) = C(Af_i^{w_i}(x)+b)+d$ with $\tilde{w}_i = (w_i, C, d)$. A, b are fixed, non-learnable. The NME holds if $C = A^{-1}, d = -Cb$. A Batch Normalization layer (or other normalization layers) can be written in the above form: A, b represent the batch statistics and C, d the learnable scaling and shifting.

Network morphism Type II. Assume $f_i^{w_i}$ has the form $f_i^{w_i}(x) = Ah^{w_h}(x) + b$ for an arbitrary function h. We replace $f_i^{w_i}$, $w_i = (w_h, A, b)$, by

$$\tilde{f}_i^{\tilde{w}_i}(x) = \begin{pmatrix} A & \tilde{A} \end{pmatrix} \begin{pmatrix} h^{w_h}(x) \\ \tilde{h}^{w_{\tilde{h}}}(x) \end{pmatrix} + b \tag{1}$$

with an arbitrary function $\tilde{h}^{w_{\tilde{h}}}(x)$ and new parameters $\tilde{w}_i = (w_i, w_{\tilde{h}}, \tilde{A})$. The NME can trivially be satisfied by setting $\tilde{A} = 0$. A skip-connections by concatenation (Huang et al., 2016) can be formulated in this way: If $h(x)$ is a sequence of layers, then one could choose $\tilde{h}(x) = x$ to realize a skip from the input of h to the layer subsequent to h.

Algorithm 1 Network architecture search by hill climbing

function NASH($model_0, n_s, n_n, n_{NM}, e_{neigh}, e_{final}, \lambda_a, \lambda_s$)
  $model_{best} \leftarrow model_0$
  for $i \leftarrow 1, \ldots, n_s$ **do**
    for $j \leftarrow 1, \ldots, n_n$ **do**
      $model_j \leftarrow ApplyNetMorphs(model_{best}, n_{NM})$
      $model_j \leftarrow$ SGDRtrain$(model_j, e_{neigh}, \lambda_s, \lambda_a)$
    $model_{best} \leftarrow \underset{j=1,\ldots,n_n}{arg\,max}\{performance_{vali}(model_j)\}$
  $model_{best} \leftarrow$ SGDRtrain$(model_{best}, e_{final}, \lambda_s, \lambda_a)$
  return $model_{best}$

Network morphism Type III. Every $f_i^{w_i}$ is replaceable by $\tilde{f}_i^{\tilde{w}_i}(x) = \lambda f_i^{w_i}(x) + (1-\lambda)h^{w_h}(x)$, $\tilde{w}_i = (w_i, \lambda, w_h)$, with an arbitrary function h and the NME holds if λ is initialized as 1. An additive skip connection (He et al., 2016) can be inserted as follows: If $f_i^{w_i}$ itself is a sequence of layers, then one could choose $h(x) = x$ to realize a skip from the input of $f_i^{w_i}$ to its output.

Note that every combinations of the network morphisms again yields a morphism. So one could for example insert a block "Conv-BatchNorm-Relu" by exploiting this.

4 Architecture Search by Network morphisms

Our proposed algorithm is a simple hill climbing strategy. We start with a small, pretrained network. Then, we apply network morphisms to generate larger ones that may perform better when trained further. Due to the network morphism, the child networks start at the same performance as their parent. In essence, network morphisms can thus be seen as a way to initialize child networks to perform well, avoiding the expensive step of training them from scratch and thereby reducing the cost of their evaluation. The various child networks can then be trained further for a brief period of time to exploit the additional capacity, and the search can move on to the best resulting child network. This constitutes one step of our proposed algorithm, which we dub Neural Architecture Search by Hill-climbing (NASH). In our implementation, $ApplyNetMorphs(model, n)$ applies n network morphisms, each of them sampled randomly from the following three: 1) Make the network deeper. 2) Make the network wider. 3) Add a skip connection. See appendix B for more details. Since a lot of networks need to be trained, the child networks can only be trained for a few epochs. Hence, an optimization algorithm with good anytime performance is required - we chose the cosine annealing strategy from Loshchilov & Hutter (2017), whereas the learning rate is implicitly restarted: the training always starts with a learning rate λ_s which is annealed to λ_a after e_{neigh} epochs. We use the same learning rate scheduler in the final training (aside from a different number of epochs).

5 Experiments

5.1 Experiments on CIFAR-10

We first construct random networks by setting set $n_n = 1$ and $n_n = 8$ afterwards. This allows us to check whether the hill climbing strategy is able to distinguish between models with high and low performance. $model_0$ was a simple conv net, see Figure 2. As a second experiment we turn off the cosine annealing with restarts (SGDR) during the hill climbing stage: the training is done with a constant learning rate. We still use the cosine decay for the final training.

In the next experiment we investigate whether the network morphisms used in our algorithm harm the final performance of the final model as well as measuring the overhead of the architecture search process by comparing the times for searching and training a model with the time needed when retraining the final model from scratch.

Lastly, we compare our algorithm against the popular WRNs (Zagoruyko & Komodakis, 2016) and Gastaldi (2017) as well as other automated architecture search methods. Beside $n_s = 5$ as earlier, we also tried $n_s = 8$ to generate larger models. For further improving the results, we take snapshots of

Table 1: Results for CIFAR-10 and 100. "Resources spent" denotes training costs in case of the handcrafted models and retraining from scratch. For our experiments, we set $n_{NM} = 5$.

dataset	model	resources	# params (mil.)	error (%)
CIFAR-10	Shake-Shake (Gastaldi, 2017)	2 days, 2 GPUs	26	2.9
	WRN 28-10 (Loshchilov & Hutter, 2017)	1 day, 1 GPU	36.5	3.86
	Baker et al. (2016)	8-10 days, 10 GPUs	11	6.9
	Cai et al. (2017)	3 days, 5 GPUs	19.7	5.7
	Zoph & Le (2017)	800 GPUs, ? days	37.5	3.65
	Real et al. (2017)	250 GPUs, 10 days	5.4	5.4
	Saxena & Verbeek (2016)	?	21	7.4
	Brock et al. (2017)	3 days, 1 GPU	16.0	4.0
	Random networks ($n_s = 5, n_n = 1$)	5 hrs	4.4	6.5
	Ours($n_s = 5, n_n = 8$)	12 hrs	5.7	5.7
	Ours ($n_s = 5, n_n = 8$, retrained)	6 hrs	5.7	6.2
	Ours ($n_s = 5, n_n = 8$, no SGDR)	12 hrs	5.8	6.8
	Ours ($n_s = 8, n_n = 8$)	1 day, 1 GPU	19.7	5.2
	Ours (snapshot ensemble)	2 days, 1 GPU	57.8	4.7
	Ours (ensemble across runs)	1 day, 4 GPUs	88	4.4
CIFAR-100	Shake-Shake (Gastaldi, 2017)	7 days, 2 GPUs	34.4	15.9
	WRN 28-10 (Loshchilov & Hutter, 2017)	1 day, 1 GPU	36.5	19.6
	Real et al. (2017)	250 GPUs, ? days	40.4	23.7
	Brock et al. (2017)	3 days, 1 GPU	16.0	20.6
	Ours ($n_s = 8, n_n = 8$)	1 day, 1 GPU	22.3	23.4
	Ours (snapshot ensemble)	2 days, 1 GPU	73.3	20.9
	Ours (ensemble across runs)	1 day, 5 GPU	111.5	19.6

the best models from every iteration while running our algorithm following the idea of Huang et al. (2017).We also build an ensemble from the models returned by our algorithm across all runs.

Results for all experiments are listed in Table 1. The hill climbing strategy actually is able to identify better performing models. (5.7% vs. 6.5%). When turning off SGDR, the resulting models perform similarly to the models without any model selection strategy (6.8% and 6.5%, respectively). See also figure 5. When retraining models from scratch, performance slightly decreases (5.7% vs. 6.2%). This indicates that our algorithm does not harm the final performance. Our method outperforms most automated architecture search methods after one day on a single GPU. We do not reach the performance of the two handcrafted architectures as well as the ones found by Zoph & Le (2017) and Brock et al. (2017). Unsurprisingly, the ensemble models perform better.

5.2 Experiments on CIFAR-100

We run our algorithm with $n_s = 8, n_n = 8$ on CIFAR-100; all other (hyper-)parameters were not changed. The results are listed in Table 1. Our method is on a par with Real et al. (2017) after one day with a single GPU. The snapshot ensemble performs similar to Brock et al. (2017) and an ensemble model build from 5 runs can compete with the hand crafted WRN 28-10. The performance of Gastaldi (2017) is again not reached.

6 Conclusion

We proposed NASH, a simple and fast method for automated architecture search based on a hill climbing strategy, network morphisms, and training via SGDR. Experiments on CIFAR-10 and CIFAR-100 showed that our method yields competitive results while requiring considerably less computational resources than most alternative approaches. Our algorithm is easily extendable, e.g., by other network morphisms, evolutionary approaches or methods for cheap performance evaluation (e.g., learning curve prediction (Klein et al., 2017)). We hope that our approach can serve as a basis for the development of more sophisticated methods that yield further improvements of performance.

7 Acknowledgements

This work has partly been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant no. 716721.

References

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *ICLR 2017*, 2016.

J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *NIPS*, 2011.

Andrew Brock, Theodore Lim, James M. Ritchie, and Nick Weston. SMASH: one-shot model architecture search through hypernetworks. *arXiv preprint*, 2017.

Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Reinforcement learning for architecture search by network transformation. 2017.

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

Xavier Gastaldi. Shake-shake regularization. *ICLR 2017 Workshop*, 2017.

David Ha, Andrew M. Dai, and Quoc V. Le. Hypernetworks. *ICLR*, 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CVPR*, 2016.

Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. 2016.

Gao Huang, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E. Hopcroft, and Kilian Q. Weinberger. Snapshot ensembles: Train 1, get M for free. *ICLR 2017*, 2017.

A. Klein, S. Falkner, J. T. Springenberg, and F. Hutter. Learning curve prediction with Bayesian neural networks. In *International Conference on Learning Representations (ICLR) 2017 Conference Track*, April 2017.

I. Loshchilov and F. Hutter. Sgdr: Stochastic gradient descent with warm restarts. In *International Conference on Learning Representations (ICLR) 2017 Conference Track*, April 2017.

H. Mendoza, A. Klein, M. Feurer, T. Springenberg, and F. Hutter. Towards automatically-tuned neural networks. In *AutoML*, 2016.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc V. Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint*, 2017.

Shreyas Saxena and Jakob Verbeek. Convolutional neural fabrics. *arXiv preprint*, 2016.

Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. A genetic programming approach to designing convolutional neural network architectures. *arXiv preprint*, 2017.

Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. *arXiv preprint*, 2016.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint*, 2016.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. 2017.

A Network morphism

Network morphism Type I'. Let $f_i^{w_i}(x)$ be some part of a NN $f^w(x)$, e.g., a layer or a subnetwork. We replace $f_i^{w_i}$ by

$$\tilde{f}_i^{\tilde{w}_i}(x) = A f_i^{w_i}(x) + b, \tag{2}$$

with $\tilde{w}_i = (w_i, A, b)$. The network morphism equation obviously holds for $A = \mathbf{1}, b = \mathbf{0}$. This morphism can be used to add a fully-connected or convolutional layer, as these layers are simply linear mappings. Chen et al. (2015) dubbed this morphism "Net2DeeperNet".

Network morphism Type II'. Assume $f_i^{w_i}$ has the form $f_i^{w_i}(x) = Ah^{w_h}(x) + b$ for an arbitrary function h. We replace $f_i^{w_i}$, $w_i = (w_h, A, b)$, by

$$\tilde{f}_i^{\tilde{w}_i}(x) = \begin{pmatrix} A & \tilde{A} \end{pmatrix} \begin{pmatrix} h^{w_h}(x) \\ \tilde{h}^{w_{\tilde{h}}}(x) \end{pmatrix} + b \tag{3}$$

with an arbitrary function $\tilde{h}^{w_{\tilde{h}}}(x)$ and new parameters $\tilde{w}_i = (w_i, w_{\tilde{h}}, \tilde{A})$. Again, the network morphism equation can trivially be satisfied by setting $\tilde{A} = 0$. A layer can be widened (i.e., increasing the number of units/ number of channels - the Net2WiderNet transformation from Chen et al. (2015)). Think of $h(x)$ as the layer to be widened. For example, we can then set $\tilde{h} = h$ to simply double the width.

Network morphism Type IV. By definition, every idempotent function f_i (e.g. Relu) can simply be replaced by $\tilde{f}_i = f_i \circ f_i$.

B ApplyNetMorphs in detail

The function *ApplyNetMorphs* from our algorithm uniformly samples one of the three actions:

- Make the network deeper, i.e., add a "Conv-BatchNorm-Relu" block. The position where to add the block, as well as the kernel size ($\in \{3, 5\}$), are uniformly sampled. The number of channels is chosen to be equal to he number of channels of the closest preceding convolution.
- Make the network wider, i.e., increase the number of channels by using the network morphism type II. The conv layer to be widened, as well as the widening factor ($\in \{2, 4\}$) are sampled uniformly at random.
- Add a skip connection from layer i to layer j (either by concatenation or addition – uniformly sampled) by using network morphism type II or IV, respectively. Layers i and j are also sampled uniformly.

C Some figures

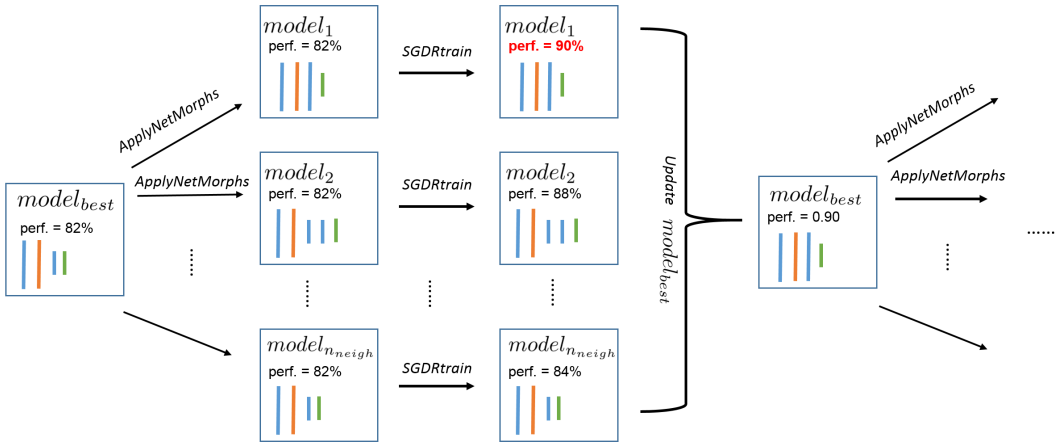

Figure 1: Visualization of our method. Based on the current best model, new models are generated and trained afterwards. The best model is than updated.

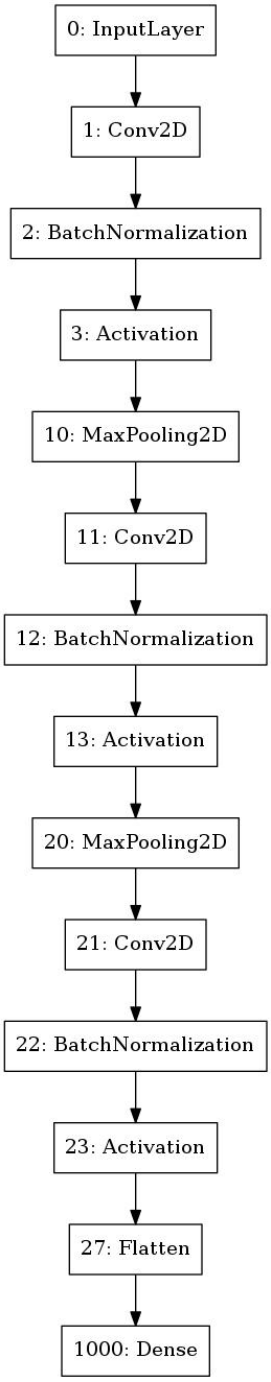

Figure 2: Initial network for our algorithm. The network is pretrained for 20 epochs.

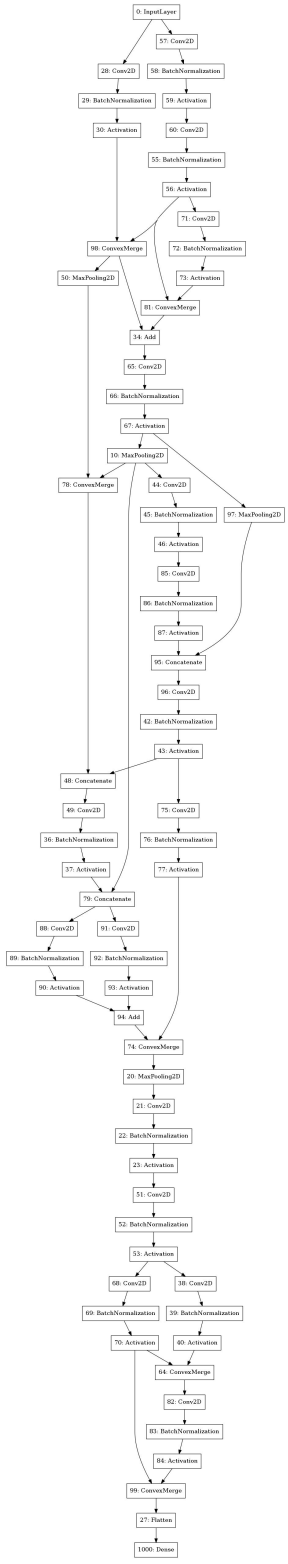

Figure 3: Network generated by our algorithm with $n_s = 5$.

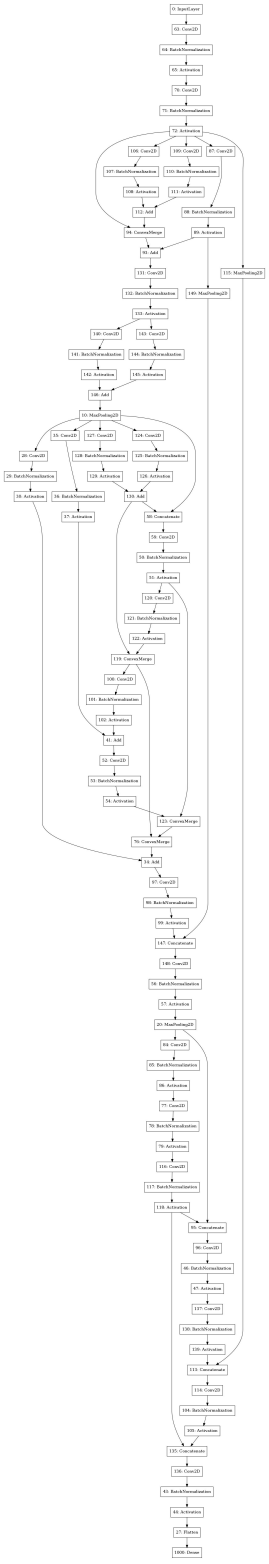

Figure 4: Network generated by our algorithm with $n_s = 8$.

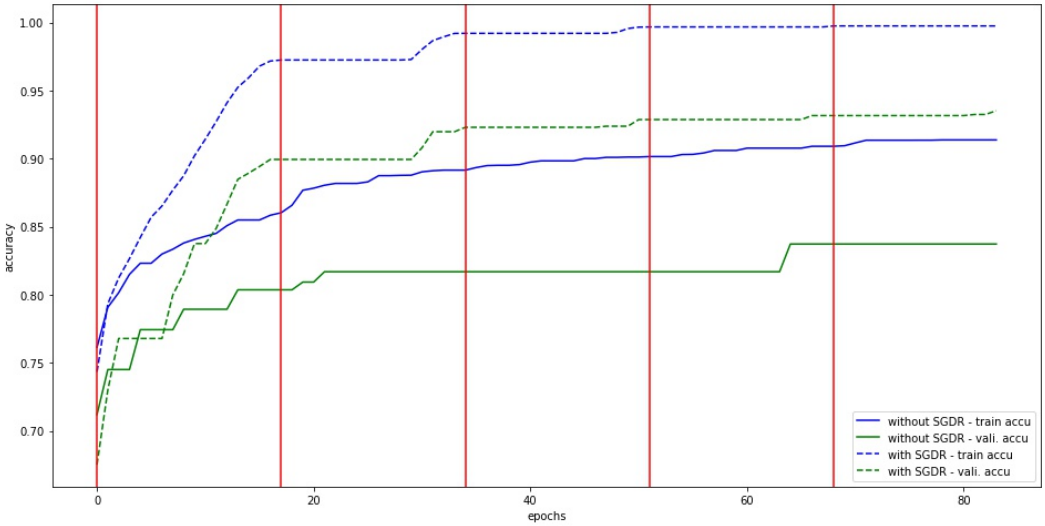

Figure 5: Exemplary learning curves across our algorithm. With and without training using SGDR in line 6 of our algorithm.