10 Evolutionary Neural Architecture Search

The design of neural network architectures, i.e. the organization of neurons into assemblies and layers and the connections between them, has played an important role in the advances in deep learning. Through a combination of human ingenuity and the need to push state-of-the-art performance, there have been several large leaps of technological innovation since the early 2010s. During this time, the technique now known as Neural Architecture Search (NAS) also emerged as its own subarea of deep learning research. The goal of NAS is to employ various methods such as reinforcement learning, gradient descent, Bayesian optimization, and evolutionary search to automate the search for novel neural network architectures, which are then trained with gradient descent to obtain the final network. The idea is that such automated search could result in architectures superior to those hand-designed by human researchers. Evolutionary optimization is particularly well suited for NAS because it can optimize not only continuous hyperparameter values, but discrete choices among alternative components, and even large structures such as graphs. Many evolutionary optimization techniques have found a new use in NAS, and new ones have been developed as well.

This chapter starts with a simple example combining NEAT topology search with backpropagation for the weights. It then expands to deep learning architectures, with examples in convolutional, recurrent, and general topologies. Particularly useful cases for NAS are multiobjective domains where aspects other than performance need to be optimized as well, and multitask domains where the needs of several tasks can be combined. NAS requires a lot of computation, so techniques have been developed for efficient search and evaluation. It may also be possible to evolve the networks entirely, without gradient descent as the second phase, in the future.

10.1 Neural Architecture Search with NEAT

The NAS idea can be illustrated by combining the NEAT topology search algorithm with the backpropagation algorithm for training the weights of each neural network topology. This concept of Backprop NEAT appeared many times even before deep learning, and in that sense it can be seen as the grandfather of modern NAS. Incidentally (as discussed in the Box below) it also encouraged the development of the NAS subfield within Google.

In Backprop NEAT, a neural network topology is evolved using the NEAT-style crossover and mutation operators. Unlike in the original version of NEAT, in this experiment many

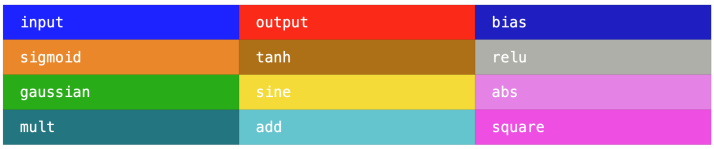


Figure 10.1: **Types of nodes and activation functions in the Backprop NEAT experiment**. The colors are used to label nodes in Figures 10.2 and 10.3. Different functions implement different computational properties that make the search for a good architecture more effective.

types of activation functions are possible, represented as different colors in the neural network (the legend is shown in Figure 10.1). The input to a neuron is the usual weighted sum of incoming connections. The `add` operator does nothing to the input, while the `mult` operator multiplies all the weighted inputs together. By allowing for a sinusoidal operator, the network can produce repetitive patterns at its output. The `square` and `abs` operators are useful for generating symmetries, and the *Gaussian* operator is helpful in drawing one-off clustered regions. The output neurons have sigmoid activation functions since the task consists of classifying examples into two categories (0 or 1).

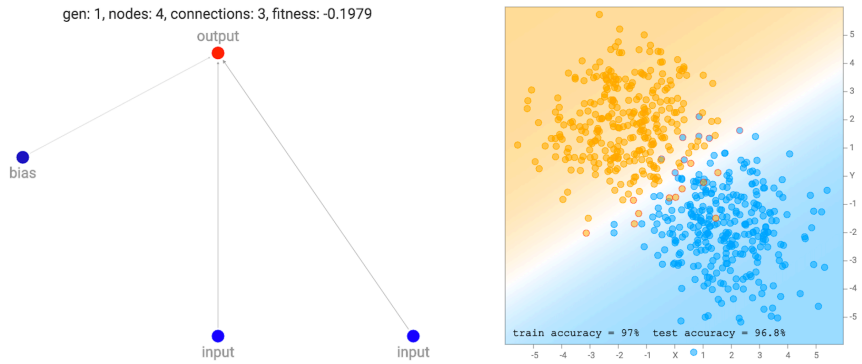
Each neural network topology that NEAT creates is represented as a computation graph. It is then possible to run backprop on this same graph to optimize the weights of the network to best fit the training data. In this manner, NEAT is strictly responsible for specifying the architecture, while backprop determines the best set of weights for it (in the original NEAT, evolution is also used to determine the weights). In this experiment, an L2 regularization term is also included in the backprop. The initial population of networks consists of minimal architectures like the one in Figure 10.2*a*, implementing logistic regression with a different set of random weights, i.e.

$$o = \sigma\left(w_1 x + w_2 y + w_3 b\right), \tag{10.30}$$

where x and y are the coordinates of the input sample, b is the bias unit (activated at 1.0), w_i are the initial random weights, and o is the output of the network. This simple network divides the plane into two halves as shown in Figure 10.2*b*. The color coding represents values from 0.0 (orange) through 0.5 (white) to 1.0 (blue). When the dataset consists of two Gaussian clusters, this simple initial network performs quite well already. In fact, when starting with an initial population of 100 simple networks with random weights, before any backprop or genetic algorithm, the very best network in the population is likely good enough for this type of dataset.

Each network architecture is assigned a fitness score based on how well they do in the classification task after training them with backprop. In addition to measuring how well each network fits the training data, using the maximum likelihood metric, the number of connections is also taken into account. Usually simpler networks are more regularized and thus generalize better to new examples, and also take less memory and are faster to run. Thus, simpler networks are preferred if they achieve similar regression accuracy than more complex ones, or if they are much simpler, even if they are somewhat less accurate. To achieve this goal, the fitting error is adjusted by the number of connections as

$$f = -E\sqrt{1+rc}, \tag{10.31}$$



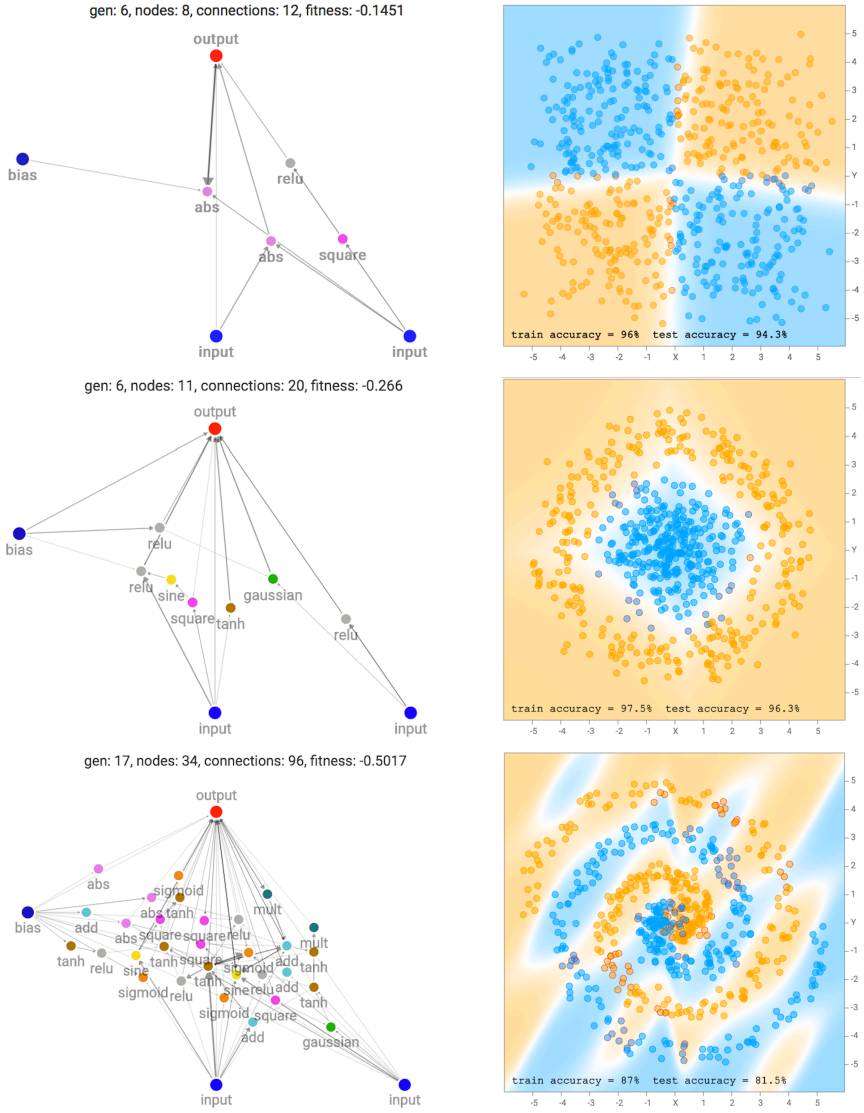
(*a*) Network architecture (*b*) Classification performance

Figure 10.2: **An example network from the first generation.** The task consists of classifying input samples (2-D points) into one of two categories (0/1). The initial population consists of networks that implement logistic regression with a different set of random weights. If the population is large enough and the classification problem simple enough, some of those initial networks may already do well in the task, as is the case in this nearly linearly separable classification task.

where f is the fitness, E is the error over the training set, c is the number of connections, and r is a proportionality factor. Thus, a network with more connections will have a fitness that is more negative than a network with fewer connections. The square root is used because intuitively it seems a network with e.g. 51 connections should be treated about the same as a network of 50 connections, while a network with five connections should be treated very differently than a network with four connections. Other concave utility functions may achieve the same effect. In a way, like the L2 regularization of weights, this type of penalty is a form of regularization on the neural network structure.

After a few generations, networks evolve that once trained, fit training data well, even in tasks that are not linearly separable (Figure 10.3). How is Backprop NEAT able to do it? In machine learning and data science in general, performance often depends on appropriate feature engineering, i.e. selecting or designing features that best represent the input. This approach has the advantage of incorporating known human expertise into the problem, making the learning task simple. For example, if the classification task consists of separating a small circle inside a big circle, the decision boundary is simply the distance from the origin. Constructing two new features by squaring each input dimension, most of the work has already been done for the network.

It is interesting to see whether NEAT can discover these features by itself without relying on human engineering. So, the raw inputs to each NEAT network will only be the x and y coordinates, and the bias $b = 1$. Any further features, such as squaring those variables, multiplying them, or putting them through a sinusoidal gate, will have to be discovered by the algorithm. Indeed, it can select the appropriate activation functions and network structure around them to implement useful features. For example with the XOR dataset, networks utilized `abs` and ReLU activation functions, which are useful in producing decision boundaries that are more or less straight lines with sharp corners (Figure 10.3). With concentric circles, the final network often included many sinusoidal, square, and Gaussian activation



(*a*) Network architecture (*a*) Classification performance

Figure 10.3: **Evolved Backprop NEAT networks for classifying data of varying complexity.** With XOR (top row), the architecture relies on `abs` and ReLU that allow the forming of long lines with sharp corners. In contrast with concentric circles (middle row), the architecture takes advantage of sinusoidal, square, and Gaussian functions to establish features that work well in such radially (nearly) symmetric domains, making the machine learning task easier. With concentric spirals, it further utilizes a complex topology to approximate the complex decision boundary. In this manner, evolution discovers hyperparameters and structures that work well for the task, similar to and possibly exceeding the ability of human engineers to design them.

functions, which makes sense given the radial symmetry of the dataset. With concentric spirals, which is almost symmetric but much more complex as well, the architectures utilized similar functions but also a complex topology that allowed it to match the complex decision boundary.

An interesting further observation is that networks that *backprop well* will tend to be favored in the evolution process, compared to networks with gradients that are unstable. A network with blown-up weight values is likely to perform poorly in classification, resulting in a poor fitness score. More generally, given a set of backprop parameters, such as a small number of backprop iterations or a large learning rate, evolution produces different kinds of networks, presumably those that learn well under such conditions. On the other hand, if the parameters are not set right, backprop may not find good weight values even if they exist, thus discarding a powerful architecture. Analogously, a person with extraordinarily high IQs may never reach their full potential if they live in a very harsh environment, or perhaps lack the people skills to influence their peers to accept their ideas. A solution in NAS is to make learning parameters evolvable as well. In that manner, good parameter values can be discovered together with architectures that work well with them. Such metalearning approaches are discussed further in Chapter 11.

10.2 NAS for Deep Learning

The Backprop NEAT experiment in the previous section introduced the concept of topology search for backpropagation neural networks. It illustrates the idea that even though gradient descent will optimize weights for a given neural network, it is also useful to optimize its hyperparameters and topology. This idea can be applied to modern deep learning as well. This section briefly outlines the history of NAS in deep learning, introduces the general approach, and reviews successes and challenges. Examples of prominent approaches and future directions are described in the sections that follow.

As deep learning rose in power and popularity, it became evident that simple fully-connected neural networks were not sufficient for most applications. Convolutional neural network (CNN) architectures grew more sophisticated, including AlexNet (Figure 10.4 Krizhevsky, Sutskever, and Hinton 2012), the winner of the 2012 ImageNet competition, which drew a lot of attention and essentially got us out of the neural network winter and into the era of deep learning. AlexNet led to the development of more complicated CNN architectures such as VGG (Simonyan and Zisserman 2014), Highway Networks (Srivastava, Greff, and Schmidhuber 2015) and Residual Networks (ResNet) (He et al. 2016b), and countless other architectures for computer vision.

Concurrently, for sequential tasks, people designed better recurrent neural network architectures that outperformed simple full-connected vanilla recurrent neural networks, such as Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber 1997), Gated Recurrent Unit (GRU) (Chung et al. 2014), and others. Most recently, with the introduction of the self-attention-based Transformer (Vaswani et al. 2017) architecture, there have been a host of proposals that claim to offer better, incremental performance to the original Transformer.

Much of this research was performed by graduate students who experimented with different architecture configurations, based on their hunch and instincts, who would try to

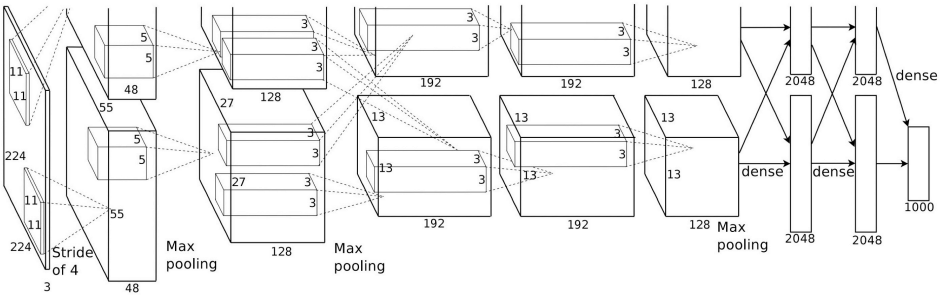


Figure 10.4: **The AlexNet deep learning architecture.** This architecture put deep learning into the spotlight when it won the ImageNet competition in 2012. There are careful engineering decisions that were involved in its design, including the principled organization into convolutional, pooling, and dense layers. More recent networks are often even more sophisticated and require a pipeline that spans network architecture and careful training schemes. Much manual labor is required in addition to the human insight to make them work, which suggests automated methods of configuring them might help. (Figure from Krizhevsky, Sutskever, and Hinton 2012)

experimentally discover new architectures that would offer some performance benefits compared to prior architectures. Some refer to this process as Graduate Student Descent (GSD), a joke on the Stochastic Gradient Descent (SGD) optimization process, hinting that the progress of machine learning research might be automated by a machine (Figure 10.5).

One of the main obstacles to the automated approach was that most deep learning tasks typically take several days to train. However, with the advent of large GPU computing clusters, it became feasible in the mid-2010s. The NAS subfield gradually emerged and became quite popular in the late 2010s. A form of graduate student descent applied to the area of NAS itself, and today, there are thousands of papers on the subject (for reviews, see Y. Liu et al. 2021; White et al. 2023), and even a popular, standardized benchmark for measuring the performance of NAS methods (Ying et al. 2019; Dong and Yang 2020; Zela et al. 2022).

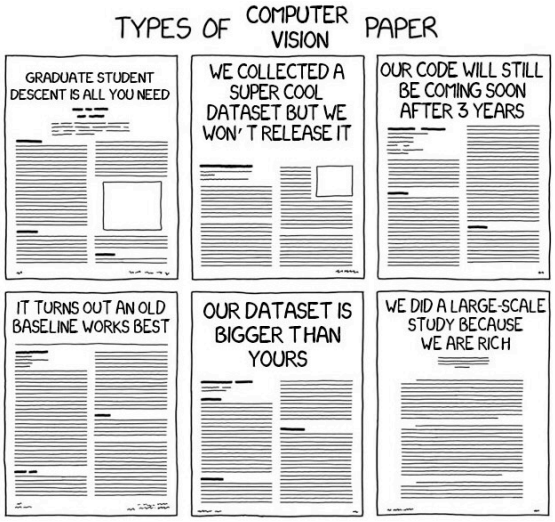


Figure 10.5: Graduate Student Descent (XKCD 2021). Some things never change! But now we have a name for it.

Info Box: Development of NAS inside Google Brain In a way, the development of NAS was related to the career path that prompted me (David Ha) to become a researcher at Google Brain and led me to conduct much of my nature-inspired research ever since. In 2016 I published the Backprop NEAT experiment (Section 10.1) as a personal blog post, and it somehow caught the attention of Jeff Dean, who reached out to me to comment on the concept of separating topology search and weight optimization, and had an interest to explore this idea deeper, potentially at Google scale. This conversation prompted me to apply and join Google Brain's residency program—in fact, Quoc Le (a co-author in the early NAS paper (Zoph and Le 2017)) was my first interviewer for the job! Quoc had a fantastic vision of developing a pipeline that could eventually automate much of the machine learning work at Google, which eventually became known as the AutoML project years later.

Quoc became my mentor and advisor, and we decided to explore two concepts: neural networks that generated weights (which became Hypernetworks (Ha, Dai, and Le 2016), my first project there), and neural network architecture search (a project led by Barret Zoph, who is a brilliant engineer and quickly learned to navigate Google's enormous compute resources with a fitting name, *Borg*!). The NAS project sought to apply topology search—define a search space for neural network architectures, and by leveraging Google's large compute resources, identify the architectures within the search space that will perform well on benchmark deep learning tasks such as image recognition or language modeling. This project got me started on large machine learning models, a path I'm on still today.

At around 2016, there were two dominant paradigms in deep learning: CNNs for image processing and RNNs for sequence processing (or some combination of CNNs and RNNs for spatial-temporal applications such as video processing). The architecture design problem for CNNs and RNNs looked quite different. For CNNs, it involved identifying the best combination of convolutional filters, which are great priors for image processing due to positional invariance property. Therefore, the task for designing, or automating the design of, CNN architectures required a search space that mainly focused on the edges (or the connections) of a graph. In contrast, sequential processing and sequence generation tasks relied on RNNs, which applied the same network architecture many times over, recurrently (hence the name). The essential element of the RNN is its memory node, i.e. a fixed structure that is replicated and activated many times. The search space mainly focused on the architecture of this node, i.e. its internal structure of cells, connections, activation functions, and specification of the state. In both cases, the problem was framed as a black-box optimization problem.

This automated search approach required enormous computation resources (Real et al. 2017); while the sampling process of architectures (the outer loop) is efficient, the calculation of the reward signal, or fitness for each candidate architecture (the inner loop) required training a neural network on the actual task. Computer vision benchmarks at the time, such as CIFAR-10, often required training the neural network for weeks on a single GPU. As a solution, researchers started to use proxies for the fitness function. For instance, for image classification, they would train for only a limited number of steps on CIFAR-10, and make the assumption that whatever metric had been achieved after n steps will be a good metric to rank the models (Miikkulainen, Liang, et al. 2023; Jiang et al. 2023; Rawal and Miikkulainen 2020). This is a good assumption since there is often a high correlation between the final performance and early-stage training performance of neural networks. Also, the tasks and benchmarks used for NAS were often smaller in scale. For instance, CIFAR-10 or a low-resolution version of ImageNet was used for training image classification models, and the Penn Treebank Dataset (PTB) was used for training language models. The authors would then demonstrate that the resulting models transfer to larger scale datasets, such as the full ImageNet or JFT-300M for images, and Wikipedia 100M or 1B benchmarks for text (Zoph et al. 2018; Real et al. 2019). Furthermore, the authors also showed that the architectures found could be scaled or stacked to have more capacity and thus achieve better performance (Real et al. 2019).

NAS did produce architectures that are useful in production, especially neural networks that achieve high performance at low computational cost for inference (in terms of inference speed and also number of parameters). Three examples are reviewed in the next section, on LSTM node design, general modular networks, and refinement of existing designs, all based on evolutionary optimization. Evolutionary NAS was also applied to the transformer architecture, to produce Evolved Transformers (So, Le, and Liang 2019) which also perform better on benchmark tasks while requiring fewer resources.

It is actually remarkable that there are many different approaches to NAS, and they all work well. It seems that you can apply almost any optimization technique—evolution, RL, Bayesian optimization, gradient descent—and get improved results. Even just random search may perform well, for instance achieving results within less than half a percent of more sophisticated NAS methods, and close to state-of-the-art performance for both image

classification and language modeling benchmarks (Real et al. 2019; Li and Talwalkar 2020). This observation suggests that much of the performance is already baked into the hand-engineered building blocks of NAS, such as convolutional filters, self-attention layers, and RNN nodes. The research community has designed them by hand to achieve SOTA. NAS has proven useful as a way to fine-tuning them, but it has not yet produced innovations that could automate the discovery of such truly fundamental concepts.

That is probably why, despite these improved MobileNet, Transformer, and RNN node architectures, people still often use the traditional MobileNet, the classical Transformer, and the original LSTM in most networks in production. The performance gains have not yet been large enough and their implementations stable enough for the software and hardware vendors to converge on the improved variants. The NAS field continues to make progress though, including successes outlined in the next few sections, and discoveries that extend to other fields, which may lead to such convergence in the future.

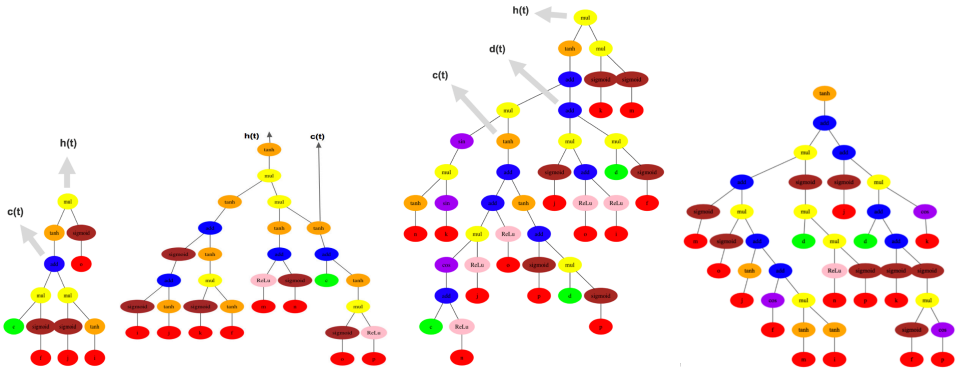
10.3 Example NAS Successes

This section reviews three NAS methods that resulted in SOTA performance at the time. The first one, the design of LSTM nodes, improved the original design that had stayed the same since the 1990s. It demonstrated that complexifying the design can add power even though such designs are difficult for humans to discover. The second, CoDeepNEAT, generalizes ideas from general neuroevolution to the level of network architectures. In principle, it could discover new architectural principles that work better than the existing human-designed ones. It has not so far—the challenge is to identify the proper building blocks and then take advantage of structure. The third, Amoebanet, utilizes structure, scaling, and regularization more explicitly by hand. It achieved SOTA on ImageNet in 2018, which was a remarkable achievement given that ImageNet was the main focus of the machine-learning community at that time. It may be possible to use an Amoeba-like approach in the future to incorporate new ideas and improve performance again. Note that even slight improvement is sometimes useful: For instance in finance, healthcare, and engineering design it translates to money, lives, and resources saved.

10.3.1 LSTM Designs

First, consider the design of better LSTM nodes. The original architecture (Figure 10.6*a*) had been developed in the 1990s (Hochreiter and Schmidhuber 1997), and despite many attempts to improve it by hand, it was deemed to be robust, general, and usually at least as good as the alternatives (Greff et al. 2017). In essence, an LSTM node is a neuron that can memorize a value in its internal memory cell indefinitely long. It contains circuitry for loading that value (the input gate), reading it out (the output gate), and erasing it (the forget gate). A sequence processing network includes many such nodes, and their internal parameters (weights, activation functions) can be modified through backpropagation. Through such learning, each node determines when and how it utilizes its memory cell best as part of the processing of sequences.

Even though this design is principled and makes sense, it turns out that it can be complexified significantly, leading to LSTM nodes that perform better. Its internal processing can be more complex, propagating through a nonlinear network with multiple paths. Its



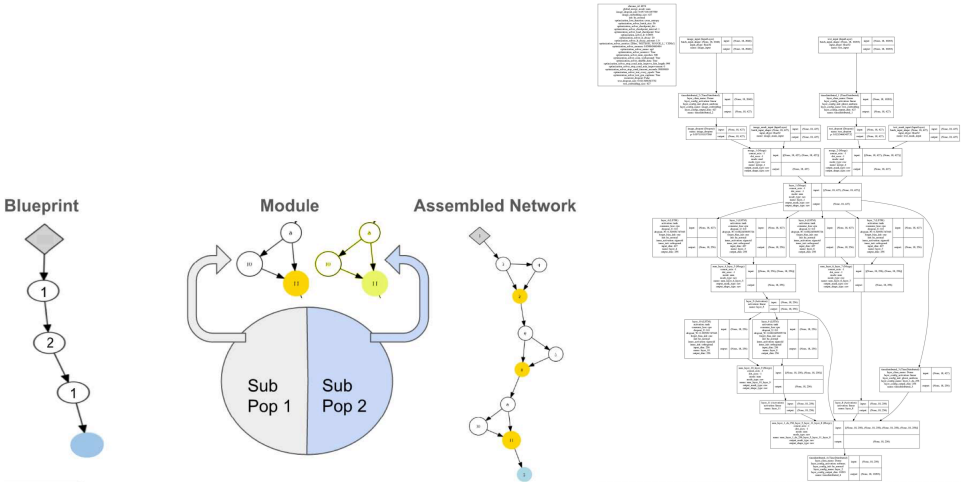
(*a*) Original LSTM (*b*) NASCell node (language modeling) (*c*) Evolved node (language modeling) (*d*) Evolved node (music modeling)

Figure 10.6: **NAS in LSTM node design.** At the lowest level, NAS can be used to design nodes in a recurrent neural network. In the node diagrams above, the $h(t)$ is the main output of the node, propagated to other nodes. The $c(t)$ and $d(t)$ are outputs of the native memory cell, propagated internally. The green input elements denote the native memory cell outputs from the previous time step (i.e. $c(t-1)$ or $d(t-1)$). The red input elements are formed after combining the node output from the previous time step (i.e. $h(t-1)$) and the new input from the current time step ($x(t)$. The other colors identify activation functions in computational cells: ReLU, sigmoid, tanh, sin, add, and multiply. In all solutions, the memory cell paths include relatively few nonlinearities. Unlike LSTM and NASCell, the evolved nodes reuse inputs and utilize extra memory cells in different parts of the node; they also discovered LSTM-like output gating. The evolved nodes for language and music modeling are different, suggesting that evolution captures and utilizes the inherent structure in these domains to perform better. In this manner, neuroevolution was able to improve upon a human design that had stayed the same for decades and was considered optimal among many variants. For an animation of this search process and an interactive demo, see https://neuroevolutionbook.com/neuroevolution-demos. (Figures from Rawal and Miikkulainen 2020)

memory state can be more complex, consisting of multiple memory cells. It can utilize a variety of activation functions in its internal nodes, and more general memory blocks. Such complexification is difficult for humans to develop, but NAS methods can do it.

The first such improvement was based on reinforcement learning (Zoph and Le 2017). A recurrent network was used to generate the node designs, trained through the REINFORCE algorithm (Ronald J. Williams 1992b) to maximize the expected accuracy on a validation set. The resulting NASCell was significantly more complex than the original LSTM design (Figure 10.6*b*). However, the exploration ability of such refinement search is somewhat limited and can be expanded through evolutionary methods.

In particular, genetic programming was used to search for trees representing the node structure, resulting in designs with multiple nonlinear paths and multiple memory cells (Figure 10.6*c*) Rawal and Miikkulainen 2020). In the language modeling domain (i.e. predicting the next word), this design was organized into two layers of 540 nodes each and evolved for 30 generations. Compared to networks of similar size, it improved 20 perplexity



(*a*) CoDeepNEAT approach (*b*) Image captioning network

Figure 10.7: **Discovering general neural architectures through coevolution of modules and blueprints.** The CoDeepNEAT approach (Miikkulainen, Liang, et al. 2023) aims at discovering modular architectures in an open-ended search space. (*a*) The blueprints represent the high-level organization of the network and modules fill in its details. The blueprint and module subpopulations are evolved simultaneously based on how well the entire assembled network performs in the task. This principle was originally developed for evolving entire networks (Moriarty and Miikkulainen 1997; Gomez and Miikkulainen 1997), but it applies in neural architecture search for deep learning as well. (*b*) The overall structure of a network evolved for the image captioning task; the rectangles represent layers, with hyperparameters specified inside each rectangle. One module consisting of two LSTM layers merged by a sum is repeated three times in the middle of the network. The approach allows discovery of a wide range of network structures. They may take advantage of principles different from those engineered by humans, such as multiple parallel paths brought together in the end in this network. For a demo of CoDeepNEAT in character recognition task, see https://neuroevolutionbook.com/neuroevolution-demos. (Figures from Miikkulainen, Liang, et al. 2023)

points over the original LSTM and 1.8 points over the NASCell, achieving the state-of-the-art performance of 62.2 at the time. Most interestingly, when the same approach was applied to the music modeling domain (i.e. predicting the next note), a different design emerged as the best (Figure 10.6*d*). This result suggests that different domains have different structure; such structure can be learned by NAS and architectures customized to take advantage of it.

These results opened the door to optimizing combinations of different kinds of memory nodes, like those used in the neural Turing machine (Section 12.3.4; Khadka, Chung, and Tumer 2019), and other recurrent network elements (Ororbia, ElSaid, and Desell 2019). As a result, the memory capacity of the model increased multifold—an improvement that likely would not have happened without such automated NAS methods.

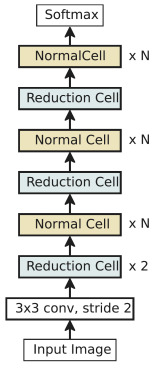
10.3.2 CoDeepNEAT

As a second example, consider the CoDeepNEAT method of discovering general network designs. CoDeepNEAT (Miikkulainen, Liang, et al. 2023; J. Liang et al. 2019) builds on several aspects of techniques developed earlier to evolve complete networks. In SANE, ESP, and CoSYNE, partial solutions such as neurons and connections were evolved in separate subpopulations that were then combined into full solutions, i.e. complete neural networks, with the global structure specified e.g. in terms of a network blueprint that was also evolved (Moriarty and Miikkulainen 1997; Gomez and Miikkulainen 1997; Gomez, Schmidhuber, and Miikkulainen 2008). Similarly, CoDeepNEAT co-evolves multiple populations of modules and a population of blueprints that specify which modules are used and how they are connected into a full network (Figure 10.7*a*). Modules are randomly selected from the specified module population to fill in locations in the blueprint. Each blueprint is instantiated in this way many times, evaluating how well the design performs with the current set of blueprints. Each module participates in instantiations of many blueprints (and inherits the fitness of the entire instantiation each time), thus evaluating how well the module works in general with other modules. The main idea of CoDeepNEAT is thus to take advantage of (and scale up with) modular structure, similarly to many deep learning designs such as the inception network and the residual network (He et al. 2016a; Szegedy et al. 2015).

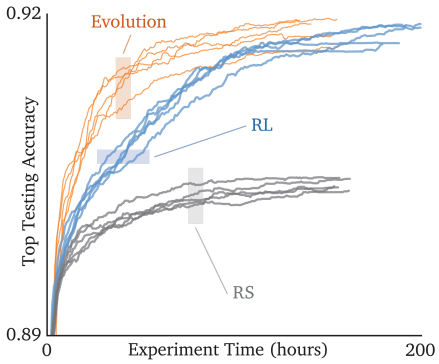
The modules and the blueprints are evolved using NEAT (Section 3.4), again originally designed to evolve complete networks and adapted in CoDeepNEAT to evolving network structure. NEAT starts with a population of simple structures connecting inputs straight to outputs, and gradually adding more modules in the middle, as well as parallel and recurrent pathways between them. It thus prefers simple solutions but complexifies the module and blueprint structures over time as necessary. It can in principle design rather complex and general network topologies. However, while NEAT can be used to create entire architectures directly, in CoDeepNEAT it is embedded into the general framework of the module and blueprint evolution; It is thus possible to scale up through repetition that would not arise from NEAT naturally.

The power of CoDeepNEAT was originally demonstrated in the task of image captioning, a domain where a competition had been run for several years on a known dataset (Miikkulainen, Liang, et al. 2023). The best human design at that point, the Show&Tell network (Vinyals et al. 2015), was used to define the search space; that is, CoDeepNEAT was set to find good architectures using the same elements as in the Show&Tell network. Remarkably, CoDeepNEAT was able to improve the performance further by 15%, thus demonstrating the power of neural architecture search over the best human solutions (Miikkulainen, Liang, et al. 2023). Similar CoDeepNEAT evolution from a generic starting point was later used to achieve a state-of-the-art in text classification (Wikidetox; J. Liang et al. 2019) and image classification (Chest X-rays;). Indeed, these successes demonstrated that with very little computational cost, neural architecture search can achieve performance that exceeds that of standard architectures, making it possible to quickly and effectively deploy deep learning to new domains.

Most importantly, the best networks utilized a principle different from human-designed networks: They included multiple parallel paths, possibly encoding different hypotheses



(*a*) AmoebaNet approach



(*a*) Comparison in ImageNet

Figure 10.8: **Evolutionary Discovery in the NASNet Search Space Compared to RL and Random Search.** (*a*) In contrast with CoDeepNEAT, the AmoebaNet method (Real et al. 2019) focuses evolution to a particular stacked architecture of inception-like normal and reduction modules (cells); these networks are then scaled to larger sizes algorithmically. AmoebaNet also promotes regularization by removing the oldest individuals in the population. (*b*) As a result, it discovers architectures that are more accurate than those discovered through random search and RL, reaching state-of-the-art accuracy in standard benchmarks like ImageNet. (Figures from Real et al. 2019)

brought together in the end (Figure 10.7*b*). In this manner, the large search space utilized by CoDeepNEAT may make it possible to discover new principles of good performance.

Such discovery is indeed the main power of CoDeepNEAT, and what it was initially designed to do. At the time, papers were coming out each outdoing each other proposing a different architecture. It seemed that the space of good architectures was large and ripe for discovery. Soon after, however, the transformers and diffusion architectures were developed and became dominant. While there is still plenty of opportunity to optimize variants of them using neuroevolution, a major question for the future is whether open-ended search methods such as CoDeepNEAT can be further developed to discover new principles that might follow them.

10.3.3 AmoebaNet

Even small improvements to performance are sometimes useful. If you are designing a network to predict financial data, half a percent can translate to millions. If it is to predict effects of treatments, it can save lives. Thus, NAS applied to the refinement of existing ideas can play an important role. Perhaps the best example of such work is the AmoebaNet system (Real et al. 2019). At its time, it improved the state-of-the-art in the ImageNet domain, which had been the focus of deep learning research for several years. Many architectures and ideas have been designed by human experts for it; AmoebaNet exceeded the performance of all of them, by utilizing evolutionary neural architecture search in a manner that mattered in practice.

There were three innovations that made this result possible. First, search was limited to a NASNet search space, i.e. networks with a fixed outer structure consisting of a stack

of inception-like modules (Figure 10.8*a*). There were two different module architectures, normal and reduction; they alternate in the stack, and are connected directly and through skip connections. The architecture of the modules is evolved, and consists of five levels of convolution and pooling operations. The idea is that NASNet represents a space of powerful image classifiers that can be searched efficiently. Second, a mechanism was devised that allowed scaling the architectures to much larger numbers of parameters, by scaling the size of the stack and the number of filters in the convolution operators. The idea is to discover good modules first and then increase performance by scaling up. Third, the evolutionary process was modified to favor younger genotypes, by removing those individuals that were evaluated the earliest from the population at each tournament selection. The idea is to allow evolution to explore more instead of focusing on a small number of genotypes early on. Each of these ideas is useful in general in evolutionary ML, not just as part of the AmoebaNet system.

Indeed, AmoebaNet's accuracy was the state of the art in the ImageNet benchmark at the time. Experiments also demonstrated that evolutionary search in NASNet was more powerful than reinforcement learning and random search in CIFAR-10, resulting in faster learning, more accurate final architectures, and ones with lower computational cost (Figure 10.8*b*). It also demonstrated the value of focusing the search space intelligently so that good solutions are in that space, yet it is not too large to find them.

Thus, LSTMs, CoDeepNEAT, and AmoebaNet demonstrated the potential of evolutionary NAS in discovering new principles as well as making practical optimizations to existing ones. A challenge for the future is to take them to transformers, diffusion networks, and beyond. In the meantime, however, such approaches are useful in two important areas: optimizing architectures for specific hardware constraints, and discovering architectures that can perform well with little data by utilizing other tasks and datasets. These opportunities will be discussed in the next section.

10.4 Multiobjective and Multitask NAS

In the NAS discussion so far, improved SOTA performance in the task has been the main and only objective. Indeed, as mentioned above, in certain domains the cost of putting together a large dataset and spending a lot of compute to achieve even small improvements can be worth it. Benchmarks are also a good motivation for research: it is fun to compete with other researchers in achieving better performance in them, and thus gain prestige and recognition.

However, when new technologies are taken to the real world, a number of new, practical challenges emerge. In particular, expertise to build good models may not be available; the models may run in the edge, with limited compute and other hardware restrictions; the data may not be sufficient in quality and quantity to train good models. Neural architecture search, and metalearning in general, can be used to cope with each of these challenges.

First, designing good models for new learning tasks still relies on scarce expertise. The available simulators such as TensorFlow, PyTorch, and Keras provide standard models as starting points, and in many cases, they work well. However, the number of datasets and problems where they can potentially be used is also very large, and applications could often benefit even from small optimizations. Searching for appropriate architectures is one such optimization; other metalearning dimensions such as activation functions, loss functions,