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,

and data augmentation are useful as well, as is optimization of general learning parameters (these approaches will be reviewed in Chapter 11). The term "AutoML" has been coined to refer to such processes in general: The user provides a dataset and a starting point for learning, and the learning system configures itself automatically to achieve better results (J. Liang et al. 2019; He, Zhao, and Chu 2021). The goal is not necessarily to achieve state-of-the-art in any particular domain but to reduce the human time and expertise needed to build successful applications. In this manner, deep learning can have a larger impact in the real world.

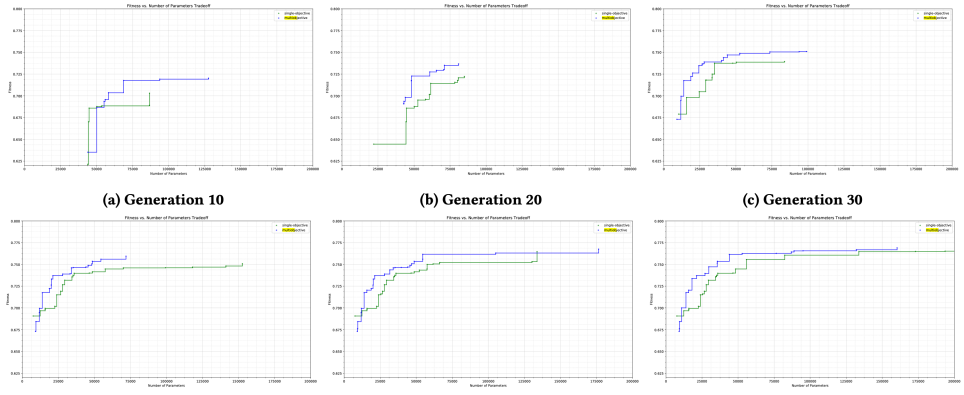
Second, many such applications cannot be deployed to run on data centers with dedicated top-of-the-line hardware, but need to run on commodity compute, or even on highly constrained compute in the edge: vehicles, drones, probes in extreme environments, as well as watches, appliances, clothing, and so on. Only a fraction of the model sizes used in research may be available in such applications, and there may be limitations on memory structure, communication, latency, etc. NAS can play a significant role in optimizing the models to perform as well as possible under such conditions.

In some cases, the constraints must be met entirely, or the solutions are unviable. As usual in evolutionary computation, such constraints can be implemented as penalty functions, thus allowing evolution to explore more broadly but eventually converge to solutions that satisfy the constraints. It may also be possible to modify the solutions algorithmically to make them comply; evolution will then find a way to optimize the solutions under such postprocessing.

In other cases, the constraints incur a cost that needs to be minimized. NAS for such applications is multiobjective, aiming at identifying good tradeoffs between performance and cost outcomes. For instance, CoDeepNEAT can be extended with multiobjective optimization to form Pareto fronts of accuracy and network size (J. Liang et al. 2019). In the domain of classifying X-ray images, a variety of tradeoffs were discovered, but there was also a sweet spot in the front: an architecture that was 1/12th of the size of the best-performing network while only giving up 0.38% in accuracy (Figure 10.9). In a similar manner, other objectives could be included, such as training time, amount of training data needed, or energy consumption. Multiobjective NAS can thus make many more deep learning applications feasible in the real world.

In the most extreme case along these lines, NAS can be used to optimize designs for neuromorphic hardware. In order to minimize energy consumption, many such architectures are based on spiking neurons, are small in size, and limited in connectivity. Standard deep learning architectures are not well suited for them, and there are many opportunities to discover creative, new designs. A most interesting and potentially fundamental way is to co-evolve the hardware design with the neural network design simultaneously. In this manner, it may be possible to discover powerful solutions that are highly specialized and customized to individual use cases. These opportunities will be discussed in more detail in Section 11.4.

The third real-world challenge is insufficient data. Indeed, data is now collected everywhere from small businesses, doctor's offices, and engineering firms to large-scale transportation, weather, business, and education systems. Unfortunately, such data is often siloed and not aggregated, and often also proprietary and intentionally kept in-house. Even though the data could in principle be used to solve many prediction and optimization problems,



(a) Generation 10 (b) Generation 20 (c) Generation 30

(d) Generation 40 (e) Generation 50 (f) Generation 60

Figure 10.9: **Simultaneous optimization of network size and performance.** The number of parameters in the network is in the x-axis and the accuracy in classifying X-ray images to 14 different diseases is in the y-axis. The curves show the Pareto fronts obtained in a single-objective evolution (of accuracy; green) and multiobjective evolution (of accuracy and number of parameters; blue). Both populations include a range of tradeoffs but the multiobjective evolution discovers consistently better ones, including one at the elbow that's 1/12th of the size and 0.38% less accurate than the top accuracy. In this manner, NAS can discover architectures that not only perform well but also adhere to cost constraints, making more applications possible in the real world. For an animation of this process, see https://neuroevolutionbook.com/neuroevolution-demos. (Figures from J. Liang et al. 2019).

there is not enough of it to take advantage of modern machine learning. Such models would simply learn to memorize and overfit and not perform well with future data.

Interestingly, in many such domains, it may be possible to build better models by utilizing other datasets (Caruana 1997; Meyerson and Miikkulainen 2019). When a model is trained to perform multiple tasks simultaneously, represented by different datasets, it learns to encode each task based on synergies and commonalities between them. Such common knowledge in turn establishes biases that make it possible to generalize better, even when the training data within each task alone would be insufficient.

An important role for NAS is to discover architectures that take the best advantage of such synergies between tasks. Many designs are possible (Figure 10.10: If the tasks are well-aligned, a single processing path with a different head for each task may be the best way to integrate them. Alternatively, many parallel paths can be constructed, and different tasks will utilize them differently. If the tasks are sufficiently different, a complex topology with different tasks performed at different levels based on customized topologies may be needed. It is difficult to tell ahead of time which architectures work well; evolutionary NAS is a good way to optimize them.

To motivate an approach, first consider training a simple network to support multiple tasks. The network consists of a few tightly connected layers and has a number of decoder layers on top, one for each task. The tasks can be real, i.e. be based on different datasets, or they can be pseudotasks, constructed artificially by assigning a different set of labels to the

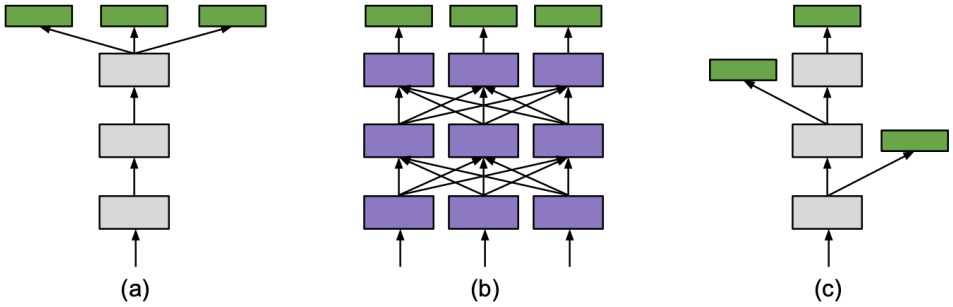


Figure 10.10: **Alternative approaches to multitask learning**. When multiple tasks are learned simultaneously, the network may discover and utilize general principles underlying them, and perform better than when trained with each task alone. (*a*) If the tasks are similar, a single column with a different head for each task may work well. (*b*) A more flexible architecture may consist of a number of modules at each level, and each task uses them differently. (*c*) In the most general case, a customized topology may be used to support a number of different tasks. It is difficult to decide which architecture works well; evolutionary NAS can be used to find optimal ways to do it. Figure from Meyerson and Miikkulainen 2018a

same training examples (Meyerson and Miikkulainen 2018b). Gradient descent can then be used to train this architecture.

In the next step, the architecture consists of multiple levels of several such modules. All modules are included at all levels, but the network learns to utilize them differently at different levels for different tasks. Through gradient descent, they learn functional primitives that are useful in several tasks (Meyerson and Miikkulainen 2018a).

This is where neuroevolution comes in. It is possible to use evolution to discover an optimal topology of these modules for each task. That is, each task has a different organization of modules into a network topology, but the modules all come from the same set, trained together via gradient descent in all tasks. In this manner, the modules still learn to encode functional primitives; evolution figures out how to use these primitives optimally in each task.

The final step, then, is to use CoDeepNEAT to evolve the structure of the modules themselves (in the CMTR method; Liang, Meyerson, and Miikkulainen 2018). In this manner, (1) high-level evolution customizes the topology for each task, (2) low-level evolution optimizes the structure of the modules so that they can extract common knowledge most effectively, and (3) gradient descent extracts the common knowledge across tasks and encodes it into the modules.

This approach was demonstrated e.g. in the Omniglot domain, i.e. in recognizing handwritten characters in multiple different alphabets (Liang, Meyerson, and Miikkulainen 2018; Lake, Salakhutdinov, and Tenenbaum 2015). While the alphabets are quite different, they are still related in that each consists of shapes and combinations of lines in a limited area. While there are only 20 examples of each character, there are 50 different alphabets, and therefore multitask learning is an effective way to combine knowledge from all alphabets to learn each one well. Moreover, evolutionary optimization makes it possible

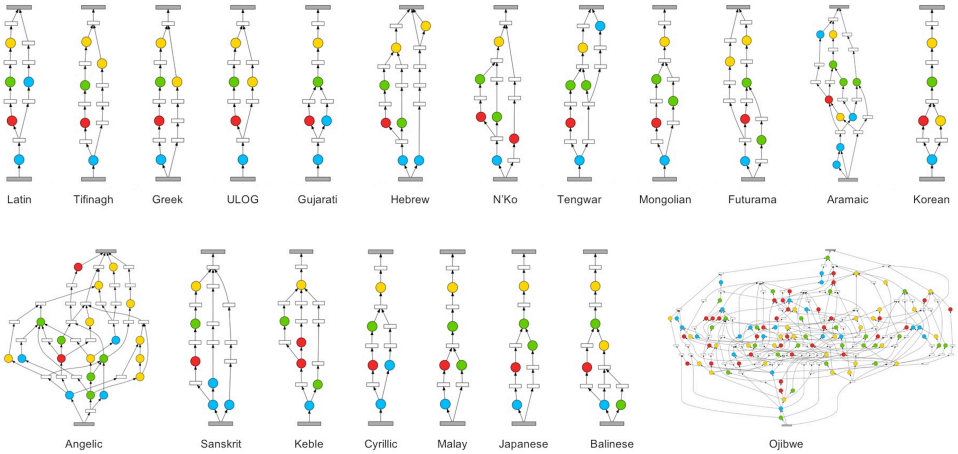


Figure 10.11: **Network topologies discovered for different handwritten alphabets.** Each network is trained to recognize handwritten characters of one alphabet. However, each topology is constructed from the same set of neural network modules (indicated by color) and thus such training results in modules that encode the underlying functional primitives of many tasks. More complex alphabets receive more complex topologies, and similar alphabets similar topologies. The resulting topologies are consistent across several runs of evolution and training, suggesting that they indeed capture underlying principles. Even though the training data is limited for each task, the primitives make it possible to learn each task well—better than if the networks were trained from scratch with their own data only. Thus, NAS can be used to tie together learning of multiple tasks so that learning with otherwise insufficient data is possible, making it possible to extend machine learning to more real-world tasks. For an animation of this evolutionary process, an interactive character recognition demo, and other demos on multitask evolution, see https://neuroevolutionbook.com/neuroevolution-demos.

to learn and utilize common knowledge well, as well as to specialize: The CMTR approach improved state-of-the-art by 30% in this domain.

It is interesting to see the solutions CMTR created (Figure 10.11). In general, the more complex the alphabet, the more complex the topology. One example is Angelic, a synthetic alphabet designed in the 1500s to communicate with angels. It is more decorative and unique than most, and the network constructed for it is complex. Also, alphabets that look similar have similar networks. For instance, Hebrew and N'ko both have dominant horizontal lines, and their network topologies are similar; Latin and Cyrillic are similar as well. Interestingly, when evolution is run multiple times, consistent topologies emerge for the same language each time, suggesting that they indeed capture essential representations for each task. It would be difficult to come up with such representations by hand, but evolutionary NAS does it reliably.

Multitask learning has been demonstrated to work well even when the tasks are very different. For instance, language learning, vision, and genomic structure prediction can all be mutually informative, even though they represent very different domains in the world. A method for aligning the parameters across such differences is needed, but with such a

method, it seems possible to support many disparate domains with many others (Meyerson and Miikkulainen 2019).

Apparently, the world is based on a set of fundamental principles and structures that repeat across domains, perhaps as low-dimensional manifolds embedded in high-dimensional spaces. Thus, learning to understand part of the world helps in understanding other parts. It may be possible to take advantage of this observation to evolve supernetworks, consisting of modules that can be reused in different configurations, or paths through the network, to learn new tasks (Fernando et al. 2017). More generally, it may be possible to construct a central facility that learns and represents these regularities as variable embeddings and different tasks are established by learning specialized encoders and decoders of this knowledge (Meyerson and Miikkulainen 2021). This approach can be instantiated through multitask learning and evolution. It may also be possible to utilize LLMs as the central facility, and then evolution to discover the customized encoders and decoders. While such architectures do not yet exist, the approaches reviewed in this section are a possible starting point for constructing them. This is one approach that might, in the long term, in constructing agents with general intelligence.

10.5 Making NAS Practical

Even in settings where NAS can make useful discoveries, the approaches are still limited by available computation. Efficient implementations can make a big difference, leading to better solutions. The approaches involve evaluating a large number of neural network designs, which is very expensive. Training a deep learning network can take several days, and a search for good designs may need to evaluate millions of candidates. If the search simply runs as an outer loop, it will be limited to a few hundred or thousand candidates.

Several principled efficiency optimizations are possible. One important one is to utilize surrogate models. Instead of modeling how the world will respond to a solution, as was done in Section 6.2.2, they model the solutions directly, i.e. how well each solution is going to perform in the task. This approach is useful in metalearning in general: In its most general form, it powers bilevel evolution, i.e. an approach where an outer-loop evolution optimizes the parameters of an inner loop evolutionary process (Section 11.2). It can be instantiated to speed up search in all aspects of metalearning, including that of activation functions (Section 11.3.2).

Surrogate models are usually trained with a sample of solutions. For instance in NAS, a set of different architectures is created and evaluated ahead of time, the model trained to map architecture descriptions to performance, and then used predict the performance of new solutions. Several such benchmark collections have already been created and they can serve as a catalyst for studying NAS methods in general (Zela et al. 2022; Dong and Yang 2020; Ying et al. 2019).

Another way of making NAS practical is to limit the search space. The Amoeba method (Section 10.3.2) already took advantage of it by optimizing the variations of a repetitive structure. In a more extreme approach, a supernet is first created, i.e. large network that consists of the entire search space, including all possible layers, their variations, and connections between them. The supernet is then trained in the task (at least partially). It then serves as a starting point for creating candidates during search, providing the search space

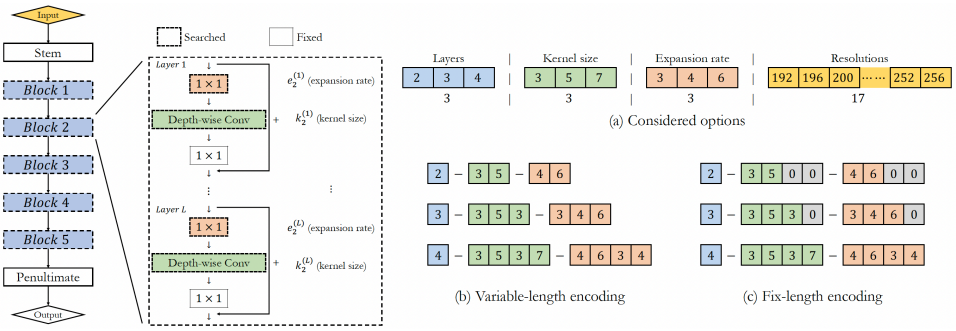


Figure 10.12: **The MSuNAS approach for evolving convolutional networks.** The idea is to make search practical by limiting the search space and by guiding the search. The search space consists of five computational blocks, and is parameterized through number of layers, kernel size, channels (that expand through the layers), and input resolution. (*a*) The paremeters are selected from a prespecified set and can be coded either as variable (*b*) or fixed (*c*) length individuals. A supernet is created with the largest values and subsumes the entire search space. Good tradeoffs between performance and other objectives are then found in this space using NSGA-II multiobjective search method. A surrogate model, trained with a sample of architectures in this space, is used to guide the search, and the trained supernet to initialize the weights of the candidates. The approach can find architectures that perform better or similar to standard architectures, and are smaller, with significantly less training. (Figure from Lu et al. 2020).

and initial evaluations. This approch makes sense if the goal is not just to find the best-performing network (for which the supernet itself might be the best choice), but at the same time, achieve other objectives like minimize the size of the solutions.

Several of these ideas were implemented in the MSuNAS approach, where the NSGA-II multiobjective optimization method was adapted to NAS of convolutional image-processing networks (Figure 10.12; Lu et al. 2020). The search space was restricted to networks with five computational blocks with four design parameters, i.e. the number of layers, the number of channels, the kernel size, and the input resolution, each with a predetermined range. A supernet was created by setting each of these parameters at their maximum values; thus all other candidates in the search space were enclosed in it. A surrogate model was trained with 2000 randomly sampled networks in this space. Each network was trained for 150 epochs on CIFAR-10, CIFAR-100, and ImageNet, and evaluated with 5000 unseen images. The supernet was trained in this task as well, and its weights used to initialize the candidates during search.

The approach found solutions that represented useful tradeoffs in this domain. The most accurate architectures performed as well or better than standard architectures, and many of them were much smaller as well. The surrogate modeling approach resulted in several times to orders of magnitude faster learning. These results suggest that NAS can be a practical and useful technique in searching variations in a limited search space.

Sometimes such methods are called one-shot methods, because the supernet is trained to represent the entire search space. The more general approach consists of black-box,

or zeroth-order, methods, where the search space is open-ended (such as CoDeepNEAT described in Section 10.3.2). Such methods have more potential for discovery, but it is more difficult to make them efficient, and therefore take advantage of them.

Intermediate approaches may provide a good tradeoff. For instance, it is possible to limit NAS to traditional convolutional networks only, i.e. those with a number of convolutional and pooling layers followed by a number of fully connected layers (as opposed to very deep networks with many skip connections such as ResNet or DenseNet). Such a limited search space allows customizing many aspects of the NAS process, making it efficient.

In one such approach, EvoCNN (Sun et al. 2020), it was possible to design a variable-length representation for the architecture that allows networks of variable sizes to be represented systematically and compactly. The population could then be initialized as a random sample of such architectures, instead of minimal networks, providing for a more comprehensive search process. On the other hand, the number of parameters was used as a fitness component during evolution, favoring smaller networks, thus making sure that the complexity that was there actually mattered. Weight initialization was also included as part of the representation as mean and standard deviation values for sets of connections. As is well-known in deep learning (and discussed in more detail below), good initialization makes it more likely that the architecture performs as well as it can, resulting in more consistent and fair evaluations. Genetic operators were then designed to operate efficiently on such architectures. With these customizations, EvoCNN performed better than other hand-designed traditional CNN architectures. Also interestingly, the evolved initialization performed better than standard initialization methods, such as Xavier (Glorot and Bengio 2010).

Part of why fully general (zeroth-order) methods are challenging to design is because it is difficult to implement even basic evolutionary search, i.e. crossover. The architectures are usually represented as graphs, and they suffer from the permutation problem (or competing conventions problem): the same functional design be coded in several different ways simply by changing the order of elements in it. The permutation problem makes crossover ineffective, which is why most black-box methods rely only in mutation.

As a matter of fact, the same issue exists in many other areas of evolutionary computation, to the extent that the entire validity and usefulness of crossover is sometimes called into question (Qiu and Miikkulainen 2023). Yet, biology utilizes crossover very effectively, creating solutions that are viable and creative (Section 9.1.1). This observation suggests that perhaps we do not understand crossover very well, and our implementations of it are lacking something.

Interestingly, NAS can be used as a domain to gain insight into the general problem of what makes crossover useful (Qiu and Miikkulainen 2023). Two architecture representations can be compared through graph edit distance (GED), measuring how many modifications are necessary to transform one to the other. This metric can then be used to construct a crossover operator that results in individuals that lie along the shortest edit path (SEP) between them. It turns out that theoretically the expected improvement from the SEP crossover is greater than the improvement from local search (i.e. mutation), from standard crossover, and from reinforcement learning. These theoretical conclusions can be demonstrated numerically, as well as in practical evaluation in various NAS benchmarks: They converge to optimal architectures faster than other methods, even with noisy evaluations.

Thus, crossover can be a useful tool in NAS if implemented in the right way. More generally, if evolutionary computation is not using crossover, it is probably leaving money on the table.

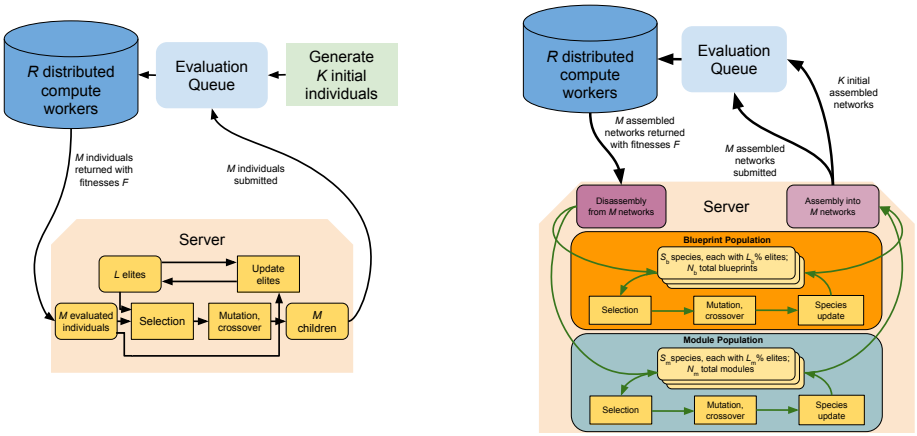
There are other useful tools that were originally developed with NAS in mind, but are useful in neuroevolution, and in evolutionary computation and in neural networks, in general. An important one is to initialize the networks in a proper way before training (Bingham and Miikkulainen 2023a). In deep learning, a fundamental challenge is that the signals (activation and gradients) may vanish, or explode. If the network weights are initialized so that the activation stays within reasonable bounds, training is more likely to be successful. In NAS, this means that the evaluation of the candidate is more reliable, making search more effective. The initialization can be done in various ways and customized to specific activation functions, topologies, layers, and even data. However, there is a general principle that works well in most cases: Setting the weights of each layer so that the outputs have zero mean and unit variance.

In a method called AutoInit, such weight initialization was derived for the most common layer types (Bingham and Miikkulainen 2023a). Experimentally, AutoInit resulted in faster and more reliable convergenge for convolutional, residual, and transformer architectures, various hyperparameter settings, model depths, data modalities, and input sizes. It was also shown particularly useful in metalearning of activation functions, and in NAS. When implemented in CoDeepNEAT, it adapted to each candidates unique topology and hyperparameters, improving its performance in several benchmark tasks. As expected, much of this improvement was due to reduced variance in evaluations. However, AutoInit also allowed utilizing a broader set of hyperparameter values and topologies. Some such solutions are difficult to train properly, and only perform well with proper initialization. Thus, intelligent initialization makes it possible for NAS to find more creative solutions as well.

Ultimately, NAS methods need to run on parallel hardware, and utilize such computation well. Like all evolutionary algorithms, NAS is well suited for such hardware because candidate evaluations can be performed at different compute nodes. However, evaluation times can sometimes be very long and vary significantly. It is therefore important that such evaluations are asynchronous: The nodes should not sit idle waiting for other candidates in a generation to finish their evaluations, but should take on other evaluations immediately (Liang, Shahrzad, and Miikkulainen 2023).

Asynchronous evaluation, therefore, is based on an evaluation queue rather than generations (Figure 10.13). Individuals are created and evaluated and the elite set updated continuously. While several such implementations exist already (including rtNEAT discussed in Section 8.1), the approach is more complex with more sophisticated NAS methods that take advantage of structure. For instance with CoDeepNEAT, individuals exist at the level of modules and blueprints, and both populations are speciated into subpopulations with their own elits. Thus, there are several evolutionary processes going on at the same time. When an assembled network is evaluated, the resulting fitnesses are incorporated into these processes asynchronously.

Note that although there are no generations, the evolutionary processes still need to progress in batches. That is, M individuals need to be evaluated and their fitnesses propagated to the current populations before another M can be generated—even though the individuals may have different ancestries and in a sense belong to different generations. As



(*a*) Evolving individual encodings (*b*) Coevolving hierarchical encodings

Figure 10.13: **Asynchronous evaluation of individual and coevolutionary encodings.** One challenge in parallelizing the evaluation of neuroevolution candidates is that the evaluation times may vary. Therefore, instead of evaluating an entire generation of candidates synchronously before generating new ones, candidates are placed in a queue and evaluated as soon as compute nodes become available. In this manner, compute nodes are never idle and evaluation can be speeded up significantly. (*a*) With encodings that represent the entire solution, the population and elites are maintained as usual, and evolution progresses in batches of M individuals. (*b*) With coevolutionary encodings such as CoDeepNEAT, the individuals are created and fitnesses distributed among participating blueprint and module populations. The process favors individuals with short evaluation times, which means that M needs to be larger when those times vary a lot. However, the speedup is also larger then, e.g. 14-fold for CoDeepNEAT. The bias towards networks that evaluate fast is also beneficial in NAS, resulting in more desirable solutions as a surprising side benefit (Figures from Liang, Shahrzad, and Miikkulainen 2023)

usual in evolution, the batch size M needs to be optimized for each problem, balancing the time used for evaluation and for search, i.e. how much evaluation noise can be tolerated. However, with variable evaluation times, batch evaluations establish a search bias: Those candidates that evaluate faster are more likely to be included in the batch, and thus more likely to reproduce. Thus, in domains where the evaluation times are relatively uniform, M can be small, and search proceed faster. However, if the times vary significantly, M needs to be larger so that evolution is based on more diverse candidates.

In NAS, such a bias is fortunately not a problem. The speedup from asynchrony increases more with variable evaluation times than the handicap from diversity. For instance in designing sorting networks, whire the times are relatively similar, asynchronous search finds solutions twice as fast than synchronous search. In CoDeepNEAT, where the times vary a lot, the speedup is 14-fold. Moreover, a bias towards faster networks is desirable in any case. Even if it is not an explicity secondary objective, smaller networks that evaluate faster are preferred over complex networks. In this sense, asynchronous evaluation provides an advantage not only in speed, but quality of solutions as well.

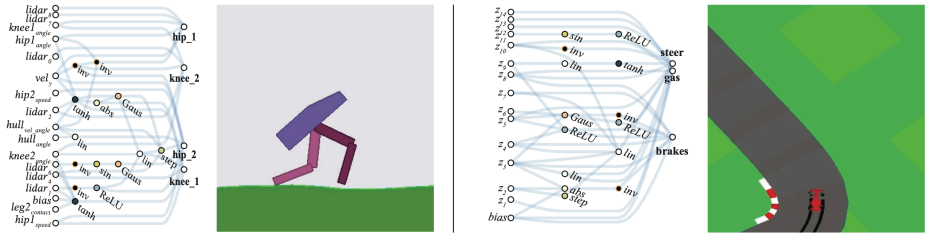
10.6 Beyond Neural Architecture Search

While NAS is still work in progress, already many interesting and useful ideas stemmed from the field — ideas that have impacted other subfields of AI. As was discussed in Section 10.2, one of the main limiting factors of NAS is the two-stage optimization process: One must search for the architecture in the outer loop, and spend a lot of computation in the inner loop to train each model. However, it turns out that the inner loop may not be as crucial in identifying good architectures as initially thought. Given that NAS mostly focuses on optimizing architectures with known, powerful building blocks, it may be possible to predict their performance without training them. A surrogate model can be trained based on a benchmark dataset of architectures and their performance for this task. Or, a hypernetwork can be used to predict the weights, making it possible to evaluate and rank candidates without having to train them (Brock et al. 2017).

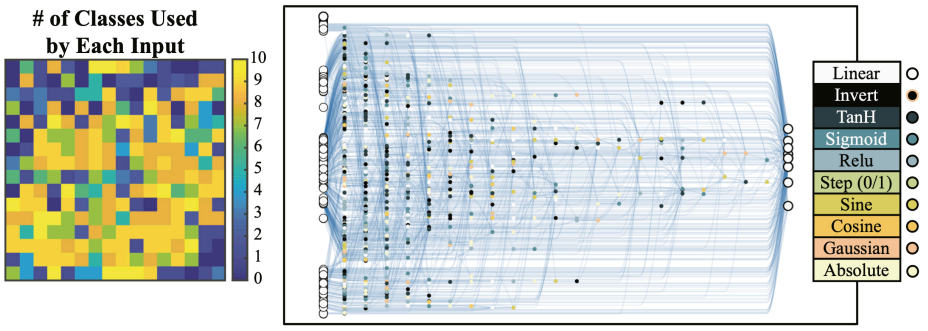
In the extreme, it turns out that even randomly initiated CNNs (Ulyanov, Vedaldi, and Lempitsky 2018) and LSTMs (Schmidhuber et al. 2007) have useful properties without any training. This leads to an important question: How important are the weight parameters of a neural network compared to its architecture? An approach called Weight Agnostic Neural Networks (WANNs; Gaier and Ha 2019) evaluated the extent to which neural network architectures alone, without learning any weight parameters, can encode solutions for a given task. The basic idea was to apply a simple topology search algorithm, NEAT, but explicitly make the weights random. To evaluate these networks, the connections were instantiated with a single shared weight parameter sampled from a uniform random distribution, and the expected performance measured over multiple such instantiations. It turned out that WANNs could perform several reinforcement learning tasks, and achieved much higher than chance accuracy on supervised tasks such as the MNIST classification (Figure 10.14). This result suggests that NAS alone may be sufficient to solve some problems without any gradient descent. Indeed, in many biological species the young are already proficient in many survival tasks without any learning; NAS with random weights can be seen as an approximation of this process.

A most compelling direction in NAS is to develep methods that discover the building blocks as well. They can be seen as components of neural network architectures that have great inductive bias at a variety of tasks. This approach would well with how biological evolution works, in that individuals are not born with simply a blank-slate neural network to be trained using gradient descent, but one that already implements a wide variety of useful innate behaviors that also impact their development. To quote Tony Zador, a computational neuroscientist (Zador 2019): *"The first lesson from neuroscience is that much of animal behavior is innate, and does not arise from learning. Animal brains are not the blank slates, equipped with a general purpose learning algorithm ready to learn anything, as envisioned by some AI researchers; there is strong selection pressure for animals to restrict their learning to just what is needed for their survival."*

Ideas have also emerged on how to move back from designing large deep learning architectures to optimizing such architectures entirely with evolution, including their weights. For instance, indirect encoding such as HyperNEAT can be used to optimize a very large number of weights by sampling the substrate more densely. In a more direct deep neuroevolution approach, deep network weights are represented compactly as a list of random



(*a*) Bipedal walking (*b*) Race-car driving



(*c*) Recognizing handwritten digits in MNIST

Figure 10.14: **Solving problems with NAS alone without gradient descent.** In the WANN approach, network architectures are evolved with a shared random value for weights. Surprisingly, without any gradient descent, they can solve reinforcement learning tasks such as bipedal walking and driving, and perform competently (at 94%) in MNIST handwritten digit classification. The diagram on the left side of (*c*) is part of an interactive demo that shows which parts of the input and network are used to classify different digits. WANN networks can be seen as a model of precocial performance in many animal species, where newborn individuals already perform well in a number of tasks necessary for survival without any experience or learning. For interactive demos, see https://neuroevolutionbook.com /neuroevolution-demos. (Figures from Gaier and Ha 2019)

number seeds: One for the initialization of the network and the rest for the random mutations that construct the network (Such et al. 2017). Another approach is based on ant-colony optimization: The ants traverse the architecture space from input to output, and the network is constructed based on their paths. Architectures of any size can be constructed in this manner, and the paths can include a weight dimension as well (ElSaid et al. 2023).

Many other promising ideas have emerged from the NAS field. Rather than searching for architecture, researchers have applied similar methods to search for better loss functions, activation functions, learning methods, and data augmentation methods. These optimizations are highly relevant even when network architectures have largely converged on a few best designs, such as transformers. Such approaches will be discussed in more detail on Chapter 11 on metalearning.

10.7 Chapter Review Questions

1. **NAS Approaches:** What are the primary methods used in Neural Architecture Search (NAS) to automate the design of neural network architectures? Why is evolutionary optimization particularly well-suited for this task?
2. **Backprop NEAT:** How does Backprop NEAT combine NEAT topology search with backpropagation? What role do activation function diversity and fitness regularization play in improving the evolved networks?
3. **Feature Discovery:** In the context of Backprop NEAT, how does the algorithm discover features that are typically engineered manually, such as those required for classifying concentric circles or XOR data?
4. **CoDeepNEAT:** How does the CoDeepNEAT approach leverage modular evolution to discover neural architectures? What advantages does its blueprint-module coevolution provide compared to evolving full architectures directly?
5. **AmoebaNet Contributions:** What innovations in AmoebaNet's evolutionary process enabled it to achieve state-of-the-art performance in ImageNet? How did these innovations improve the efficiency and accuracy of the NAS process?
6. **Multiobjective Optimization:** How does multiobjective NAS differ from single-objective NAS? What advantages does it offer when deploying neural networks in resource-constrained environments?
7. **Pareto Fronts:** Explain the concept of Pareto fronts in the context of NAS. How are they used to optimize trade-offs between objectives such as model accuracy and size?
8. **Multitask Learning:** What are the benefits of using NAS to discover architectures for multitask learning? How do alternative designs (e.g., single-column vs. complex topologies) address differences between tasks?
9. **Module and Topology Co-Evolution:** In multitask NAS, how does the co-evolution of module structures and task-specific topologies (e.g., in CMTR) enhance learning across tasks with limited data?
10. **NAS Efficiency:** What strategies, such as surrogate modeling and supernets, have been developed to make NAS computationally practical? How do they maintain effectiveness while reducing search costs?