11 Optimization of Neural Network Designs

11.1 Designing Complex Systems

Many areas of technical design are too complex for humans to optimize, and instead, automated methods must be used. VLSI design has long relied on machine optimization, but other areas of engineering are starting to rely on them as well. The systems have become larger, with many interacting elements, and several simultaneous performance goals. The sheer dimensionality and search space is too large to handle without automated search.

Evolutionary optimization is particularly well suited to such scaling. In some cases like designing circuitry for a 70-bit multiplexer, it was possible to find solutions in a space with $2^{2^{70}}$ potential solutions. While it is hard to imagine a space that large, consider that if that number of potential solutions was printed on paper with a 10pt font, it would take light 95 years to travel from the beginning to the end of the number (Miikkulainen 2021). In others like designing an optimal schedule for metal casting, there are variables for each type of object in each melting heat, and there may be tens of thousands of heats, resulting in billion variables (Deb and Myburgh 2017). Such scaling is possible because the population can discover partial solutions that can then be used as stepping stones to construct more complete ones, thus requiring exploration of only a fraction of the space and combinations of dimensions.

On the other hand, sometimes the scale is not the main problem, but complexity is: Problems can have nonlinear interactions and even be deceptive so that good solutions are overlooked. It is not just that search needs to be automated, but it should be intelligent enough to handle deception, such as evolutionary search. For instance, the original nose-cone of the Shinkansen bullet train was long and sleek, with great aerodynamics, but it created a bang when going into a tunnel. In the next version, the engineers wanted to eliminate the bang, but it was difficult to do by hand. However, they were eventually able to do so by harnessing evolutionary optimization: a cone with deep grooves on both sides (Ishida Lab 2018). It was unconventional and unlikely to be discovered by human engineers, but it got the job done. Similarly, evolution discovered that it may be advantageous to keep the lights on 24hrs in computer-controlled greenhouses: Basil doesn't need to sleep (Miikkulainen 2021). Further, webpage designs were found that violated well-known design principles with garish colors and active language, yet they were more effective in engaging users: What the human designers referred to as an "ugly widget generator" actually beat their design by 45% (Miikkulainen et al. 2020).

Similar stories abound in all areas of engineering from drug design and medical treatments to programming and autonomous control (see e.g. Lehman et al. 2020, for examples). As a matter of fact, the annual Human-Competitive Results competition at the GECCO Conference has showcased hundreds of such approaches since 2004 (Goodman 2023).

This insight applies to neuroevolution as well. While so far in this book, evolution has been used to optimize the network itself, i.e. its topology and weights, any aspect of the design can be evolved. Opportunities include the overall architecture, activation functions, loss functions, data augmentation, learning mechanisms, and even the neuroevolution optimizer itself. As a result, the networks can perform more accurately, generalize better, and/or use fewer resources than those designed by hand. Collectively, these approaches are called metalearning, which is the topic of this chapter.

11.2 Bilevel Neuroevolution

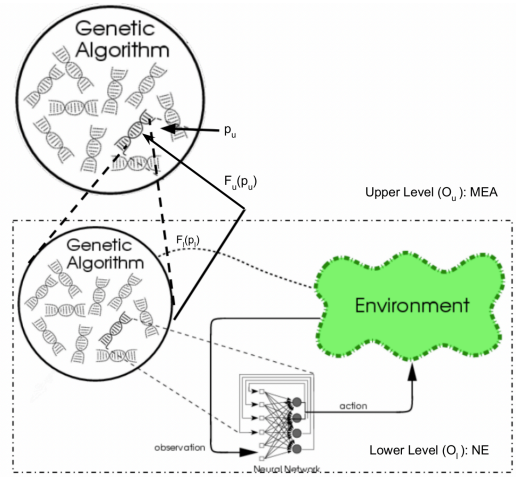
Several examples of neuroevolution discovering complex and robust behavior were reviewed in Chapter 6. Indeed, many such domains include a large number of variables that interact nonlinearly, making it difficult to design control algorithms using traditional methods. While neuroevolution can often be used effectively to construct robust controllers, it is still crucial to get the parameter settings right. Most often, the experiments require a painstaking search in the space of learning parameters, such as mutation and crossover rates and extent, population size, elite percentage, number of stochastic evaluations, etc. There are many such parameters and they interact nonlinearly, making the usual grid search of possible combinations ineffective.

An elegant and compelling solution is to use bilevel evolution to optimize the parameters (Liang and Miikkulainen 2015). That is, the optimization process is defined in terms of two nested problems (Figure 11.1*a*):

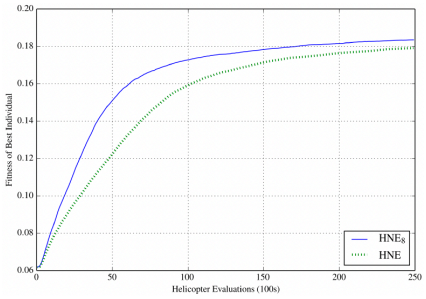
$$\underset{p_u}{\text{maximize}}\, F_u(p_u) = E[F_l(p_l)|(p_u)] \tag{11.32}$$

$$\text{subject to } p_l = O_l(p_u), \tag{11.33}$$

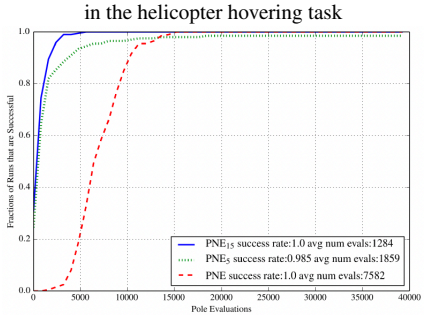
where $E[F_l(p_l)|p_u]$ is the expected performance of the neural network with parameters (i.e. weights) p_l, obtained by the lower-level optimization algorithm O_l (i.e. neuroevolution) with parameters p_u, which are in turn maximized by a separate upper-level optimization algorithm O_u.



(*a*) Bilevel neuroevolution



(*b*) Improvement over human fine tuning in the helicopter hovering task



(*c*) Improvement with more parameters in the double pole balancing task

Figure 11.1: **Enhancing neuroevolution with bilevel optimization**. Neuroevolution performance depends crucial on a proper setting of its hyperparameters. They can be evolved as part of the optimization process, resulting in bilevel neuroevolution. (*a*) More specifically, neural networks with parameters (weights) p_l are evolved using a low-level neuroevolution algorithm O_l with parameters p_u. The p_u are in turn optimized with an upper-level MEA algorithm O_u. The expected fitness $F_l(p_l)|p_u$ is taken as the fitness of p_u. In this manner, the neuroevolution process can be optimized automatically, which makes it possible to solve harder problems with it. (*b*) Neuroevolution with eight hand-tuned evolution parameters (HNE) is successful in helicopter hovering task, but when those same parameters are optimized at the same time through bilevel evolution (HNE$_8$), better solutions are found faster. In this manner, bilevel evolution can be harnessed to improve upon human design of neuroevolution experiments. (*c*) The cumulative success of neuroevolution with five hand-tuned evolutionary parameters (PNE), five bilevel-optimized parameters (PNE$_5$), and fifteen bilevel-optimized parameters (PNE$_1$5) in the double pole balancing task. More parameters allow bilevel evolution to develop a more powerful neuroevolution parameterization, resulting in faster discovery of solutions. Therefore, when bilevel optimization is available, it is better to make the neuroevolution method more flexible and configurable, even beyond the human ability to optimize. For animations in helicopter hovering, see https://neuroevolutionbook.com/neuroevolution-demos. (Figures from Liang and Miikkulainen 2015)

Bilevel evolution is a special case of meta-evolutionary EAs (MEAs; Eiben and Smit 2011; Grefenstette 1986; Sinha et al. 2014) where evolution is used to optimize algorithms off-line. It is related to self-adaptive EAs where evolutionary parameters are adjusted on-line depending on progress in the optimization (Kramer 2010; Kumar et al. 2022). In its most straightforward form, each fitness evaluation of each high-level individual p_u requires running an entire neuroevolution experiment. The crucial idea of bilevel optimization is to estimate the fitness of p_u without having to run such an experiment every time. In essence, the idea is the same as surrogate optimization for decision-making, discussed in Section 6.2.2. Each run of a neuroevolution experiment can be considered as a sample, and a predictor model learned to approximate the fitness landscape. The upper-level search can then be done mostly against the surrogate, with only occasional neuroevolution experiments needed.

A simple approach is to fit e.g. a quadratic function to these samples (Sinha et al. 2014). A more complex one is to train a random forest or a neural network, as was done in Section 6.2.2: Such models are nonparametric, i.e. more general, and less prone to overfitting. Forming the surrogate is still difficult because there are usually very few samples and they are noisy. One way to deal with this problem is to construct the fitness F_u from multiple metrics over several neuroevolution runs with p_u, including best and average fitness and standard deviation, diversity of the population, and the shape of the learning curve. In effect, the idea is to predict the eventual performance of p_u after prolonged evolution, and to take into account the reliability of this estimate.

To see the value of bilevel optimization, consider e.g. the benchmark task of evolving a neural network for helicopter hovering. The goal is to keep the helicopter as close as possible to a point in 3D space in windy conditions, with 12 state variables (coordinates, angles, velocities) as the input, and four action variables (aileron, elevator, rudder, and rotor pitch) as the output. The task is difficult because there are many variables that interact, their values are noisy, and the domain is unstable. However, neuroevolution can solve it with a careful hand-tuning of eight evolutionary parameters: mutation probability, rate, amount, replacement rate and fraction, population size, crossover probability, and crossover averaging rate (Koppejan and Whiteson 2011). Remarkably, such hand-tuning still leaves money on the table: by optimizing the parameter further with bilevel evolution, it is possible to evolve solutions that perform significantly better, both by learning faster and achieving better final accuracy (Figure 11.1*b*). Also, using a good surrogate is crucial: while using a random forest surrogate improves bilevel optimization significantly compared to not using a surrogate, quadratic fitting is too unreliable and actually decreases performance.

A common rule of thumb is that humans can take into account seven +/- two variables at once, which is well in line with the helicopter hovering result. However, with bilevel evolution, it may be possible to increase the number of variables significantly. Would such an extension result in better performance? For instance in the standard benchmark task of double pole balancing, it is common to specify the values of five parameters by hand: mutation rate and amount, replacement fraction, initial weight range, and population size. There are, however, many other parameters that could be included, such as 1-pt, 2-pt, and uniform crossover probability, tournament, truncation, and roulette selection probability, etc. They are not strictly necessary to parameterize an effective neuroevolution experiment, but they do make it possible to establish a more complex search.

It turns out such extra customization pays off significantly. It is much faster to find solutions when 15 evolutionary parameters are optimized rather than only five (Figure 11.1)*c*. This is an important result because it suggests that bilevel optimization changes how we should think about problem-solving. Simple methods may be easy to understand for people, but when they can be optimized automatically, it is better to make the method more flexible and configurable, even beyond human ability. Such complexity translates to better performance through bilevel optimization.

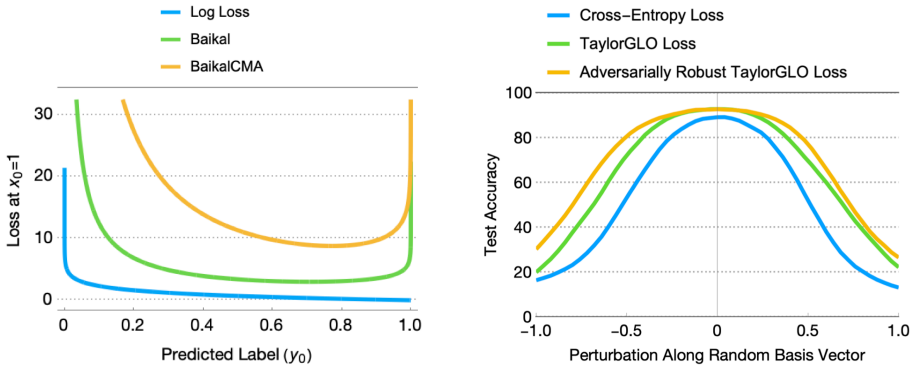
As more compute becomes available, bilevel optimization is likely to become an increasingly important element of neuroevolution. It can also be extended in several ways. For instance, instead of fixed parameters p_u, it may be possible to discover parameter adaptation schedules that change the parameters during the course of individual neuroevolution runs, similarly to self-adapting EAs. They may themselves take the form of a neural network that observes the performance of the run and outputs optimal current parameters at its output. While the designs of neuroevolution algorithms has naturally focused on compact and parsimonious methods, it may be possible to design them with bilevel optimization in mind, which means creating many more configuration parameters, and thus take advantage of the power of expanded optimization. Also, better surrogate modeling techniques can be developed, perhaps by utilizing knowledge of the domain, benchmark collections, and methods for estimating fitness in neural architecture search.

11.3 Evolutionary Metalearning

Besides the architecture, there are several other aspects of machine learning system design that need to be configured properly for the system to perform well. Those include learning hyperparameters (such as the learning rate), activation functions, loss functions, data sampling and augmentation, and the methods themselves. Approaches similar to those used in NAS can be applied to them; however, the evolutionary approach has an advantage in that it is the most versatile: It can be applied to graphs, vectors of continuous and discrete parameters, and configuration choices. This ability is particularly useful as new architectures are developed. For instance, at this writing, work has barely begun on optimizing designs of transformer (Vaswani et al. 2017) or diffusion (Sohl-Dickstein et al. 2015) architectures. They have elements such as attention modules, spatial embeddings, and noise transformations that are different from prior architectures, yet may be parameterized and evolution applied to optimize their implementation. Most importantly, evolution can be used to optimize many different aspects of the design at the same time, discovering and taking advantage of synergies between them. Several such approaches are reviewed in this section.

11.3.1 Loss functions

Perhaps the most fundamental is the design of a good loss function. The mean-squared-error (MSE) loss has been used for a long time, and more recently the cross-entropy (CE) loss has become popular, especially in classification tasks. Both of those assign minimal loss to outputs that are close to correct, and superlinearly larger losses to outputs further away from correct values. They make sense intuitively and work reliably, so much so that alternatives are not usually even considered.



(*a*) Loss function profiles (*b*) Performance with weight perturbation

Figure 11.2: **Regularization and Robustness with Evolved Loss Functions.** When loss functions are evolved as part of the optimization process, surprising synergies emerge. (*a*) The standard loss function, such as Log Loss (or Cross-Entropy) has a high loss for outputs that are far from correct (1.0 in this case) and a low loss otherwise. In contrast, evolutionary optimization of loss functions through GLO/TaylorGLO (Gonzalez and Miikkulainen 2020, 2021b) discovered a new principle: When the output is very close to the correct one, a high loss is incurred. This principle, termed Baikal loss for its shape, discourages overfitting, thus regularizing the network automatically, leading to better generalization. Such a loss is effective but counterintuitive, and thus unlikely to be discovered by human designers. (*b*) The Baikal loss also makes the network performance more robust. This effect can be quantified by perturbing the network weights. With Baikal loss, the network's performance is less affected than with Cross-Entropy loss. This effect can be further magnified by making robustness against adversarial inputs an explicit second objective in evolution. Thus, loss-function optimization can be used to improve not just regularization, but robustness as well. (Figures from Gonzalez and Miikkulainen 2020, 2021a)

However, it turns out that it is possible to improve upon them, in a surprising way that would have been difficult to discover if evolution had not done it for us (Gonzalez and Miikkulainen 2020, 2021b). If outputs that are extremely close to correct are penalized with a larger loss, the system learns to avoid such extreme output—which minimizes overfitting (Figure 11.2*a*). Such loss functions, called Baikal loss for their shape, lead to automatic regularization. Regularization in turn leads to more accurate performance on unseen examples, especially in domains where the amount of available data is limited, as is the case in many real-world applications.

Baikal loss was originally discovered with a classic genetic programming approach where the function was represented as a tree of mathematical operations (Gonzalez and Miikkulainen 2020). The structure of the tree was evolved with genetic algorithms, and the coefficients in the nodes with CMA-ES (N. Hansen and A. Ostermeier 2001). This approach is general and creative in that it can be used to explore a large search space of diverse functions. However, many of those functions do not work well and often are not even stable. In the follow-up TaylorGLO method (Gonzalez and Miikkulainen 2021b), the functions were

represented instead as third-order Taylor polynomials. Such functions are continuous and can be directly optimized with CMA-ES, making the search more effective.

Regularization in general is an important aspect of neural network design, there are many techniques available, such as dropout, weight decay, and label smoothing (Srivastava et al. 2014; Szegedy et al. 2016; Hanson and Pratt 1988), but how they work is not well understood. Loss-function optimization, however, can be understood theoretically, and thus provides a starting point to understanding regularization in general (Gonzalez and Miikkulainen 2021a). It can be described as a balance of two processes, one a pull toward the training targets, and another a push away from overfitting. The theory leads to a practical condition for guiding the search toward trainable functions.

Note that Baikal loss is a general principle; evolutionary optimization was crucial in discovering it but it can now be used on its own in deep learning. It is still possible to customize it for each task and architecture, and even small modifications to the standard Baikal shape may make a difference. Optimization may also have a significant effect on various learning challenges, for instance when there is not much training data (Gonzalez, Landgraf, and Miikkulainen 2019), or when the labels are particularly noisy (Gao, Gouk, and Hospedales 2021). It may also be possible to modify the loss function during the course of learning, for instance by emphasizing regularization in the beginning and precision towards the end (similarly to activation functions; Section 11.3.2).

It turns out that loss functions that regularize also make networks more robust, and this effect can be further enhanced by including an explicit robustness goal in evolution (Figure 11.2*b*). One way to create such a goal is to evaluate performance separately wrt. adversarial examples. This result in turn suggests that loss-function optimization could be an effective approach to creating machine learning systems that are robust against adversarial attacks.

Loss-function optimization can also play a major role in systems where multiple loss functions interact, such as Generative Adversarial Networks (GANs; (Gonzalez, Kant, and Miikkulainen 2021)). GANs include three different losses: discriminative loss for real examples and for fake examples, and the generative loss (for fake examples). It is difficult to get them right, and many proposals exist, including those in minimax, nonsaturating, Wasserstein, and least-squares GANs (Arjovsky, Chintala, and Bottou 2017; Mao et al. 2017; Goodfellow et al. 2014). Training often fails, resulting e.g. in mode collapse. However, the three losses can be evolved simultaneously, using performance and reliability as fitness. In one such experiment on generating building facade images given the overall design as a condition, the TaylorGLO approach was found to result in better structural similarity and perceptual distance than the Wasserstein loss (Gonzalez, Kant, and Miikkulainen 2021). Although this result is preliminary, it suggests that evolutionary loss-function optimization may make more complex learning systems possible in the future.

11.3.2 Activation functions

Early on in the 1980s and 1990s, sigmoids (and tanh) were used almost exclusively as activation functions for neural networks. They had the intuitively the right behavior as neural models, limiting activation between the minimum and maximum values, a simple derivative that made backpropagation convenient, and a theorem suggesting that universal computing could be based on such networks (Cybenko 1989; Hornik, Stinchcombe, and White 1989).

There were indications, however, that other activation functions might work better in many cases. Gaussians achieved universal computing with one less layer, and were found powerful in radial basis function networks (Park and Sandberg 1991). Ridge activations were also found to provide similar capabilities (Light 1992).

However, with the advent of deep learning, an important discovery was made: Activation function actually made a big difference wrt. vanishing gradients. In particular, rectified linear units (ReLUs), turned out important in scaling up deep learning networks (Nair and Hinton 2010). The linearly increasing region does not saturate activation or gradients, resulting in less signal loss. Moreover, it turned out that in many cases ReLU could be improved by adding a small differentiable dip at the boundary between the two regions, in a function called Swish (Ramachandran, Zoph, and Le 2017). This result suggested that there may be an opportunity to optimize activation functions, in general and for specific architectures and tasks.

Like with loss functions, there is a straightforward opportunity in evolving functions through genetic programming (Bingham, Macke, and Miikkulainen 2020). Similarly to loss functions, such an approach can be creative, but also results in many functions that make the network unstable. A more practical approach is to limit the search space to e.g. computation graphs of two levels, with a focused set of operators, that are more likely to result in useful functions. This approach was taken e.g. in the Pangaea system (Bingham and Miikkulainen 2022). Given a list of 27 unary and seven binary operators, two basic two-level computation graph structures, and four mutation operators, evolution can search a space of over ten trillion activation functions.

However, finding an effective function is only part of the challenge. The function also needs to be parameterized so that it performs as well as possible. While coefficients multiplying each operator can be evolved together with the structure, it turns out that such fine tuning can be done more efficiently through gradient descent. In other words, in Pangaea evolution and gradient descent work synergetically: evolution discovers the general structure of the function, and gradient descent finds its optimal instantiation.

The method is powerful in two ways: it finds general functions that perform better than previous functions (such as ReLU, SeLU, Swish, etc.) across architectures (such as All-CNN, Wide ResNet, Resnet, and Preactivation Resnet) and tasks (such as CIFAR-10, CIFAR-100). However, it is most powerful in discovering activation functions that are specialized to architecture and task, apparently taking advantage of the special requirements in each such context.

Furthermore, performance can be further improved by allowing different functions at different parts of the network, and at different times throughout training (Figure 11.3). The optimal designs change continuously over time and space. Different activation functions are useful early in training when the network learns rapidly and late in training when fine-tuning is needed; similarly, more nonlinear functions are discovered for later layers, possibly reflecting the need to form a regularized embedding early, and make classification decisions later.

The Pangaea results suggest an intriguing duality: While neural network learning is mostly based on adapting a large number of parameters (i.e. weights), perhaps a similar effect might be achieved by adapting the activation functions over space and time? Perhaps

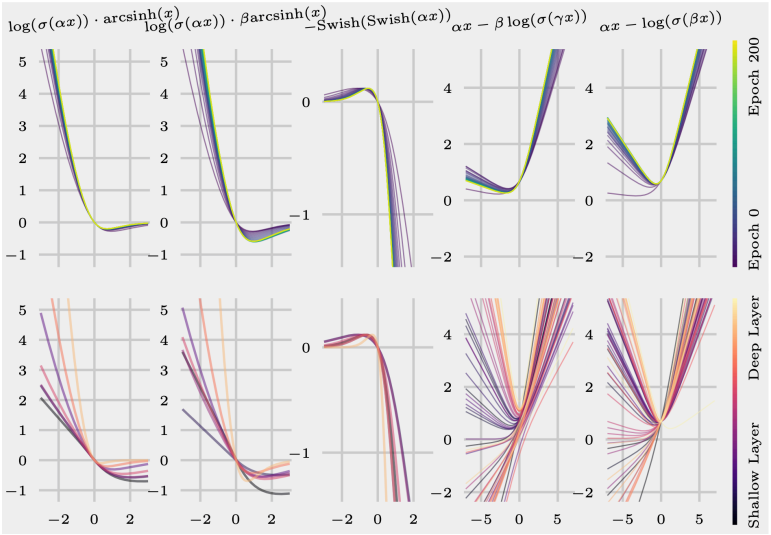


Figure 11.3: **Activation Functions Discovered over Space and Time.** Activation functions can be seen as fundamental to network performance as its weights. Pangaea (Bingham and Miikkulainen 2022) combines evolution of activation function structure synergetically with gradient descent of its parameters. It is possible to discover general functions, but the approach is most powerful in customizing them to a particular architecture and task. Moreover, the functions change systematically over learning time as well as through different depths of layers, presumably starting with coarse learning and regularization and transforming into fine-tuning and classification. These results suggest a possible duality with weight learning, and a possible synergy for the future. (Figure from Bingham and Miikkulainen 2022)

the two mechanisms could be used synergetically? Evolution of the activation function structure provides the foundation for this approach, which still needs to be developed fully.

11.3.3 Data use and augmentation

Another important opportunity for evolutionary optimization of supervised learning systems is to optimize the training data. For instance, it may be possible to form embeddings of the training samples through an autoencoder, and then form a strategy for utilizing different kinds of samples optimally through time (Gonzalez, Landgraf, and Miikkulainen 2019). In this manner, evolution could discover ways for balancing an imbalanced dataset or designing curricular learning from simple to more complex examples. Especially in domains where not a lot of labeled samples are available, such techniques could result in significant improvements. It may also be possible to extend the methods to utilize multiple datasets optimally over time in a multitask setting.

Another possibility is to evolve methods for augmenting the available data automatically through various transformations. Different datasets may benefit from different transformations, and it is not always obvious ahead of time how they should be designed. For instance, in an application to develop models for estimating the age of a person from an image of

their face, evolution was used to decide vertical and horizontal shift and cutout, as well as a direction of flip operations, angle of rotation, degree of zoom, and extent of shear (Miikkulainen, Meyerson, et al. 2021). Unexpectedly, it chose to do vertical flips only—which made little sense for faces, until it was found that the input images had been rotated 90 degrees! It also discovered a combination of shift operations that allowed it to obfuscate the forehead and chin, which would otherwise be easy areas for the model to overfit.

A particularly interesting use for evolved data augmentation is to optimize not only the accuracy of the resulting models but also to mitigate bias and fairness issues with the data. As long as these dimensions can be measured (Sharma, Henderson, and Ghosh 2020), they can be made part of the fitness, or separate objectives in a multiobjective setting. Operations then need to be designed to increase variance across variables that might otherwise lead to bias through overfitting—for instance gender, ethnicity, and socioeconomic status, depending on the application. While evolutionary data augmentation is still new, this area seems like a differentiated and compelling opportunity for it.

11.3.4 Learning methods

An interesting extension of NAS is to evolve the learning system not from high-level elements, but from the basic algorithmic building blocks (mathematical operations, data management, and ways to combine them)—in other words, by evolving code for supervised machine learning. In this manner, evolution can be more creative in discovering good methods, with fewer biases from the human experimenters.

The AutoML-Zero system (Real et al. 2020) is a step towards this goal. Given an address space for scalars, vectors, and matrices of floats, it evolves setup, predict, and learn methods composed of over 50 basic mathematical operations. Evolution is implemented as a linear GP, and consists of inserting and removing instructions and randomizing instructions and addresses. Evaluation consists of computing predictions over unseen examples.

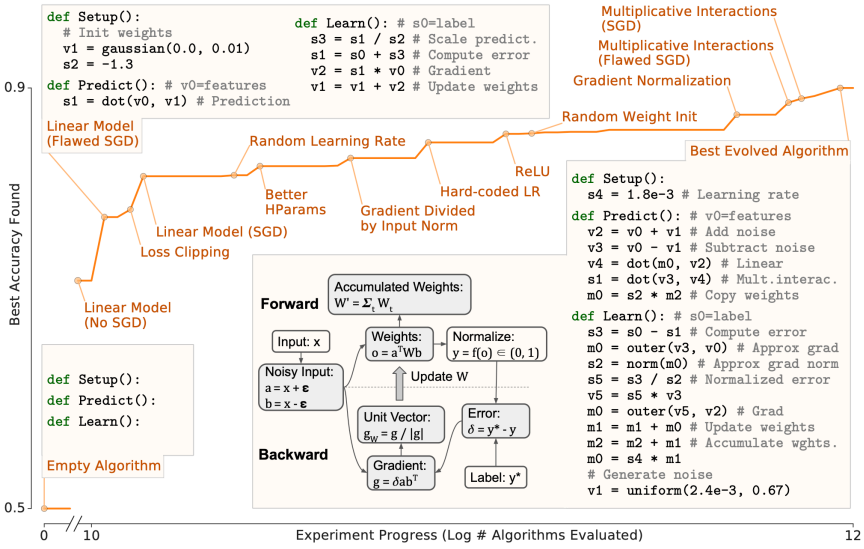
Starting from empty programs, AutoML-Zero first discovered linear models, followed by gradient descent, and eventually several extensions known in the literature, such as noisy inputs, gradient normalization, and multiplicative interactions (Figure 11.4). When given small datasets, it discovers regularization methods similar to dropout; when given few training steps, it discovers learning-rate decay.

Thus, the preliminary experiments with AutoML-Zero suggest that evolutionary search can be a powerful tool in discovering entire learning algorithms. As in many metalearning approaches, the main power may be in customizing these methods to particular domains and constraints. A crucial aspect will be to guide the evolution within the enormous search space toward meaningful solutions, without hampering its ability to create, again a challenge shared with most of metalearning.

11.3.5 Utilizing surrogates

While evolutionary metalearning can discover more effective neural network designs, it is also challenging in three ways: It is computationally very expensive to evaluate all the different designs; it is difficult to gain insight into what works; and it is not clear how the search spaces should be defined so that they are fast to search and contain good solutions.

One way to make progress toward meeting these challenges is to perform a full search in as large a search space as possible, thus forming a benchmark dataset that makes it possible




```
def Setup():
  # Init weights
  v1 = gaussian(0.0, 0.01)
  s2 = -1.3
def Predict(): # v0=features
  s1 = dot(v0, v1) # Prediction
```

```
def Learn(): # s0=label
  s3 = s1 / s2 # Scale predict.
  s1 = s0 + s3 # Compute error
  v2 = s1 * v0 # Gradient
  v1 = v1 + v2 # Update weights
```

```
def Setup():
  s4 = 1.8e-3 # Learning rate
def Predict(): # v0=features
  v2 = v0 + v1 # Add noise
  v3 = v0 - v1 # Subtract noise
  v4 = dot(m0, v2) # Linear
  s1 = dot(v3, v4) # Mult.interac.
  m0 = s2 * m2 # Copy weights
def Learn(): # s0=label
  s3 = s0 - s1 # Compute error
  m0 = outer(v3, v0) # Approx grad
  s2 = norm(m0) # Approx grad norm
  s5 = s3 / s2 # Normalized error
  v5 = s5 * v3
  m0 = outer(v5, v2) # Grad
  m1 = m1 + m0 # Update weights
  m2 = m2 + m1 # Accumulate wghts.
  m0 = s4 * m1
  # Generate noise
  v1 = uniform(2.4e-3, 0.67)
```

Figure 11.4: **Evolutionary Discovery of Learning Methods.** At the highest level, metalearning extends to the learning mechanisms themselves. In AutoML-Zero (Real et al. 2020), sequences of instructions for setup, prediction, and learning are evolved through mutation-based regularized search. AutoML-Zero first discovered simple methods such as linear models, then several known extensions such as ReLU and gradient normalization, and eventually more sophisticated techniques such as multiplicative interactions. The approach could potentially be useful in particular in customizing learning methods to different domains and constraints. (Figure from Real et al. 2020)

to analyze what works. These insights may then be used to construct a surrogate approach that makes it possible to search in larger spaces without having to evaluate candidates through full training.

Such an approach, AQuaSurF, was demonstrated in the task of discovering effective activation functions (Bingham and Miikkulainen 2023b). Based on the work described in Section 11.3.2, an exhaustive set of 2913 different activation functions were created from a three-node computational graph of Pangaea and tested on three architecture/task settings, All-CNN/CIFAR-10, ResNet-56/CIRAF-10, and MobileViTv2-0.5/Imagenette. Thus, they covered basic convolutional, residual, and transformer designs in the visual domain. In each case, the networks were trained fully to evaluate how well each function performed in the particular setting.[2]

Most activation functions performed poorly, but a small number of functions performed very well, confirming that activation-function metalearning is difficult but also worthwhile. Most interestingly, two trends were also observed: (1) There were clusters of functions that performed well across architectures and architectures, representing refinements of general solutions; and (2) the very best performance in each setting was achieved by a few functions that performed poorly in other settings, in other words, by activation functions that were

2. This dataset is available at https:// github.com/cognizant-ai-labs/act-bench.

specialized to the architecture and task. This result suggests that metalearning can be most powerful when it is used to customize the designs to the particular problem.

The benchmark collection was then used to construct an effective surrogate for full network evaluations. It turned out that a combination of Fisher-information-matrix (FIM) eigenvalues and the function shape is a powerful surrogate.

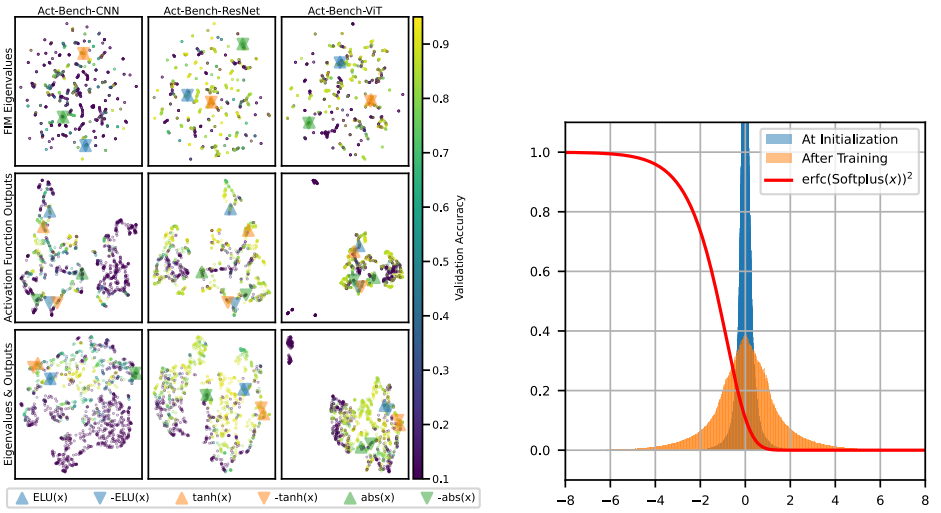
First, FIM quantifies how much information the network parameters carry about the data distribution, and thus serves as a characterization of network behavior. It has been used in many studies to illustrate learning ability, generalization, robustness to perturbations, and loss-function shape of neural networks (Jastrzebski et al. 2021; T. Liang et al. 2019; Liao et al. 2018; Karakida, Akaho, and Amari 2019). The information in FIM is represented compactly in its eigenvalues; there are as many eigenvalues as there are network weights, but they can be binned into a histogram of a lower dimensionality. The histogram vector then forms a computational characterization of the network. Networks with different activation functions have different such characterizations, and the space of this FIM-eigenvalue-histogram vectors can be used as a surrogate search space for good activation functions.

However, the FIM depends on other factors as well, including the architecture, loss function, and data distribution, which makes them rather noisy. An additional surrogate representation is useful in compensating for such noise: the shape of the activation function itself. This shape can be represented as a sampling of activation function values for inputs distributed as $\mathcal{N}(0, 1)$, as they would be in a properly initialized network (Bingham and Miikkulainen 2023a). Using both FIM and output together form a powerful surrogate (Figure 11.5*a*): functions that perform similarly are clustered together, making it easy to search for good functions.

Indeed the search for good activation functions was highly effective in this surrogate space. Even a simple search like k-nearest neighbors regression could find the best functions fast and reliably.

However, the surrogate approach also turned out effective in activation optimization beyond the benchmark settings in three ways. First, it scaled up to a much larger search space of 425,896 functions for which the performance was not known, as well as to the harder CIFAR-100 task with the same architectures. In each case, it discovered new activation functions that performed better than any of the known functions so far. Second, those discoveries also transferred to new settings: The best functions performed better than any previously known functions on ResNet-50 on the full ImageNet dataset. Thus, it is possible to discover good functions efficiently in smaller tasks and then use them to improve performance in larger ones. Third, the approach also extended to new architectures and baseline functions. For instance with CoAtNet Dai et al. 2021 on Imagenette, initialized with the best previously known activation functions, the approach outperformed all baselines. Thus, the surrogate approach is a powerful way to optimize designs for new settings.

Interestingly, AQuaSurF was able to achieve these results by balancing refinement and novelty. Many of the functions it discovered were similar e.g. to the well-known functions of ELU and Swish, with minor changes to their shape. This result suggests that these are generally good functions, but also that such customizations matter; AQuaSurF is well-equipped to find them.



(*a*) Surrogate spaces (*b*) Using the sigmoid

Figure 11.5: **Utilizing surrogates to discover surprising activation functions.** Surrogate modeling can be used to evaluate activation function candidates without full training, making it possible to search in larger spaces which may result in more innovative solutions. (*a*) UMAP embeddings of the 2913 activation functions in the three benchmark settings (columns) in three different surrogate spaces: FIM eigenvalues (top row), function outputs (middle row), and both (bottom row). UMAP is a dimensionality-reduction technique that preserves the structure of high-dimensional spaces well, in this case 13692, 16500, and 11013 FIM eigenvalue histogram dimensions and 1000 function output samples. Function performance is indicated by color coding. Similar colors cluster best in the bottom row, suggesting that using both FIM and output features as the surrogate space makes search for good functions the easiest. (*b*) The best activation function in the CoAtNet experiment turned out to be a sigmoid. The histograms indicate values with which it is activated in the network. At initialization (blue histogram), it is used similarly to ReLU; after training (orange histogram) both saturation regions are used. This discovery suggests that sigmoidal activations may be useful in specific situations, challenging the conventional wisdom in deep learning. (Figures from Bingham and Miikkulainen 2023b)

However, in many cases, AQuaSurF also found designs that were very different from the existing ones, yet performed at least as well. Some of them had discontinuous derivatives, some did not saturate at either side, and some had positive instead of negative bumps. The biggest surprise was discovered in the CoAtNet experiment on ImageNette (Figure 11.5*b*). This function was essentially a sigmoid, similar to those used extensively during the early days of neural networks, but largely discarded in favor of ReLU in deep learning. Why would it be discovered again in these experiments?

In deep learning, the linearly increasing region of ReLU helped avoid vanishing gradients. Interestingly, if we look at how the sigmoid is used, by plotting which parts of the function are actually activated during performance, it indeed provides behavior similar to

ReLU early in training: The function is activated around the nonlinearity, but does not reach the saturating region that occurs with larger activations. However, later training also takes advantage of the saturating region. In this manner, the same activation function can be used in two ways: presumably to keep the gradients from vanishing early, and to commit to decisions later. This result challenges the common approach in deep learning design, and demonstrates the power of neuroevolution in metalearning good designs.

In sum, surrogate optimization techniques make it possible to scale up neuroevolution metalearning; in doing so, it is possible to identify principles that would be difficult for human designers to discover.

11.3.6 Synergies

Perhaps the most important future direction in evolutionary metalearning is to discover and utilize synergies between the different aspects of the learning system design. For instance, the best performance was reached by optimization activation functions for the specific architecture; It might be possible to optimize the architecture simultaneously to emphasize this effect.

Simply running evolution on all these design aspects simultaneously is unlikely to work; the search space would be prohibitively large. Similarly, adding more outer loops to the existing process (where supervised learning is the inner loop and metalearning is the outer loop) is likely prohibitive as well. However, it might be possible to alternate evolution of different aspects. Better yet, techniques from bilevel (or multilevel) optimization could be useful—the idea is to avoid full inner-outer loop structure, but instead use e.g. surrogate models to evaluate outer loop innovations (Sinha et al. 2014; Liang and Miikkulainen 2015).

A practical approach is to simply add constraints, and search in a smaller space. A first such step was already taken in the EPBT system (J. Liang et al. 2021), which combines hyperparameter tuning, loss-function optimization, and population-based training (PBT) into a single loop. That is, hyperparameters and loss functions are evolved at the same time as the networks are being trained. Hyperparameter tuning is limited to those that do not change the structure of the networks (e.g. learning rate schedules) so that they can be continuously trained, even when the hyperparameters change. Similarly, loss-function optimization is limited to TaylorGLO coefficients (J. Liang et al. 2021) that can be changed while training is going on. Even so, the simultaneous evolution and learning was deceptive, and needed to be augmented with two mechanisms: quality-diversity heuristic for managing the population and knowledge distillation to prevent overfitting. The resulting method worked well on optimizing ResNet and WideResnet architectures in CIFAR-10 and SVHN, but also illustrates the challenges in taking advantage of synergies of metalearning methods.

Similarly, promising results were obtained in an experiment that compared human design with evolutionary metalearning (Miikkulainen, Meyerson, et al. 2021). Using the same datasets and initial model architectures, similar computational resources, and similar development time, a team of data scientists and an evolutionary metalearning approach developed models for age estimation in facial images (Figure 11.6). The evolutionary metalearning approach, LEAF-ENN, included optimization of loss functions (limited to linear combinations of MSE and CE), learning hyperparameters, architecture hyperparameters, and data augmentation methods. Evolution discovered several useful principles that the data scientists were not aware of: Focusing data augmentation to regions that mattered most, and