nature methods

REVIEW ARTICLE

https://doi.org/10.1038/s41592-019-0403-1

Deep learning for cellular image analysis

Erick Moen[1,3], Dylan Bannon[1,3], Takamasa Kudo[2], William Graf[1], Markus Covert[2] and David Van Valen[1]*

Recent advances in computer vision and machine learning underpin a collection of algorithms with an impressive ability to decipher the content of images. These deep learning algorithms are being applied to biological images and are transforming the analysis and interpretation of imaging data. These advances are positioned to render difficult analyses routine and to enable researchers to carry out new, previously impossible experiments. Here we review the intersection between deep learning and cellular image analysis and provide an overview of both the mathematical mechanics and the programming frameworks of deep learning that are pertinent to life scientists. We survey the field's progress in four key applications: image classification, image segmentation, object tracking, and augmented microscopy. Last, we relay our labs' experience with three key aspects of implementing deep learning in the laboratory: annotating training data, selecting and training a range of neural network architectures, and deploying solutions. We also highlight existing datasets and implementations for each surveyed application.

Advances in imaging have transformed the biological sciences, enabling researchers to access temporal and spatial variations inherent in living systems. Progress in optics has yielded microscopes capable of imaging over a range of spatial scales, from single molecules to entire organisms. Concurrently, improvements in fluorescent probes have enhanced the brightness, photostability, and spectral range of fluorescent proteins and of small-molecule dyes. Combined, these advances allow for a variety of dynamic measurements in living cells, from long-term imaging of single molecules[1,2], to simultaneous measurements of multiple biosensors[3,4], to observations of the development of entire organisms[5–9]. They have also led to impressive measurements in fixed samples, with spatial genomics now driving the simultaneous measurement of dozens of proteins or thousands of mRNA species in fixed cells and tissues while preserving spatial information[10–12].

Concurrent with these technological advances has been an increasing demand in the biosciences for image analysis. Modern imaging data increasingly require quantification to be informative[13]. Typical tasks include unsupervised image exploration (comparing features of collections of images, for example, by identifying changes in cellular morphology in an imaging-based drug screen), image classification (predicting a label for an image—for example, determining whether a stem cell has differentiated), image segmentation (identifying the parts of an image that correspond to distinct objects—for example, identifying single cells in images), and object tracking (following an object—for example, a single cell in a live embryo—among frames of a movie). In response to this demand, researchers and companies have developed software libraries, use-case-based implementations, and general-purpose computer vision ecosystems. MATLAB was one of the first commercial platforms to support solutions for computer vision and continues to enjoy frequent use. Recently, the development of open-source data-science libraries for Python (e.g., NumPy[14], SciPy[15], Pandas[16], Scikit-image[17], Scikit-learn[18], Matplotlib[19], and Jupyter[20]) has led to a rise in Python's popularity. Both MATLAB and Python now contain ready-made implementations of common computer-vision algorithms. Traditionally, experimentalists wrote software tools that drew from these libraries. As analysis tasks became more common, several software tools were created to improve accessibility through a graphical front-end. For example, there are tools for single-cell analysis of bacteria (SuperSegger[21], Oufti[22], Morphometrics[23]), single-cell analysis of mammalian cells (CellProfiler[24,25], Ilastik[26], Microscopy Image Browser[27]), and general-purpose image analysis (ImageJ[28], OMERO[29]). These tools and ecosystems have transformed experimental design, rendered quantitative and statistical analyses automatable and high-throughput, and yielded a plethora of critical biological insights.

Excitingly, deep learning has expanded the range of problems that computer vision can solve[30]. Here, "deep learning" refers to a set of machine-learning techniques, specifically, neural networks that learn effective representations of data with multiple levels of abstraction[30]. Note the contrast with conventional machine learning, in which representations are manually designed through feature engineering. In deep learning, the learning can be supervised or unsupervised. Supervised approaches, which have been the most successful, attempt to maximize performance on an annotated dataset. Unsupervised approaches are used to reconstruct original data after compression into a low-dimensional space. Although these techniques have existed in mathematical form for several decades, they gained attention when a deep-learning-based method won the 2012 ImageNet Large Scale Visual Recognition Challenge[31]. Since then, there has been a major increase in the variety of problems that can be solved with deep learning. In addition, improvements in computer hardware and deep learning frameworks have placed these tools within reach of the typical software developer. While deep learning has been predominantly applied commercially, it is now starting to emerge in the physical[32–34], chemical[35,36], medical[37,38], and biological sciences[39–42] with applications for images and other data types.

Given the central role that observation—and therefore imaging—plays in the biological sciences, deep learning has the potential to revolutionize understanding of the inner workings of living systems. Indeed, currently a 'gold rush' is taking place, with numerous groups seeking to apply these methods to their data in order to extract novel biological insights. Nonetheless, deep learning has yet to be widely adopted throughout the life and medical sciences. Importantly, many of the software tools mentioned above (with the recent exceptions of CellProfiler[25] and ImageJ[43]) do not yet feature

[1]Division of Biology and Bioengineering, California Institute of Technology, Pasadena, CA, USA. [2]Department of Bioengineering, Stanford University, Stanford, CA, USA. [3]These authors contributed equally: Erick Moen, Dylan Bannon. *e-mail: vanvalen@caltech.edu

deep learning. In our opinion, for deep learning to truly transform the life sciences, its application needs to be as routine as BLAST searches. The barriers to spreading deep learning throughout biology labs are both cultural and technical. The mathematics renders some of the inner workings of deep learning algorithms opaque; the unique requirements of deep learning necessitate a different way of thinking about writing software. Specifically, the need for annotated data means that data and software must be jointly developed—an approach recently termed Software 2.0[44] (Fig. 1). The amount of data and the computational resources required for deep learning constitute a significant barrier to adoption, as does the knowledge required to optimize model performance and to interpret what deep learning models have learned. To harness the full power of these tools, life scientists must familiarize themselves with them to enhance their existing workflows and to set the stage for currently unforeseen analyses.

By focusing on use cases that are common in quantitative cell biology, this Review serves as a practical introduction to deep learning for the analysis of biological images. It builds on prior reviews of the intersection of deep learning and the life sciences[42,45–47] by incorporating a discussion of our labs' joint experiences in applying these methods to cellular imaging data, and is meant to make these methods less opaque to new adopters. First, we review the practical mechanics of deep learning, including the mathematical underpinnings, recent advances in neural-network architectures, and existing software frameworks. Next, we outline what we feel are the key components of effective, laboratory-scale deep learning solutions. We then review four use cases: image classification, image segmentation, object tracking, and augmented microscopy. For each use case, we cover problem specification, the state of the field with respect to algorithms and biological applications, and publicly available datasets. We close by sharing some of the lessons that our labs have learned while adapting these methods to biological data, and by suggesting directions for future work.

The practical mechanics of deep learning

In deep learning, an algorithm learns effective representations for a given task entirely from data. An introduction to the mathematics underlying the training of deep learning models is given in Box 1, troubleshooting advice is given in Box 2, and a glossary of commonly used terms is given in Box 3. Because the most successful solutions have been supervised, we believe that there are three essential components to the successful application of deep learning to biological image analysis: construction of a pertinent and annotated training dataset, effective training of deep learning models on that dataset, and deployment of trained models on new data.

Training data are critical to successful applications of deep learning; this requirement is one of the key disadvantages of this method. In our experience, assembling sufficient high-quality data often takes as much, if not more, time as programming the deep learning solution. Robust solutions require datasets that capture the diversity of images likely to be encountered during analysis. As much as possible, annotations for these datasets need to be error free, because errors can be learned. While training data may be limited, computational approaches can extract the most utility out of existing data. Image normalization reduces variation from distinct acquisition conditions[42,48]. Data-augmentation operations such as rotation, flipping, and zooming can also increase the image diversity in a limited dataset; these operations are generally standard practices regardless of the dataset size or type[49]. Transfer learning is another approach for creating robust models with limited data. In transfer learning, a deep learning model is trained on a large dataset to learn general image features, and then is fine-tuned on a smaller dataset to learn to perform a specific task[50,51]. While these approaches enable well-performing networks to emerge from limited datasets, considerable performance boosts arise from large annotated datasets[52]. For some

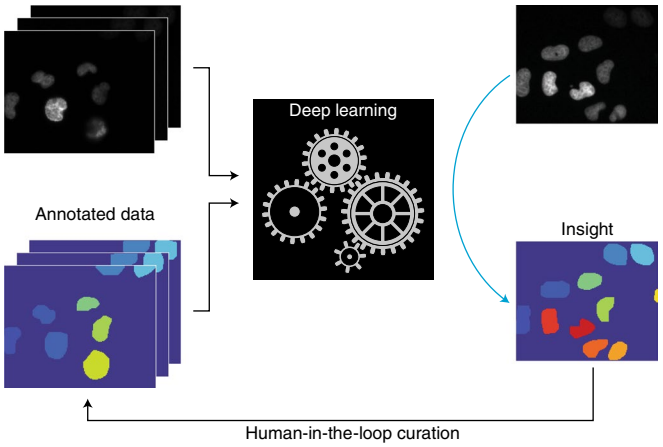


Fig. 1 | Software 2.0 combines data annotations with deep learning to produce intelligent software. Annotations produced by expert annotators or by a crowd can be used to train deep learning models to extract insights from data. Once trained, these models can be deployed to process new, unannotated data. The human-in-the-loop extension involves the identification of model errors, error correction to produce new training data, and retraining on an updated dataset.

uses, such as the detection of diffraction-limited spots, it has been possible to produce simulated images with a known annotation[53]. In other strategies, the curated outputs of traditional computer-vision pipelines have been used as training data[54]. Training data have also been produced manually by experts using annotation tools such as Fiji/ImageJ[48], Cellprofiler[52], and the Allen Cell Structure Segmenter[55]. Crowdsourcing, a cost-effective source of large datasets, is extensively used in fields such as automated driving; existing tools are being adapted for biological images. Enterprise commercial solutions include Figure Eight, which was recently acquired by Appen, and Samasource. The Quanti.us[56] tool features a graphical user interface for biological image annotation for use on Amazon Mechanical Turk, as does Amazon's Ground Truth tool, which uses active learning to reduce data-labeling costs. Gamification has also yielded some very promising results[57]. Importantly, the community acknowledges that the annotated datasets that power deep learning algorithms should be publicly available, as a comprehensive and expansive set of training data specific to biological problems would aid the development of deep learning algorithms considerably.

Once training data have been acquired, a deep learning model can be trained to accurately make predictions for new data. This task has several unique software and hardware requirements. Currently, Python is the most popular language for deep learning; existing frameworks include Tensorflow/Keras[58,59], PyTorch[60], MXNet[61], CNTK[62], Theano[63], and Caffe[64]. Although these frameworks have important differences, there are also several commonalities. First, all of them construct a computation graph that outlines all the computations made by a deep learning model as input data are transformed into the final output. Second, they all automatically perform derivatives, which enables them to carry out optimizations like those described in Box 1 without additional work by the user once the computation graph is specified. Third, they provide an easy gateway for specialized hardware such as graphical processing units (GPUs) and tensor processing units[65–67]. Because deep learning models often contain millions of parameters, specialized hardware is needed to perform these computations quickly. Fourth, these frameworks all contain implementations of common mathematical objects, optimization algorithms, hyperparameter settings, and performance metrics—meaning that users can quickly apply deep learning to their data without having to reproduce these implementations on their own. Although a considerable amount of

Box 1 | Training a linear image classifier

To illustrate the workflow for training a deep learning model in a supervised manner, here we consider the case of training a linear classifier to recognize grayscale images of cats and dogs. Each image is an array of size (N_x, N_y, 1), where N_x and N_y are the number of pixels in the x and y dimensions, respectively, and 1 is the number of channels in the image. For this exercise, we collapse the image into a vector of size (N_xN_y, 1). The classification task is to construct a function that takes this vector as input and predicts a label (0 for cats, 1 for dogs). A linear classifier performs this task by producing class scores that are a linear function of each pixel value. Mathematically, this is written as

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} w_{0,0} & \cdots & w_{0,NxNy-1} \\ w_{1,0} & \cdots & w_{1,NxNy-1} \end{bmatrix} \begin{bmatrix} x_0 \\ \vdots \\ x_{NxNy-1} \end{bmatrix} = \begin{bmatrix} \sum w_{0,j}x_j \\ \sum w_{1,j}x_j \end{bmatrix}$$

where y_0 and y_1 are the class scores, W is a matrix of class weights, and x is the image vector. The class with the highest score is the predicted class. The learning task then tunes the $w_{i,j}$ values so that a loss function that measures the classifier's performance on some training dataset is minimized. A common loss function is the cross-entropy, or softmax, loss. To arrive at this loss function, we first transform our class scores y_0 and y_1 into probabilities by defining

$$p_i = \frac{e^{\text{Class } i \text{ score}}}{\sum_{\text{All classes}} e^{\text{Class score}}}$$

These probabilities reflect the model's certainty that an image belongs in class i. The loss evaluated for a collection of images is defined as

$$\text{Loss} = -\sum_{\text{Images}} \log p_{\text{Correct}} + \lambda \sum_{i,j} w_{i,j}^2$$

where p_{Correct} is the probability assigned to the correct class for that image. This equation has two terms. The first can be thought of as the negative log likelihood of choosing the correct class. The second term is called L2 regularization; it penalizes large weights to control against overfitting.

To minimize the loss function, most optimization algorithms used in deep learning are a variation of stochastic gradient descent. First, the weights are randomly initialized to some small value. The choice of initialization can affect the training of deep models considerably; best practices for initialization include the initialization settings of He et al.[167]. Next, we select a small batch of images, called a minibatch, and identify the direction in which to change the weights so that the loss function will be reduced the most when evaluated on that minibatch. We then perturb the weights a small amount in that direction. The correct direction in which to perturb the weights is captured by the gradient of the loss function with respect to the weights. Mathematically, this gradient leads to the update rule

$$w_{i,j} \rightarrow w_{i,j} - \text{lr}\frac{\partial \text{loss}}{\partial w_{i,j}}$$

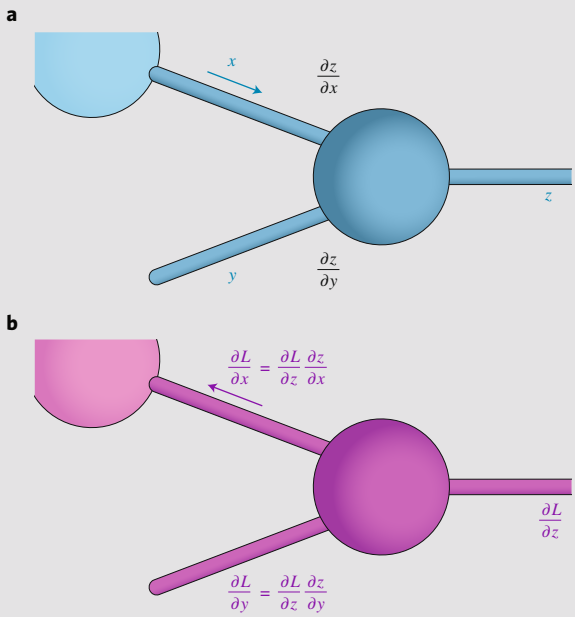
where lr, the scalar that scales the gradient at each step, is the learning rate. For the case of the linear classifier, we can compute these gradients analytically. The gradient for one image is given by

$$\frac{\partial \text{loss}}{\partial w_{i,j}} = x_j\left(p_i - 1\left(i \text{ is correct label}\right)\right) + 2\lambda w_{i,j}$$

where 1() is an indicator function that is 1 when the statement inside the parentheses is true and 0 when false. The gradient for the minibatch of images is the sum of the gradients for all images in the batch. This equation provides some information on how gradient descent changes the weights. The first term leads to an increase in weights that correspond to the correct class and a decrease in weights that correspond to the incorrect class. If the model is certain ($p_i \approx 1$) and correct, the contribution will be minimal; the opposite is true if the model is certain and wrong. The gradient is scaled by the relevant pixel value x_j, which causes the model to pay attention to bright pixels. The contribution of the regularization term pushes weights toward zero, preventing any one weight from getting too big. Once the weights are updated, the user then selects another batch of images and repeats the process until the loss is sufficiently minimized. The accuracy and loss of the algorithm on the validation dataset are often used to develop stopping criteria. Training is often stopped when the validation loss ceases to improve or when the training and validation error curves start to diverge, signifying overfitting.

While the linear classifier highlights several key features of training, in practice there are some important differences. Variants of the loss function shown above have been developed to address issues surrounding class imbalance in datasets. Several variants of stochastic gradient descent exist, including with momentum[168–170], RMSprop[171], Adagrad[172], Adadelta[173], and Adam[174]. Recent work suggests that networks trained with stochastic gradient descent with momentum have better performance with respect to generalization[175,176]. We have presented the learning rate as a static parameter, but in practice it often decreases as training progresses.

Importantly, the mathematical structure of deep learning models is more complicated than the linear model presented here. While this simplification may appear problematic with respect to analytical computation of the gradients for training, all deep learning models are compositional. This allows one to iteratively use the chain rule[177] to derive analytical expressions for the gradients, even for complicated functions, as shown in the figure in this box.



Computing gradients with backpropagation. **a**, During the forward pass, local derivatives are computed alongside the original computation. **b**, During the backward pass, the chain rule is used in conjunction with the local derivatives to compute the derivative of the loss function with respect to each weight.

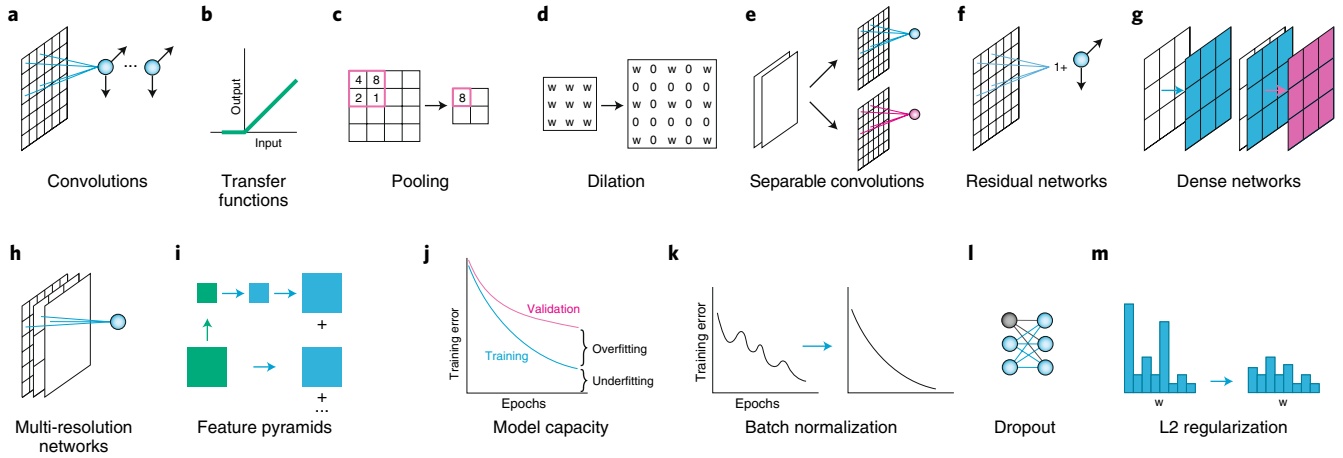


Fig. 2 | Common mathematical components of deep learning models. a, Convolutions extract local features in images, and the weights of each filter can be tuned to extract the best feature for a given dataset and task. **b**, Transfer functions such as those applied by the common rectified linear unit (ReLU) make possible the learning of nonlinear relationships. **c**, Pooling operations like max pooling downsample to produce spatially coarse feature maps[154]. Deep learning architectures often use iterative rounds of the three operations in **a**–**c** to produce low-dimensional representations of images. **d**, Dilations allow convolutional and pooling kernels to increase their spatial extent while keeping the number of parameters fixed[48,155]. When used correctly, dilations allow classification networks trained on image patches to be used for dense pixel-level prediction. **e**–**i**, Modern deep learning models make use of several architectural elements. **e**, Separable convolutions perform the convolution operation on each channel separately, which reduces the computing power while preserving accuracy[156,157]. **f**, Residual networks learn the identity mapping plus a small residual and enable the construction of very deep networks[68]. **g**, Dense networks allow each layer to see every prior layer[69], which improves error propagation and encourages both feature reuse and parameter efficiency[69,70]. **h**, Multi-resolution networks allow the classification layers to see both fine and coarse feature maps[86]. **i**, Through feature pyramids, object-detection models detect objects at distinct length scales[91,158]. **j**, A plot of the training error during training reveals the relationships among overfitting, underfitting, and model capacity. The tradeoff among these attributes determines which network architectures are suitable for a given task. "Underfitting" refers to models with insufficient representational power, and "overfitting" refers to models that have learned features specific to training data and hence generalize poorly to new, unseen data. Increased model capacity reduces underfitting but can increase the risk of overfitting. **k**–**m**, Numerous regularization techniques ensure that deep learning models learn general features from data. **k**, Batch normalization both regularizes networks and reduces the time needed for training[159]. It was initially created to mitigate covariate shift but was recently found to smooth the landscape of the loss function[160]. **l**, Dropout randomly turns off filters during training[161], which regularizes the network by forcing it to not overly rely on any one feature to make predictions. Batch normalization and dropout are typically not used together in the same model[162]. **m**, L2 regularization penalizes large weights and reduces overfitting.

programming is still required to adapt these frameworks to cellular imaging data, they substantially reduce the barrier to entry.

These frameworks have greatly simplified the training and deployment of deep learning models. Programming aspects are often reduced to finding a deep learning architecture that yields the best performance for a particular task. Recent strategies have incorporated a search throughout the space of potential architectures to identify the most effective model architecture[68–70] (Fig. 2). In our experience, the choice of architectural features often comes down to a tradeoff between overfitting and underfitting; this is also known as the bias–variance tradeoff in statistical modeling[71] (Fig. 2j). Overfitting occurs when models perform well on a training dataset but perform poorly on a withheld validation dataset, whereas underfitting occurs when models perform poorly on training data because they are unable to capture the variation in a training dataset. These two outcomes are often (but not always) two faces of the same coin, which we call model capacity: the representational power of a deep learning model. Models with high model capacity perform well on large datasets but are prone to overfitting. Models with lower model capacity may generalize better but are at risk for underfitting. Overfit models can be unreliable on unseen data, and underfit models have suboptimal performance[71]. Overfitting is a particularly important issue with small datasets. Techniques for mitigating overfitting are discussed in Box 2 and Fig. 2j–m. Recent years have seen architectural advances, such as residual networks[68], that have increased model capacity, which can lead to overfitting[72]. We recommend using model architectures with high model capacity only when enough data are present to avoid the model fragility that comes with overfitting. When data are limited, models with limited capacity and trained with regularization techniques are more likely to be robust. An alternative approach is to use transfer learning when adapting deep learning models to small datasets. This strategy often requires that pretrained models be modified to be compatible with the new dataset and task (changing the number of channels for an input image, etc.); users often are unable to make substantial changes to the model architecture. Despite these limitations, transfer learning can be very effective when data are limited. Whether training datasets are sufficiently large can be assessed with a cross-validation analysis. In this approach, one computes the degree of overfitting on models trained on varying fractions of the available training data. If the amount of data is sufficient, then the degree of overfitting should be stable even when the size of the training dataset is reduced. While overfitting is an important issue, other practical concerns come into play (Box 2), including optimization of hyperparameters such as the learning rate, choice of training algorithm, and issues surrounding class balancing.

Once trained, deep learning models must be deployed to process new data. While deployment can be achieved with scripts and Jupyter Notebooks[48], an alternative and arguably more effective approach is to use built-in deployment tools in several frameworks. For instance, both Tensorflow and MXNet have built-in deployment features that enable models to be deployed on a server and accessed through standard internet communication protocols[73], which allows the models to be shared beyond the original user. Associated software and hardware requirements mean that additional layers of software engineering beyond what is typical for academic software are often required for a deployment solution to be useful. First, containerization tools such as Docker[74] have been essential for the

Box 2 | Troubleshooting

While deep learning can solve many problems in biological image analysis, the creation of well-performing models often requires a substantial amount of troubleshooting. Here we provide guidance on navigating common issues that arise in the training of deep learning models.

Training performance. Very poor performance during training, defined as a classification error equal to or worse than random chance, can usually be traced to an issue with data or with training parameters. For small datasets, errors in training data can lead to poor performance. These errors often go unnoticed until the training data are manually inspected. Improper image normalization can lead to poor performance, as images can lose their informative features. Errors in the code that performs image augmentation and feeds data into the training pipeline can also yield poor performance. The learning rate is often the first parameter to be adjusted when the training data are free of errors and the performance is still very poor. Changing the model architecture to increase model capacity can also be an effective solution.

Overfitting. Deep learning models can learn complex relationships among data and annotations. As a result, there may be concern as to whether a deep learning model has learned something general that will work on real data or whether the model's learning is unique to the training dataset. This phenomenon is called overfitting and is generally measured as the difference between model accuracy for a training dataset and that for a validation dataset. The amount of overfitting that can be tolerated varies by task; several percentage points may be tolerable for segmentation but might cause an image classifier to misclassify important rare categories. Several regularization techniques exist to mitigate overfitting, but they often come at the expense of model capacity. Batch normalization[159] has strong regularization properties, as does dropout[161]. Typically, only one of these methods is used in a model, as performance can suffer if both are used simultaneously[162]. Increasing the strength of L2 regularization also mitigates overfitting, but at the expense of model capacity. Increasing the range of data-augmentation operations creates a more varied training dataset and hence more robust models[49]. Because overfitting often gets worse the longer that training proceeds, stopping training early can also be effective[178]. The choice of model architecture is especially important for small datasets. Architectures with large model capacities can be especially prone to overfitting on small datasets, although this tendency can be somewhat mitigated with transfer learning by pretraining on larger datasets[50]. Finally, the training algorithm used affects overfitting: recent work has demonstrated that models trained with stochastic gradient descent with momentum generalize better than models trained with other algorithms[175,176].

Class imbalance. Training datasets for classification tasks often have different numbers of examples for each label, which can lead to poorly performing models. As an example, consider a dataset in which 90% of the examples are label A and 10% are label B. The deep learning model may learn to predict everything as being class A and then report an overall classification accuracy of 90%. However, the reported accuracy is misleading, and the model is too inaccurate on class B to be used. There are several solutions to class imbalance. Resampling the training data to yield an identical number of elements in each class is one approach. This strategy can include downsampling to match the smallest class size, upsampling to match the largest class size, or both. Caution must be taken with downsampling when the least-represented class is much smaller (approximately tenfold) than all the other classes, as the diversity of the training data will be severely reduced. Another way to account for class imbalance is to introduce a class weight term into the loss function. This term, which is often taken as $(N_{\text{Total examples}}/N_{\text{Examples in class } i}) \times (1/N_{\text{Classes}})$ for each class i, multiplies each example data's individual contribution to the loss. This class weight term can be computed for the entire training dataset or for each minibatch on the fly.

Assessing performance. Following performance metrics during and after training is an important part of creating deep learning models. For performance assessment, the training data are often split into two portions, one for training and one for validation. If the performance on the validation dataset is used to modify training parameters, then it is possible to overfit the model to the validation data, even though these data were not used explicitly during training. Therefore, some researchers split their data into three parts: one for training, one for validation during training, and one for testing real-world performance. Our groups have achieved good success by reserving 10–20% of annotated data for testing.

Once the dataset is split, the remaining issue is choosing a performance metric. As seen from the class imbalance example, simple metrics such as accuracy can misrepresent performance. Useful metrics vary by problem type. For classification tasks, assessment of the accuracy for each individual class is more informative than an average across all classes. A confusion matrix[179] goes one step further, as it reveals the frequency of each type of misclassification. For segmentation tasks, both pixel-level (for example, the Dice and Jaccard indices[180]) and instance-level metrics (precision[181], recall[181], and mean average precision[182]) can be used to measure performance. Quantification of the rates at which specific errors occur, such as false splitting or merging of instance masks, has yielded important insights into the failure modes of deep learning models[52]. The appropriate metrics should be used on both training and validation datasets at the end of each training epoch.

Hyperparameter optimization. A hyperparameter is a parameter that is set before learning begins. Hyperparameters include L2 regularization strength, learning rate, initialization settings for parameters, and details of the deep learning architecture (number of layers, types of layers, number of filters in each layer, etc.). The optimization of hyperparameters, an essential part of the training process for deep learning models, usually consists of three phases: selection of an initial condition, selection of an optimization objective, and a search to find the best hyperparameters. While choosing initial conditions can be tricky, packages like Keras come with best-practice default settings for hyperparameters such as learning rate, L2 regularization strength, and weight initialization[59]. In practice, our groups often use the per-class training and validation accuracies as targets, with the goal of maximizing the training accuracies across classes while minimizing the gap between training and validation accuracies to minimize overfitting. Finally, there are several strategies for searching hyperparameter space. Grid searches in which the learning rate and regularization strength are tuned are often effective. Model architecture can also be modified; our groups often use the tradeoff between model capacity and overfitting as a guide to determine the changes that should be made. Our prior work has shown that it is important to match a model's receptive field size with the relevant feature size in order to produce a well-performing model for biological images[48]. The Python package Talos is a convenient tool for Keras[59] users that helps to automate hyperparameter optimization through grid searches[183].