Neural Network

Zhongjian Lin
University of Georgia

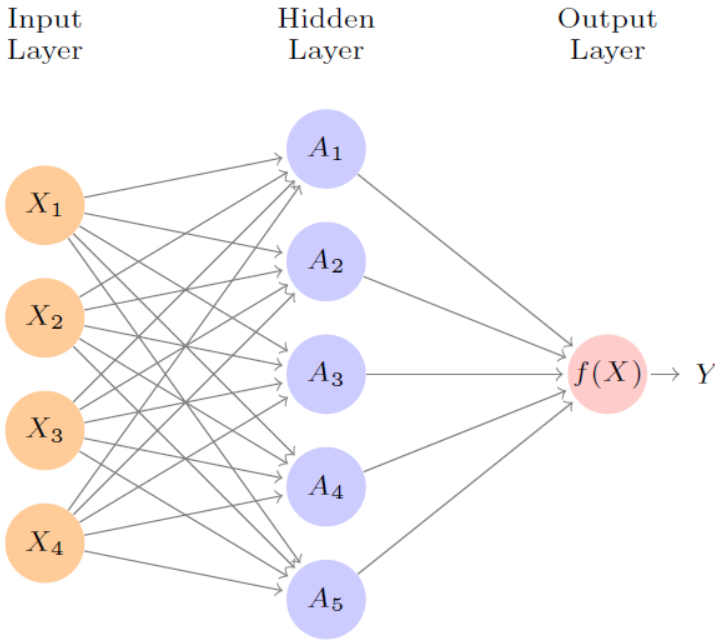
October 23, 2024

Deep Learning

Deep learning is a very active area of research in the machine learning learning and artificial intelligence communities. The cornerstone of deep learning is the *neural network*. Neural networks rose to fame in the late 1980s. There was a lot of excitement and a certain amount of hype associated with this approach, and they were the impetus for the popular *Neural Information Processing Systems* meetings (NeurIPS). This was followed by a synthesis stage, where the properties of neural networks were analyzed by machine learners, mathematicians and statisticians; algorithms were improved, and the methodology stabilized. Then along came SVMs, boosting, and random forests, and neural networks fell somewhat from favor. Part of the reason was that neural networks required a lot of tinkering, while the new methods were more automatic. Also, on many problems the new methods outperformed poorly-trained neural networks.
Neural networks resurfaced after 2010 with the new name deep learning, with new architectures, additional bells and whistles, and a string of success stories on some niche problems such as image and video classification, speech and text modeling. Many in the field believe that the major reason for these successes is the availability of ever-larger training datasets, made possible by the wide-scale use of digitization in science and industry.

Single Layer Neural Network

A neural network takes an input vector of p variables $X = (X_1, X_2, \cdots, X_p)$ and builds a nonlinear function $f(X)$ to predict the response Y. We have built nonlinear prediction models using trees, boosting and generalized additive models. What distinguishes neural networks from these methods is the particular structure of the model. Following figure shows a simple feed-forward neural network for modeling a quantitative response using $p = 4$ predictors. In the terminology of neural networks, the four features $X_1, \cdots, X_4$ make up the units in the *input layer*. The arrows indicate that each of the inputs from the input layer feeds into each of the K *hidden units*.



Neural Network

The neural network model has the form

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X) = \beta_0 + \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j). \tag{1}$$

It is built up here in two steps. First the K activations $A_k, k = 1, \cdots, K$, in the hidden layer are computed as functions of the input features $X_1, \cdots, X_p$,

$$A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j),$$

where $g(z)$ is a nonlinear *activation function* that is specified in advance. These K activations from the hidden layer then feed into the output layer, resulting in

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k.$$

The name *neural network* originally derived from thinking of these hidden units as analogous to neurons in the brain

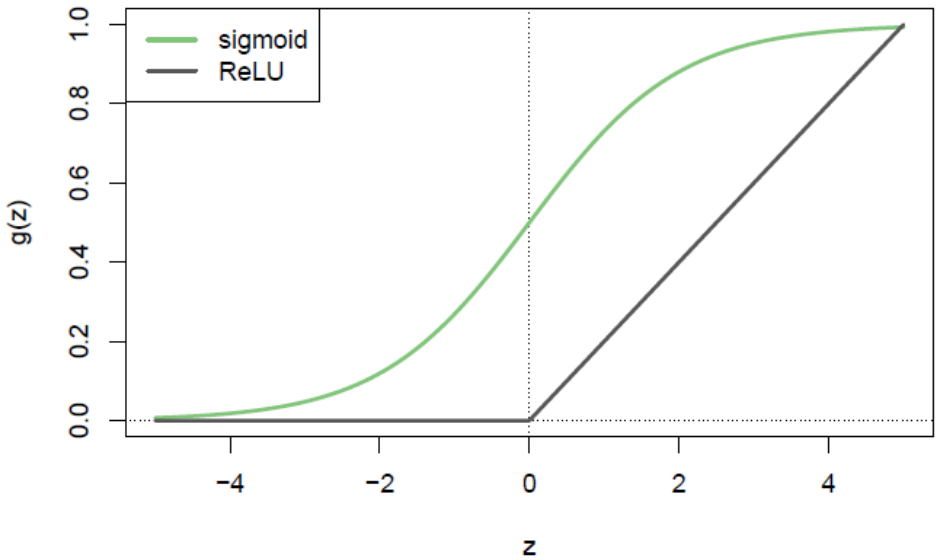
Activation Function

In the early instances of neural networks, the *sigmoid* activation function was favored

$$g(z) = \frac{e^z}{1 + e^z},$$

which is the same function used in logistic regression to convert a linear function into probabilities between zero and one. The preferred choice in modern neural networks is the ReLU (rectified linear unit) activation function, which takes the form

$$g(z) = (z)_+ = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases}$$



Nonlinearity

The nonlinearity in the activation function $g(\cdot)$ is essential, since without it the model f(X) would collapse into a simple linear model in $X_1, \cdots, X_p$. Moreover, having a nonlinear activation function allows the model to capture complex nonlinearities and interaction effects.
Fitting a neural network requires estimating the unknown parameters in Equation (1). For a quantitative response, typically squared-error loss is used, so that the parameters are chosen to minimize

$$\sum_{i=1}^{n} [y_i - f(x_i)]^2$$

Multilayer Neural Networks

Modern neural networks typically have more than one hidden layer, and often many units per layer. In theory a single hidden layer with a large number of units has the ability to approximate most functions. However, the learning task of discovering a good solution is made much easier with multiple layers each of modest size.

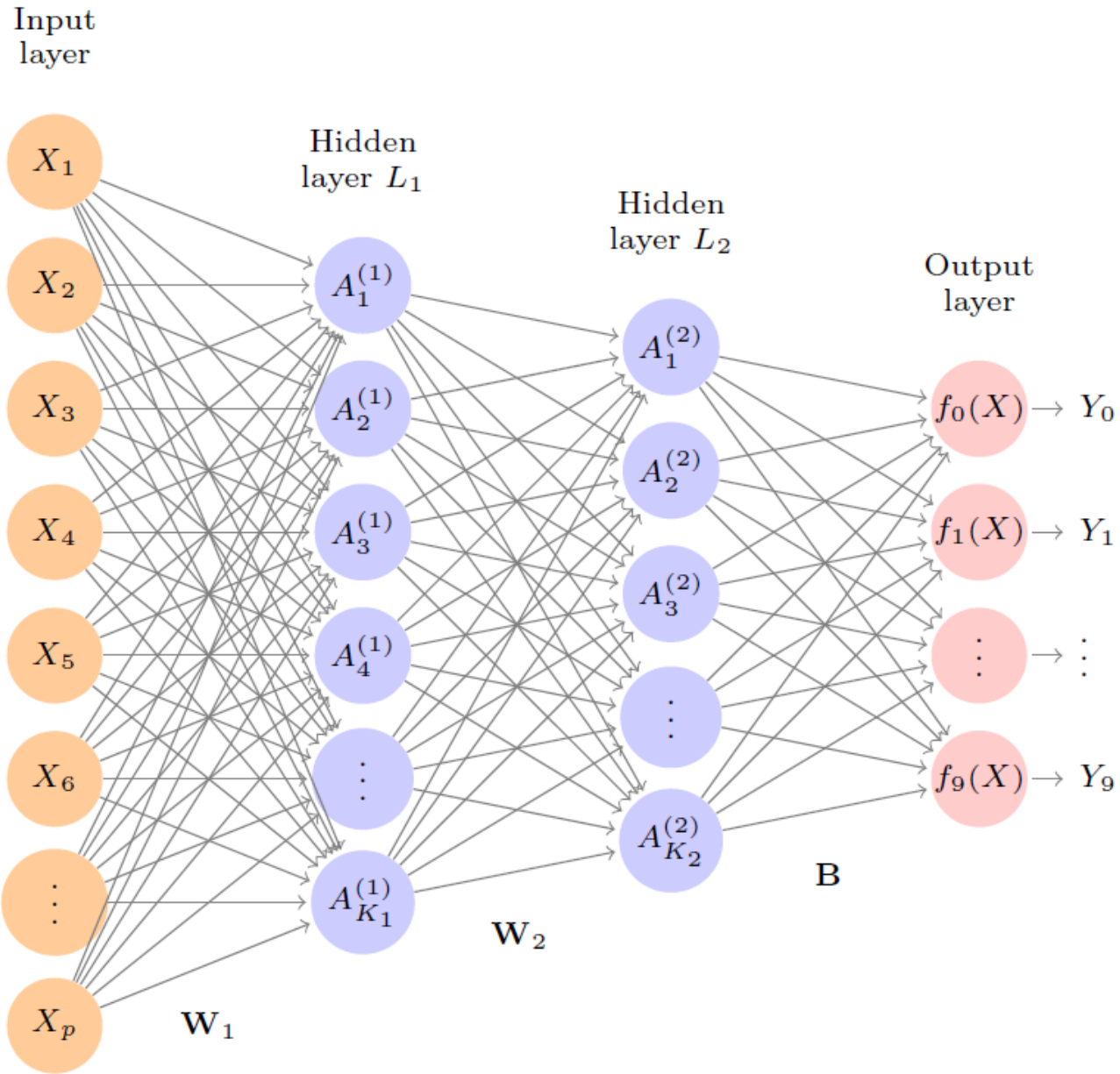
Handwritten Digits

The idea is to build a model to classify the images into their correct digit class 0–9. Every image has $p = 28 \times 28 = 784$ pixels, each of which is an eight-bit grayscale value between 0 and 255 representing the relative amount of the written digit in that tiny square.

These pixels are stored in the input vector X (in, say, column order). The output is the class label, represented by a vector $Y = (Y_0, Y_1, \cdots, Y_9)$ of 10 dummy variables, with a one in the position corresponding to the label, and zeros elsewhere. There are 60,000 training images, and 10,000 test images. On a historical note, digit recognition problems were the catalyst that accelerated the development of neural network technology in the late 1980s at AT&T Bell Laboratories and elsewhere. Pattern recognition tasks of this kind are relatively simple for humans. Our visual system occupies a large fraction of our brains, and good recognition is an evolutionary force for survival. These tasks are not so simple for machines, and it has taken more than 30 years to refine the neural-network architectures to match human performance.

- It has two hidden layers L_1 (256 units) and L_2 (128 units) rather than one.
- It has ten output variables, rather than one. Thus it is a *multi-task learning*.
- The loss function used for training the network is tailored for the multiclass classification task.

Multilayer Neural Network



We have the first hidden layer as

$$A_k^{(1)} = h_k^{(1)}(X) = g(w_{k0}^{(1)} + \sum_{j=1^p} w_{kj}^{(1)} X_j, k = 1, \cdots, K_1,$$

and the second hidden layer as

$$A_l^{(2)} = h_l^{(2)}(X) = g(w_{l0}^{(2)} + \sum_{k=1^{K_1}} w_{lk}^{(2)} A_k^{(1)}, l = 1, \cdots, K_2.$$

We now get to the output layer, where we now have ten responses rather than one. The first step is to compute ten different linear models

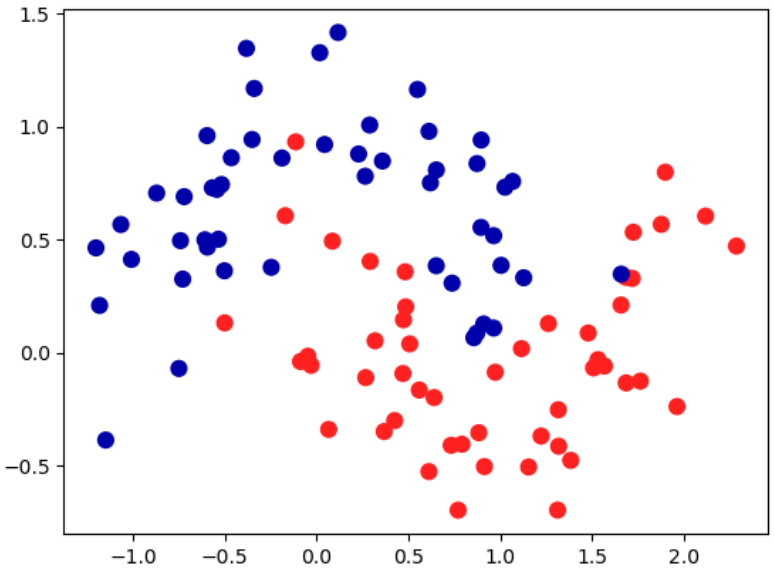
$$Z_m = \beta_{m0} + \sum_{l=1}^{K_2} \beta_{ml} A_l^{(2)}, m = 0, 1, \cdots, 9.$$

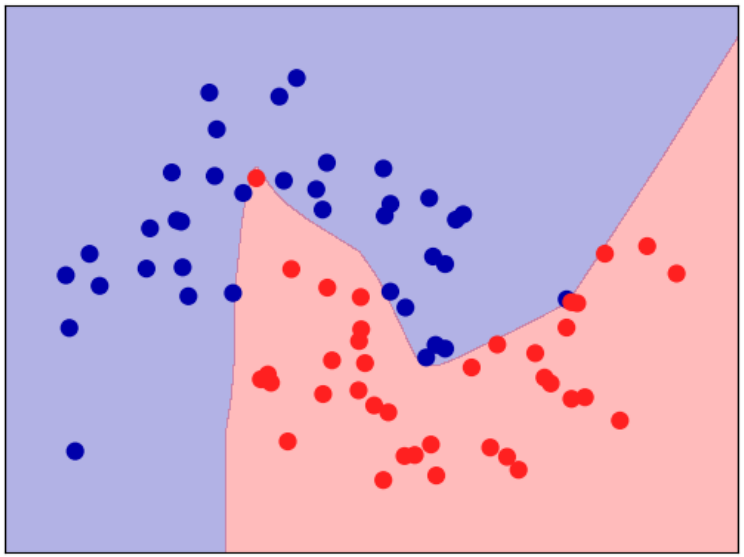
We further estimate the class probabilities $f_m(X) = P(Y = m|X) = \frac{e^{Z_m}}{\sum_{l=0}^{9} e^{Z_l}}$.
The estimation of parameters is to minimize the negative multinomial log-likelihood

$$-\sum_{i=1}^{n} \sum_{m=0}^{9} y_{im} \log[f_m(x_i)].$$

Tuning Neural Networks

Let's look into the workings of the MLP by applying the MLPClassifier to the two_moons dataset.



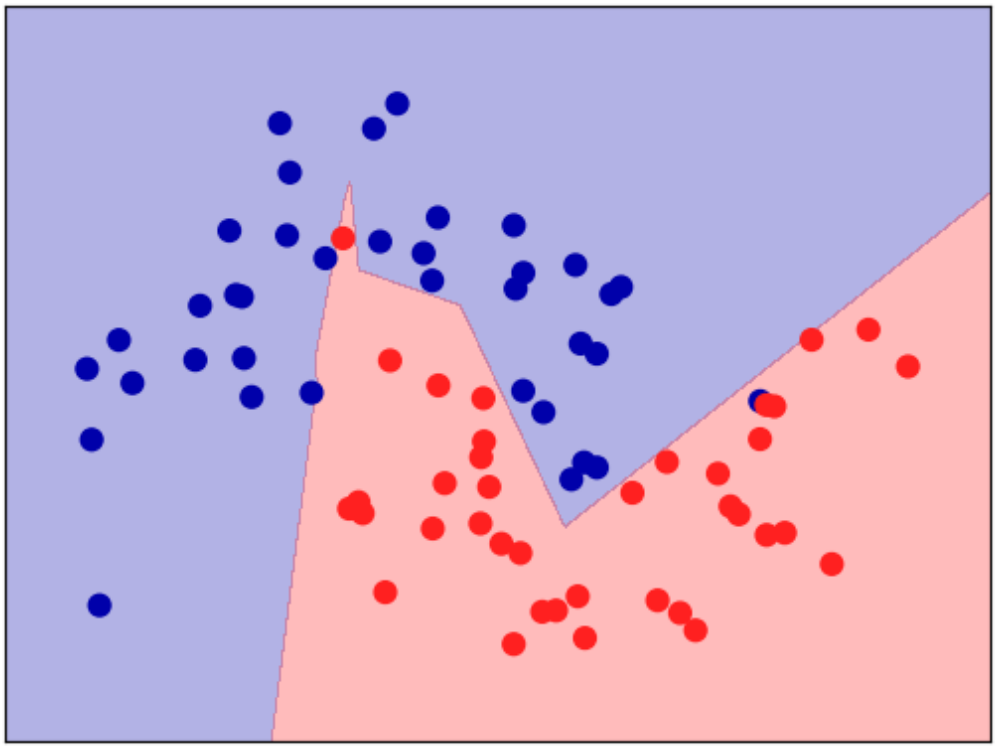


As you can see, the neural network learned a very nonlinear but relatively smooth decision boundary.

Tuning Neural Networks

By default, the MLP uses 100 hidden nodes, which is quite a lot for this small dataset. We can reduce the number (which reduces the complexity of the model) and still get a good result:

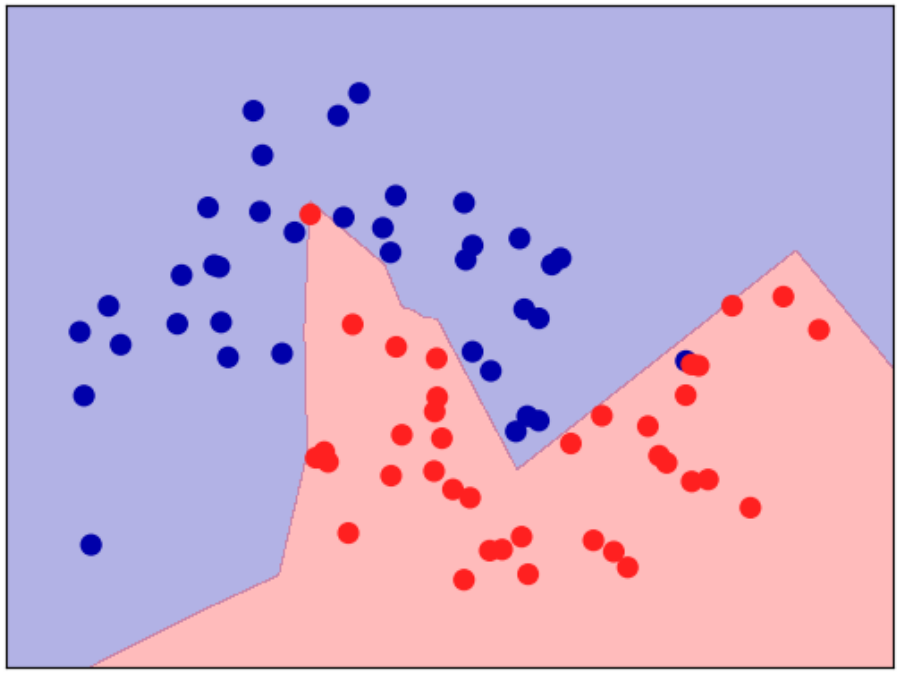
```
mlp = MLPClassifier(solver='lbfgs', random_state=0, hidden_layer_sizes=[10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=60, cmap=mglearn.cm2)
```



Tuning Neural Networks

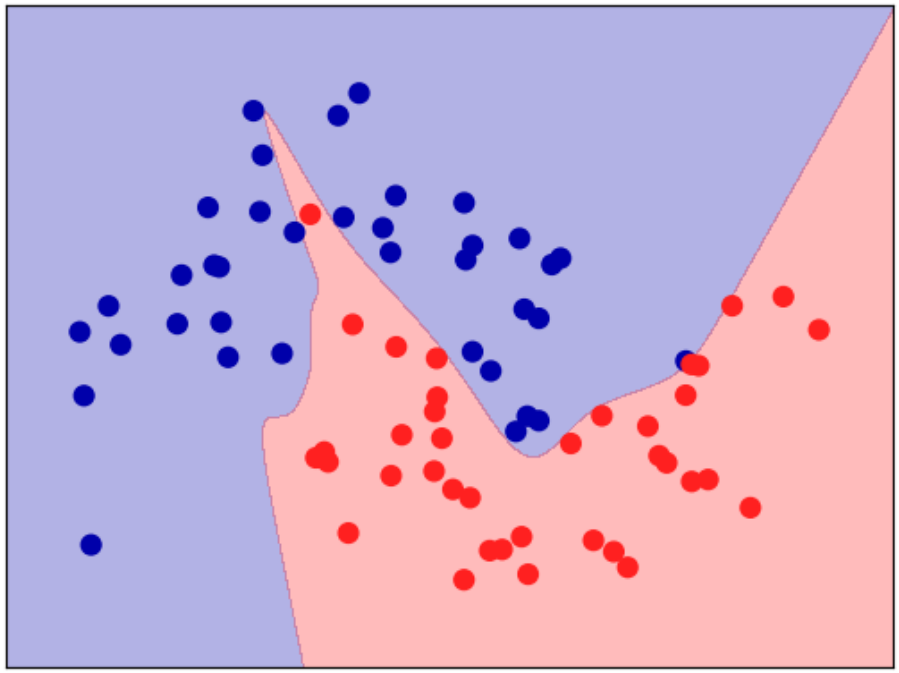
With a single hidden layer, this means the decision function will be made up of 10 straight line segments. If we want a smoother decision boundary, we could either add more hidden units, add second hidden layer, or use the "tanh" nonlinearity

```
mlp = MLPClassifier(solver='lbfgs', random_state=0, hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=60, cmap=mglearn.cm2)
```



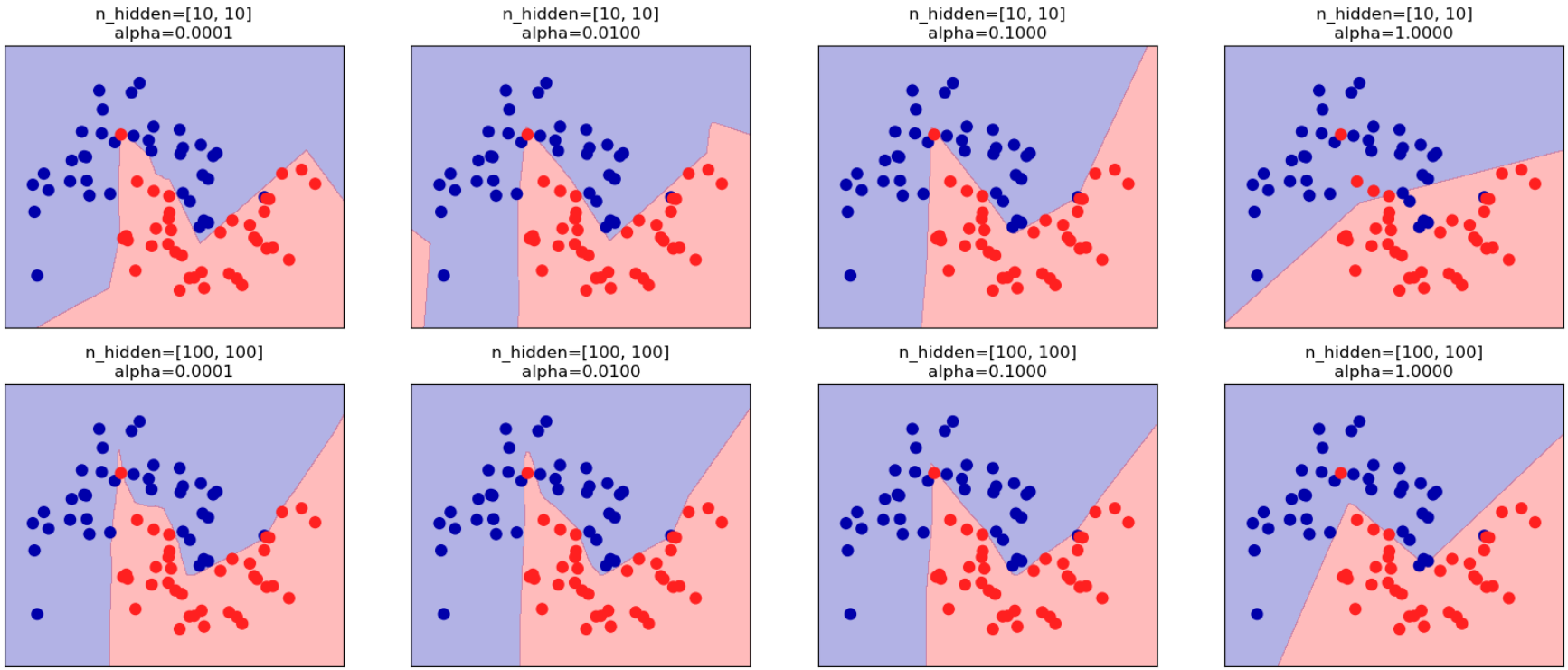
Tuning Neural Networks

```
# using two hidden layers, with 10 units each, now with tanh nonlinearity.
mlp = MLPClassifier(solver='lbfgs', activation='tanh',
  random_state=0, hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=60, cmap=mglearn.cm2)
```



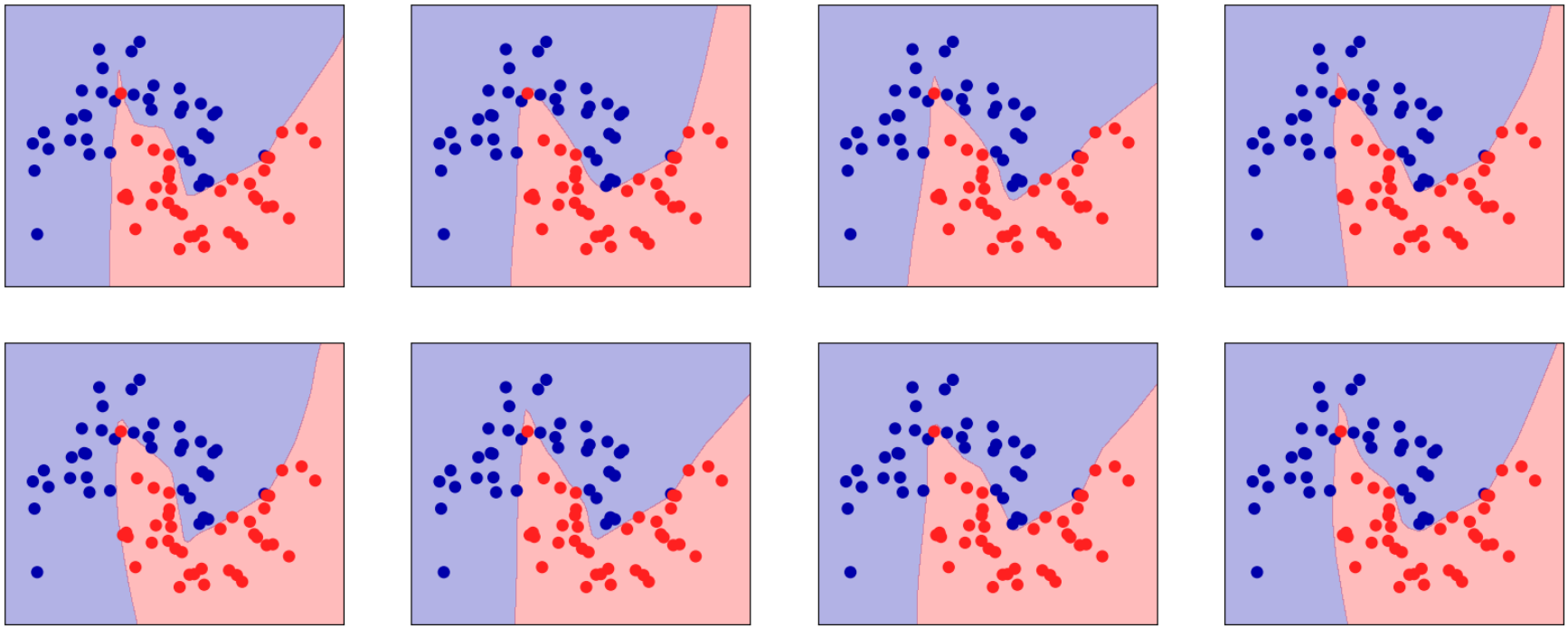
Tuning Neural Networks

Finally, we can also control the complexity of a neural network by using an "l2" penalty to shrink the weights towards zero, as we did in ridge regression and the linear classifiers. The parameter for this in the MLPClassifier is alpha (as in the linear regression models), and is set to a very low value (little regularization) by default.



Tuning Neural Networks

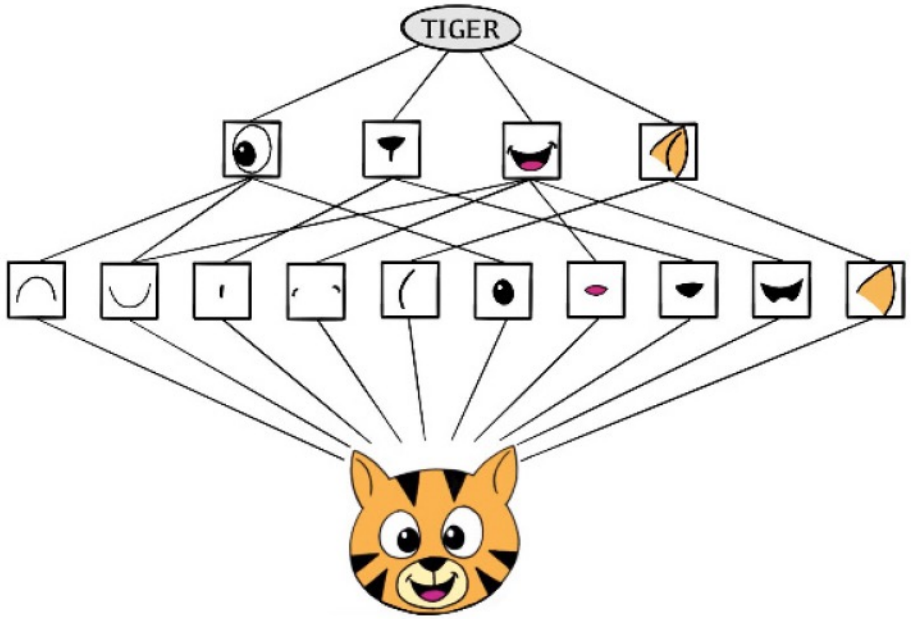
There are many ways to control the complexity of a neural network: the number of hidden layers, the number of units in each hidden layer, and the regularization (alpha), etc. An important property of neural networks is that their weights are set randomly before learning is started, and this random initialization affects the model that is learned. That means that even when using exactly the same parameters, we can obtain very different models when using different random seeds. If the networks are large, and their complexity is chosen properly, this should not affect accuracy too much.



Convolutional Neural Networks

Neural networks rebounded around 2010 with big successes in image classification. Around that time, massive databases of labeled images were being accumulated, with ever-increasing numbers of classes. Figure 10.5 shows 75 images drawn from the CIFAR100 database.4 This database consists of 60,000 images labeled according to 20 superclasses (e.g. aquatic mammals), with five classes per superclass (beaver, dolphin, otter, seal, whale). Each image has a resolution of 32×32 pixels, with three eight-bit numbers per pixel representing red, green and blue. The numbers for each image are organized in a three-dimensional array called a feature map. The first two axes are spatial (both are 32-dimensional), and the third is the channel axis,5 representing the three colors.

A special family of convolutional neural networks (CNNs) has evolved for convolutional neural networks classifying images such as these, and has shown spectacular success on a wide range of problems. CNNs mimic to some degree how humans classify images, by recognizing specific features or patterns anywhere in the image that distinguish each particular object class.

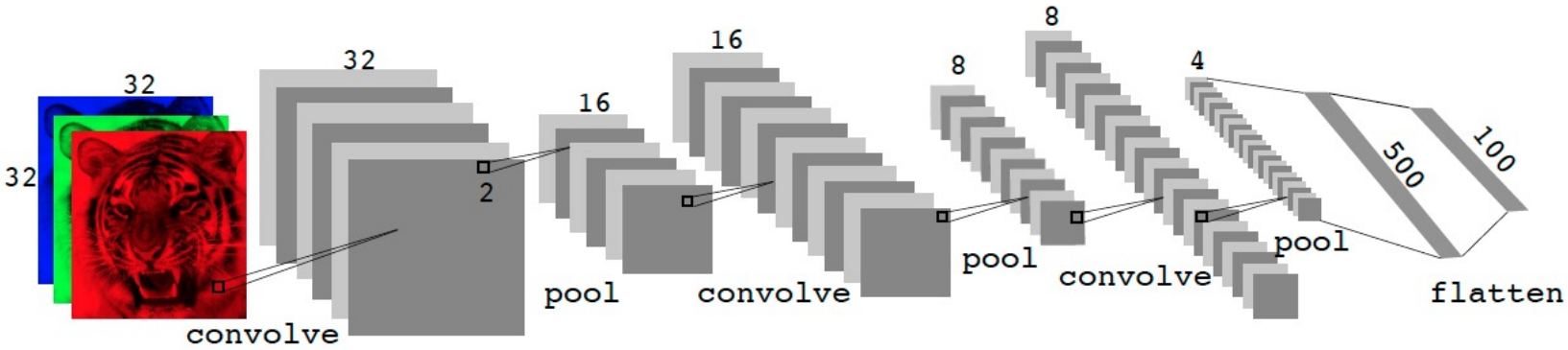


The network first identifies low-level features in the input image, such as small edges, patches of color, and the like. These low-level features are then combined to form higher-level features, such as parts of ears, eyes, and so on. Eventually, the presence or absence of these higher-level features contributes to the probability of any given output class.

The convolutional neural network combines two specialized types of hidden layers, called convolution layers and pooling layers.

- Convolution layers search for instances of small patterns in the image.
- Pooling layers downsample these to select a prominent subset.

In order to achieve state-of-the-art results, contemporary neural network architectures make use of many convolution and pooling layers.



Convolution Layers

A convolution layer is made up of a large number of convolution filters, each of which is a template that determines whether a particular local feature is present in an image. A convolution filter relies on a very simple operation, called a convolution, which basically amounts to repeatedly multiplying matrix elements and then adding the results. consider a very simple example of a 4×3 image:

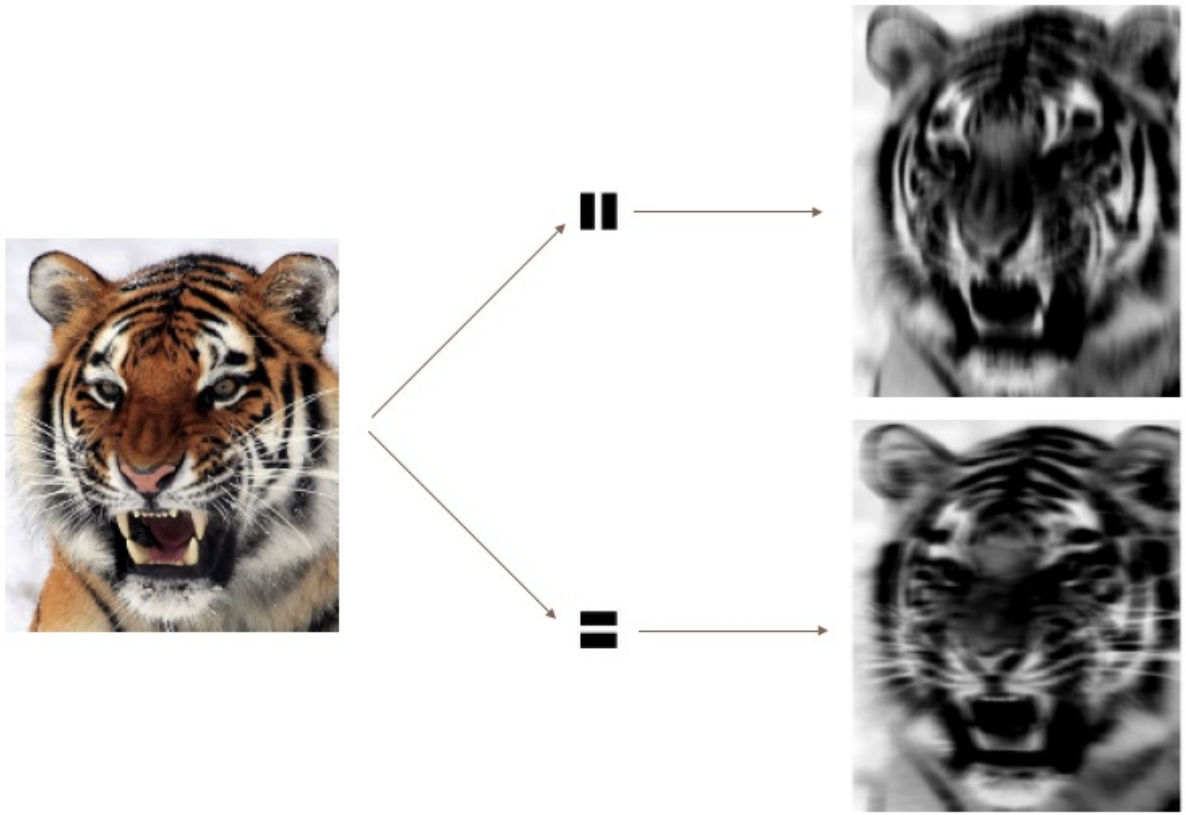
$$\text{Original Image}= \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}$$

Consider a 2×2 filter of the form $\text{Convolution filter}= \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$. We have

$$\text{Convolved Image}= \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{bmatrix}$$

If a 2×2 submatrix of the original image resembles the convolution filter, then it will have a large value in the convolved image; otherwise, it will have a small value. Thus, the convolved image highlights regions of the original image that resemble the convolution filter.

Convolution Layers



We see that the horizontal stripe filter picks out horizontal stripes and edges in the original image, whereas the vertical stripe filter picks out vertical stripes and edges in the original image.

Pooling Layers

A pooling layer provides a way to condense a large image into a smaller summary image. While there are a number of possible ways to perform pooling, the max pooling operation summarizes each non-overlapping $\times 2$ block of pixels in an image using the maximum value in the block. This reduces the size of the image by a factor of two in each direction, and it also provides some location invariance: i.e. as long as there is a large value in one of the four pixels in the block, the whole block registers as a large value in the reduced image.

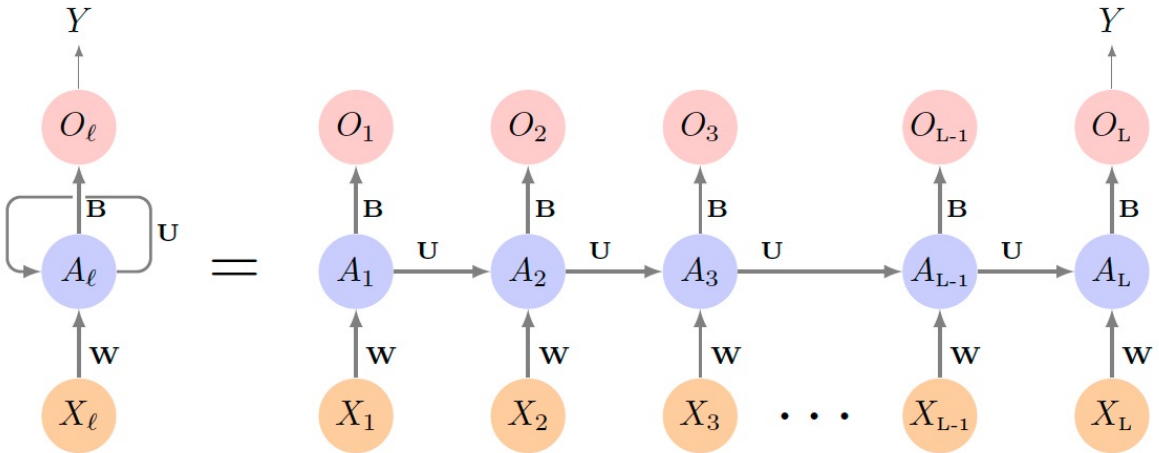
$$\text{max pool} \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}$$

Recurrent Neural Networks

Many data sources are sequential in nature, and call for special treatment when building predictive models. Examples include:

- Documents such as book and movie reviews, newspaper articles, and tweets. The sequence and relative positions of words in a document capture the narrative, theme and tone, and can be exploited in tasks such as topic classification, sentiment analysis, and language translation.
- Time series of temperature, rainfall, wind speed, air quality, and so on. We may want to forecast the weather several days ahead, or climate several decades ahead.
- Financial time series, where we track market indices, trading volumes, stock and bond prices, and exchange rates. Here prediction is often difficult, but as we will see, certain indices can be predicted with reasonable accuracy.
- Recorded speech, musical recordings, and other sound recordings. We may want to give a text transcription of a speech, or perhaps a language translation. We may want to assess the quality of a piece of music, or assign certain attributes.
- Handwriting, such as doctor's notes, and handwritten digits such as zip codes. Here we want to turn the handwriting into digital text, or read the digits (optical character recognition).

In a recurrent neural network (RNN), the input object X is a sequence. Consider a corpus of documents, such as the collection of IMDb movie reviews. Each document can be represented as a sequence of L words, so $X = \{X_1, X_2, \cdots, X_L\}$, where each X_l represents a word. The order of the words, and closeness of certain words in a sentence, convey semantic meaning. RNNs are designed to accommodate and take advantage of the sequential nature of such input objects, much like convolutional neural networks accommodate the spatial structure of image inputs. The output Y can also be a sequence (such as in language translation), but often is a scalar, like the binary sentiment label of a movie review document.



When to use NNs

The performance of deep learning has been rather impressive. It nailed the digit classification problem, and deep CNNs have really revolutionized image classification. We see daily reports of new success stories for deep learning. Many of these are related to image classification tasks, such as machine diagnosis of mammograms or digital X-ray images, ophthalmology eye scans, annotations of MRI scans, and so on. Likewise there are numerous successes of RNNs in speech and language translation, forecasting, and document modeling. Should we discard all our older tools, and use deep learning on every problem with data?
This is a regression problem, where the goal is to predict the Salary of a baseball player in 1987 using his performance statistics from 1986. We have 263 players and 19 variables. We randomly split the data into a training set of 176 players (two thirds), and a test set of 87 players (one third). We used three methods for fitting a regression model to these data.

- A linear model.
- The same linear model was fit with lasso regularization.
- A neural network with one hidden layer consisting of 64 ReLU units

Model Performance

Model	# Parameters	Mean Abs. Error	Test Set R^2
Linear Regression	20	254.7	0.56
Lasso	12	252.3	0.51
Neural Network	1345	257.4	0.54

We see similar performance for all three models. We spent a fair bit of time fiddling with the configuration parameters of the neural network to achieve these results. It is possible that if we were to spend more time, and got the form and amount of regularization just right, that we might be able to match or even outperform linear regression and the lasso.
But with great ease we obtained linear models that work well. Linear models are much easier to present and understand than the neural network, which is essentially a black box.

Occam's razor principle

Occam's razor principle

When faced with several methods that give roughly equivalent performance, pick the simplest.

We have a number of very powerful tools at our disposal, including *neural networks, random forests and boosting, support vector machines*, to name a few. And then we have *linear models, and simple variants of these*. When faced with new data modeling and prediction problems, it's tempting to always go for the trendy new methods. Often they give extremely impressive results, especially when the datasets are very large and can support the fitting of high-dimensional nonlinear models. However, if we can produce models with the simpler tools that perform as well, they are likely to be easier to fit and understand, and potentially less fragile than the more complex approaches. Wherever possible, it makes sense to try the simpler models as well, and then make a choice based on the performance/complexity tradeoff. Typically we expect deep learning to be an attractive choice when the sample size of the training set is extremely large, and when interpretability of the model is not a high priority.