Neural Networks

One of the generalized linear models we discussed in the last chapter was logistic regression. Logistic regression develops a model with a decision boundary that is, well, linear. Consider the two two-dimensional classification data sets in Figure 7, where in the first, logistic regression clearly separates the two classes, and in the second, it fails. Would we be able to modify the feature space in the second data set so that logistic regression would succeed?

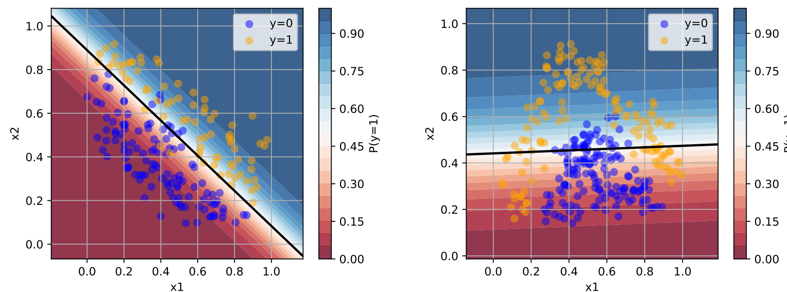

Figure 7: A linearly and non-linearly separable data set and the result of modelling with logistic regression.

Clearly, even in the second data set from Figure 7, there are patterns that would benefit from linear modelling. When observed by a human, we would probably break the data with two lines, and then declare an enclosed region to be that for a target class and everything else to be the remaining class. But that would require the inference of two separate decision boundaries, and then something to join them into a resulting classifier. Perhaps a combination of two logistic regressions up front and then a single logistic regression to join their output?

Combinations of logistic regressions

Let us dive into this idea of combining logistic regressions. We can construct a system, an architecture, with two logistic regressions at the input layer, marking the outputs as a_{00} and a_{01}, where the first index is the number of the layer and the second the number of

A disclaimer: combining logistic regressions should here be understood as a conceptual stepping stone: in modern neural networks, the internal computational units typically use more general activation functions, and we will discuss this a bit later.

computational units (logistic regressions) at that layer. Since we have two layers, we could mark the activation at the second layer as a_{10}, but since this is already our output $P(y = 1)$, we mark it as y in the graph. The architecture of such a system is sketched in Figure 8.

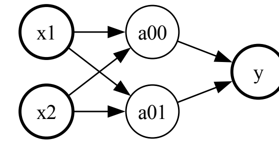

Figure 8: Combination of three logistic regressions.

Just a reminder: to compute the activations, we first compute the weighted sum of the inputs, $z = w \cdot x + b$, and then transform this sum into probabilities, like in the case of logistic regression with $a = \sigma(z)$. For logistic regression, we used this transformation to map a weighted sum to a probability, but essentially, as we deal with probabilities only at the output, we could use some other non-linear transformation functions in all other computational nodes of our emerging architecture.

There is of course a reason why transformation functions should be non-linear. If they were linear, they would make no sense as combinations of linear functions are still linear and then we would be back to a single computational node with logistic regression, which would fail to create non-linear decision boundaries.

This proposed architecture of our model solves our classification problem nicely, as shown in Figure 9. But does it internally construct the decision boundaries as we would have expected? Not exactly. The two internal computational nodes have their "decision boundaries" at right angles (Figure 10), but are misaligned.

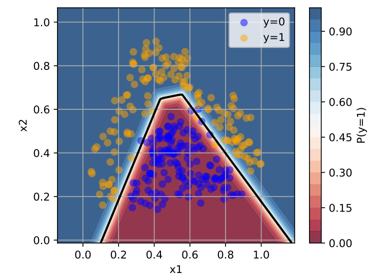

Figure 9: Decision boundary and probability contours for a v-shaped classification data set.

We were of course wrong to expect that the "decision boundaries" of the two logistic regressions in the first layer of our architecture would clearly separate locally the points of different classes, as the next computational unit of the next level weights the distances to the lines and sums them up, and does not say, for instance, if the two distances are both positive (as we, human, would do). Interestingly, and as expected (even after some thinking), these two lines point in the right direction and are offset just right to be used in the final logistic regression. If needed, we could use their weights (but not their position) for some further interpretation.

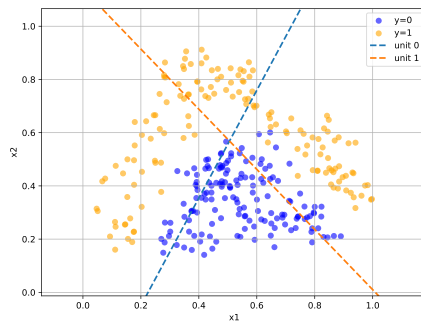

Figure 10: "Decision boundary" by two internal logistic regressions from the model in Figure 8.

We should not stop here. For instance, we could continue with even more complex and non-linear classification problems, like the one depicted in Fig. 11. For this somewhat more complex data, we have designed two alternative model architectures, again, in principle, just combinations of logistic regressions, that can both find

appropriate (and expected) decision boundaries.

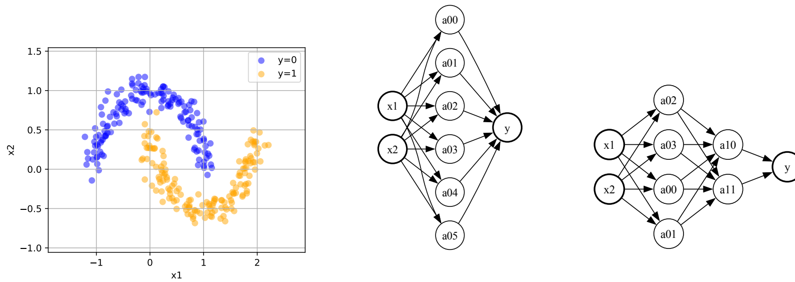

Figure 11: Somehow more complex data sets and two variants of model architecture that could perhaps deal with it.

Notice that in the second architecture from Figure 11, we have introduced an additional layer of computational units. Intentionally, this layer includes only two computational units, with their activations being amenable to visualization. Note that these two activations are being sent to our final unit, a logistic regression that outputs the target class probability. Logistic regressions have linear decision boundaries, and if this architecture "works", the points in the plot of two activations at the penultimate layer of our architecture should have linear separation in terms of the class (Fig. 12). And it does!

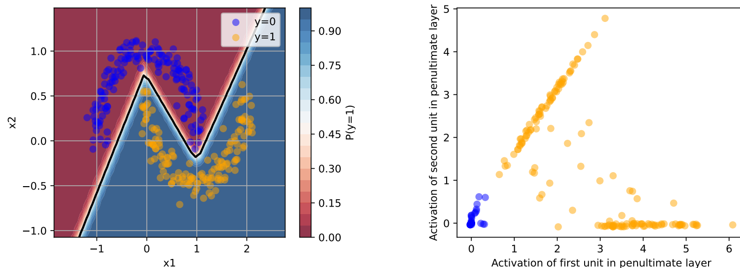

Figure 12: Decision boundary of the two-layered model and activation space of the penultimate layer, conveniently for visual purposes consisting of two computational units.

The combinations of logistic regressions we have explored above have been historically labeled as *neural networks*. This is a historical term, since they were not really designed as combinations of logistic regressions, but rather as computational units that would mimic neurons in our brain. More on history a bit later; it is now time to formalize this concept and write some code to implement it.

Important observation

Before we dive deeper into formalism and coding, one important observation: above, we have used logistic regression at the very end of our model architecture. The final model was therefore a generalized linear model. Everything before that computational unit was used to

prepare the data for that model, that is, to fit the data so that it can be modeled with a GLM.

From this perspective, the earlier layers of a neural network act as a learned feature transformation, while the final layer behaves like a generalized linear model applied to those features. Unlike manual preprocessing, however, this transformation is learned together with the final model in a single optimization procedure. In that sense, the network adapts its internal representation specifically to make the final prediction task easier.

Formal definition

In its standard feedforward form, a neural network is a parametric function $f_\theta : \mathbb{R}^d \to \mathbb{R}^k$ defined as a composition of affine transformations and element-wise non-linear activation functions. The network is organized into layers: an input layer, one or more *hidden layers*, and an output layer. Each hidden layer consists of *hidden units* (or nodes) whose activations are not directly observed, but instead serve as intermediate representations of the input.

Other neural network architectures, including convolutional, recurrent, and transformer-based, extend this basic form, but for now it suffices to start with the canonical form and extend it later.

For an input $x \in \mathbb{R}^d$, the network with L layers computes a sequence of activations

$$a^{(0)} = x, \qquad z^{(\ell)} = W^{(\ell)} a^{(\ell-1)} + b^{(\ell)}, \qquad a^{(\ell)} = \sigma^{(\ell)}\left(z^{(\ell)}\right), \quad \ell = 1, \ldots, L,$$

where layers $\ell = 1, \ldots, L-1$ are hidden layers, and $\ell = L$ is the output layer. Here, $W^{(\ell)}$ and $b^{(\ell)}$ are the parameters (weights and biases) of layer ℓ, $\sigma^{(\ell)}$ are non-linear activation functions, and $\theta = \{W^{(\ell)}, b^{(\ell)}\}_{\ell=1}^{L}$. The final output $f_\theta(x) = a^{(L)}$ is interpreted according to the task (e.g., probabilities in classification or real values in regression).

Just like with every other model that we have dealt with so far, the parameters θ are learned from data by minimizing some loss function $\mathcal{L}(f_\theta(x), y)$ over a training set, where the loss is chosen according to the task at hand. It can be any of those we have discussed in the context of generalized linear models, or extended to more complex losses when dealing with more structured data.

Implementation

We can represent the computation defined by a neural network as a computational graph and then use it to compute gradients of the model parameters. Everything else we have introduced so far applies: in the implementation we will use the same `train` function as before, which supports batch learning. We can also use regularization, just as we did to smooth linear regression models (see the previous

lectures). By "implementation" we therefore refer only to the construction of the computational graph corresponding to a particular neural network architecture.

Let us start with a note on activation functions. Above, we have used the logistic function, but in the internal nodes of neural networks we more often use simpler non-linearities that allow for more efficient training and better gradient propagation; hence we will follow this recommendation in our implementation.

The ReLU function (Rectified Linear Unit) is defined as

$$\text{ReLU}(z) = \max(0, z),$$

that is, it returns the input if it is positive and zero otherwise. It is a simple yet very effective non-linear function that enables faster learning and alleviates the vanishing gradient problem. This happens when gradients shrink as they are propagated backward through the network, which is common with sigmoid or tanh activations. As a result, earlier layers receive almost no learning signal and update very slowly. The ReLU function mitigates this issue by having a constant derivative (equal to 1) for positive inputs, which allows gradients to propagate more effectively and leads to faster and more stable training. To support ReLU, we extend our automatic differentiation library (the `Value` class) with its forward computation and backward gradient propagation:

```
def relu(self):
    out = Value(0 if self.data < 0 else self.data, (self,), 'ReLU')

    def _backward():
        self.grad += (out.data > 0) * out.grad
    out._backward = _backward

    return out
```

We construct a neural network from three conceptual classes: a neuron (a computational unit), a layer of the neural network consisting of neurons at the same level, and the neural network itself, which connects layers together.

```
class Neuron:

    def __init__(self, nin, activation='relu'):
        self.w = [Value(random.uniform(-1,1)) for _ in range(nin)]
        self.b = Value(0)
        self.activation = activation
        self.out = None
```

```
    def __call__(self, x):
        act = sum((wi*xi for wi,xi in zip(self.w, x)), self.b)
        out = act.sigmoid() \
           if self.activation == 'sigmoid' else act.relu()
        self.out = out
        return out

    def parameters(self):
        return self.w + [self.b]

    def __repr__(self):
        return f"Neuron({len(self.w)})"
```

The `Neuron` class implements the functionality of a single neuron, which takes `nin` inputs, computes their weighted sum with an added bias, and passes the result through an activation function (ReLU by default, but it can also be sigmoid). The function `parameters()` returns the list of the neuron's parameters (its weights and bias).

```
class Layer:

    def __init__(self, nin, nout, **kwargs):
        self.neurons = [Neuron(nin, **kwargs) for _ in range(nout)]

    def __call__(self, x):
        out = [n(x) for n in self.neurons]
        return out[0] if len(out) == 1 else out

    def parameters(self):
        return [p for n in self.neurons for p in n.parameters()]

    def __repr__(self):
        return f"Layer of [{', '.join(str(n) for n in self.neurons)}]"
```

The `Layer` class represents a single level of the network, that is, a layer of neurons that all receive the same input. The computation in `__call__` invokes each neuron, which independently computes its activation. If the layer contains only a single neuron (i.e., it is the output neuron), the call returns a scalar; otherwise, it returns a list of activations. The parameters of the layer are obtained by collecting the parameters of all its neurons.

```
class NeuralNetwork:

    def __init__(self, nin, nouts):
        sz = [nin] + nouts
        self.layers = [Layer(sz[i], sz[i+1], \
            activation="sigmoid" if i == len(nouts)-1 else "relu")
            for i in range(len(nouts))]
```

```
    def __call__(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

    def parameters(self):
        return [p for layer in self.layers
            for p in layer.parameters()]

    def loss(self, X, ys):
        eps = 1e-8
        yhats = [self(x) for x in X]
        return -sum(y * (yhat + eps).log() + \
           (1-y) * (1-yhat + eps).log() \
           for y, yhat in zip(ys, yhats)) / len(ys)

    def __repr__(self):
        return f"NN of [{', '.join(str(layer)
            for layer in self.layers)}]"
```

The `NeuralNetwork` class constructs the network by sequentially connecting its layers. The user specifies the number of inputs and, in a list, the number of neurons in each layer. All neurons in hidden layers use ReLU, while the final layer uses the sigmoid function to compute class probabilities. The `__call__()` function elegantly implements the forward pass: each layer takes the activations from the previous layer and passes its output to the next, until the final output layer produces the prediction. The network collects its parameters from the parameters of its layers, which in turn collect them from their neurons. Recall that these parameters will be updated during gradient descent according to their gradients.

In the above implementation we have included a `loss` function for binary classification, corresponding to the negative log-likelihood (cross-entropy) loss used in logistic regression. The reader can easily adapt this part of the implementation for more general use, or define a suitable superclass from which neural network models for different analysis tasks can be derived.

Training of the model with two hidden layers, as used to render the results in Fig. 12 is then straightforward:

```
n_hidden = [4, 2]
model = NeuralNetwork(2, n_hidden + [1])
train(model, X, ys, learning_rate=0.2, n_epochs=2000,
      batch_size=50, report_every=100)
```

Our data set is simple and convergence is relatively fast despite our naive implementation. At this point, let us note that our imple-

mentation with autograd library from previous lessons is only for education purposes, and serious implementations should use serious Python libraries designed for fast training of neural networks, like `pytorch`.

Choice of activation functions and the problem of vanishing gradients

Activation functions introduce non-linearity into the network and are therefore essential. Without them, the composition of layers would collapse into a single affine transformation, regardless of the number of layers. Historically, neural networks used smooth activation functions such as the logistic (sigmoid) function

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

and the hyperbolic tangent

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

These functions are bounded and differentiable, which makes them suitable for gradient-based optimization. However, for large positive or negative inputs their derivatives become very small. When gradients are propagated backward through many layers, they can shrink to near zero, effectively preventing earlier layers from learning. This phenomenon is known as the *vanishing gradient problem*.

To address this issue, modern neural networks typically use the rectified linear unit (ReLU), defined as

$$\mathrm{ReLU}(z) = \max(0, z).$$

This function is simple and computationally efficient. More importantly, it has a constant derivative for positive inputs and does not saturate in that region, which allows gradients to propagate more effectively and leads to faster and more stable training.

A drawback of ReLU is that for negative inputs the output is zero, and the corresponding gradients vanish; in some cases, this can lead to units that never activate. Variants such as leaky ReLU alleviate this issue by allowing a small non-zero slope for negative inputs. The leaky ReLU function is defined as

$$\mathrm{LeakyReLU}(z) = \begin{cases} z, & z \geq 0, \\ \alpha z, & z < 0, \end{cases}$$

where $\alpha \in (0, 1)$ is a small constant (typically $\alpha \approx 0.01$). Unlike ReLU, this function has a non-zero derivative for negative inputs, which allows gradients to propagate even when the unit is not active.

In practice, ReLU (or a related variant) is used in hidden layers, while the activation function in the output layer is chosen according to the task. For binary classification, we typically use the sigmoid function; for multi-class classification, the softmax function; and for regression, no activation function is applied, leaving the output as a linear combination of the final layer's inputs.

Regularization

Neural networks are flexible models with many parameters and are therefore particularly prone to overfitting. As with the models we have discussed in our previous lessons, regularization techniques can be used to constrain the model so that it generalizes better to unseen data.

A direct extension of regularization from generalized linear models is the use of penalties on the model parameters. The most common is the ℓ_2 penalty (also known as weight decay), where we add to the loss function a term proportional to the sum of squared weights,

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \sum_{\ell} \|W^{(\ell)}\|_2^2,$$

with $\lambda > 0$ controlling the strength of regularization. This encourages the network to keep its weights small, leading to smoother mappings and reduced sensitivity to noise in the data. The ℓ_1 penalty,

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \sum_{\ell} \|W^{(\ell)}\|_1,$$

promotes sparsity by encouraging some weights to become exactly zero. While this is useful in linear models for feature selection, it is much less common in neural networks. Neural networks typically rely on distributed representations, where many small contributions combine to form useful features, and the non-smooth nature of the ℓ_1 penalty can make optimization more difficult. In practice, ℓ_2 regularization is used far more frequently and is considered the standard choice.

In practice, the ℓ_2 penalty is often implemented directly in the optimization procedure as *weight decay*. Instead of explicitly adding the regularization term to the loss, the update rule for a parameter w is modified to

$$w \leftarrow (1 - \eta\lambda)\, w - \eta \nabla_w \mathcal{L},$$

where η is the learning rate. This formulation makes the effect of regularization particularly transparent: at each step of gradient descent, the weights are slightly shrunk towards zero. For this reason, weight decay is the standard way of implementing ℓ_2 regularization

in modern neural network libraries, and the two terms are often used interchangeably in practice.

Another widely used regularization technique is *dropout*. During training, each activation $a_i^{(\ell)}$ is independently set to zero with probability p, and kept with probability $1 - p$. This can be written as

$$\tilde{a}^{(\ell)} = m^{(\ell)} \odot a^{(\ell)}, \qquad m_i^{(\ell)} \sim \text{Bernoulli}(1 - p),$$

where $\odot$ denotes element-wise multiplication. This prevents the network from relying too heavily on any single computational unit and forces it to develop redundant, more robust internal representations.

Dropout was proposed by Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov in 2014 in an article published by Journal of Machine Learning Research.

Dropout can also be interpreted as a form of ensemble learning: at each iteration, we effectively train a different subnetwork obtained by removing a random subset of units, and all these subnetworks share parameters. The final model can thus be seen as an approximation to averaging predictions over a large collection of such subnetworks. At test time, all units are used. In modern implementations, this scaling is typically applied during training (so-called "inverted dropout"), so that no additional adjustment is needed at test time.

Dropout was widely used in early training of neural networks (circa 2012–2016), but is less common in modern architectures. It remains useful in smaller models or when data is limited, while alternatives such as weight decay and data augmentation are now more standard.

Most modern deep learning libraries like PyTorch implement inverted dropout by default.

A particularly simple yet effective form of regularization is *early stopping*. Instead of minimizing the training loss indefinitely, we monitor performance on a validation set and stop training at iteration t^* where the validation loss $\mathcal{L}_{\text{val}}^{(t)}$ is minimized. Beyond this point, continued optimization typically reduces training error but increases validation error, indicating overfitting. Early stopping can be interpreted as constraining the norm of the parameters implicitly: gradient descent tends to first capture large-scale structure in the data and only later fit noise. By halting training early, we prevent the model from reaching overly complex solutions. In practice, early stopping is easy to implement and often provides substantial improvements in generalization.

Finally, regularization can also be achieved at the level of the data through *data augmentation*. Instead of modifying the model, we expand the training set by creating additional data points from the existing ones while preserving their labels. For example, in a two-dimensional data set, we may slightly perturb a point $x = (2, 1)$ into nearby points such as $(2.1, 1.0)$ or $(1.9, 1.2)$, and keep the same class label. In doing so, we express the assumption that small changes in the input should not affect the output. This increases the diversity of the training data and reduces the model's tendency to memorize individual examples, leading to more robust and generalizable predictions.

Initialization and normalization

Training neural networks with gradient-based methods is highly sensitive to the scale of inputs, activations, and parameters. If these quantities are not properly controlled, optimization can become slow or even fail altogether.

A simple but very effective step is to normalize the input features. In practice, we typically transform each feature to have approximately zero mean and unit variance. This ensures that all inputs are on a comparable scale, which leads to more stable gradients and faster convergence of gradient descent.

Equally important is the initialization of the network parameters. If the weights are initialized with values that are too large, the activations can grow rapidly as they propagate through the layers, leading to unstable behavior and exploding gradients. If they are too small, the activations shrink towards zero, and gradients can vanish, preventing the network from learning. Modern initialization schemes address this by choosing the initial weights so that the variance of activations remains approximately constant across layers. Common choices include Xavier initialization (suitable for sigmoid or tanh activations) and He initialization (designed for ReLU networks).

Xavier (Glorot) initialization (2010) sets $\mathrm{Var}(w) = \frac{2}{n_{\mathrm{in}}+n_{\mathrm{out}}}$ and is suitable for sigmoid/tanh activations. He initialization (2015) uses $\mathrm{Var}(w) = \frac{2}{n_{\mathrm{in}}}$, better matching ReLU units where roughly half of activations are zero. Both aim to keep activations and gradients at a stable scale across layers, improving training.

These considerations can be understood in terms of signal propagation through the network: we would like both the forward activations and the backward gradients to remain within a reasonable range as they pass through many layers. Proper input normalization and parameter initialization help maintain this balance and are therefore essential for efficient training.

In more advanced settings, one may also normalize intermediate activations during training, for example by using batch normalization. Such techniques further stabilize optimization, but for the purposes of this chapter, careful input preprocessing and sensible initialization already provide most of the practical benefits.

Depth, width, and model capacity

The architecture of a neural network is determined by its *depth* (the number of layers) and its *width* (the number of units per layer). Increasing either allows the network to represent more complex functions. In fact, even a network with a single hidden layer can approximate any continuous function on a bounded domain, provided it has sufficiently many units. This observation, often referred to as the *universal approximation property*, suggests that depth is not strictly necessary for expressiveness. However, such shallow networks may require an impractically large number of units to represent functions

The universal approximation property was established by Cybenko (Approximation by superpositions of a sigmoidal function, in *Mathematics of Control, Signals, and Systems* 1989) and Hornik (Approximation capabilities of multilayer feedforward networks, in *Neural Networks* 1991). An accessible discussion can be found in Goodfellow et al., *Deep Learning* (2016), Chapter 6.

that can be described much more compactly with deeper architectures.

A *deep neural network* is a neural network with more than one hidden layer. The term "deep" refers to the stacking of layers, which enables the model to learn increasingly abstract representations. The term "deep" became common in the 2000s with the resurgence of multi-layer neural networks, particularly through work by Geoffrey Hinton and collaborators.

Depth enables neural networks to build *compositional representations*. Each layer transforms the output of the previous one, gradually constructing more abstract features from the input. This hierarchical structure often leads to more efficient representations: functions that would require many units in a shallow network can be implemented with far fewer parameters in a deep one. Width, on the other hand, controls the richness of representations within a layer by allowing the network to learn multiple features at the same level of abstraction. In practice, both depth and width contribute to the expressive power of the model, and their appropriate balance depends on the problem at hand.

The overall *capacity* of a neural network—its ability to fit a wide range of functions—is therefore governed by its architecture and the number of parameters. Models with insufficient capacity may underfit the data, failing to capture important patterns, while overly large models may overfit, adapting too closely to the training data and generalizing poorly. As with other models, controlling capacity through architectural choices and regularization is essential for good performance. In practice, selecting an appropriate depth and width is often guided by experimentation, domain knowledge, and validation performance rather than strict theoretical rules.

Learning rate and its scheduling

While the form of the model determines what functions can be represented, the learning rate largely determines whether these functions can actually be found in practice. Even with correct gradients and a well-designed architecture, inappropriate step sizes can prevent convergence altogether. This makes the learning rate a key practical parameter: even with correct gradients, poor step sizes can prevent the model from converging.

The *learning rate* η controls the magnitude of parameter updates. If it is too large, optimization may become unstable and fail to converge; if it is too small, training progresses very slowly and may stall. A common strategy is therefore to begin with a relatively larger learning rate and gradually reduce it during training. Such *learning rate schedules*—for example, step-wise or exponential decay—allow rapid initial progress followed by finer adjustments near a solution. In practice, simple schedules combined with monitoring of training and validation performance are usually sufficient, and careful tuning of the learning rate often has a greater impact than more complex model modifications.

Beyond fully connected networks

The neural networks we have considered so far are *fully connected* (or dense) networks, where each unit in a layer is connected to all units in the previous layer. Such architectures are very general and can, in principle, approximate a wide range of functions. However, in many applications the input data exhibit additional structure that can be exploited to design more efficient and effective models. Incorporating this structure into the architecture introduces an *inductive bias* that can greatly improve learning.

A great example are *convolutional neural networks* (CNNs), which are designed for data with spatial structure, such as images. Instead of connecting every input to every unit, CNNs use local connections and shared weights, applying the same small filter across different parts of the input. This reduces the number of parameters and allows the network to detect local patterns (such as edges or textures) regardless of their position.

Convolutional neural networks were introduced by Yann LeCun and collaborators in the late 1980s. One of the first successful applications was LeNet-5 (1998), developed at AT&T Bell Labs and published in the Proceedings of the IEEE, where CNNs were used for handwritten digit recognition.

Another important class are *recurrent neural networks* (RNNs), which are tailored to sequential data. In these models, the computation is performed step by step, and the network maintains a hidden state that summarizes past inputs, making them suitable for tasks involving time series or text. More recently, *transformer* architectures have become dominant in many sequence modelling tasks. They rely on attention mechanisms that allow the model to directly relate different parts of the input sequence to each other, enabling efficient modelling of long-range dependencies.

RNN ware also introduced early, in the 1980s, by John Hopfield (1982) and David Rumelhart, Geoffrey Hinton, and Ronald Williams (1986).

Transformer architectures are a much more recent development (introduced by Vaswani et al., 2017). They have largely replaced recurrent models in many applications, particularly in natural language processing, due to their ability to model long-range dependencies efficiently through attention mechanisms and parallel computation.

Finally, neural networks are not limited to predicting outputs from inputs, but can also be used to *generate* data. In *generative machine learning*, the goal is to model the underlying distribution of the data so that new samples can be drawn from it. Examples include models that generate realistic images, text, or audio. Architectures such as autoencoders, which learn compressed representations by reconstructing their inputs, generative adversarial networks (GANs), and transformer-based language models fall into this category. While these models build on the same principles discussed in this chapter—compositions of differentiable functions trained by gradient-based optimization—their objectives and applications differ and are broad and consitute the core of modern AI.

Interpretation of neural networks

An obvious question is whether neural networks, like simpler models, can be interpreted. In linear models, interpretation is often straightforward: coefficients directly describe the effect of each input

variable on the output. The parameters of a neural network, however, are distributed across many layers and interact in a highly non-linear way. As a result, individual weights or units typically do not have a simple, standalone interpretation. Instead, the model should be understood as learning a sequence of transformations that map the input data into a representation that is useful for the final task.

That said, neural networks are not entirely opaque. We can often gain insight into their behavior by examining their predictions, analyzing how outputs change with respect to inputs, or visualizing intermediate representations, as we have done in a simple example in this chapter. These approaches provide a more global understanding of what the model has learned, even if a simple parameter-level interpretation is not available.

Interpretability is an active area of research. Methods such as feature attribution, saliency maps, and attention analysis aim to make neural networks more transparent, especially in high-stakes applications.

Backpropagation: a manual approach to gradient computation

Although throughout this chapter we have relied on automatic differentiation to compute gradients, this is not the only way to train a neural network. Historically, neural networks were first implemented by deriving the necessary partial derivatives by hand and then programming the corresponding parameter updates directly. This classical procedure, known as *back-propagation*, makes explicit how errors at the output layer are propagated backward through the network to determine how each weight and bias should be adjusted. Today, such derivations are usually hidden inside machine learning libraries, but understanding them remains valuable: they reveal the structure of learning in neural networks and clarify what automatic differentiation is actually doing for us. For this reason, we briefly step back from the autograd perspective and present the classical derivation.

We start with some conventions. We will assume that all units of the neural network take values between 0 and 1. We refer to this value as *activation* and denote it by a. By $\hat{y}$ we will now denote the output of the entire network. For simplicity, we will assume a sigmoid activation function at the output layer and use a squared-error loss. (Other choices, such as softmax with cross-entropy, require slightly different derivatives.)

Consider a simple neural network with one input feature x and one output $\hat{y}$, and one neuron in each of the two hidden layers (Fig. 13).

Let us, for simplicity, assume we are dealing with only one example in the training set, and define a cost function as a squared error:

$$J = \frac{1}{2}(a_3 - y)^2.$$

We will use a notation that explicitly denotes the layer number.

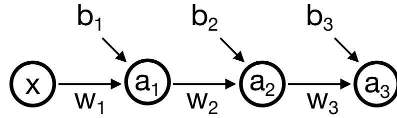

Figure 13: An example of a network with a single input and output and one neuron per layer.

The input layer is $l = 1$, and the output layer is $l = L$. For the network in Fig. 13, we have $L = 4$. The activation of layer l is denoted by $a^{(l)}$.

We can thus write that the weighted sum of inputs for a neuron at layer l is

$$z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)},$$

and the activation of that neuron is

$$a^{(l)} = \sigma(z^{(l)}),$$

so that the cost function becomes

$$J = \frac{1}{2}(a^{(L)} - y)^2.$$

While we can use any activation function here, we will use the sigmoid activation function for convenience.

To implement gradient descent, we need to determine how the cost function J depends on the parameters of the neural network. For instance, how does J depend on the weight $w^{(L)}$? We can use the chain rule to compute the partial derivative:

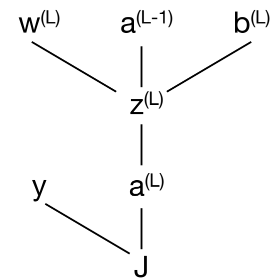

Figure 14: Dependency tree of the cost function J on some of the parameters from the neural network in Fig. 13.

$$\frac{\partial J}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \times \frac{\partial a^{(L)}}{\partial z^{(L)}} \times \frac{\partial J}{\partial a^{(L)}} \tag{1}$$

$$= a^{(L-1)} \times \sigma(z^{(L)})(1 - \sigma(z^{(L)})) \times (a^{(L)} - y). \tag{2}$$

We can interpret the terms in this expression as follows: $a^{(L-1)}$ denotes the activation of the previous layer, $\sigma(z^{(L)})(1 - \sigma(z^{(L)}))$ is the derivative of the activation function, and $(a^{(L)} - y)$ represents the prediction error.

So far, we have assumed a single training example. To generalize to a dataset of m instances, we modify the cost function:

$$J = \frac{1}{2m} \sum_{i=1}^{m} \left(a_i^{(L)} - y_i \right)^2.$$

Let us now consider a more general network with multiple neurons per layer. Again, we assume a single training instance. Consider the fragment shown in Fig. 15.

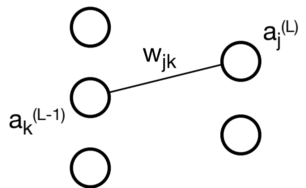

Figure 15: A fragment of a neural network showing the relation between the k-th neuron in layer $(l-1)$ and the j-th neuron in layer l.

We have:

$$z_j^{(l)} = \sum_{i=0}^{n_{l-1}-1} w_{ji}^{(l)} a_i^{(l-1)} + b_j^{(l)}, \tag{3}$$

$$a_j^{(l)} = \sigma(z_j^{(l)}), \tag{4}$$

$$J = \frac{1}{2} \sum_{j=0}^{n_L-1} (a_j^{(L)} - y_j)^2. \tag{5}$$

The weight $w_{ji}^{(l)}$ connects the i-th neuron in layer $(l-1)$ to the j-th neuron in layer l.

To perform gradient descent, we compute

$$\frac{\partial J}{\partial w_{ji}^{(l)}} = \frac{\partial z_j^{(l)}}{\partial w_{ji}^{(l)}} \times \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \times \frac{\partial J}{\partial a_j^{(l)}}.$$

To propagate derivatives to the previous layer, we use

$$\frac{\partial J}{\partial a_i^{(l-1)}} = \sum_{j=0}^{n_l-1} \frac{\partial z_j^{(l)}}{\partial a_i^{(l-1)}} \times \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \times \frac{\partial J}{\partial a_j^{(l)}}.$$

Using these expressions, we can recursively propagate gradients backward through the network and compute the influence of every parameter. This procedure is known as *backpropagation*. Intuitively, it:

- converts the discrepancy between output and target into an error derivative,
- propagates this error backward through the network,
- uses these derivatives to compute gradients with respect to the weights and biases.

While the scalar formulation makes the derivation transparent, practical implementations rely on matrix notation. Let the dataset contain m instances. For layer l, we define:

$$\boldsymbol{Z}^{(l)} = \boldsymbol{A}^{(l-1)} \boldsymbol{W}^{(l)} + \mathbf{1}(b^{(l)})^\top, \tag{6}$$

$$\boldsymbol{A}^{(l)} = \sigma(\boldsymbol{Z}^{(l)}), \tag{7}$$

where $\mathbf{1}$ is a column vector of ones.

We start backpropagation at the final layer by defining

$$\delta^{(L)} = \frac{\partial J}{\partial \boldsymbol{Z}^{(L)}}.$$

For the squared-error loss with sigmoid activation, this gives

$$\delta^{(L)} = (\boldsymbol{A}^{(L)} - \boldsymbol{Y}) \odot \left(\boldsymbol{A}^{(L)} \odot (1 - \boldsymbol{A}^{(L)}) \right). \tag{8}$$

The gradient with respect to the weights is then

$$\frac{\partial J}{\partial \boldsymbol{W}^{(L)}} = \frac{1}{m}\left(\boldsymbol{A}^{(L-1)}\right)^{\top}\delta^{(L)}, \qquad \frac{\partial J}{\partial b^{(L)}} = \frac{1}{m}\sum_{i=1}^{m}\delta_i^{(L)}.$$

For lower layers, the error is propagated backward as

$$\delta^{(l)} = \left(\delta^{(l+1)}(\boldsymbol{W}^{(l+1)})^{\top}\right) \odot \left(\boldsymbol{A}^{(l)} \odot (1 - \boldsymbol{A}^{(l)})\right).$$

The same formulas are then used to compute gradients for all layers. This recursive computation of derivatives is known as back-propagation.

From neural networks to generative AI

The ideas developed in this chapter can also lead to much larger generative models that dominate modern artificial intelligence. Although today's neural network-based systems are far more complex in architecture and scale, they are still built from the same basic ingredients: compositions of differentiable transformations, trained from data by gradient-based optimization, with gradients computed by automatic differentiation. What changes in modern generative models is the architecture, the scale of computation, and the objective. For example, complex deep networks such as language models generate text by predicting likely continuations in a sequence, and diffusion-based models generate images, audio, or video by learning to reverse a gradual noising process. Both are nonetheless neural networks trained by the same underlying principles discussed here.