Machine Learning with Scaly

Machine Learning with Scaly

Table of Contents

1 A First Model....4
1.1 Two Kinds of Points....4
1.2 One Neuron....5
1.3 How Wrong....6
1.4 Downhill....7
1.5 Better Inputs....8
1.6 Hidden Layers....10
1.7 Backpropagation....12
1.8 The Field....13
1.9 Generalization....16
1.10 The Field's Program....16
2 Many Points at Once....17
2.1 A Row for Each Point....17
2.2 Backpropagation with Matrices....18
2.3 Where the Numbers Live....20
2.4 Fast....21

1 A First Model

In this chapter we develop a program that can tell two kinds of points apart. We begin with a single neuron and then build, from layers of neurons, a small network that finds its own weights.

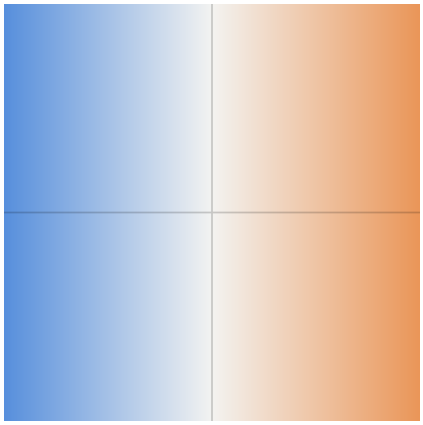
1.1 Two Kinds of Points

Let there be a plane with the coordinates x and y, each from -6 to +6. Into this plane we put points of two kinds: orange and blue. In the programs the kind of a point is represented by a number: `1` for orange and `-1` for blue.

A *model* is a function that computes, from the coordinates x and y of a place in this plane, a guess of the kind of a point at that place: a value between -1 and +1. At -1 the model is sure that a blue point lies there; at +1 it predicts an orange one with certainty. Values in between express uncertainty: the sign says which kind the model leans to, and the nearer the value is to 0, the less sure it is; at 0 it cannot decide.

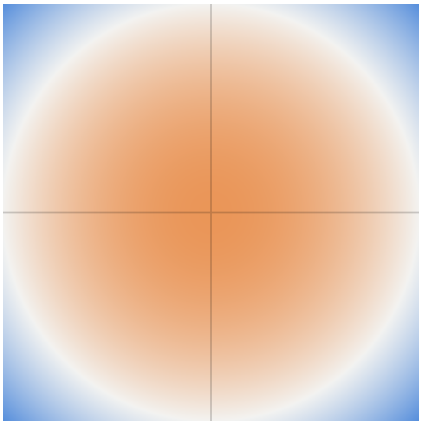
We could write such a function any way we like, for example

$$f(x, y) = \frac{x}{6}$$



It is +1 at the right edge, -1 at the left and 0 on the upright axis: on the right it guesses orange, on the left blue, and the nearer the middle, the less sure it is. Or

$$f(x, y) = 1 - \frac{x^2 + y^2}{36}$$



It is +1 at the centre, −1 in the four corners, because there $x^2 + y^2 = 36 + 36 = 72$, and 0 on the circle of radius 6 around the centre, where $x^2 + y^2 = 36$. Whether such a function can predict the points' kinds well depends on where they lie: the first suits points that are orange on the right and blue on the left, the second an orange disc in a blue ring.

Machine learning, on the other hand, does not make the formula up. It fixes only its general *form*, a formula with numbers left open, the *parameters*. In the second example the 36 could be such an open number, $f(x, y) = 1 - (x^2 + y^2)/c$, and c would decide how large the orange disc is; where the formula falls below −1, it counts as −1. Which values the parameters get, a program finds out from examples, points whose kind is known. Finding them is what this book calls *learning*, and a function with its parameters found a *trained model*.

TRY IT

What an open number does, you can try for yourself: open the page with the model $f(x, y) = 1 - (x^2 + y^2)/c$ and move c back and forth. The colouring follows the formula: a large c makes the orange disc large, a small one small.

1.2 One Neuron

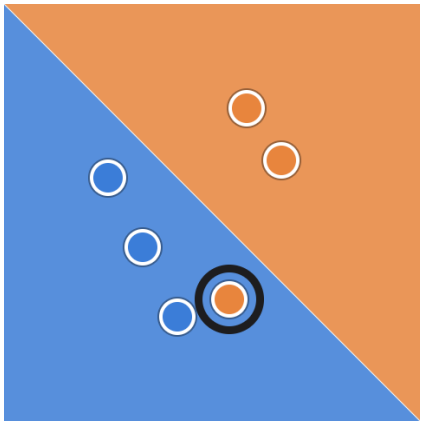
The simplest form of such a function is a *neuron*. It multiplies each of its inputs by a number of its own, its *weight*, adds them up and adds one more number, its *bias*:

$$z = w_1 x + w_2 y + b$$

The result z is called the neuron's *weighted sum*. Where it is positive the neuron guesses orange, where it is negative, blue. The points where z is exactly 0 form a straight line, so a neuron parts the plane by a line: the weights turn it, the bias moves it.

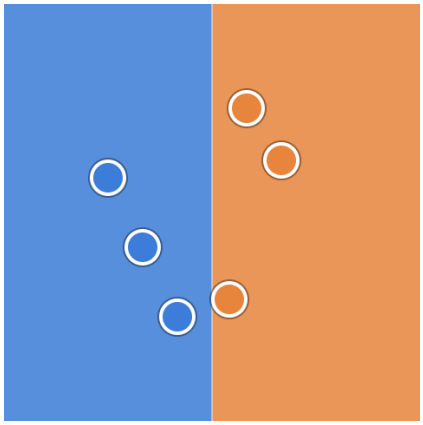
Now let us try to adjust the neuron so that it can tell, as well as possible, the colour of six points we scatter over the plane at will: three orange and three blue. We begin with the weights 1 and 1 and the bias 0:

$$z = x + y$$



Five of the six lie on the right side; the one in the ring is orange but lies below the line. With $w_2 = 0$ the line turns to the upright axis, and all six are right:

$$z = x$$



Changing the weights and the bias step by step until the guesses are right is the essence of what we shall call learning.

TRY IT

Open the page with the neuron and pull at w_1, w_2 and b: the line turns and moves, and the page counts how many of the six points lie right. With a click and a right click you can put down points of your own as well. Below the picture stands the function in Scaly that the page computes for it.

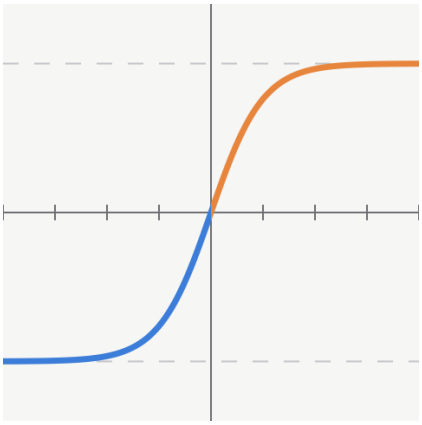
1.3 How Wrong

To change the weights by itself a program needs a measure of how wrong the neuron is. This measure is a number that gets smaller as the neuron gets better. Counting the wrong guesses does not serve: a small change of a weight usually changes no guess at all, so the count does not say which way to turn.

So we first map the weighted sum z to a number between -1 and 1 by a smooth function, here

the hyperbolic tangent:

$$a = \tanh(z)$$



With this function the model can answer as the beginning of the chapter asked: near +1 or -1 where it is sure, near 0 where it is not. Such a function we shall call the neuron's *activation function*, and a its answer for a particular point of the plane. Now the answer can be compared with the kind t the point really has:

$$L = \frac{1}{2}(a - t)^2$$

L is the *loss* of one point: 0 where the answer is exactly the kind, larger the further it is from it, and every small change of a weight changes it a little. The mean of the losses of all points is the neuron's loss, and that is the measure we were looking for.

The hyperbolic tangent is not the only possible activation function. In large networks one mostly takes an even simpler function today, *ReLU*: $\max(0, z)$, that is z itself where z is positive and 0 elsewhere. Its slope is always 1 for positive z, while that of tanh becomes nearly 0 far out. What that means for learning shows in the next section.

1.4 Downhill

So the loss of our neuron is a function of the three values w_1, w_2 and b. Now we want to find the values for which it becomes as small as possible. Trying every combination is out of the question: each of the three values can be any number, so there are infinitely many. Even if we tried only 100 numbers for each value, that would be 100^3, a million combinations, and for each we would have to compute the loss over all points. For three values a computer still manages that, but the network for XOR that we build later already has 13 numbers and so $100^{13} = 10^{26}$ combinations, and large networks have millions of weights. So we begin with some values and move from there step by step in the direction in which the loss falls fastest, with all three values at once. Which direction that is can be read off each value separately: how much does the loss change when the value changes a little? That is the *partial derivative* of the loss with respect to that value, for w_1 written $\partial L / \partial w_1$. It is partial because only this one value changes while the others are held fixed; the round ∂ instead of a d marks that.

The loss does not depend on w_1 directly. It depends on the answer a, the answer on the sum z, and the sum on w_1. Each link of that chain has a derivative that is easy to work out:

- How much does the loss change when the answer changes? $\frac{\partial L}{\partial a} = a - t$, the derivative of $\frac{1}{2}(a-t)^2$.
- How much does the answer change when the sum changes? $\frac{\partial a}{\partial z} = 1 - a^2$, the derivative of tanh.
- How much does the sum change when w_1 changes? $\frac{\partial z}{\partial w_1} = x$, because w_1 is multiplied by x.

The *chain rule* says that the derivative of the whole chain is the product of the derivatives of its links:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_1} = (a - t)(1 - a^2)x$$

For w_2 the last factor is y, for the bias it is 1. Added up over all points and divided by their number, these partial derivatives are the *gradient*: the direction in which the loss rises fastest. One step of *gradient descent* moves each of the three values a little the other way,

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

where the small number η is the *learning rate*.

And when does one stop? When the loss is small enough, or when it hardly falls any more. The pages of this chapter stop as soon as the loss is below 0.02 or has fallen by less than 0.0001 over a hundred steps. Such a rule is called a *stopping criterion*.

TRY IT

Open the page where the neuron learns. It begins with values that guess badly: only two of the six points are right. Press “One step” several times in a row: w_1, w_2 and b move a bit each time, the line turns, the loss falls. Press “Learn”: after some thirty steps all six are right; after a good eighty the loss is below 0.02, and learning stops. w_2 is then nearly 0, as in our attempt by hand. With the learning rate 0.01 it goes much more slowly. Below the picture stands the step in Scaly.

Try the start values $w_1 = w_2 = -3$ as well (the page with these values): then the neuron hardly moves, and after a hundred steps the page stops, since the loss no longer falls. Nearly all points lie on the wrong side, and far from the line, where tanh is nearly ±1 and hardly rises any more. Since the factor $1 - a^2$ is nearly 0 there, so is the gradient: a neuron that is sure and wrong hardly learns. This is called *saturation*, and it is why one begins with small weights.

1.5 Better Inputs

Our neuron parts the plane by a line. But some arrangements of points cannot be parted by a line, for example a disc of orange points inside a ring of blue ones: every line then has orange

and blue on both sides.

TRY IT

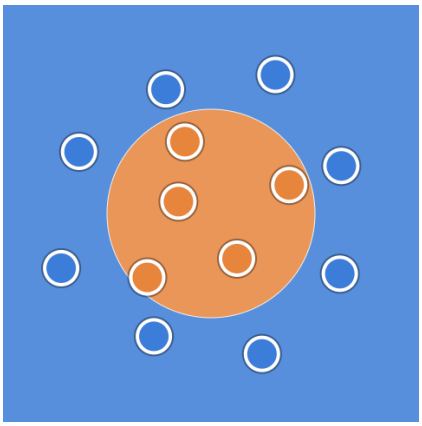
Open the page where the neuron learns and put down unfavourable points yourself: some orange ones in the middle and a ring of blue ones around them. A click puts down an orange point, a right click a blue one, a click on a point takes it away. Press “Learn”: the neuron finds no line that fits, settles for a plane that is unsure everywhere, and stops, since the loss no longer falls.

What the neuron lacks is not skill at sums but the right view of the points. Let us give it the squares of x and y as its inputs instead:

$$z = w_1 x^2 + w_2 y^2 + b$$

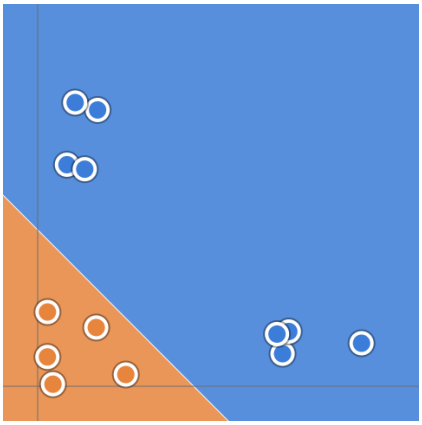
With $w_1 = w_2 = -1$ and $b = 9$, say, z is 0 exactly where $x^2 + y^2 = 9$, on the circle of radius 3 around the centre. Inside, z is positive, outside negative:

$$z = -x^2 - y^2 + 9$$



Why can the neuron do this all of a sudden? Let us call its new inputs $u = x^2$ and $v = y^2$. Then $z = w_1 u + w_2 v + b$ is a straight line again, only not in the plane of x and y but in that of u and v. Drawn there, the disc lies near the origin and the ring further out, and a line parts them:

$$z = -u - v + 9$$



So the neuron still draws a line, but a line in the inputs it is given; on the plane of the points it becomes a circle. Choosing what a model sees is called *feature engineering*, and good inputs can make a hard problem easy.

TRY IT

Open the page with the neuron given x^2 and y^2. It begins with the disc and the ring and guesses badly at first. Press "Learn": after some hundred steps all points are right, and the orange area settles around the disc; after a good two hundred the loss is below 0.02. Here too you can put down points and take them away.

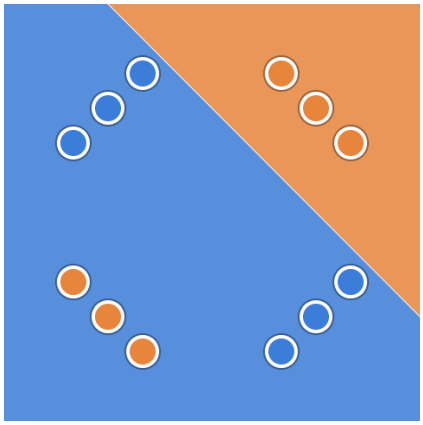
1.6 Hidden Layers

For a disc the right inputs are easy to guess; for most data they are not. So we leave it to neurons to compute the inputs. It is this idea that makes neural networks strong.

Let us take several neurons instead of one. Each of them is given the same inputs x and y, but has weights and a bias of its own and so draws a line of its own; its answer says on which side of it a point lies. Neurons that read the same inputs like this form a *layer*. Now we form one more layer. It is given not x and y but the answers of the neurons of the first layer, and here it consists of a single neuron, the *output*. So the computation runs in two stages: from x and y the first layer computes its answers, and from those answers the output computes its own. From outside one sees only the inputs and the output, so the layer in between is called a *hidden layer*; its answers are intermediate results.

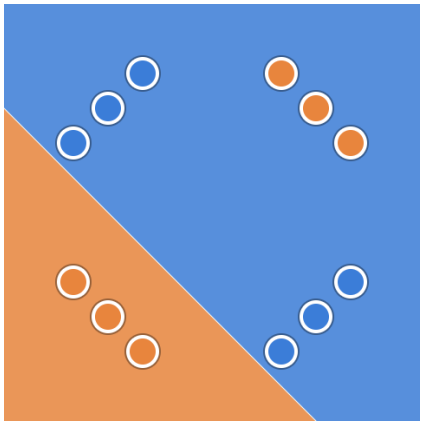
The data that shows it best is called XOR: orange at the upper right and the lower left, blue at the other two corners. No single line parts them. Two neurons in a hidden layer can, if each cuts off one of the orange corners. Let their answer here simply be the sign sgn of their weighted sum, +1 or −1. The first says +1 only at the upper right:

$$h_1 = \mathrm{sgn}(x + y - 3)$$



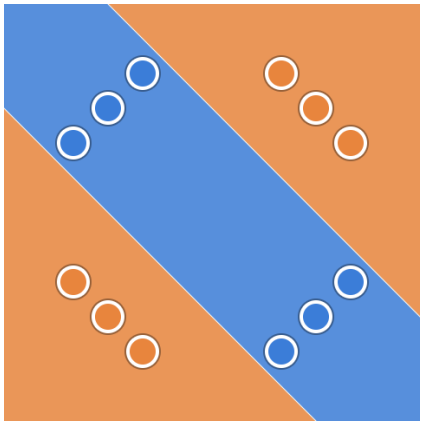
the second only at the lower left:

$$h_2 = \text{sgn}(-x - y - 3)$$



The output reads the two answers. If either of them says +1, $z = h_1 + h_2 + 1$ is +1, so orange; if both say −1, it is −1, so blue:

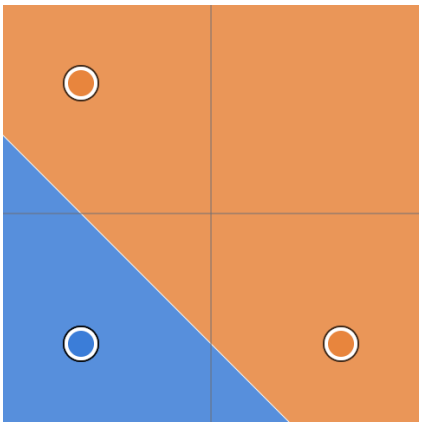
$$z = h_1 + h_2 + 1$$



How the output manages that shows when each point is put where the hidden layer takes it.

The output is a neuron like our first one, only its inputs are called h_1 and h_2 instead of x and y. So it does not see the plane of the points but a new plane with the axes h_1 and h_2, and in it each point has the place (h_1, h_2): the three orange points at the upper right land at $(+1, -1)$, the three at the lower left at $(-1, +1)$, and all six blue ones at $(-1, -1)$. In this plane a single line parts the two kinds, and that line is the one the output draws:

$$h_1 + h_2 + 1 = 0$$



The output joins the two lines into an “or”: orange where the first or the second neuron says yes. Two lines and a neuron that joins them do what one line alone cannot. With more neurons and more layers, borders of any intricacy can be built this way.

TRY IT

Open the page with a network of one hidden layer. It has three hidden neurons and begins on the points of XOR with small weights that guess only half of the points right. Press “Learn”: after a good hundred steps all twelve are right. The small pictures show which line each hidden neuron has found; the output joins them. Further down the page the network stands in layers, one above the other, with the weights as lines; drag it with the mouse to turn it. How the network finds its thirteen numbers is the next section’s subject.

The activation function matters here. If each neuron simply passed on its weighted sum, the output would add up sums of sums, which is again only a weighted sum of x and y, so again only a line. Only the activation function, which bends each neuron's sum, lets the layers do more than one neuron could. In our example by hand that is the sign; in the network from the box above, the hyperbolic tangent.

1.7 Backpropagation

How does a hidden neuron learn? Its weights are as far from the loss as can be, behind the output neuron. The chain rule still works, with a chain one link longer. The clever part is the order in which it is computed.

First everything is computed forward, as before: each hidden neuron's answer h_j, then the output's sum and its answer a. Then backward. The output neuron's share of the error, the loss by its sum, is the familiar

$$d = (a - t)(1 - a^2)$$

and its weights' derivatives follow from it as for a single neuron: dh_j for the weight v_j that reads hidden neuron j. Hidden neuron j reaches the loss through that weight, so its answer's slope is dv_j, and through its own tanh its share of the error becomes

$$d_j = dv_j(1 - {h_j}^2)$$

From there its weights are once more a single neuron's: $d_j x$, $d_j y$ and d_j for its bias. The error is handed back through the network layer by layer, each layer's share computed from the share of the layer after it. That is *backpropagation*, and it is the whole secret of how networks of any depth learn.

TRY IT

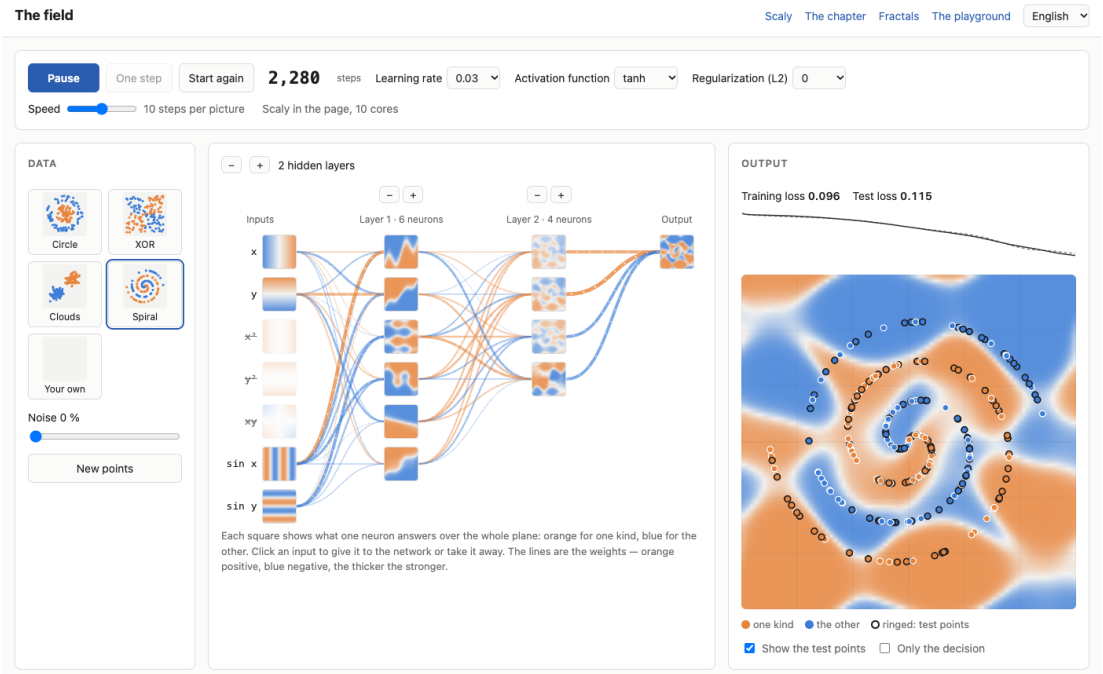
Open the page with the network for XOR once more. Below the picture there is now the procedure `network_step`: backpropagation as derived here. For each point it computes forward the answers h_j and a, then back d and from it each d_j, and at the end it moves each of the thirteen numbers a little against its gradient. Each press of One step is one call of this procedure.

The hidden neurons on the page start from different weights on purpose. Had they all started alike, they would answer alike, get the same gradient in every step and stay alike forever, three copies of one line. That is why a network usually starts from random weights.

A network needs nothing more to learn, however many layers it has: compute forward, hand the error back, move each weight a little against its gradient.

1.8 The Field

The networks in our boxes were small and each built for one task. The *field* is a page on which you put a network together yourself and watch it learn. You choose the points, the inputs, how many hidden layers it has and how many neurons in each, and how it learns. It computes exactly as the last sections derived. At the top of the page lies a bar with the buttons. Below it stand three parts side by side, from left to right the data, the network and the output; on a narrow screen they stand one below the other. The picture shows the page with the spiral, after a little over 2000 steps.



The bar

Pause stops the network; the same button is then called Learn and lets it learn on. While it is stopped, One step takes exactly one step of gradient descent over all the points it learns from. Start again gives the network new random weights and keeps the points. Beside them the page counts the steps since the start.

The Learning rate is the one from the section “Downhill”, to choose from 0.001 to 3. Under Activation function there are four to choose from, for the neurons of the hidden layers: the hyperbolic tangent, ReLU, the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$, which runs between 0 and 1 instead of between -1 and $+1$, and none. Without an activation function each hidden neuron simply hands on its weighted sum, and the whole network, however many layers it has, draws only one line. The output neuron always answers with the hyperbolic tangent. Regularization (L2) is explained in the next section.

The Speed says how many steps the network takes between two pictures, in five levels from 1 to 100. The page draws a picture every 30 milliseconds at most.

The data

Five small pictures choose the points: Circle, a disc inside a ring; XOR, the four quarters of the plane, the kind by the sign of xy; Clouds, two heaps around $(2, 2)$ and $(-2, -2)$; Spiral, two arms wound into each other; and Your own, an empty plane. Into it you put points yourself, an orange one with a click, a blue one with a shift-click or a right-click. For the other four the page draws 300 points, half of each kind.

The Noise slider moves each point at random, up to 50 %, so that the kinds reach into each other; for the clouds it makes the heaps wider instead. New points draws the same arrangement anew. A change of data only exchanges the points; the network learns on

with the weights it has.

The network

The network stands in columns, from left to right in the order in which it is computed. The first column holds the seven possible inputs: x, y, x^2, y^2, xy, $\sin x$ and $\sin y$. You know the squares from the section “Better Inputs”; new are the sines of the two coordinates, which go up and down a little less than twice across the plane. A click on an input gives it to the network or takes it away. One taken away is pale and struck through, and one always stays.

Then come the hidden layers, each a column. With − and + above the network you take a layer away or add one, up to six; with − and + above a column, a neuron of that layer, up to eight. Without a hidden layer the network is a single neuron as at the beginning of the chapter. The last column is the output neuron.

Each neuron, each input too, is a small square. It shows what the neuron answers over the whole plane, orange where the answer is positive, blue where it is negative, like our pictures for XOR. The lines between the squares are the weights: orange positive, blue negative, the thicker, the larger their size. On each line a light runs from left to right, the faster, the stronger the weight. When the network's shape, its inputs or its activation function change, it starts over from new random weights.

The output

At the top stand two numbers, the Training loss and the Test loss, below them their course since the last start: the training loss solid, the test loss dashed. What training and test points are is explained in the next section.

The large picture is the plane, from −6 to 6 both ways, with all the points. Beneath them lies the output neuron's answer at every place, the stronger the colour, the nearer to +1 or −1. Only the decision shows instead only on which side of zero the answer lies, and with it the border the network draws. Show the test points shows and hides the ringed points.

Below the three parts the page says in a few sentences what happens, and at the very bottom the program can be unfolded that the last section of this chapter is about. The links in the boxes set the page up to fit. What they choose stands in the address after the #. Without such settings the page starts with the circle, the inputs x and y and two hidden layers of four and two neurons.

TRY IT

Open the field with the spiral, only x and y as inputs and one hidden layer of four neurons. The network starts learning at once, but its loss falls only slowly and stays high: four lines are not enough for the spiral. Then click sin x and sin y in the column of the inputs. The network starts again, and after a few seconds its loss has fallen to a fraction.

1.9 Generalization

So far the network has learned from all the points, and we have only asked how well it meets those very points. But a model is meant to place points rightly that it has never seen as well. That is why the field splits its points into two halves. The network learns from the one: these are the *training points*, and their mean loss, the *training loss*, is the loss gradient descent makes smaller. The other half, the *test points*, drawn with a ring, the field holds back: the network never learns from them. Their mean loss, the *test loss*, says how well the network *generalizes*.

When the data is noisy and the network large, the two losses part ways. The network learns the training points ever better, their noise included, and draws borders around single stray points. The test loss then stops falling and rises again. That is *overfitting*. One remedy is *regularization*: for every weight w a penalty λw^2 is added to the loss. λ is a small number we choose. Large weights become expensive. The penalty's derivative is $2\lambda w$, so every step of gradient descent pulls every weight a little towards zero. Smaller weights give smoother borders, because a neuron with small weights goes over from -1 to $+1$ only slowly. Because the penalty is the square of the weights, this form is called *L2 regularization*.

TRY IT
Open the spiral with 30 % noise and four hidden layers of eight. After some seconds the training loss is far below the test loss, and while the training loss keeps falling, the test loss rises. The plane shows islands around single points. Choose 0.01 under Regularization (L2) and click Start again: the penalty keeps the weights small. The training loss stays higher, but the test loss stays well below the one before and hardly rises any more, and the borders are smoother.

1.10 The Field's Program

Behind the page stands one Scaly program, `field.scaly`, built for WebAssembly. It is the program of the page with the network for XOR grown up: everything the page lets you set, and the backpropagation that `network_step` writes out for a network with one hidden layer, here in loops over any number of layers. The numbers the page shows it reads straight out of the program's memory, without a copy. The picture of the plane is a `for` over its 6400 points, which Scaly spreads over the cores of the machine by itself. The same program also runs without the page, on the command line: `field 0 500` lets the network with hidden layers of four and two neurons learn the circle for 500 steps, writes down the two losses every hundred steps and at the end how many of the 200 test points it places rightly. Its source is on GitHub.

Everything this chapter derived by hand, the chain rule through every link and the gradient of every weight, a program can derive too. That is the subject of the chapters ahead: first how to write the computation for all points at once, with matrices, and how Scaly holds these many numbers and computes with them fast; then a tape that records the forward computation and plays it back to get every derivative.

2 Many Points at Once

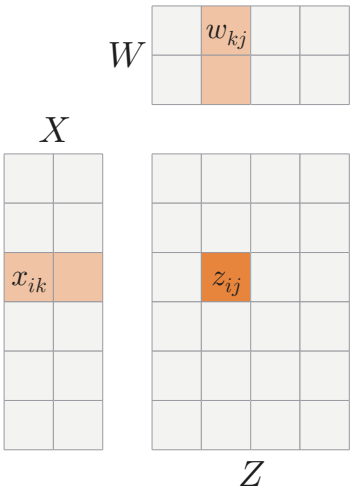
Our network computes point by point so far. In this chapter we write the same computation for all points at once, with matrices, and see how Scaly holds the many numbers and computes with them fast.

2.1 A Row for Each Point

A neuron forms its weighted sum for one point, $z = w_1 x + w_2 y + b$. For the 200 training points of the circle those are 200 sums, and a layer of four neurons forms 800. Writing the inputs of all points one below the other, a row for each point, gives a table of 200 rows and 2 columns, a *matrix* X. The layer's weights we write as a matrix W too, a column for each neuron, so 2 rows and 4 columns.

The sum of neuron j for point i is then row i of X with column j of W, multiplied number by number and added up, plus the neuron's bias:

$$z_{ij} = x_{i1} w_{1j} + x_{i2} w_{2j} + b_j$$



All these sums together are a matrix again, Z, a row for each point and a column for each neuron. The computation that forms it from X and W is called the *matrix product* and is written like a multiplication:

$$Z = XW + b \qquad H = \tanh(Z)$$

The bias is added to every row, and tanh acts on each number by itself. Every further layer is the same line again, with H instead of X as its input. The whole forward computation of our network is three such lines.

TRY IT

Watch the matrix product as an animation. For a small example, X with three rows and W

with three columns, one number of Z appears after the other: row i and column j light up, then the two pairs of numbers that are multiplied, and below them the calculation runs along. With One step you can work it out with it at your own pace.

What do we gain by this? The computation is exactly the same: the same multiplications and additions in the same order, and the program of this chapter arrives number for number at the field's result. Gained are three things. The program becomes clearer: the forward computation is three lines, and the number of points no longer appears in them. The matrix product is the one computation processors and graphics cards are built for; large matrices spread well over the cores and onto the card. With our 200 points and four neurons none of this shows yet, the matrices are tiny. And it prepares the next chapter: a program that forms derivatives by itself then needs a rule for a handful of matrix operations instead of for every single multiplication.

2.2 Backpropagation with Matrices

Backwards it works the same way. In chapter 1 every neuron had, for every point, its share of the error, the error after its sum. Write these shares one under the other as well, a row for each point and a column for each neuron, and they make a matrix D the size of Z: d_{ij} is neuron j's share of the error for point i. For the output it is a single column, row by row the familiar $(a-t)(1-a^2)$.

One layer further forward, neuron k gets its share through all the weights with which the neurons of the layer behind read its answer, and through its own tanh:

$$d'_{ik} = (\textstyle\sum_j d_{ij} w_{kj})(1 - h_{ik}^2)$$

With a single output the sum has one term, and that is the dv_j of chapter 1. The sum itself is a matrix product again: row i of D with row k of W. To make that a row times a column, W is mirrored, its rows becoming columns; that is called the *transpose* W^T. The tanh acts on each number on its own, number times number at the same place, written $\odot$:

$$D' = (DW^T) \odot (1 - H^2)$$

What remains are the derivatives of the weights. In chapter 1 every point added to weight v_j the product dh_j: the answer the weight reads, times the share of the neuron that reads it. Added up over all points and divided by their number, that was the derivative. For w_{kj} it is the same, point i adds $h_{ik} d_{ij}$, and the sum over all points is row k of H^T times column j of D:

$$\textstyle\sum_i h_{ik} d_{ij} = (H^T D)_{kj}$$

All these sums together are the matrix product $H^T D$, the size of W, one number for each weight. The bias reads a 1 instead of an answer, so its sums are simply the column sums of D. With that a whole learning step is written in matrices: forwards, each layer's Z and H; backwards, each layer's D, from it $H^T D$ and the column sums; and then every weight is decreased by its derivative as before, divided by the number of points and times the learning rate.

In the program of this chapter each of these formulas is a function. `back` makes from the D of a layer, its weights W and the answers H of the layer before it that layer's D':

```
function back(D: Matrix, W: Matrix, H: Matrix) returns Matrix
{
    var below zeros(D.rows, W.rows)
    for i in D.rows
    {
        var k 0
        while k < below.cols
        {
            var s 0.0
            var j 0
            while j < W.cols
            {
                s := s + D.at(i, j) * W.at(k, j)
                j := j + 1
            }
            let out H.at(i, k)
            below.put(i, k, s * (1.0 - out * out))
            k := k + 1
        }
    }
    below
}
```

The outer loop goes over the points i, the middle one over the neurons k of the layer before, the inner one makes the sum over j. In doing so it reads row k of W, and that is column k of W^T: the transpose is never written down, it is enough to read W the other way round. And the $\odot$ is the multiplication in `below.put`, each sum times the slope of its own tanh.

`weight_grad` is the matrix product $H^T D$ in the same pattern; here too H^T is never written down, the inner loop reads column k of H. `bias_grad` adds up the columns of D, `out_share` makes the D of the output, and `descend` decreases every number of a matrix by its derivative. That makes the whole learning step:

```
procedure step(mutable net: Net, X: Matrix, T: Matrix, rate: f64)
{
    ; forward, layer by layer
    let H1 layer(X, net.W1, net.b1)
    let H2 layer(H1, net.W2, net.b2)
    let A layer(H2, net.W3, net.b3)
    ; backward, layer by layer
    let D3 out_share(A, T)
    let D2 back(D3, net.W3, H2)
    let D1 back(D2, net.W2, H1)
    ; every weight a little against its gradient
    descend(net.W3, weight_grad(H2, D3), rate, X.rows)
    descend(net.b3, bias_grad(D3), rate, X.rows)
```

```
    descend(net.W2, weight_grad(H1, D2), rate, X.rows)
    descend(net.b2, bias_grad(D2), rate, X.rows)
    descend(net.W1, weight_grad(X, D1), rate, X.rows)
    descend(net.b1, bias_grad(D1), rate, X.rows)
}
```

At the top the three lines forward, below them the three backward, from the output to the input, because each D comes from the D of the layer behind it. Only when all the D are there do the weights change: `back` needs the old ones. The points appear in these lines only as `X.rows`, their number, which `descend` divides by. The whole program is on GitHub.

2.3 Where the Numbers Live

In the program a matrix is a record of its two sizes and a `Vector[f64]` with its numbers, row after row. Rows and columns count from 0 there, and the number in row i and column j is at place $i \cdot n + j$ when the matrix has n columns:

```
define Matrix(rows: int, cols: int, data: Vector[f64])
{
    function at(this, i: int, j: int) returns f64
        this.data[i * this.cols + j]

    procedure put(mutable this, i: int, j: int, v: f64)
        this.data.put(i * this.cols + j, v)
}

function zeros(rows: int, cols: int) returns Matrix
{
    var data Vector[f64]#(rows * cols)
    for k in rows * cols
        data.put(k, 0.0)
    Matrix(rows, cols, data)
}
```

`zeros` makes a matrix in which every number is 0. The `#` after the type says where its numbers live; more on that in a moment.

A learning step makes quite a few such matrices: forward the answers of the three layers, backward the three D, and the derivatives of the weights. Together that is a good 2800 numbers or 22 KB a step, 1.1 GB for 50,000 steps. Only one step of it is ever needed: once the weights have changed, everything else is out of date.

Other languages clear such intermediate values away with a *garbage collector*, which looks from time to time at what the program can still reach; in C you give each one back with `free`. Scaly does neither. What a call makes lives on a memory page of that call, and when it returns, the whole page is freed at once. What it returns must outlive that, so it makes it right away on its caller's page. That is the `#` in `zeros`. That `layer`, `back` and the other functions return their result from `zeros`, the compiler sees by itself, and it hands the page on. So all

the matrices of a learning step live on the page of `step`, and that is freed when `step` returns. Only the weights stay: the main program made them, and `descend` writes into them.

Measured, the program needs at most 2.08 MB of memory with 500 steps, and just as much with 50,000 steps. The learning steps come and go, and the memory does not grow with them.

There is one trap. The main program has a page too, and it lives until the end. Write the output of the losses straight into the main loop, and the answers it computes for it lie on that page and stay: after 50,000 steps, with an output every 100 steps, that is 18.7 MB. So the output is a procedure of its own, and what comes about in it goes with it:

```
procedure report(step: int, net: Net, X: Matrix, T: Matrix,
                 TX: Matrix, TT: Matrix)
{
    let train loss(answer(net, X), T)
    let test loss(answer(net, TX), TT)
    print "step `step` train `train` test `test`"
}
```

2.4 Fast

That leaves the question of how fast all this is. For our circle it is hardly worth asking: built with **scaly build --release**, the program takes a hundredth of a second for its 500 learning steps. With images it is another matter; the handwritten digits of MNIST are 60,000 images of 784 points each. How the time grows with the size shows in the matrix product on its own, $C = AB$ for two square matrices of n rows and columns:

```
function product(A: Vector[f32], B: Vector[f32], n: int) returns Vector[f32]
{
    var C Vector[f32]#((n * n) as size_t)
    for t in n * n
    {
        let i t / n
        let j t % n
        var s: f32 0.0
        var k 0
        while k < n
        {
            let a A[(i * n + k) as size_t]
            s := s + a * B[(k * n + j) as size_t]
            k := k + 1
        }
        C.put(t as size_t, s)
    }
    C
}
```

A round of the loop makes one number of C, row i of A times column j of B. Here the

matrices are just their `Vector`, row after row, and their numbers are `f32` instead of `f64`: they take half the memory, and that is how a graphics card computes. The time for one product, measured on a MacBook Pro with an M1 Pro, with one core, with all ten, on the graphics card, and as Scaly decides by itself:

n	one core	all cores	card	Scaly decides
64	0.18 ms	0.10 ms	0.33 ms	0.12 ms
128	1.7 ms	0.29 ms	0.43 ms	0.33 ms
256	15.6 ms	2.0 ms	1.06 ms	1.7 ms
512	127 ms	17.9 ms	5.0 ms	5.0 ms
1024	1.08 s	0.20 s	35 ms	35 ms
2048	25.4 s	8.0 s	0.27 s	0.27 s

With one core the time grows like n^3: twice the size is eight times the work. Only from 1024 to 2048 it takes about 24 times as long. The numbers of a column of B lie far apart in memory, each n places after the one before, and at 2048 they no longer fit into the processor's fast intermediate memory, the cache; it then mostly waits for memory. All cores together are faster from 64 on, at 1024 a good five times. The graphics card overtakes them between 128 and 256, and at 2048 it computes thirty times as fast as all the cores.

The last column is the one that matters. The program says only `for`; where it goes, Scaly decides while it runs: it measures what a round costs and hands the loop to the cores or to the card where that pays. It makes the faster choice; only at 256, where cores and card are close, it switches between the two and ends up in between. And everywhere the result is the same, bit for bit: the card adds up every sum in the same order as the processor.

The loops of our circle program are of the same kind. Each round fills one row of the matrix it writes, and so that Scaly sees this, the inner loops run up to that matrix's number of columns: `while j < H.cols` in `layer`. So Scaly spreads all six loops over the points across the cores by itself as soon as that pays; with 200 points it does not pay yet. The whole measuring program is on GitHub.