SMALL EXACT NEURAL NETWORKS

N. P. STRICKLAND

Here we discuss some neural networks where one can specify the weights explicitly.

1. Binary boolean functions

We can represent truth values by 0 and 1. There are 16 possible boolean functions of two arguments, and they can all be represented by 2-layer networks. Explicitly, if $x, y \in \{0,1\}$ then $xy = (x+y-1)_+$. Now suppose we have $f\colon \{0,1\}^2 \to \mathbb{R}^m$ with

$$f(0,0) = u_0 \qquad f(0,1) = u_1 \qquad f(1,0) = u_2 \qquad f(1,1) = u_3.$$

We then find that

$$f(x,y) = u_0 + (u_2 - u_0)x + (u_1 - u_0)y + (u_3 - u_2 - u_1 + u_0)xy.$$

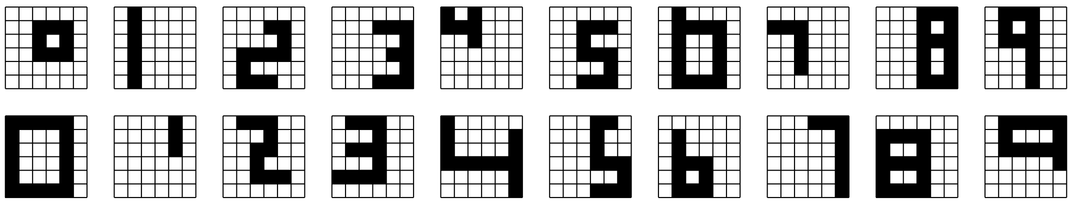
This leads to an evident encoding of f as a RELU layer (encoding $(x,y) \mapsto (x,y,xy)$) followed by a linear layer. In particular, we can apply this to the function $f\colon \{0,1\}^2 \to \mathbb{R}^{16}$ which computes all 16 boolean functions simultaneously. This is implemented in the file `bool.py`.

A similar construction computes any function of n boolean variables as a RELU layer with $2^n - 1$ outputs followed by a linear layer. For particular boolean functions it will be more efficient to use a deeper network.

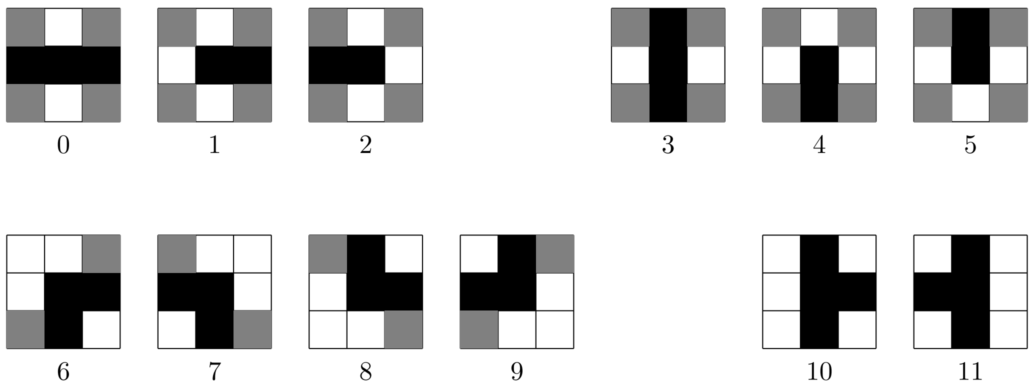
2. Digit recognition

A standard initial exercise in Machine Learning is to recognise images of handwritten digits from the mnist dataset. Here we do an even simpler version of that.

Consider digits drawn as 6×6 pixel images using straight horizontal and vertical lines, as illustrated below.



We can build a neural network to recognise such digits as described below. The first step is a convolutional layer with twelve kernels of shape 3×3, whose effect is illustrated by the pictures below.

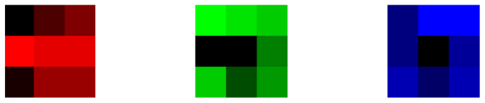


In more detail, we say that a pixel has type i if the surrounding 3×3 block matches the above picture marked i, in the sense that black pixels must be black, white pixels must be white, and grey pixels can be either colour. To interpret this for pixels on the edge, we assume that the image is surrounded by white pixels. For a valid digit image, each black pixel must have type i for some i. If we start with an image represented as an element of $M_6(\{0,1\})$, then the convolutional layer will replace it by an element of $M_6(\{0,1\}^{12})$. If a pixel has type i for some i, then the corresponding entry in the new matrix will be e_i. If a pixel does not have type i for any i then the corresponding entry will be 0.

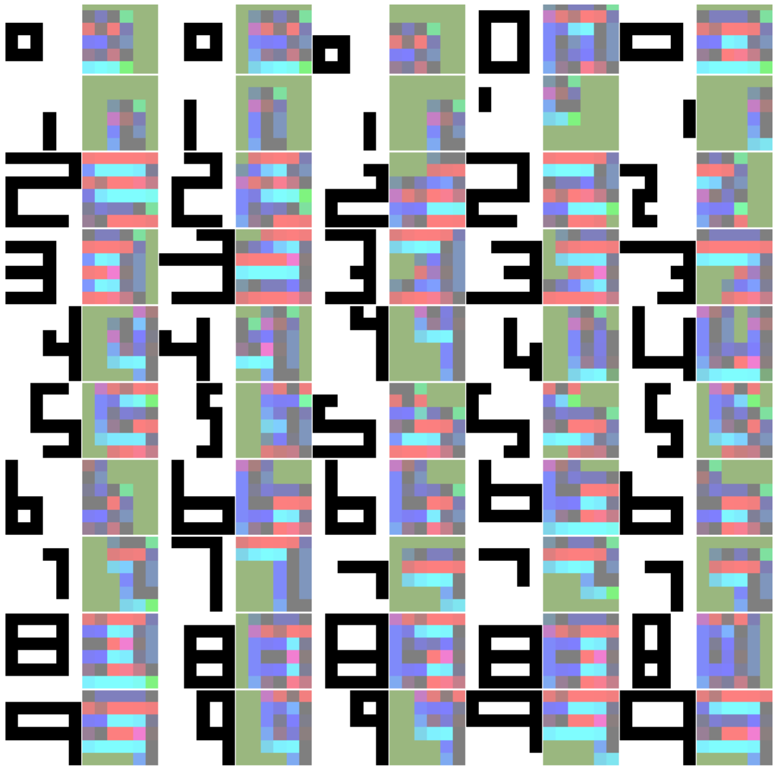
We can now take the sum over all matrix entries to count the total number of pixels of each type. The numbers for types 0 and 3 will depend on the precise geometry of the represented digit, but the numbers for the other types will not. These numbers are sufficient to distinguish all digits except for 2 and 5. To distinguish these, we need to check whether the pixels of types 6 and 8 appear above or below those of types 7 and 9, which is also easily done by a neural network layer. All this is implemented in `digits.py`. The method used there does not always reject non-digits; ideally it should be enhanced to do that.

We have also trained a neural network in the conventional way to perform the same task. We can achieve perfect performance with a smaller model than the handcrafted model described above. Specifically, we have a model consisting of a 3-channel convolutional layer with kernels of size 3×3, followed by a dense layer with 3 outputs, followed by a dense layer with 10 outputs. We train this model to predict the one-hot encoding of the relevant digit, and then apply argmax to predict the digit itself.

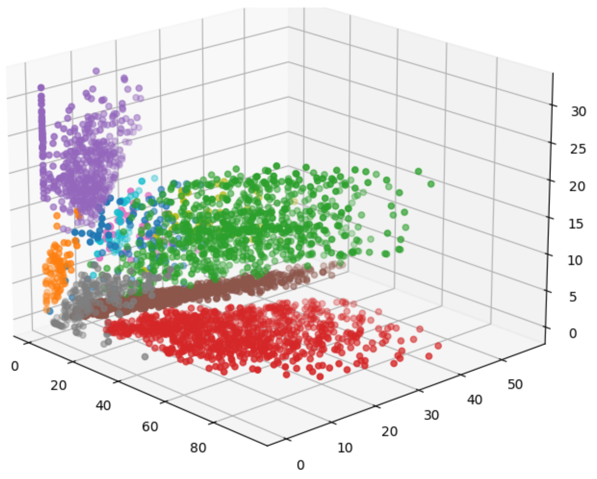
For the convolutional layer, we can interpret the three output channels as red, green and blue, so the layer has the effect of colouring the image. The kernels can be displayed as follows:



The effect on a random selection of examples is shown below.



The output of the second layer gives a point in $\mathbb{R}^3$ for each digit image. These points are best displayed using a 3D viewer which allows the diagram to be rotated interactively. The figure below shows the view from one angle (with different digits marked in different colours).



3. Noughts and crosses

We would like to train a neural network to play noughts and crosses (also known as Tick Tack Toe). Code for this is in the file `ox.py` (and also the Maple worksheet `ox.mw`).

The game is played on a 3×3 board; we number the board positions as $0, \dots, 8$ in the obvious way. We have two players: player 0 places Os on the board and player 1 places Xs on the board. We represent the game state as an array of shape 9×2, with a 1 at position (i, j) if player j has placed a counter at position i. There are 8 horizontal, vertical or diagonal lines on the board, corresponding to the following sets of positions:

$$W = \{\{0,3,6\}, \{1,4,7\}, \{2,5,8\}, \{0,1,2\}, \{3,4,5\}, \{6,7,8\}, \{0,4,8\}, \{2,4,6\}\}.$$

A player wins if they have covered at least one of these lines.

We are mostly interested in game states such that

(a) The state is empty or could be reached by a legal sequence of moves with the last move played by player 1.

(b) The game has not finished, so neither player has won, and there is at least one blank spot on the board.

Let X be the set of states of this type. We find that $|X| = 4519$. We want a function F that accepts an element of X and generates a probability distribution on the board positions. Player 0 can then sample randomly from that distribution to decide where to play. This may result in an illegal move, where the selected position is already occupied; if so we declare that player 0 has lost.

There is a well-known function F_1 which is optimal: if a player uses that algorithm then they are guaranteed to win or draw.

The most obvious approach is just to train a model to replicate the behaviour of F_1. We have successfully done this, with two hidden layers of size 20 between the input layer (of shape 9×2) and the output layer (of size 9).

We have also attempted to do two different flavours of reinforcement learning: a policy gradient approach and a deep Q-learning approach. Neither has so far been successful. It is not clear whether our implementation is flawed or whether our training hyperparameters are inappropriate.

4. Conway's Game of Life

Conway's Game of Life can be described as follows. Fix N and let X be the set of $N \times N$ matrices with entries in $\{0, 1\}$. For $A \in X$ we have $A_{ij} \in \{0, 1\}$ for $0 \leq i, j < N$, and we take A_{ij} to be 0 for all other pairs (i, j). In this extended matrix, every entry A_{ij} has 8 adjacent entries; we define $(NA)_{ij}$ to be the sum of these. We then define $(PA)_{ij}$ to be 1 if $0 \leq i, j < N$ and $((A_{ij} = 0 \text{ and } (NA)_{ij} = 3) \text{ or } (A_{ij} = 1 \text{ and } (NA)_{ij} \in \{2, 3\}))$. This defines $P\colon X \to X$, and the idea is to study the sequence $(P^n A)_{n \in \mathbb{N}}$. Here, however, we just want a convolutional neural network to calculate the function P. The first step is to define

$QA = A + 2NA$; we then have $(PA)_{ij} = 1$ iff $(QA)_{ij} \in \{5, 6, 7\}$. Also, if we put

$$f(x) = (x-4)_+ - (x-5)_+ - (x-7)_+ + (x-8)_+,$$

we find that for $x \in \mathbb{Z}$ we have $f(x) = 1$ if $x \in \{5, 6, 7\}$ and $f(x) = 0$ otherwise; so $(PA)_{ij} = f((QA)_{ij})$. This description of P can be translated in a straightforward way into a convolutional neural network, as is done in the file `life.py`.

5. Resettable running total

Example 5.1. Given an infinite sequence of natural numbers $(x_i)_{i\geq 0}$, we can define $y_0 = x_0$ and

$$y_{i+1} = \begin{cases} y_i + x_{i+1} & \text{if } x_{i+1} > 0 \\ 0 & \text{if } x_{i+1} = 0. \end{cases}$$

In other words, our output is the running total of the inputs, except that the total resets to zero whenever the input is zero. This can be computed by a Gated Recurrent Unit network (GRU), as is done in `rnn.py`.

6. Miscellaneous other examples

Example 6.1. For integer x we find that

$$\begin{aligned} (n-x)_+ - (n-1-x)_+ &= I(x \geq n) \\ 1 - (n-x)_+ + (n-1-x)_+ &= I(x < n) \\ (n+1-x)_+ - 2(n-x)_+ + (n-1-x)_+ &= I(x = n). \end{aligned}$$

It follows that these functions can be computed by 2-layer networks with RELU activation in the first layer. The same therefore holds for composites of these functions with additive functions $\mathbb{Z}^n \to \mathbb{Z}$.

It seems that standard training algorithms struggle to find these weights.

Example 6.2. Given a list $(x_0, \ldots, x_{n-1}) \in \{0,1\}^n$, we can define $y_i = p\left(x_i - \sum_{j<i} x_j\right)$. If $x = 0$ then $y = 0$. Otherwise, if i is least such that $x_i = 1$ then $y = e_i$.

Example 6.3. Suppose that $u \in \{0,1\}^n$, and define $f\colon \{0,1\}^n \to \{0,1\}$ to be the delta function concentrated at u. Put $U = \{i \mid u_i = 1\}$ and $e = (1, \ldots, 1) \in \{0,1\}^n$. We then find that

$$\begin{aligned} f(x) &= \left(1 + \sum_{i\in U}(x_i - 1) - \sum_{i\notin U} x_i\right)_+ \\ &= (1 + (x-e).u - (e-u).x)_+ = (1 - e.u + (2u-e).x)_+ . \end{aligned}$$

Thus, f can be calculated by a one-layer RELU network. This is used as an ingredient in many of the examples described later.

Suppose we wanted to train a network to calculate f. If we work in terms of the variables $2x_i - 1 \in \{1, -1\}$ then a change in u just corresponds to multiplying some weights by -1. Thus, we can restrict attention to the case $u = e$, where $f(x) = (e.x - (n-1))_+$.

Put $X = \{x \in \{0,1\}^{n+1} \mid x_0 = 1\}$ and $e' = (1, \ldots, 1) \in X$. We want to find $a \in \mathbb{R}^{n+1}$ such that $(a.x)_+ = \delta_{x,e'}$ for all $x \in X$, so we use the loss function

$$L(a) = \sum_{x\in X} ((a.x)_+ - \delta_{x,e'})^2/2.$$

Note that there are actually many solutions a with $L(a) = 0$: they are given by $a_0 = 1 - \sum_{i=1}^n a_i$ with $a_i \geq 1$ for $i = 1, \ldots, n$. Any reasonable regularisation penalty should pick out the solution where $a_0 = 1 - n$ and $a_i = 1$ for $i > 0$.

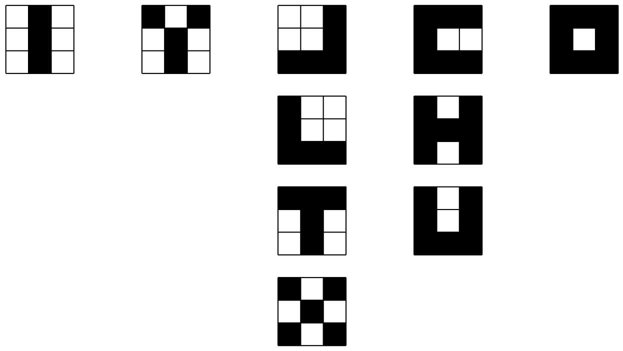
Put $X_i = \{x \in X \mid x_i = 1\}$. We find that the partial derivatives are

$$L_{,i}(a) = \sum_{x\in X_i} (a.x)_+ - H(a.e'),$$

where H is the Heaviside function. When a is close to a solution then the terms $(a.x)_+$ will be small for $x \in X_i \setminus \{e'\}$, and $H(a.e')$ will be 1, and $(a.e')_+ - 1$ will also be small, so $L_{,i}(a)$ will be small. There is code for this analysis in `maple/train_delta.mw`.

Example 6.4. It should be possible to write an explicit network to translate English to German, where the only allowed English words are "zero" and "one" (to be translated as "null" and "eins"). Text processing networks typically start with a word tokenizer, which would make the problem trivial. Instead we should assume that the input is given as a sequence of characters.

Example 6.5. The letters C,H,I,J,L,O,T,U,X,Y can be represented using as 3×3 pixel images. These are shown below, grouped according to the number of black pixels.



It should be possible to write an explicit network to recognise them, and then make a convolutional version to find them in a larger grid. Then we could perhaps try to count the number of appearances of each letter. The file `words_3x3.txt` contains 181 words that use only the letters listed here; we could also try to detect and count these words.

Example 6.6. Consider a piecewise-linear function $f\colon \mathbb{R} \to \mathbb{R}$ with finitely many corners, so there are real numbers $a_1 < \cdots < a_r$ and slopes $m_0, \ldots, m_r$ and a constant c such that

(a) $f(x) = m_0 x + c$ on $(-\infty, a_1]$
(b) $f(x)$ has slope m_i on $[a_i, a_{i+1}]$ for $i = 1, \ldots, r-1$
(c) $f(x)$ has slope m_r on $[a_r, \infty)$.

Then

$$f(x) = c + m_0 x + \sum_{i=1}^{r} (m_i - m_{i-1})(x - a_i)_+.$$

However, the situation with piecewise-linear functions in more than one variable seems unclear.