the mickey mouse guide to machine learning

$\mathbf{u} = \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix}$ is the input to the neural net. $u_i \in \{0,1,2,3,4,5,6,7,8,9\}$

the task of the neural net is to identify which of the u_i is the median. more than one u_i can be the median if there are tied values.

to train the model, the alogorithm is given samples which comprise pairs : $\mathbf{u}, \mathbf{f}$ where $\mathbf{f}$ identifies which element(s) of $\mathbf{u}$ is/are the median(s).

$$\mathbf{f} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix} \qquad f_i = \begin{cases} 1, & \text{if } u_i \text{ is median} \\ 0, & \text{if } u_i \text{ is not median} \end{cases}$$

the first layer of the neural net computes: $\mathbf{y} = \mathbf{W} * \mathbf{u} + \mathbf{c}$ and $\mathbf{z} = relu(\mathbf{y})$

$$\mathbf{W} = \begin{pmatrix} W_{00} & W_{01} & W_{02} \\ W_{10} & W_{11} & W_{12} \\ \vdots & \vdots & \vdots \\ W_{70} & W_{71} & W_{72} \end{pmatrix} \qquad \mathbf{c} = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_7 \end{pmatrix}$$

relu stands for rectified linear unit. it maps each y_i to z_i according to

$$z_i = \begin{cases} y_i, & \text{if } y_i \geq 0 \\ 0, & \text{if } y_i < 0 \end{cases}$$

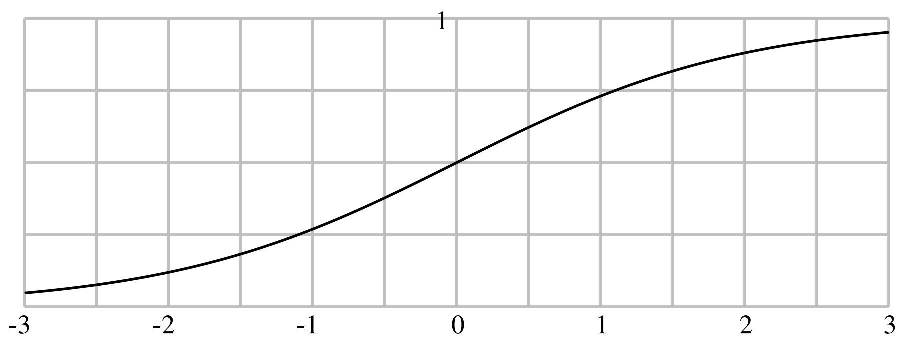
the second layer of the neural net does: $\mathbf{g} = \mathbf{M} * \mathbf{z} + \mathbf{b}$ and $\hat{\mathbf{f}} = sigmoid(\mathbf{g})$

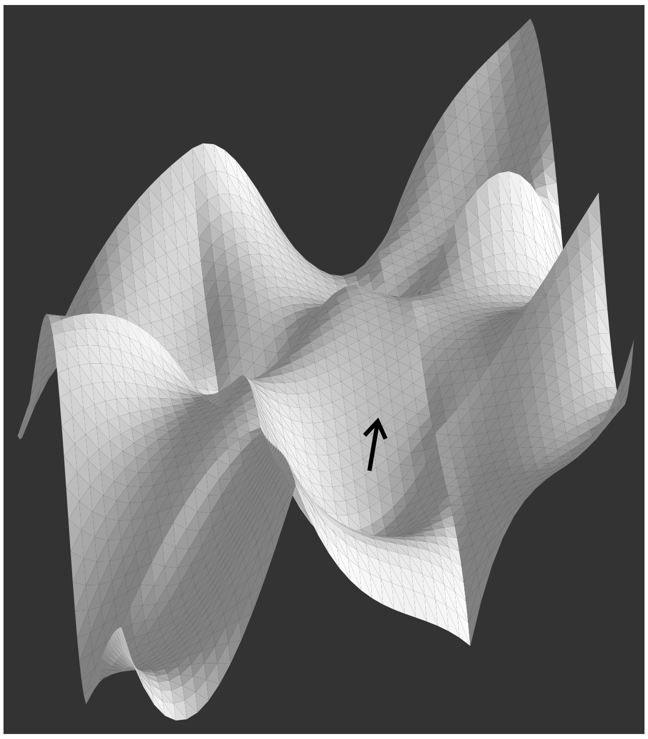
$$\mathbf{M} = \begin{pmatrix} M_{00} & M_{01} & M_{02} & M_{03} & M_{04} & M_{05} & M_{06} & M_{07} \\ M_{10} & M_{11} & M_{12} & M_{13} & M_{14} & M_{15} & M_{16} & M_{17} \\ M_{20} & M_{21} & M_{22} & M_{23} & M_{24} & M_{25} & M_{26} & M_{27} \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix}$$

$\hat{\mathbf{f}}$ is the output of the neural net. we would like

$$\hat{f}_i = \begin{cases} \text{close to 1}, & \text{when } u_i \text{ is median} \\ \text{close to 0}, & \text{when } u_i \text{ is not median} \end{cases}$$

when $\hat{\mathbf{f}}$ is the result of a training cycle, it can be thought of as the neural net estimate of $\mathbf{f}$. the sigmoid function, $\hat{f}_i = \frac{1}{(1+e^{-g_i})}$ looks like:





the figure shows a function q of two variables. the arrow shows the direction of steepest ascent at a point on the surface at the base of the arrow. the x and y components of this vector are $\nabla q = \begin{pmatrix} \frac{\partial q}{\partial x} \\ \frac{\partial q}{\partial y} \end{pmatrix}$. when x,y change in the direction ∇q, the corresponding point on the surface goes uphill. one way to find a minimum of q is to start somewhere on the surface and go downhill. this can be done by repeatedly taking steps in the x-y plane of $-\epsilon \nabla q$ where ϵ is a small +VE constant. this is called gradient descent. the gradient descent method generalises to higher dimensions.

the loss function L measures how well/badly the output $\hat{\mathbf{f}}$ approximates to $\mathbf{f}$ for a given $\mathbf{u}$. in this case $L = \frac{1}{3}\sum(\hat{f}_i - f_i)^2$. for a particular training cycle, $\mathbf{u}$ and $\mathbf{f}$ are fixed so L becomes a function of all the parameters W_{ij}, c_i, M_{ij}, b_i (59 of them in all). think of L as a 59-dimensional surface. it is convenient to treat all these variables as a single vector $\mathbf{\Theta}$. the training bit is taking a small step $-\epsilon \nabla L(\Theta)$ in the parameter space (i.e. gradient descent) so that the loss L decreases for the given $\mathbf{u}$ (and perhaps other inputs in the neighbourhood of $\mathbf{u}$). note that this is not the same as seeking the minimum of a function. at the next training cycle most likely there will be a different $\mathbf{u}, \mathbf{f}$ so the gradient descent is happening on a different surface. machine learning is about minimising the loss across all possible inputs. if the set of possible inputs were known and finite then in principle one could construct a meta-loss function that sums the losses for all inputs and then find Θ that gives the smallest meta-loss. this is rarely possible or practical so instead gradient descent is used to tweak the parameters for lots of input samples. machine learning is akin to a tailor making many small adjustments to a suit until it nicely fits the body within. it is important that the training set is representative of the set of all possible inputs.

$\nabla L(\Theta)$ is calculated in stages. ∇L with respect to $\hat{\mathbf{f}}$ is

$$\begin{pmatrix} \frac{\partial L}{\partial \hat{f}_0} \\ \frac{\partial L}{\partial \hat{f}_1} \\ \frac{\partial L}{\partial \hat{f}_2} \end{pmatrix} = \frac{1}{3}\begin{pmatrix} 2(\hat{f}_0 - f_0) \\ 2(\hat{f}_1 - f_1) \\ 2(\hat{f}_2 - f_2) \end{pmatrix}$$

the jacobian matrix $\mathbf{J}\hat{\mathbf{f}}$ with respect to Θ is

$$\begin{pmatrix} \frac{\partial \hat{f}_0}{\partial W_{00}} & \cdots & \frac{\partial \hat{f}_0}{\partial b_2} \\ \frac{\partial \hat{f}_1}{\partial W_{00}} & \cdots & \frac{\partial \hat{f}_1}{\partial b_2} \\ \frac{\partial \hat{f}_2}{\partial W_{00}} & \cdots & \frac{\partial \hat{f}_2}{\partial b_2} \end{pmatrix} \quad \text{and} \quad ((\nabla L(\Theta))^T = (\nabla L(\hat{\mathbf{f}}))^T * \mathbf{J}\hat{\mathbf{f}}$$

this looks untidy but $(\nabla L(\Theta))^T$ is just $\nabla L(\Theta)$ converted to a row vector. $\mathbf{J}\hat{\mathbf{f}}$ is in turn derived in steps. $\hat{f}_i = \frac{1}{(1+e^{-g_i(\Theta)})}$ so $\frac{\partial \hat{f}_i}{\partial \theta_k}$ (the i,k th member of $\mathbf{J}\hat{\mathbf{f}}$) $= \frac{w}{(1+w)^2}\frac{\partial g_i(\Theta)}{\partial \theta_k}$ where $w = e^{-g_i}$.

$\frac{\partial g_i(\Theta)}{\partial \theta_k}$ is the i,k th element of $\mathbf{Jg}(\Theta)$. the chain rule of calculus is applied again and again until one arrives at something trivial to differentiate such as $y_i = \sum W_{ij} u_j + c_i$. then it is just a matter of multiplying matrices to get the numbers for $\nabla L(\Theta)$ at the current value of Θ (which defines the state of the neural net). next Θ is adjusted by $-\epsilon \nabla L(\Theta)$ and the modified model is ready to go for the next training sample. choosing a good value for ϵ is something of a black art.

after-thought: the median-of-three function has symmetry. for every input of the kind $(1, 2, 3)^T$ there is a counterpart $(3, 2, 1)^T$. this suggests that $\hat{f}_2$ behaves like a "mirror image" of $\hat{f}_0$ and that a neural net with fewer parameters can do the job. however the maths of the machine learning with symmetry conditions becomes messier.