Jack's Machine Learning Notes

Jack Fraser-Govil

May 18, 2023

1 Perceptrons

The first 'Machine Learning' tool was the *perceptron*, which is a form of a binary classifier: given a vector (representing 'features' of a dataset), it could assign either a 0 or a 1 to it, indicating membership of one group or another.

The perceptron algorithm is highly simple:

$$\mathcal{P}_{b,\mathbf{w}}(\mathbf{x}) = \text{Perceptron}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}\cdot\mathbf{x} + b > 0 \\ 0 & \text{else} \end{cases} \tag{1}$$

This, in essence, splits the d-dimensional domain into two by defining a hyperplane ($d-1$ dimensional): those which lie one one side of the hyperplane fall into one category, those which lie on the other side are not in that category.

1.1 Biases & Dimensionality

Note that, strictly speaking, we do not need the 'bias' term, b – the vector $\mathbf{w}$ is sufficent to define the hyperplane. In fact, as a general statement you only require d parameters to split the d-dimensional plane into two.

Indeed, we see that including b makes the solution degenerate, since $(\alpha\mathbf{w})\cdot\mathbf{x} + \alpha b$ gives identical classifications for all $\alpha > 0$ since our classifications are based only on when this value is greater than 0. We could therefore divide all of our solutions by b and simply look for when the perceptron value was greater than or less than 1. So why is the 'bias' term b included?

The answer is twofold. Firstly, keeping the b separate makes the convergence of the algorithm quicker since it (by comparison with the 2D case $y = mx + b$) allows us to move the hyperplane 'vertically' in space, without altering the gradient of the plane: if only $\mathbf{w}$ was used, simple translations would require simultaneous alterations of all values in the vector.

Secondly, it also prevents the numerical issues which are encountered when (using the 2D case as an example), $m \to \infty$ for a vertical line – using $\mathbf{w}$ and b instead of just m and c does not require the infinity. Thus allowing for degenerate solutions has its advantages.

1.2 Training

'Training' the perceptron amounts to finding the value of $\mathbf{w}$ and b (for simplicity, we will refer to this combination as $\tilde{\mathbf{w}} = [b, \mathbf{w}]$, where $\tilde{\mathbf{x}} = [1, \mathbf{x}]$) which produces the smallest number of errors.

For each datum j in the training dataset, where $\mathbf{x}_j$ is the features and d_j is the (correct) label, the weights are updated as follows:

$$\tilde{\mathbf{w}} \to \tilde{\mathbf{w}} + r\left(d_j - \mathcal{P}_{\tilde{\mathbf{w}}}(\mathbf{x}_j)\right)\tilde{\mathbf{x}} \tag{2}$$

Where r is the 'learning rate'. In short: whenever the Perceptron makes a mistake ($d_j \neq \mathcal{P}$), it updates its vectors slightly in the direction which makes it less-incorrect.

When an optimal solution exists (see later), this algorithm is guaranteed to converge to the global optimum, though the choice of the learning rate r can affect how rapidly that convergence occurs.

1.3 Extensions & Limitations

In the simple case of a 2D classifier, the naive perceptron is limited to simply drawing straight lines ('decision boundaries') through the data - a clear limitation of the method which extends to arbitrarily high dimensions. You can, however, get a bit cleverer.

Imagine that, in our 2D case, you have some which looks like it has a quadratic decision boundary. Rather than feeding in (x, y), you feed in (x, x^2, y) into a new 3D perceptron, such that you are comparing $w_0 y + w_1 x + w_2 x^2 + b >?0$. This model **is** capable of learning the quadratic decision boundary – however, you had to feed in the non-linearity manually. This can extend to arbitrarily complex models – even if the data is only 2D in reality, you could feed in the vector $\mathbf{x} = (x, y, \sin(x), \cos^2(x), xy, x^9.04e^y\ldots)$, which would allow the model to tune to arbitrary decision boundaries – those familiar with basis functions might suggest that something along the lines of $(y, x, x^2 - 1, x^3 - 3x, x^4 06x^2 + 3)$ (i.e. the Hermite polynomials) would be an appropriate choice, since – when enough dimensions are included – they can arbitrarily approximate any function.

However, the Perceptron has a fundamental flaw, insofar that, no matter how clever you get with transforming your input data, it remains a *linear* classifier: the operation $\tilde{\mathbf{w}} \cdot \tilde{\mathbf{x}}$ amounts to splitting your projected space in two: if the data is not "linearly separable", then the Perceptron will only be able to make crude approximations.

A simple example of a function which is not linearly separable is the XOR operation: a famous 1969 book (Minsky & Papert) proved this result definitively, which led to a pause in the development of machine learning for more than 10 years.

2 Feedforward Neural Networks

The failure of peceptrons was twofold: the was only a single neuron (and hence a single division of the plane), and they were discontinuous (they were either right or wrong, they couldn't move smoothly between these options). The Multi-Layer Perceptron solves this problem by chaining together several modified perceptrons, and has three components:

- **Input Layer**: performs preprocessing on the input, transforming it into (usually) a 1D feature vector
- **Hidden Layers**: several layers of perceptron 'neurons'
- **Output Layers**: The final layer which makes a prediction of the desired output

Each neuron in each layer acts as a Perceptron, with the exception that they use a (usually) continuous activation function, ϕ, instead of the Heaviside function the original perceptron used. The output of each neuron is a single real number - which is then assembled into a 'layer vector', and passed on to the next layer.

Each neuron i at layer d can also (in the general case) have their own activation function ϕ_d^i, though in practice it would be normal to have the activation function be the same across the entire layer, $\phi_d^i = \phi_d$.

The algorithm for passing the information forward through the network is as follows:

$$[\mathbf{x}_i]_j = \phi_i^j(\underbrace{\tilde{\mathbf{w}}_{i,j} \cdot \tilde{\mathbf{x}}_{i-1}}_{y_i^j}) \tag{3}$$

It is therefore clear that each layer d produces a vector $\mathbf{x}_d$ which has size equal to the number of neurons in the layer, N_d, and therefore that each neuron must be equipped a vector $\tilde{\mathbf{w}}_d^i$ of length $N_{d-1} + 1$.

If ϕ is chosen to be a linear function of the input, then this entire endeavour collapses back into the simple perceptron case: the problem reduces to a single matrix multiplication $\mathbf{x}_n = W\tilde{\mathbf{x}}_0$, which suffers from all of the previously discussed linear problems. If, however, the function is non-linear, then the network does not simplify, and is capable of fitting itself to non-linear problems: the more nodes and more layers that the network possesses, the higher degrees of non-linearity which can be accounted for, and hence the better able to fit the data the network becomes. Common choices for non-linear ϕs include:

- The Rectified Linear Unit (ReLU) function:

$$\phi_{\text{relu}}(x) = \begin{cases} x & x > 0 \\ 0 & \text{else} \end{cases}$$

- The sigmoid/logistic function:

$$\phi_{\text{sigmoid}}(x) = \frac{1}{1 + \exp(-x)}$$

- The Leaky-ReLU function:

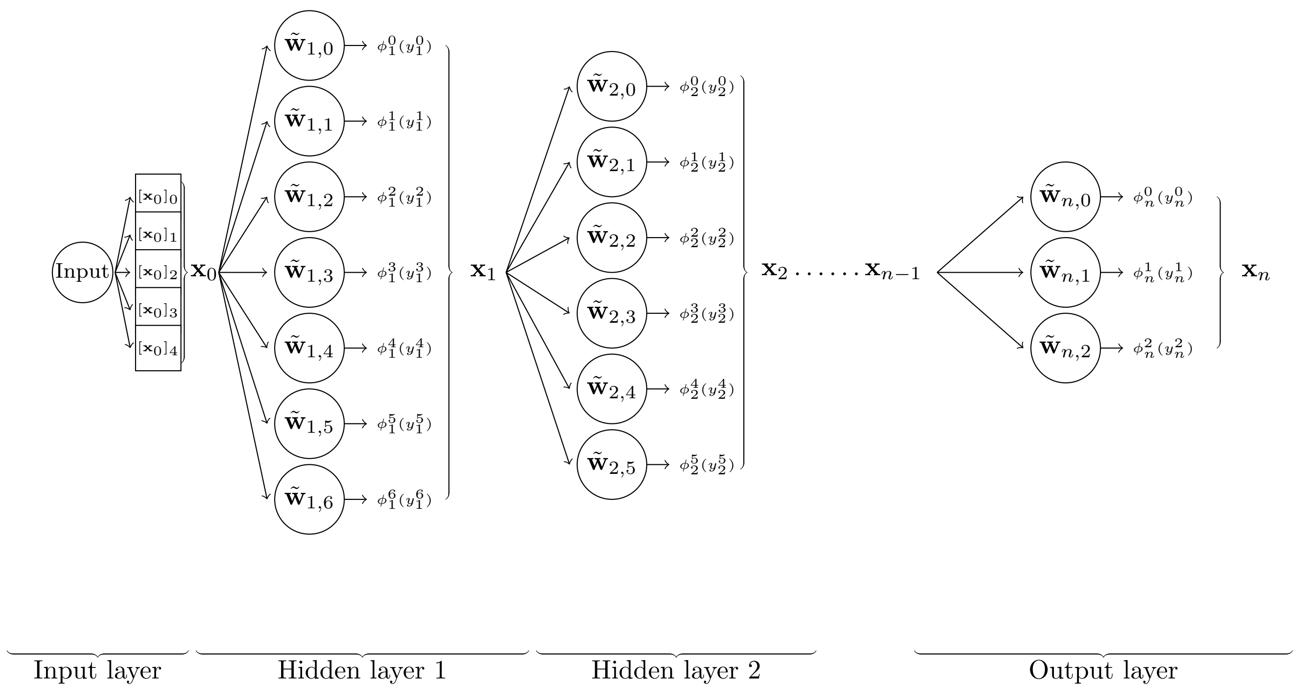
$$\phi_{\text{leaky}}(x) = \begin{cases} x & x > 0 \\ 0.01x & \text{else} \end{cases}$$

These have various advantages (sigmoid maps between 0 and 1, so is good for output layers which have to make probability decisions, but has vanishing gradients for $|x| \gg 0$ – the ReLU and Leaky-ReLU avoid this, but have non-smooth derivatives which can cause problems with convergence).

However, in becoming non-linear, the network loses any ability to be analytically optimisable: instead we must numerically optimise it, by using i.e. gradient descent methods to find the values of $\{\mathbf{w}_d^i\}$ which produce the greatest fits to the data.

2.1 Training an MLP

The act of training a Multi-Layer Perceptron is therefore equivalent to an optimisation problem, where the


Input
$[\mathbf{x}_0]_0$
$[\mathbf{x}_0]_1$
$[\mathbf{x}_0]_2$
$[\mathbf{x}_0]_3$
$[\mathbf{x}_0]_4$
$\mathbf{x}_0$
$\tilde{\mathbf{w}}_{1,0}$ … $\tilde{\mathbf{w}}_{1,6}$
$\phi_1^0(y_1^0)$ … $\phi_1^6(y_1^6)$
$\mathbf{x}_1$
$\tilde{\mathbf{w}}_{2,0}$ … $\tilde{\mathbf{w}}_{2,5}$
$\phi_2^0(y_2^0)$ … $\phi_2^5(y_2^5)$
$\mathbf{x}_2 \ldots\ldots \mathbf{x}_{n-1}$
$\tilde{\mathbf{w}}_{n,0}$, $\tilde{\mathbf{w}}_{n,1}$, $\tilde{\mathbf{w}}_{n,2}$
$\phi_n^0(y_n^0)$, $\phi_n^1(y_n^1)$, $\phi_n^2(y_n^2)$
$\mathbf{x}_n$
Input layer
Hidden layer 1
Hidden layer 2
Output layer

function to be minimised is the so-called 'loss function': the error rate of the optimiser measured over a training set of data. The simplest case is a standard L_2-norm error rate:

$$\begin{aligned}\mathcal{L} &= \sum_{\text{data, } j} 0.5\left(\mathcal{M}(\mathbf{x}_j) - \mathbf{d}_j\right)^2 + R(\mathcal{M}(\mathbf{x}_j)) \\ &= \sum_{\text{data, } j} \mathcal{L}_j\end{aligned} \tag{4}$$

Here the function $\mathcal{M}(\mathbf{x}_j)$ is shorthand for the processing of the MLP: $\mathbf{x}_j$ is set to $\mathbf{x}_0$ and passed through the network, such that $\mathcal{M}(\mathbf{x}_j) = \mathbf{x}_n(j)$. The function $R(\mathbf{x})$ is a *regularisation function*, and is used as a prior to prevent the optimisation from running away into territory which can kill nodes or prevent optimisation into the desired region: it can also prevent overfitting by preferring worse, but simpler, fits.

Each epoch of optimisation therefore relies on our ability to compute the derivative of $\mathcal{L}$ with respect to the parameters - that being each of the (scalar) values $w_{ijk} = [\tilde{\mathbf{w}}_i^j]_k$ - the kth element of the weight vector in the jth node of the ith layer - simple gradient descent would then tell us to update w_{ijk} according to:

$$w_{ijk} \to w_{ijk} - \alpha \frac{\partial \mathcal{L}}{\partial w_{ijk}} \tag{5}$$

More complex optimisation routines use diffeent update rules - but all first-order methods will rely simply on being able to compute $\frac{\partial \mathcal{L}}{\partial w_{ijk}}$ (and usually $\mathcal{L}$ itself).

Recalling that $y_i^j(q) = \tilde{\mathbf{w}}_{i,j} \cdot \tilde{\mathbf{x}}_{i-1}(q)$, with $\mathbf{x}_0(q)$ being the q^{th} input vector, and $[\mathbf{x}_i]_j = \phi_i^j(y_i^j(q))$, we see that:

$$\frac{\partial \mathcal{L}_q}{\partial w_{ijk}} = \left.\frac{\partial \mathcal{L}_q}{\partial y_i^j}\right|_{y_i^j = y_i^j(q)} [\tilde{\mathbf{x}}_{i-1}(q)]_k \tag{6}$$

This is easy to compute for the output layer, since:

$$\frac{\partial \mathcal{L}_q}{\partial y_n^j} = \left[(\mathbf{x}_n(q) - \mathbf{d}_q) + \left.\frac{\partial R}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_n(q)}\right] \cdot \hat{e}_j \times \phi_n^{j,\prime}(y_n^j) \tag{7}$$

Where $\phi'(x)$ is the derivative of the relevant activation function. The derivative for each of the inner layers can then be computed by considering the chain rule:

$$\frac{\partial \mathcal{L}_q}{\partial y_i^j} = \sum_{p \in L_{i+1}} \frac{\partial y_{i+1}^p}{\partial y_i^j} \frac{\partial \mathcal{L}_q}{\partial y_{i+1}^p} \tag{8}$$

Here the sum runs over the nodes in the next layer; i.e. the change of $\mathcal{L}$ with respect to the y in one layer can be computed as a weighted sum of the change of the ys in the next layer, and the change those ys have on $\mathcal{L}$: by working backwards from the final layer, we can therefore fill in each $\frac{\partial \mathcal{L}}{\partial y_i^j}$ sequentially. The y

derivatives can be computed since:

$$y_{i+1}^j = \sum_k w_{i+1,jk}\phi_i^k(y_i^k) \iff \frac{\partial y_{i+1}^p}{\partial y_i^j} = \phi_i^{j,\prime}(y_i^j) w_{i+1,pj} \tag{9}$$

Hence:

$$\frac{\partial \mathcal{L}_q}{\partial y_i^j} = \phi_i^{j,\prime}(y_i^j) \sum_{p\in L_{i+1}} w_{i+1,pj} \frac{\partial \mathcal{L}_q}{\partial y_{i+1}^p} \tag{10}$$

Since the derivative infomation travels backwards through the network, this is termed 'backpropagation' - though in reality this is simply a ML-jargon term for a recurrence relationship.

3 Convolutional Neural Networks

FNN work well on simple classification tasks - however, things get more complicated when we wish to do things with image recognition and processing. Consider that we wish to process a 100x100 pixel image and classify it into one of three categories using 4 layers, with (20,10,5,3) - even this simple network has in excess of 200,000 free parameters that need training - driven by the fact that our 100x100 image flattened out into a vector of size 100^2. In doing so, we also discard any form of spatial correlation data: image pixels are often highly correlated, with surrounding pixels being similar – information which is lost when the image is naively flattened.

Convolutional Layers are an attempt to rectify these problems, by replacing any number of the layers with a 'convolutional layer', in which a convolution/correlation kernel is passed over the input vector (or, more accurately, tensor). For simplicity's sake, we will assume that the input data for our convolutional layer is a square image of size W pixels: i.e. a 2D matrix with C (corresponding to multiple input channels - i.e. RGB) values between 0 and 255.

Rather than splitting the layer into a number of distinct neurons, each of which has their own set of weights $\tilde{\mathbf{w}}$, a convolutional layer is split up into a number, F, of *filters*. Each filter contains C mask functions of size W, corresponding to the number of input channels - in the first layer this might be 1 (for monochrome images) or 3 (for RGB images).

The layer then takes the input vector $\mathbf{x}$ (which has dimension Cn^2), and performs a pad/folding operation, $\hat{F}_p$ which translates this vector into C $(n+2p)\times(n+2p)$ matrices - the additional $2p$ elements in each dimension are set to provide a zero-padding of distance p around the original image:

$$M_c = \hat{F}_{c,p}\mathbf{x} \iff M_c = \begin{pmatrix} 0_{p,p} & 0_{,p}n & 0_{p,p} \\ 0_{n,p} & \hat{X}_f & 0_{n,p} \\ 0_{p,p} & 0_{,p}n & 0_{p,p} \end{pmatrix} \tag{11}$$

The chosen 'folding' method is (assuming zero-indexing) to use:

$$(M_c)_{ij} = \begin{cases} 0 & \text{if } i,j < p \text{ or } i,j \geq n+p \\ (\mathbf{x})_{cn^2+(i-p)n+j-p} & \text{else} \end{cases} \tag{12}$$

Each filter is equipped with C mask functions, which are themselves $W\times W$ matrices, denoted Q^{fc} - these are the weights for each filter. Each filter has CW^2 parameters, giving the entire layer FCW^2 parameters. Each filter is then split up into a number of 'output pixels' (similar to neurons - but rather than acting on the entire dataset, they act on a limited portion of it):

$$Y_{ij} = \sum_{c=0}^{C}\sum_{a=0}^{W}\sum_{b=0}^{W} (Q^{fc})_{ab}(M_f)_{i+a,j+b} \tag{13}$$

That is, for each input channel, the mask takes a weighted sum of all pixels within a distance $W/2$ - providing the spatial correlation we are searching for - before then performing a weighted sum over all of the input channels (with the weights being determined by the convlution itself). Each value of Y_{ij} is then passed through an activation function, to produce the 'activation map':

$$X_{ij} = \phi(Y_{ij}) \tag{14}$$

Since each layer has F filters, the output of the layer is an F-channel square image with dimension $n+2p-W+1$. This information is then unspooled back into a vector $\mathbf{x}_{i+1}$, which now has length $F(n+2p-W+1)^2$.

3.1 Pooling Layers

Due to the necessity of padding the image (to prevent under-weighting features near the edge), the convolutional layers can often cause the images to grow in size - increasing the number of neurons needed in the next layer - whilst not adding any new information. This is the exact opposite of what the convolutional layer was supposed to do - which was to concentrate and compile spatial information into fewer dimensions than raw pixel counts.

To deal with this, convolution layers often pass their output through a pooling layer: a square mask of size q which passes over the activation map X with a stride equal to the mask size, performing some form of simplification operation: common choices are simply taking the maximum value of the pixels, or the mean. If the pooling mask is of size 2×2, the stride means that the image size gets halved - a $n \times n$ image gets reduced to $\frac{n}{2} \times \frac{n}{2}$ - hopefully without losing much relevant information.

Written analytically, the two most common pooling functions are:

$$\begin{aligned}(\text{MaxPool}_q(X))_{ij} &= \max_{a,b=0\to q-1}\left(\{X_{iq+a,jq+b}\}\right) \\ (\text{MeanPool}_q(X))_{ij} &= \frac{1}{q^2}\sum_{a,b=0}^{q}\left(X_{iq+a,jq+b}\right)\end{aligned} \tag{15}$$

The derivatives of these functions are then:

$$\begin{aligned}\frac{\partial\left(\text{Max}_q(X)\right)_{ij}}{\partial X_{ab}} &= \begin{cases}1 & \text{if } a,b \text{ in pool and } (\text{Max}_q(X))_{ij} = X_{ab} \\ 0 & \text{else}\end{cases} \\ \frac{\partial\left(\text{Mean}_q(X)\right)_{ij}}{\partial X_{ab}} &= \begin{cases}\frac{1}{q^2} & \text{if } a,b \text{ in pool} \\ 0 & \text{else}\end{cases}\end{aligned} \tag{16}$$

Here 'in pool' means that X_{ab} is one of the consituent elements of $\text{Pool}_q(X)_{ij}$ - i.e. ab lies within a distance q of ij.

3.2 Training Convolutional Neural Networks

Since CNNs combine Convolutional Layers, Pooling layers and standard FNN layers, they add an additional level of complexity into the training process - since each layer must now be aware of the 'type' of its surrounding layers, in order to properly compute the relevant derivatives.

The equivalent of Eq. (6) is:

$$\frac{\partial\mathcal{L}_q}{\partial(Q^{\ell fc})_{ij}} = \left.\frac{\partial\mathcal{L}_q}{\partial Y^{\ell f}_{ij}}\right|_{Y^{\ell f}_{ij} = Y^{\ell,f}_{ij}(q)} \sum_{a,b}(M_c)_{i+a,j+b} \tag{17}$$

This is indicative that although we have fewer parameters for each filter, each parameter is connected to multiple inputs - hence the sum running over M_c.

We can then use the same logic as in the FNN case to write $\frac{\partial\mathcal{L}_q}{\partial Y^{\ell f}_{ij}}$ via the chain rule – assuming it passes through an activation function $\phi_\ell(Y)$ and a pooling function $\text{Pool}_q(X)$ to produce the vector used in the next layer:

$$\frac{\partial\mathcal{L}_q}{\partial Y^{\ell f}_{ij}} = \phi'_\ell(Y^{\ell f}_{ij})\left.\frac{\partial \text{Pool}^{fmn}_q}{\partial X}\right|_{X=\phi(Y^{\ell f}_{ij})}\frac{\partial\mathcal{L}}{\partial x_{fnm}} \tag{18}$$

Here we have chosen the pooling function such that it only acts on the output pixel which is 'in pool' for the ij input pixel (else we would have to sum over all output pixels until the derivative was non-zero): in short: the rate of change of $\mathcal{L}$ w.r.t. the feature pixel Y_{ij} is equal to the product of the activation derivative (as always), the pooling derivative (i.e. the extent to which this pixel features in the next layer), and the rate of change of $\mathcal{L}$ w.r.t. the input into the next layer – hence this is still computed through a backpropagation operation.

We already know how to compute $\frac{\partial\mathcal{L}}{\partial x_{fnm}}$ if the next layer is a FNN layer – Eq. (10) – we therefore merely need to compute how a CNN layer changes to an incremental change to their input.

$$\frac{\partial\mathcal{L}_q}{\partial X^{\ell c}_{wv}} = \sum_f\sum_{i,j}\begin{cases}Q^{(\ell+1)fc}_{ab}\frac{\partial\mathcal{L}_q}{\partial Y^{\ell+1 f}_{ij}} & \text{if } w,v \text{ affected } i,j \\ 0 & \text{else}\end{cases} \tag{19}$$

I.e., for each output pixel Y_{ij} the input pixel X_{cwv} either entered into the sum (with a prefactor of Q), or it did not, because wv and ij were too far away. We therefore must simply sum over all pixels ij which were influenced by wv, and select the appropriate filter member, Q_{ab}, for each – this is a fiddly operation (indices must be tracked appropriately), but it is not particularly conceptually difficult – just easy to lose track of which pixels affect which!