DeepNeuralNetwork

Model

- The network takes inputs $\boldsymbol{x}_n$ and gives outputs z_n (assume 1D output and can generalise)
- The training data is labelled y_n
- For a single neuron the activation is as below

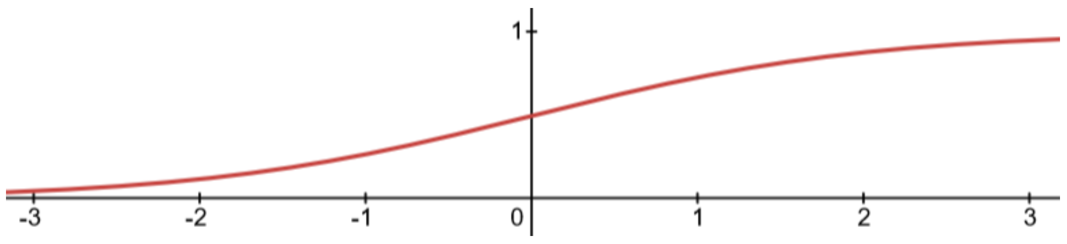
$$z_n = \sigma\left(\sum_k x_{k,n} w_k + b\right)$$

- We describe the layers of neurons as matrices

$$z_n = \boldsymbol{w}\boldsymbol{x}_n + \boldsymbol{b}$$

- The weight matrix must be (output rows, input columns)
- The input vector must be (input rows, 1)
- The output vector must be (output rows, 1)
- The sigmoid function is

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



Log-likelihood maximisation

- Our neural network has unknown parameters (weights and biases) that we'd like to estimate
- The likelihood is the probability of the observed data (if the data is discrete) or the probability density function (if the data is continuous)
- The maximum likelihood estimator is the parameter value (the set of weights and biases) that maximises the likelihood
- We use $\widehat{\boldsymbol{p}}$ to refer to the top/best parameter value (as the hat makes it look like the parameters are at the top of a mountain)
- Instead of maximising likelihood we often maximise log likelihood as logarithms simplify the differentiation and are increasing functions to $\widehat{\boldsymbol{p}}$ changes
- If the data consists of many datapoints that are all independent then the likelihood of the dataset is the product of all the likelihoods of the individual datapoints

Training

- We want to minimise the loss function for a given set of weights
- This is equivalent to maximising the likelihood of the training dataset, or in this case the log likelihood
- The likelihood of a single data item as given by our probability model is below, where we assume the model is predicting $M(\boldsymbol{x})$ which is the parameter in a Bernoulli distribution

$$P(y|\boldsymbol{x}) = \begin{cases} M(\boldsymbol{x}) & if\ y = 1 \\ 1 - M(\boldsymbol{x}) & if\ y = 0 \end{cases} = [1 - M(\boldsymbol{x})]^{1-y}[M(\boldsymbol{x})]^{y}$$

- The likelihood of the entire dataset is given below

$$P[\{y_n\}_{n=1}^{N}|\{\boldsymbol{x}_n\}_{n=1}^{N}] = \prod_{n=1}^{N}[1 - M(\boldsymbol{x}_n)]^{1-y_n}[M(\boldsymbol{x}_n)]^{y_n}$$

- The negative log likelihood to be minimised can be given as below, noting that $M(\boldsymbol{x}) = z_n$

$$L = -\sum_{n}[y_n \log(z_n) + (1 - y_n)\log(1 - z_n)]$$

- We need to find the parameters (weights and biases) that maximise this loss function as these would
- If we were doing regression with a Gaussian error (not classification) then the best loss function would be mean square error
- For the next 2 lines only, let $y{\sim}f(\boldsymbol{x}) + N(0, \sigma^2)$

$$P[\{y_n\}_{n=1}^{N}|\{\boldsymbol{x}_n\}_{n=1}^{N}] = \prod_{n=1}^{N}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\left[\frac{y_n - M(\boldsymbol{x})}{2\sigma}\right]^2}$$

$$L = -\sum_{n}\left[-\frac{1}{2}\ln(2\pi\sigma^2) - \frac{1}{2\sigma}[y_n - M(\boldsymbol{x})]^2\right]$$

- Moving back to our Bernoulli classification problem, we now differentiate the loss with respect to the various weights to allow us to minimise it via gradient descent

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \eta\frac{\partial L}{\partial \boldsymbol{w}}$$

- For a single neuron we can differentiate the loss easily

$$\frac{\partial L}{\partial w_k} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_k}$$

$$L = y\ln(z) + (1 - y)\ln(1 - z)$$

$$x = \sigma(w_0 x_0 + w_1 x_1 + \cdots + w_n x_n) = \sigma(a)$$

$$\frac{\partial L}{\partial z} = \frac{y}{z} + (1 - y)\left(\frac{-1}{1 - z}\right) = \frac{y(1 - z) + (1 - z)(-z)}{z(1 - z)} = \frac{y - z}{z(1 - z)}$$

$$\frac{\partial z}{\partial w_k} = \frac{\partial z}{\partial a}\frac{\partial a}{\partial w_k} = \sigma'(a)x_k$$

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

$$\sigma'(a) = -(-e^{-a})(1+e^{-a})^{-2} = e^{-a}[\sigma(a)]^2 = \left[\frac{1-\sigma(a)}{\sigma(a)}\right][\sigma(a)]^2 = \sigma(a)[1-\sigma(a)] = z(1-z)$$

$$\frac{\partial L}{\partial w_k} = \frac{y-z}{z(1-z)} z(1-z)x_k = (y-z)x_k$$

- The step to take to minimise L can be vectorised below

$$\frac{\partial L}{\partial \boldsymbol{w}} = (y-z)\boldsymbol{x}$$

- For a deeper network (output z, hidden layer activations z_k, input x_j) we use the chain rule

$$\frac{\partial L}{\partial w_k} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_k} = \frac{y-z}{z(1-z)} z(1-z)z_k = (y-z)z_k$$

$$\frac{\partial L}{\partial w_{jk}} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial z_k}\frac{\partial z_k}{\partial w_{jk}} = \frac{y-z}{z(1-z)} z(1-z)w_k z_k(1-z_k)x_{jk} = (y-z)z_k w_k(1-z_k)x_{jk}$$

- A library such as pytorch or TensorFlow would use automatic differentiation to keep track of the derivatives so rather painful maths like this can be avoided

Training considerations

- Overfitting
  - Where the model learns noise in the training data, so starts to fit the test data less well
  - Early stopping where we stop training the model when the validation data accuracy starts to increase
  - Regulariser where we penalise large weights
  - Dropout where we set 50% of the neurons to zero every time the network learns to build many different resilient pathways so the network does not rely on unique training set features
- Training speed
  - Choose the learning rate so it is small enough to prevent divergence but large enough to learn quickly
  - Batch the data so we don't compute the gradient over the entire dataset and we still get sufficient data to avoid the noise from SGD
  - Normalise the inputs so they are around 0 and 1 to make sure the initial weights are good sizes and that the inputs initially are equally important
- Activation functions
  - Using sigmoid makes the gradients at the start of a network very small (they vanish) so we use Relu instead to provide non-linearity
  - The non-linearity of an activation function means that we can predict non-linear outputs – otherwise the network would behave as a single neuron regardless of the size of the dataset