Deep Learning – Neural Nets

The McCulloch-Pitts Perceptron, 1943

- "Perception" and "Electronic"
- A very simplified model of how a real neuron works
- "Perceptron" and "Neuron" used almost interchangably,
  but Perceptron is a little bit old fashioned now
- Any number of inputs, a_i
  Sometimes on/off, sometimes continuous
- Each input has an associated weight, w_i
  Changing the weights is how it learns
- Output or State computed from weighted inputs
  $\sum a_i \times w_i$
- But a non-linear Activation Function is applied to that sum
  $f(\sum a_i \times w_i)$
- Could be (but usually isn't) a cut-off function
  $f(x)$ = if $x < k$ then 0 else 1
  each perceptron can have its own value for k
- Or the Sigmoid (or Logistic) function, popular
  $f(x) = 1 / (1 + e^{-x})$
  easy to differentiate, that will turn out to be useful
- Or the Rectified Linear Unit (ReLU) function
  $f(x) = \max(0, x)$
- Or the Softplus function, a continuous approximation to ReLU
  $f(x) = \log_e(1 + e^x)$
- Or the Hyperbolic Tangent function tanh
  $f(x) = (e^{2x} - 1) / (e^{2x} + 1)$
  but that gives a range of -1 to +1
- There can also be a Bias
  Typically a constant value -1 or +1
  that is treated as an extra input with its own weight
  it can shift the curve given by f to the left or right

Layers

- One perceptron recognizes a straight-line/plane/hyperplane
  Separating one set of possible inputs from the other
- Originally arranged in a single layer
  Usually all receiving the same inputs from sensors
  And all recognizing different Features
- But that can't recognize such a simple thing as exclusive or
- Now usually arranged in multiple layers
  This is what Deep Learning refers to
  There is an input layer and an output layer,
  any others are called Hidden Layers
- Just two layers can compute any continuous function
  As close as you like, just add more perceptrons

- Maybe softmax for final outputs when classifying
  $e^{thisoutput} / \sum e^{alloutputs}$
  Suppose the actual outputs are 0.3, -0.4, 1.2, 0.2
  the softmaxes are 0.206, 0.102, 0.505, 0.186
  they will always add up to 1, so can use as probabilities,
  small ones get closer together, so biggest stands out
- Big things, vision, language, etc. can use many layers
- Neurons in real brains are not really arranged in layers
  And are definitely not Fully Connected either

Training in general

- Improve performance by adjusting weights throughout the network
- Diminishing Returns – stop when it isn't making much difference any more
- Use a training set of course - large set of pairs (inputs, desired outputs)
- adjust each link according to the amount of error it was responsible for
- For each training point, work out the Local Error of the entire network
  Just the difference between the actual output and the correct output
  squared of course, negative errors are just as bad as positive ones
  If there are multiple outputs, just average all their errors
  That is Mean Squared Error, MSE
  Sometimes Root Mean Square, RMS, is used instead
- Repeat that for the entire training set, averaging all the errors
- If that average error is small enough
  the network has Converged. Training is over.

Training with a single layer (inputs are not neurons)

n_j represents the j^{th} neuron in the output (only) layer.

N is the number of output neurons

f is the activation function, $f(x) = 1 / (1 + e^{-x})$

$w_{i,j}$ is the weight on the connection from input i to output j

z_i^0 is the value of input i
z_j^1 is the value of output j

$z_j^1 = f(\Sigma(z_i^0 \times w_{i,j}))$

v_j is the total weighted input to n_j: $\Sigma(z_i^0 \times w_{i,j})$

t_j is the target, the correct value for z_j^1

E_j is the error in output j: $z_j^1 - t_j$

Total error, by Mean Square Error, $E = \Sigma E_j^2 \ / \ N$

To adjust any particular weight $w_{i,j}$,

we need to know what effect a change in $w_{i,j}$ would have on E: $\delta E/\delta w_{i,j}$

In this section, we are always talking about changes, not actual values:

The total error E due to $w_{i,j}$ depends on the output z_j^1

$$\delta E/\delta w_{i,j} = \delta E/\delta z_j^1 \times \delta z_j^1/\delta w_{i,j}$$

The output $z_{1,j}$ depends on its weighted inputs v_j

$$\delta E/\delta w_{i,j} = \delta E/\delta z_j^1 \times \delta z_j^1/\delta v_j \times \delta v_j/\delta w_{i,j}$$

E expanded = $((z_0^1 - t_0)^2 + (z_1^1 - t_1)^2 + (z_2^1 - t_2)^2 + (z_3^1 - t_3)^2 \ldots) \ / \ N$

$\delta E/\delta z_j^1 = z_j^1 - t_j$ all terms except the jth are independent of z_j^1 so deriv. is 0
and $(z_j^1 - t_j)^2 = z_j^{1\,2} - 2z_j^1 t_j + t_j^2$
and ignoring the constant $2/N$

$\delta z_j^1/\delta v_j = \delta f(v_j)/\delta v_j = z_j^1 \times (1 - z_j^1)$ because $\delta f(x)/\delta x = f(x) \times (1 - f(x))$

$\delta v_j/\delta w_{i,j} = z_i^0$ because $v_j = \Sigma(z_i^0 \times w_{i,j})$ and other terms are indep. of $w_{i,j}$

So the correction to $w_{i,j}$ should be proportional to $(z_j^1 - t_j) \times z_j^1 \times (1 - z_j^1) \times z_i^0$

$w_{i,j}$ becomes $w_{i,j} - \eta \times (z_j^1 - t_j) \times z_j^1 \times (1 - z_j^1) \times z_i^0$

η is the learning rate

Do that for every single weight

And repeat that many thousands of times through the training set.

For a network with three inputs and two outputs, we would calculate

$$\delta E/\delta w_{0,0} = \delta E/\delta z_0^1 \times \delta z_0^1/\delta v_0 \times \delta v_0/\delta w_{0,0}$$

$$\delta E/\delta w_{1,0} = \delta E/\delta z_0^1 \times \delta z_0^1/\delta v_0 \times \delta v_0/\delta w_{1,0}$$

$$\delta E/\delta w_{2,0} = \delta E/\delta z_0^1 \times \delta z_0^1/\delta v_0 \times \delta v_0/\delta w_{2,0}$$

$$\delta E/\delta w_{0,1} = \delta E/\delta z_1^1 \times \delta z_1^1/\delta v_1 \times \delta v_1/\delta w_{0,1}$$

$$\delta E/\delta w_{1,1} = \delta E/\delta z_1^1 \times \delta z_1^1/\delta v_1 \times \delta v_1/\delta w_{1,1}$$

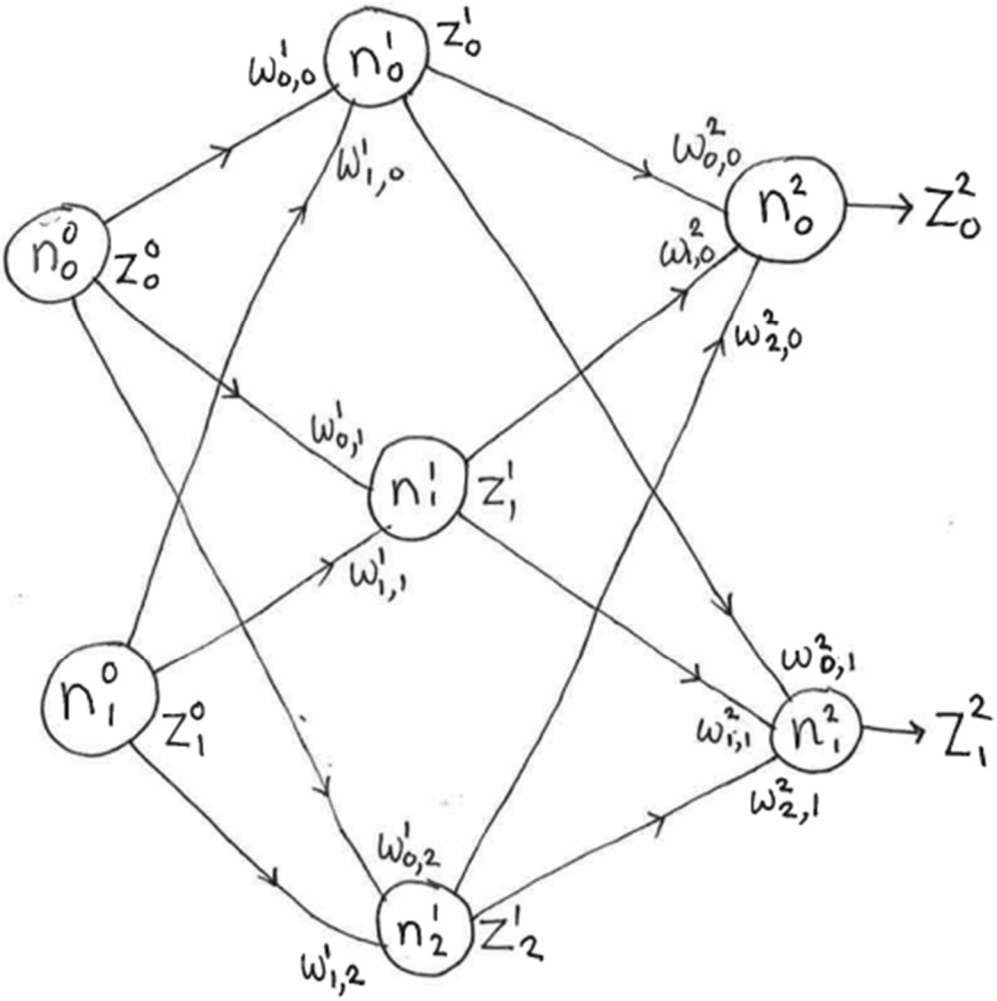
$$\delta E/\delta w_{2,1} = \delta E/\delta z_1^1 \times \delta z_1^1/\delta v_1 \times \delta v_1/\delta w_{2,1}$$

Note that the first two terms remain the same for each neuron: precompute.

Training with multiple layers

Back Propagation: work backwards from the output error,
distributing the blame for that error amongst all the weights
and adjusting those weights proportionately.

Each layer creates a different representation of the inputs
Inner layers often discover meaningful features of the input
The output layer usually has one neuron for each feature you want to detect or for each possible classification of the input.



n_i^ℓ is the i^{th} neuron in layer ℓ, layer 0 is just inputs not really neurons

z_i^ℓ is the output from n_i^ℓ

$w_{i,j}^\ell$ is the weight on the output from $n_i^{\ell-1}$ as input to n_j^ℓ

f is the activation function, $f(x) = 1 / (1 + e^{-x})$

$z_0^2 = f(z_0^1 \times w_{0,0}^2 + z_1^1 \times w_{1,0}^2 + z_2^1 \times w_{2,0}^2)$

in general ℓ represents any layer and L represents the last layer:

$z_j^\ell = f(\Sigma(n_i^{\ell-1} \times w_{i,j}^\ell)) \qquad \text{for } \ell > 1$

And as before, we split this into two parts

$v_j^\ell = \Sigma(n_i^{\ell-1} \times w_{i,j}^\ell)$
$z_j^\ell = f(v_j^\ell)$

t_i is the target value for output i, what v_j^L should be

E_j is the error in output j

$E_0 = z_0^L - t_0$

$E_1 = z_1^L - t_1$

Total error, by Mean Square Error, $E = (E_0 + E_1)^2 / 2$

Now for weights in the hidden layer, $w_{i,j}^\ell$, using $\delta E/\delta w_{0,1}^1$ as an example

$w_{0,1}^1$ affects E by two different paths,
from n_0^0 through $w_{0,1}^1$ to n_1^1 then through $w_{1,0}^2$ to n_0^2, and
from n_0^0 through $w_{0,1}^1$ to n_1^1 then through $w_{1,1}^2$ to n_1^2

For the first path we get
$\varepsilon_0 = \delta E/\delta z_0^2 \times \delta z_0^2/\delta v_0^2 \times \delta v_0^2/\delta z_1^1 \times \delta z_1^1/\delta v_1^1 \times \delta v_1^1/\delta w_{0,1}^1$
and for the second path we get
$\varepsilon_1 = \delta E/z_1^2 \times \delta z_1^2/\delta v_1^2 \times \delta v_1^2/\delta z_1^1 \times \delta z_1^1/\delta v_1^1 \times \delta v_1^1/\delta w_{0,1}^1$

As before:

$\delta E/\delta z_j^L = z_j^L - t_j$

$\delta z_j^L/\delta v_j^\ell = z_j^\ell \times (1 - z_j^\ell)$

And a differential we haven't seen before: $\delta v_j^\ell/\delta z_i^{\ell-1} = w_{i,j}^\ell$

$\delta v_j^\ell/\delta w_{i,j}^\ell = z_i^{\ell-1}$

ε_0 and ε_1 must be added together to get $\delta E/\delta w_{0,1}^1$

$w_{0,1}^1$ becomes $w_{0,1}^1 - \eta \times (\varepsilon_0 + \varepsilon_1)$

Vision (just a little bit)

- Given a megapixels-big RGB image with one input neuron per pixel
  and a fully connected design
  there would be trillions of weights to work with
- And Locality matters
  Neighbouring pixels contribute to real features, distant ones don't
  But a neuron's position within a layer makes no difference
  Might as well just present the pixels in random order – can't be right
- Also Spatial Invariance
  A cat looks the same regardless of where it appears in an image
  Cats should be recognized the same way regardless of position
  So we expect there to be groups of neurons all with the same weights
- Convolutional Neural Networks
  From studies of the Visual Cortex; the Receptive Field of a neuron
  Groups of neurons all connected to the same neighbouring pixels
  Those groups, Kernels, all have the same set of weights
  Kernels of size k, with a Stride of s
  The operation of such a layer can be treated as a Matrix operation
  GPUs are good for them

Memory

- Recurrent neural networks
- There can be loops back in the connections
  A delay is required at each step
- A neuron's new state can depend in some way on its previous state
  And those of its neighbours too
- Just like building a flip-flop out of nand gates
- Can analyse sequential data