Representations for AI Problem Solving

The Big Choice

1. **Crafted by human engineers?** Hard work to design, but the system’s decision-making is often understandable.

2. **Discovered by the AI system?** Relatively easy to facilitate, but a headache to interpret results, particularly with non-symbolic systems (e.g. neural nets).
1. Brief History of Neural Networks
2. Gradient Descent Learning
Early History of Neural Networks

- McCulloch + Pitts (1943) - neuron model similar to logic gates; no weights and no learning, but special excitatory and inhibitory connections.

- Rosenblatt (1958) - 3-layered neural network called perceptron, but now we use the term to refer to the nodes in Rosenblatt's output layer. The other 2 layers could not adapt.

- Widrow + Hoff (1960) - Adalines + Delta rule for training them, where error signal is based on the weighted sum of inputs, not the output of the activation function.

- Minsky + Papert (1969) - Proved that functions that were not linearly separable, e.g. XOR, were not representable by single-layered neural networks (with any type of neuron). Most hard problems were not linearly separable, and nobody knew how to train multilayer NNs.


We don’t do those here!


NN explosion (1985-1995) - Widespread applications.

Hibernation (1995-2005) - Trappings of local minima + failure of deep nets due to attenuated backprop signals seen as major weaknesses.

Deep Learning (2006- present) - Unsupervised pre-training, ReLUs, Convolution and GPUs enable truly multilayer (10 - 100+) networks.
The Perceptron

Keith L. Downing  Deep Learning: Lecture 1
The Delta Rule

\[ \delta = T - Y \]

\[ \Delta w = \eta \delta X \]

T = target output value
\( \delta \) = error

Node N
XOR - The (Near) Death of NNs

Keith L. Downing
Deep Learning: Lecture 1
Linear Separability

Keith L. Downing  Deep Learning: Lecture 1
Not Linearly Separable $\rightarrow$ Need Hidden Layer

Keith L. Downing  Deep Learning: Lecture 1
Backpropagation

Training/Test Cases: \{(d_1, r_1) (d_2, r_2) (d_3, r_3) \ldots \}

E = r_3 - r^* 

d_3 \rightarrow \text{Encoder} \rightarrow \text{Decoder} \rightarrow r^* 

\frac{dE}{dW} \rightarrow \text{Training/Test Cases: \{(d_1, r_1) (d_2, r_2) (d_3, r_3) \ldots \}} 

Neural Net

N times, with learning

1 time, without learning

Keith L. Downing

Deep Learning: Lecture 1
NETtalk (Sejnowski + Rosenberg, 1986)

- IBM's DECTalk: **several man years** of work → Reading machine.
- NETtalk: **10 hours** of backprop training on a 1000-word text, T1000.
- 95% accuracy on T1000; 78% accuracy on novel text.
- Improvement during training sounds like a child learning to read.
- Concept layer is key. 79 different (overlapping) clouds of neurons are gradually formed, with each mapping to one of the 79 phonemes.
1. Train the ANN (typically using backprop) on historical data to learn 
\[X(t_{-k}), X(t_{-k+1}), \ldots, X(t_0)] \mapsto [X(t_1), \ldots, X(t_{m-1}), X(t_m)]\]

2. Use to predict future value(s) based on the past k values.

Sample applications (Ungar, in *Handbook of Brain Theory and NNs, 2003*)

- Car sales
- Airline passengers
- Currency exchange rates
- Electrical loads on regional power systems.
- Flour prices
- Stock prices (*Warning*: often tried, but few good, documented results).
Key Elements of Recent Deep-Learning Success

1. Rectified Linear Units (ReLU)
2. Convolution
3. Dropout - different random subsets of neurons are temporarily silenced on different training cases.
4. Experience replay
5. Deep Nets + Reinforcement Learning - the system generates its own targets!
6. Deep Nets + RL + Monte Carlo Tree Search - rollouts, even in deterministic domains, are a good replacement for (bad) heuristics.
7. GPUs
The Universe of Deep Learning

- **General DL**: Feed Fwd + Backprop over many layers
- **Sequence Models**: Recurrent Nets, LSTM, GRU
- **Image Models**: Convolution Nets (CNNs)
- **Others**: Deep Reinforcement Learning, Adversarial Nets, Unsupervised Nets

* Nuts and Bolts of Applying Deep Learning, Andrew Ng (2016) - YouTube

Keith L. Downing  Deep Learning: Lecture 1
Use partial derivatives of an objective function, \( L \), (w.r.t. system parameters, \( W \)) to modify \( W \) so as to optimize (i.e. maximize or minimize) \( L \).

**Simple Example**

- \( C = \) set of cases; each case is a pair \((a, b)\)
- \( W = \) parameters (weights) of the system. \( W = \) \((x, y)\)
- \( L(a, b) = ax + by = \) the objective function
- Goal: Maximize \( \sum_{k \in C} L(a_k, b_k) \)
- **Problem**: You will never know the complete contents of \( C \). You will have to tune \( x \) and \( y \) to give the best results for the sample of cases that you do experience.
- **Solution**: Each time you see a case \((a_j, b_j)\), you can *nudge* \( x \) and \( y \) in directions that increase \( L(a, b) \) for that case: \((a_j, b_j)\).

Mathematically (use gradients, with \( \eta = \) learning rate):

- \( \Delta x = \eta \frac{\partial L(a, b)}{\partial x} \bigg|_{(a_j, b_j)} = \eta a_j \)
- \( \Delta y = \eta \frac{\partial L(a, b)}{\partial y} \bigg|_{(a_j, b_j)} = \eta b_j \)
Gradient-Descent Learning

Use partial derivatives of an objective (a.k.a. loss or cost) function, L, (w.r.t. system parameters, W) to modify W so as to minimize L.

Simple Example

- C = set of cases; each case, c_i is a vector: \((f_{i,1}, f_{i,2}, \ldots f_{i,m}, t_i) = \) features + a target value.
- W = parameters (weights) of the system, a vector.
- \(P(f_i) = f_i \cdot W \) = the prediction function
- Goal: Minimize \(L(C) = \sum_{k \in C} |P(f_k) - t_k| \) = the loss function.
- Problem: You only see some of the cases, but you want to perform proper predictions for new cases as well. You will have to tune W to give the best results for the sample of cases that you do experience.
- Solution: Each time you see a case \((c_j)\), you can nudge the \(w_i\) in directions that decrease \(L(c_j)\) for that case: \((f_j, t_j)\).

Mathematically (use gradients, with \(\eta = \) learning rate):

\[ \Delta w_i = -\eta \frac{\partial L(c)}{\partial w_i} |(f_j, t_j) = ????? \]
Differentiating the Loss Function

The mean-squared error (MSE) is a common loss function:

$$L(C) = \frac{1}{\|C\|} \sum_{k \in C} (P(f_k) - t_k)^2$$

By the chain rule of calculus:

$$\frac{\partial L(C)}{\partial w_i} = \frac{1}{\|C\|} \sum_{k \in C} 2(P(f_k) - t_k) \frac{\partial (P(f_k) - t_k)}{\partial w_i} = \frac{2}{\|C\|} \sum_{k \in C} (P(f_k) - t_k)f_{k,i}$$

since: $\frac{\partial t_k}{\partial w_i} = 0$ and $\frac{\partial P(f_k)}{\partial w_i} = f_{k,i}$ So when we have a sample $S$ of cases, we estimate the partial derivative (a.k.a. gradient) as:

$$\frac{2}{\|S\|} \sum_{k \in S} (P(f_k) - t_k)f_{k,i}$$

And we update $w_i$ to try to reduce the loss:

$$\Delta w_i = -\eta \frac{2}{\|S\|} \sum_{k \in S} (P(f_k) - t_k)f_{k,i}$$
Complexifying Neural Networks

How do these networks get more complicated?

- Non-linear activation functions (applied to summed inputs)
- Multiple output neurons
- Multiple layers of neurons separating inputs from outputs
- More elaborate objective (loss) functions
Despite increasingly complex NNs, the essence of gradient-descent learning remains the same: For EVERY weight $w_i$ in the network:

$$\triangle w_i = -\eta \frac{\partial L(c)}{\partial w_i}$$

Calculating these gradients just gets more complicated.

ML packages such as Tensorflow, Theano and Torch automate this!!
Types of Gradient Descent Learning

- **Batch Gradient Descent:**
  - Run ALL cases in the training set through the net.
  - Calculate gradients for each case.
  - Update weights via gradient combinations (e.g., averages)
  - Repeat
    * Each round is called an *epoch*.

- **Stochastic Gradient Descent (SGD):**
  - Send N randomly-selected cases (the *minibatch*) through the net.
  - Calculate gradients for each of the N cases.
  - Update weights based on combined gradients
  - Repeat
    * All cases in a minibatch experience the same weights on the forward pass.
  ** Potentially unstable for small N.
Matrices for Neural Networks

Incoming weights

Output Values

\[
\begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3
\end{bmatrix}
\begin{bmatrix}
w_{11} & w_{12} \\
w_{21} & w_{22} \\
w_{31} & w_{32}
\end{bmatrix}
= 
\begin{bmatrix}
  1 \\
  2
\end{bmatrix}
\]

\[
\begin{bmatrix}
x_1 w_{11} + x_2 w_{21} + x_3 w_{31} \\
x_1 w_{12} + x_2 w_{22} + x_3 w_{32}
\end{bmatrix}
= 
\begin{bmatrix}
  1 \\
  2
\end{bmatrix}
\]

Sum weighted inputs to 1

Sum weighted inputs to 2
Standard Representations

\[ X = \begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3
\end{bmatrix} \]

* Standard notation: activation vectors are COLUMN vectors:

3 x 1 Column Vector

\[ X^T \]

1 x 3 Row Vector

\[ W = \begin{bmatrix}
  w_{11} & w_{12} \\
  w_{21} & w_{22} \\
  w_{31} & w_{32}
\end{bmatrix} \]

1 x 2 Row Vector

3 x 2 Matrix

\[ X^T W = (W^T X)^T \]
Transpose all weight matrices (just once) and then use $W^T X$ to produce new column vectors. Then activation vectors need not be transposed back and forth between column and row vectors.

$$W^T \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = 2 \times 1 \text{ Column Vector}$$

$$X_0 = W^T X_0$$

$$X_1 = F_{\text{act}}(W^T X_0)$$

$$Z = \text{a 2} \times 5 \text{ Weight Matrix}$$

$$X_2 = G_{\text{act}}(Z^T X_1)$$

$$X_2 = G_{\text{act}}(Z^T F_{\text{act}}(W^T X_0))$$

$F_{\text{act}}$ and $G_{\text{act}}$ are activation functions that are applied to each tensor element. They do not alter its shape.
A bias node has a constant output of 1 but has K independent output weights, one per node in the layer. These are modified by gradient descent just like all other weights.
Adding a Column Vector for Biases

- \( B_w \) = a column vector of biases, one per node in layer that produces \( X_1 \).
- \( B_z \) = a column vector of biases, one per node in the layer that produces \( X_2 \)

\[
X_1 = F_{act}(W^T X_0 + B_w)
\]
\[
X_2 = G_{act}(Z^T X_1 + B_z)
\]

... or....
\[
X_2 = G_{act}(Z^T F_{act}(W^T X_0 + B_w) + B_z)
\]
Using a MiniBatch of Inputs

- Instead of using a single m x 1 column vector, as input, use a minibatch(C) of n vectors: one column per vector. C= m x n matrix.
- Every activation vector (originally a q x 1 column vector) now becomes a q x n matrix, one column per minibatch item.
- The weight matrices do not change.
- The q x 1 bias vectors become q x n matrices, with the original column copied n times. This is known as broadcasting: expanding a matrix along one or more dimensions via copying.
- The activation functions remain the same.

\[
X_1 = F_{act}(W^TC + B_w) \\
X_2 = G_{act}(Z^T X_1 + B_z) \\
\text{... or...} \\
X_2 = G_{act}(Z^T F_{act}(W^TC + B_w) + B_z)
\]
Forward Propagation of a MiniBatch

\[
\begin{align*}
W^T & = \begin{bmatrix}
w_{11} & w_{21} & w_{31} \\
w_{12} & w_{22} & w_{32}
\end{bmatrix} \\
C & = \begin{bmatrix}
\\text{2 x 3 Matrix} & \\text{Size 4 Minibatch} \\
\text{2 x 4 Matrix} & \text{3 x 4 Matrix}
\end{bmatrix}
\end{align*}
\]

\[
F_{\text{act}}(W^T C) = \begin{bmatrix}
\\text{Outputs of the 2} \\
\text{Layer-1 nodes} \\
\text{caused by the} \\
\text{first minibatch case}
\end{bmatrix}
\]

\[
\begin{bmatrix}
\\text{Outputs of the 2} \\
\text{Layer-1 nodes} \\
\text{caused by the} \\
\text{fourth minibatch case}
\end{bmatrix}
\]

Most tensor calculations, including those deriving gradients, work seamlessly with the extra tensor dimension associated with the minibatch.
Classical Activation Functions

1. **Identity**

2. **Step**

3. **Ramp**

4. **Logistic**

5. **Hyperbolic Tangent (tanh)**
Sigmoids

- The logistic (a.k.a. sigmoid) was very popular for neural networks, because:
  - Common in biological systems
  - Implements a step/ramp but is continuous at the threshold, thus simplifying gradient calculations.
- But the sigmoid saturates for inputs of large (pos or neg) magnitude. These flat regions of the sigmoid curve have near-zero derivatives.
- Once a sigmoid saturates, the weights leading into that node have near-zero gradients and thus change very little. Learning halts.
- Hence, sigmoids are a poor choice for hidden nodes, particularly in deep networks.
- However, they are still useful for output nodes, especially when they can be combined with an objective function that reduces the risk of saturation (as discussed later).
- They are also still very useful in neural networks that do not do backpropagation, and thus do not calculate gradients.
- The hyperbolic tangent (tanh) resembles the sigmoid, but since tanh(0) = 0 and it can also output negatives, it behaves like an identity function (for small inputs). This makes it easier to train.
Contemporary Activation Functions

These have **greatly** improved the performance of DL systems.

- **ReLU**
  \[ x_i = \max(0, V_i) \]

- **Softplus**
  \[ x_i = \log(1 + \exp(V_i)) \]

- **Leaky ReLU**
  \[ x_i = \max(cV_i, V_i) \]
  where \( c \) is small, e.g., 0.03

- **Exponential Linear Unit (ELU)**
  \[ x_i = \begin{cases} 
  c(\exp(V_i) - 1), & \text{if } V_i < 0 \\
  V_i, & \text{else} \end{cases} \]
  Often, \( c = 1 \)
The Rectified Linear Unit (ReLU)

Other activation functions yield error gradients that are strongly attenuated during backprop, such that only the last few layers of weights adapt appropriately.

ReLU’s permit sustained influence of gradients upon all weights, throughout the network.

By making many activation levels a hard zero, ReLU’s sparsify patterns, which is more biologically accurate AND better for learning.

Comparing Gradients: Sigmoid -vs- ReLU

- $z =$ sum of inputs to the activation function.
- $y =$ output of the activation function.

**$f_T =$ Rectified Linear Unit (ReLU):** $f_T(z) = \max(0, z)$

$$
\frac{\partial f_T(z)}{\partial z} = \begin{cases} 
1 & \text{when } z > 0 \\
0 & \text{otherwise}
\end{cases}
$$

Thus, the gradient is large for $z > 0$

**$f_T =$ Sigmoid:** $f_T(z) = \frac{1}{1+e^{-z}}$

$$
\frac{\partial f_T(z)}{\partial z} = y(1-y)
$$

- This has max = 0.25, and when the sigmoid is saturated, it will be much less than that: $\frac{\partial f_T(z)}{\partial z} = 0.048$ when $y = 0.95$.
- These gradients get multiplied repeatedly (via the chain rule) as backprop moves upstream, so products can get very small, very quickly.
ReLUs: When \( V_i \) (sum of inputs) is non-positive, the neuron outputs a zero AND its gradient becomes zero, so nothing changes. Inactive neurons often fail to come back to life.

**Solution:** **Leaky ReLUs:** Outputs may be small negative values, but these keep the neuron semi-active and yield non-zero gradients which can eventually revive the neuron.

- Variations of this: Randomized Leaky ReLU (RReLU) and Parametric Leaky ReLU (PReLU)

**Instability:** Gradients can be unstable (bouncing around from positive to 0) when \( V_i \approx 0 \). This is due to the discontinuity of the ReLU (and Leaky ReLU) at 0.

**Solution:** **Exponential Linear Unit (ELU):** This is negative for \( V_i < 0 \) (like the Leaky ReLU) but is continuous at \( V_i = 0 \), thus reducing oscillations of the gradient.
Many classification problems benefit from an output layer that represents a probability distribution over the possible classes. Hence, all outputs must be non-negative, and they must sum to 1. No individual activation function can enforce this summation condition. The outputs must be combined and scaled. For an output layer of size N with a vector of values, $v_i$:

$$\text{softmax}(v_i) = \frac{e^{v_i}}{\sum_{k=1}^{N} e^{v_k}}$$

Note how this easily combines both pos and neg $v_i$, insuring that every softmax’d value is positive. Softmax embodies competition among the outputs, thus modelling lateral inhibition found in many brain regions.
### Scaling via Softmax

<table>
<thead>
<tr>
<th></th>
<th>1.0</th>
<th>1.0</th>
<th>2.0</th>
<th>1.0</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Softmax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.15</td>
<td>0.15</td>
<td>0.40</td>
<td>0.15</td>
<td>0.15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1.0</th>
<th>1.0</th>
<th>2.0</th>
<th>-1.0</th>
<th>-1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Softmax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>0.2</td>
<td>0.54</td>
<td>0.03</td>
<td>0.03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Softmax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.01</td>
<td>0.03</td>
<td>0.08</td>
<td>0.23</td>
<td>0.64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Softmax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.07</td>
<td>0.18</td>
<td>0.50</td>
<td>0.18</td>
<td>0.07</td>
</tr>
</tbody>
</table>
Our Modeling Objective

- Given a data source, $S$, build a model that best captures the statistical relationships (between features and classes) in $S$.

- If $S$ is considered one big state, then our model ($\theta$) should maximize the probability of that state, i.e., the probability of ALL cases in that state being true:

$$
\pi(S) = \prod_{k \in S} p(f_k, t_k) = \prod_{k \in S} p(t_k | f_k) \times p(f_k)
$$

where $f_k = \text{features}$ and $t_k = \text{class (target)}$ for case $k$. 

\[ (1) \]
Our model ($\theta$) will estimate $p(t_k|f_k)$ with $p_\theta(t_k|f_k)$. E.g., $\theta$ are the weights of a neural network that takes $f_k$ as input and outputs a vector of probabilities ($P_k$), one for each class.

$\theta$ and $p(f_k)$ are independent, as the latter are determined solely by $S$.

Thus, we want to choose $\theta$ to maximize:

$$\pi_\theta(S) = \prod_{k \in S} p_\theta(t_k|f_k) \times p(f_k)$$

That is equivalent to maximizing:

$$\log(\pi_\theta(S)) = \sum_{k \in S} \log(p_\theta(t_k|f_k)) + \log(p(f_k)) = \sum_{k \in S} \log(p_\theta(t_k|f_k)) + \sum_{k \in S} \log(p(f_k))$$

The latter (red) summation is independent of $\theta$, so maximizing $\pi_\theta(S)$ is equivalent to maximizing the former (blue) summation.

The **conditional maximum likelihood estimator** is defined as:

$$\theta_{ML} = \arg\max_{\theta} \sum_{k \in S} \log(p_\theta(t_k|f_k))$$

I.e., the $\theta$ that maximizes the log output values of nodes corresponding to the **correct** class for each case.
Assume each case \( k \in S \) (or, more likely, a sample (\( \tilde{S} \)) of \( S \)) is run through a model (\( \theta \)) to produce \( P_k \). We then take the log of each probability to yield \( \log(P_k) \), which is then compared to a one-hot target vector (\( T_k \)) that encodes the correct class. The comparison is performed via vector dot product.

For case \( k \), this means that \( T_k \cdot \log(P_k) = \log(p_\theta(t_k|f_k)) \), and thus:

\[
\sum_{k \in \tilde{S}} T_k \cdot \log(P_k) = \sum_{k \in \tilde{S}} \log(p_\theta(t_k|f_k))
\]

The same \( \theta_{ML} \) maximizes both, and minimizes the negation:

\[
- \sum_{k \in \tilde{S}} T_k \cdot \log(P_k)
\]

Thus, the conditional maximum likelihood estimate minimizes the cross entropy, a common objective function. It also minimizes mean-squared error (MSE) for regression problems.

Cross-entropy and MSE are very popular cost functions for DL.
Cross-Entropy as Cost Function

- Useful when targets and outputs both represent probability distributions. So combine softmax'd outputs with cross-entropy cost function.
- When combined with sigmoid, the log counteracts the exp to reduce saturation (and gradient decay).
- Useful for classification problems, especially when targets are one-hot vectors and all network outputs are normalized (to sum to 1).

\[ H(P, T) = -\sum_{k=1}^{\|T\|} T_k \log(P_k) \]

where \( P \) = network’s output (prediction) and \( T \) = target vector

Reduction example:

- Prediction: 
  - 0.15 0.35 0.25 0.25
  - 0.05 0.05 0.9 0.00

- One-hot target:
  - 0 0 1 0
  - 0 0 1 0

- \(-T_i \log_2(P_i)\) for i=1:
  - 0 0 2.0 0
  - 0 0 0.15 0

- Reduced cost when predictions match the hot bit:
  \( H(P, T) = 2.0 \) \( H(P, T) = 0.15 \)
Cross-Entropy to Compare 2 Distributions

<table>
<thead>
<tr>
<th>Prediction</th>
<th>0.15</th>
<th>0.35</th>
<th>0.25</th>
<th>0.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>0.8</td>
<td>0.05</td>
<td>0.05</td>
<td>0.1</td>
</tr>
</tbody>
</table>

\[-T_i \log_2(P_i)\]

\[
\begin{align*}
2.2 & \quad 0.08 & \quad 0.1 & \quad 0.2 \\
0.26 & \quad 0.17 & \quad 0.17 & \quad 0.67 \\
\end{align*}
\]

Pay heavy cost when a high-probability target is not matched by the prediction.

Pay much lower cost when a low-probability target is not matched.

\[
H(P, T) = 2.58 \\
H(P, T) = 1.27
\]

Keith L. Downing

Deep Learning: Lecture 1
Logistic (or *Logit*) Regression: Single-value prediction where output represents probability of class membership.

- The output node typically uses a sigmoid activation function (a.k.a. *logistic function*), which outputs a value in range (0,1), interpreted as the probability of class membership, $p_k$, for the kth case.

- Each target value, $t_k$, is binary.

- Cost Function known as *Log Loss* applied to minibatch C:

$$L(C) = -\frac{1}{\|C\|} \sum_{k \in C} t_k \log(p_k) + (1 - t_k) \log(1 - p_k)$$

Depending on $t_k$, only one of 2 terms are non-zero for each case:

- $t_k = 1$: $p_k$ should be **high** to reduce cost.
- $t_k = 0$: $p_k$ should be **low**, so $(1 - p_k)$ is **high** to reduce cost.

Same principle as cross-entropy, but now the single output node represents two classes: YES or NO. In the case of NO, the second term of the sum kicks in, asking, ”How close is the prediction to a NO?”