



THE UNIVERSITY *of* EDINBURGH
informatics

Machine Learning Practical (MLP)

Single Layer Networks

Stochastic gradient descent

Classification

Changjian Li

Single Layer Networks

Terminology

Learn a system which maps an input vector x to an output vector y

- **Runtime:** Compute the output y for each input x
- **Training:** Optimise the parameters of the network such that the correct y is computed for each x
- **Generalisation:** We are most interested in the output accuracy of the system for unseen test data
- **Single Layer Network:** Use a single layer of computation (linear / affine transformation) to map between input and output

Single Layer Networks

Input vector $x = (x_1, \dots, x_d)^T$

Output vector $y = (y_1, \dots, y_K)^T$

Weight matrix W : w_{ki} is the weight from input x_i to output y_k

Bias vector b : b_k is the bias for output k

$$y_k = \sum_{i=1}^d w_{ki}x_i + b_k \quad ; \quad \mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Single Layer Networks

Input vector $x = (x_1, \dots, x_d)^T$

Output vector $y = (y_1, \dots, y_K)^T$

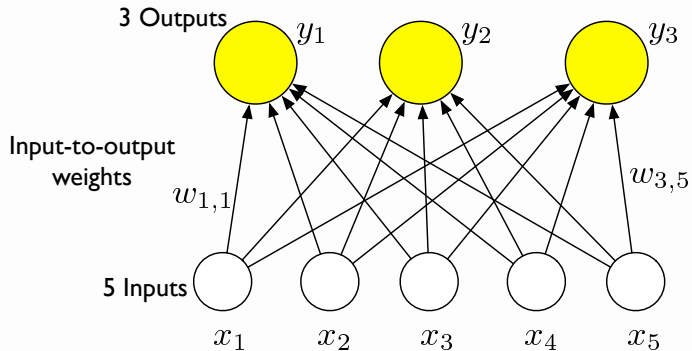
Weight matrix W : w_{ki} is the weight from input x_i to output y_k

Bias vector b : b_k is the bias for output k

$$y_k = \sum_{i=1}^d w_{ki}x_i + b_k \quad ; \quad \mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Also known as Linear Regression

Graph Visualization



$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$
$$y_k = \sum_{i=1}^d W_{ki}x_i + b_k$$

Training Single Layer Networks

Training set N input/output pairs $\{(\mathbf{x}^n, \mathbf{t}^n) : 1 \leq n \leq N\}$

Target vector $\mathbf{t}^n = (t_1^n, \dots, t_K^n)^T$ – the target output for input $\mathbf{x}^n$

Output vector $\mathbf{y}^n = \mathbf{y}^n(\mathbf{x}^n; \mathbf{W}, \mathbf{b})$ – the output computed by the network for input $\mathbf{x}^n$

Training Single Layer Networks

Training set N input/output pairs $\{(\mathbf{x}^n, \mathbf{t}^n) : 1 \leq n \leq N\}$

Target vector $\mathbf{t}^n = (t_1^n, \dots, t_K^n)^T$ – the target output for input $\mathbf{x}^n$

Output vector $\mathbf{y}^n = \mathbf{y}^n(\mathbf{x}^n; \mathbf{W}, \mathbf{b})$ – the output computed by the network for input $\mathbf{x}^n$

Trainable parameters weight matrix $\mathbf{W}$, bias vector $\mathbf{b}$

Training Single Layer Networks

Training set N input/output pairs $\{(\mathbf{x}^n, \mathbf{t}^n) : 1 \leq n \leq N\}$

Target vector $\mathbf{t}^n = (t_1^n, \dots, t_K^n)^T$ – the target output for input $\mathbf{x}^n$

Output vector $\mathbf{y}^n = \mathbf{y}^n(\mathbf{x}^n; \mathbf{W}, \mathbf{b})$ – the output computed by the network for input $\mathbf{x}^n$

Trainable parameters weight matrix $\mathbf{W}$, bias vector $\mathbf{b}$

Supervised learning There is a target output for each input

Training problem Set the values of the weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ such that each input $\mathbf{x}^n$ is mapped to its target $\mathbf{t}^n$

Error function Define the training problem in terms of an error function E ; training corresponds to setting the weights so as to minimise the error

Error function

- Error function should measure how far an output vector is from its target – e.g. (squared) Euclidean distance – *mean square error*.

E^n is the error per example:

$$E^n = \frac{1}{2} \|\mathbf{y}^n - \mathbf{t}^n\|^2 = \frac{1}{2} \sum_{k=1}^K (y_k^n - t_k^n)^2$$

E is the total error averaged over the training set:

$$E = \frac{1}{N} \sum_{n=1}^N E^n = \frac{1}{N} \sum_{n=1}^N \left(\frac{1}{2} \|\mathbf{y}^n - \mathbf{t}^n\|^2 \right)$$

Error function

- Error function should measure how far an output vector is from its target – e.g. (squared) Euclidean distance – *mean square error*.

E^n is the error per example:

$$E^n = \frac{1}{2} \|\mathbf{y}^n - \mathbf{t}^n\|^2 = \frac{1}{2} \sum_{k=1}^K (y_k^n - t_k^n)^2$$

E is the total error averaged over the training set:

$$E = \frac{1}{N} \sum_{n=1}^N E^n = \frac{1}{N} \sum_{n=1}^N \left(\frac{1}{2} \|\mathbf{y}^n - \mathbf{t}^n\|^2 \right)$$

- Training process: set $\mathbf{W}$ and $\mathbf{b}$ to minimise E given the training set

$$\mathbf{W}^*, \mathbf{b}^* = \arg \min_{\mathbf{W}, \mathbf{b}} \underbrace{\frac{1}{N} \sum_{n=1}^N \left(\frac{1}{2} \|\mathbf{y}^n - \mathbf{t}^n\|^2 \right)}_{E(\mathbf{W}, \mathbf{b})}$$

Gradient Descent

- **Weight space:** A $K \times d$ dimension space – each possible weight matrix corresponds to a point in weight space.
 $E(W)$ is the value of the error at a specific point in weight space (given the training data).

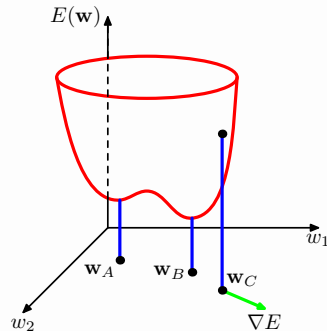


Image credit: 'C. M. Bishop,
H. Bishop, Deep Learning'

Gradient Descent

- **Weight space:** A $K \times d$ dimension space – each possible weight matrix corresponds to a point in weight space.
 $E(\mathbf{W})$ is the value of the error at a specific point in weight space (given the training data).
- **Gradient** of $E(\mathbf{W})$ given $\mathbf{W}$ is $\nabla_{\mathbf{W}}E$, the matrix of partial derivatives of E with respect to the elements of $\mathbf{W}$:

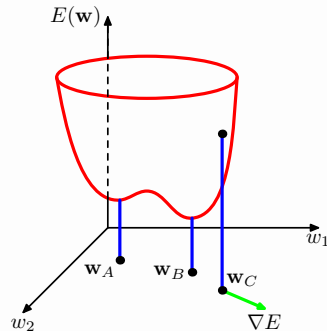


Image credit: 'C. M. Bishop,
H. Bishop, Deep Learning'

Gradient Descent

- **Weight space:** A $K \times d$ dimension space – each possible weight matrix corresponds to a point in weight space.
 $E(W)$ is the value of the error at a specific point in weight space (given the training data).
- **Gradient** of $E(W)$ given W is $\nabla_W E$, the matrix of partial derivatives of E with respect to the elements of W :
- **Gradient Descent Training:** adjust the weight matrix by moving a small direction down the gradient, which is the direction along which E decreases most rapidly.
 - update each weight w_{ki} by adding a factor $-\eta \cdot \partial E / \partial w_{ki}$
 - η is a small constant called the *step size* or *learning rate*.
- Adjust bias vector similarly

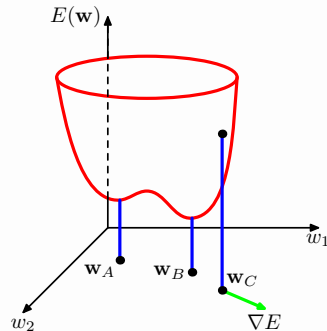


Image credit: 'C. M. Bishop, H. Bishop, Deep Learning'

Gradient Descent Procedure

1. Initialise weights and biases with small random numbers
2. For each epoch (complete pass through the training data)
 - 2.1 Initialise total gradients: $\Delta w_{ki} = 0, \Delta b_k = 0$
 - 2.2 For each training example n :
 - 2.2.1 Compute the error E^n
 - 2.2.2 For all k, i : Compute the gradients $\partial E^n / \partial w_{ki}, \partial E^n / \partial b_k$
 - 2.2.3 Update the total gradients by accumulating the gradients for example n

$$\Delta w_{ki} \leftarrow \Delta w_{ki} + \frac{\partial E^n}{\partial w_{ki}} \quad \forall k, i \qquad \Delta b_k \leftarrow \Delta b_k + \frac{\partial E^n}{\partial b_k} \quad \forall k$$

- 2.3 Update weights:

$$\begin{aligned} \Delta w_{ki} &\leftarrow \Delta w_{ki} / N; & w_{ki} &\leftarrow w_{ki} - \eta \Delta w_{ki} & \forall k, i \\ \Delta b_k &\leftarrow \Delta b_k / N; & b_k &\leftarrow b_k - \eta \Delta b_k & \forall k \end{aligned}$$

3. Terminate: after a number of epochs; or when error stops decreasing (threshold).

Gradient Descent Pseudocode (linear network)

```
1: procedure GDTRAINING( $X, T, W, b$ )
2:   Initialize  $W$  and  $b$  to small random numbers
3:   while not converged do
4:     Initialise total gradients:  $\Delta w_{ki} = 0, \Delta b_k = 0$ 
5:     for  $n \leftarrow 1, N$  do
6:       for  $k \leftarrow 1, K$  do
7:          $y_k^n \leftarrow \sum_{i=1}^d w_{ki} x_i^n + b_k$ 
8:          $g_k^n \leftarrow y_k^n - t_k^n$ 
9:         for  $i \leftarrow 1, d$  do
10:           $\Delta w_{ki} \leftarrow \Delta w_{ki} + g_k^n \cdot x_i^n$ 
11:           $\Delta b_k \leftarrow \Delta b_k + g_k^n$ 
12:       for  $k \leftarrow 1, K$  do
13:         for  $i \leftarrow 1, d$  do
14:           $w_{ki} \leftarrow w_{ki} - \eta \cdot \Delta w_{ki} / N$ 
15:           $b_k \leftarrow b_k - \eta \cdot \Delta b_k / N$ 
```

Applying gradient descent to a single-layer network

- Error function:

$$E = \frac{1}{N} \sum_{n=1}^N E^n \quad E^n = \frac{1}{2} \sum_{k=1}^K (y_k^n - t_k^n)^2 \quad \text{remember} \quad y_k^n = \sum_{i=1}^d w_{ki} x_i + b_k$$

Applying gradient descent to a single-layer network

- Error function:

$$E = \frac{1}{N} \sum_{n=1}^N E^n \quad E^n = \frac{1}{2} \sum_{k=1}^K (y_k^n - t_k^n)^2 \quad \text{remember} \quad y_k^n = \sum_{i=1}^d w_{ki} x_i + b_k$$

- Gradients:

$$\boxed{\frac{\partial E^n}{\partial w_{rs}}} = \frac{\partial E^n}{\partial y_r} \cdot \frac{\partial y_r}{\partial w_{rs}} = (y_r^n - t_r^n) \cdot x_s^n$$

$$\boxed{\frac{\partial E}{\partial w_{rs}}} = \frac{1}{N} \sum_{n=1}^N \frac{\partial E^n}{\partial w_{rs}} = \frac{1}{N} \sum_{n=1}^N (y_r^n - t_r^n) x_s^n$$

Applying gradient descent to a single-layer network

- Error function:

$$E = \frac{1}{N} \sum_{n=1}^N E^n \quad E^n = \frac{1}{2} \sum_{k=1}^K (y_k^n - t_k^n)^2 \quad \text{remember} \quad y_k^n = \sum_{i=1}^d w_{ki} x_i + b_k$$

- Gradients (grads_wrt_params):

$$\boxed{\frac{\partial E^n}{\partial w_{rs}}} = \underbrace{\frac{\partial E^n}{\partial y_r}}_{\text{error.grad}} \cdot \frac{\partial y_r}{\partial w_{rs}} = \underbrace{(y_r^n - t_r^n)}_{\text{output error}} \cdot \underbrace{x_s^n}_{\text{input}}$$

$$\boxed{\frac{\partial E}{\partial w_{rs}}} = \frac{1}{N} \sum_{n=1}^N \frac{\partial E^n}{\partial w_{rs}} = \frac{1}{N} \sum_{n=1}^N (y_r^n - t_r^n) \cdot x_s^n$$

Applying gradient descent to a single-layer network

- Error function:

$$E = \frac{1}{N} \sum_{n=1}^N E^n \quad E^n = \frac{1}{2} \sum_{k=1}^K (y_k^n - t_k^n)^2 \quad \text{remember} \quad y_k^n = \sum_{i=1}^d w_{ki} x_i + b_k$$

- Gradients (grads_wrt_params):

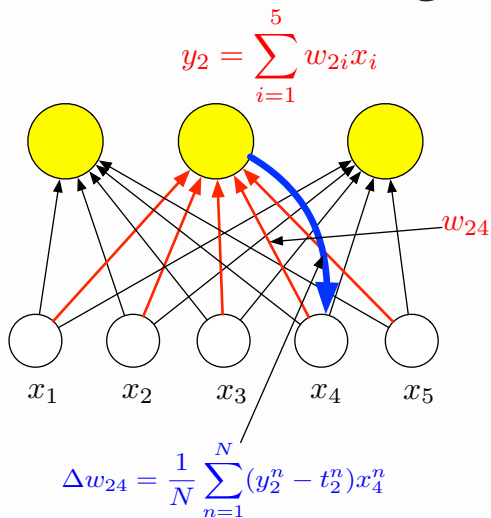
$$\boxed{\frac{\partial E^n}{\partial w_{rs}}} = \underbrace{\frac{\partial E^n}{\partial y_r}}_{\text{error.grad}} \cdot \frac{\partial y_r}{\partial w_{rs}} = \underbrace{(y_r^n - t_r^n)}_{\text{output error}} \cdot \underbrace{x_s^n}_{\text{input}}$$

$$\boxed{\frac{\partial E}{\partial w_{rs}}} = \frac{1}{N} \sum_{n=1}^N \frac{\partial E^n}{\partial w_{rs}} = \frac{1}{N} \sum_{n=1}^N (y_r^n - t_r^n) \cdot x_s^n$$

- Weight update

$$w_{rs} \leftarrow w_{rs} - \eta \cdot \frac{1}{N} \sum_{n=1}^N (y_r^n - t_r^n) x_s^n$$

Applying gradient descent to a single-layer network



Gradient Descent

Gradient Descent

☹ Training by batch gradient descent is **very slow** for large training data sets

- The algorithm sums the gradients over the entire training set before making an update
- To be able to fit a model to a dataset well, many updates are typically needed

Gradient Descent

☹ Training by batch gradient descent is **very slow** for large training data sets

- The algorithm sums the gradients over the entire training set before making an update
- To be able to fit a model to a dataset well, many updates are typically needed

How can we make GD faster?

Stochastic Gradient Descent (SGD)

- In SGD the complete gradient $\partial E / \partial w_{ki}$ (obtained by averaging over the entire training dataset) is approximated by the gradient for a point $\partial E^n / \partial w_{ki}$
- The weights are updated after each training example rather than after the batch of training examples
- Inaccuracies in the gradient estimates are washed away by the many approximations
- Present the training set in random order, to prevent multiple similar data points (with similar gradient approximation inaccuracies) appearing in succession

SGD Pseudocode (linear network)

```
1: procedure GDTRAINING( $X, T, W, b$ )
2:   Initialize  $W$  and  $b$  to small random numbers
3:   while not converged do
4:     Initialise total gradients:  $\Delta w_{ki} = 0, \Delta b_k = 0$ 
5:     for  $n \leftarrow 1, N$  do
6:       for  $k \leftarrow 1, K$  do
7:          $y_k^n \leftarrow \sum_{i=1}^d w_{ki} x_i^n + b_k$ 
8:          $g_k^n \leftarrow y_k^n - t_k^n$ 
9:         for  $i \leftarrow 1, d$  do
10:           $\Delta w_{ki} \leftarrow \Delta w_{ki} - g_k^n \cdot x_i^n$ 
11:           $\Delta b_k \leftarrow \Delta b_k + g_k^n$ 
12:       for  $k \leftarrow 1, K$  do
13:         for  $i \leftarrow 1, d$  do
14:           $w_{ki} \leftarrow w_{ki} - \eta \cdot \Delta w_{ki} / N$ 
15:           $b_k \leftarrow b_k - \eta \cdot \Delta b_k / N$ 
```

```
1: procedure SGDTRAINING( $X, T, W, b$ )
2:   initialize  $W$  and  $b$  to small random numbers
3:   randomize order of training examples in  $X$ 
4:   while not converged do
5:     for  $n \leftarrow 1, N$  do
6:       for  $k \leftarrow 1, K$  do
7:          $y_k^n \leftarrow \sum_{i=1}^d w_{ki} x_i^n + b_k$ 
8:          $g_k^n \leftarrow y_k^n - t_k^n$ 
9:         for  $i \leftarrow 1, d$  do
10:           $w_{ki} \leftarrow w_{ki} - \eta \cdot g_k^n \cdot x_i^n$ 
11:           $b_k \leftarrow b_k - \eta \cdot g_k^n$ 
```

Minibatches

- Batch gradient descent – compute the gradient from the batch of N training examples
- Stochastic gradient descent – compute the gradient from 1 training example each time
- Intermediate – compute the gradient from a **minibatch** of M training examples –
 $M > 1, M \ll N$

Minibatches

- Batch gradient descent – compute the gradient from the batch of N training examples
- Stochastic gradient descent – compute the gradient from 1 training example each time
- Intermediate – compute the gradient from a **minibatch** of M training examples – $M > 1, M \ll N$
- Benefits of minibatch:
 - Computationally efficient by making best use of vectorisation, keeping processor pipelines full – can parallelise the forward prop and gradient computations by processing examples in a minibatch together
 - Possibly smoother convergence as the gradient estimates are less noisy than using a single example each time

SGD Pseudocode (linear network)

```
1: procedure SGDTRAINING( $X, T, W, b$ )
2:   initialize  $W$  to small random numbers
3:   randomize order of training examples in  $X$ 
4:   while not converged do
5:     for  $n \leftarrow 1, N$  do
6:       for  $k \leftarrow 1, K$  do
7:          $y_k^n \leftarrow \sum_{i=1}^d w_{ki} x_i^n + b_k$ 
8:          $g_k^n \leftarrow y_k^n - t_k^n$ 
9:         for  $i \leftarrow 1, d$  do
10:           $w_{ki} \leftarrow w_{ki} - \eta \cdot g_k^n \cdot x_i^n$ 
11:           $b_k \leftarrow b_k - \eta \cdot g_k^n$ 
```

SGD Pseudocode (linear network)

How would you modify this code for minibatch training?

```
1: procedure SGDTRAINING( $X, T, W, b$ )
2:   initialize  $W$  to small random numbers
3:   randomize order of training examples in  $X$ 
4:   while not converged do
5:     for  $n \leftarrow 1, N$  do
6:       for  $k \leftarrow 1, K$  do
7:          $y_k^n \leftarrow \sum_{i=1}^d w_{ki} x_i^n + b_k$ 
8:          $g_k^n \leftarrow y_k^n - t_k^n$ 
9:         for  $i \leftarrow 1, d$  do
10:           $w_{ki} \leftarrow w_{ki} - \eta \cdot g_k^n \cdot x_i^n$ 
11:           $b_k \leftarrow b_k - \eta \cdot g_k^n$ 
```

Classification

MNIST Digit Classification



Classification and Regression

Regression predict the value of the output given an example input vector - e.g. what will be tomorrow's rainfall (in mm)

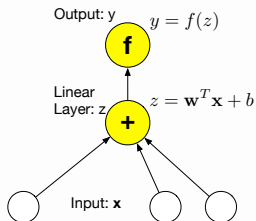
Classification and Regression

Regression predict the value of the output given an example input vector - e.g. what will be tomorrow's rainfall (in mm)

Classification predict the category given an example input vector
e.g. will it be rainy tomorrow (yes or no)?

- **Binary output:** 1 (yes) or 0 (no)
- **Probabilistic output:** p , $1 - p$ (for a 2-class problem)

Activation functions for two-class classification



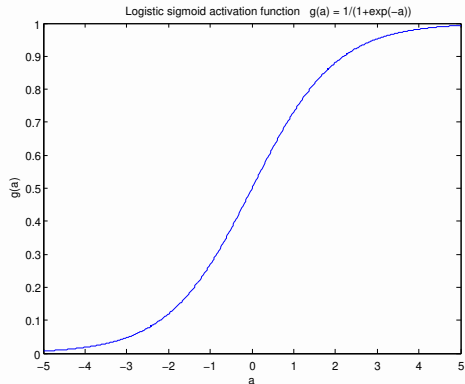
Single-layer network, binary/sigmoid output

Binary (**step** function):
$$f(z) = \begin{cases} 1 & \text{if } z \geq 0.5 \\ 0 & \text{if } z < 0.5 \end{cases}$$

Probabilistic (**logistic sigmoid** function):

$$f(z) = \frac{1}{1 + \exp(-z)}$$

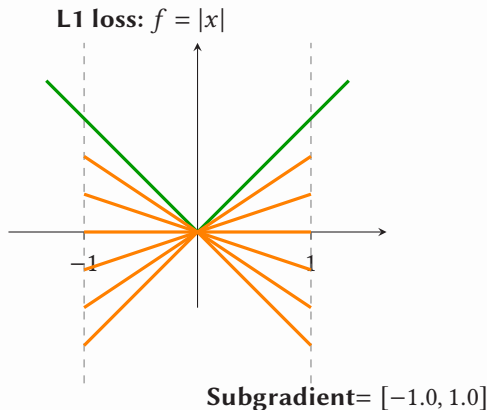
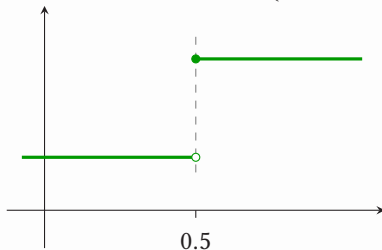
Logistic sigmoid function



Sigmoid single layer networks

- Binary output: activation is not differentiable.
 - Non-convex function
 - No subgradients

Step function: $f(x) = \begin{cases} 2.0, & x \geq 0.5, \\ 1.0, & x < 0.5. \end{cases}$



Sigmoid single layer networks

- Binary output: activation is not differentiable.
- Probabilistic output: single layer network with logistic sigmoid output – statisticians would call this *logistic regression*.

Let z be the value of the weighted sum of inputs, before the activation function, so:

$$z = \sum_i w_i x_i + b = \mathbf{w}^\top \mathbf{x} + b$$

$$y = f(z)$$

- Two classes, so single output y ,
 - y for $t = 1$ (class 1),
 - $1 - y$ for $t = 0$ (class 2)

Sigmoid single layer networks

- Training sigmoid single layer network: Gradient descent requires $\partial E / \partial w_i$ for all weights:

$$\frac{\partial E^n}{\partial w_i} = \frac{\partial E^n}{\partial y^n} \frac{\partial y^n}{\partial z^n} \frac{\partial z^n}{\partial w_i}$$

For a sigmoid:

$$y = f(z) \quad \frac{dy}{dz} = f'(z) = f(z)(1 - f(z))$$

Sigmoid single layer networks

- Training sigmoid single layer network: Gradient descent requires $\partial E / \partial w_i$ for all weights:

$$\frac{\partial E^n}{\partial w_i} = \frac{\partial E^n}{\partial y^n} \frac{\partial y^n}{\partial z^n} \frac{\partial z^n}{\partial w_i}$$

For a sigmoid:

$$y = f(z) \quad \frac{dy}{dz} = f'(z) = f(z)(1 - f(z))$$

- Therefore gradients of the error w.r.t. weights and bias (grads_wrt_params):

$$\boxed{\frac{\partial E^n}{\partial w_i}} = \underbrace{(y^n - t^n)}_{\text{output error}} \underbrace{f(z^n)(1 - f(z^n))}_{f'(z^n)} \underbrace{x_i^n}_{\text{input}}$$

$$\boxed{\frac{\partial E^n}{\partial b}} = \underbrace{(y^n - t^n)}_{\text{output error}} \underbrace{f(z^n)(1 - f(z^n))}_{f'(z^n)}$$

Cross-entropy error function (1)

- If we use a sigmoid single layer network for a two class problem (C_1 (target $t = 1$) and C_2 ($t = 0$)), then we can interpret the output as follows

$$y \sim P(C_1 \mid \mathbf{x}) = P(t = 1 \mid \mathbf{x}; \mathbf{W})$$

$$(1 - y) \sim P(C_2 \mid \mathbf{x}) = P(t = 0 \mid \mathbf{x}; \mathbf{W})$$

Cross-entropy error function (1)

- If we use a sigmoid single layer network for a two class problem (C_1 (target $t = 1$) and C_2 ($t = 0$)), then we can interpret the output as follows

$$y \sim P(C_1 \mid \mathbf{x}) = P(t = 1 \mid \mathbf{x}; \mathbf{W})$$

$$(1 - y) \sim P(C_2 \mid \mathbf{x}) = P(t = 0 \mid \mathbf{x}; \mathbf{W})$$

- Combining, and recalling the target is binary ($t = 0$ or $t = 1$):

$$P(t \mid \mathbf{x}; \mathbf{W}) = y^t \cdot (1 - y)^{1-t}$$

This is a Bernoulli distribution. We can write the log probability:

$$\ln P(t \mid \mathbf{x}; \mathbf{W}) = t \ln y + (1 - t) \ln(1 - y)$$

Cross-entropy error function (2)

- Optimise the weights $\mathbf{W}$ to maximise the log probability – or to minimise the negative log probability.

$$E^n = -\ln P(t^n \mid \mathbf{x}^n; \mathbf{W}) = -(t^n \ln y^n + (1 - t^n) \ln(1 - y^n)) .$$

This is called the **cross-entropy error function**

Cross-entropy error function (2)

- Optimise the weights $\mathbf{W}$ to maximise the log probability – or to minimise the negative log probability.

$$E^n = -\ln P(t^n | \mathbf{x}^n; \mathbf{W}) = -(t^n \ln y^n + (1 - t^n) \ln(1 - y^n)) .$$

This is called the **cross-entropy error function**

- Require `grads_wrt_params` for gradient descent training: $\partial E / \partial w_i$ (w_i connects the i th input to the single output).

$$\frac{\partial E^n}{\partial y^n} = -\frac{t^n}{y^n} + \frac{1 - t^n}{1 - y^n} = \frac{-(1 - y^n)t + y(1 - t^n)}{y^n(1 - y^n)} = \frac{(y^n - t^n)}{y^n(1 - y^n)}$$

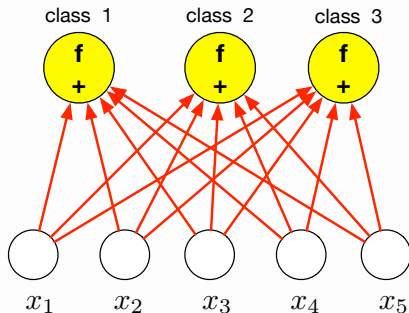
$$\boxed{\frac{\partial E^n}{\partial w_i}} = \frac{\partial E^n}{\partial y^n} \cdot \frac{\partial y^n}{\partial z^n} \cdot \frac{\partial z^n}{\partial w_i} = \frac{(y^n - t^n)}{y^n(1 - y^n)} \cdot y^n(1 - y^n) \cdot x_i^n = \boxed{(y^n - t^n)x_i^n}$$

Derivative of the sigmoid $y(1 - y)$ cancels.

Exercise: What is the gradient for the bias ($\frac{\partial E^n}{\partial b}$)?

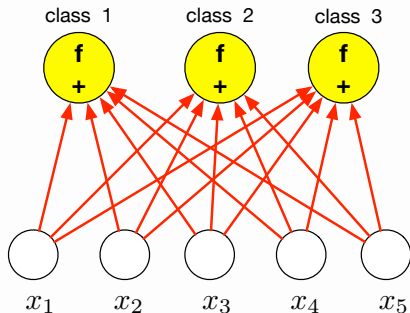
Multi-class networks

- If we have K classes, then use a “one-from-K” (“one-hot”) output coding – target of the correct class is 1, all other targets are 0



Multi-class networks

- If we have K classes, then use a “one-from-K” (“one-hot”) output coding – target of the correct class is 1, all other targets are 0



How should the output look like for multi-class problems?

Multi-class networks

- If we have K classes use a “one-hot” (“one-from-N”) output coding – target of the correct class is 1, all other targets are zero
- It is possible to have a multi-class net with sigmoids
- This will work...but we can do better

Multi-class networks

- If we have K classes use a “one-hot” (“one-from-N”) output coding – target of the correct class is 1, all other targets are zero
- It is possible to have a multi-class net with sigmoids
- This will work...but we can do better
- Using multiple sigmoids for multiple classes means that the outputs of the network are not constrained to sum to one
- To interpret the outputs of the net as class probabilities, require $\sum_k P(C_k|\mathbf{x}) = 1$
- Solution – use an output activation function with a sum-to-one constraint: **softmax**

Softmax

$$y_k = \frac{\exp(z_k)}{\sum_{j=1}^K \exp(z_j)}$$

$$z_k = \sum_{i=1}^d w_{ki}x_i + b_k$$

- This form of activation has the following properties
 - Each output will be between 0 and 1
 - The denominator ensures that the K outputs will sum to 1

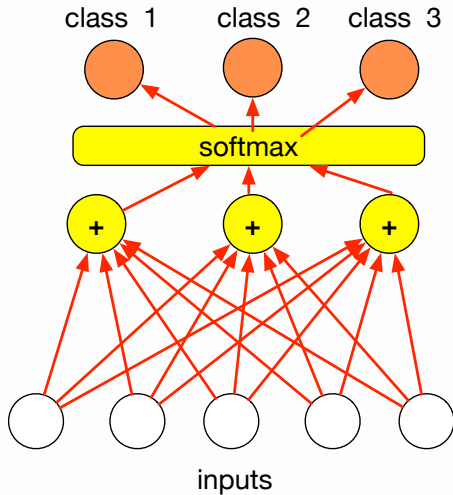
Softmax

$$y_k = \frac{\exp(z_k)}{\sum_{j=1}^K \exp(z_j)}$$

$$z_k = \sum_{i=1}^d w_{ki}x_i + b_k$$

- This form of activation has the following properties
 - Each output will be between 0 and 1
 - The denominator ensures that the K outputs will sum to 1
- Using softmax we can interpret the network output y_k^n as an estimate of $P(C_k|x^n)$
- Softmax is the multiclass version of the two-class sigmoid – as sigmoid models a Bernoulli distribution, so softmax models a Multinoulli (Categorical) distribution

Softmax



Softmax – Training (1)

- We can extend the cross-entropy error function to the multiclass case

$$E^n = -\ln P(C_{\ell^n}|\mathbf{x}^n) = -\sum_{k=1}^C t_k^n \ln y_k^n$$

where C_{ℓ^n} is the correct class for example n : ($t_{\ell^n}^n = 1$).

Softmax – Training (1)

- We can extend the cross-entropy error function to the multiclass case

$$E^n = -\ln P(C_{\ell^n} | \mathbf{x}^n) = -\sum_{k=1}^C t_k^n \ln y_k^n$$

where C_{ℓ^n} is the correct class for example n : ($t_{\ell^n}^n = 1$).

- Computing grads_wrt_params:

$$\boxed{\frac{\partial E^n}{\partial w_{ki}}} = \sum_{c=1}^C \frac{\partial E}{\partial y_c} \cdot \frac{\partial y_c}{\partial z_k} \cdot \frac{\partial z_k}{\partial w_{ki}} = \sum_{c=1}^C -\frac{t_c}{y_c} \cdot \frac{\partial y_c}{\partial z_k} \cdot x_i$$

$$\boxed{\frac{\partial E^n}{\partial b_k}} = \sum_{c=1}^C \frac{\partial E}{\partial y_c} \cdot \frac{\partial y_c}{\partial z_k} \cdot \frac{\partial z_k}{\partial b_k} = \sum_{c=1}^C -\frac{t_c}{y_c} \cdot \frac{\partial y_c}{\partial z_k}$$

Softmax – Training (2)

- Note that the k th output unit – and hence the weight w_{ki} – influences the error function through all the output units, because of the normalising term in the denominator. We have to take this into account when differentiating.

Softmax – Training (2)

- Note that the k th output unit – and hence the weight w_{ki} – influences the error function through all the output units, because of the normalising term in the denominator. We have to take this into account when differentiating.
- If you do the differentiation you will find:

$$\frac{\partial y_c}{\partial z_k} = y_c(\delta_{ck} - y_k)$$

Where δ_{ck} is called the Kronecker delta: $\delta_{ck} = 1$ if $c = k$, $\delta_{ck} = 0$ if $c \neq k$

- We can put it all together to obtain `grads_wrt_params`:

$$\boxed{\frac{\partial E^n}{\partial w_{ki}}} = (y_k^n - t_k^n)x_i^n \quad \boxed{\frac{\partial E^n}{\partial b_k}} = (y_k^n - t_k^n)$$

Softmax output with cross-entropy error function results in gradients with the same form as for linear outputs with mean square error!

Exercises

1. Modify the SGD pseudocode for sigmoid outputs
2. Modify the SGD pseudocode for softmax outputs
3. For softmax and cross-entropy error, show that

$$\frac{\partial E^n}{\partial w_{ki}} = (y_k^n - t_k^n)x_i^n$$

(use the quotient rule of differentiation, and the fact that $\sum_{c=1}^K t_c y_k = y_k$ because of 1-from- K coding of the target outputs)

4. Think of activation and error functions when an input can belong to multiple categories (e.g. two digits in an image).

Summary

- **Reading:**
 - Nielsen chapter 1
 - Goodfellow et al chapter 1; sections 4.3 (p79–83), 5.1, 5.7, 5.9, 6.1, 6.2, 8.1
- Stochastic gradient descent (SGD) and minibatch
- Classification and regression
- Sigmoid activation function and cross-entropy
- Multiple classes – Softmax
- Next lecture: multi-layer networks and hidden units



<https://forms.cloud.microsoft/e/WdeYNv4A5c>