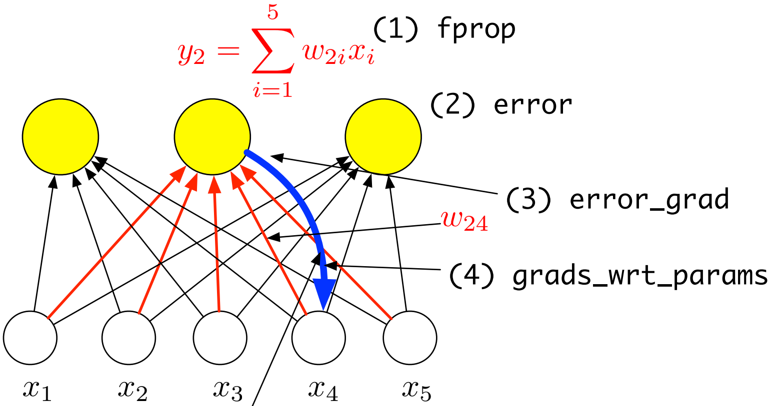
Single Layer Networks (2)
Stochastic gradient descent; Classification

Steve Renals

Machine Learning Practical — MLP Lecture 2
25 September 2018
http://www.inf.ed.ac.uk/teaching/courses/mlp/

Single Layer Networks

Recap: Gradient descent for a single-layer network



$$\Delta w_{24} = \frac{1}{N} \sum_{n=1}^{N} (y_2^n - t_2^n) x_4^n$$

Stochastic Gradient Descent (SGD)

- 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
  - Since the update steps (η) are small, many updates are needed

Stochastic Gradient Descent (SGD)

- 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
  - Since the update steps (η) are small, many updates are needed
- Solution: **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

Stochastic Gradient Descent (SGD)

- 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
  - Since the update steps (η) are small, many updates are needed
- Solution: **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)

procedure SGDTRAINING($\boldsymbol{X}, \boldsymbol{T}, \boldsymbol{W}$)

    initialize $\boldsymbol{W}$ to small random numbers

    randomize order of training examples in $\boldsymbol{X}$

    while not converged **do**

        for $n \leftarrow 1, N$ **do**

            for $k \leftarrow 1, K$ **do**

                $y_k^n \leftarrow \sum_{i=1}^{d} w_{ki} x_i^n + b_k$

                $g_k^n \leftarrow y_k^n - t_k^n$

                for $i \leftarrow 1, d$ **do**

                    $w_{ki} \leftarrow w_{ki} - \eta \cdot g_k^n \cdot x_i^n$

                end for

                $b_k \leftarrow b_k - \eta \cdot g_k^n$

            end for

        end for

    end while

end procedure

SGD Pseudocode (linear network)

procedure SGDTraining($\boldsymbol{X}, \boldsymbol{T}, \boldsymbol{W}$)
    initialize $\boldsymbol{W}$ to small random numbers
    randomize order of training examples in $\boldsymbol{X}$
    while not converged **do**
        for $n \leftarrow 1, N$ **do**
            for $k \leftarrow 1, K$ **do**
                $y_k^n \leftarrow \sum_{i=1}^{d} w_{ki} x_i^n + b_k$
                $\boldsymbol{g_k^n \leftarrow y_k^n - t_k^n}$
                for $i \leftarrow 1, d$ **do**
                    $\boldsymbol{w_{ki} \leftarrow w_{ki} - \eta \cdot g_k^n \cdot x_i^n}$
                end for
                $\boldsymbol{b_k \leftarrow b_k - \eta \cdot g_k^n}$
            end for
        end for
    end while
end procedure

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 – $M > 1,\ M << N$

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 – $M > 1$, $M << N$
- 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)

procedure SGDTRAINING($\boldsymbol{X}, \boldsymbol{T}, \boldsymbol{W}$)
    initialize $\boldsymbol{W}$ to small random numbers
    randomize order of training examples in $\boldsymbol{X}$
    while not converged **do**
        for $n \leftarrow 1, N$ **do**
            for $k \leftarrow 1, K$ **do**
                $y_k^n \leftarrow \sum_{i=1}^{d} w_{ki} x_i^n + b_k$
                $g_k^n \leftarrow y_k^n - t_k^n$
                for $i \leftarrow 1, d$ **do**
                    $w_{ki} \leftarrow w_{ki} - \eta \cdot g_k^n \cdot x_i^n$
                end for
                $b_k \leftarrow b_k - \eta \cdot g_k^n$
            end for
        end for
    end while
end procedure

SGD Pseudocode (linear network)

How would you modify this code for minibatch training?

procedure $\textsc{SGDTraining}(\boldsymbol{X}, \boldsymbol{T}, \boldsymbol{W})$
  initialize $\boldsymbol{W}$ to small random numbers
  randomize order of training examples in $\boldsymbol{X}$
  while not converged **do**
    for $n \leftarrow 1, N$ **do**
      for $k \leftarrow 1, K$ **do**
        $y_k^n \leftarrow \sum_{i=1}^{d} w_{ki} x_i^n + b_k$
        $g_k^n \leftarrow y_k^n - t_k^n$
        for $i \leftarrow 1, d$ **do**
          $w_{ki} \leftarrow w_{ki} - \eta \cdot g_k^n \cdot x_i^n$
        end for
        $b_k \leftarrow b_k - \eta \cdot g_k^n$
      end for
    end for
  end while
end procedure

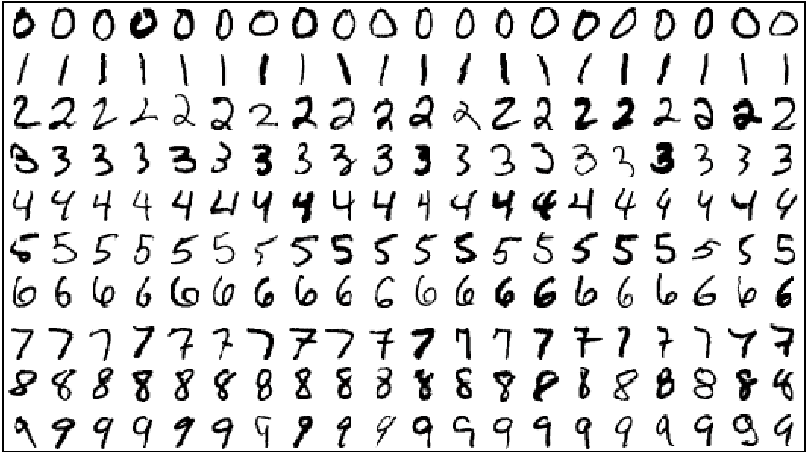
SGD Pseudocode (linear network)

How would you vectorise this code?

procedure $\textsc{SGDTraining}(\boldsymbol{X}, \boldsymbol{T}, \boldsymbol{W})$
    initialize $\boldsymbol{W}$ to small random numbers
    randomize order of training examples in $\boldsymbol{X}$
    while not converged **do**
        for $n \leftarrow 1, N$ **do**
            for $k \leftarrow 1, K$ **do**
                $y_k^n \leftarrow \sum_{i=1}^{d} w_{ki} x_i^n + b_k$
                $g_k^n \leftarrow y_k^n - t_k^n$
                for $i \leftarrow 1, d$ **do**
                    $w_{ki} \leftarrow w_{ki} - \eta \cdot g_k^n \cdot x_i^n$
                end for
                $b_k \leftarrow b_k - \eta \cdot g_k^n$
            end for
        end for
    end while
end procedure

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)

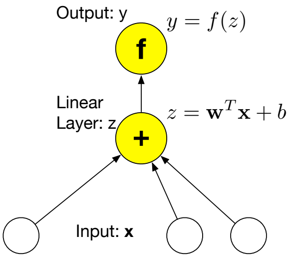
Classification and Regression

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

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)?
- Classification outputs:
  - **Binary**: 1 (yes) or 0 (no)
  - **Probabilistic**: p, $1 - p$ (for a 2-class problem)
- One could train a linear single layer network as a classifier:
  - Output targets are 1/0 (yes/no)
  - At run time if the output $y > 0.5$ classify as yes, otherwide classify as no
- This will work, but we can do better....
- Constrain the outputs to binary or probabilistic using an **activation function** on the output unit

Activation functions for two-class classification

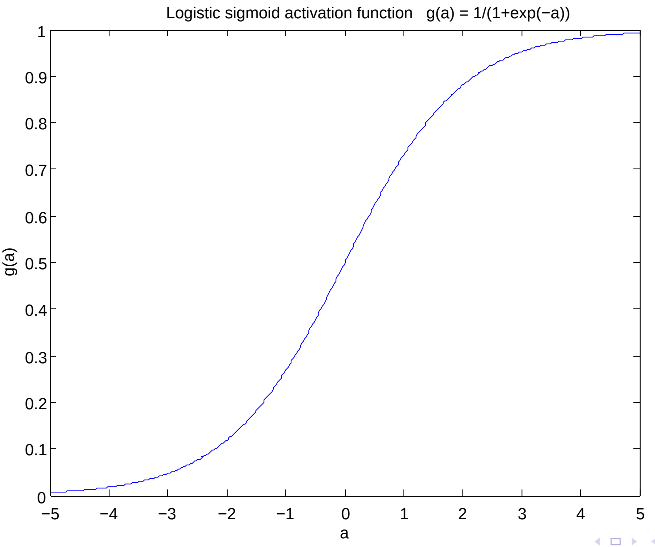


Single-layer network, binary/sigmoid output

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): $\boxed{f(z) = \dfrac{1}{1 + \exp(-z)}}$

Logistic sigmoid function



Sigmoid single layer networks

- Binary output: activation is not differentiable. Can use *perceptron learning* to train binary output single layer networks

Sigmoid single layer networks

- Binary output: activation is not differentiable. Can use *perceptron learning* to train binary output single layer networks
- 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:

$$z = \sum_i w_i x_i + b = \boldsymbol{w}^\mathsf{T} \boldsymbol{x} + b$$
$$y = f(z)$$

- Two classes, so single output y, with weights $\boldsymbol{w}$

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:

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

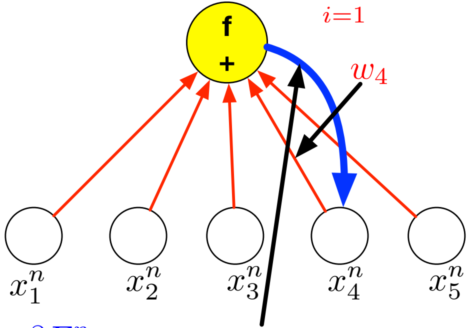
- Therefore gradients of the error w.r.t. weights and bias (`grads_wrt_params`):

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

$$\boxed{\frac{\partial E^n}{\partial b}} = (y^n - t^n) f(z^n)(1 - f(z^n))$$

Applying gradient descent to a sigmoid single-layer network

$$y^n = f(z^n); \qquad z^n = \sum_{i=1}^{5} w_i x_i^n + b$$



$$\frac{\partial E^n}{\partial w_4} = (y^n - t^n)\underbrace{y^n(1-y^n)}_{f'(z^n)} x_4^n$$

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

$$
\begin{aligned}
y &\sim P(C_1 \mid \boldsymbol{x}) = P(t = 1 \mid \boldsymbol{x}; \boldsymbol{W}) \\
(1 - y) &\sim P(C_2 \mid \boldsymbol{x}) = P(t = 0 \mid \boldsymbol{x}; \boldsymbol{W})
\end{aligned}
$$

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

$$\begin{aligned} y &\sim P(C_1 \mid \boldsymbol{x}) = P(t = 1 \mid \boldsymbol{x}; \boldsymbol{W}) \\ (1 - y) &\sim P(C_2 \mid \boldsymbol{x}) = P(t = 0 \mid \boldsymbol{x}; \boldsymbol{W}) \end{aligned}$$

- Combining, and recalling the target is binary

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

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

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

Cross-entropy error function (2)

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

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

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

Cross-entropy error function (2)

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

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

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 ith input to the single output).

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

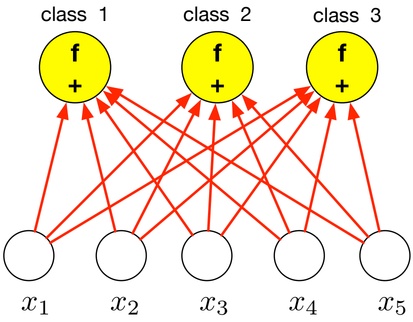
$$\boxed{\frac{\partial E}{\partial w_i}} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w_i} = \frac{(y-t)}{y(1-y)} \cdot y(1-y) \cdot x_i = \boxed{(y-t)x_i}$$

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

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

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
- It is possible to have a multi-class net with sigmoids



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

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

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

- 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

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

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

- 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 $P(C_k|\boldsymbol{x}^n)$
- Softmax is the multiclass version of the two-class sigmoid – as sigmoid models a Bernoulli distribution, so softmax models a Multinoulli (Categorical) distribution

Softmax – Training (1)

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

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

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 kth 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 kth 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`:

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

  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)

Summary

- **Reading**:
  - Nielsen chapter 1
  - Goodfellow et al sections 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