Stochastic gradient descent; Classification

Steve Renals

Machine Learning Practical — MLP Lecture 2
28 September 2016
Single Layer Networks
Applying gradient descent to a single-layer network

\[ y_2 = \sum_{i=1}^{5} w_{2i} x_i \]

\[ \Delta w_{24} = \sum_n (y_2^n - t_2^n) x_4^n \]
Output - predicted observation

\[ y = \hat{r}_t = \sum_{i=1}^{d} w_i x_i + b \]

\[ x_1 = r_{t-d} \quad x_{d-1} = r_{t-2} \quad x_d = r_{t-1} \]

Input - previous d observations
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 ($\eta$) are small many updates are needed

- Solution: **Stochastic Gradient Descent (SGD)**
- In SGD the true gradient $\frac{\partial E}{\partial w_{ki}}$ (obtained by summing over the entire training dataset) is approximated by the gradient for a point $\frac{\partial E^n}{\partial w_{ki}}$
- 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
- To prevent multiple similar data points (all with similar gradient approximation inaccuracies) appearing in succession, present the training set in random order
SGD Pseudocode (linear network)

1: **procedure** \( \text{SGDTraining}(X, T, W) \)
2: initialize \( W \) to small random numbers
3: randomize order of training examples in \( X \)
4: **while** not converged **do**
5: **for** \( n \) ← 1, \( N \) **do**
6: **for** \( k \) ← 1, \( K \) **do**
7: \( y_k^n \leftarrow \sum_{i=1}^{d} w_{ki} x_i^n + b_k \)
8: \( \delta_k^n \leftarrow y_k^n - t_k^n \)
9: **for** \( i \) ← 1, \( d \) **do**
10: \( w_{ki} \leftarrow w_{ki} - \eta \cdot \delta_k^n \cdot x_i^n \)
11: **end for**
12: \( b_k \leftarrow b_k - \eta \cdot \delta_k^n \)
13: **end for**
14: **end for**
15: **end while**
16: **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 – $M > 1, M << N$
- Benefits of minibatch:
  - Computationally efficient by making best use of vectorisation, keeping processor pipelines full
  - Possibly smoother convergence as the gradient estimates are less noisy than using a single example each time
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**: 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, otherwise classify as no

This will work, but we can do better....

Output activation functions to constrain the outputs to binary or probabilistic (logistic / sigmoid)
Two-class classification

Single-layer network, binary/sigmoid output

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

Probabilistic (sigmoid function): \( f(a) = \frac{1}{1 + \exp(-a)} \)
Sigmoid function

Logistic sigmoid activation function \( g(a) = \frac{1}{1+\exp(-a)} \)
Sigmoid single layer networks

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

- Probabilistic output: sigmoid single layer network (statisticians would call this logistic regression). Let $a$ be the *activation* of the single output unit, the value of the weighted sum of inputs, before the activation function, so:

  $$y = f(a) = f\left(\sum_i w_i x_i + b\right)$$

- Two classes, so single output $y$, with weights $w_i$.
Training sigmoid single layer network: Gradient descent requires $\frac{\partial E}{\partial w_i}$ for all weights:

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

For a sigmoid:

$$y = f(a) \quad \frac{dy}{da} = f(a)(1 - f(a))$$

(Show that this is indeed the derivative of a sigmoid.)

Therefore gradients of the error w.r.t. weights and bias:

$$\frac{\partial E^n}{\partial w_i} = (y^n - t^n) f(a^n)(1 - f(a^n)) \frac{\delta^n}{f'(a^n)} x_i^n$$

$$\frac{\partial E^n}{\partial b} = (y^n - t^n) f(a^n)(1 - f(a^n))$$
Applying gradient descent to a sigmoid single-layer network

\[ y = f \left( \sum_{i=1}^{5} w_i x_i + b \right) \]

\[ w_4 = w_4 - \eta (y^n - t^n) y^n(1 - y^n) x_4^n \delta^n \]

\[ f'(a^n) \]
If we use a sigmoid single layer network for a two class problem \((C_1 \text{ (target } t = 1) \text{ and } C_2 \text{ (} t = 0))\), then we can interpret the output as follows:

\[
y \sim P(C_1 \mid x) = P(t = 1 \mid x) \\
(1 - y) \sim P(C_2 \mid x) = P(t = 0 \mid x)
\]

Combining, and recalling the target is binary

\[
P(t \mid x, W) = y^t \cdot (1 - y)^{1-t}
\]

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

\[
\ln P(t \mid x, W) = t \ln y + (1 - t) \ln(1 - y)
\]
Cross-entropy error function (2)

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

$$E^n = -(t^n \ln y^n + (1 - t^n) \ln(1 - y^n)) .$$

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

- Gradient descent training requires the derivative $\partial E/\partial w_i$ (where $w_i$ connects the $i$th input to the single output).

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

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial a} \cdot \frac{\partial a}{\partial w_i}$$

$$= \frac{(y - t)}{y(1 - y)} \cdot y(1 - y) \cdot x_i = (y - t)x_i$$

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 use a “one-from-$K$” ("one-hot") 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.
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 $\sum_k P(k|x)$ is not constrained to equal 1 – we want this if we would like to interpret the outputs of the net as class probabilities.

Solution – an activation function with a sum-to-one constraint: **softmax**
Softmax

\[ y_k = \frac{\exp(a_k)}{\sum_{j=1}^{K} \exp(a_j)} \]

\[ a_k = \sum_{i=1}^{d} w_{ki} x_i + b_k \]

- 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 \( y^n_k \) as an estimate of \( P(k|x^n) \)
- Softmax is the multiclass version of the two-class sigmoid
We can extend the cross-entropy error function to the multiclass case

\[ E^n = - \sum_{k=1}^{C} t^n_k \ln y^n_k \]

Again the overall gradients we need are

\[
\frac{\partial E^n}{\partial w_{ki}} = \sum_{c=1}^{C} \frac{\partial E}{\partial y_c} \cdot \frac{\partial y_c}{\partial a_k} \cdot \frac{\partial a_k}{\partial w_{ki}} = \sum_{c=1}^{C} -\frac{t_c}{y_c} \cdot \frac{\partial y_c}{\partial a_k} \\
\frac{\partial E^n}{\partial b_k} = \sum_{c=1}^{C} \frac{\partial E}{\partial y_c} \cdot \frac{\partial y_c}{\partial a_k} \cdot \frac{\partial a_k}{\partial b_k} = \sum_{c=1}^{C} -\frac{t_c}{y_c} \cdot \frac{\partial y_c}{\partial a_k}
\]
Softmax – Training (2)

- Note that the $k$th activation $a_k$ – 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:

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

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

- We can put it all together to find:

$$\frac{\partial E^n}{\partial w_{ki}} = (y^n_k - t^n_k)x^n_i$$

$$\frac{\partial E^n}{\partial b_k} = (y^n_k - t^n_k)$$

Softmax output and cross-entropy error function results in gradients which are the delta rule!
Exercises

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

$$\frac{\partial E^n}{\partial w_{ki}} = (y^n_k - t^n_k)x^n_i$$

(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

- 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