CS-667 Advanced Machine Learning

Nazar Khan

PUCIT

Lectures 4-7
Neural Networks
March 3, 7, 9, 14 2016

Announcements

- From now onwards, lectures will take place on Mondays and Wednesdays from 8:15 am till 9:45 am.
- Project 1a is due on Monday, 7-th March.
  - Relevant material has been placed on `\\printsrv`.
  - Try to finish by tomorrow.
  - Each sample is a 784×1 vector that represents a 28×28 image. To visualise the k-th training sample as an image, you may use the following commands:

```
imagesc(reshape(train_x(k,:),28,28)');
axis image;
colormap gray;
```

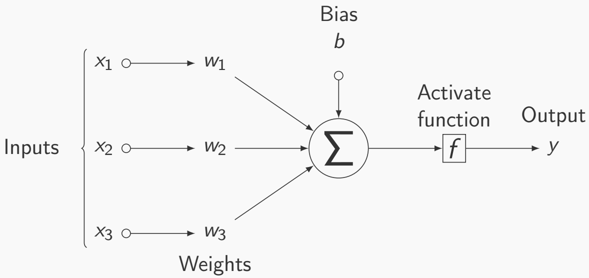
Logistic Regression Tips

- In case you are having memory issues or training takes a lot of time, you might want to use the following tips:
  - Type in the command 'doc spdiags'
  - Do not use the inv() function to for matrix inverse. Use the \ operator. For more help, consult Google or Matlab documentation.
- Also, don't forget to homogenise the inputs by appending a 1 at the end of each input. This will absorb the bias term.
- Lastly, if you start getting a warning message like "Warning: Matrix is close to singular or badly scaled", then
  - first look at the difference between Exercises 1.1 and 1.2 from Chapter 1 and their solutions.
  - then look at the programming solutions to both problems. We have done both the exercises as well as their programming solutions in CS 567.

Neural Networks

- So far, we have learned $\mathbf{w}^*$ for mapping inputs $\mathbf{x} \in \mathbb{R}^D$ to targets $\mathbf{t}$.
- Often, working in a transformed space $\phi \in \mathbb{R}^M$ makes it easier to learn the mapping.
- However, not all mappings are useful for the problem at hand. Is there an optimal mapping ϕ^*?
- Neural networks learn the optimal mapping ϕ^* and also the optimal parameters $\mathbf{w}^*$.

Neural Networks
The Neuron



- The function of a biological neuron can be modelled as $y = f\left(\sum_j w_j x_j + b\right)$.

Neural Networks
The Neuron

- Model the output of the k-th neuron as
  $y_k = f(a_k) = f(\mathbf{w}_k^T \mathbf{x}) = f\left(\sum_j w_{kj} x_j\right)$ where
  - The x_j constitute values of input signals feeding into the neuron.
  - The w_{kj} are weights determining the importance given to input x_j by this neuron.
  - Dot-product $a_k = \sum_j w_{kj} x_j$ is called the *activation*.
  - $f(\cdot)$ is called the *activation function*. Determines behaviour of the neuron in response to its activation.
- The perceptron that we studied earlier is a very simple neuron model with f being the step function.

$$f(a) = \begin{cases} 1 & \text{if } a \geq 0 \\ 0 & \text{if } a < 0 \end{cases} \tag{1}$$

Neural Networks

- The linear models that we covered can be represented as

$$y_k(\mathbf{x}, \mathbf{W}) = f(\mathbf{w}_k^T \phi(\mathbf{x})) = f\left(\sum_{j=0}^{M} w_{kj}\phi_j(\mathbf{x})\right)$$

where y_k is the k-th output and index j starts from 0 to reflect bias inclusion.

$$f = \begin{cases} \text{identity} & \text{for regression} \\ \text{logistic sigmoid} & \text{for binary classification} \\ \text{softmax} & \text{for multiclass classification} \end{cases}$$

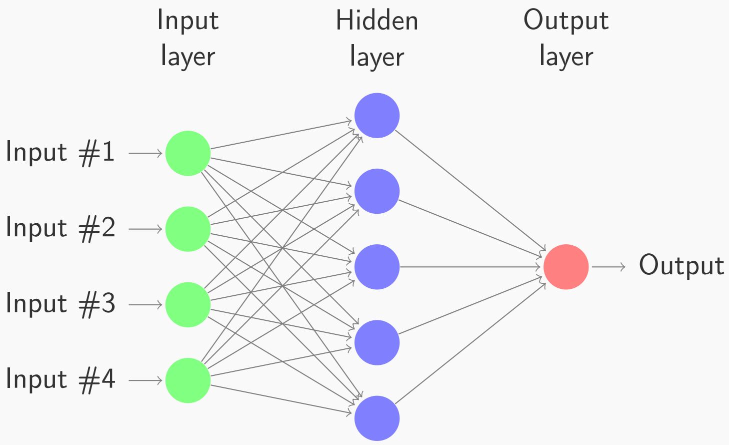
- Each $\phi_j(\mathbf{x})$ can be seen as a *basis function*.
- So far, the basis functions were fixed. Now we *adapt* them to the problem.

Neural Networks

- Model $\phi_j(\mathbf{x})$ as a non-linear function $h(a_j)$ where *activation* $a_j = \mathbf{w}_j^T\mathbf{x}$ with adjustable parameters $\mathbf{w}_j$.
- So the k-th output can be written as

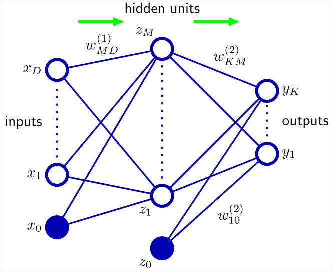
$$\begin{aligned} y_k(\mathbf{x}, \mathbf{W}) = f(a_k) = f(\mathbf{w}_k^T\boldsymbol{\phi}(\mathbf{x})) &= f\left(\sum_{j=0}^{M} w_{kj}\phi_j(\mathbf{x})\right) \\ &= f\left(\sum_{j=0}^{M} w_{kj}h(a_j)\right) \\ &= f\left(\sum_{j=1}^{M} w_{kj}h\left(\sum_{i=0}^{D} w_{ji}x_i\right)\right) \end{aligned}$$

Neural Networks



These computations can be visualised graphically as *forward propagation of information* through the so-called *neural network*.

Neural Networks



A two-layer neural network.

- 3 neuron types:
  - input x_i
  - hidden z_j
  - output y_k
- 2 weight layers:
  - hidden-input $w^{(1)}_{ji}$
  - output-hidden $w^{(2)}_{kj}$
- To differentiate between different layer parameters, we can write

$$y_k(\mathbf{x}, \mathbf{W}) = f\left(\sum_{j=1}^{M} w^{(2)}_{kj} h\left(\sum_{i=0}^{D} w^{(1)}_{ji} x_i\right)\right)$$

Neural Networks
As Multilayer Perceptrons

- The computation $y_k(\mathbf{x}, \mathbf{W}) = f\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)\right)$ can be viewed in two stages:
  1. Compute $z_j = h(\mathbf{w}_j^T \mathbf{x})$, followed by
  2. $y_k = f(\mathbf{w}_k^T \mathbf{z})$.
- Both stages resemble the perceptron model.
- Therefore, another name for such neural networks is *multilayer perceptrons* or simply *MLP*.
- However, there is a key difference:
  - Perceptron uses a non-differentiable step-function non-linearity.
  - MLP uses a differentiable sigmoidal non-linearity. So we can train via gradient based approaches.
- Therefore, despite the name, MLPs never use perceptrons!

Neural Networks
As Universal Approximators

- Neural networks are considered to be *universal approximators*.
- A two-layer network with linear outputs can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy.
  - Provided that the network has a sufficiently large number of hidden units.

Neural Networks for Regression
Univariate

- In the polynomial fitting example from Chapter 1, given inputs and targets $\{\mathbf{x}_n, t_n\}$, we wanted to find the *optimal parameters $\mathbf{w}^*$ of the polynomial* that best fits the data.
- Assuming i.i.d data and $t_n \sim \mathcal{N}(y(\mathbf{x}_n, \mathbf{w}), \beta^{-1})$, we wrote the likelihood function whose maximisation corresponded to minimisation of the SSE function

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y(\mathbf{x}_n, \mathbf{w}) - t_n)^2$$

- By replacing the polynomial in function $y(\mathbf{x}_n, \mathbf{w})$ by the neural network function, we can minimise $E(\mathbf{w})$ to find *optimal parameters $\mathbf{w}^*$ of the neural network*.

Neural Networks for Regression
Multivariate

- Similarly, for multivariate targets, assuming multivariate Gaussian density leads to the SSE function

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} ||\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n||^2$$

where $y_k = a_k = \mathbf{w}_k^T\mathbf{x}$.

- Notice and prove that

$$\frac{\partial E_n}{\partial a_k} = \underbrace{(y_{kn} - t_{kn})}_{\text{error}_n}$$

Neural Networks for Classification
Binary

- Similarly, for binary classification, we can assume a Bernoulli distribution on targets which leads to minimisation of the *cross-entropy* function

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \left(t_n \ln y(\mathbf{x}_n, \mathbf{w}) + (1 - t_n) \ln(1 - y(\mathbf{x}_n, \mathbf{w}))\right)$$

where $y(\mathbf{x}_n, \mathbf{w}) = P(\mathcal{C}_1 | \mathbf{x}_n) = \sigma(a) = \sigma(\mathbf{w}^T \mathbf{x}_n)$.

- Notice (and prove) that

$$\frac{\partial E_n}{\partial a} = \underbrace{(y_n - t_n)}_{\text{error}_n}$$

Neural Networks for Classification
Multiclass

- For multiclass classification, we can minimise the *multiclass cross-entropy* function

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K}\left(t_{kn}\ln y_k(\mathbf{x}_n,\mathbf{w})\right)$$

where $y_k(\mathbf{x}_n,\mathbf{w}) = P(\mathcal{C}_k|\mathbf{x}_n) = \frac{e^{a_k}}{\sum_{j=1}^{K} e^{a_j}}$ and $a_k = \mathbf{w}_k^T\mathbf{x}$.

- Notice (and prove) that

$$\frac{\partial E_n}{\partial a_k} = \underbrace{(y_{kn} - t_{kn})}_{\text{error}_n}$$

- In the following we will denote the error $\frac{\partial E}{\partial a_k}$ as δ_k.

Neural Networks for Classification

- Note that we can learn classifiers via SSE minimisation also, but the cross-entropy formulations
  1. can be derived probabilistically,
  2. train faster, and
  3. generalise better.

Optimisation

- For optimisation, we notice that $\mathbf{w}^*$ must be a *stationary point* of $E(\mathbf{w})$.
  - Minimum, maximum, or saddle point.
  - A saddle point is where gradient vanishes but point is not an extremum (Example).
- The goal in neural network minimisation is to find a local minimum.
- A global minimum, even if found, cannot be verified as globally minimum.
- Due to symmetry, there are multiple equivalent local minima. Reaching any suitable local minimum is the goal of neural network optimisation.
- Since there are no analytical solutions for $\mathbf{w}^*$, we use iterative, numerical procedures.

Optimisation

- Options for iterative optimisation
  - Online methods
    - Stochastic gradient descent
    - Stochastic gradient descent using mini-batches
  - Batch methods
    - Batch gradient descent
    - Conjugate gradient descent
    - Quasi-Newton methods
- Online methods
  - converge faster since parameter updates are more frequent, and
  - have greater chance of escaping local minima because stationary point w.r.t to whole data set will generally not be a stationary point w.r.t an individual data point.
- Batch methods: Conjugate gradient descent and quasi-Newton methods
  - are more robust and faster than batch gradient descent, and
  - decrease the error function at each iteration until arriving at a minimum.

Backpropagation

- For all gradient based methods, however, we must first compute the gradient $\nabla_{\mathbf{w}} E(\mathbf{w})$.
- We have seen that many error functions of practical interest can be written as a sum of terms

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

- So the essential gradient is $\nabla_{\mathbf{w}} E_n(\mathbf{w})$ which we write in its complete form $\nabla_{\mathbf{w}} E(\mathbf{y}(\mathbf{x}_n, \mathbf{w}))$.

Backpropagation

- For the output layer weights

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}} = \delta_k z_j$$

- For the hidden layer weights, using the chain rule

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} = \underbrace{\sum_{k=1}^{K} \underbrace{\frac{\partial E}{\partial a_k}}_{\delta_k} \underbrace{\frac{\partial a_k}{\partial z_j}}_{w_{kj}} \underbrace{\frac{\partial z_j}{\partial a_j}}_{h'(a_j)}}_{\delta_j} \underbrace{\frac{\partial a_j}{\partial w_{ji}}}_{x_i} = \delta_j x_i$$

- For each layer, notice the familiar form

$$\text{gradient} = \text{error} \times \text{input}$$

Backpropagation

- It is important to note that

$$\delta_j = h'(a_j) \sum_{k=1}^{K} \delta_k w_{kj}$$

  yields the error δ_j at hidden neuron j by *backpropagating* the errors δ_k from all output neurons that use the output of neuron j.
- More generally, compute error δ_j at a layer by *backpropagating* the errors δ_k from next layer.
- Hence the names *error backpropagation*, *backpropagation*, or simply *backprop*.
- Very useful machine learning technique that is not limited to neural networks.

Backpropagation

$$\delta_j^{(1)} = h'(a_j) \sum_{k=1}^{K} \delta_k^{(2)} w_{kj}$$

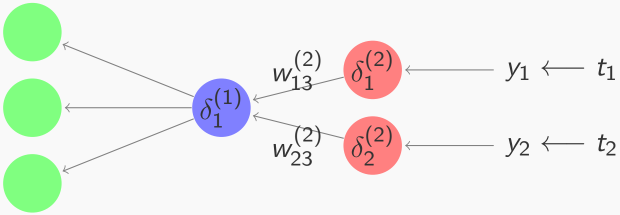


Figure: Visual representation of backpropagation of delta values of layer $l+1$ to compute delta values of layer l.

Backpropagation

Learning Algorithm

1. Forward propagate the input vector $\mathbf{x}_n$ to compute activations and outputs of every neuron in every layer.
2. Evaluate δ_k for every neuron in output layer.
3. Evaluate δ_j for every neuron in <u>every</u> hidden layer via backpropagation.
4. Compute derivative of each weight via $\delta\times$input.
5. Update each weight.

Background Math
A $(-1, 1)$ sigmoidal function

- Since range of logistic sigmoid $\sigma(a)$ is $(0, 1)$, we can obtain a function with $(-1, 1)$ range as $2\sigma(a) - 1$.
- Another related function with $(-1, 1)$ range is the tanh function.

$$\tanh(a) = 2\sigma(2a) - 1 = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

  where σ is applied on $2a$.
- Preferred over logistic sigmoid as activation function $h(a)$ of hidden neurons. (Read Yann LeCun's "Efficient Backprop" paper to understand why.)
- Just like the logistic sigmoid, derivative of $\tanh(a)$ is simple: $1 - \tanh^2(a)$. (Prove it.)

Backpropagation
A Simple Example

- Two-layer MLP for multivariate regression from $\mathbb{R}^D \longrightarrow \mathbb{R}^K$.
- Linear outputs $y_k = a_k$ with SSE $E_n = \frac{1}{2}\sum_{k=1}^{K}(y_k - t_k)^2$.
- M hidden neurons with $\tanh(\cdot)$ activation functions.

Forward propagation

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i$$

$$z_j = \tanh(a_j)$$

$$y_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j$$

$$\delta_k = y_k - t_k$$

Backpropagate

$$\delta_j = (1 - z_j^2)\sum_{k=1}^{K} w_{kj}^{(2)} \delta_k$$

- Compute derivatives $\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \delta_j x_i$ and $\frac{\partial E_n}{\partial w_{kj}^{(2)}} = \delta_k z_j$.

Backpropagation
Verifying Correctness

- *Numerical derivatives* can be computed via finite *central differences*

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon} + O(\epsilon^2)$$

- *Analytical derivatives* computed via backpropagation **must be compared** with numerical derivatives for a few examples to verify correctness.
- Any implementation of analytical derivatives (not just backpropagation) must be compared with numerical derivatives.
- Notice that we could have avoided backpropagation and computed all required derivatives numerically.
  - But cost of numerical differentiation is $O(W^2)$ while that of backpropagation is $O(W)$ where W is the total number of weights (and biases) in the network. (Why?)

Project 2
Backpropagation for MLPs

- Implement the backpropagation algorithm for training an MLP.
  - Code up a generic implementation.
  - Verify correctness of analytical derivatives.
  - Understand the experiment and network used for Figure 5.3 in Bishop's book.
  - Regenerate Figure 5.3 using your implementation.
- Submit your_roll_number_MLP.zip containing
  - code,
  - generated image, and
  - report.txt/pdf explaining your results.
- Due Monday (March 21, 2016 before 5:30 pm) on \\printsrv.

Regularization in Neural Networks

- Recall that over-fitting can be lessened via regularization.
  1. Penalise magnitudes of weights: $\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$.
  2. Separately penalise magnitudes of weights of each layer: $\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \sum_{l=1}^{L} \frac{\lambda_l}{2}\mathbf{w}^{(l)^T}\mathbf{w}^{(l)}$.
  3. *Early stopping* by checking $E(\mathbf{w})$ on a validation set. Stop when error on validation set starts increasing.
  4. *Tangent propagation*: $\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \underbrace{\frac{\partial y}{\partial x}}_{\text{rough idea}}$.
  5. Training with transformed data.
  6. Building invariance into the network structure. We cover this in the next lecture on Convolutional Neural Networks.