6 Neural Networks

Contents

Abstract

This chapter introduces how we can dealing with non-linear hypotheses with neural network. Although because I have completed the dedicated course "Neural networks for machine learning" on Coursera, please see this course for more explicit details (to avoid duplication). This chapter predominately discusses neural networks for classification.

6.1 Key Remarks

1. Neural networks are one of a few ML algorithms where we continue to improve our prediction accuracy as we increase the number of training examples (most others tend to level off even if we increase the data by a significant amount).

2. Build models that use hardware as efficiently as possible (that way when hardware improves your learning speeds up essentially for free).

3. Set a baseline as the fastest your code can ever run, i.e. the speed of light, or in speech recognition it's 0% WER (improving speed by 10x on already badly written code is not an effective way to analyse this).

4. As discussed at `https://youtu.be/MVyauNNinC0?t=36m18s`, one of the best ways to train a DNN is to try to fool/trick them.

6.2 Representation

Notation 1. **Activation units and vector representation of neural networks**

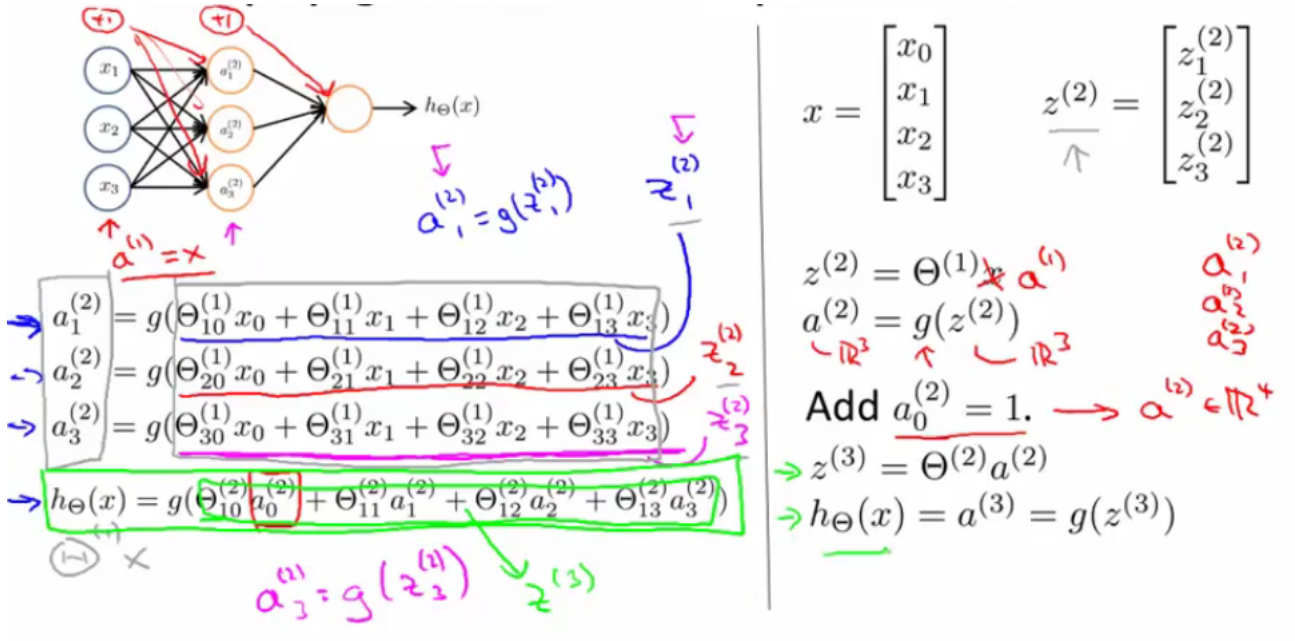

Figure 6.1: A network with s_j units in layer j, s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$. The process of ***computing*** $h_\Theta(x)$ is called **forward propagation** (since we must start with the activations of the input units, and then we forward propagate these to compute the activations in the hidden layer and so on). *(c.f back propagation, where we are given $h_\Theta(x)$ and want to learn the weights/parameters.)*

Note. There is a Θ matrix for each layer l in the network, with $\Theta^{(l-1)}_{a_l, a_{l-1}}$ (a_l :=index of activation node in layer l being computer, a_{l-1} :=index of activation node in layer $l-1$ being multiplied by).

Remark. Looking at how we compute each activation unit, $a_i^{(j)}$, we notice that what this neural network is just doing logistic regression (with **"learned" inputs**).

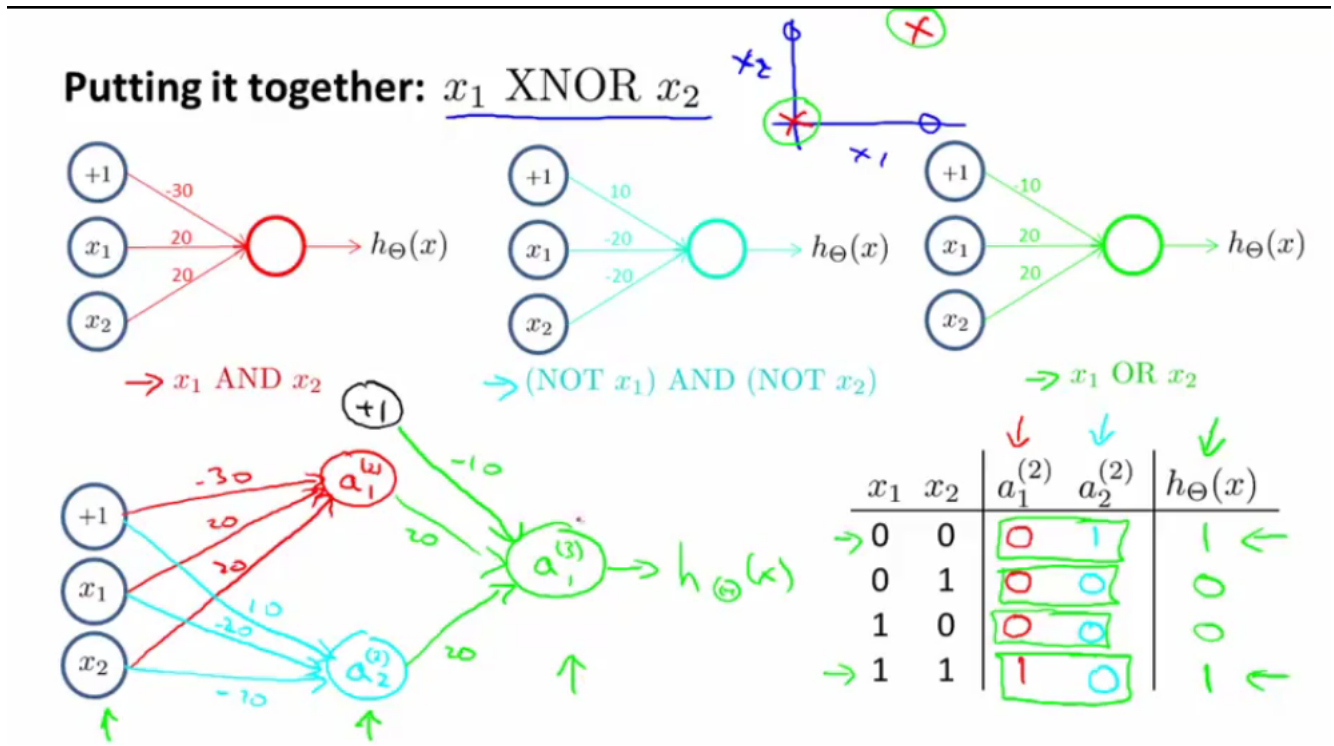

Figure 6.2: Neural network are used to learn more complex functions with each layer.

6.3 Multiple output units: One-vs-all

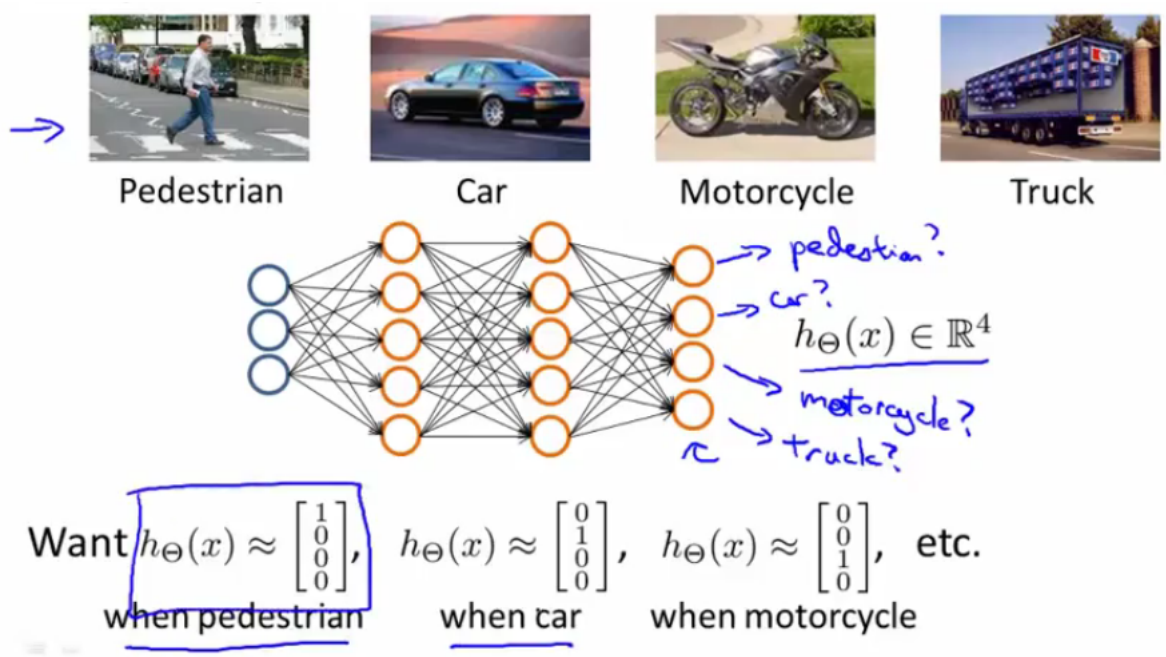

Figure 6.3: In the one vs. all (equal to 4 here) representation, instead of letting $y \in \{0,1\}$, we let $y \in \left\{[1,0,0,0]^T, [0,1,0,0]^T, [0,0,1,0]^T, [0,0,0,1]^T\right\}$.

6.4 Evaluation: Cost Function

Notation 2.

$$\begin{aligned} L &:= \text{total } \# \text{ layers in net work} \\ s_l &:= \# \text{ units (excluding bias unit) in layer } l \\ s_L &:= \# \text{ units in output layer} \\ K &:= \# \text{ output units/classes} \end{aligned}$$

The cost function for neural networks is a generalisation of the cost function used from logistic regression (with regularisation); so $h_\Theta(x) \in \mathbb{R}^K$ (i.e. $(h_\Theta(x))_i$ =probability that the i^{th} output is 1), and we take the sum over these K output units in the cost function:

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(h_\Theta\left(x^{(i)}\right)\right)_k + \left(1-y_k^{(i)}\right)\log\left(1-\left(h_\Theta\left(x^{(i)}\right)\right)_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{ji}^{(l)}\right)^2$$

Remark. The first half of the cost function is saying ("average sum of logistic regression")

- For each training data example $(1, ..., m)$.
  - Some for each position in the output vector.

6.5 Optimisation: Stochastic gradient descent using backpropagation to compute the gradient.

To find $\min_\Theta J(\Theta)$ using gradient descent or one of the advanced optimisation algorithms, we need to compute $J(\Theta)$ and $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$ (recall $\Theta_{ij}^{(l)} \in \mathbb{R}$).

6.5.1 Gradient Computation for the case of one training example

Computing the gradient in the case of one training example $(\mathbf{x}, \mathbf{y})$, we see that:

Given one training example (x, y):

Forward propagation:

$$a^{(1)} = x$$
$$z^{(2)} = \Theta^{(1)} a^{(1)}$$
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$
$$z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$
$$z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$

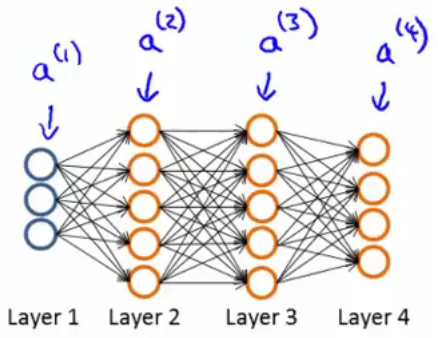

Figure 6.4: **Step 1 - Forward Propagation (Calculating the activations $\mathbf{a}^{(i)}$)**
This algorithm takes the initial input into that network and pushes the input through the network to the generation of an output hypothesis.

Gradient computation: Backpropagation algorithm

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.

For each output unit (layer L = 4)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

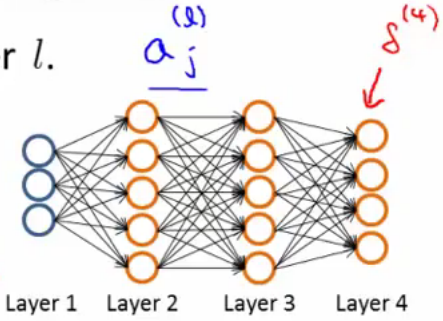

Figure 6.5: **Step 2 - Backpropagation** (Calculating the gradients $\boldsymbol{\delta}^{(i)}$)
This algorithm takes the output you got from your network, compares it to the real value y and calculates how wrong the network was (i.e. how wrong the parameters were) It then, using the error you've just calculated, back-calculates the error associated with each unit from the preceding layer (i.e. layer $L-1$). These "error" measurements for each unit can be used to calculate the partial derivatives, which can then be used to calculate the partial derivative for gradient descent.

Note. The $.*$ denotes element-wise multiplication.

Remark. We do not calculate $\boldsymbol{\delta}^{(1)}$ as the first layer does not have any error associated with it (it's just the features we observed in our training set).

Remark. When the cost function is **not** regularised (i.e. $\lambda = 0$), it can be shown that $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$.

6.5.2 Gradient Computation for m training examples

For a training set $\{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$:

Algorithm 1. ***Backpropagation Algorithm***
Set $\Delta_{ij}^{(l)} = 0$, $\forall l, i, j$ (i.e. this has each node as one dimension and each training data example as the other).
For $i = 1, ..., m$*:*
Set $a^{(1)} = x^{(i)}$
Perform forward propagation to compute $a^{(l)}$ *for* $l = 2, 3, ..., L$
Using $y^{(i)}$*, compute* $\delta^{(L)} = a^{(L)} - y^{(i)}$
Compute $\delta^{(L-1)}$,$\delta^{(L-2)}$,...,$\delta^{(2)}$
$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ *(or vectorised form* $\mathbf{\Delta}^{(l)} := \mathbf{\Delta}^{(l)} + \delta^{(l+1)} \left(a^{(l)}\right)^T$*)*

$$D_{ij}^{(l)} := \begin{cases} \frac{1}{m}\Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{m}\Delta_{ij}^{(l)} & \text{if } j = 0 \end{cases}$$

Remark. For quite a complicate proof, it can be shown that the partial derivatives $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$.

6.6 Backpropagation in practice

6.6.1 Unrolling parameters

With neural networks, we are working with sets of matrices:

$$\Theta_1, \Theta_2, \Theta_3, ...; D_1, D_2, D_3, ...$$

In order to use optimizing functions such as `fminunc()`, we need to "unroll" all the elements and put them into one long vector (MATLAB):

```
thetaVector = [ Theta1(:); Theta2(:); Theta3(:) ];
deltaVector = [ D1(:); D2(:); D3(:) ];
```

6.6.2 Gradient checking

Backpropagation has a lot of details. Small bugs can easily creep in so that it looks like $J(\Theta)$ is decreasing, but in reality it may not be decreasing by as much as it should.
We can approximate the derivative of our cost function, for some small $\varepsilon > 0$, as

$$\frac{\partial}{\partial \Theta} J(\Theta) \approx \frac{J(\Theta + \varepsilon) - J(\Theta - \varepsilon)}{2\varepsilon}$$

and for multiple theta matrices, we can approximate the derivative w.r.t. Θ_j as

$$\frac{\partial}{\partial \Theta_j} J(\Theta) \approx \frac{J(\Theta_1, ...\Theta_{j-1}, \Theta_j + \varepsilon, \Theta_{j+1}, ..., \Theta_n) - J(\Theta_1, ...\Theta_{j-1}, \Theta_j - \varepsilon, \Theta_{j+1}, ..., \Theta_n)}{2\varepsilon}$$

Remark. Andrew Ng typically uses $\varepsilon = 10^{-4}$.

Finally check that gradApprox ≈ deltaVector.

Note. Once you've verified once that your backpropagation algorithm is correct, then you don't need to compute gradApprox again. The code to compute gradApprox is very slow.

6.6.3 Random initialisation

Initializing all theta weights to 0 does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly. Thus we use **random initialization**, which serves the purpose of symmetry breaking.
Initialise each $\Theta^{(l)} \in \mathbb{R}^{n_{l-1}, n_l}$ (where $n_l :=$# neurons in layer l (I WORKED THIS OUT LATE - CHECK)) to a random value between $[-\varepsilon, \varepsilon]$ with

$$\varepsilon = \frac{\sqrt{6}}{\sqrt{n_l + n_{l-1}}}$$

E.g. $\Theta^{(1)} = \texttt{rand}(n_{l-1}, n_l) * 2\varepsilon - \varepsilon \in \mathbb{R}^{n_{l-1}, n_l}$ (where the `rand` function produces a $n_{l-1} \times n_l$ matrix).

6.6.4 Momemtum [1]

Momentum changes the path you take to the optimum.

Once you have an objective function, you have to decide how to move around on it. Steepest descent on the gradient is the simplest approach, but you're right that fluctuations can be a big problem. Adding momentum helps solve that problem. If you're working with batch updates (which is usually a bad idea with neural networks) Newton-type steps are another option. The new "hot" approaches are based on Nesterov's accelerated gradient and so-called "Hessian-Free" optimization.

6.7 Implementation summary and some best practices of using neural networks for machine learning

1. **REPRESENTATION:** Pick a network architecture, including how many hidden units there are in each layer and how many layers in total.
   (a) The higher the # hidden units per layeris usually produces more accurate results, but we must take into account the computation cost.
   (b) # *hidden layers* :
      i. *"1 hidden layer is sufficient for the large majority of problems".*
      ii. If the data **linearly separable**, then you **don't need any hidden** layers at all.
      iii. DNN=more than 3; most papers on deep learning use networks with 5-7 layers.
      iv. Geoff Hinton: "Add layers until you overfit (if you're not overfitting, your network isn't big enough.). Then add dropout or another regularization method."
   (c) # neurons for each layer:
      i. Roughly same for all hidden layers and usually at 1 to 3 times the input variables (or mean of the # input and # output nodes).
      ii. Yoshia Bengio: "It is a hyper-parameter to be optimized, as usual."
      iii. WARNINGS:
         A. **Too few nodes** will lead to high error for your system as the predictive factors might be too complex for a small number of nodes to capture.
         B. **Too many nodes** will overfit to your training data and not generalise well.
   (d) # input units = dimension of features $x^{(i)}$; # output units = # classes
2. **EVALUATION:** Randomly initialise the weights (as discussed above).
3. **EVALUATION:** Implement forward propagation to get $h_\theta\left(x^{(i)}\right)$.
4. **EVALUATION:** Compute the cost function
5. **EVALUATION:** Implement back propagation to compute partial derivatives.

```
for i = 1:m {
     Forward propagation on (x^i, y^i) --> get activation (a) terms
     Back propagation on (x^i, y^i) --> get delta (δ) terms
     Compute Δ := Δ^l + δ^(l+1)(a^l)^T
}

With this done compute the partial derivative terms
```

Figure 6.6: Backpropagation is usually done with a for loop over training examples, like above. It can be done without a for loop, but this is much more complicated way of doing things.

   (a)
   (b) Use gradient checking to confirm your back propagation works. Then disable gradient checking.

[1] http://stats.stackexchange.com/a/70146/75092

6. **OPTIMISATION:** Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

   (a) If $J(\Theta)$ is not convex, then some algorithms can be susceptible to local minimum. In practice this is not usually a huge problem, but we can't guarantee programs with find global optimum should find good local optimum at least.

6.7.1 Dealing with overfitting

Data Augmentation

The easiest and most common method to reduce overfitting on image data is to artificially enlarge the dataset using label-preserving transformations (e.g. by flipping the image, rotating it slightly, translating the image, altering the intesites of pixels). From Krizhevsky et al.'s seminal 2012 paper on ImageNet classification (http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf):

- The first form of data augmentation consists of generating image translations and horizontal reflections. We do this by extracting random 224×224 patches (and their horizontal reflections) from the 256×256 images and training our network on these extracted patches. This increases the size of our training set by a factor of 2048, though the resulting training examples are, of course, highly interdependent [meaning highly dependent on each other]. Without this scheme, our network suffers from substantial overfitting, which would have forced us to use much smaller networks.

Dropout

[c.f. random forests; "averages noisy models to create a model with low variance."]

This technique consists of setting to zero the output of each hidden neuron with probability 0.5. The neurons which are "dropped out" in this way do not contribute to the forward pass and do not participate in back-propagation. So every time an input is presented, the neural network samples a different architecture, but all these architectures share weights.

This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.

Remark. From Krizhevsky again: "Without dropout, our network ex- hibits substantial overfitting. Dropout roughly doubles the number of iterations required to converge."