Artificial Neural Networks

2nd February 2017, Aravindh Mahendran, Student D.Phil in Engineering Science, University of Oxford

Contents

Introduction 1
Perceptron 2
- LMS / Delta Rule for Learning a Perceptron model 4
- Demo for Simple Synthetic Dataset 4
- Problems 6
- Stochastic Approximation of the Gradient 6

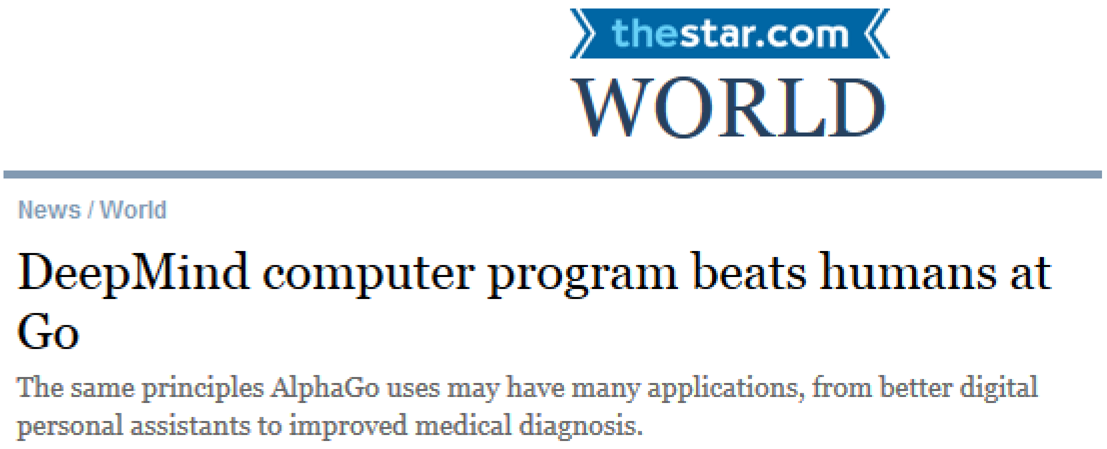
Sigmoid Unit 6
- Backpropagation algorithm 7
- Backpropagation Algorithm for Vector Input and Vector Output 9
- Problems 9
- Demo on Face Pose Estimation 9
- Expressive Power 12

Deep Learning 12
- LeNet 13

Conclusion 14
References 14

Introduction

Artificial Neural Networks (ANN) are a class of models that have been successfully used in several machine learning problems. Recent successes include the imagenet image classification challenge (http://www.image-net.org/challenges/LSVRC/2012/results.html) where team SuperVision (Krizhevsky et.al. 2012) beat the non-ANN competitors by a very large margin. Deep Artificial Neural Networks have recently made news in the context of being able to play ATARI games, beating the European Champion on the Chinese Go, etc .


thestar.com

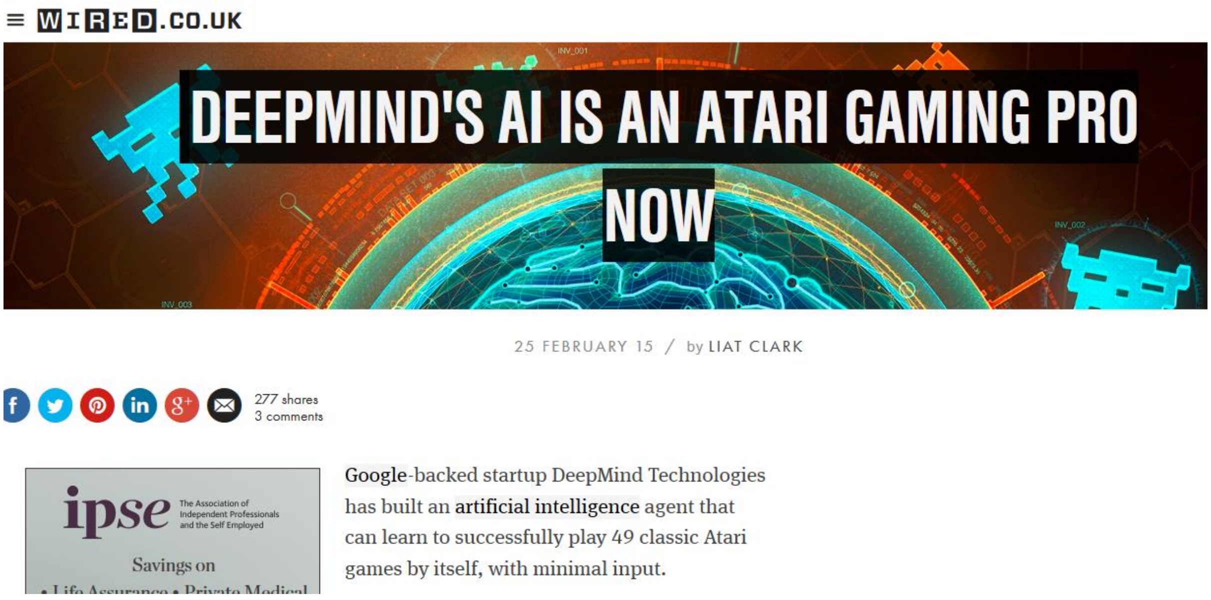
WORLD

News / World

DeepMind computer program beats humans at Go

The same principles AlphaGo uses may have many applications, from better digital personal assistants to improved medical diagnosis.

Screenshot from - http://www.thestar.com/news/world/2016/01/27/deepmind-computer-program-beats-humans-at-go.html


≡ WIRED.CO.UK

DEEPMIND'S AI IS AN ATARI GAMING PRO NOW

25 FEBRUARY 15 / by LIAT CLARK

277 shares
3 comments

ipse The Association of Independent Professionals and the Self Employed

Savings on

Google-backed startup DeepMind Technologies has built an artificial intelligence agent that can learn to successfully play 49 classic Atari games by itself, with minimal input.

Screenshot from http://www.wired.co.uk/news/archive/2015-02/25/google-deepmind-atari

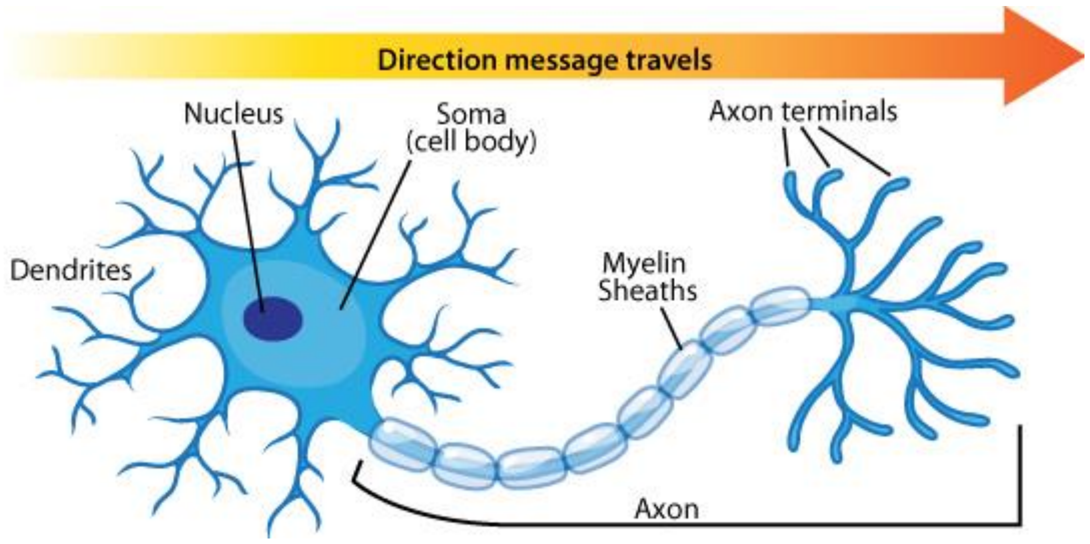
ANNs are good for problems where the nature of the target function is hard to guess. Also they are really slow to train. The imagenet winner in 2012 took 6 days to train on 2 GPUs but the trained model is really fast at test time.

Perceptron

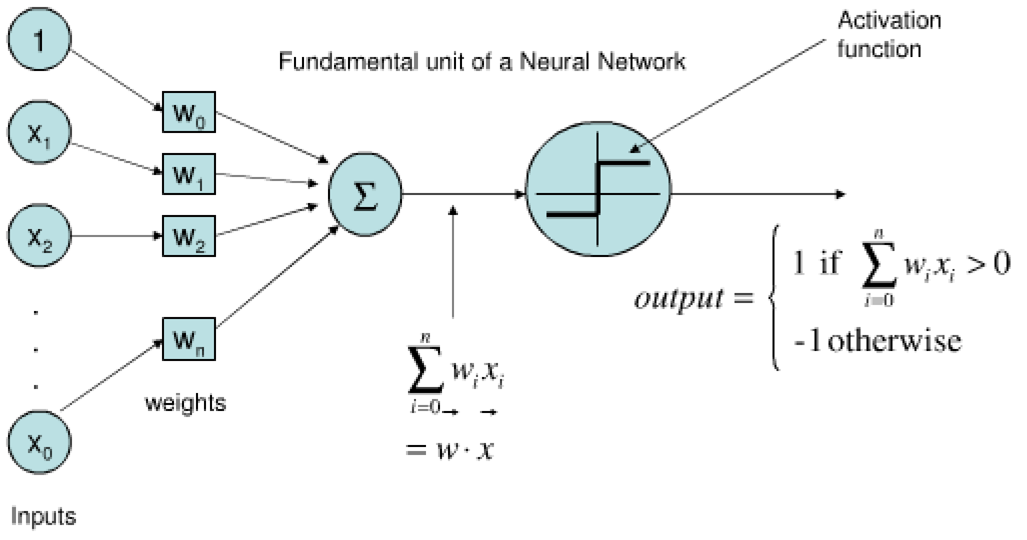
People studied the real neuron and made a very simple mathematical model of it called the perceptron. This was proposed by Rosenblatt in 1957 and it took a vector x ∈ R^n as input and gave a scalar output

$$y = \begin{cases} 1\, if\, \boldsymbol{w}^t\boldsymbol{x} > 0 \\ -1\, otherwise \end{cases}$$

Compare below a depiction of a real neuron and the perceptron model.



https://askabiologist.asu.edu/sites/default/files/resources/articles/neuron_anatomy.jpg


Fundamental unit of a Neural Network
Activation function
1
x_1
x_2
x_0
w_0
w_1
w_2
w_n
weights
Inputs
Σ
$\sum_{i=0}^{n} w_i x_i = \vec{w} \cdot \vec{x}$
$output = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

http://tex.stackexchange.com/questions/104334/tikz-diagram-of-a-perceptron

What differences can you guess between the real neuron and the perceptron model?

A biological neuron's output is in the form of voltage spikes communicated via neurotransmitters. The perceptron has a binary output that is not time varying.

Then why this model?

The firing rate of a biological neuron can be plotted against aggregated input voltage. The resulting curve is like a sigmoid function but not exactly a sigmoid function. The threshold activation function is an approximation of this. Thus the output of a perceptron unit can be thought of as the firing rate of a neuron rather than the neuron output itself.

The perceptron unit is parametrized by the $\boldsymbol{w} \in \boldsymbol{R}^{\boldsymbol{n}}$ vector. It can represent any linear decision boundary in n dimensional space. Note that the first input is $x_0 = 1$ which accommodates the offset term of the hyperplane neatly in vector notation. For any given machine learning problem, say the classification of apples against oranges, we will need to learn this $\boldsymbol{w}$ from training data.

LMS / Delta Rule for Learning a Perceptron model

The original method used for learning a perceptron is called the “Perceptron rule”. In this lecture we shall not discuss it as it is rarely used now a days. Instead we shall discuss the gradient descent method applied to perceptron learning. This is called LMS rule or delta rule. This rule minimizes the following loss function:-

$$E_D = \frac{1}{2N}\sum_{d=1}^{N}(y_d - \widehat{y_d})^2$$

Where y_d is the ground truth target value (say +1 for apples and -1 for oranges) and $\widehat{y_d} = \boldsymbol{w}^t\boldsymbol{x}$ is the prediction corresponding to the linear part of the perceptron. Why minimize this error? Because, if we pick a **w** that minimizes this error we can be confident that the perceptron will output +1 for apples and -1 for oranges and thus separate them.

This error is minimized using gradient descent. What is gradient descent? Recall that the gradient of a function is the direction of steepest increase of that function. Thus if we move along the negative of this direction then we shall be following a path of steepest decrease in that function value. Thus the gradient descent method simply calculates the gradient of E_D and moves in its opposite direction. Because this property of the gradient is true only locally, we follow the chosen direction for a small step size η and then recalculate a new direction of steepest change. This η is a hyper-parameter typically chosen by hand.

The pseudo code is shown below:-

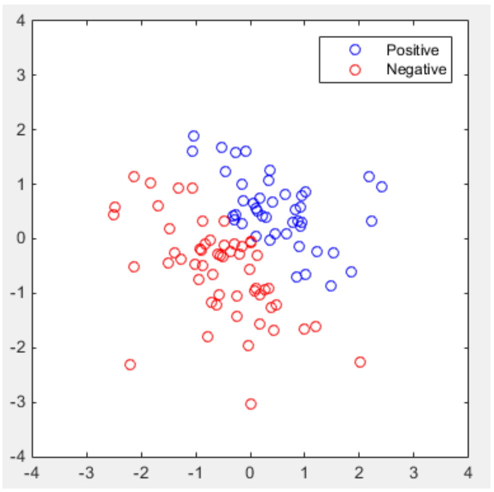
```
1. Initialize w to small random values
2. Repeat until satisfied
    a. $\nabla_{\mathbf{w}}(E_D) = \frac{1}{N}\sum_{d=1}^{N}(\boldsymbol{y_d} - \widehat{\boldsymbol{y_d}})(-\boldsymbol{x_d})$
    b. $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta\boldsymbol{\nabla}_{\boldsymbol{w}}(E_D)$
```

The gradient descent procedure simply follows the direction of negative of the gradient to minimize the loss function E_D.

Demo for Simple Synthetic Dataset

The purpose of this demo is to understand LMS/Delta rule and how it is implemented in matlab.

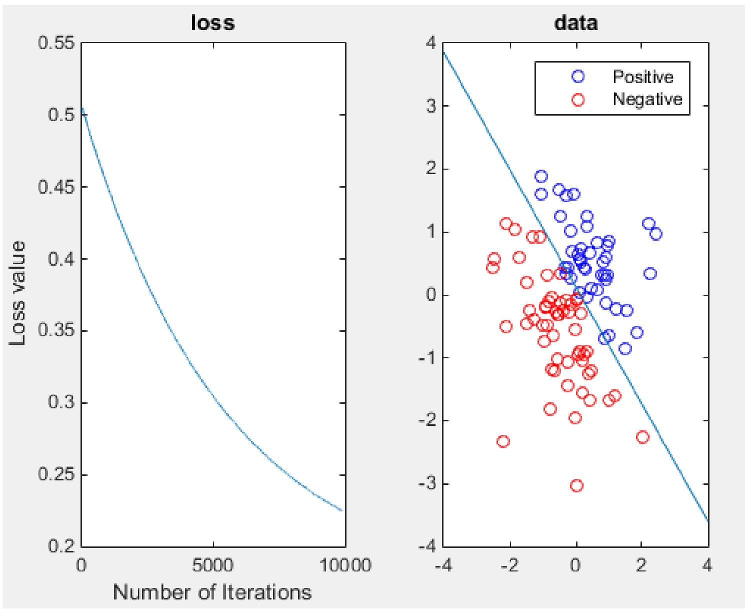
The dataset is shown below



The source code is as follows:-

```
clear;
close all;

%% Load and prepare the data
load('data.mat'); %loads X_positive X_negative
X = [X_positive, X_negative];
X_augmented = [X; ones(1, size(X, 2))];
Y_target = [ones(1, size(X_positive, 2)), -ones(1, size(X_negative, 2))];

%% Initialize the weights and set the hyperparameters
w = randn(3, 1) * 0.01;
eta = 0.0001;
loss = zeros(10000,1);
maxIterations = 10000;

for i=1:maxIterations % 10000 iterations of gradient descent
    Y_predicted = w' * X_augmented;
    loss(i) = sum((Y_target - Y_predicted).^2)/(2*numel(Y_predicted));
    grad = sum(bsxfun(@times, (Y_target - Y_predicted), -X_augmented),
2)/numel(Y_predicted);
    w = w - eta * grad;

    if(mod(i, 100) == 1)
        figure(1);
        clf;
        subplot(1,2,1);
        plot(loss(1:i));
        title('loss');
        xlabel('Number of Iterations');
        ylabel('Loss value');
        subplot(1,2,2);
        plot(X_positive(1,:), X_positive(2,:), 'bo');
        hold on;
        plot(X_negative(1,:), X_negative(2,:), 'ro');
        legend('Positive', 'Negative');
        axis([-4,4,-4,4]);
        x1 = -4;          y1 = (-w(3) - w(1)*x1)/w(2);
        x2 = 4;           y2 = (-w(3) - w(1)*x2)/w(2);
        line([x1; x2], [y1; y2]);
        title('data');
        drawnow;
        pause(1);
    end
end
```

This results in the following visual output.



Our loss function is convex so we are guaranteed to converge to the right answer. I have used a very small learning rate in the above demo to be able to show how the line is moving towards the true solution.

Problems

1. The linear perceptron cannot classify non-linearly separable data such as images of cats and dogs.

   Solution. Use multiple layers of perceptrons but with a differentiable non-linearity such as a sigmoid unit discussed later in these notes.

2. The dataset can be very large. It can take several hours to compute a single gradient of the loss over dataset D.

   Solution. Use the stochastic approximation of the gradient using a single sample or a group of samples. This is discussed in the section below.

Stochastic Approximation of the Gradient

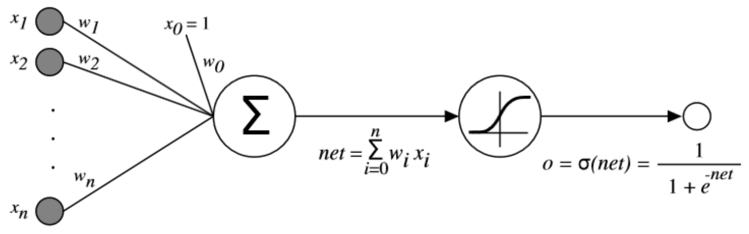
Instead of computing the gradient over the entire dataset we can approximate it using a minibatch of data or even a single randomly selected sample. This method is called the stochastic gradient descent. The pseudo code is shown below:-

```
1. Initialize **w** to small random values
2. Repeat until satisfied
    a. Sample a minibatch of K datapoints $K < N$; call it $\widetilde{D}$
    b. $\nabla_{\mathbf{w}}(E_{\widetilde{D}}) = \frac{1}{K}\sum_{d=1}^{K}(\boldsymbol{y_d} - \widehat{\boldsymbol{y_d}})(-\boldsymbol{x_d})$
    c. $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta\nabla_{\boldsymbol{w}}(E_{\widetilde{D}})$
```

It can be shown that the stochastic approximation can be arbitrarily close to the true gradient for small enough η. The stochastic gradient descent approximation can be applied to non-linear problems as well where it has the added benefit of helping avoid some local minima due to its stochastic nature.

Sigmoid Unit

A real world problem will often involve non-linear decision boundaries. The perceptron model cannot provide good accuracies for such problems. However, if we stack together multiple layers of several perceptrons then a very powerful class of models is obtained commonly referred to as 'multi-layer feedforward neural networks'. Unfortunately, the threshold non-linearity in each layer makes this non differentiable. Thus we cannot use gradient descent to train it; there is no gradient in the first place. We therefore replace the threshold non-linearity with a sigmoid non-linearity. The resulting model is shown below:-



x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

[Image source: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-20/www/mlbook/ch4.pdf slide 16]

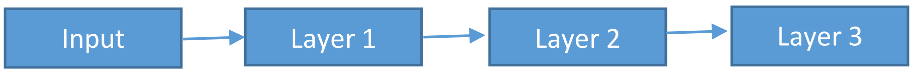
The sigmoid non-linearity is differentiable. Using the notation as in the figure above:-

$$\frac{d\sigma}{d(net)} = o\,(1-o)$$

Where we have rewritten the derivative in terms of the network output. But many questions remain.

1. How do we connect these sigmoid neurons?

   Ans. In general we can connect neurons arbitrarily but in this lesson we only focus on a special case called the linear chain; shown below:-



2. How do we train such a network with many sigmoid units?

   Ans. We train such a network using the backpropagation algorithm. The algorithm easily generalizes to arbitrary directed acyclic graphs but for ease of presentation we shall describe it for the linear chain model.

Note that cyclic connections are also used in practice. These are referred to as recurrent neural networks and they are usually trained using backpropagation through time. The generalization to acyclic graphs is more trivial than the generalization to a graph with cycles.

Backpropagation algorithm

We shall discuss the backpropagation algorithm by example. Consider a network that performs the following mathematical operations

$$I \to y_1 = w_1 I \to y_2 = \sigma(y_1) \to y_3 = w_3 y_2 \to y_4 = \sigma(y_3) \to E = \frac{1}{2}(y - y_4)^2$$

This network receives a scalar input $I \in \mathbb{R}$, which passes through 2 sigmoid units with weights w_1 and w_3 respectively. It produces a scalar output E by comparing the prediction $y_4$with y. For simplicity we shall decompose this into five layers – layer 1 performs the multiplication with $w_1$to transform I into $y_1$and layer 2 performs the non-linearity transforming y_1 into y_2. Layer 3 performs the multiplication with w_2 and layer 4 performs the sigmoid non-linearity transforming $y_3$into y_4. Lastly, layer 5 converts y_4 into the scalar loss E.

All of the variables are real valued scalars. We shall generalize this to vectors after deriving the scalar version here.

The backpropagation algorithm is the same as gradient descent. Gradient descent for the network above is written below:-

```
For $n = 1\ to$ numIterations
    a. $\Delta w_1 = 0, \Delta w_2 = 0$
    b. for d = 1 to numData
        i. $\Delta w_1 = \Delta w_1 - \frac{\partial E_d}{\partial w_1}$
        ii. $\Delta w_2 = \Delta w_2 - \frac{\partial E_d}{\partial w_2}$
    c. $w_1 = w_1 + \eta_n \Delta w_1$
    d. $w_2 = w_2 + \eta_n \Delta w_2$
```

The above algorithm computes the gradient by iterating over each data point. The subscript d is used to denote the variable value corresponding to a single data point $E_d, I_d, y_{1d}, y_{2d}, y_{3d}, y_{4d}$.

The backpropagation algorithm contributes an efficient way of computing the partial derivatives $\frac{\partial E_d}{\partial w_1}$ and $\frac{\partial E_d}{\partial w_2}$. This contribution is simply the chain rule. Backpropagation works by each layer computing two things:-

a) Derivative of the loss with respect to its input.
b) Derivative of the loss with respect to its parameters.

This is done for all the layers starting from layer 5 down to layer 1.

Layer 5:

1. $\frac{\partial E_d}{\partial y_{4d}} = y_{4d} - y_d$
2. No parameters for layer 5.

Each layer passes the derivative of the loss with respect to its input, to the layer below it. The layer below treats this as “top derivative” or backpropagated error δ_d.

So Layer 4:

1. $\frac{\partial E_d}{\partial y_{3d}} = \frac{\partial E_d}{\partial y_{4d}} \times \frac{\partial y_{4d}}{\partial y_{3d}} = \delta_d \times \frac{\partial y_{4d}}{\partial y_{3d}}$ where the chain rule has been used to simplify the problem. Note that layer 4 now only needs to compute the derivative of its output with respect to its input and doesn’t bother about the layers above it. Thus,
$$\frac{\partial E_d}{\partial y_{3d}} = \delta_d y_{4d}(1 - y_{4d})$$
2. No parameters for layer 4.

Again layer 4 will pass $\frac{\partial E_d}{\partial y_{3d}}$ to layer 3 in the form of δ_d (The backpropagated error).

Layer 3:

1. $\frac{\partial E_d}{\partial y_{2d}} = \delta_d \times \frac{\partial y_{3d}}{\partial y_{2d}} = \delta_d \times w_3$
2. $\frac{\partial E_d}{\partial w_3} = \delta_d \times y_{2d}$ where the chair rule was again used to make the problem local and independent of the operations performed at deeper layers.

Layer 2:

1. $\frac{\partial E_d}{\partial y_{1d}} = \delta_d \times y_{2d}(1 - y_{2d})$
2. No parameters at layer 2.

Layer 1:

1. $\frac{\partial E_d}{\partial I_d} = \delta_d w_1$
2. $\frac{\partial E_d}{\partial w_1} = \delta_d I_d$

Done!

Thus the backpropagation algorithm allows for the efficient implementation of gradient descent over a multilayer feedforward neural network by cleverly applying the chair rule.

Backpropagation Algorithm for Vector Input and Vector Output

The scalar example illustrated above is a degenerate case. The generalization to vectors just involves more linear algebra.

$$I \in \mathbb{R}^{N}, W_1 \in \mathbb{R}^{N_1 \times N}, y_{1d} \in \mathbb{R}^{N_1}, y_{2d} \in \mathbb{R}^{N_1}, W_3 \in \mathbb{R}^{N_3 \times N_1}, y_{3d} \in \mathbb{R}^{N_3}, y_{4d} \in \mathbb{R}^{N_3}$$

All the partial derivatives are generalized to gradients and all scalar multiplications are generalized to matrix vector operations.

This section on vector input and vector output for backpropagation has been omitted from the writeup.

Problems

1. Vanishing gradients – If we stack up too many sigmoid units then the gradient magnitude decays as it travels backwards. This makes gradient updates in the lower layers extremely small and learning impractically slow and ineffective. This is addressed by pre-training the network using unsupervised learning or by using ReLU activation units instead of sigmoid. More on this in the section on deep learning.
2. Local Optimum – The loss function with two or more layers is highly non-linear with lots of local optimum. There is no guarantee that backpropagation will converge to the global optimum solution. This is helped partly by using stochastic gradient descent. Additional tricks include initialization with small weights so that we are near the linear region of the sigmoid where the problem is less non-linear. Alternatively, we can use a momentum term to help skip small local optima in the optimization landscape.
$$\Delta w(n) = \frac{\eta \partial E_d}{\partial w} + \alpha \Delta w(n-1)$$
$\alpha \in (0,1)$ is called the momentum term.
3. Overfitting – These multilayer feedforward neural networks are very expressive and thus they can fit those idiosyncrasies of training data that do not reflect the general behavior of data at test time. This can a problem in any setting with a reasonable amount of noise. It is called overfitting to training data. Techniques such as early stopping with cross validation, weight decay are used to prevent overfitting.
Weight decay reduces all the weights in each iteration by a small fraction of their current value.
$w \leftarrow w - \zeta w$ Where $\zeta$is very small. Smaller weights push the network closer to the linear region of the sigmoid making it less likely for the network to be able to fit unreasonable idiosyncrasies in the training data.
4. Lots of Parameter Tuning – There are lots of design choices involved in solving a learning problem using a neural network. How many neurons should we use in each layer? How to set the learning rate? The momentum term? Often the learning rate needs to change after a few iterations and this is also manually specified by the programmer.
5. Lack of Interpretability – The neurons other than the input and output neurons are called hidden layer neurons. The output of hidden layer neurons is often hard to interpret making it difficult to justify our neural network solution to scientists and people in other domains, where interpretability is very important.

Demo on Face Pose Estimation

A simple example has been illustrated by Tom Mitchell in his book “Machine Learning”. He trains a 2 layer network of sigmoid units (similar to the example above with $N_1 = 3, N_3 = 4$ to classify the pose of a face shown in an image. The image is 30 x 32 pixels grayscale and is made into a vector and fed into the network. The network should output whether the face is looking left, right, up or straight. Several design choices are made when designing the network; other than the network architecture itself. These are:-

1. Input representation
   The full resolution images are downsampled to 30 x 32 pixels by local averaging pixel values. Pixel values normally range from 0 to 255. They are linearly scaled to lie between 0 to 1.
2. Output representation
   Four output neurons are used for a one hot encoding of the four possible results – 1000 for left, 0100 for right, 0010 for up and 0001 for straight. Further, instead of using 1 or 0 he uses 0.9 and 0.1. This is because sigmoid units cannot actually produce 0 and 1 with finite weights while they can produce 0.9 and 0.1 with finite weights. This prevents the weight values from exploding out of control.
3. Other learning parameters
   Learning rate is set to 0.3, momentum to 0.3 and a single example stochastic approximation of the gradient is used.

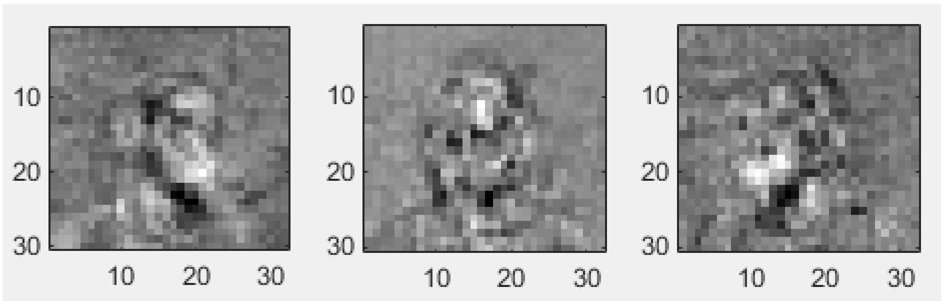
Demo Code

I've reimplemented some of this experiment in matlab using the matconvnet toolbox. The network was trained without cross validation and the initialization was different but the rest of the experiment was followed closely. Momentum was not used.

The network converges to the following weights in layer 2

```
 4.6527    0.0500   -4.8483    0.1235
-0.0712   -0.6170   -4.4124    5.1136
 0.0710    4.6472   -4.8370    0.1013
```

And the following weights in layer 1



Which look like right, straight and up left respectively explaining the weight matrix above. At test time this model got 124 out of 139 correct – 89% accuracy (I used http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-8/faceimages/trainset/all_test1.list for the test set and http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-8/faceimages/trainset/all_train.list for the train set).

Let's have a quick look at the code to see how backpropagation is used in practice.

```
run C:\Users\aravindh\Software\matconvnet\matlab\vl_setupnn.m

load('facedata.mat'); % loads X for the face images, Y for their ground
truth labels
% X is 30 x 32 x 1 x 277 dimensions and
```

```matlab
% Y is 1 x 1 x 4 x 277 dimensions
% X_test is 30 x 32 x 1 x 139
% Y_test is 139 x 1 dimensions containing integers between 1,2,3,4

% hyperparameters for the training
lr = 0.03;
numIterations = 1000;
numHiddenUnits = 3;

% Initialize the parameters for the network
f1 = 0.05 * randn(1, 1, 30*32, numHiddenUnits, 'single');
b1 = zeros(numHiddenUnits, 1, 'single');
f3 = 0.05 * randn(1, 1, numHiddenUnits, size(Y, 3), 'single');
b3 = zeros(size(Y,3), 1, 'single');

loss = zeros(numIterations*size(X,4), 1);
for i=1:numIterations
    for j=1:size(X, 4)
        x_cur = reshape(X(:,:,1,j), [1, 1, 30*32, 1]);
        y_cur = Y(1,1,:,j);

        % forward pass
        h1 = vl_nnconv(x_cur, f1, b1);
        h2 = vl_nnsigmoid(h1);
        h3 = vl_nnconv(h2, f3, b3);
        h4 = vl_nnsigmoid(h3);
        [loss_cur, dzdh4] = vl_nnsqeuclidean(h4, y_cur, [], single(1));
        loss((i-1)*size(X,4) + j) = loss_cur;

        % backward pass
        dzdh3 = vl_nnsigmoid(h3, dzdh4);
        [dzdh2, dzdf3, dzdb3] = vl_nnconv(h2, f3, b3, dzdh3);
        dzdh1 = vl_nnsigmoid(h1, dzdh2);
        [dzdx_cur, dzdf1, dzdb1] = vl_nnconv(x_cur, f1, b1, dzdh1);

        % Update the parameters
        f1 = f1 - lr * dzdf1; b1  = b1 - lr * dzdb1;
        f3 = f3 - lr * dzdf3; b3 = b3 - lr * dzdb3;
    end
    if(mod(i, 10) == 1)
        fprintf(1, 'Iteration %d\n', i);
        figure(1);
        plot(loss(1:(i-1)*size(X,4) + j));
        drawnow;
        pause(0.5);
    end
end

% Visualize the learnt network
figure;
subplot(1,3,1); imagesc(reshape(f1(:,:,:,1), [30, 32, 1])); axis image;
colormap gray;
subplot(1,3,2); imagesc(reshape(f1(:,:,:,2), [30, 32, 1])); axis image;
colormap gray;
subplot(1,3,3); imagesc(reshape(f1(:,:,:,3), [30, 32, 1])); axis image;
colormap gray;

squeeze(f3)
```

```
% Let's test it on the test set.
numTestData = size(X_test, 4);
X_test = reshape(X_test, [1, 1, 30*32, numTestData]);
h1 = vl_nnconv(X_test, f1, b1);
h2 = vl_nnsigmoid(h1);
h3 = vl_nnconv(h2, f3, b3);
h4 = vl_nnsigmoid(h3);
correct = 0;
for i=1:numTestData
    [~, result_cur] = max(squeeze(h4(1,1,:,i)));
    if(result_cur == Y_test(i))
        correct = correct + 1;
    end
end
accuracy = correct / numTestData;
```

Expressive Power

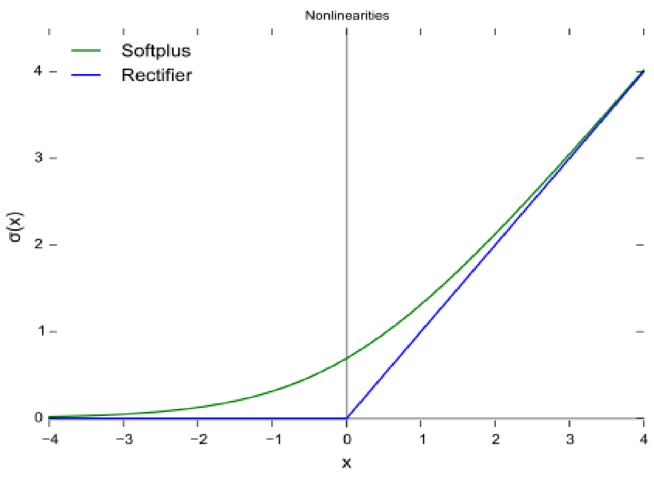
1. Two layers of sigmoid units can express any Boolean function. But the hidden layer may need to be very fat.
2. Any bounded continuous function can be approximated arbitrarily well by a two layer network with sigmoid units in the hidden layer and (unthresholded) linear units in the output layer. (Cybenko 1989, Hornik et. al. 1989)
3. Any function can be approximated to arbitrary accuracy with a network of three layers, where the output layer again has linear units. (Cybenko 1988).

Thus a multilayer feedforward neural network is very expressive which makes them prone to overfitting. But in the worst case, we need to use a large number of hidden layer neurons – exponential in the input dimension. This is computationally infeasible for real world problems in computer vision and speech recognition where the data is in several thousand if not millions of dimensions. This problem is addressed by going deeper with several hidden layers as discussed in the next section.

Deep Learning

An artificial neural network with many hidden layers is called a deep neural network. Deep neural networks can express very complicated functions but without many hidden layer neurons. Despite this knowledge they were not very popular until recently. This is because training such a deep network is very difficult. The gradients at the lower layers are very small because of the non-linear nature of sigmoid units at each layer. This is called the problem of “vanishing gradients”. This is addressed by replacing the sigmoid with a rectified linear unit (ReLU) or by pre-training the network using unsupervised learning. We shall discuss only the ReLU units in this lesson.

"Rectifier and softplus functions". Licensed under CC0 via Wikipedia - https://en.wikipedia.org/wiki/File:Rectifier_and_softplus_functions.svg#/media/File:Rectifier_and_softplus_functions.svg



The ReLU has a non-linearity at 0. It is linear for any input greater than 0. Thus the gradients do not get squashed and remain significant very deep down the network. But there is no gradient at 0 itself. We use an element in the subgradient instead of the gradient and use sub-gradient descent instead of gradient descent.

Another important idea is to use convolution to scale up to large images. This idea has its roots in the neo-cognitron model by Fukushima (1980). It was later developed into the convolutional neural network (ConvNet) by Lecun et. al. (1988). The key idea is that instead of multiplying each pixel by a weight and thus using a million weights for 1 megapixel image, we convolve a small kernel of weights over the image. This assumes that image statistics are translation invariant and the same weights can be applied everywhere in the image. How does this help?

1. The number of weights is now much less than 1 million for a 1 mega pixel image.
2. The small number of weights can use different parts of the image as training data. Thus we have several orders of magnitude more data to train the fewer number of weights.
3. We get translation invariance for free.
4. Fewer parameters take less memory and thus all the computations can be carried out in memory in a GPU or across multiple processors.

The last of these points may not sound like a big deal but for modern deep convolutional neural networks with upto 16 layers this can make the difference between several months of training and several days of training. GPU code is typically 10 times faster for these models than their CPU counterparts.

LeNet

A modern neural network is nothing like its biological counterpart. We can put any sequence of differentiable / sub-gradientable operations and use backpropagation to train the parameters. A modern deep neural network used to solve digit recognition is composed of convolution, ReLU and max pooling and softmax layers. Of these you are already familiar with Convolution and ReLU. Max Pooling layers take the local maximum across a small spatial neighborhood. This downsamples the input and makes it invariant to local spatial deformations. Softmax is used to convert neuron scores into a vector of probabilities. This is typically used at the top of the network to measure the log likelihood of the ground truth class. This implements maximum liklihood learning.

A network used for digit recognition is shown below:-

Convolution -> Max Pooling -> Convolution -> Max Pooling -> Convolution -> ReLU -> Convolution -> Softmax -> Log likelihood loss.

This can be trained using the MatConvNet toolbox developed by our group. The steps are as follows:-

1. Create a cell array of structure objects. Each structure object tells the layer name, layer type(‘conv’, ‘pool’, ‘relu’, ‘softmaxloss’) along with layer parameters.
2. Create an imdb object from the data.
3. Call cnn_train in the matconvnet examples folder with the net, imdb, function object to retrieve mini batches and optional training options.

MatConvNet will return the trained network model in the same cell array format but with trained values for the learnable parameters (weights). The cnn_train function is implemented using the same backpropagation algorithm. As an example, I shall show how backpropagation works for the ReLU layer. Recall that the layer needs to compute the derivate of the loss w.r.t. its parameters and w.r.t. its inputs. This, in principle, is done by computing the derivative of the ReLU output w.r.t. its inputs (note that there are no parameters in the ReLU layer). In practice, however, just like in the sigmoid learning units we don't need to explicitly compute the derivative of the ReLU output w.r.t. the input. We can directly compute the hadamard product of the "backpropagated error" w.r.t. the element wise derivative of the ReLU. Let's have a quick look at the code in the file matconvnet/matlab/vl_nnrelu.m

There are two modes of operation – forward mode vs backward mode. In the forward mode, only 1 input is available – y = vl_nnrelu(x). The code evaluates y = max(x, 0); to evaluate the ReLU forward.

In the backward mode two inputs are available dzdx = vl_nnrelu(x, dzdy); where dzdx is the derivative of the loss z w.r.t the input x and dzdy is the "backpropagated error" (the derivative of the loss z w.r.t. the output y). dzdx is computed using the hadamard product dzdy .* (x > 0); where (x > 0) is the element wise derivative of the ReLU.

The vl_nnrelu.m function implements additional functionality to support a leak. This is not required at the moment.

Conclusion

Thanks for your time. Feel free to email me with any questions at aravindh.mahendran@new.ox.ac.uk

References

Cybenko 1989 - https://www.dartmouth.edu/~gvc/Cybenko_MCSS.pdf

Cybenko 1988 – Continuous Valued Neural Networks with two Hidden Layers are Sufficient (Technical Repor), Department of Computer Science, Tufts University, Medford, MA

Fukushima 1980 - http://www.cs.princeton.edu/courses/archive/spr08/cos598B/Readings/Fukushima1980.pdf

Hinton 2006 - http://www.cs.toronto.edu/~fritz/absps/ncfast.pdf

Hornick et. al. 1989 - http://www.sciencedirect.com/science/article/pii/0893608089900208

Krizhevsky et. al. 2012 - http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

Lecun 1998 - http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf

Tom Mitchell, Machine Learning, 1997