Fully-connected neural networks

Adam Boult (www.bou.lt)

June 30, 2025

Contents

Preface

This is a live document, and is full of gaps, mistakes, typos etc.

Part I

Feedforward neural networks

Chapter 1

Multi-layer perceptrons

1.1 Forward pass through a Multi-Layer Perceptron (MLP)

1.1.1 Recap: Perceptron

In the perceptron we have input vector $\mathbf{x}$, and output:

$a = f(\mathbf{wx}) = f(\sum_i^n w_i x_i)$

1.1.2 Adding additional layers to the perceptron

We can augment the perceptron by adding a hidden layer.

Now the output on the activation function is an input to a second layer. By using different weights, we can create a second vector of inputs to the second layer.

Θ^j is a matrix of weights for mapping layer j to $j+1$.

In a 2-layer perceptron we have Θ^0 and Θ^1.

If we have s units in the hidden layer, n features and k classes:

- The dimension of Θ^0 is $(n+1) \times s$
- The dimension of Θ^1 is $(s+1) \times k$

These include the offsets for each layer.

1.1.3 The activation function of a multi-layer perceptron

For a perceptron we had:

$a = f(\mathbf{wx}) = f(\sum_i^n w_i x_i)$.

Now we have:

$a_i^1 = f(\boldsymbol{x\Theta^0}) = f(\sum_i^n x_i \Theta_i^0)$

$a_i^2 = f(\boldsymbol{a^1\Theta^1}) = f(\sum_i^n a_i^1 \Theta_i^1)$

For additional layers this is:

$a_i^j = f(\boldsymbol{a^{j-1}\Theta^{j-1}}) = f(\sum_i^s a_i^{j-1} \Theta_i^{j-1})$

We refer to the value of a node as a_i^j, the activation of unit i in layer j.

1.1.4 Dummies in neural networks

1.2 Regressing on unbounded outputs with neural networks

1.2.1 Introduction

Can not have a sigmoid function at the last step.

Alternatively can apply a sigmoid function to the unbounded output to make it bounded.

1.3 Deep neural networks can represent complicated functions

1.3.1 More layers allow for more complex function

With additional hidden layers we can map more complex functions.

These allow the effective combination of logic gates.

1.3.2 2 hidden layers can map highly complex functions

With only two hidden layers we can map any function for classification, including discontinuous functions.

1.3.3 Increasing numbers of dimensions in a unit

Topology of layers. Increasing number of units in subsequent layers is like increasing dimension.

We are trying to make data linearly separable. it may be that we need additional dimensions to do this, rather than a series of transformations within the existing number of dimensions.

eg for a circle of data within a circle of data, there is no linear separable line, so no depth without increasing dimensions will split data.

Chapter 2

Multi-class neural networks

2.1 More than 2 classes

2.1.1 Local Winner Takes All (LWTA) layer

The output for all nodes in a layer is 0 unless it is the greatest.

2.1.2 Maxout layer

Single node, spits out the max of all inputs.

Part II

Training feedforward neural networks

Chapter 3

Backpropagation

3.1 Back propagation

3.1.1 Recap on the delta rule

To arrive at the delta rule we considered the cost function:

$E = \sum_j \frac{1}{2}(y_j - a_j)^2$

This gave us:

$\frac{\delta E}{\delta \theta^i} = -\sum_j (y_j - f(\sum_i \theta^i x_j^i)) f'(\sum_i \theta^i x_j^i) x_j^i$

By defining $z_j = \sum_i \theta^i x_j^i$ and $a = f$ we have:

$\frac{\delta E}{\delta \theta^i} = -\sum_j (y_j - a(z_j)) a'(z_j) x_j^i$

$\delta_i = -\frac{\delta E}{\delta z_j} = \sum_j (y_j - a_j) a'(z_j)$

So:

$\frac{\delta E}{\delta \theta_i} = \delta_i x_{ij}$

$\frac{\delta E}{\delta \theta^i} = -\sum_j (y_j - a(z_j)) f'(z_j) x_j^i$

We define delta as:

$\delta_i = -\frac{\delta E}{\delta z_j} = \sum_j (y_j - a_j) a'(z_j)$

So:

$\frac{\delta E}{\delta \theta_i} = \delta_i x_{ij}$

$\Delta\theta_i = \alpha \sum_j (y_j - a_j) a'(z_j) x_{ij}$

We update the parameters using gradient descent:

$\Delta\theta_i = \alpha \delta_i x_{ij}$

Or, setting $\delta_i = -\frac{\delta E}{\delta z_j} = \sum_j (y_j - a_j) a'(z_j)$

$\Delta\theta_i = \alpha \delta_j x_{ij}$

3.1.2 Adapting the delta rule for multiple layers

Let's update the rule for multiple layers:

And used the chain rule:

$\frac{\delta E}{\delta \theta^i} = \frac{\delta E}{\delta a_j} \frac{\delta a_j}{\delta z_j} \frac{\delta z_j}{\delta \theta^i}$

Where $a_j = f(z_j)$ and $z_j = \boldsymbol{\theta x_j}$

$\frac{\delta E}{\delta \theta_{li}} = \frac{\delta E}{\delta a_{lj}} \frac{\delta a_{lj}}{\delta z_{lj}} \frac{\delta z_{lj}}{\delta \theta_{li}}$

Previously $\frac{\delta z_{lj}}{\delta \theta_{li}} = x_i$. We now use the more general a_{li}. For the first layer, these will be the same.

We can then instead write:

$\Delta\theta_i = \alpha \delta_{lj} a_{li}$

Now we need a way of calculating the value of δ_{lj} for all neurons.

$\delta_i = -\frac{\delta E}{\delta z_{lj}}$

If this is an output node, then this is simply $\sum_j (y_j - a_j) a'(z_j)$

If this is not an output node, then the impact of change in the parameter will affect the results through all intermediate neurons.

In this case:

$\frac{\delta E}{\delta z_{lj}} = \sum_{k \in succl} \frac{\delta E}{\delta z_k} \frac{\delta z_k}{\delta z_{lj}}$

$\frac{\delta E}{\delta z_{lj}} = \sum_{k \in succl} -\delta_k \frac{\delta z_k}{\delta a_{kj}} \frac{\delta a_{kj}}{\delta z_{lj}}$

$\frac{\delta E}{\delta z_{lj}} = \sum_{k \in succl} -\delta_k \theta_{kj} a'_{kj}$

$\delta_i = a'_{kj} \sum_{k \in succl} \delta_k \theta_{kj}$

For each layer there is a matrix, where the columns and rows represent the *theta* between the current layer and the next layer. We have a matrix for each layer in the network.

3.1.3 Initialising parameters

We start by randomly initialisng the value of each θ.

We do this to prevent each neuron from moving in sync.

Chapter 4

Alternatives to backpropagation

4.1 Alternatives to backpropagation

4.1.1 Greedy pretraining

4.1.2 Cascade-correlation learning architecture

This is a method for both building and training.

We start with a bare bones network. We then add nodes one by one, training and then fixing their values.

4.1.3 Extreme learning machines

This is an alternative to backprobagation for training a feedforward neural network.

We start with random parameters for each layer W_i.

We have:

$\hat{y} = W_2\sigma(W_1x)$

Etc.

We calculate:

$W_2 = \sigma(W_1x)^+Y$

So W_1 is random and not updated.

W_2 is assigned to minimise loss, where W_2 has no activation function.

Chapter 5

Regularising neural networks

5.1 Regularising neural networks

5.1.1 Feature normalisation

5.1.2 Dropout, and dropout layers

5.1.3 L_2 regularisation (including how to change backprob algorithm)

5.1.4 Sparse networks

Parameters are set to 0 and not trained.

5.1.5 Parameter sharing

Parameters share the same value and are trained together.

5.1.6 Weight decay

After each update, multiply the parameter by $p < 1$.

5.1.7 The anomoly detection problem

Can change input to get any classification.

5.1.8 Early stopping

5.1.9 Residual blocks

In a node we have:

$a_{ij} = \sigma_{ij}(W_{ij}a_{i-1})$

That is, the value of a node, is the activation on the sum of the weights of the previous layer.

Residual block however look further back that one layer. They include the full data from an older layer (without weights)

$a_{ij} = \sigma_{ij}(W_{ij}a_{i-1} + a_k)$

Chapter 6

Normalising neural networks

6.1 Optimisation

6.1.1 Input layer normalisation

We normalise the input layer.

This speeds up training, and makes regularisation saner.

6.1.2 Batch normalisation

We can normalise other layers. We take each input, subtract the batch mean divided by the batch standard deviation.

Batch normalisation and covariance shift

Batch normalisation can make networks better adapted for related problems.

Training with batch normalisation

Chapter 7

Addressing gradient problems with the Rectified Linear Unit (ReLU)

7.1 Link/activation functions: Regression

7.1.1 Absolute value rectification

$a(x) = |x|$

7.1.2 Rectified Linear Unit (ReLU)

The function

$a(z) = \max(0, z)$

The derivative

Its differential is 1 for values of z above 0, and 0 for values of z below 0.

The differential is undefined at $z = 0$, however this is unlikely to occur in practice.

Notes

The ReLU activation function induces sparcity.

7.1.3 Noisy ReLU

7.1.4 Leaky ReLU

7.1.5 Parametric ReLU

7.1.6 Softplus

The function

$a(z) = \ln(1 + e^z)$

The derivative

Its derivative is the sigmoid function:

$a'(z) = \frac{1}{1 + e^{-z}}$

Notes

The softplus function is a smooth approximation of the ReLU function.

Unlike the ReLU function, Softplus does not induce sparcity.

7.1.7 Exponential Linear Unit (ELU)

7.2 Link/activation functions: Regression

7.2.1 Convexity

The error function for neural networks is nearly convex.

7.2.2 Unstable gradient problem

Vanishing gradient problem

Gradients can be become small, and so propagation can be very slow.

Exploding gradient problem

Gradients can become too large and not converge.

ReLU

This addresses the unstable gradient problem.

7.2.3 Curse of dimensionality

As parameter space gets bigger, data requirements get bigger.

Sparse models can help. Eg ReLU.

7.2.4 Representational sparsity

This is where the values in nodes are often 0, as opposed to just the parameters.

Chapter 8

Pre-training neural networks

8.1 Pre-training

8.1.1 Pre-training

Pre-training. eg train on general pictures before specific stuff. means you fit many parameters for detecting edges etc firsts

Learning rate

Reduce the learning rate.

8.1.2 Resetting parameters

Replace last layer (softmax) for new problem.

8.1.3 Freeze layers

Freeze feature learning of early layers.

8.1.4 Catastrophic interference

When retraining model on new data, model may forget answers to old data.

Part III

Other types of feedforward neural network

Chapter 9

Probabilistic neural networks

9.1 Probabilistic neural networks

9.1.1 Introduction

9.1.2 Input layer

The input is the feature vector.

9.1.3 Pattern layer

The first layer has a node for each observation in the training set.

In each node, the value is the distance from the input to the comparator.

This can be calculated using Gaussian distribution, or another method.

9.1.4 Summation layer

One neuron for each category.

We map from the pattern layer to the summation layer according to the actual label of each training item.

Ie, if a sample is red, it will be fed only to the red neuron.

The values are summed.

Largest value is selected.

9.1.5 Architectures for images

Chapter 10

Convolutional layers for neural networks

10.1 Convolutional layers

10.1.1 Convolutional layers

Can connect each node in first hidden layer to a subset of the input layer, eg one node for each 5x5 pixels

We also share weights for each of the first layer. Much fewer parameters, and can learn all good stuff

This also uses windows. Instead of max we multiply the window by a matrix elementwise and sum the values

Each matrix can represent some feature, like a curve.

We can use multiply convolution matrices to create multiple output matrices.

Matrices are called kernels. they are trained. start off random

Training convolutional layers

10.1.2 Invariance of convolutional layers (rotation, translation)

10.1.3 Flattening layers

We split the data up everytime we use convolional layers

Flattening layers bring them all back together

Parameters are that for pooling layers (height, width, stride, padding, but also set of convolutions.

10.1.4 Multi-scale convolutions

We use different window sizes in parallel.

10.1.5 Inception modules

10.2 Pooling (max pooling, average pooling, subsampling)

10.2.1 Pooling layers

The input is a matrix. We place a number of windows on the input matrix. The max of each window is an input to the next layer.

Means fewer parameters, easier to compute, less chance of overfitting

Parameters: height, width of window, stride (amount shifts by each window)

We can also add padding to the edge of the image so we don't lose data.

Same padding (use 0), valid padding (no padding)

Pooling layer compresses, takes 2x2. Max pooling returns highest activiation

10.2.2 Vector of Locally Aggregated Descriptors (VLAD)

10.3 Other window layers

10.3.1 Capsules

Primary capsule layers

Outputs of convolutions are scalars. however we can also create vectors, if we associate some convolutions with each other

eg if we have 6 convolutions, the output of these can be used to create a 6 dimensional vector for each window.

Normalisation in primary capsule layers (vector squishing)

We can normalise the length of these vectors to between 0 and 1.

The output of this repesents the chance of finding the feature they are looking for, and the orientation

If the vector length is low, feature not found. if high, feature found.

We have orientation from vector, and position from window

Routing capsule layers

We now have a layer of position and orientation of basic shapes (triangles, rectangles etc)

We want to know which more complex thing they are part of.

So the output of this step is again a matrix with position and orientation, but of more complex features

To determine the activation from each basic shape to the next feature we use routing-by-agreement.

This takes each basic shape and works out what it would look like if the complex feature was present.

If a complex feature has two basic shapes, they will both have the same predicted complex shape. Otherwise the relationship is spurious and they will not

If they agree we have a high weight

This process is complex and computationally expensive.

However we don't need pooling layers now

caps net

Does normal conv first, then primary, then secondary.

caps: reconstruction

We have vector space of feature position and orietnation. we can recreate output

Part IV

Generative neural networks

Chapter 11

Autoencoders and Variational Autoencoders (VAE)

11.1 Autoencoders

11.1.1 Autoencoders

Autoencoders are a type of neural network.

For autoencoders the goal for the output layer is the reconstructed input layer, rather than a classification.

By including sparsity in the neural network we can reduce the dimensions. This splits the network into an encoder and a decoder.

Middle vector is called latent variables.

11.2 Variational autoencoders

11.2.1 Variational Autoencoders (VAE)

Like AE, but force middle vector to have unit gaussian by adding new loss function

Now we can generate new images by sampling for latent normal of unit 1.

Chapter 12

Restricted Boltzmann Machines (RBMs)

12.1 Restricted Boltzmann Machines (RBMs)

12.1.1 Restricted Boltzmann Machines (RMBs)

12.1.2 Deep Belief Networks (DBNs)

12.1.3 Training with Contrastive Divergence (CD)

Chapter 13

Self-organising maps

13.1 Self-organising maps

Chapter 14

Generative Adversarial Networks (GANs)

14.1 Generative adversarial networks

14.1.1 Generative Adversarial Networks (GANs)

Introduction

The GAN generator

Generator: take latent multivar normal as input layer

Output of generator is of same dimension as input iamges

Generative NN: need probability distribution on input layer

Generator typically deconvolutional

Downside of gan. only discriminates between real and fake.

The GAN discriminator

Discriminator: use normal CNN for deep