Sebastian Borgeaud dit Avocat

Brain tumour segmentation using Convolutional Neural Networks

Computer Science Tripos – Part II

Fitzwilliam College

March 20, 2017

Proforma

Name:	**Sebastian Borgeaud dit Avocat**
College:	**Fitzwilliam College**
Project Title:	**Brain tumour segmentation using Convolutional Neura**
Examination:	**Computer Science Tripos – Part II, June 2017**
Word Count:	**INSERT**
Project Originator:	Duo Wang
Supervisor:	Dr. Mateja Jamnik & Duo Wang

[1]This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I, Sebastian Borgeaud dit Avocat of Fitzwilliam College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

List of Figures

Acknowledgements

This document owes much to an earlier version written by Simon Moore [?]. His help, encouragement and advice was greatly appreciated.

Chapter 1

Introduction [14%]

1.1 Motivation

- Motivation for choosing this problem: Openly available data, important problem (include count of people affected by brain tumours)
- Then motivate the choice of conv nets to tackle this problem.
- Also mention that this is an opportunity for me to explore the field of ML further and gain some practical experience working in that field.

1.2 Related Work

- Here I will introduce the previous work done. In particular, this should contain a short intro to the history of neural nets and conv nets. Then a brief history of the brain tumour segmentation problem.
- Next mention the main paper [1] and the BraTS challenge/conference.
- Mention more recent developments, such as the usage of ResNets.
- This paragraph should introduce the reader to the problem and to what has been done previously.

Chapter 2

Preparation [26%]

During the first phase of my project, the aim was to replicate the method used by Pereira et al [1], so it was crucial to first fully understand the steps taken in the paper to then be able to reimplement them. Unfortunately, the paper didn't include any source code which meant that if something wasn't fully explained in details, I would have to find out what was actually done. This turned out to be a problem for the pre-processing step as the proposed method uses a normalisation developed by Nyul [CITATION]. This normalisation requires human input, preferably from a domain expert, and I was therefore not able to use that normalisation method. The paper also proposed a second normalisation method, which used a combination of winsorizing and N4 normalization. This method performed slightly worse but had the advantage of being fully automated, which is why I chose to use this method.

2.1 Starting point

2.2 Theoretical background

2.2.1 Artificial Neural networks

To understand how convolutional neural networks work, it is important to be familiar with ordinary neural networks. These are made up from a sequence of layers of neurons, each neuron having a set of trainable weights that can be adjusted to change the overall function computed by the neural network. An example of the structure of such a neural network can be found in figure 2.1.

Each neuron in layer $n + 1$ is connected to every neuron in layer n and computes as an output

$$y = f_{act}((\sum_{i=1}^{n} y_i w_i) + b)$$

where f_{act} is a non-linear, differentiable activation function and y_i is the output of neuron i in the previous layer. A neuron is connected to every neuron in the previous layer, which is why this layer is also referred to as a fully connected layer.

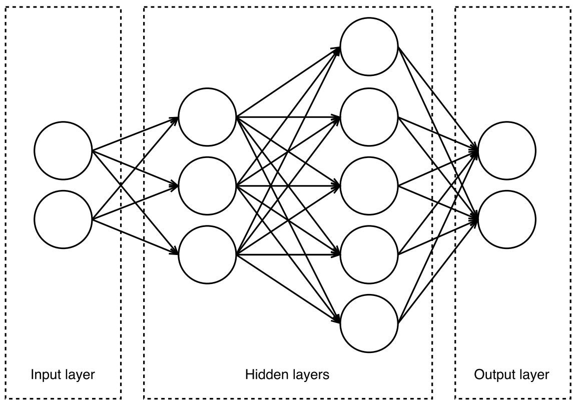

Figure 2.1: Structure of a simple neural network with two hidden layers.

Activation functions

The most common activation functions are the Sigmoid function,

$$S(x) = \frac{1}{1 + e^x}$$

the hyperbolic tangent

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

the rectifier

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

and the leaky rectifier, for some $0 < \alpha < 1$

$$f(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

This functions can be seen plotted in figure 2.2. Historically, the hyperbolic tangent function or the sigmoid function have been used as activation functions. However, as the magnitude of the gradients of those functions is always below 1, these activation function create a problem of vanishing gradients for deeper networks, as we have to multiply those gradients together for each layer. [CITATIONS!!!!] This is why it is nowadays preferred to use the rectifier or the leaky rectifier functions, especially with deeper architectures.

In a classification problem, the output layer will consist of K nodes, one for each class. Using a softmax activation function for the last layer, we can view the output of node k as the probability $P(y^{(i)} = k \mid \mathbf{x}^{(i)}; \theta)$ since the the output of each neuron will range between 0 and 1 and the sum of all outputs will be 1. The softmax activation computes for each output k

$$\sigma(\mathbf{x})_k = \frac{e^{\mathbf{x}_k}}{\sum_{j=1}^{K} e^{\mathbf{x}_j}} \tag{2.1}$$

where $\mathbf{x}$ is the vector consisting of all outputs from the previous layer.

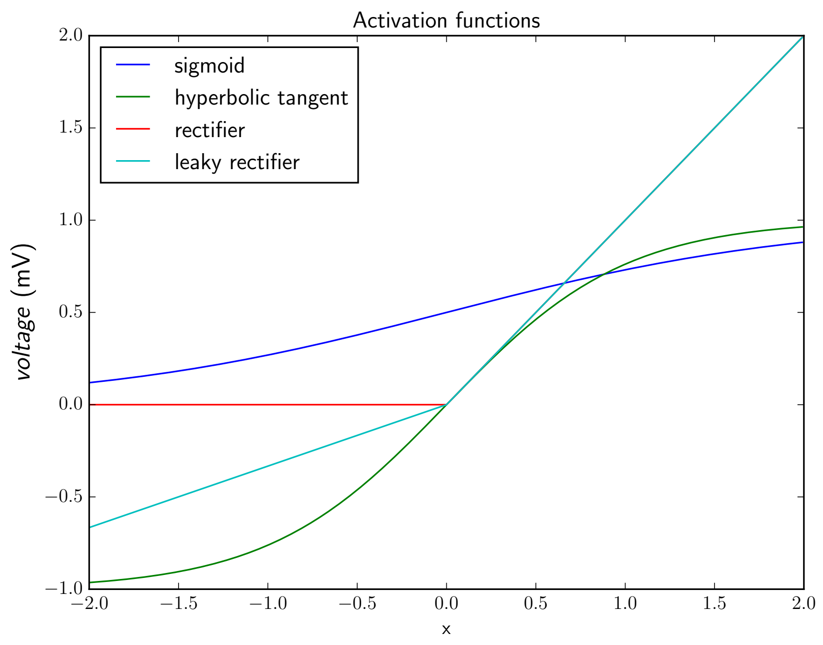

Figure 2.2: Plot of the activation functions.

Loss function

The next step is to compute how well our network approximates our training data with a loss function, to then decide how to change the weights of the network in order to minimise the loss. Since the aim of the network is to classify the central pixel(s) of the patch, we use the categorical cross-entropy loss function

$$\mathcal{L}(\theta) = \frac{1}{m}\Big[\sum_{i=1}^{m}\sum_{j=1}^{k} \mathbb{1}[y^{(i)} = k] \log(P(y^{(i)} = k \mid \mathbf{x}^{(i)}; \theta))\Big] \tag{2.2}$$

where $\mathbb{1}$ is the indicator function, $\log(P(y^{(i)} = k \mid x^{(i)}; \theta))$ is the probability outputted by the network.

Optimisation

The next step is to calculate the gradient $\frac{d\mathcal{L}(\theta)}{d\theta}$, so that we can apply gradient descent and update our weight vector to

$$\theta = \theta - \epsilon\frac{d\mathcal{L}(\theta)}{d\theta} \tag{2.3}$$

where ϵ is the learning rate, a small positive value.

Since every basic operation used in the neural network is differentiable, the entire network will also be differentiable, which in turn makes it possible to calculate the gradient of a loss function with respect to the weights in the network. This process is called backpropagation.

The loss functions, as described in equation 2.2 is computed using the entire training data set $\mathbb{X} = \{(\mathbf{x}^{(1)}, y^{(1)}), ..., (\mathbf{x}^{(m)}, y^{(m)})\}$. In the case of deep learning where it is usual to have a very large training data set, this would be very memory costly and slow down the training phase unnecessarily. Therefore, stochastic gradient descent is used instead, where we process the training data sequentially in batches, each time computing the loss for that batch and updating the weights.

Momentum

A further optimisation used to speed up the training process is momentum update. Minimising the loss function can be interpreted as moving a small particle down a hilly terrain in the hyper-dimensional space defined by the loss function. Since the gradient is related to the force experienced by that particle, this suggest that the gradient should only influence the velocity vector and not the position directly. This leads to the velocity update

$$v = \mu v - \epsilon \frac{d\mathcal{L}(\theta)}{d\theta} \tag{2.4}$$

where μ is the momentum, typically set to 0.9. We then update our weights by simply adding the velocity to the current value.

$$\theta = \theta + v \tag{2.5}$$

Typically a slightly different version, called the Nesterov momentum, is used as it has been shown to work better in practice[CITATION].

Further optimizations

Should I explain adam optimisation??

2.2.2 Convolutional neural networks

Convolutional neural networks are different as they make the explicit assumption that the inputs will be images. This allows us to take advantage of some properties to make the forward function more effective and greatly reduce the number of weights in our network.

A typical convolutional network constists of three types of layers: **convolutional layers**, **pooling layers** and **fully-connected layers**.

Fully-connected layers

These are exactly the same as for ordinary neural networks.

Convolutional layers

Unlike a fully-connected layer, a convolutional layer is typically three-dimensional: widht, height and depth. The parameters of a layer are a set of learnable filters, each spatially small along the width and height but with the depth equal to the depth of input volume. The layer then computes a two-dimensional activation map by convolving the filter with the input and computing the dot product at each point. This means that the function learned by a filter is space-invariant, as the filter is convolved with the entire input. Each filter computes such a two-dimensional activation map that can then be stacked along the depth axis to produce the output volume.

The connectivity pattern is inspired by the organization of the animal visual cortex. Individual neurons respond to stimuli in a small region of space known as the **receptive field**. Every element in the output can then be interpreted as the output of a neuron whose receptive field is the width and height of the filter and who shares its weights with all its neighbours to the left and right spatially.

The size of the output volume is determined by three hyperparameters.

1. The **depth** corresponds to the number of filters in the layer and is therefore equal to the depth of the output volume.

2. The **stride** which determines by how many pixels we slide the filter. When the stride is greater than 1 the output width and height will be smaller than the input width and height.

3. The **padding** determines with how many zeros we pad the input width and height. This is particularly useful when we want to preserve the input dimensions.

The computation done by such a filter in shown in figure 2.3.

Pooling layers

The function of the pooling layer is to reduce the spatial size of the input layer in order to reduce the number of parameters and computation in the network. The most common pooling layer implementation is the **max-pooling** which just convolves a two-dimensional maximum operator of a given size, typically 2x2. The **stride** again determines the step size.

A convolutional neural network typically consists of a sequence of these layers, starting with pairs of convolutional neural networks and max pooling layers. The idea behind this is that each pair can learn more and more abstract features using the features learned by the previous layer. For example, the first pair might learn to recognise edges, the second layer shapes, etc.. The last few layers of the network then consist entirely of fully-connected layers. These learn how to classify the data using the features learned by the convolutional and max pooling layers.

2.2.3 The overfitting problem

A common problem to arise in neural networks is that of **overfitting**, i.e. when the model isn't able to generalise and instead just memorises the training data. Due to the large number of weights in convolutional neural networks, this problem occurs frequently when using them. Many techniques and heuristics have been developed in order to help reduce overfitting. In particular, I used three of them: **L2 weight normalisation**, **Dropout** and **Batch Normalisation**

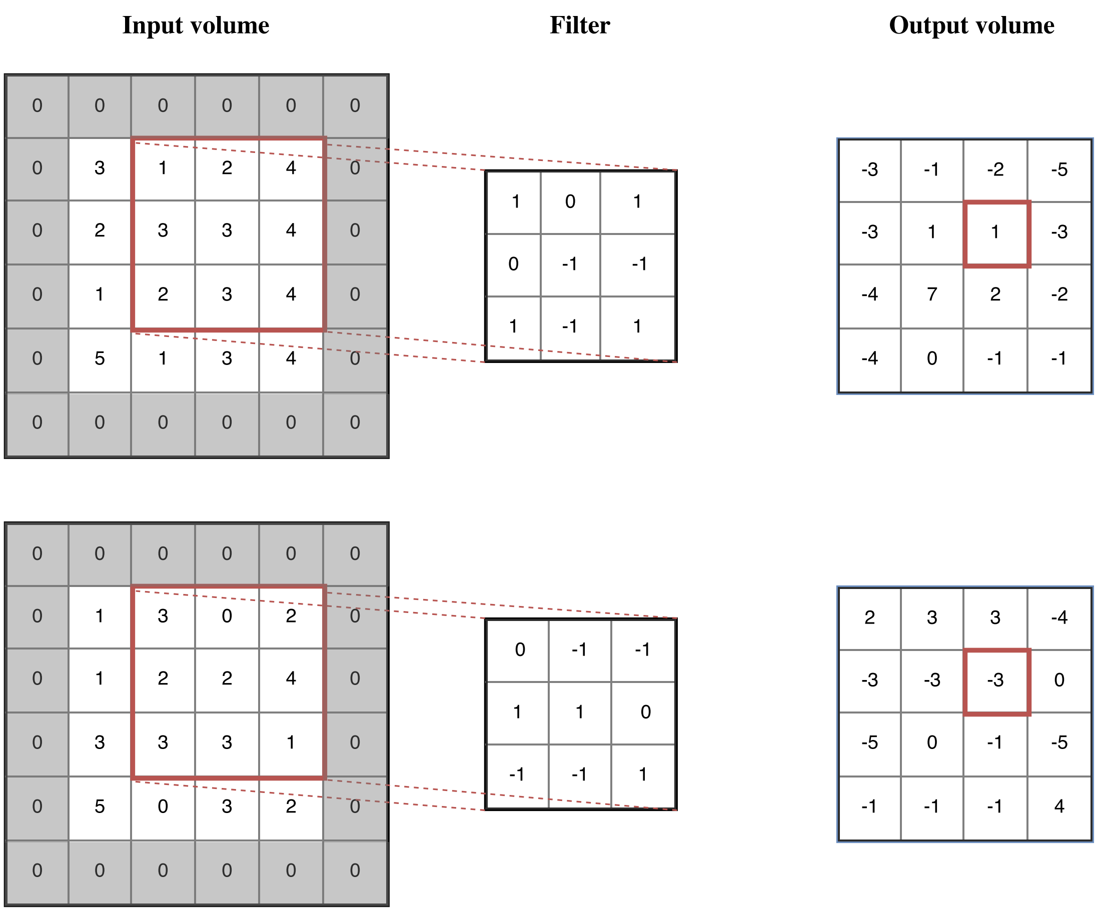

Figure 2.3: Example computation done by a 3x3 filter in a convolutional layer. Both the stride and the padding are 1, so as to keep the output dimensions identical. The output at depth 0 for the marked element is $1\cdot1+2\cdot0+4\cdot1+3\cdot0+3\cdot-1+4\cdot-1+2\cdot1+3\cdot-1+4\cdot1 = 1$ and similarly at depth 1 the output for that element is -3.

L2 weight normalisation

Dropout

Batch Normalisation

2.3 Data source

2.3.1 BraTS Challenge

1. Explain what data is available and in what format and how much. Also mention what has been done already to the data, i.e. that it has been skull striped (?) and aligned.

2. Explain that the distribution among the classes is highly non-uniform which can be a probelm, including table of frequencies.

3. Finally also mention that i am only interested in the high-grade glioma case explaining why (time reasons, harder, normally used to report results)

2.3.2 Online submition platform

Not sure this is worth an entire subsection, could be added to the intro chapter.

1. Explain how the segmentations are submitted
2. Explain the different test sets: Challenge and Leaderboard and explain why Challenge was chosen.

Chapter 3

Implementation [40%]

3.1 Data pre-processing

3.1.1 N4ITK

3.1.2 Normalizations

Winsorizing

Linear transformations

3.1.3 Patch extraction

3.2 Pereira model

I implemented the convolutional neural networks using the Keras (Add reference) library, which allows users to create neural networks by combining different layers together. The more common layers come as part of the library but Keras allows the user to define layers as well. The architecture proposed by Pereira et al [1] used the following layers:

- two-dimensional convolutional layers
- fully connected layers
- two-dimensional max pooling layers

Futhermore, we can easily add Dropout to the model by adding an instance of a Dropout layer where required.

My implementation of the model proposed by Pereira [1] looks as follows in Keras:

```
model = Sequential()

model.add(Convolution2D(64, 3, 3, border_mode='same', init=init, input_shape = input_shape, W_regularizer=l2(l)))
model.add(LeakyReLU(alpha))

model.add(Convolution2D(64, 3, 3, border_mode='same', init=init, W_regularizer=l2(l)))
```

```
    model.add(LeakyReLU(alpha))

    model.add(Convolution2D(64, 3, 3, border_mode='same', init=init,
   W_regularizer=l2(l)))
    model.add(LeakyReLU(alpha))

    model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), border_mode='
   valid'))

    model.add(Convolution2D(128, 3, 3, border_mode='same', init=init,
   W_regularizer=l2(l)))
    model.add(LeakyReLU(alpha))

    model.add(Convolution2D(128, 3, 3, border_mode='same', init=init,
   W_regularizer=l2(l)))
    model.add(LeakyReLU(alpha))

    model.add(Convolution2D(128, 3, 3, border_mode='same', init=init,
   W_regularizer=l2(l)))
    model.add(LeakyReLU(alpha))

    model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), border_mode='
   valid'))

    model.add(Flatten())

    model.add(Dense(256, init=init, W_regularizer=l2(l)))
    model.add(LeakyReLU(alpha))
    model.add(Dropout(0.1))

    model.add(Dense(256, init=init, W_regularizer=l2(l)))
    model.add(LeakyReLU(alpha))
    model.add(Dropout(0.1))

    model.add(Dense(5, init=init, W_regularizer=l2(l)))
    model.add(Activation('softmax'))
```

This makes it easy to create deep, convolutional networks and to experiment with them. This reason, together with the active community of Keras users is why I chose to use the Keras library.

Before training the models, we further need to specify which loss function and which algorithm should be used to respectively specify and minimise the loss function. Again, keras makes this very simple:

```
    sgd = SGD(lr = 3e-5, decay =0.0, momentum = 0.9, nesterov = True)
    model.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=[
   'accuracy', dice])
```

3.2.1 Architecture

3.2.2 Implementation of the architecture in Keras

3.3 My model

3.3.1 Architecture

3.3.2 Implementation

3.4 Training

3.4.1 Cambridge High Performance Computing cluster

3.4.2 NVIDIA TITAN X GPU

3.5 Segmentation

3.5.1 Pereira model

3.5.2 My model

Chapter 4

Evaluation (+ Conclusion [20%])

4.1 Metrics used for the evaluation

4.1.1 Regions of evaluation

4.1.2 Dice score

4.1.3 Positive predictive value

4.1.4 Sensitivity

4.2 Evaluation of the model proposed by Pereira et al.

4.3 Evaluation of my model

4.4 Comparison

Chapter 5

Conclusion

5.1 Summary of achievements

5.2 Future Work

I hope that this rough guide to writing a dissertation is LaTeX has been helpful and saved you time.

Bibliography

[1] Sergio Pereira, Adriano Pinto, Victor Alves, and Carlos A. Silva. Brain tumor segmentation using convolutional brain tumor segmentation using convolutional neural networks in mri images. *IEEE Transactions on medical imaging*, 35(5):1240–251, May 2016.