University of Chester

Visualisor for animating the mathematical operations involved in neural networks

Robert Graham Keeling

2019-2020

Research Dissertation

Department of Computer Science, Electronics and Electrical Engineering

Disclaimer

This work is original and has not been previously submitted in support of any other course or qualification.

Robert Graham Keeling, 03/05/2020

Abstract

This dissertation sought to create a neural network visualisor that focuses upon animating the mathematical operations involved in training neural network models, rather than upon creating meaningful data representations for knowledge extraction. The key reason behind this is to allow for people to more easily come to an understanding of how these networks work, as many people currently view them as a black box. This is true even for many of those who currently utilise APIs such as TensorFlow to create neural networks. The project was partially successful in that in animated many of the operations involved in training the architectures supported, but not all of the operations. The visualisor was functional but not particularly aesthetically pleasing, and while it allows the user to visualise a model, the model has to be loaded programmatically. The animations that where provided did offer an easy to understand visual explanation of how the operations involved in neural networks work.

Acknowledgements

I would like to thank all those who advised me throughout the development process of this project including;

- Neil Vaughn
- Andrew Muncey
- Richard Stocker
- Nigel John

Contents Page

List of Figures/Photographs

Introduction

Background

Neural networks where first conceptualised long before the computing power needed to implement large scale networks was available. For this reason, they were relatively obscure until recently. In recent years, large scale networks have been implemented for a variety of purposes (Roberts, 2017) and the number of fields to which they have been applied is continuing to grow. From facial recognition on modern smart-phones to personalised shopping recommendations, the utilisation of neural networks is ever expanding and offers numerous benefits to those companies that make usage of them.

Neural networks have even been used for the psychological manipulation of entire populaces in order to affect the result of democratic elections. Cambridge Analytica, amongst other groups, built a data-set of Facebook data such as likes and comments alongside psychological profiles of individuals composed by professional psychologists. They developed a network to learn a mapping between the Facebook data and the psychological profiles. When the network was trained it was used to make predictions about the psychological profiles of many Facebook users who had never been psychologically profiled. This information was used to target manipulative adds to people based upon their hopes and fears, for example an American who was scared of burglars would get an advert stating that Donald Trump would "defend their second amendment rights, to defend their home from burglars", while someone who liked hunting would get an add stating that Donald Trump would "defend their second amendment rights, to go hunting with friends". These targeted, manipulative adds where used by both Donald Trump's election campaign and the vote leave campaign in the Brexit referendum, as well as numerous elections in smaller nations around the world (Rosenberg et al., 2018).

Aims, Objective and Hypothesis

Due to the rapidly growing number of applications for neural networks and the huge influence these networks are having globally, interest in the subject is increasing consummately. The topic is often shrouded in mystery and even amongst those who utilise networks for various tasks, knowledge of how these networks work internally is often lacking (Bleicher, 2017). This project aims to shine a lot on the internal workings of neural networks, it aims to achieve this objective by developing a visualisor that will provide visualisations of all tensors within a network and animations showing what operations are performed on those tensors during both the forward and backward pass of a network. The scope of this project has been limited to only fully connected networks and convolutional networks, as attempting to include support for many network architectures in the time frame available would be infeasible. The project also seeks to implement the network itself as well as the visualisor. Several visualisers for neural networks have been developed but they focus on creating meaningful representations of data for knowledge extraction and not upon creating easy to understand animations of the processes. The hypothesis of this project is that:

> The visualisor developed will be better than pre-existing visualisers at explaining the mathematical operations involved in neural networks.

Personal Involvement

My own personal motivation for the project is to improve my own understanding of neural networks. For my AI module, I implemented a simple fully connected network using 2D tensors (matrices). I wanted to build upon this work and further my knowledge of the topic. As I was reading up on operations such as max-pooling and convolution, I found it quite difficult to understand and visualise the operations. Once I did grasp how these operations work, it became clear to me that I could create software to visually explain the operations and that such animations would have been incredibly useful to me when I was first attempting to grasp these concepts. I am considering a career in data-science and I am hoping that this dissertation will be a step towards a role in this field.

Stages of Work

The majority of the work involved in the project was programming based, much of the design of the network API and visualisor was done within the python language, defining classes with empty bodies first and then filling the classes with function stubs before finally implementing the function bodies. The tensor class was implemented first, then the network layer types, then the overall network and code to create a network. Once this was complete several networks where created and trained using a variety of architectures. Finally, the network visualisor was developed, starting with a tensor visualisor, then layer visualisers and finally the overall network visualisor. Once the visualisor was finished it was tested for bugs and the code tweaked to iron out a few issues.

Literature Review

Overview

Neural networks where first conceptualised in the 1940s (Anderson, 1998), however due to the computational power needed to train large scale networks, they have only recently come into widespread usage in a variety of fields. Neural networks can be mathematically described as a differentiable mapping between an input set and an output set. The input set could be audio data, images, videos or indeed any other type of data such as personal or sales data. The output set may represent classifications, boxes around objects in images or objects in 3D space, amongst many other possibilities (Dar, 2019).

Initially the mapping between the input set and the output set is randomised, the weights and biases within the mapping are learned via the process of gradient descent. Gradient descent requires both that the mapping is differentiable and a loss function from which the error of the network can be calculated. The loss function may be known, as is the case in reinforcement learning, in which many known examples are available (there is labelled data available). The loss function of a reinforcement learning based network can be as simple as the cross entropy of the output of the network from a given input with the desired output for the given input. However, the loss function may be unknown and can be made to be a learnable parameter, as is the case in Q learning (Shyalika, 2019).

Fully Connected Mappings Via Matrices

The design of the mapping used within a network can have a huge impact on the overall accuracy of the network. The simplest design (excluding single perceptrons, etc.) for the mapping is fully connected, in which neurons are organised into layers and each neuron in a layer connects to every neuron in the previous layer. Each neuron takes the weighted sum of every neuron in the previous layer, applies a differentiable activation function to that sum and outputs the result of that operation. The weights used are the learnable parameters of the network (Sakryukin, 2018).

This mapping can be modelled using matrices, as an example, consider a data-set with 900 values per input and 12 possible classifications for an input. The input could be represented as a matrix of dimensions 1x900 and the output would be a matrix of dimensions 1x12. The known labels need to be encoded in such a way that each classification is linked to one of the 12 output nodes, this is achieved by representing the classification as a 1x12 matrix where one of the 12 values is set to 1 and the rest are set to 0. A fully connected network with no hidden layers would use only one weight matrix that directly maps the input to the output. For this example, a matrix of dimensions 900x12 would be used, taking the dot product of input matrix (1x900 dot 900x12) would result in a matrix of dimensions 1x12, the same dimensions as the encoded classifications. It is possible to pass multiple examples through the network at one time by encoding the input matrix to be (n x input neurons) and the output of the network to be (n x number of classifications) where n is the number of examples to be viewed at once, this has no impact on the weight matrices provided that the updates to the weights in the backward pass are divided by n (Sakryukin, 2018).

Modern neural networks do not usually map the input to the output directly, instead they map into intermediate values (hidden layers) which are then mapped further to the output, this allows for networks to develop deep internal abstract representations of the inputs that build

upon the representations in the prior layers (Brownlee, 2019). An example of such an architecture for the data-set described above could include two hidden layers of 100 neurons and 50 neurons respectively. In this case the network would require three weight matrices of dimensions, 900x100, 100x50 and 50x12. The image below shows the matrices involved in creating a fully connected neural network of the specified dimensions, note that the matrices are labelled tensors as matrices are simply tensors of two spatial dimensions.

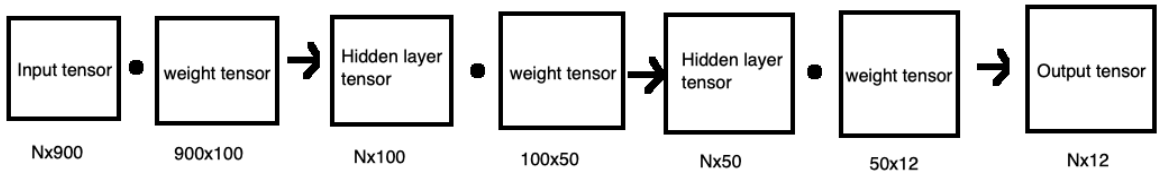

Figure 1 – fully connected network matrices example

Backpropagation, Learning Rates and Gradient Descent Optimisation

Backpropagation works by first calculating the error of the network and then calculating the gradient of the output neurons with respect to the loss function, from this the gradients of the weights to the previous layer with respect to the loss function are calculated. In addition, the gradients of the neurons in the previous layer with respect to the loss function are calculated, from these values the step is repeated until the gradient with respect to the loss function has been found for every weight in the network. Once the gradients are known for each weight, the values of the weights are adjusted in order to travel down the gradient reducing the value of the loss function (Nielsen, 2019). Over many iterations over the training data the networks mapping becomes increasingly accurate and can eventually be used to make predictions about data for which the classification is unknown.

The amount that a weight is to be adjusted by is normally the negative of the gradient of the weight with respect to the loss function multiplied by a hyperparameter known as the learning rate. The learning rate is an important hyperparameter as it affects how quickly the network adjusts its weights. Too low of a rate will result in a network that gets stuck in local minima of the loss function, resulting in poor performance. Too high of a rate will result in a network that never converges, again, resulting in poor performance. A common approach is to start with a higher rate and then gradually reduce the rate over time. Other newer approaches include cyclical learning rates and a wide variety of adaptive learning rate implementations (Smith, 2017).

One technique that can improve the training time of networks and to a lesser extent the performance of those networks is known as momentum. This technique entails storing the first set of weight updates from the first cycle of the network, then upon the next cycle the weight updates are added to the first set of weight updates and multiplied by a value (momentum decay value). These momentum tensors/matrices are then used to update the weight values. This technique means that if a gradient is consistent over many cycles the rate at which it is traversed is increased, allowing for minima of the loss function to be more quickly discovered. There are many variations of this technique including; Adam, Adagrad, Adamax, Nadam and RMSprop (Ruder, 2016).

Activation Functions

An important consideration in the design of neural networks is the choice of activation functions, in each layer (aside from the input layer) a differentiable function is applied to the weighted sums in order to introduce non-linearity into the network. Common choices for this activation include; sigmoid, tanh, ReLu and leaky ReLu (Bircanoğlu & Arıca, 2018). Many more activations exist and the development of novel activation functions is a field of active research within the neural network community. Currently, tanh is most frequently used in fully connected layers while ReLu or leaky ReLu is used in convolutional layers within CNNs. One benefit of ReLu over sigmoid or tanh is that it mitigates the problem of disappearing gradients that occurs in deep networks using sigmoid or tanh activations. Vanishing gradients occur where networks have many hidden layers and the gradient backpropagated to early layers in the network becomes increasingly small, hindering learning (Kolbusz et al., 2017).

Advanced Network Architectures

Fully connected networks are the simplest design and in recent years many newer architectures have been developed. Architectures are designed take advantage of known relations between the input neurons, for example convolutional networks take advantage of the fact that images are 2-dimensional by encoding each input as 2-dimensional matrix. This allows for the network to better capture special relations between input neurons (pixels) than in a fully connected architecture. CNNs can also be applied to non-image data-sets where the data can be meaningfully represented as 2-dimensional matrix (Gu et al., 2018). Another modern architecture are recursive networks, these work well on time series data sets as they are designed to consider the classification of an input based upon the inputs that preceded it (Venkatachalam, 2019). LSTMs are another new architecture that work particularly well on time series data, they incorporate specially designed memory layers within the network to better allow for the learning of long-term patterns.

Other architectures have been designed that produce output other than classifications, for example, FCNNs (Fully Convolutional Neural Networks) output 2d tensors that represent boxes around images in pixels allowing for not only image classification but for the identification within an image of where objects are positioned (Milletari et al., 2016). CNNs or FCNNs can be extended to provide support for video data and other complex data formats by including recursive layers within their network architectures (Wang et al., 2016). This allows for networks to analyse video where the desired output depends upon not just one image, but a series of images. GANs (Generative Adversarial networks) can be used to generate images or other data. GANs consist of two networks, a discriminator and a generator, the generator attempts to create fake input data that is consistent with the training data while the discriminator attempts to distinguish the fake input data from the generator with real input data from the training set. The output of the discriminator network is used to calculate the loss of the generator network (Radford et al., 2015).

Network Overfitting and Dropout

One problem with large networks, especially when training on limited data-sets, is that the network can overfit the training data. When a network is overfit it will perform very well on examples it has seen but will perform much more poorly on new examples. Overfitting is caused by the network becoming dependant on small non-general features of the input data in order to make its' predictions. One approach to avoiding this problem is to train many models on the data-set and form a composite network from these models. This approach has

numerous drawbacks, the main problem is the increased time taken to train numerous networks in order to develop a composite network (Srivastava et al., 2014)
.

The current standard approach to dealing with the problem of network overfitting is via the usage of dropout. Dropout is simple to implement and simply involves preventing a given percentage of neurons within a layer from firing during training. When a neuron is dropped out during the forward pass of the network it is also not updated during the backward pass. Different neurons are dropped out each cycle, although the percentage dropped out is normally the same across cycles. The result of this is that the network does not become dependent on any one internal representation and is forced to develop many internal representations of objects; this results in networks that generalise much better than those trained without drop-out. Drop-out is only applied during training, during testing or usage every neuron is allowed to fire (Srivastava et al., 2014).

Pre-Existing Visualisers

Several visualisation engines have been developed for visualising various aspects of data and networks; however, these visualisers focus on creating meaningful visual representations of data in order to extract knowledge from the data (Samek et al., 2016). While these visualisers are more useful in terms of data representation, they do not seek to visualise all of the mathematical operations performed on data representations. That is the key differentiating factor between this visualisor and other pre-existing visualisers, this project will seek to show all the operations that are performed within a variety of network architectures in order to allow users to more easily understand them.

Design

Design of Visualisor and User Interface

Many complex visualisation engines have been designed to visualise data and networks in order to extract knowledge from data and gain new insights into a variety of fields. These visualisers are designed to utilise GPUs and can produce complex multi-dimensional data representations. The key aim of the visualisor developed during this project was different from the key aim of other visualisers, in that this visualisor seeks to shine a light on the mathematical operations involved in neural networks, rather than to create meaningful visual representations of data. For this reason, it was decided that the visualisor would only display a 2D slice of any given tensor at any one time, as this would be the most understandable and easy to follow way to animate the mathematical operations performed within a network. In addition, implementing a GPU based 3D graphics program on top of the network implementations seemed somewhat infeasible in the time frame of the project.

As only a 2D slice of any given tensor would be visualised at any given time, it was clear that the tensor visualisor component would need to provide support for navigating through the higher spatial dimensions within the tensor. It was also clear that when displaying multiple tensors that are related in some way, there would need to be a linking so that navigating one tensors higher special dimensions would also correctly navigate through the related tensors higher special dimensions. An example of this would be displaying all the tensors involved in a forward pass through a convolutional layer, navigating through the dimensions of the input tensor should also navigate through the dimensions of the output tensor, as should navigating through the third special dimension of the kernel tensor.

Convolution is often described as sliding a filter over an image, it was this description that inspired the animation of the convolution operation. In the animation a highlighting rectangle would pass over the input tensor and the values within the rectangle would be displayed in an enlarged view, near to where the kernel tensor is displayed. In addition, the result of the cross product between the selection and the kernel tensor will also be displayed in an enlarged view, near to the selection and kernel tensor displays. The sum of the values in the cross-product tensor will be displayed and a highlighting rectangle will display where this value was placed in the output tensor. This process will continue until all the selections from the input tensor have been animated.

It was clear that the max-pooling operation could be displayed in a similar manner to the design of the convolutional animation, with a rectangle highlighting a selection of the input tensor and that selection being displayed in an enlarged view. However rather than displaying additional tensors the enlarged view would merely need to highlight the largest number it contains, while a highlighting rectangle on the output tensor indicated where this largest value ended up being placed within the output tensor.

The plan was to include several settings within the visualisor to allow the user to adjust various aspects of the visualisation, one of the key settings would be whether or not to display the digits within the tensors and also how many characters to truncate the digits at. Settings for the animations where to be implemented, offering users the ability to start the animation, pause the animation and speed it up or slow it down. In addition, controls where to be included to allow the user to manually click on values within the tensor in order to select them for inspection with any of the operation visualisers. There would also be an option to

change before the view between the forward and backward pass operations of the network, to allow the user to easily access both sets of animations.

The visualisor was also to provide support for displaying the tensors involved in and animating the operations involved in the backward pass through the network. For this it was clear that the final output of the network would need to be displayed next to the desired output of the network and the result of taking the actual output from the desired output would need to be displayed next to these. From this each step in the backwards pass would need to be displayed; this step would vary for different layers.

In order to animate the backward pass through a max-pool layer the input to the reverse-max-pool operation will need to be displayed alongside the mask tensor from the original max-pool operation and the process through which the delta values are mapped to the original larger image size animated. A similar approach could be taken with the convolutional layer, with the backward pass animation displaying the input to the reverse-convolutional operation alongside the convolution mask and the output of the reverse-convolution operation.

In order to animate activation functions within the network it may be best to simply display the equation and show the operation taking place step by step for a single value and then to simply explain to the user that this operation was performed for all values within a tensor. It would also be valuable to include a graphical representation of each of the activation functions in order to best explain their function to the user. The activation functions could be included with the displays for the more complex operations such as convolution or they could be represented as their own layer with their own unique visualisers.

The overall network visualisor was designed to make use of both the tensor visualisor and the operation visualisers, offering a menu bar to navigate through the various layers of the network. The menu bar was to allow for the user to select both layers and the operations linking those layers, as well as any tensors used in the operations between layers. Therefore, the user could view the input tensor on its own or view the input tensor alongside the kernel tensor and the result of the convolution of the two in the convolution view and also view the kernel tensor and result tensor separately within their own displays.

Overall this design aims to explain and animate the operations within neural networks in the simplest way possible, so that people may more quickly come to a deeper understanding of the internal workings of these networks. The user should be easily able to navigate between forward and backward passes of the network and be able to select each layer in order to see an easy to understand animation of the operations involved in each layer.

Choice of Language

Many languages where considered for this project, the main options came down to Rust, C and Python. Rust was a consideration due to its performance and safe concurrency, with compile time checks for data-races and other potential pitfalls of parallel programming (Klabnik & Nichols, 2019). One downside to rust was that it's a new language and does not have as developed GPU interfaces as C. However, it became clear that it would be extremely difficult to include GPU support within the project within the time frame allocated. In order to achieve the key aims of the project, it was decided that python would be used for the development in order to allow for faster development and the inclusion of more features within the time frame available. While the choice to use python will result in a library that is

slower than those already available, the primary aim of the project is to visualise all of the mathematical operations that occur within different neural network architectures and the reduction in performance should not impact too heavily on this goal.

Programming Paradigm

An object orientated approach was taken for the project, with initial planning of what classes would be necessary within the project and how those classes relate to one another. This planning was performed within the python programming language, each class was initially declared containing only a pass statement. Once all the necessary classes had been identified and the relationships between the classes declared, the classes where filled with function stubs containing only pass statements. This technique incorporates the planning aspects of the project into the development process and allowed for more rapid development than other approaches (Parmar, 2019).

Key programming theory and concepts drove the development of the visualisor, with abstraction and composition informing much of the design. Putting these theoretical concepts into practise resulted in a code-base that was easy to navigate and in which bug-tracing was not a major issue. Another key concept that was applied to the design was the Liskov-Substitution Principle, which states that subtypes of a given type should be interchangeable for the parent type (Haoyu & Haili, 2012). This informed the design of the network layer classes as each was designed to provide an interface that was consistent across all the layer classes, allowing for the network class to treat all layers in the same manner except for where they need to differ.

Design of Classes

Several classes where required for the project; a tensor class, a class for each layer type implemented, a network class and several classes for the visualiser. As CNNs where to be included within the project, it was clear that the tensor class would have to support tensors of n-dimensions and would therefore need to be recursively defined. Each layer class would utilise the tensor class and the network class would utilise the layer classes, taking a compositional approach to the design.

The visualiser was split into separate components that where logically distinct. Firstly, a class for visualising a single tensor, displaying a 2d slice of a given tensor and providing controls for navigating through any additional dimensions. Then a class for each operation that would animate the operations and provide controls for those animations. Finally, a class that would visualize a complete network. The classes for visualising the operations would utilize the tensor visualisor class and the class to represent and animate an overall network would utilize the operation visualisers.

Design of Tensor, Layers and Network

The design of the actual neural network API was also informed by composition and abstraction, with the tensor class handling all logic relating to operations on tensors and between tensors. The tensor class needed to provide functions for numerous mathematical operations such as addition, subtraction, division or multiplication of either a number or another tensor. More specialised operations were also required including; dot-product, max-pool, convolution, reverse-convolution and numerous activation functions. The tensor class also needed to be able to be initialised either randomly from a set of dimensions or from a known n-dimensional list or array.

Each Layer class was designed to have a consistent interface so that the network class could treat them in the same manner. The layers where designed to store tensors and call various operations on these tensors in a variety of manners depending on the layer type. Each layer was to have a forward pass method and a backward pass method, the forward pass would take an input perform a set of operations upon the input and return an output, the backward pass would take an error gradient tensor as input and perform a set of operations upon it, calculating error gradient tensors for both its own value and for the values that where passed to it in the forward pass. The backward pass of each layer would return the error tensor of the previous hidden layer, this error tensor is then passed as input to the backward pass of the previous layer until all the weights within the network have been adjusted.

Implementation

Tensor Class Implementation

The tensor class was the first class implemented in the project, as the tensors where to allow n-spatial dimensions, it was clear that many of the functions could be most easily represented as recursive functions or recursive function pairs. The first functions to be developed where initialisation functions that allowed for a tensor to either be initialised with pre-existing data or for a new random tensor to be generated from a list of dimensions.

The next functions to be developed where for basic mathematical operations on tensors, such as; addition, subtraction, division and cross-multiplication. These basic operations would need to accept either another tensor of equal dimensions or a stand-alone number as an input. Once the function to add two tensors or a tensor and a number was designed, it was clear that subtraction, division and multiplication could be implemented in much the same way. This is because the logic to access the data in the multidimensional arrays is the same for all of these basic operations is the same and the only difference is whether to add, subtract, multiply or divide once the values have been properly accessed. It was therefore clear that higher order functional programming could be utilised in this case in order to reduce code repetition and to allow for a sleeker, easier to read code-base. Shown below is a code snippet that highlights this usage of higher order functional programming techniques.

```
def recursive_math(array, other, func, start=False):
    if isinstance(array, Tensor):
        array.check_dims(other)
        array = array.array
    other = other if not isinstance(other, Tensor) else other.array
    to_return = []
    if isinstance(array[0], list):
        if isinstance(other, list):
            for i, j in zip(array, other):
                to_return.append(Tensor.recursive_math(i, j, func))
        else:
            for i in array:
                to_return.append(Tensor.recursive_math(i, other, func))
    else:
        if isinstance(other, list):
            for i, j in zip(array, other):
                to_return.append(func(i, j))
        else:
            for i in array:
                to_return.append(func(i, other))
    if start:
        return Tensor(array=to_return)
    return to_return

def __add__(self, other):
    return Tensor.recursive_math(self, other, lambda x,y:x+y, True)

def __radd__(self, other):
    return Tensor.__add__(self, other)

def __sub__(self, other):
    return Tensor.recursive_math(self, other, lambda x,y:x-y, True)

def __rsub__(self, other):
    return Tensor.__sub__(self, other)

def __mul__(self, other):
    return Tensor.recursive_math(self, other, lambda x,y:x*y, True)

def __rmul__(self, other):
    return Tensor.__mul__(self, other)

def __div__(self, other):
    return Tensor.recursive_math(self, other, lambda x,y:x/y, True)
```

Figure 2 – higher order function code snippet

The above code snippet also demonstrated the use of python's double underscore methods, these "magic" methods can be implemented by custom classes that wish to emulate the behaviour of built in types. For the purposes this project this is extremely useful as it allows for operator overloading, whereby two tensors may be added together in code by simply placing a + symbol between them, rather than by calling the __add__ method of the first tensor with the second tensor or number as an argument. Defining __radd__ allows for numbers to be added to tensors in code by writing 5+A as well as A+5 (where A is a tensor), without this method only the later would be valid. Other "magic" methods where utilised by

the tensor class including __str__ which affects how a class is printed by a print statement, these allowed for quicker development and clearer code in classes that utilised the tensor class. In addition, the usage of the lambda operator within python to create anonymous on the fly functions is shown in the code above. Lambda statements are extremely useful as they allow for creation of small functions without a full def statement, they are especially useful for creating button call-backs that take arguments programmatically while iterating over a collection.

After this several activation functions where developed, they only operate upon a single tensor and where relatively trivial to implement in comparison to other aspects of the project. All of the activation functions take an argument called deriv that tells the function whether to apply the activation function or the derivative of that activation function. The activations that where included in the project include; sigmoid, tanh, ReLu, leaky ReLu and SoftMax. Each of these activations are used for different purposes and the choice of which activation to use after each layer is an important consideration in network design.

Several more specialised functions where required within the tensor class. These functions included the dot-product operation, max-pooling and convolution as well as reverse methods for max-pooling and convolution. All of these functions involved traversing through n-dimensional tensors in different ways and performing various operations on the data-sets. It was also necessary to include methods for reshaping tensors; this is often necessary when using tensors for neural network. For example, the output of a convolutional layer must first be flattened into a 2-dimensional tensor before it is passed to a fully connected layer.

Implementation of Network Layer Classes

A class was developed for each layer type to be included within the network library, these classes where designed to have a shared interface, with all layer classes containing a forward and backward pass function that take the same arguments and return the same type of values. This shared interface allows for the network class to use the layers interchangeably and not worry about the internal differences between layer types. The first layer type developed was that of the input layer, this layer was the simplest as it simply stored an input tensor and returned this input tensor when forward pass is called on it.

The next layer type to be implemented was a fully connected layer, this layer type stores a tensor within it that represented the weights of the layer. The forward pass of this layer type takes in an input tensor, it then calculates the dot-product of the input tensor with its internal weight tensor and returns this as an output. As with all the other trainable layer types, the forward pass also takes an argument called train which specifies whether drop-out should be applied to the output of the layer. Another key component of the forward pass of this component is that it checks if the input tensor is of greater than 2-spatial dimensions and flattens it if it is before attempting to take the dot product. The backward pass of this layer takes a tensor that represents the error gradients of the loss function with respect to the output of the layer, from this the gradient with respect to the weights of the layer and of the input to the layer are calculated. The gradient with respect to the weights is multiplied by the learning rate and then used to update the values of the weights; the gradient with respect to the input to the layer is returned and used by a previous layer as input to the previous layers backward pass method.

The next layer type to be implemented was the max-pool layer type. This layer performs a max-pooling operation on the tensor passed to it in the forward pass. During the forward pass

the max-pooling mask generated by the operation is stored within the layer and the pooled tensor is returned. The backward pass of this layer calls a reverse max-pooling operation on the delta tensor passed to it and the mask that was generated during the forward pass. The result of this is that error gradients are only propagated backwards through neurons that contribute to the final output of the network.

The convolutional layer type was the next to be implemented, this layer type internally stores a tensor of three spatial dimensions, which represents a set of image kernels. During the forward pass this layer performs a convolution operation on the tensor passed as input and the kernel tensor, this operation returns both an output and a mask for usage in reversing the convolution operation. The mask is stored for later usage and the output tensor returned from the forward pass for utilisation within the network class. During the backward pass of this layer, the mask generated by the forward pass is used to reverse the convolution operation. The mask stores information about how much each input contributed to a given output and it is used to ensure the error gradients are back-propagated correctly.

Finally, an output layer type was implemented, this layer inherits from the fully connected layer type and simply sets some default parameters for a fully connected layer that are necessary for an output layer. One such parameter is that the activation function needs to be soft-max rather than tanh or sigmoid, this is because soft-max produces a probability distribution where all the classification probabilities sum to one. In addition, drop-out is never used in the final layer of a network, therefore this convenience class sets the value of drop-out to zero.

Network Class Implementation

Once all the layer types had been implemented, the network class was implemented to link the layers together into a network. The forward pass through the network was fairly simple to implement, it simply takes an input tensor and passes it through the first layer, using the output of each layer as the input to the next. As all code related to reshaping input tensors for various layer types is handled internally by the layer classes the forward pass operation within the network can be relatively simple. The same is true of the backward pass through the network, however the network class handles momentum for the backward pass so the code within the network for a backward pass is slightly more complex than for the forward pass.

The forward and backward pass methods of the network need not be called by an external user, as they are wrapped by higher level convenience methods such as train, test and predict. The train method takes two tensors, one representing inputs and another representing desired network outputs, it also optionally takes a learning rate, number of cycles and batch size. The batch size is used to split the training set into smaller batches to improve training time, this is known as stochastic gradient descent. This train method performs all the necessary forward and backward pass operations on the network to train the network. Predict simply performs a forward pass over an input tensor and performs a one hot operation on the output of the of the network, returning this prediction tensor. Test is used to check the networks accuracy; it takes an input tensor and a desired output tensor and returns a score representing the networks accuracy on the test data set. Both test and predict do not apply drop-out to any of the layers, as drop-out should only be used during the training process and not while using a trained network.

Implementation of Network Visualisor

Tkinter was used for the development of all of the graphical components of the project as it is simple to use and allowed for rapid development of the visualisor. The network visualisor class was to consist of individual components for handling increasingly high-level tasks. A tensor visualisor class was developed that could display a given tensor as a 2D slice through the tensor with controls for navigating through the higher special dimensions of the tensor. The tensor visualisor class also included a function for highlighting selections of 2D tensor slices and returning a sub-tensor containing the highlighted values, this function was used by the convolution and max-pool visualisers to animate those operations step by step. It also included a function for linking the navigation controls of two or more tensor visualisers so that related tensors could be displayed in an easy to navigate manner. The tensor visualisor also allowed for the highest value within a 2D slice to be highlighted in yellow.

The convolution and max-pool visualisers where implemented on top of the tensor visualisers, both utilising the linking feature of the tensor visualisor class. They both also made use of the highlighting and sub-selection tool to animate their operations, the sub-selection tensors returned by the highlighting function where used to display enlarged views of the sub-selections in their own tensor visualisers. For the max-pooling operation the tensor visualisor class's max value highlighting feature was utilised by the enlarged sub-selection tensor visualisor. Finally, an overall network visualisor was implemented using these components. The network visualisor provided a navigation bar to select different layers in the network as well as individual tensors within the network.

Testing and Evaluation

Testing of Network Performance

The networks performance was only tested on the MNIST data-set, if the primary aim of the project was to develop a network API to outperform other APIs it would have been important to test the APIs performance over a wider range of data-sets. However, this was not the primary aim of the project and it was only important that the API learned from data successfully and it was clear from testing on the MNIST data-set that this was the case. Unfortunately, most networks trained only achieve a network accuracy of 80%, with a few approaching 85%, while this was certainly far from the performance of APIs such as TensorFlow (which can reach over 99% on this data-set), it was enough to demonstrate that learning was taking place within the network.

Testing of Visualisor

The visualisor achieved many of the aims it set out to achieve, but not all of them. The visualisor succeeded in creating representations of tensors as 2D slices with controls for navigating through the higher spatial dimensions of the tensor. It also succeeding in displaying the operations involved in a forward pass through both a convolutional layer and a max-pooling layer. Another key success of the visualisor was in the linking of the navigation controls through the higher spatial dimensions of related tensors, both in max-pooling where the relationship is simple and in convolution which involved three related tensors. The visualisor failed in that it did not provide animations for the operations involved in a backward pass through a convolution layer and max-pool layer, in addition no meaningful animation was provided to show the dot-product operation that occurs within a fully connected layer.

Overall, the visualisor met most of its key aims, providing meaningful animations of many operations involved in neural networks and with further development it could achieve the overall aim of providing animations for all of the operations involved in training convolutional neural networks, as well as possibly including support for other architectures. The overall hypothesis of the project was found to be correct as although the visualisor did not provide meaningful animations for all the operations, it did provide meaningful animations for many of them and these animations successfully show how the operations are carried out on tensors.

Conclusion

Strengths of the Project

The project had many strengths and many of the key aims of the project where achieved. The visualisor successfully provided animations for many of the key operations involved in neural networks. In addition, the network class was successful in that it was successfully able to learn a mapping from data. Much of the code in the project is succinct and well organised, following python standards for naming conventions and adhering to the style rules specified in PEP xxx. In addition, the usage of higher order functional programming within the code greatly reduced code repetition and allowed for a smaller code base. The application of abstraction and composition within the project resulted in well organised code that is easy navigate.

Weaknesses of the Project

The project did fall short of meeting all of its objectives, animations where not provided for all operations involved in training a CNN or a fully connected network. Specifically, there was no animation for a dot-product operation or for any of the backward pass operations. In addition, the backward pass of the convolutional layer was never properly implemented with the error gradient only being calculated for the weights of the kernel tensor and not for the input to the layer. The visualisor was also somewhat rushed and the design of the interface, while functional, is certainly not ideal and does not represent good UI design; this was caused by an over focus on trying to implement more features within the network and tensor classes, resulting in a rush to finish the visualisor. In addition, there are many aspects of the code base that would benefit from refactoring; for example, the activation functions within the tensor class all have to access the internal array in the same manner, a higher order function could be written to access the array and take as a parameter an activation function to apply to each value in the array.

Potential Improvements

There is much room for improvement within the project, the most obvious of which would be to provide support for all of the operations involved within the architectures already included. This would mean that the visualisor provided animations for every mathematical operation involved including the backward pass operations, activation functions and dot product operations. Fixing the backward pass through the convolutional layer of a network would also be a massive improvement for the project. In addition, it would be valuable to use existing libraries to perform some operations so that the output of the operations could be checked against these established libraries to ensure that the operations are implemented correctly within the project.

There are many areas of the code base that would massively benefit from refactoring, the most obvious areas of concern are within the visualisor components as this aspect of the project was somewhat rushed. The design of the visualisor could be massively improved to provide a more consistent user interface, as the current design is fairly scattershot with more focus paid to functionality than to appearance. The tensor class could also be expanded to provide support for direct indexing and slicing of the internal array by implementing the “magic” methods necessary to emulate the behaviour of array types.

Once the improvements had been made to provide animations for all of the operations involved in the currently supported architectures and to fix the backward pass through the convolutional layer, it would be possible to begin to include support for other architectures. The most obvious architecture that support could next be provided for is Fully Convolutional, this architecture does not require any additional layers to be implemented as it simply makes use of all the layers involved in convolutional networks except for fully connected layers. This architecture can be used to not only identify what is in an image but where an object is in an image, for this reason they are utilised within the larger AI systems of self-driving cars amongst other things. Rather than labelling training data with classifications, this architecture requires data to be labelled as boxes around objects in images. Beyond FCNNs, several other architectures could be supported; generative adversarial networks (GANs) could be particular interesting to visualise. Other architectures that could be supported included; recursive networks (RNNs), recursive convolutional networks (RCNNs) (particularly useful for video data) and autoencoder networks (utilised in natural language processing).

The user interface would also benefit massively from being expanded to provide support for loading in training data and models within the GUI, as currently the only way to load a model into the visualisor is programmatically and there is no way to train a model within the GUI. Including the ability to load in a model through the GUI and design and train a model through the GUI would massively improve the accessibility of the program and would allow less technical users to effectively use the software.

Another area for improvement is the manner in which data can be loaded into the software, currently the software only supports data that has been converted to a python list and pickled. The software would benefit massively from being extended to provide support for loading in data that is stored in a variety of common data storage formats, such as; CSV, JSON or petastorm. This would allow for data to be loaded from these formats without requiring the user to write the format conversion code themselves.

It would be massively beneficial to rewrite the tensor and network classes to make usage of GPUs. This is because many of the operations involved in these classes are highly parallelisable and because GPUs can perform thousands of time quicker than traditional CPUs on parallelisable problems. If this improvement where to be made the network class could be utilised to create much larger models in reasonable time. Training a model for images containing millions of pixels using the current network implementation would take an unreasonable amount of time, making use of GPUs would allow for such models to be created within a reasonable amount of time. Making use of more specialised equipment, like googles TPUs (Tensor Processing Units) or other ASICs (Application Specific Integrated circuits) would increase the performance of the network further and allow for the training of even more complex models.

Final Summary

To summarise, the project succeeded in many ways and many of the key aims where achieved. A neural network library was developed that could effectively learn from data and visualisation software was developed that effectively animated many of the operations involved in the neural network architectures supported. Although animations where not provided for all of the operations, the animations that where provided where effective at explaining visually how the operations work. In addition, the linking of related tensor visualisers was successful and the user was easily able to navigate through the different tensors and layers within the network. There are many areas in which the project could be

improved, both in terms of improving the performance of the network class and in terms of improving the appearance and functionality of the GUI. Overall, within the time frame available many complex features where successfully implemented and the hypothesis of the dissertation was found to be true.

In order to further develop my understanding of this topic, I intend to further develop the software in my own time. In addition, I plan to further my knowledge of pre-existing APIs by developing a variety of novel network architectures for a range of data-sets using TensorFlow and other complex APIs that exploit the highly parallelisable nature of the computations involved in ANNs. I also plan to expand my mathematical knowledge, specifically I intend to further my understanding of set theory, map theory and calculus.

References

Anderson, J. A. (1998). *Talking nets: an oral history of neural networks*. Cambridge, Mass: The MIT Press.

Bircanoğlu, C., & Arıca, N. (2018, May). A comparison of activation functions in artificial neural networks. In *2018 26th Signal Processing and Communications Applications Conference (SIU)* (pp. 1-4). IEEE.

Bleicher, A. (2017, August 9). Demystifying the Black Box That Is AI. Retrieved May 5, 2020, from https://www.scientificamerican.com/article/demystifying-the-black-box-that-is-ai/

Brownlee, J. (2019, August 6). How to Configure the Number of Layers and Nodes in a Neural Network. Retrieved May 5, 2020, from https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/

Dar, P. (2019, January 9). 25 Open Datasets for Deep Learning Every Data Scientist Must Work With. Retrieved May 5, 2020, from https://www.analyticsvidhya.com/blog/2018/03/comprehensive-collection-deep-learning-datasets/

Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., ... & Chen, T. (2018). Recent advances in convolutional neural networks. *Pattern Recognition*, *77*, 354-377.

Haoyu, W., & Haili, Z. (2012). Basic design principles in software engineering. In *2012 Fourth International Conference on Computational and Information Sciences* (pp. 1251-1254). IEEE.

Karlik, B., & Olgac, A. V. (2011). Performance analysis of various activation functions in generalized MLP architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, *1*(4), 111-122.

Klabnik, S., & Nichols, C. (2019). *The Rust Programming Language (Covers Rust 2018)*. No Starch Press.

Kolbusz, J., Rozycki, P., & Wilamowski, B. M. (2017, June). The study of architecture MLP with linear neurons in order to eliminate the "vanishing gradient" problem. In *International Conference on Artificial Intelligence and Soft Computing* (pp. 97-106). Springer, Cham.

Milletari, F., Navab, N., & Ahmadi, S. A. (2016, October). V-net: Fully convolutional neural networks for volumetric medical image segmentation. In *2016 Fourth International Conference on 3D Vision (3DV)* (pp. 565-571). IEEE.

Nielsen, M. (2019, December 1). Neural Networks and Deep Learning. Retrieved May 5, 2020, from http://neuralnetworksanddeeplearning.com/chap2.html

Parmar, A. (2019, December 14). Python Pass Example: Pass Statement in Python. Retrieved May 5, 2020, from https://appdividend.com/2019/02/05/python-pass-statement-tutorial-with-example/

Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

Roberts, E. (2017). Applications of Neural Networks. Retrieved May 5, 2020, from https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Applications/index.html

Rosenberg, M., Confessore, N., & Cadwalladr, C. (2018, March 17). How Trump Consultants Exploited the Facebook Data of Millions. Retrieved May 5, 2020, from

https://www.nytimes.com/2018/03/17/us/politics/cambridge-analytica-trump-campaign.html

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Sakryukin, A. (2018, June 8). Under the Hood of Neural Networks. Part 1: Fully Connected. Retrieved May 5, 2020, from https://towardsdatascience.com/under-the-hood-of-neural-networks-part-1-fully-connected-5223b7f78528

Samek, W., Binder, A., Montavon, G., Lapuschkin, S., & Müller, K. R. (2016). Evaluating the visualization of what a deep neural network has learned. *IEEE transactions on neural networks and learning systems*, *28*(11), 2660-2673.

Shyalika, C. (2019, November 16). A Beginners Guide to Q-Learning. Retrieved May 5, 2020, from https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c

Smith, L. (2017, April 4). Cyclical Learning Rates for Training Neural Networks. Retrieved May 5, 2020, from https://arxiv.org/abs/1506.01186

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), 1929-1958.

Venkatachalam, M. (2019, June 22). Recurrent Neural Networks. Retrieved May 5, 2020, from https://towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce

Wang, L., Wang, L., Lu, H., Zhang, P., & Ruan, X. (2016, October). Saliency detection with recurrent fully convolutional networks. In *European conference on computer vision* (pp. 825-841). Springer, Cham.

Link to Project Folder on One Drive

Click here