UPTEC F 16060
Examensarbete 30 hp
Oktober 2016

Classification Performance of Convolutional Neural Networks

Niklas Mattsson

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Classification Performance of Convolutional Neural Networks

Niklas Mattsson

The purpose of this thesis is to determine the performance of convolutional neural networks in classifications per millisecond, not training or accuracy, for the GTX960 and the TegraX1. This is done through varying parameters of the convolutional neural networks and using the Python framework Theano's function profiler to measure the time taken for different networks. The results show that increasing any parameter of the convolutional neural network also increases the time required for the classification of an image. The parameters do not punish the network equally, however. Convolutional layers and their depth have a far bigger negative impact on the network's performance than fully-connected layers and the amount of neurons in them. Additionally, the time needed for training the networks does not appear to correlate with the time needed for classification.

Handledare: Ian Wainwright
Ämnesgranskare: Thomas Schön
Examinator: Tomas Nyberg
ISSN: 1401-5757, UPTEC F 16060

Populärvetenskaplig sammanfattning på svenska

Att skapa smarta maskiner som kan vi kan träna för att lösa uppgifter på egen hand är något som vi människor har försökt göra länge. På senare tid har vi börjat komma närmare och problem som förut fanns och ansågs mycket svåruppnåeliga är nu lösta. Till dessa hör bland annat att program nu kan klassificera bilder, i begränsad skala, med olika innehåll bättre än människor, den som anses vara den bästa spelaren av brädspelet Go har blivit besegrad av ett smart program.

Tekniken som dessa program använder kallas för maskininlärning, och de flesta av dem använder sig av neurala nätverk i flera lager. När neurala nätverk används i flera lager sammankopplade med icke-linjära samband kallas den typen för maskininlärning för "Deep Learning". Det är väldigt tungt beräkningsmässigt att träna dessa Deep Learning-nätverk och det har varit problematiskt. Den senaste tiden har det skett en stor utveckling av grafikkort och dess inbyggda processorer. Det har visat sig att man genom att använda dessa grafikkort för att utföra träningen på nätverken kan vi förkorta ned tiden som krävs med en faktor tio. Denna utveckling har ökat användningen, forskningen och intresset för maskininlärning i allmänhet och deep learning i synnerhet.

Som tidigare nämnt så är ett av de områden som implementationen av denna teknik analysering av klassificering av bilder. Fokus har varit på att få en så hög träffsäkerhet på klassificeringen som möjligt, det vill säga: 'Hur många gånger av 100 bilder olika bilder på katter och hundar kan vi få vårt program att gissa rätt på vilket djur som visas?'. Då detta är under konstant förbättring och utredning har denna studie har istället fokuserat på hur lång tid det tar att klassificera en bild med olika typer av tränade nätverk.

Studien gjordes på svartvita bilder med handskrivna siffror och resultaten visar att vilken typ av nätverk som används och vilken typ av grafikkortsprocessor som används har stor påverkan på tiden det tar att analysera bilder. När någon skapar ett nätverk som ska analysera bilder under tidspress är det alltså av stor betydelse att det testas och byggs upp på ett sådant sätt att det är lämpat för den giva tidsbegränsningen. Det går alltså inte att ha ett nätverk som har rätt gällande om det är en hund eller en katt 95% av gångerna om du vill att det ska kunna analysera bilder väldigt snabbt.

Contents

1 Introduction

1.1 Purpose of the Project

Deep learning with convolutional neural networks (CNNs) has proven to excel at image analysis in environments with sufficient resources. How these solutions perform in environments with limited resources is not fully known and is an area of interest for the practical implementations of CNNs in e.g. hand-held devices and other machines with limited performance.

Information to be gained from this thesis is the time taken to analyse an image depending on the network used and how the parameters in the networks are set. This is of importance for practical implementations due to time-limitations in many fields. In-depth testing of the the training aspect of the network is not within the scope of the project, as the training will be performed in a high-end environment and using well-known CNN architectures.

The information presented in this thesis could be used to help and prepare the implementation of CNNs in embedded systems such as mobile devices, infrared cameras and other similar applications where it is of interest to detect objects in pictures or classify pictures as a whole.

The hypothesis is that increasing the parameters will increase the time it takes to analyse an image. The network parameters will likely not influence the performance of the network equally. For example, convolutional layers are expected to be more expensive than fully-connected layers due to their computational complexity.

1.2 Background

Machine learning is a field of science that has been explored since the early 1950's. Deep learning in the sense of a multiple-level architecture of non-linear functions can be traced back to 1965 [1]. The goal of machine learning and deep learning is to create method with which computers that can learn how to solve problems on their own, or at least with as little human interaction as possible. In image analysis and object recognition the CNN has shown much success. The first modern CNN was developed in 1989 by Yann LeCun. He used a CNN to classify handwritten digits [2]. Lately, machine learning, and deep learning in particular, has taken big leaps forward in terms of accuracy and efficiency, causing a gain in popularity. One major reason behind the performance gains is the improvement in graphics processing units (GPUs), which in some computational tasks can outperform central processing units (CPUs) by an order of magnitude or more. Deep learning has started to gain popularity as a tool that is used in a multitude of other fields of science, such as image analysis, speech recognition and data mining. Recent accomplishments include identifying objects in images with greater accuracy than humans [3] and beating one of the worlds greatest player in the classic board game Go [4].

The amount of branches in machine learning is too extensive to cover them all here and therefore the introduction of the thesis will instead focus on the main types of learning and the applications of them. This master's thesis project will focus on the deep learning branch of machine learning and its performance and power efficiency characteristics.

1.3 Different Techniques for Learning

Supervised, unsupervised and reinforcement learning are the three main techniques of learning for machine learning programs. Each will be introduced below with examples of application areas. It is worth mentioning that the different techniques can be combined, and the descriptions below detail only one of the techniques at a time, in their most basic form. This is to give the reader a general idea of the concepts of the learning process.

1.3.1 Supervised Learning

In supervised learning, the program receives some form of input, e.g. images, sounds or words, as well as the corresponding output, often called label, which tells the training process how the program should classify, or in other ways, learn from the input Examples of labels are 'a cat', the sound of the letter 'K' or 'balloon'. The goal of supervised learning is to produce the correct output given a new input after the training process has been finished. Supervised learning needs a very big amount of classified training data to be able to learn more complex tasks. Some applications of supervised learning are:

Vehicle Control Teaching a program to control a vehicle through supervised learning can be done in different ways and of different magnitudes. An example is a person driving a vehicle on a road while the program analyses the road and the actions the driver takes.

When done correctly a car can learn to how to drive through an off-road environment at high speeds [5].

Image Analysis In image analysis the input is an image and the label is what the image should be classified as. The label can e.g. be a letter, a colour or an animal. A well-trained system should be able to correctly classify an image, even if the object in the image is arranged in an unusual way, such as the cat in Figure 1. Other implementations include facial recognition being able to map out specific features in the face such as the nose, eyes, and mouth [6].

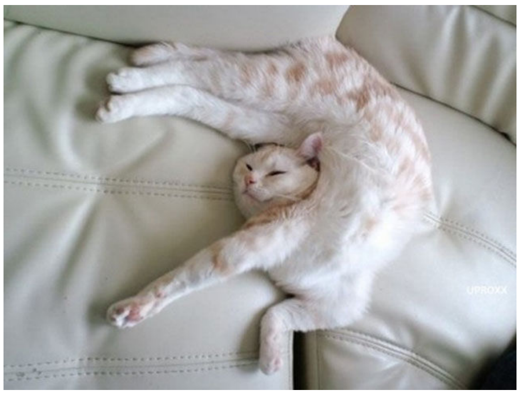

Figure 1: Cat in an odd position.

1.3.2 Reinforcement Learning

With reinforcement learning, the program receives the state it is currently in as an input, performs an action and then receives a reward for doing the performed action in the current state based on how good the results of the action were. The goal of the reinforcement learning is to maximize future rewards. During the training process the program will favour actions that previously resulted in higher rewards given a similar state, and in doing so, a desired behaviour of the program will be achieved. Reinforcement learning requires no data to learn from, as opposed to supervised learning. The input needed in reinforcement learning is instead a function to calculate the reward. The training data which the program then learns from and uses to evaluate its action is then in some sense created by the program itself during the training process. Some applications of reinforcement learning are:

Games In games, the reward function is different depending on the game, but is often increased by simply winning the game and decreased by losing. The moves during the game are then analysed to get an understanding of efficient and inefficient moves. An example of this is a program that is learning to play Super Mario World as in Figure 2. The reward increases as Mario travels to the right as that is the goal of the game. Through reinforcement learning, the program will learn that pressing right on the controller will increase the reward. In the same way it will learn to jump over enemies and conquer other problems it faces [7].

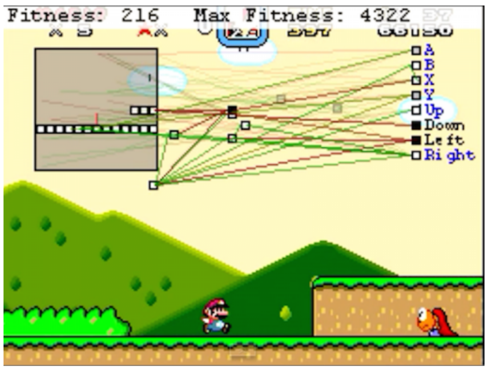

Figure 2: Reinforcement learning program that learns how to play Super Mario World.

Exploration In exploration, the goal is finding the optimal path from start to finish. This could be anything from finding a path through a maze to finding a specific point in an open area. The reward would increase as the program gets closer to the end of the maze, the point in the open area or reach a specific room in a house. In the case of reaching a specific room in a house the exploration algorithm is not only able to find the room, but also through an iterative process find the most efficient way to the room [8].

1.3.3 Unsupervised Learning

During unsupervised learning, the program receives some form of input but obtains no desired output or reward as information. The program is however given a goal to meet, such as sorting the input, finding patterns or matching historical data. Having no control over how the goal is met can be problematic, but the fact that the input data does not have to be pre-processed or labeled will in most cases save a lot of time. Another strength of unsupervised learning is that it is able to categorize information without prior knowledge of what the information contains. Some applications of unsupervised learning are:

Data Mining An example of data mining is using data from the stock market, trying to discover trends and changes over time in order to try to predict future changes of the market [9], [10].

Text Mining Unsupervised learning can be used in a text mining application to sort texts according to the information contained within them without understanding the meaning of the words, or what words are. Finding and sorting information this way can make processes for searching for information easier [11], [12].

1.4 Approaches to Machine Learning

As previously introduced, machine learning programs can use different techniques for learning. There are also a multitude of approaches to applying these learning methods by using different networks. Some of these will be introduced below.

1.4.1 Neural Networks

Neural networks (NNs) are models inspired by the structure and function of biological neural networks. Computations are made in interconnected groups of neurons and can through activation functions represent non-linear relationships. The connections between the neurons are weighted, and the machine learning part is letting a learning algorithm decide the optimal value for those weights instead of having a person testing and tweaking them manually. NNs are commonly used for regression and classification problems, but can be altered in a multitude of ways and can therefore be used in almost all types of machine learning problems.

1.4.2 Deep Learning

Deep learning is the machine learning approach that will be implemented in this project and will therefore get a more complete introduction below.

1.5 Deep Learning in Depth

Deep learning is a branch of machine learning that attempts to model complex abstractions in data by using a multiple-level architecture most commonly of NNs, and non-linear transformations in its algorithms. Using these techniques the goal is to reach a 'true' artificial intelligence, in the sense that the machine can learn how to perform very complex tasks in a similar way to how a human does. Building and training these systems is very computationally heavy, but recent advances using GPU-based implementations have increased the success and popularity of deep learning. Two of the most common approaches to deep learning are recurrent NNs and CNNs. CNNs will be used in this project, but both will be briefly introduced below.

1.5.1 Recurrent Neural Networks

Recurrent neural networks (RNNs) are used to handle tasks that involve sequential inputs such as speech and language, i.e. data with a temporal aspect [13]. RNNs handle an input sequence one element at a time while maintaining a state vector that contains information about the past elements of the sequence. This is useful for sequential inputs since knowing all the different states at different times has a large impact on how to interpret the data. Training RNNs can be difficult, but once trained they are powerful dynamic systems.

1.5.2 Convolutional Neural Networks

Convolutional neural networks have been proven to beat humans in object recognition in images [3] and with extensive amount of data and the right CNN, a machine can reach as much as 99% accuracy in recognizing objects in images [14]. This project will focus on deep learning from an image recognition perspective and will therefore use CNN, which will be described in detail in Section 2.

2 Theory

2.1 Data Preprocessing

Before using data for training, it is preferable to preprocess it to make it easier to analyse by making it more homogeneous. There are several different methods of data preprocessing and choosing a set of methods in order to obtain the best result is a topic in and of itself. In this project, mean subtraction and normalisation will be used.

Mean subtraction involves subtracting the mean across every individual feature in the data and can be geometrically interpreted as centring the cloud of data around the origin along every dimension. To get the mean, $\hat{\mu}$, of the data matrix, 'X', (1) is used.

$$\hat{\mu} = \frac{\sum_{i=1}^{N} X_i}{N} \tag{1}$$

Normalisation is used to adjust the data so that it is approximately of the same scale. It is done by dividing the data in each dimension by its standard deviation. The standard deviation, $\hat{\sigma}$, is calculated as shown in (2).

$$\hat{\sigma} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(X_i - \hat{\mu})^2} \tag{2}$$

Mean subtraction and normalisation are then applied as shown in (3).

$$X_{ut} = \frac{X_{in} - \hat{\mu}}{\hat{\sigma}} \tag{3}$$

2.2 Convolutional Neural Network

A CNN is composed of different parts that together form the network. Each part will be discussed in detail, but the basic usage of CNNs is on the following form:
Preprocessed image $\rightarrow$ $(Conv \rightarrow ReLU \rightarrow Pool) * M \rightarrow (FC \rightarrow ReLU) * K \rightarrow FC \rightarrow Classification$

Where:

- Conv - Convolution Layer

- ReLU - Rectified Linear Unit
- Pool - Pooling Layer
- FC - Fully-Connected Layer
- M and K are numbers representing the number of times each operation is performed.

First, a convolutional layer (ConvLayer) is applied to the image. The ConvLayer performs convolutions in smaller areas in order to extract data from neighbouring pixels. Secondly, the rectified linear unit (ReLU) is applied to the output of the ConvLayer to extract linear and non-linear relations in the data. Lastly, pooling is applied to the output from the ReLU to extract the most meaningful information. This sequence can be repeated as many times as desired and as allowed by the size of the input image, as the number of pixels gets reduced in the pooling layer as well as in the ConvLayer, depending on the type of padding used. An example of the reduction of the image's size between the layers can be seen in Figure 3. The 'shrinkning of the data size' is indicative of spatial information being extracted as larger sections of the initial image are allowed to influence successively fewer intermediate data points. The picture shows that the sequence above is followed by fully-connected layers, which rasterize the data into a 2D matrix used to produce an output in the form of an vector. This vector is then used to produce a probability vector for the input image, using a Softmax Classifier.

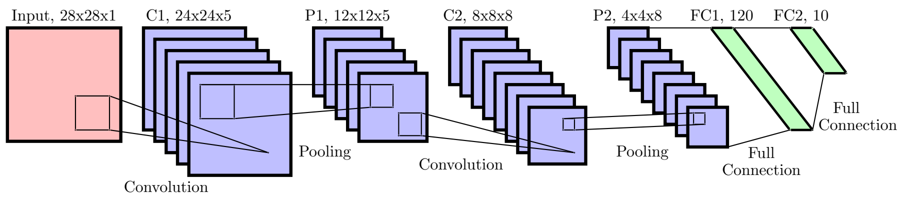

Figure 3: CNN with M = 2 and K = 1. This type of architecture is called "LeNet".

2.2.1 Convolutional Layer

For analysing images, ConvLayers have proven to be very successful in achieving good results. They utilise the fact that an image is a high-dimensional input consisting of smaller features that together form the image. To extract the smaller features, a 'receptive field' is used. The receptive field is a small matrix, with the dimensions [n x n x depth], which is applied across the entire previous layer using the same weights. After each movement the receptive field performs a logistic regression operation and outputs a pre-defined amount of neurons in the depth-direction of the network. These neurons are connected to the receptive field output, but not any other area of the image. In Figure 4, a 5x5x1 receptive field in the red layer is shown as a smaller darker red area outputting five neurons to the blue layer. The receptive field is then applied over the entire red area, outputting neurons at every point which then form the blue layer. How many pixels the receptive field moves between each operation is called 'stride'. With a stride of one, the blue layer will have approximately the same height and width as the red layer. With a stride of two, it will be approximately half.

'Approximately' is used above with regards to the size, as the actual size depends on what type of padding is used along the edges. 'Zero-padding', which means filling out the area around the edges with zeroes, results in the exactly same width and height for the following layer, while 'valid-padding' results in both the width and height being decreased.

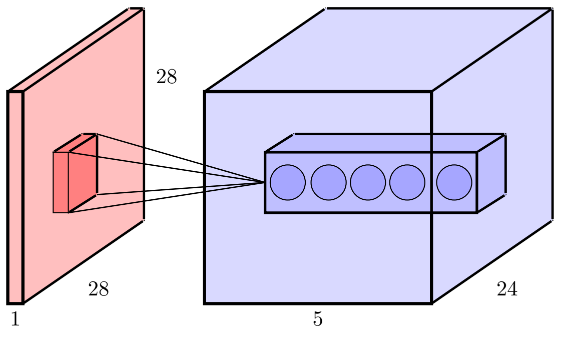

Figure 4: Example of a convolutional layer operation. The red layer contains one feature maps and the blue layer contains five feature maps.

2.2.2 Rectified Linear Unit

The rectified linear unit (ReLU) can be regarded as an 'activation function' for the network's neurons. ReLU is applied on the output of a neuron as shown in (4). The output of the ReLU-function is equal to the input, x, if the input is positive and zero if the input is negative. This non-linear function is necessary for the network to be able to represent non-linear relationships between neurons.

$$f(x) = max(0, x) \tag{4}$$

2.2.3 Pooling

Pooling is a technique that is used to extract the most useful information from an area in a convolutional layer and thus reducing the amount of information to be analysed in the next step. The input area of the pooling layer is called filter layer and is the pooling layer's equivalent of the ConvLayer's receptive field. The stride parameter is present in pooling as well, and works in the same way as in the ConvLayer, but does not have to take the same value. There are different kinds of pooling, but the variant that has been shown to be the most successful is max pooling, and it will therefore be used in this project. An example of how max pooling can be used is shown in Figure 5, where a stride and filter size of 2x2 is used.

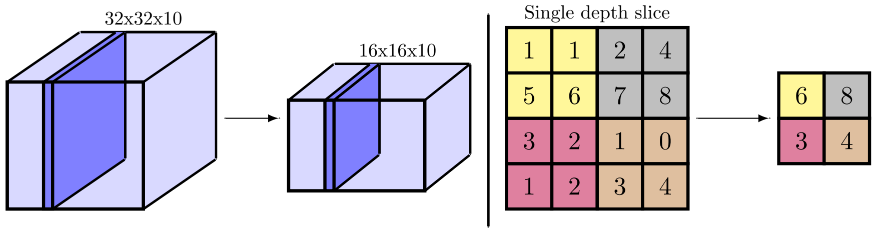

Figure 5: Example of pooling with a stride and filter size of two.

2.2.4 Fully-Connected Layer

A fully-connected layer (FC Layer) is a type of NN where all neurons in adjacent layers are fully pairwise connected, but neurons in the same layer share no connection. An example of two such fully-connected layers is shown in Figure 6.

The input layer's size is usually the multiplicative product of the image dimensions: width, height and depth, with depth being the number of colour channels. The number of neurons in the hidden layers are hyper parameters that the creator of the CNN sets at the start after taking the types of objects to classify and data to analyse into account. The number of neurons in the output layer is equal to the number of classes that the network should be able to identify.

The number of weights and biases are directly related to the amount of neurons in the adjacent layers. The NN to the left in Figure 6 has a total of [3x4] + [4x2] = 20 weights arranged in two weight matrices and 4 + 2 = 6 biases arranged in two bias vectors while the NN to the right in the same picture has [3x4] + [4x4] + [4x2] = 36 weights arranged in three weight matrices and 4 + 4 + 2 = 10 biases arranged in three bias vectors.

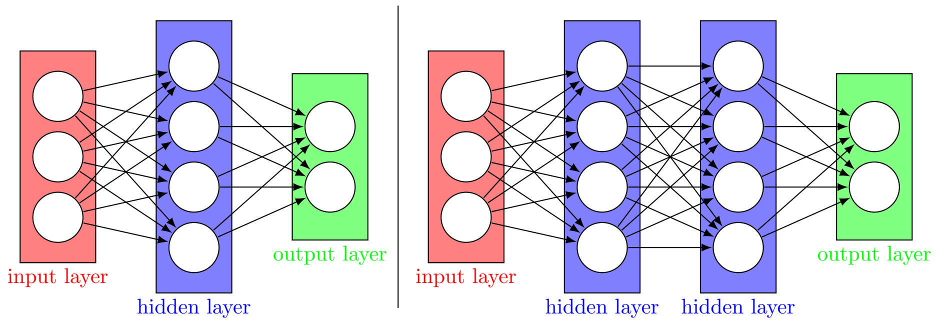

Figure 6: Example of two fully-connected networks - to the left with one hidden layer and to the right with two hidden layers. Circles represent neurons and arrows represent weighted connections, between neurons.

2.3 Training the Convolutional Neural Network

2.3.1 Setting up the Data

When training the CNN, the dataset has to be split into three parts: the training set, the validation set and the test set. The training set is used to train the CNN using logistic regression, which will be explained below. After each round of training, the network is tested on a random part of the validation set. The result is used to evaluate the network's accuracy in detecting the correct objects, in order to determine if weights are being adjusted in the best way. The test set is used to evaluate the performance of the CNN after the training process has finished. It can, however, be dangerous to alter the network based on the results using the test set, due to the risk of overfitting. The exact size of the different sets vary depending on the dataset given, but usually the training set is composed of approximately 75-80% of the total data, while the validation and test sets both use half of the remaining data, approximately 10-12.5% each [15].

2.3.2 Logistic Regression

Figure 7 shows the different parts involved in the logistic regression for finding the weight values for the connections between the neurons in a CNN. The different parts will be discussed in detail below but briefly introduced here. x_i is the i:th input image, with its data arranged in a vector. With the help of a linear classifier, $Wx_i + b$, a 'logit' vector is produced from the input. That logit vector is then input into the Softmax classifier which produces a vector of numbers between zero and one that sum to one. This vector can be interpreted as a probability vector containing the probabilities of the object belonging to each class. The result of the Softmax classifier is used in the Cross-Entropy loss function with the one-hot labels. A one-hot encoded vector is a vector with a value of one for the correct class and zero for the others. The average Cross-Entropy loss is then minimized to train the network. To minimize the loss, the weights are adjusted through stochastic gradient descent as they get closer to the minimum loss. α is the length of the step taken between weight estimate iterations. In Figure 7, we see the loss function of a neural network with two weights get minimised through stochastic gradient descent.

Linear Classifier The linear classifier (LC) described in (5) outputs a score called 'logit' from an image data vector, x_i, a weight matrix, W, and a bias vector, b. The weights are initialized as small random values from a normal distribution, and the biases are initialized as zero. Optimising these weights and biases is the main component of the training of the network.

$$f(x, W, b) = Wx_i + b = Y \tag{5}$$

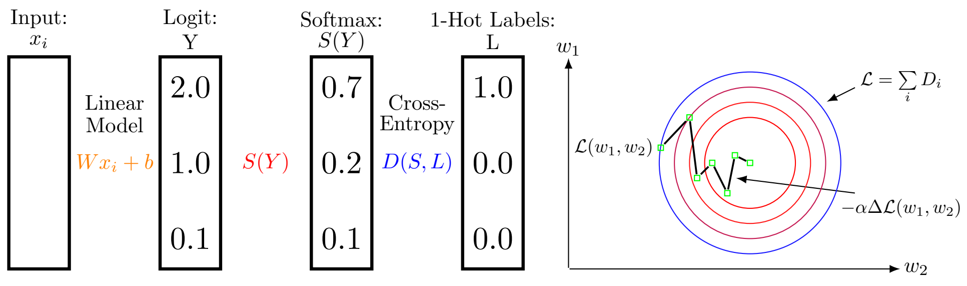

Figure 7: Logistic regression calculation steps.

(5) shows a FC Layer with with zero hidden layers. In comparison, the equation for the FC layer to the left in Figure 6 would look like (6). This equation becomes increasingly nested as more layers are used.

$$f(x, W_1, W_2, b_1, b_2) = W_2(W_1x_i + b_1) + b_2 = Y \tag{6}$$

Cross-Entropy Loss Function The cross-entropy loss uses the Softmax function shown in (7). The function takes the logit, y_i, from the LC and 'squashes' the vector's sum to one, using gradient-log-normalization. The resulting values can be interpreted as probabilities. y_j is the score corresponding to class j. They are not set probabilities, however. The probabilities can be altered by multiplying or dividing the exponents. Multiplication moves the probabilities closer to zero or one, making the classifier more confident, while division moves the probabilities closer to uniform distribution, making the classifier less confident.

$$S(y_j) = \frac{e^{y_j}}{\sum_k e^{y_k}} \tag{7}$$

The cross-entropy loss is calculated as shown in (8). L_j is the one-hot encoded vector for the j:th class. S_j is the estimated probability of the picture being of class j.

$$D(S, L) = -\sum_j L_j Log(S_j) \tag{8}$$

To get the training loss over the entire training set, $\mathcal{L}$, we average the cross entropy across all images 'i', as shown in (9).

$$\mathcal{L} = \frac{1}{N}\sum_i D(S(Wx_i + b), L) \tag{9}$$

Gradient Descent Gradient descent is used to minimize the loss function i.e. improve the accuracy of the network. One iteration is shown in (10). It takes the gradient of the loss function from (9) with respect to every single weight evaluated at the current weight estimate, and then multiplies that result with a learning rate, or step, α. α is a parameter that decides how heavily the calculated gradient influences the adjustment of the weights for the next iteration.

$$\mathcal{L} \rightarrow \mathcal{L} - \alpha \Delta \mathcal{L}(w_1, w_2, ..., w_n) \tag{10}$$

A common analogy is the 'hiker's analogy'. Imagine a blindfolded hiker trying to reach the bottom of a hilly area. The slope's direction that the hiker feels with his feet is the calculated gradient, and the length of the step he takes after feeling the steepest direction of the slope is decided by α.

A variant of the gradient descent is the Stochastic Gradient Descent (SGD). This technique is used to lessen the amount of time it takes to train the network. This is achieved by randomly selecting small batches from the training set during training, instead of using the entire training set.

After having reached an approximate minimum using a batch, the loss function is updated using (10). Following the update, a new batch is chosen and the process is repeated. The batch, being small and random, could give the wrong direction in which to minimize the function, but it has been shown that the overall training time is heavily reduced by using this technique.

Backpropagation Due to this thesis not examining the training of CNNs in any great depth, the description of backpropagation will be kept short. Backpropagation is a technique used when evaluating connections between neurons. Each neuron can completely independently, when it receives an input, calculate two things - its output value and the local gradient of the input, considering the output value. That process is called the forward pass. When that process is over, backpropagation begins. During backpropagation, each neuron learns the gradient of its output value with regards to the entire network. Using the chain rule, the neuron takes the gradient of the entire network and multiples it with each gradient the neuron has a connection with. It can be thought of as the neurons communicating with one another using the gradient signal, and deciding if, and how much, they should increase or decrease their output.

Regularization Regularization techniques are used to prevent the model from overfitting to the training data. Overfitting a model during the training process results in poor performance on unseen data due to the network having started to 'memorize' the training data rather than 'learning' from it to generalize a trend. The three techniques that will be used for regularization in this project are early termination, L2-regularization, and dropout, all further detailed below.

Early termination Early termination is a regularization technique that uses the validation set performance as a test to avoid overfitting. The way it does this by stopping the training process whenever the performance of the network on the validation set stops improving.

L2-Regularizaton L2-regularization is shown in (11) and uses the L2-norm on the weight matrix to penalize large weights so that no single large weight can have a large influence on the score. The tends to lead to improved generalization and prevention of overfitting. The drawback of this technique is that the hyper parameter β must be tuned in some way, often through potentially time-consuming cross-validation.

$$\mathcal{L}' = \mathcal{L} + \beta \frac{1}{2} ||W||_2^2 \tag{11}$$

Dropout When using dropout during training, a parameter is set to a value between 0 and 1. For every neuron that is about to be used, a random value is generated. If this value is below the parameter's value, that neuron does not get activated. The output of the activated neurons is multiplied by an appropriate factor. For instance, if the dropout parameter is set to 0.5, the results get multiplied by two. By doing this, the network can never rely on a single activated neuron, and is instead forced to learn redundant representations. This makes the network more robust, prevents overfitting, and makes the network act as if it is taking the consensus of an ensemble of networks. When evaluating the trained network after having used dropout, the average of the trained values is used to get the consensus information of the network.

3 Experiments and Results

3.1 Experiments

The method for determining the number of classifications per millisecond used in this project consists of three main steps. The first step is building the different networks, the second step is training, the networks and the third step is measuring the time taken to perform classifications of the networks once they are fully trained. All the steps are done with the mNIST-dataset, shown in Figure 8, which is composed of pictures of numbers in grey scale of size 28x28 pixels, and the graphic card Nvidia Geforce GTX 960 (GTX960) [16] or TegraX1 [17], which is a system on a chip (SoC), is used. For the purpose of this project the mNIST-images will be called 'tiles' as multiple of tiles, of e.g. 28x28 pixels, would be used in larger real-world cases. The networks are built using Theano [18] which uses NVIDIA CUDA [19] and the NVIDIA CUDA Deep Neural Network library (cuDNN) [20].

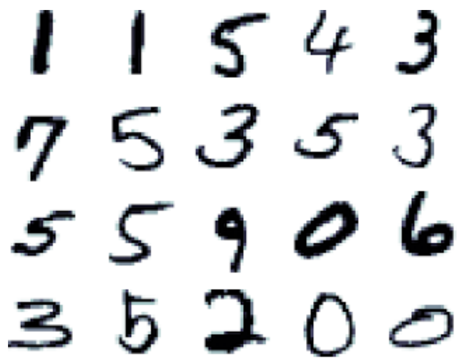

Figure 8: Example numbers from the mNIST-dataset.

The networks were varied in five different ways; the number of convolutional layers (ConvLayers), the depth in the ConvLayers, the number of Fully-Connected layers (FC-layers), the number of neurons in the FC-layers, and the size of the batch, the last meaning the number of tiles that are classified as a batch.

The different networks were grouped together in four different groups for easier and more relevant comparison between one another. The different groups are:

- Group 1: Varying the number of FC-layers and the batch size.
- Group 2: Varying the number of ConvLayers and the batch

size.

- Group 3: Varying the number of FC-layers and their number of neurons.
- Group 4: Varying the number of ConvLayers and their depth.

When varying the number of FC-layers, the number of ConvLayers is always kept at one. When varying the number of ConvLayers the number of FC-layers is always kept at one. When varying the batch size the values 16, 32, 64, ..., 4096 are used. When varying the number of neurons the values 128, 256, 512, 1024 and 2048 are used. Initially the values 100, 200, 400, 800 and 1600 were used for the neurons, but this turned out to be a poor choice as can be seen in section 4. When varying the depth, the values 5, 10, 20, 40 and 80 are used.

The building and training of the networks was done using the framework Theano, version 0.8, which uses the programming language Python. Building the networks consists of programming how each layer should behave and what characteristics they should have.

The ConvLayers characteristics are decided by the parameters input and output depth, the path size, the stride of the patch as well as which padding that is used at the edges. The size of the receptive field and the stride of the receptive field used are both five, and valid padding is used. The input depth to the first ConvLayer will be one for the mNIST-dataset and the depth variable for all other ConvLayers. The output depth parameter is varied in the networks of test group 4, while set to 20 in all other groups as mentioned above. The size of the receptive field is 5x5 and its' stride is five, and with valid padding this results in the output dimensions of a picture being four less in width and height, e.g. an input picture of dimensions 28x28 is outputted as a picture of dimensions 24x24.

The PoolLayers characteristics are decided by the parameters patch size and the stride of the patch. In this project the patch size and the stride of the patch will be set to two at the last ConvLayer and one at all the previous.

The FC-Layers characteristics are decided by the number of input and output neurons. The amount of input neurons in the first

FC-layer after the last ConvLayer is the product of the depth, the width and the height of that layer. The number of output neurons in the first FC-layer and the number of input and output neurons in the rest of the FC-layers are equal to the number of neurons that is set for the specific test group that is being tested.

The network is trained for 60 epochs with a learning rate of 0.03. After the training is complete the network is saved with cPickle to a pkl-file.

The performance analysis is done by activating the profiler in the Theano function that performs the classification. The code executing for the classification is shown in the code block below.

```
CUDA_LAUNCH_BLOCKING=1 python ./mNISTXConvYFC.py Z
```

Where 'CUDA_LAUNCH_BLOCKING=1' is used to turn off asynchronous execution to get meaningful profiling results. X is the number of ConvLayers, Y is the number of FC-layers and Z is the batch size. The python file that is executed loads the network and then uses the classification method on the loaded network 20 times. An example of the code being executed with numbers is listed below:

```
echo "First loop"
CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv1FC.py 16

CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv2FC.py 16

CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv3FC.py 16

CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv4FC.py 16

CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv5FC.py 16

echo "Second loop"
CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv1FC.py 16

CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv2FC.py 16

CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv3FC.py 16

CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv4FC.py 16

CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv5FC.py 16
```

```
echo "Third␣loop"
CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv1FC.py 16

CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv2FC.py 16

CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv3FC.py 16

CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv4FC.py 16

CUDA_LAUNCH_BLOCKING=1 python ./mNIST1Conv5FC.py 16
```

The above code snippet tests the performance of part of 'Group 1' of the networks. After running the code above the computer is restarted and '16' is changed to '32' in the code. After having done that the computer is restarted again and '32' is changed to '64' and so on until 'Group 1' is done. The rest of the groups are tested in the exact same way but with varying different numbers according to the group of networks they belong to. The tests were performed in the console with the service lightdm disabled to remove any impact from any graphics activities on the test machine. Initially the performance tests were done in one large bulk looping through all the networks in succession, this resulted in problems which can be seen in Table 1 and read more about in section 4.

To get the performance in tiles/ms the median of the three loops is used and that time is then used as 'total time for 20 tests in seconds' in (12).

$$Tiles/ms = \frac{20 * \text{batch size}}{1000 * \text{total time for 20 tests in seconds}} \tag{12}$$

Comparing the performance for different networks when increasing and decreasing the different parameters results in graphs and an equation which calculates the performance.

3.2 Results

In all graphs below, the y-axis label 'Tiles/ms' is used due to the fact that the images classified by the network are of the size 28x28 pixels. An image of that size could be considered a tile of a larger image. Figure 9 shows an example of a larger image being analysed by smaller image tiles.

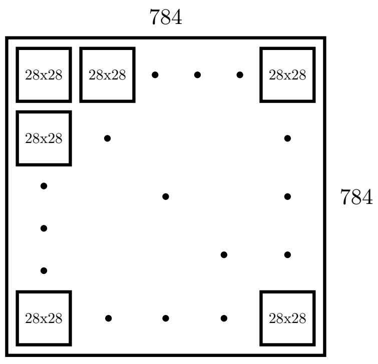

Figure 9: An image of pixel size 784x784 being analysed by using smaller tiles of pixel size 28x28.

Figure 10 shows that the performance when varying FC-layers and batch size for the GTX960.

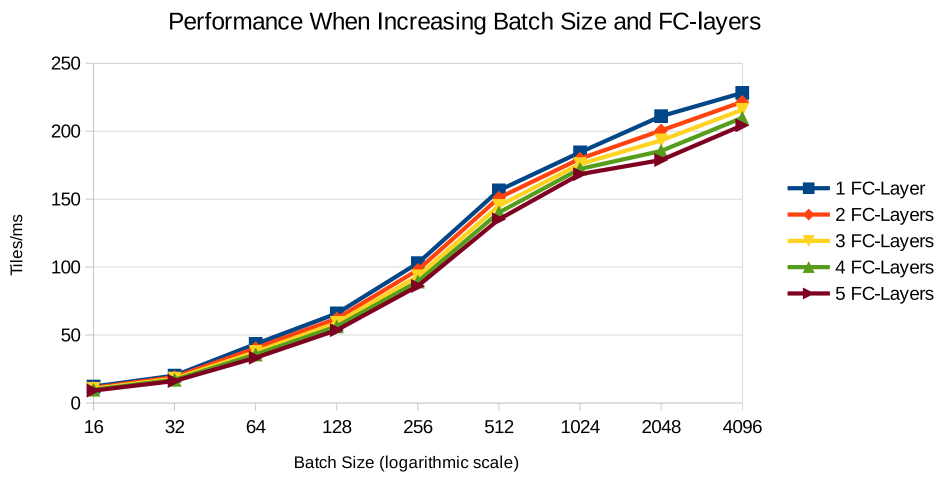

Figure 10: Performance when varying batch size and number of FC-layers. The number of ConvLayers is one.

Figure 11 shows that the performance when varying the number of ConvLayers and batch size for the GTX960.

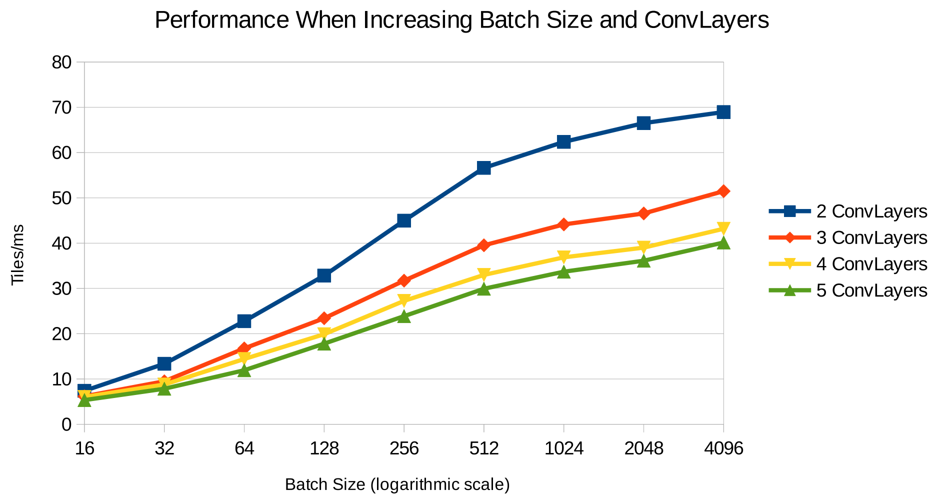

Figure 11: Performance when varying batch size and number of ConvLayers. The number of FC-layers is one.

Figure 12 shows that the performance when varying the number of FC-layers and neurons for the GTX960.

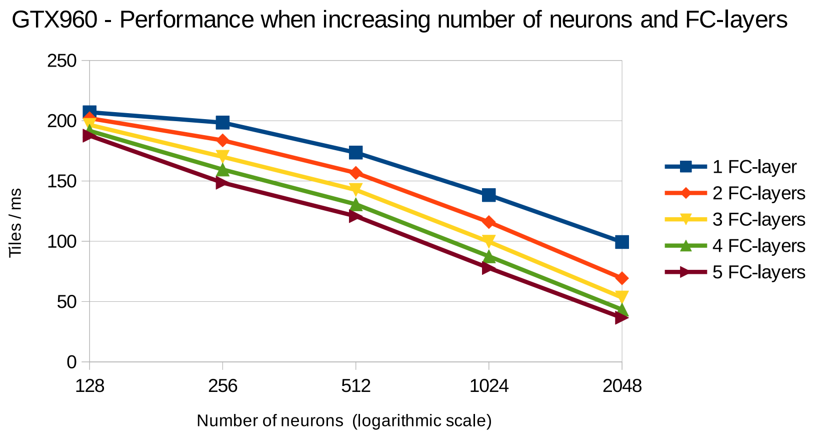

Figure 12: Performance when varying number of neurons and FC-layers. The number of ConvLayers is one.

Figure 13 shows that the performance when varying the number of FC-layers and neurons for the TegraX1.

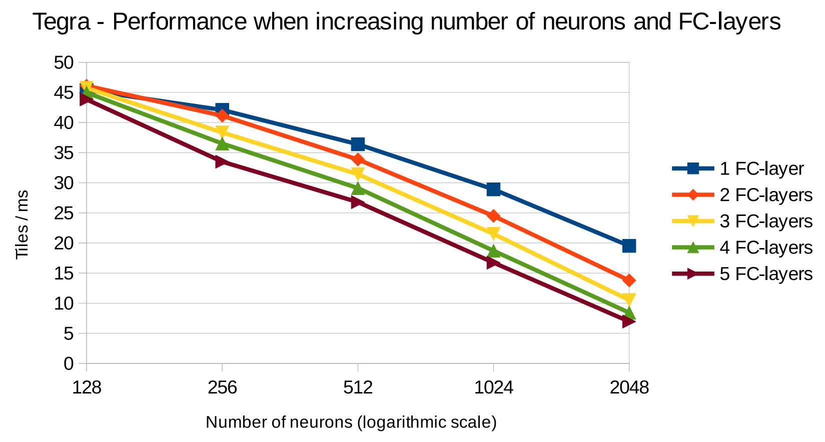

Figure 13: Performance when varying number of FC-layers and neurons. The number of ConvLayers is one.

Figure 14 shows the speedup factor when comparing the GTX960 with the TegraX1 when varying the number of FC-layers and neurons.

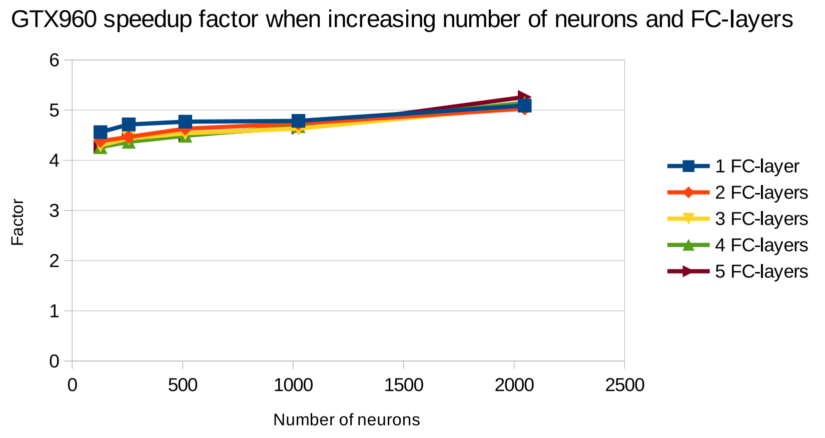

Figure 14: How many times faster the classifications are on the GTX960 when varying the number of FC-layers and neurons. The number of ConvLayers is one.

Figure 15 shows that the performance when varying the number of ConvLayers and their depth for the GTX960.

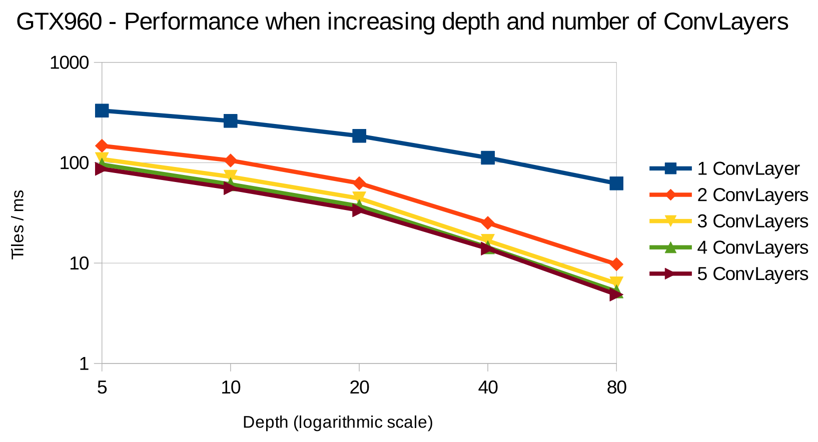

Figure 15: Performance when varying depth and number of ConvLayers. The number of FC-layers is one.

Figure 16 shows that the performance when varying the number of ConvLayers and their depth for the TegraX1.

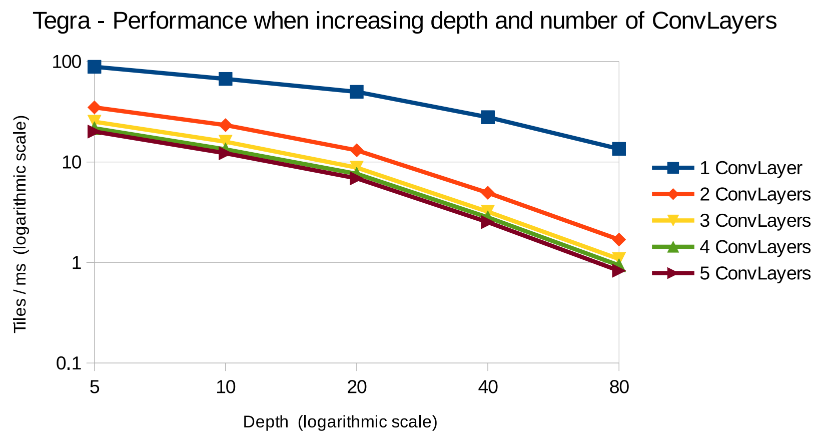

Figure 16: Performance when varying depth and number of ConvLayers. The number of FC-layers is one.

Figure 17 shows the speedup factor when comparing the GTX960 with the TegraX1 when varying the number of ConvLayers and their depth.

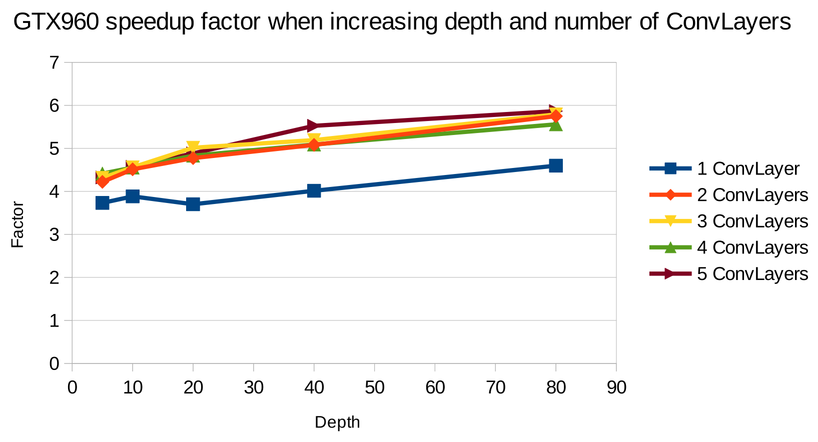

Figure 17: How many times faster the classifications are on the GTX960 when varying the number of ConvLayers and their depth. The number of FC-layers is one.

The figures above show how the performance varies when changing the number of different layers and their properties. To make a more in-depth analysis it is possible to look at profiler to see which individual operations that are taking the majority of the time when making classifications. These operations are GpuAlloc, GpuDot22, GpuDnnPool, GpuDnnConv, and GpuElemwise.

GpuAlloc is the time taken to allocate the memory in the GPU and varies with number of ConvLayers and their depth. The variation in time needed for this process depending on this is for the GTX960 shown in Figure 18.

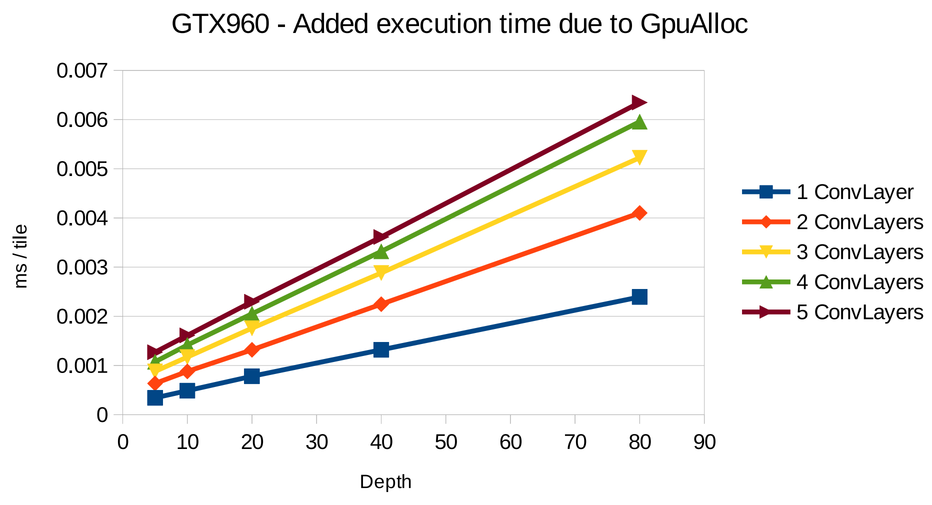

Figure 18: Time taken per ms for the GpuAlloc-process when varying number of ConvLayers and their depth.

The variation in time needed for the GpuAlloc-process for the TegraX1 shown in Figure 19.

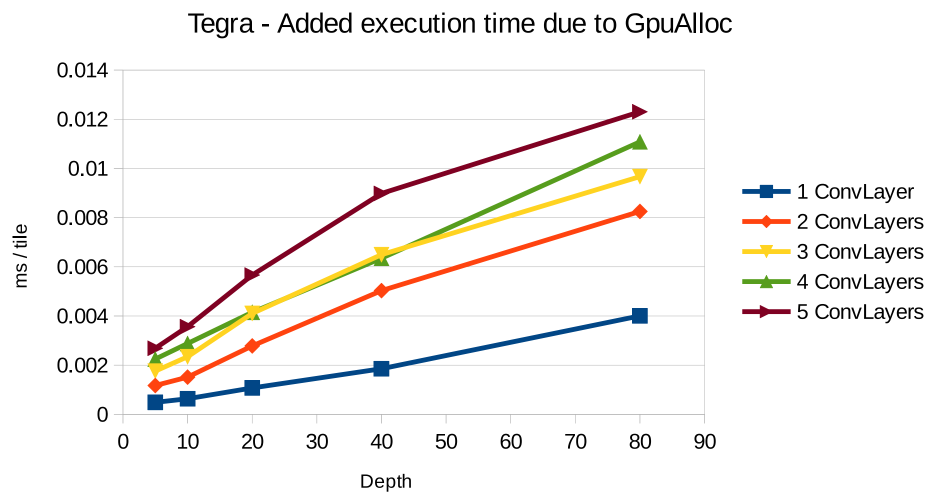

Figure 19: Time taken per ms for the GpuAlloc-process when varying number of ConvLayers and their depth.

Figure 20 shows the speedup factor when comparing the GTX960 with the TegraX1 for the GpuAlloc-operation.

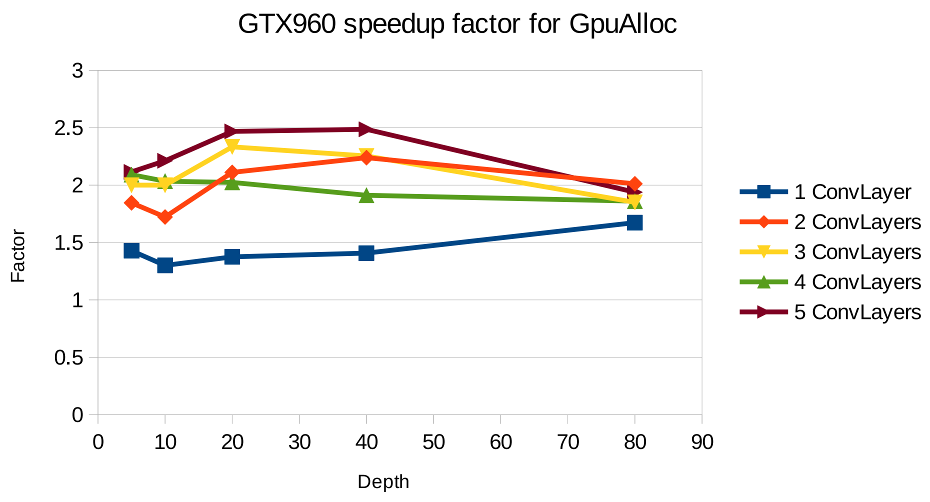

Figure 20: How many times faster the classifications are on the GTX960 when performing the GpuAlloc-operation when varying the number of ConvLayers and their depth.

GpuDot22 is the time taken for the product operations in the FC-layers. And it varies with the number of FC-layers and their number of neurons. The variation in time needed for this process depending on this is for the GTX960 shown in Figure 21.

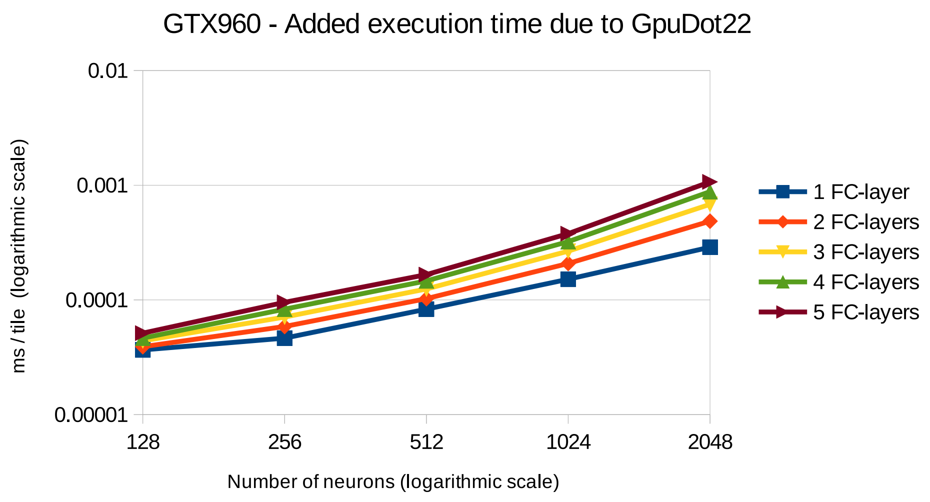

Figure 21: Time taken per ms for the GpuDot22-process when varying number of FC-layers and their number of neurons.

The variation in time needed for the GpuDot22-process for the

TegraX1 shown in Figure 22.

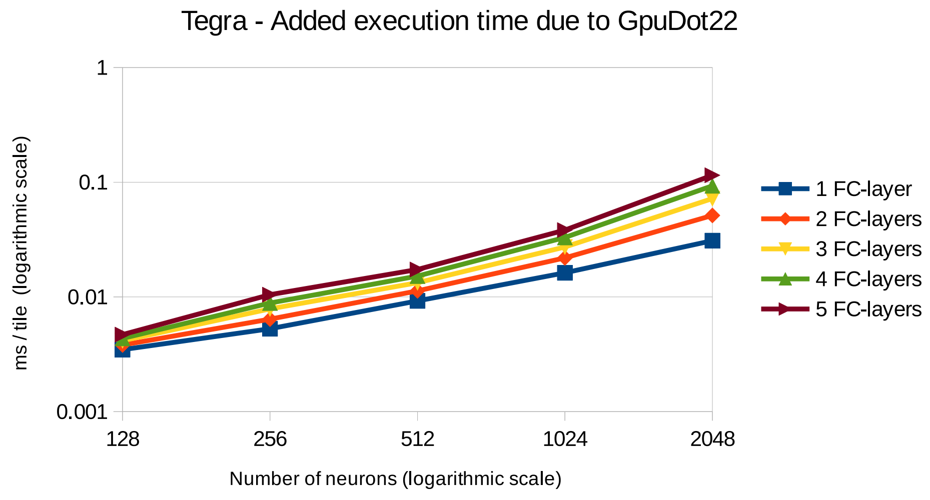

Figure 22: Time taken per ms for the GpuDot22-process when varying number of FC-layers and their number of neurons.

Figure 23 shows the speedup factor when comparing the GTX960 with the TegraX1 for the GpuDot22-operation.

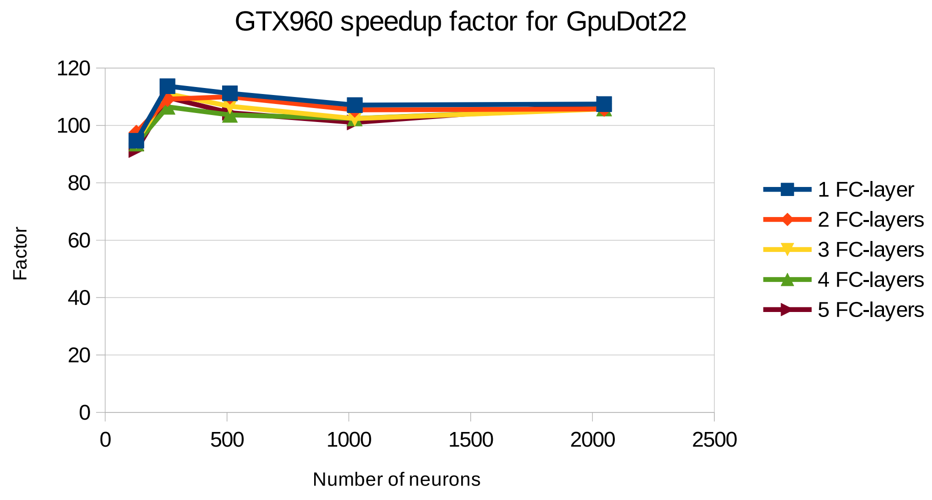

Figure 23: Time taken per ms for the GpuDot22-process when varying number of FC-layers and their number of neurons.

GpuDnnPool is the time taken for the pooling operation. It varies with the number of ConvLayers and their depth. The variation in time needed depending on this is shown in Figure 24.

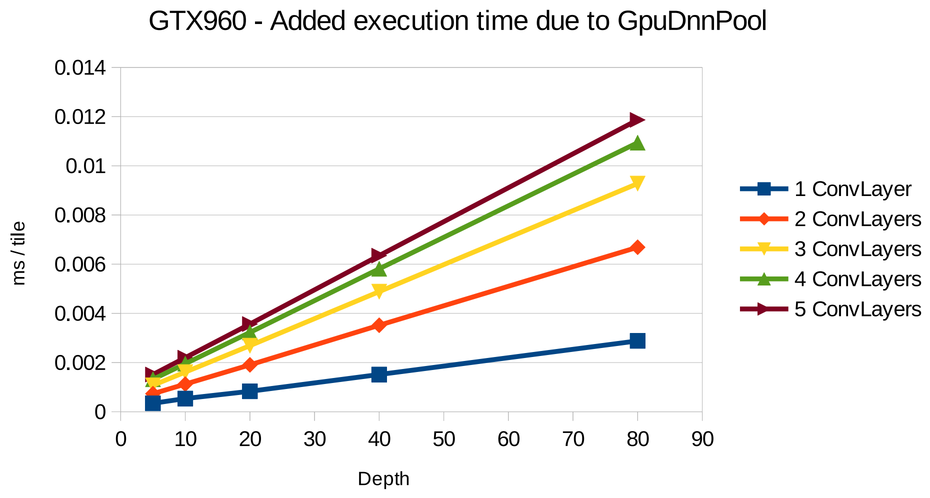

Figure 24: Time taken per ms for the GpuDnnPool-process when varying number of ConvLayers and their depth.

The variation in time needed for the GpuDnnPool-process for the TegraX1 shown in Figure 25.

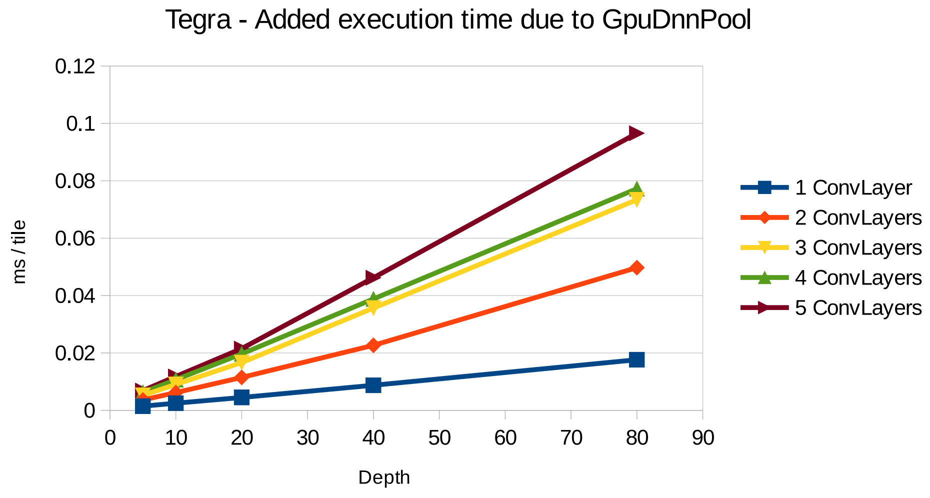

Figure 25: Time taken per ms for the GpuDnnPool-process when varying number of ConvLayers and their depth.

Figure 26 shows the speedup factor when comparing the GTX960 with the TegraX1 for the DpuDnnPool-operation.

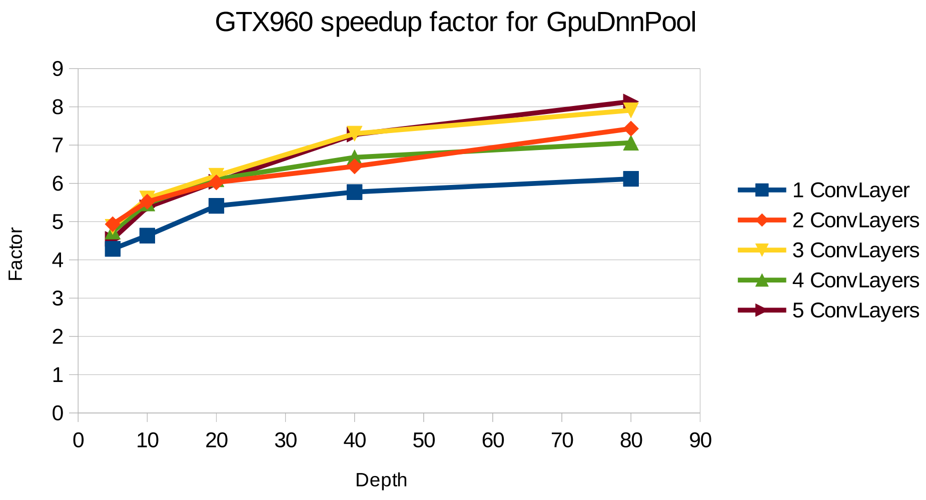

Figure 26: Time taken per ms for the GpuDnnPool-process when varying number of ConvLayers and their depth.

GpuDnnConv is the time taken for the convolution operation. It varies with the number of ConvLayers and their depth. The variation in time needed for this process depending on this is for the GTX960 shown in Figure 27.

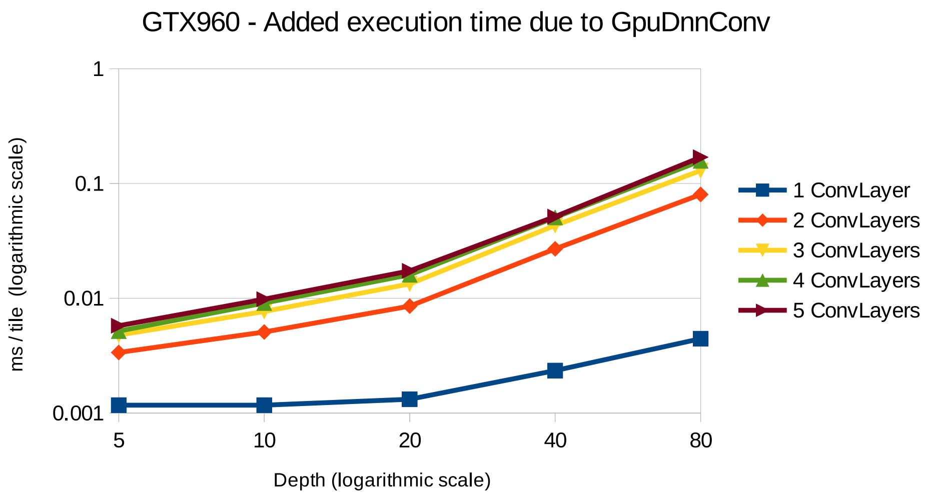

Figure 27: Time taken per ms for the GpuDnnConv-process when varying number of ConvLayers and their depth.

The variation in time needed for the GpuDnnConv-process for the TegraX1 shown in Figure 28.

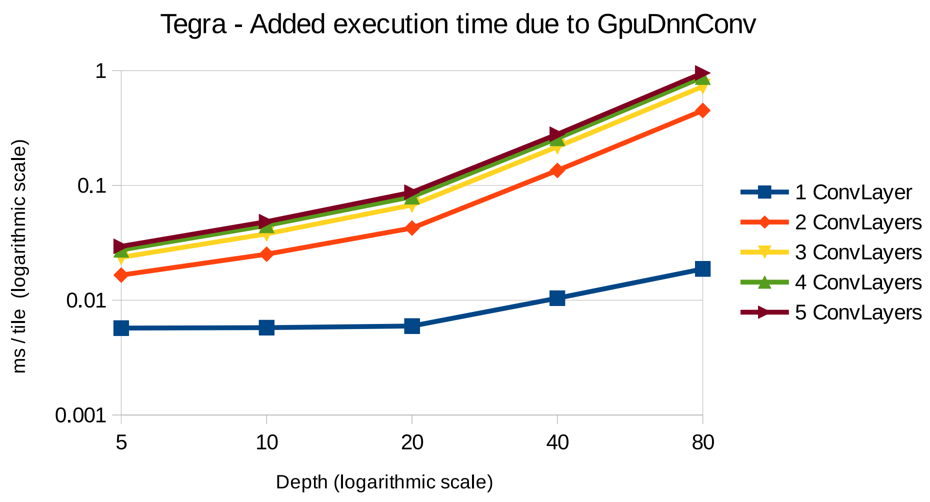

Figure 28: Time taken per ms for the GpuDnnConv-process when varying number of ConvLayers and their depth.

Figure 29 shows the speedup factor when comparing the GTX960 with the TegraX1 for the GpuDnnConv-operation.

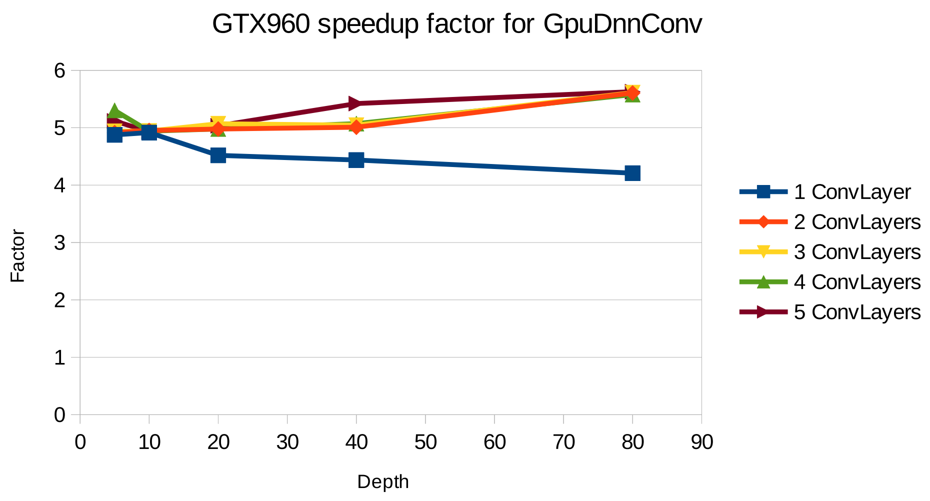

Figure 29: Time taken per ms for the GpuDnnConv-process when varying number of ConvLayers and their depth.

Elemwise is the time taken for every elemental operation. It varies with the number of ConvLayers and their depth, as well as the number of FC-layers and their number of neurons. The variation in time needed for this process depending on this is for the GTX960 shown in Figure 30.

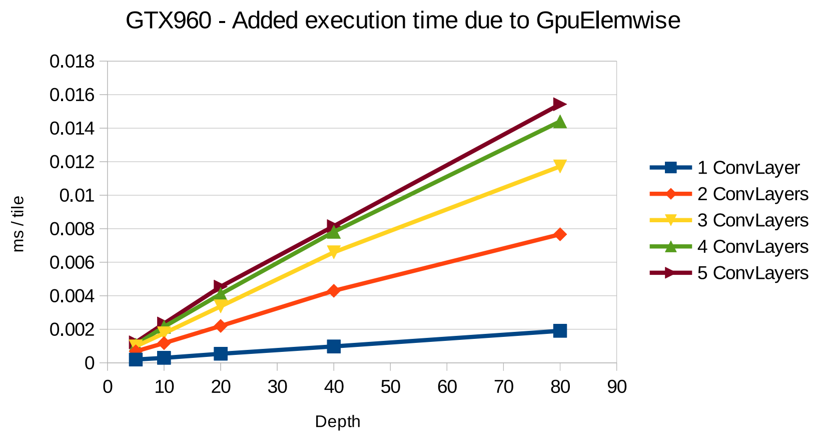

Figure 30: Time taken per ms for the GpuElemwise-process when varying number of ConvLayers and their depth.

The variation in time needed for the GpuElemwise-process for the TegraX1 shown in Figure 31.

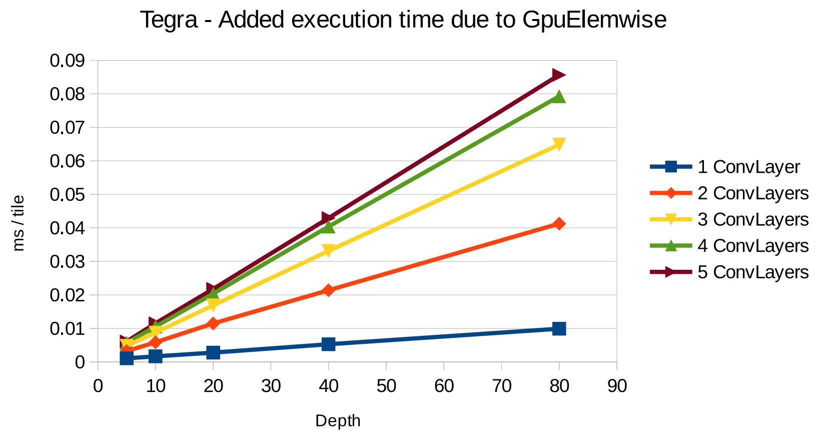

Figure 31: Time taken per ms for the GpuElemwise-process when varying number of ConvLayers and their depth.

Figure 32 shows the speedup factor when comparing the GTX960 with the TegraX1 for the GpuElemwise-operation.

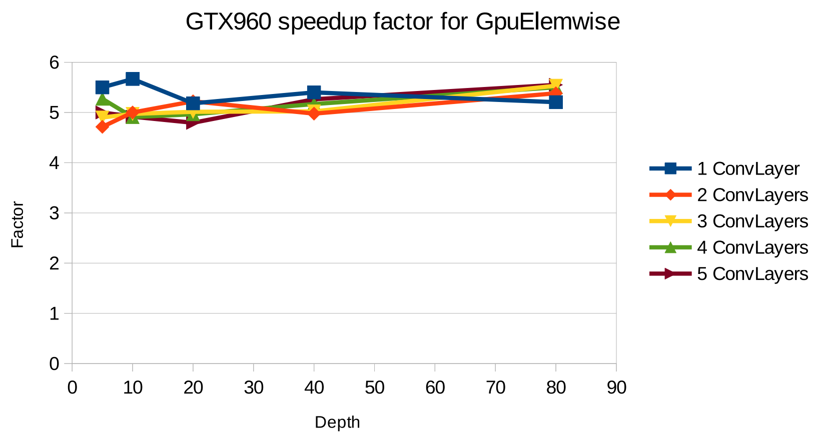

Figure 32: Time taken per ms for the GpuElemwise-process when varying number of ConvLayers and their depth.

The variation in time needed depending on the number of FC-layers and their number of neurons is shown in Figure 33.

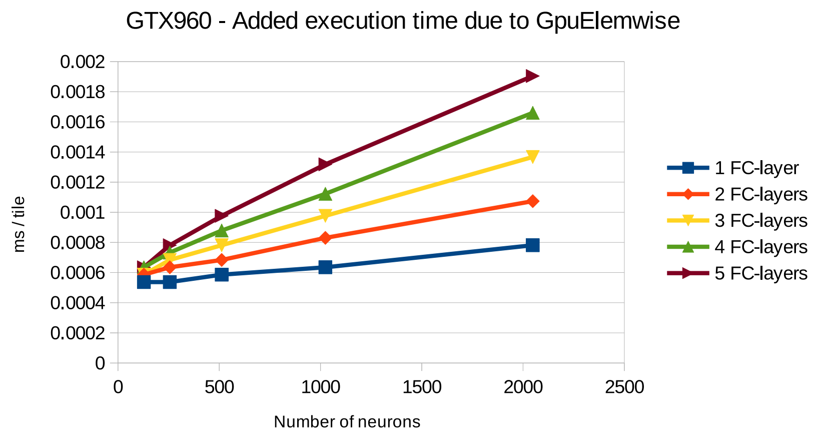

Figure 33: Time taken per ms for the GpuElemwise-process when varying number of FC-layers and their number of neurons.

The variation in time needed for the GpuElemwise-process for the TegraX1 shown in Figure 34.

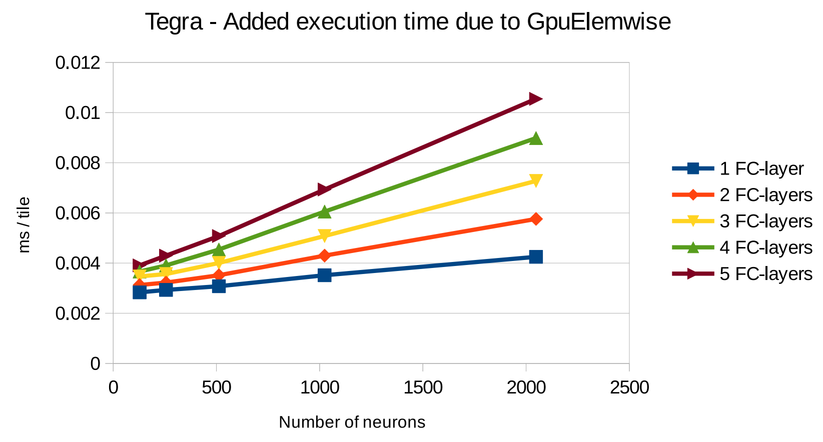

Figure 34: Time taken per ms for the GpuElemwise-process when varying number of FC-layers and their number of neurons.

Figure 35 shows the speedup factor when comparing the GTX960 with the TegraX1 for the GpuElemwise-operation.

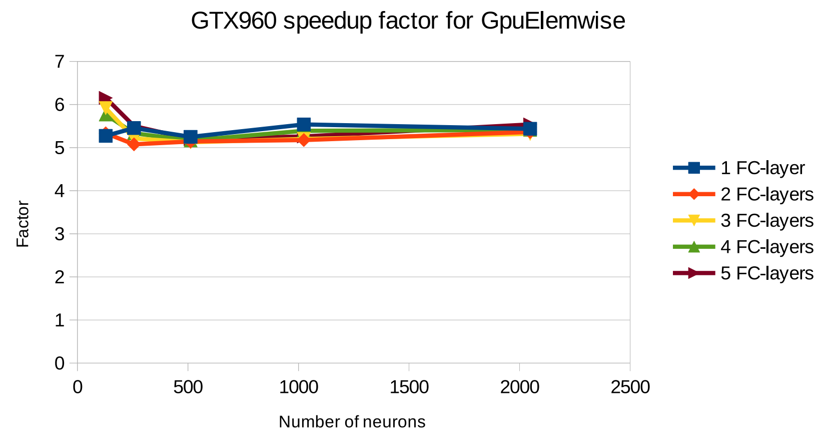

Figure 35: Time taken per ms for the GpuElemwise-process when varying number of FC-layers and their number of neurons.

Figure 36 shows that the performance is higher when the number of neurons is at a multiple of 128.

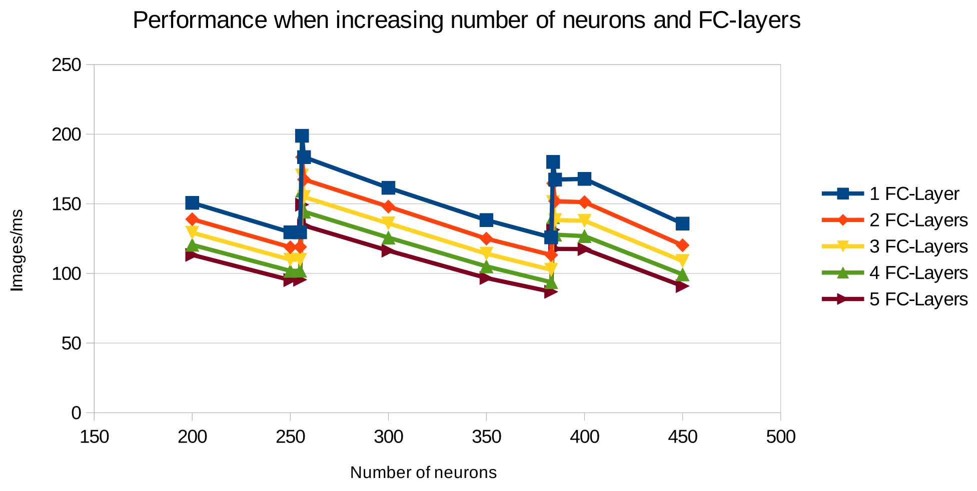

Figure 36: Performance when varying the number of neurons and FC-layers.

Table 1: Performance change for four networks during a long performance test without restarts.

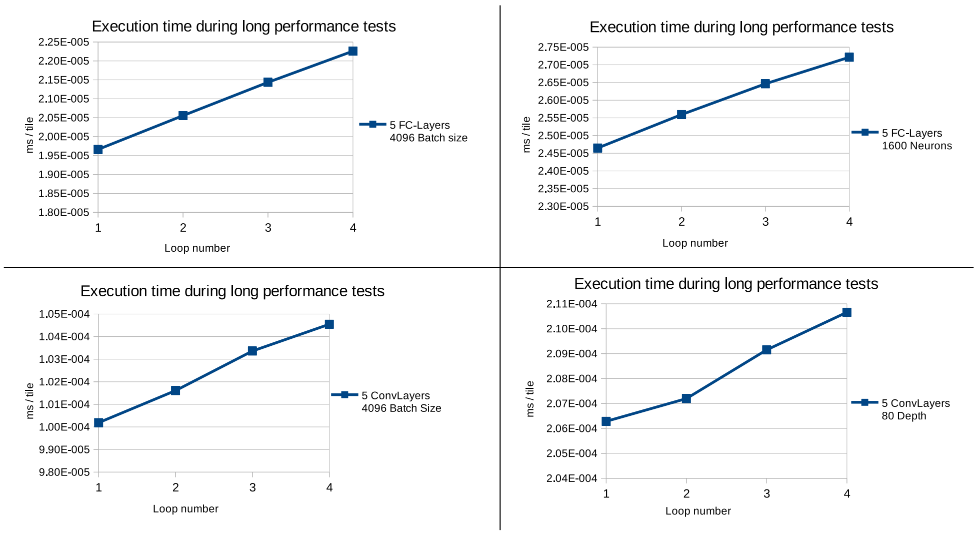

Table 1 shows how the performance varied for four different networks when the classification process for all the networks was looped four times in a row without restarting the computer.

Table 2: Performance and time taken to train until a certain accuracy has been reached.

Type of network	Tiles/ms	90%	95%	97%
5 ConvLayers and 1 FC-layer	33.6	48.9 s	119.9 s	256.3 s
1 ConvLayer and 5 FC-layers	148.6	63.8 s	139.6 s	260.4 s
3 ConvLayers and 3 FC-layers	42.0	35.1 s	96.7 s	183.5 s

Table 2 shows three different network's performance and the time it takes to train them to different percentage of accuracy.

4 Discussion

4.1 Measurement Analysis

As shown in Figure 10 and Figure 11 choosing a batch size that is too small results in poor performance. While a very large batch size will result in a better performance the rate of improvement of performance corresponding to batch size reduces after reaching a batch size of 512. Training very large batch sizes requires a significant amount of GPU-memory and will most probably result in a longer training time. Therefore a batch size of 1024 was considered suitable and gives a consistent base line for all measurements as some tests would not fit using a larger batch size due to memory constraints on the GPU. This test was not performed with the TegraX1 since it, due to it being less powerful compared to the GTX960, will have a reduced rate of improvement of performance corresponding to batch size before a batch size of 512. Therefore a batch size of 1024 is a fitting base line for the TegraX1 as well.

Figure 36 shows that the performance of FC-layers varies with the number of neurons in a way such that when the number of neurons is precisely $128 + 128 * n$, the best performance is achieved. To ensure that the performance achieved in the tests of the FC-layers are possible to compare the number of neurons chosen were 128, 256, 512, 1024 and 2048. The initial plan was to use the values 100, 200, 400, 800 and 1600, but that lead to strange results as the performance when using 400 neurons was better than when using 200 neurons. This is probably an issue with the Nvidia GPU-driver, or Theano and CUDA and the way GpuDot22-operation have been implemented. It probably has to do with the way memory is arranged in parts of power of twos and by utilising this the developers wants to make the calculations more efficient and have succeeded in doing so at the exact points of $128 + 128 * n$ neurons, but not when using another amount of neurons. This test was not performed with the TegraX1 since there is no reason for why it should not behave the same way.

The performance tests were initially done in one large bunk, testing all the different networks consecutively, and looping the whole process four times. As can be seen in Table 1 this resulted in increasingly worse performance for each loop. Table 1 shows only four networks as example, but each network's performance behaved in the same way. Why this happens is an interesting topic of discussion and not entirely clear. The first thought that might come to mind is that the increased heat of the GPU will result in slower performance as the clock frequency of the GPU is lowered if the GPU becomes too warm. However, when considering that the performance test took several hours, this explanation seems unlikely due to the fact that the GPU will have reached its' maximum temperature long before the end of the test, and it should not run slower due to heat issues after having reached the maximum temperature. This is reinforced by the fact that the time taken for convolutions and other mathematical operations doesn't increase. What increases in time is the operation 'CudaAlloc'. Taking this information into account makes it probable that the source of this problem instead could be fragmentation in the GPU's memory. This would explain why it would take more time for CUDA to allocate the memory slots and if the fragmentation in the memory increases over time that would mean that the time to allocate the memory also would increase. If memory fragmentation the reason this behaviour occurs. Then again, this strange behaviour seems to be an issue with Theano or the Nvidia GPU-driver. if it happens on other GPUs and if this can be avoided in some way are all topics of interests that could use further investigation. This problem was overcome by rebooting the computer after each subset. This action removed the traces of this behaviour.

When disabling asynchronous execution while using the profiler to get the time it takes to predict a certain amount of images a potential problem is that asynchronous execution could be faster than the time achieved in the test. The profiler, however, shows that the operations are done in succession even when asynchronous execution is enabled. This means that the synchronous and asynchronous execution times are the same.

4.2 Performance Analysis

Figures 15 and 16 shows a large decrease in performance from adding a second ConvLayer compared to adding a third, fourth or fifth. The reason for this is that the first ConvLayers creates 20 pictures from 1 picture, while all the subsequent ConvLayers creates 20 pictures from 20 pictures. In other words the first ConvLayer uses [1 x width x height] + [20 x width x height] pixels in its' calculations and the second layer uses [20 x width x height] + [20 x width x height] just as the layers afterwards. This large difference in data used results in the large decrease in performance when adding a second ConvLayer.

Even when the depth is constant after the second ConvLayer the decrease of performance isn't linearly decreasing, as is shown in Figures 15 and 16. This is due to the height and width becoming smaller with each successive ConvLayer. Starting at 28x28 the first layer outputs a 24x24 tile, the second ConvLayer a 20x20 tile and so on. This reduction of tile size leads to a reduction in time taken due to a lower amount of computations needed for each successive convolution. Theano did not have the option to used zero-padding, which would have kept the same picture size through all ConvLayers and would have been interesting to investigate as well.

It is easily seen in Figures 12, 13, 15 and 16 that FC-layers are cheaper than ConvLayers by a large margin as many more tiles can be classified per ms. Depending on the choice of amount of neurons and depth as much as five FC-layers or more can be added to the network without decreasing the performance to that of a network consisting of only two ConvLayers. ConvLayers are however often needed to get a good classification performance with the network. This does however show that when going for real-time performance it would be wise to compare the different variations of networks that can be built to achieve the same predication ratio to get the best performance in tiles/ms as well.

Table 2 shows that the time it takes to train a network to a certain degree of accuracy doesn't correlate with performance in tiles/ms in anyway. This means that even if the time to train the network is known, the performance of it must still be tested.

When comparing the performance of the GTX960 and the TegraX1 the results are mostly as expected. The speedup factor in nearly all cases is approximately 5-8, which correlates with the difference in computation specifications between the GTX960 and the TegraX1. The two processes that differ from the expected speedup factor is the GpuDot22-process and the GpuAlloc-process. The GpuDot22-process is approximately 105 times faster on the GTX960. The writer of this thesis do not know why this is the case, after research it is found that the process calls for other processes, and somewhere in these sub-processes the reason for ths very high speedup may be found. The GpuAlloc-process is approximately 2 times faster on the GTX960. This is to be more or less expected since the process performs no computations, and only allocating the memory should not cause an equal slow down. As can be seen in figure 14 and 17 the overall speedup factor is not influenced by these two deviant processes.

When analysing the performance, an equation that could summarise all the graph's data and output a performance in 'tiles/ms' with a given amount of number of layers, numbers of neurons and depth would be desirable. To test if this is possible an analysis on the data from the GTX960 is done due it it having less irregular points of measurements. FC-layers and ConvLayers have performances that are independent of one another which makes this easier. This is done in (13). Where Neurons is the number of neurons, FC is the number of FC-layers, ConvLayers is the number of ConvLayers and D is their Depth.

$$\begin{aligned} ms/tile = {} & C_1 * FC * Neurons[GpuDot22] \\ & + C_2 * FC * Neurons[GpuElemwise] \\ & + C_4 * ConvLayers * Depth[GpuElemWise] \\ & + C_3 * ConvLayers * Depth[GpuAlloc] \\ & + C_5 * ConvLayers * Depth[GpuDnnConv] \\ & + C_6 * ConvLayers * Depth[GpuDnnPool] + C_7 \end{aligned} \tag{13}$$

Where $C_1, C_2, ..., C_7$ are constants. C_7 being the combined overhead for all operations.

(13) is composed of many different operations, but none of their execution time is dependent on another. They can therefore be seen as six different equations that are combined to make the entire performance equation. This means that by analysing the data of Figures 18, 21, 24, 27, 30 and 33 we can decide each equation. In (15), (16), (17) and (18) the number of ConvLayers is specified in the sum-term as 'n'. 'Tilesize' is the pixelsize of the tile used, in this thesis it is 28 since images from the mNIST-dataset are 28x28 pixels large.

(14) shows the added time required per tile from the GpuDot22-operation. The increased time is related to the number of neurons (Neurons) squared and the number of FC-layers (FC).

$$ms/tile_{GpuDot22} = (6.8 * 10^{-6} + 6.6 * 10^{-11} * Neurons^2) * FC \tag{14}$$

15 shows the added time required per tile from the GpuElemwise-

operation. The increased time is related to the number of neurons, the number of FC-layers, the number of ConvLayers and their depth.

$$\begin{aligned} ms/tile_{GpuElemwise} &= (4.9 * 10^{-5} + 1.9 * 10^{-7} * Neurons) * FC \\ &\quad + 2.4 * 10^{-7} * Depth * \sum_{n=2}^{5} (tilesize - 4n)^2 \end{aligned} \tag{15}$$

16 shows the added time required per tile from the GpuAlloc-operation. The increased time is related to the number of ConvLayers and their depth.

$$ms/tile_{GpuAlloc} = 3.5 * 10^{-8} * Depth * \sum_{n=2}^{5} (tilesize - 4(n-1))^2 \tag{16}$$

17 shows the added time required per tile from the GpuDnnPool-operation. The increased time is related to the number of ConvLayers and their depth.

$$ms/tile_{GpuDnnPool} = 8.2 * 10^{-8} * Depth * \sum_{n=2}^{5} (tilesize - 4(n-1))^2 \tag{17}$$

18 shows the added time required per tile from the GpuDnnConv-operation. The increased time is related to the number of ConvLayers and their depth squared.

$$ms/tile_{GpuDnnConv} = 3.03 * 10^{-8} * Depth^2 * \sum_{n=2}^{5} (tilesize - 4n)^2 \tag{18}$$

The combined overhead was calculated to be approximately $C_7 = 9.8 * 10^-3$. (14), (15), (16), (17) and (18) are all approximate, and due to overhead they may differ more when using smaller values. The reason for the first ConvLayer being excluded in the equations are also due to the overhead being much larger than the time for the operations on a single ConvLayer. Despite this the equations are rather straightforward and give a good general idea of how the different operations influence the performance of the network, and can be used to give an approximation of the execution time per tile. What's noteworthy is that (15) and (18) are dependent on the output layer's tile size while (16) and (17) use the input layer's tile size.

GpuAlloc allocates the memory in the GPU and it's no surprise that it uses the input layer's tile size when doing its operations. GpuDnnConv reads the data from the input layer, calculates and outputs the data to the output layer. The calculations and output of the data takes a significant larger amount of time than reading the data from the input layer, hence the dependency on the output layers tile size. The GpuElemwise-operation is performed after the GpuDnnConv-operation which explains why it's dependent on the output layer's tile size. The reason for the operation GpuDnnPool being dependent on the output layer's tile size is not as clear and in fact strange since the GpuDnnPool-operation is performed after the GpuDnnConv-operation. One plausible reason would be the developers of Theano have implemented the operation in such a way that it reads and calculates values from the saved input layer after the GpuDnnConv-operation and then outputs the values to the output layer.

5 Conclusion and Further Developments

Image analysis with CNNs is a still-evolving field of science which is improving in both terms of training efficiency and accuracy at a fast rate. In many cases through more complex networks with more layers than before. While the increase in accuracy is a desirable factor for CNNs, increasing accuracy often means increased complexity and lowered performance in terms of classifications per ms. As shown in this report a network that is too complex will result in a low performance in terms of number of images/ms that can be analysed. When considering integrating a CNN into an embedded system that has to be able to analyse a certain amount of images/ms, the networks complexity has to be taken into account due to the desired real-time performance.

It is important to note that all tools used in this project such as Theano, Cuda and cuDNN are still undergoing intensive development and improvement. Whether this development focuses on improving the training aspect of the CNNs or the real-time performance is very unclear but it is likely that the first one is the case. Even when the real-time performance gets improved due to faster GPUs or better software, it's clear that testing the performance of trained networks still will have be to done.

Adding FC-layers is, compared to adding ConvLayers, cheaper by such a large margin that When trying to improve the accuracy for a CNN designed for an embedded system increasing and/or modifying the number of FC-layers should have priority.

From the tests performed on both the GTX960 and the TegraX1 it seems that the performance correlates with the specifications of the GPU making the classifications.

Further tests could be to try other programs than Theano such as Torch or Caffe, perform tests on other GPUs and SoCs, and investigate the source of the increasing time demanded by the 'CudaAlloc'-process. As well as varying the ConvLayers receptive field's size, stride, padding and the pooling layer's equivalents. Other testing would include finding networks for different datasets that optimises both classification performance and training accuracy.

References

[1] T. Dettmers, “History of deep learning.” https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-history-training/, Dec 2015.

[2] Y. LeCun, L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, I. Guyon, D. Henderson, R. E. Howard, and W. Hubbard, “Handwritten digit recognition: Applications of neural net chips and automatic learning,” in *Neurocomputing, Algorithms, Architectures and Applications* (F. Fogelman, J. Herault, and Y. Burnod, eds.), (Les Arcs, France), Springer, 1989.

[3] A. Karpathy, “Lessons learned from manually classifying cifar-10.” http://karpathy.github.io/2011/04/27/manually-classifying-cifar10/, 2011.

[4] G. DeepMind, “Alphago.” https://deepmind.com/alpha-go, 2016.

[5] G. H. David Stavens and S. Thrun, “Online speed adaptation using supervised learning for high-speed, off-road autonomous driving.” Stanford Artificial Intelligence Laboratory.

[6] B. Heisele, “Visual object recognition with supervised learning,”

[7] SethBling, “Reinforcement learning in games.” https://www.youtube.com/watch?v=qv6UVOQ0F44, June 2015.

[8] M. Studio, “Path finding.” http://mnemstudio.org/path-finding-q-learning-tutorial.htm, November 2015.

[9] M. Y. T. Kimoto, K Asakawa and M. Takeoka, “Stock market prediction system with modular neural networks.” Neural Networks, 1990., 1990 IJCNN International Joint Conference on, pages 1–6,, 1990.

[10] J. Millin, “In search of structure: Unsupervised learning in foreign exchange.” School of Informatics, University of Edinburgh, 2010.

[11] D. D. McDonald and U. Kelly, “The value and benefits of text mining.” http://bit.ly/jisc-textm.

[12] A. S. Simona Balbi and N. Triunfo, “Text mining tools for extracting knowledge from firms annual reports.” Dipartimento

di Matematica e Statistica - Università “Federico II” di Napoli - Italy.

[13] T. Mikolov, “Recurrent neural network based language model.” School of Informatics, University of Edinburgh, Dec 2015.

[14] “Classification dataset results.” `http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html`, 2016.

[15] I. Guyon, “A scaling law for the validation-set training-set size ratio.” AT&T Bell Laboratories, Berkeley, California.

[16] “Geforce gtx 960 specifications.” `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-960/specifications`, 2016.

[17] “Tegrax1 specifications.” `http://www.nvidia.com/object/tegra-x1-processor.html`, 2016.

[18] T. T. D. Team, “Theano: A Python framework for fast computation of mathematical expressions.” arXiv:1605.02688, May 2016.

[19] NVIDIA, “Cuda.” `http://www.nvidia.com/object/cuda_home_new.html`, 2016.

[20] NVIDIA, “Cuda cudnn.” `https://developer.nvidia.com/cudnn`, 2016.