USE OF LEARNED OPTIMISERS IN COMPUTER VISION

181297
University of Sussex
Department of Informatics
BSc Computer Science (with an industrial placement year)
Supervisor: Dr Ivor Simpson

Nikolas Eleftheriou

2021

1 Statement of Originality

This report is submitted as part of the requirements for the degree of BSc Computer Science (with and industrial placement year) at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged. I hereby give permission for a copy of this report to be loaned out to students in future years.

Signature for statement of originality:

Nikolas Eleftheriou

[illegible]

2 Acknowledgments

I would like to express my sincere gratitude to my supervisor Dr. Ivor Simpson for providing me the opportunity to do the project. His enthusiasm about the topic, patience, insightful comments and helpful information have helped me greatly throughout my research and writing of this thesis. His extensive experience and professional expertise in the area of machine learning and computer vision has enabled me to successfully complete this research.

Abstract

Machine learning approaches have emerged as the preferred approach for developing computer vision solutions. Despite the success of learned features in machine learning, optimization algorithms are still designed by hand. The design of gradient-based optimization algorithms for neural networks such as RMSProp, Adam and SGD as well as adapting optimizer parameters to suit a particular problem, have been researched significantly. However, in recent years, there has been an emerging field of interest in machine learning called meta-learning. The objective of meta-learning is to enable models to learn from a limited number of experiences very quickly, adapt to novel tasks and to build models that generalise better. One of the objectives of this project is to implement the meta-learning method called "Learning to learn by gradient descent by gradient descent"[3] which has shown that optimizers can be learned automatically using LSTM recurrent neural networks and be generalised to new problem domains with similar network architectures. The aim of the project is first, to develop the model in TensorFlow to get a greater understanding of how meta-learning works, as well as transparency (code will be posted on GitHub) and usability. At every stage of the project the performance of the meta-learner is critically analyzed and compared against standard hand crafted optimizers. Additionally, the project explores the application of learnt optimizers in computer vision tasks on well known datasets, where more complex machine learning architectures are used to solve a particular problem. The project takes into account the training time of the algorithms, complexity as well as the algorithm's suitability in the chosen domain. Finally, the project examines the application of the learned optimizer to a model that utilizes the Spatial Transformer Network [16] to learn how to geometrically correct an input image to fit a given example image and test how well the project generalises. In this research I show that learning to learn can be used to optimize neural networks for computer vision tasks. However, tuning the meta-learner's hyper-parameters has been significantly challenging, thus achieving exceptional results has been difficult. This study identifies the fields that have had the most impact on the meta-learner's performance and suggests potential areas for researchers to explore. This will increase the learned optimizer's efficiency and could eventually replace conventional optimizers in machine learning, especially in the field of computer vision, where model generalisation is critical.

Contents

1 Statement of Originality **2**

2 Acknowledgments **3**

3 Introduction **7**
3.1 Project Overview 7
3.2 Aims and Motivation 8
3.3 Summary of Findings 9
3.4 Professional Considerations 9
3.5 Requirements Analysis 9

4 Background Research **11**
4.0.1 Affine Transformations 11
4.1 Bilinear interpolation 12
4.2 Gradient Descent for Machine Learning 13
4.2.1 Batch Gradient Descent 13
4.2.2 Stochastic Gradient Descent - SGD 14
4.2.3 Mini-batch gradient descent 14
4.3 Artificial Neural Networks - ANNs 14
4.3.1 Recurrent Neural Networks - RNN 15
4.3.2 Long short-term memory Networks - LSTM 16
4.3.3 Convolutional Neural Networks 17

5 Method **19**
5.1 Learning to learn by gradient descent by gradient descent 19
5.2 Learning to Learn Implementation 20
5.2.1 Python Libraries 20
5.2.2 Implementation description 20
5.2.3 Gradient Pre-processing 21
5.2.4 Parameter Updates Rescaling 21
5.3 Spatial Transformer Networks - STN 22

6 Study **23**
6.1 Methodology 23
6.2 Experiment 1 - Linear Regression 23
6.3 Experiment 2 - Quadratic functions 23
6.4 Experiment 3 - Neural Network training on Fashion-MNIST Dataset 25
6.4.1 Experiment 3.1 - Neural Network training on Fashion MNIST with multiple layers 27
6.5 Experiment 4 - Learning to Learn with CIFAR-10 dataset 28
6.5.1 Experiment 4.1 - CIFAR-10 using only Convolutional Layers 28
6.5.2 Experiment 4.1 - CIFAR-10 using both convolutional and dense layers 29
6.6 Experiment 5 - Correcting geometric properties of images using Affine Transformations with STN on Affnist MNIST 30
6.6.1 Experiment 5.1 - Generalize affine transformations 31

7 Discussion **34**

8 Conclusion **36**

9 Future Work **36**

10 Appendix **40**

3 Introduction

Machine learning has greatly advanced in the past two decades. Machine learning techniques have emerged as the preferred method for developing solutions in the area of computer vision, speech recognition, natural language processing and many other applications[17]. This is due to the fact that training a machine to anticipate output behaviour is much simpler than manually programming it by predicting the optimal answer for all potential inputs.

More specifically, deep learning, a subset of machine learning, enables computer models to understand and represent data at various levels of abstraction, simulating how the brain works. Deep learning is a diverse collection of techniques that includes neural networks, hierarchical probabilistic models, and a combination of unsupervised and supervised function learning algorithms. Deep learning has made significant advances in a wide range of machine vision challenges, including image classification [29], motion tracking [8], movement recognition [5], semantic segmentation[23] and more.

Image classification is a complicated procedure that is influenced by a variety of factors. There have been numerous advancements in designing a more robust classification model, including the use of fully connected layers, convolutional layers, batch normalisation, and max pooling layers, as well as the introduction of modules that apply image warping to enhance the model's performance.

Image warping can be described as the act of mapping the pixels of an image between a source space (u,v) and a target space (x,y) and create a warp [13]. Image warping is a common tool in the context of image editing and is used for creative purposes, such as real-time or static object deformation and shape manipulation in computer graphics. Besides computer graphics, it also arises in many image analysis problems such as correcting the geometric distortions originated by all the different limited imaging systems[11], as well as restoring a feature's location on a picture to aid in classification tasks and introducing spatial invariance to the model.

To accomplish that, the warp fields are optimised to minimise a cost function, which often involves penalties on the effectiveness of an intended edit being applied, smoothness or other aspects of realism of the result. The warp fields are optimised using traditional optimizers. Despite the success of image warping methods using traditional optimizers, applying machine learning optimization or meta-learning will enable better cost functions, thus better results.

3.1 Project Overview

Initially the project was more focused in the application of the "learning to learn by gradient descent" on an image warping problem which is a common tool in the context of image editing. However, there is limited material in that area with respect to machine learning and Python which would require rewriting computer graphics libraries in Python, already written in C++ (e.g. generate meshes). In addition, image warping deals with many calculations per warp, making C++ a more suitable language to use due to performance speed compared to Python. Furthermore, implementing the learning to learn model has been a difficult task as it requires an advanced machine learning implementation, and the specifics of this will be discussed further in the report. Another difficulty encountered during the investigation was hyper-parameter tuning of the meta-model, which becomes more difficult as models get more complex. As a result, the project became more focused on machine learning, especially the application of meta-learning optimisers in computer vision tasks.

Meta-learning is especially fascinating from the viewpoint of deep learning. Meta-learning aims to allow models to learn from a small number of experiences, to easily learn or adapt to novel tasks, and to create models that generalise better[10]. It is a significant topic to explore as it is a relatively new area of research and in theory can provide many benefits to deep learning. This project investigates the application of casting the design of handcrafted optimisers as a learning problem in isolation and analyse in depth the data gathered in terms of performance, time and generalization. The findings are gathered using well-known datasets, starting with basic linear regression tasks and progressing to more complicated ones such as image classification. The outcomes of each project experiment are critically reviewed and analyzed. After evaluating the results of the initial trials, which revealed the meta-learner's operational state, the project moves on with the meta-learner's integration into the wider area of computer vision, primarily image classification. For each experiment in the project the performance of the models is critically analysed in comparison to handcrafted machine learning optimisers as well as the smoothness and other aspects of realism where applicable.

3.2 Aims and Motivation

Having had a long-standing interest in machine learning and seeing its development as it becomes part of our daily lives, my enthusiasm in the field has increased. More specifically, I'm interested in the application of machine learning in computer vision due to the increasing commercial need for computer vision systems to solve real-world problems. For computer vision, machine learning provides efficient methods for automating model acquisition, updating procedures, adapting task parameters and representations using experience for creating, verifying, and modifying hypotheses.

The primary goal of this study is to investigate in depth the meta-learning method known as "Learning to Learn by Gradient Descent by Gradient Descent."[3] and analyse its performance compared to the handcrafted optimisers. Given the above, the objective is to acquire extensive knowledge and form an implementation strategy for the "learning to learn" optimisation. Using the method outlined in the research paper [3], I will be able to investigate and demonstrate my understanding of how to build a machine learning model using learned optimisers. Furthermore, through understanding both the theory and the implementation strategy in depth, I will be able to implement this approach in the context of computer vision.

Additionally, this project aims to explore and understand existing work in the field of computer vision and more specifically in image classification. Throughout time, many approaches were developed in an attempt to provide solutions for a number of various problems regarding image classification. The goal is to analyse the performance of various models that are optimised using handcrafted optimisers compared to the models using the meta-learning approach.

In the context of analysis, the intention of the project is to analyse the performance, smoothness, and other aspects of realism of learned and handcrafted machine learning optimisers for image classification and image warping where applicable. This will take into consideration the training time of the algorithms as well as the suitability of the algorithm in the chosen domain.

3.3 Summary of Findings

In this paper I found that the "Learning to learn" meta-learning algorithm can be used to optimise neural networks for computer vision. In some cases, with the right amount of hyper-parameter tuning, meta-learning performs equivalently well compared to Adam in image classification methods using fully connected layers. Furthermore, I discovered that tuning the learned optimiser is extremely difficult and time-consuming when models combine both convolutional and fully connected layers, and I was unable to match or outperform hand-crafted optimiser performance for those model architectures. Ultimately, it is demonstrated that the model can generalise well on the problem of applying affine transformations in correcting geometric properties of input images and has the potential to be a great candidate for replacing hand-crafted optimisers with the correct hyper-parameter tuning.

3.4 Professional Considerations

Throughout the project, to ensure that all professional considerations are taken into account, I will strictly adhere to the British Computing Society's (BCS) code of ethics.

As this project is primarily research-based, there is an extensive use of other people's work. It is important to give credit to the authors for the any external research or work included in the project. Therefore, I will properly credit the author of every work cited in the project to avoid plagiarism.

Since this project makes use of third-party datasets and software, I would emphasise the importance of adhering to the BCS code of conduct section 1b "have due regard for the legitimate rights of third parties". The experiments were run on Google Colab, a free to use open source project based on Jupyter Notebook. All other software used at any stage during the project did not require any specialist permission to use, and if it did, the specialist permission would have been obtained before it was used. Furthermore, I will be transparent about the source of the datasets used in this project, properly accrediting the people/organizations that provide them and ensuring that I have the necessary permissions before using them.

3.5 Requirements Analysis

The requirements analysis section is used to concisely describe the system requirements. Within this section, a list of what makes an ideal "Learning to Learn" meta-learning model and a definitive list of requirements are established. These objectives are to be met by completion of the dissertation.

No.	**Requirement**
1	Implement the "Learning to Learn"[3] meta-learning model using Python and Tensorflow v2+
2	The LSTM Optimiser should be able to generalise to similar tasks
3	The LSTM Optimiser should be adaptable to model changes like batch size or activation functions
4	The LSTM Optimiser should provide efficient alternatives to regular optimisers
5	Track performance of machine learning models over training steps
6	Track performance of machine learning models over epochs
7	Test the LSTM Optimiser's performance against ADAM, SGD, RMSprop and Adagrad traditional optimisers
8	Test the ability to optimise a simple linear regression problem using the LSTM optimiser
9	Test the ability to optimise a quadratic linear regression problem using the LSTM optimiser
10	Test the ability to classify well known datasets including Fashion-MNIST and CIFAR-10
11	Test the ability to optimise a neural network containing fully connected layers
12	Test the ability to optimise a neural network containing convolutional layers
13	Test the ability to optimise a neural network containing convolutional and fully connected layers
14	Implement and include a Spatial Transformer Network (STN) block in a neural network
15	Use the STN block to correct geometric properties of an input image based on a target image
16	Test the ability to optimise the neural network using STN for applying geometric transformations using an LSTM optimiser
17	Test the generalisation ability of the model using the STN to apply geometric transformations on different unseen inputs data

4 Background Research

4.0.1 Affine Transformations

Geometric transformations are extremely important in image processing. First, in real world problems we often want to define an object relative to another and construct an image. In addition, when it comes to animation, not only the objects of the image can move around, but also we can move the camera creating the illusion of motion as sequences of images are rendered. [15]

There are multiple image transformations that can be applied to an image to produce a new transformed image. Rotation, scaling, translation, and shear are among some of the operations, and where they are combined, the operation is known as an affine transformation. Scaling is used to scale a given point (x, y) by a scalar, shearing is used to offset the x by a number proportional to y and x by a number proportional to x, and rotating is used to rotate the point around the origin by an angle θ. Apart from translation, the other image transform operations are simply linear transforms that can be represented by a matrix multiplication of a point represented as a vector. This can be represented by the following equation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} ax + by \\ dx + ey \end{bmatrix} = \begin{bmatrix} a & e \\ d & e \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

To apply a more complex transform, such us rotate the object, scale it up to a new size and apply shear transformation, this can be illustrated as:

$$x' = S(H(Rx) = (SHR)x = Mx$$

Where

- S = Shear transformation
- H = Scale transformation
- R = Rotation transformation

The above equation defines a sequence of three transformations. As matrix multiplication is associative, the parenthesis can be removed. Multiplying the three matrices S H and R together gives us a new matrix M which contains all transformations for point x.

In matrix form the above transformation operations can be represented as:

$$Scale : \begin{bmatrix} s'_x & 0 \\ 0 & s_y \end{bmatrix} Shear : \begin{bmatrix} 1 & h_x \\ h_y & 1 \end{bmatrix} Rotate : \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$$

Where

- s_x and s_y are the scale of a point x, y,
- h_x and h_y are the horizontal and vertical shear factor
- θ represents a counterclockwise rotation angle around the origin

To represent translation in a matrix notation we need to introduce another dimension. The previous matrices' functionality will not be affected, but this structure allows us to apply all transformations using matrices:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + c \\ dx + ey + f \\ 1 \end{bmatrix}$$

Now we can define the following transformation matrices

$$Scale: \begin{bmatrix} s'_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} Shear: \begin{bmatrix} 1 & h_x & 0 \\ h_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} Rotate: \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} Translate: \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

Where Δx and Δy are the translation in direction x and y respectively.

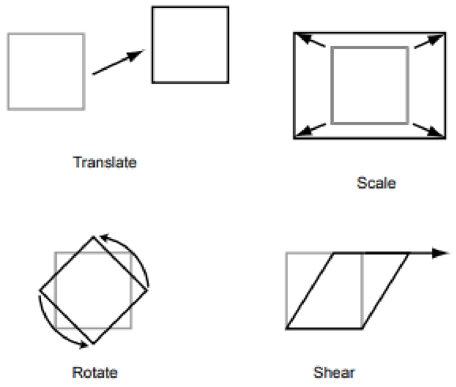


Figure 1: Representation of the affine transformation operations [15].

4.1 Bilinear interpolation

When applying transformation matrices to an input image, the matrix multiplication output contains floating point values, making it difficult to map these values to the corresponding pixel value at the output location. Additionally, when applying a scale transformation by a non-integral scale factor there are pixels that are not assigned suitable pixel values. To address this issue, bilinear interpolation is used to resample images and textures in computer vision and image processing.

Figure 2 shows an example of an output when a transformation matrix is applied to a point P. The red dots with labels Q_{xy} represent the nearest integral pixel indices. Further, the weighted average of distances from the nearest $Q's$ is used to measure the pixel intensity value at P. Finally, the points R1 and R2 are intermediate representations that will aid in math calculations. Pixel intensities are given by the following equation:

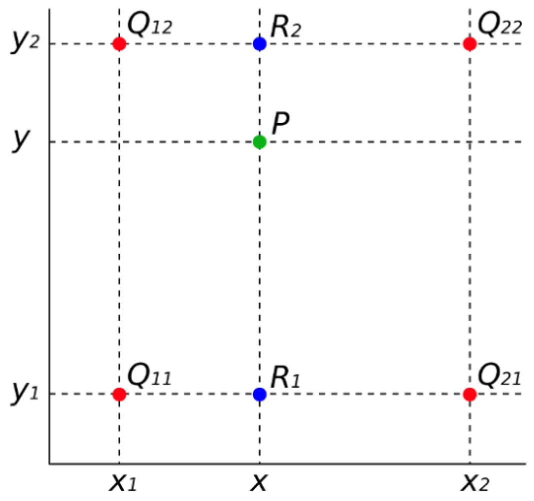


Figure 2: The four red dots represent the data points, and the green dot represents the interpolation point [33].

$$
\begin{aligned}
R_1 &= \frac{(x_2 - x)}{(x_2 - x_1)} Q_{11} + \frac{(x - x_1)}{(x_2 - x_1)} Q_{21} \\
R_2 &= \frac{(x_2 - x)}{(x_2 - x_1)} Q_{12} + \frac{(x - x_1)}{(x_2 - x_1)} Q_{22} \\
P &= \frac{(y_2 - y)}{(y_2 - y_1)} R_1 + \frac{(y - y_1)}{(y_2 - y_1)} Q_{R_2}
\end{aligned}
$$

Where $x_2 - x_1 = y_2 - y_1 = 1$ as the distance from a given pixel a to pixel b is 1

4.2 Gradient Descent for Machine Learning

Gradient descent is an optimisation algorithm that is commonly used in machine learning models and is the most popular neural network optimisation algorithm. This algorithm is used to find the values of the parameters θ of an objective function f which minimizes a cost function. Gradient descent has three main variants that vary in the amount of data used to calculate the gradient of an objective function [27].

4.2.1 Batch Gradient Descent

This is the vanilla variant of the gradient descent which calculates the gradient of the cost function for the entire training dataset with the parameters θ:

$$\boldsymbol{\theta = \theta - \eta \cdot \nabla_\theta J(\theta)}$$

Where:

- η = learning rate
- $J(\theta)$ = is the objective function parameterised by the model's parameters θ

The drawback of this version is that it can be very slow, as the gradients for the whole dataset need to be calculated for every update. This can be easily represented in a pseudo algorithm:

Algorithm 1 Batch Gradient Descent

```
1: while $iteration < epoch$ do
2:     $paramsGradient \leftarrow evaluateGradient(lossFunction, data, parameters)$
3:     $parameters \leftarrow parameters - learningRate * paramsGradients$
4:     $iteration \leftarrow iteration + 1$
```

For a number of epochs we evaluate the gradient (paramsGradient) of a loss function for the whole dataset and then update the parameters of the function based on the learning rate. The learning rate determines the scale of the update we perform at each iteration. This approach is guaranteed to find the global minimum for convex error functions and the local minimum for non-convex functions.

4.2.2 Stochastic Gradient Descent - SGD

This method updates the parameter for each training example x_i and label y_i. This increases the training speed of SGD compared to the batched gradient descent. The SGD performs frequent updates, resulting in significant fluctuations of the objective function.

$$\boldsymbol{\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x_i; y_i)}$$

The heavy fluctuation of SGD allows the algorithm to jump to new local minima, but it complicates the convergence to the desired local minima. The solution to this problem is to reduce the learning rate gradually to allow the SGD to converge to the target local or global minimum similarly as the vanilla version.

4.2.3 Mini-batch gradient descent

This version combines both batch gradient descent and SGD that carries out an update for every mini batch of n training examples

$$\boldsymbol{\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x_{(i:i+n)}; y_{(i:i+n)})}$$

This approach decreases the variance of parameter changes and makes use of highly optimised matrix optimisations widely used in deep learning libraries because of their high efficiency. A common batch size is between 50 and 256, but the size can vary depending on the application.[27]

4.3 Artificial Neural Networks - ANNs

Artificial neural networks are computer systems that simulate the way human brain processes information, inspired by biological neural networks. An ANN is a collection of multiple weight-related single units, artificial neurons or processing elements arranged in layers that form the neural structure of the model[30]. Each connected neuron has a weighted input, a transfer function and an output where the behaviour of the network is determined by the network's architecture, the transfer functions of each neuron and the learning rule. Each weight is an adjustable parameter which allows the system to "learn". The weighted sum of the inputs constitutes each neuron's activation. The output of each neuron is produced from the activation signal passed through the transfer function. During training time, the weight of each neuron is adjusted with the goal to optimise the inter unit connections by minimizing the error predictions until a specified level of precision is achieved by the network. When a neural network is trained and tested, the output for a given input can be predicted. [1]

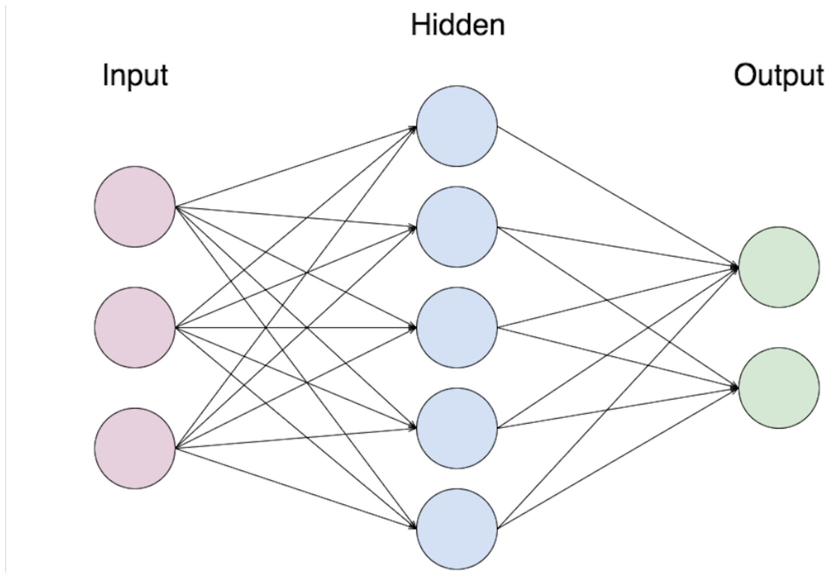


Figure 3: A simple graphical representation of an Artificial neural network organised in layers [7].

4.3.1 Recurrent Neural Networks - RNN

Recurrent Neural Networks is a feed-forward neural network that has an internal memory which allows it to handle a sequence of inputs by having recurrent hidden states where each hidden state's activation depends on the previous activation. Therefore, all RNN's inputs relate to each other, making this network applicable for a number of applications like speech recognition, translation etc. For example, if the input is n words long the network will unroll into an n-layer neural network. The mechanism of the RNN follow this pattern: [31]

- x_t represents the input at a given step t
- h_t represents the hidden state and the memory at a given step t. we calculate h_t based on the previous hidden state and the input at the current step.

$$h_t = \sigma(W_{ph}h_{t-1}, W_{ch}x_t)$$

where

- W represents the weight
- W_{ph} represents the weight of the previous hidden state
- W_{ch} represents the weight of the current hidden state
- σ represents a non-linear function, normally tanh or ReLU

We represent output as:

$$o_t = softmax(W_{ho}h_t)$$

Where W_{ho} is the weight of the output state

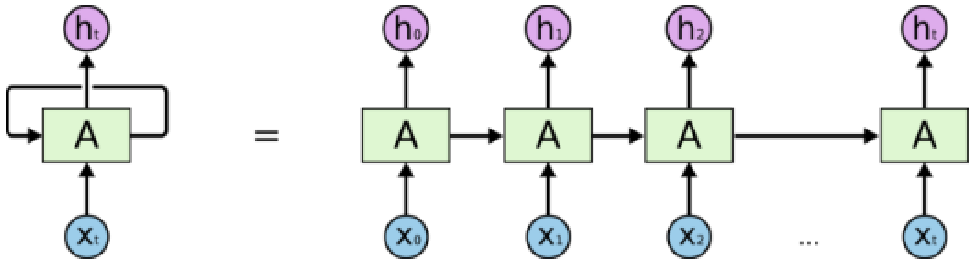


Figure 4: Shows an unrolled recurrent neural network [21].

To train an RNN we can use a back-propagation algorithm called Back-Propagation Through Time [12]. However, training an RNN with long-term dependencies causes the gradients to explode or vanish, making training an RNN difficult for such a task.

4.3.2 Long short-term memory Networks - LSTM

The Long Short-Term Memory [14] algorithm was introduced to resolve the RNN issue of vanishing gradients and make it possible to learn long-term dependencies. The LSTM structure has three gates.

Input gate: Responsible for choosing which input values should be used to modify the memory. This is achieved by using the *sigmoid* function that decides which values will be let through and the *tanh* activation function determining the weight of the values depending on the level of importance. The weighted values range from -1 to 1.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_C)$$

Forget gate: This gate decides which details to discard from the block using the *sigmoid* function.

$$f_t = \sigma(w_f x_t \cdot [h_t - 1, x_t] + b_f$$

Output gate: The output is determined upon the input and the memory of the block. Similarly, as above, the sigmoid function is used to evaluate which values to let through and the tanh function gives a weight to the values passed based on the level of importance multiplied with the output of sigmoid.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t + b_o]$$
$$h_t = o_t * \tanh(C_t)$$

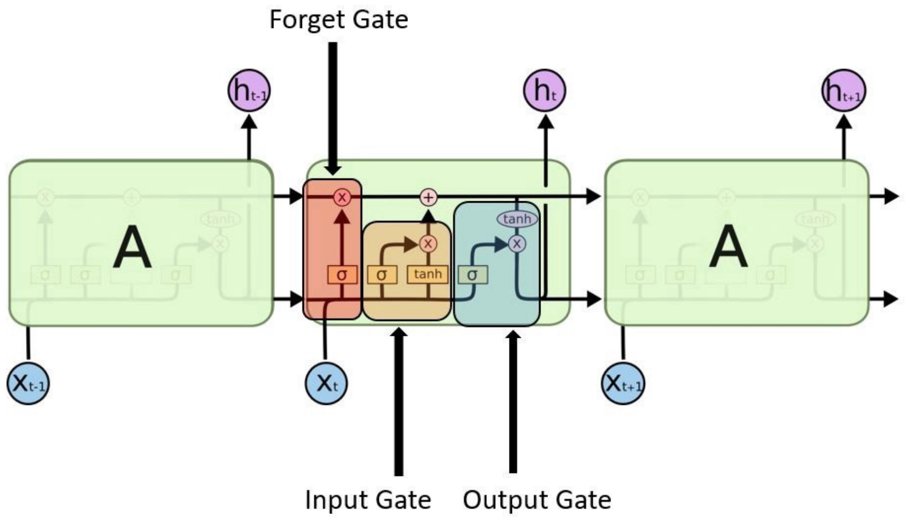


Figure 5: Shows an LSTM network with the three gates clearly displayed [21].

4.3.3 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a Deep Learning algorithm that can take an image as input, assign importance to the network's trainable variables based on various aspects/objects in the image, and differentiate between them. The network's name comes from the mathematical operation between matrices called convolution. When compared to fully connected neural networks, CNN requires fewer parameters, allowing for the creation of larger models to solve more complex problems, which is not possible with traditional fully connected networks due to limited computational resources. The CNN has an excellent performance in machine learning problems and more specifically in applications that deal with image data making it a perfect tool for solving computer vision related tasks [2]. Through the application of relevant filters, a convolutional network can successfully capture the spatial and temporal dependencies in an image. Further, because of the reduced number of parameters involved and the reusability of weights, the architecture performs better, which is more fitting with the image dataset.

The CNNs overall architecture consist of three types of layers: convolution layers, pooling and fully connected layers. The functionality is broken down into these components: The input layer which holds the pixel values of the image, the convolutional layer which determines the output of neurons and applies the activation function, the pooling layer which performs down-sampling along the spatial dimensionality which reduces the parameters within the activation and finally the fully connected layers that produce the final output [25].

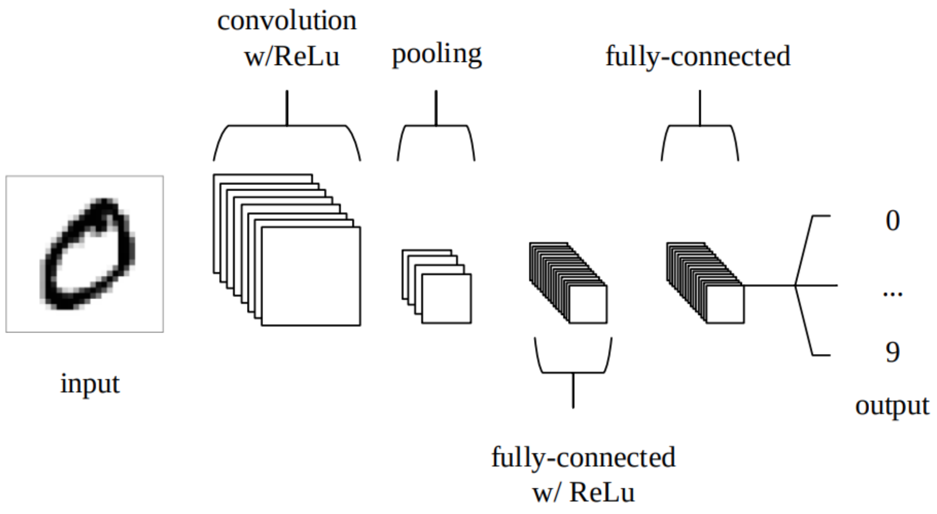


Figure 6: Shows a high level CNN architecture [25].

5 Method

5.1 Learning to learn by gradient descent by gradient descent

The "Learning to Learn" approach shows that we can cast the design of optimisation algorithms as learning problems allowing the algorithm to automatically learn and make use of the structure of the given problems[3]. This approach replaces the hand-designed optimisers with a learned update rule which is called optimiser g specified by its own parameters ϕ. Compared to the vanilla gradient descent equation these feature is represented as:

$$\boldsymbol{\theta_{t+1} = \theta_t + g_t(\nabla f(\theta_t), \phi)}$$

The algorithm can be described as an RNN which takes the output from the optimisee and feeds that output to the optimiser function g which maintains its own state and provides dynamic parameter updates to the target model. The loss function can be represented as

$$\mathcal{L}(\phi) = E_f[f(\theta^*(f, \theta))] \tag{1}$$

Where $\theta^*(f, \theta)$ represents the final optimisee parameters which act as the input of the optimiser parameters ϕ. The objective function above depends only on the final parameter value, but training the optimiser requires an objective function that depends on the entire optimisation trajectory. In other words, the loss of the optimiser is the sum of the losses of the optimisee as it learns. This can be represented as:

$$\mathcal{L}(\phi) = E_f[\sum_{t=1}^{T} w_t f(\theta_t)] \tag{2}$$

Where:

$$\begin{aligned} \theta_{t+1} &= \theta_t + g_t \\ \begin{bmatrix} g_t \\ h_t + 1 \end{bmatrix} &= m(\nabla_t, h_t, \phi) \\ w_t &= \text{are arbitrary weights associated with each time-step} \end{aligned}$$

By applying gradient descent on ϕ we can minimise the value $\mathcal{L}(\phi)$. By sampling a random function f and applying back-propagation we can compute the gradient estimate $\frac{\partial \mathcal{L}(\phi)}{\partial \phi}$. We set the gradients along the dashed edges to $\frac{\partial \nabla_t}{\partial \phi} = 0$, to avoid computing second derivatives of f assuming that the optimisee gradients do not depend on the parameters of the optimiser.

From the equation (2) we observe that gradients exist only if the weight w_i is not 0. If all weights are set to 1, only the final optimisation stage provides data that corresponds to the objective function(5.1), making the back-propagation through time inefficient. To overcome this issue they have relaxed the objective at intermediate points along the trajectory by applying a constraint to the weight value $w_t > 0$.

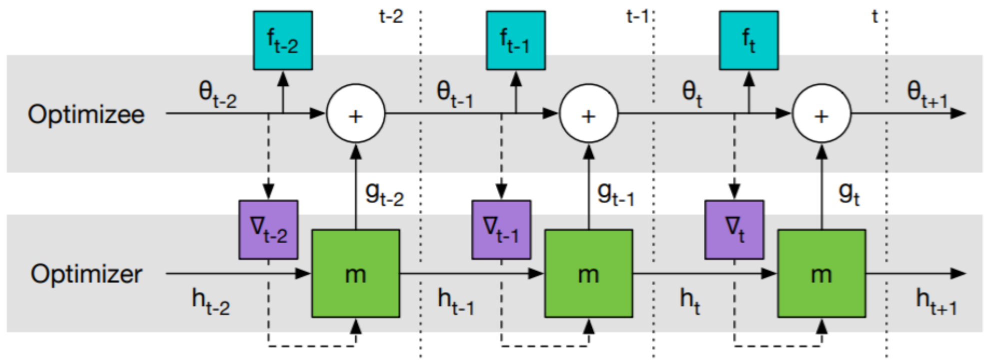


Figure 7: Shows a computational graph for calculating the gradient of the optimiser [3].

As discussed in Section 4.3.1 RNNs cannot be trained with long term dependencies. Therefore, an LSTM (*See Section 4.3.2 for more details*) is used to implement the learning to learn approach Where m is used as an optimiser that operates coordinate-wise on the objective function's parameters. Operating "coordinate-wise" means that each coordinate has its own state and information is not shared across coordinates. For each objective function parameter a separate activation function is used achieving different behaviour at each coordinate. For the learning to learn implementation, a two layer LSTM network is used that has the previous hidden state and the optimisee gradient for a single coordinate as input and outputs the update for the optimisee parameter.

5.2 Learning to Learn Implementation

This section is used to describe the details of implementing the learning to learn algorithm. It aims to provide a more detailed view of how this meta-learner works in practice and how it can be implemented.

5.2.1 Python Libraries

Rather than writing all the methods for this task, I have utilised pre-built libraries. This option was chosen because it allows a variety of different features to be explored within the study in a timely manner. This proved to be a significant decision regarding the choice of programming language as the aim is to utilise the already tested and currently available libraries for this project as extensively as possible. The programming language used to complete the examples below is Python using Tensorflow version 2 framework, which contains the largest portion of libraries used in the example with regards to machine learning.

5.2.2 Implementation description

As mentioned above to implement this we need two distinct neural networks. The first one is called optimisee or the task specific neural network. This is the neural network that performs the original assigned task. This task can range from regression to classification and more. The weights of this model are updated by another neural network called the optimiser using the sum of losses across every training step produced by the optimisee network. Implementing this concept requires unrolling the entire computational graph of the optimisee training in order for TensorFlow to compute the gradients, which is extremely computationally expensive. In order for the gradients to flow through the network and generate the computational graph, the optimisee weights need to be copied and updated during training. As a result, the models using the LSTM optimiser need to be created in a custom way which allows the model's forward pass to use the maintained weights rather than the auto generated weights.

The LSTM optimiser operates coordinate-wise which shares parameters between cells, but it has different hidden states for each parameter which reduces the number of LSTM parameters required. Updates to the LSTM optimiser were done using the Adam or RMSprop optimisers which produced better results than the other optimisers.

Training two neural networks on top of each other is a unique use case, and it took a significant amount of time and effort to create the appropriate computational graphs to allow for training and testing of our LSTM optimiser. Furthermore, training was extremely expensive as each higher-level LSTM training step required the entire training loop of the base optimisee network to be unrolled.

Furthermore, for simple problems like linear or quadratic, the LSTM model is able to learn without any pre-processing of the optimisee gradients. For more complex machine learning models that make use of fully connected layers or convolutional layers, the LSTM optimiser is not able to learn. This is due to the fact that different input coordinates can have very different magnitudes, making it extremely difficult for the optimiser to learn because neural networks disregard small variations in input signals and focus on larger input values. Hence, training more complex models requires a lot of hyper-parameter tuning and more complex pre-possessing methods.

5.2.3 Gradient Pre-processing

To address this problem, we must perform pre-processing on the optimisee gradients, which are the optimiser's inputs. Re-scaling all inputs by an appropriate constant did not help the process of model learning. Instead of just passing in the optimisee gradients for each step, we are passing $(\frac{\log(|\nabla|}{p}), sgn(\nabla))$ if $|\nabla| >= e^{-p}$, where ∇ is the gradient at a current timestamp, otherwise the value passed was $(-1, e^p)$. For this pre-processing system, p is greater than zero and is a parameter that controls how small gradients are ignored [3].

5.2.4 Parameter Updates Rescaling

Additionally, the optimiser produces better results when the updates produced at each step are re-scaled by the factor in the range 0.1 - 0.001. As the meta learner consists of n LSTM layers the output that is produced contains k number of outputs where k is the number of LSTM cell's hidden size. This must be passed through a dense layer that converts the k outputs into a single value that corresponds to the update of one parameter at a current timestamp. Applying a batch normalization layer after the LSTM output and the dense layer would standardise the inputs, giving them a mean of zero and a standard deviation of one, which accelerates the model and enables it to produce better results.

5.3 Spatial Transformer Networks - STN

As described in section 4.3.3, CNNs have been very successful in computer vision applications especially in tasks related to classification [29], localisation [28], semantic segmentation[20] and others. On a CNN model distinguishing object pose and component deformation from texture and shape is a desirable property. This is partially satisfied with the addition of max pooling layers making the model spatially invariant to the features position. Nonetheless, in practice CNNs are not invariant to large transformations of the input data [6] due to the small spatial support of max-pooling layers.

A solution to this problem has been addressed with a Spatial Transformer module [16] which is a module with spatial transformation capabilities that can be used in a normal neural network. The STN is a mechanism that can spatially transform an image dynamically by generating a transformation for each input sample. The transformations can include cropping, scaling, rotations and non-rigid deformations and are performed on the entire feature map. As a result, networks with spatial transformers simplify image recognition by transforming the features of interest to canonical, which are easier to extract [16].

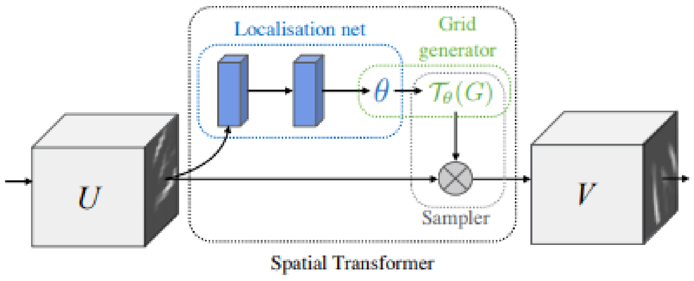


Figure 8: Displays the architecture of a Spatial Transformer Module [16].

The spatial transformer network consists of three modules the Localisation Net, Grid Generator and the Sampler as shown in Figure 8. The localisation net takes an input image and produces θ, the parameters of spatial transformation to be applied to the input image. Then the grid generator creates a sampling grid which is a set of points from which the input map should be sampled in order to generate an output using the predicted transformation parameters. Finally, the sampler takes as inputs the feature map and the sampling grid producing the output map.

This self-contained module can be inserted with any number, at any point within a CNN architecture, which results in spatial transformer networks. In addition, this module is computationally very quick and does not slow down training, resulting in very little time overhead. Placing spatial transformers within a CNN enables the network to learn how to dynamically transform the feature maps in order to better minimise the network's total cost function throughout training[16].

6 Study

The aim of these sections is to analyze the performance of the "learning to learn" meta-learner in isolation. This section aims to explore whether classifying the optimisation as a learning problem improves the performance of machine learning problems and identify strengths and weaknesses of that approach. Some experiments used in this section were inspired by the 'Learning to learn by gradient descent by gradient descent' [3] to get a variety of problems to test this approach that can be adapted to solve a wider range of machine learning problems.

6.1 Methodology

The experiments below were trained using LSTM optimisers with 20 - 40 hidden units in each layer of the LSTM. As stated in Section 5.1, each optimiser is trained by minimising Equation 2 using truncated back-propagation through time. Additionally, the learned optimisers are optimised using ADAM and with learning rates that worked best for each scenario.

For each experiment the trained LSTM optimisers are compared against standard optimisers used in Deep learning such as ADAM, RMSProp, SGD and Adagrad optimiser. The default values were left for each of the standard optimiser hyper-parameters.

6.2 Experiment 1 - Linear Regression

As proof of principle in this experiment we consider training an optimiser on a simple linear regression problem with some added noise on the function. In particular the function we are optimising is of the form:

$$\boldsymbol{f(\theta) = AX + B + Noise}$$

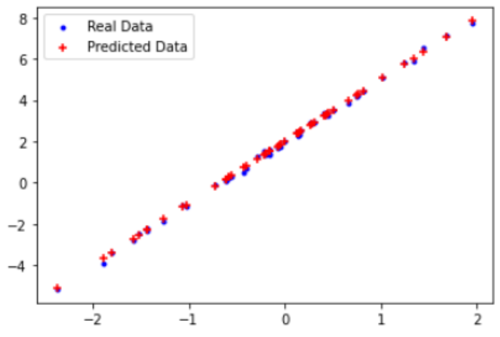
For this example the true value of A is 3 and the true value of B is 2. The optimisers were trained using 200 random samples of X and validated using 50 samples for validation and 50 for testing. Additionally, the optimiser was unrolled for 20 steps and trained for 50 epochs. For the linear regression we have not used any preprocessing neither batch training which validates that the meta-learner can be used to solve simple linear regression problems.

Learning curves for different optimisers can be seen in Figure 22 for validation data. Each colour on the graph represents a different optimiser. In Graph (b), we see that the LSTM optimiser outperforms the hand-crafted optimisers by a huge amount in less epochs. Finally Graph (a), shows the LSTM optimiser predictions on the unseen test data.

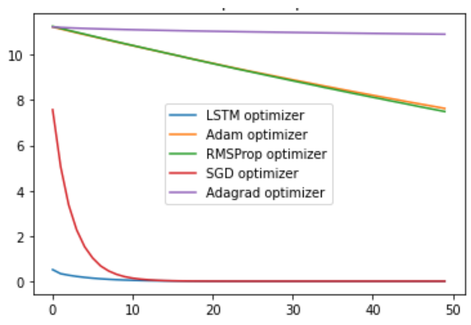
6.3 Experiment 2 - Quadratic functions

This experiment was influenced by the quadratic experiment used in the original paper [3] (section 3.1) where the optimiser is trained to predict data on a simple class of synthetic 10 dimensional quadratic functions. The function that was minimized in this example is shown below:

$$\boldsymbol{f(\theta) = \|W\theta - y\|_2^2} \tag{3}$$



(a) LSTM Optimizer Predictions on Test data



(b) Validation losses

Figure 9: Figure (a) displays the predictions and the actual data of the LSTM Optimizer model when tested on unseen data. Figure (b) displays the validation losses of handcrafted and LSTM learnt optimisers for the linear regression problem over 50 epochs.

The matrices W have a size of 10x10 and both W and y vector were drawn from a Gaussian distribution. The optimisers in this test were trained using 200 random drawn data from the function shown on Equation 3 and they were tested against newly sampled unseen data. In this example the LSTM optimisers were unrolled for 20 iterations and had 2 layers of hidden size 40. Finally, for this experiment there was no pre-processing or post-processing of the data.

The models trained with handcrafted optimisers trained faster than the model trained with the learned optimiser. However, the results of the meta-optimiser show that the model generalizes and converges much sooner in terms of number epochs compared to the standard optimisers. The results of the experiment can be seen below on Figure 10 where it is clear that the trained optimiser performs better than the traditional handcrafted optimisers.

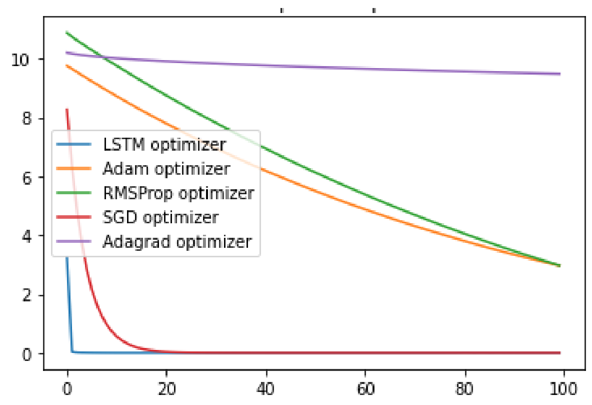
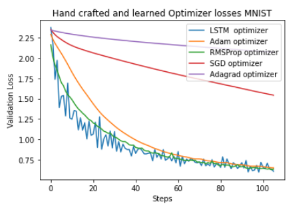
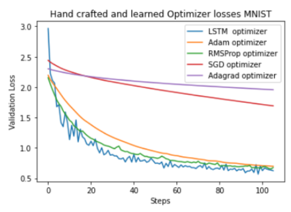


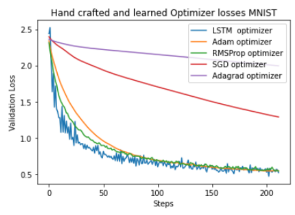
Figure 10: Handcrafted and LSTM learnt optimisers validation losses for the quadratic problem over 100 epochs.



(a) 1 Dense 20 Hidden neurons



(b) 1 Dense 40 Hidden



(c) 1 Dense 40 optimiser trained for 100 steps

Figure 11: Handcrafted and LSTM trained optimisers validation losses for the fashion MNIST classification problem over 100 steps for the first two graphs and 200 steps for the rightmost figure. The figure captions describe the architecture of the model.

6.4 Experiment 3 - Neural Network training on Fashion-MNIST Dataset

In this experiment we investigate whether trainable optimisers can learn to optimise a small neural network on using Fashion MNIST[34] dataset. Fashion MNIST is a dataset which consists of training 60,000 examples and 10,000 test examples. Similar to the normal MNIST dataset, this dataset contains 10 classes and each data example is a 28x28 gray scale image. Instead of using the normal MNIST which contains a lot of handwritten digits, the Fashion MNIST was used, which contains images representing clothes. This can be considered a better example dataset to test if the LSTM optimiser works in a more realistic scenario. Using the MNIST dataset, classic machine learning algorithms can achieve accuracy as high as 97% where in fashion-MNIST the equivalent is 89% [26].

For this experiment the base network consists of one fully connected layer with 20 neurons and one classification output layer with a softmax activation function. The objective function for this network is the cross entropy of the base network's output with parameters θ. This problem is harder when compared to Experiment 1 and 2 as the number of trainable parameters has increased due to the nature of dense layers. First, the custom dense layer class was created to allow a variable containing the weights to be specified when calling the dense layer. This enabled the gradient to flow, allowing the LSTM optimiser to generate parameter updates. However, this process proved to be challenging as the optimiser was unable to learn. After analysing the network's data, a wide range of numbers was observed being used as inputs for the LSTM optimiser. The LSTM network was disregarding the small gradient updates given from the optimisee network making training harder. To solve this problem, the pre-processing method explained in section 5.2.3 was applied to the optimisee gradient output. Another intriguing feature discovered is that the LSTM optimiser was not able to learn when the activation function of the dense layers was set to Relu. This is due to the sparsity of activations the model receives with Relu as it switches off the negative values.

The LSTM optimiser hyper-parameter tuning was complex and time-consuming since the LSTM optimiser's training was very unstable when the parameters were not set correctly. In this analysis the model parameters that were configured were the number of LSTM layers and their hidden units, the optimiser used, usefulness of L2 weight regularization on the LSTM layers, batch normalization layer and activation functions.

Moreover, the original paper [3] displays the graphs in a number of steps rather than epochs. To be able to present that information for the handcrafted optimisers, custom training loops were created to log the loss values for each step. Similarly, the modifications required to accommodate that feature for the LSTM optimiser training loop were created. Figure 11 shows the results gathered when training the LSTM optimiser using batches of 512 data for 100 steps. Additionally, the LSTM optimiser had 2 layers with 20 hidden units and was unrolled for 20 steps during training.

Figure 11 (a) shows the validation losses for a model with a single fully connected layer containing 20 hidden neurons. Graph (b) displays the validation losses of a similar model containing 40 hidden neurons instead of 20. Lastly, graph (c) shows the same model as (b) where the LSTM optimiser was trained for 100 steps but the model was trained for 200 steps. According to the learning curve graphs, the LSTM optimiser outperforms the handcrafted optimisers in the first stages, but RMSprop and Adam have comparable output towards the last steps. Ultimately, we can see that the LSTM optimiser generalises very well on the figure 11 graph (c), despite the fact that it was trained for 100 steps, while the model using the LSTM optimiser was trained for 200 steps. The meta-learner is able to continue improving the performance of the targeted model even though it was not trained for that long as indicated in graph (c).

One question left unanswered in the paper was how well the model performs on unseen test data. Table 1 shows the results gathered when the five models were tested on unseen test data. From the graphs on Figure 11, the LSTM optimiser seems to outperform the handcrafted optimisers. However, it achieved 77% test accuracy same as the RMSprop where Adam optimiser has performed better achieving accuracy score of 78%.

LSTM optimiser	Adam optimiser	RMSprop optimiser	SGD optimiser	Adagrad optimiser
77%	78 %	77 %	59%	32%

Table 1: Fashion MNIST test accuracy of the handcrafted optimisers and the LSTM optimiser over 100 steps with a model with one dense layer and 20 hidden units.

Furthermore, in the original paper [3] the LSTM optimiser was only evaluated on models trained up to 200 steps which might not the best indicator on how well does it perform. Figure 12 represents the results gathered from training the LSTM optimiser using 3 LSTM hidden layers of 20 hidden units. The LSTM optimiser was optimised using the RMSprop optimiser, it was unrolled 20 times and it was trained using iterations of 128 training data per batch. In addition, the training data were split into 90% training and 10% validation. Finally, the optimiser was trained over 5 epochs containing 422 steps each.

LSTM optimiser	Adam optimiser	RMSprop optimiser	SGD optimiser	Adagrad optimiser
85%	86 %	86 %	85%	81%

Table 2: Fashion MNIST test accuracy of the handcrafted optimisers and the LSTM optimiser.

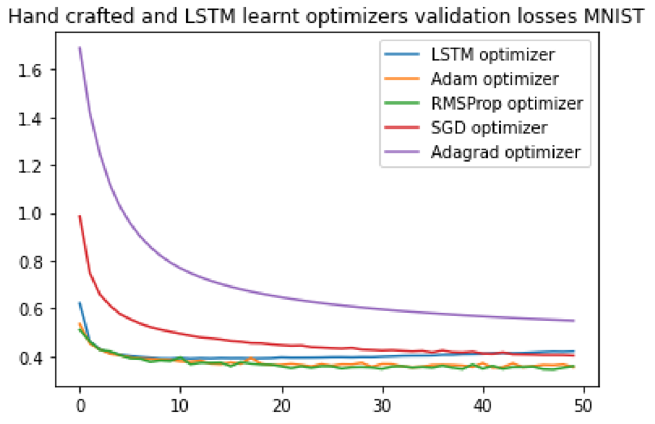


Figure 12: Hand crafted and LSTM learnt optimisers validation losses for the fashion MNIST classification problem over 50 epochs.

The training results show that the LSTM was able to converge faster than the handcrafted optimisers but it reached a limit where it was no longer learning. Even though the LSTM optimiser model had some parameter tuning done and the handcrafted optimisers were using the default parameters, the meta-learner was unable to outperform the Adam and RMSprop optimisers. Moreover, Table 2 displays the results of testing the models on unseen test data. Compared to the other optimisers, the LSTM model was able to generalise relatively well. Although it did not outperform the Adam and RMSprop, the difference was only 1%. Additionally, the process of successfully training the LSTM optimiser and then train the model took significantly longer. As a result, when delivery speed is critical, the LSTM optimiser is not the best choice for solving this classification problem.

6.4.1 Experiment 3.1 - Neural Network training on Fashion MNIST with multiple layers

In this analysis we investigate whether trainable optimisers will learn to optimise a small neural network on using Fashion MNIST dataset but this time with multiple dense layers. For this experiment the base network is a network with two fully connected layers and one classification output layer with a softmax activation function. The aim in this experiment is to observe how the LSTM optimiser works with more than one dense layers. Again for this experiment, the same pre-processing techniques were used as previously mentioned, to achieve the best performance possible.

The LSTM optimiser had two layers of 20 hidden neurons in the first iteration of this process, and it was trained for 100 steps and unrolled for 20 iterations per step. Furthermore, for the LSTM optimiser implementation, weight decay and a batch normalisation layer were used in the optimiser's network to obtain the final parameter updates from the LSTM layers results. Table 3 shows the test accuracy results after training all models for 1 epoch. The LSTM optimiser, along with the model that uses Adam optimiser, has the highest accuracy score. This proves that the LSTM optimiser when tuned correctly can perform very well and even outperform one of the most used optimisers RMSprop.

LSTM optimiser	Adam optimiser	RMSprop optimiser	SGD optimiser	Adagrad optimiser
77%	77 %	75 %	60%	27%

Table 3: Fashion MNIST test accuracy of the handcrafted optimisers and the LSTM optimiser with a model consisting of 2 dense layers with 20 hidden neurons each.

6.5 Experiment 4 - Learning to Learn with CIFAR-10 dataset

In this section, we want to evaluate whether the trainable neural optimisers' can learn to optimise classification performance for CIFAR-10 dataset [18] using both convolutional and dense layers. The CIFAR-10 dataset contains 60000 32x32 colour images divided into 10 classes, each with 6000 images. There are 50000 training images and ten thousand test images.

6.5.1 Experiment 4.1 - CIFAR-10 using only Convolutional Layers

Optimising convolutional layers with the LSTM optimiser was challenging, as the neural network could not learn how to train convolutional layers correctly. To solve this issue, we first created a model consisting of only convolutional layers for classification. More specifically, the model's architecture was composed of 2 convolutional layers consisting of 16 and 32 hidden units followed by max pooling layers. For the output layer, a convolutional layer with 10 hidden units was used followed by a global average pooling layer with softmax activation function. Similarly with experiment 3, training the model using the LSTM optimiser without pre-processing explained in section 5.2.3 was unsuccessful. Following that, neither the pre-processing strategy nor the tuning of the learning rate improved the learning mechanism of the convolutional neural network significantly. For CNN layers, Relu activation function performed better than the other activation functions but training was still unstable. The use of batch normalisation before Relu activation for each CNN layer improved the model's training significantly. Another significant parameter that contributed significantly in the training of the model was the rescaling of the parameter updates of the LSTM optimiser by 0.001. As a result, the LSTM optimiser was restricted from altering the trainable parameters significantly causing the model to loose its training momentum.

Following the above configuration, the LSTM optimiser was trained for 300 steps using 200 images per batch size. In addition, the LSTM optimiser had 2 units of 20 hidden layers and it was optimised using Adam. Figure 13 shows the performance of this trained optimiser in comparison to the handcrafted techniques. The graph shows the losses over the held out validation set. From the figure 13 we observe that the trained optimiser performs better than the models optimised using SGD and Adagrad optimisers and it performs similarly to RMSprop. However, Adam has outperformed all optimisers including the trained LSTM optimiser.

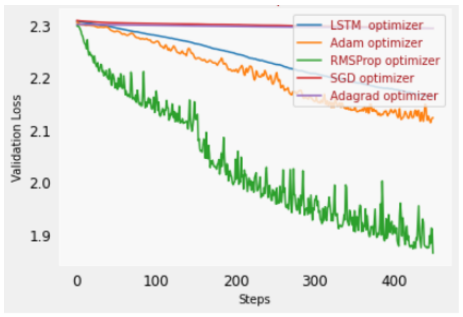


Figure 13: Hand crafted and LSTM optimisers validation losses for the CIFAR-10 classification problem on neural network using only convolutional layers over 450 Steps.

6.5.2 Experiment 4.1 - CIFAR-10 using both convolutional and dense layers

Then, using the information learned about the learned optimiser's behaviour when working with convolutional layers, we investigate whether it can learn to optimise a neural network that includes both dense and convolutional layers. The model we aim to optimise in this section contains 3 convolutional layers of 16, 32 and 64 hidden neurons with batch normalization and Relu activation layers, followed by a dense layer that has 20 hidden neurons. Previously, for each optimisee parameter the architecture was utilizing a single LSTM architecture with shared weights, but separate hidden states. For this example due to the differences between dense and convolutional layers the LSTM optimiser needed to be modified to include two distinct LSTM layers, one for the CNN layers and one for the dense layers. With this modification a coordinatewise decomposition of shared weights and individual hidden states is still used, as in the LSTM optimisers of previous experiments, but LSTM weights are now shared only between parameters of the same kind.

Figure 14 shows the validation losses on a held out test set. The training of models is entirely dependent on the training of the LSTM optimiser prior to model training. Tuning the learnt optimiser's hyper-parameter to obtain high validation accuracy and low validation loss is challenging since each parameter update has a large impact on the LSTM neural network and affect training. In this experiment, Adam once again outperforms all the other optimisers. As seen on figure 14 the LSTM optimiser struggles to optimise the model due to the challenges faced while training the optimiser.

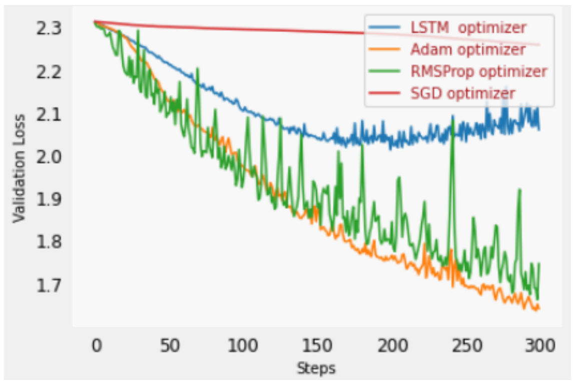


Figure 14: Hand crafted and LSTM optimisers validation losses for the CIFAR-10 classification problem over 300 Steps. The LSTM optimiser contains 2 LSTMs with 3 layers and 20 hidden units, one for CNN layers and one for dense layers.

6.6 Experiment 5 - Correcting geometric properties of images using Affine Transformations with STN on Affnist MNIST

The aim of this example is to see if the model can learn to automatically correct the shape of a given picture using an example image fed to the model during training. The Affnist MNIST dataset, which is based on the MNIST dataset [24], was used in this experiment. The Affnist MNIST differs in that it features images of dimension 40x40 that have had affine transformations applied to them. The model utilizes the Spatial Transformer's architecture where there is a localization network, a grid generator and a sampler as described in section 5.3. To train the model all figures of eight from the dataset were gathered and an example image was chosen. This example image serves as a role model for the network as it learns which affine transformation matrix to apply to the rest of the training images to make them resemble the example image. The loss function for this problem is a custom pixel similarity loss based on the multi-scale structural similarity (SSIM)[32] between the corrected images that have undergone affine transformation and the example image. The loss equation is represented as:

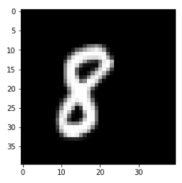


Figure 15: The eight that was used as an example target image to train the model in Experiment 5.

$$\boldsymbol{Loss = 1 - SSIM(x, y)}$$

Where:

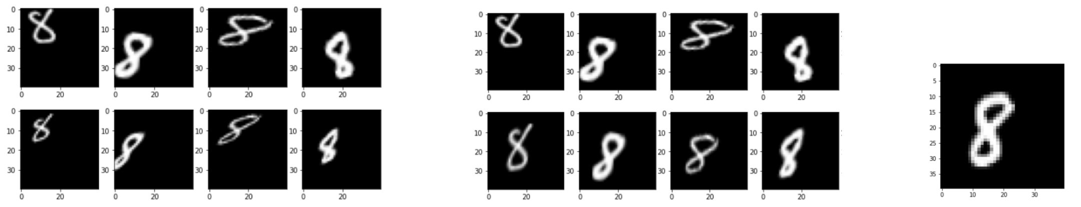
$$SSIM(x, y) = [I(x, y)]^{\alpha} \cdot [c(x, y)]^{\beta} \cdot [s(x, y)]^{\gamma}$$

Where:

$$I(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2\mu_y^2 + C_1}$$
$$c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2\sigma_y^2 + C_2}$$
$$s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}$$

Where: μ_x, μ_y, σ_x,σ_y, and σ_{xy} are the local means, standard deviations, and cross-co-variance for images x, y [32].

The neural network's architecture utilizes both convolutional and fully connected layers. The techniques applied when constructing the model are similar to the ones described in section 6.5. The LSTM optimiser used was trained for 20 epochs using 2 hidden layers and 15 hidden neurons and unrolled for 20 steps. During training, the LSTM optimiser was learning to remove the figure from the image rather than learning to apply the right transformation to align the pixels from the input image to the example image, which caused a challenge when tuning the LSTM's hyper-parameters. This was not a challenge for the handcrafted optimisers.



(a) Model optimised with LSTM optimiser.

(b) Model optimised with Adam.

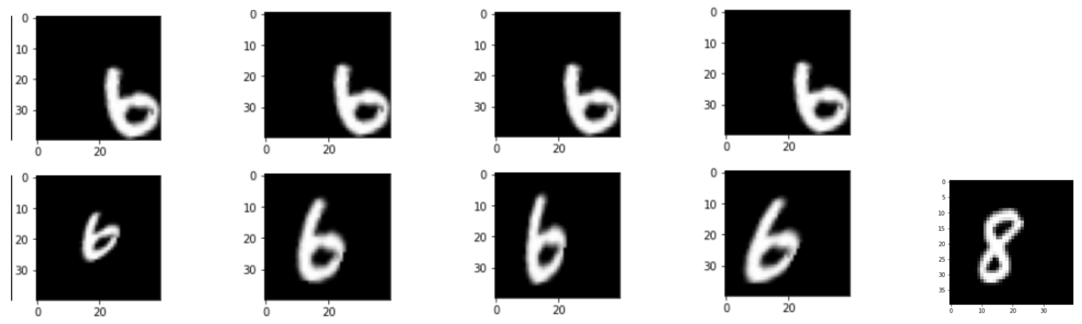
(c) Target number 8 image

Figure 16: Shows the affine transformations applied on unseen images from the model using a model with LSTM optimiser on the left and Adam on the right. The first row shows the input image and the second row displays the image with the affine transformation applied. The model is trying to optimize the affine transformations to match the image on Figure (c).

Figure 16 presents the results of the models when given other unseen warped eight figures. Both models are trying to make the input image match the example image as seen on Figure 16 (c). However, the model trained with Adam produces results closer to the example image as seen from the results. Additionally, compared to the learnt optimiser, handcrafted optimisers trained much faster and produced more realistic results.

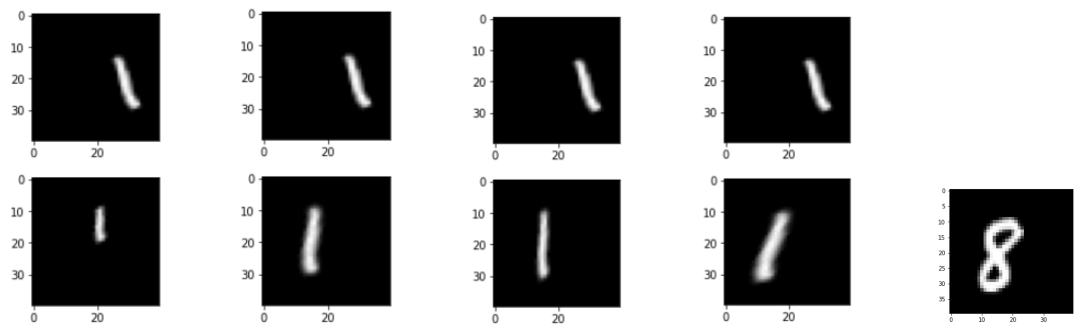
6.6.1 Experiment 5.1 - Generalize affine transformations

The next step in this test would be to see whether the models can learn to generalise to other input images requiring similar transformations. Instead of using only number eight, figures of number six were used which resemble eight, as well as figures of number one which are created using different pixels. Using the trained models from the previous experiment we evaluate the realism of the output images based on the new different inputs. Figure 17 shows the input and the output of images with number six. From the results, we observe that the models can generalise on other inputs with no issues. Using the example target image with number 8 from Figure 15, all models learned how to apply rotation correctly. Additionally, The LSTM optimiser makes the figure six more centered but also smaller than the remaining optimisers. On the contrary, the handcrafted optimisers learnt to apply scaling correctly with Adam producing a figure of number six closer to the Figure 15. Similarly, Figure 18 shows the output images when affine transformation is applied to the input with number one. The optimisers follow the same trends as in the case demonstrating the number six. More specifically, LSTM optimiser is centering the given number but also it scales it down. Further, the remaining optimisers are rescaling the input correctly but RMSprop did not achieve to apply the correct rotation as output images of the inputs with number one appear to be leaning on the right.



(a) LSTM optimiser (b) Adam (c) SGD (d) RMSprop (e) Example 8 image

Figure 17: Shows the affine transformations applied on a different set of images. The first row shows the input image and the second row displays the image with the affine transformation applied. The models at each column where trained with the optimiser mentioned in the caption below (e.g. LSTM optimiser for the first column). The models are tested on unseen figures and the target is to match the transformations based on the example image on Figure (c). Figure (c) was the number that was used to train the models to correct any input images using affine transformations on the initial Experiment 5.



(a) LSTM optimiser (b) Adam (c) SGD (d) RMSprop (e) Example 8 image

Figure 18: Shows the affine transformations applied on a different set of images. The first row shows the input image and the second row displays the image with the affine transformation applied. The models at each column where trained with the optimiser mentioned in the caption below (e.g LSTM optimiser for the first column). The models are tested on unseen figures and the target is to match the transformations based on the example image on Figure (c). Figure (c) was the number that was used to train the models to correct any input images using affine transformations on the initial Experiment 5.

7 Discussion

Our findings shows that a neural optimiser can be successfully implemented in all of the experiment scenarios. First, we showed that when a meta-learner is used to solve basic linear regression or quadratic problems, it does not only improve the model but also outperforms standard optimisers. Furthermore, as more complex structures are used, such as dense layers, we see that the meta-learner performs very well. According to the statistics relating to the testing and validation losses, the LSTM optimiser learns faster. However, the accuracy of the test data shows that the model did not outperform Adam optimiser in any case. Additionally, training the optimiser for longer, does not correlate to improved results. In most cases, training the LSTM optimiser for longer causes the model to "unlearn" as it gives incorrect parameter updates. When more complex scenarios were implemented that contained convolutional layers, the LSTM was struggling to learn and tuning the hyper-parameters to ease training was very complicated and time-consuming. In the original paper [3], a model was presented which was learning to optimise complex models that contained both fully connected and convolutional layers and was also outperforming Stochastic Gradient Descent, RMSprop and Adam.

Our results show that it is feasible to optimise such a network with a neural optimiser, but as demonstrated the model did not outperform the pre-existing optimisers. Several factors could have contributed to this. As mentioned earlier in the paper, building the LSTM optimiser using TensorFlow machine learning framework was challenging. The LSTM optimiser is a neural network itself which uses the optimisee network to train itself. Then the trained LSTM optimiser is used to optimise a newly initialized optimisee network. As a result, training the two networks on top of each other is very challenging and is not a basic use case of TensorFlow. Consequently, it took a considerable amount of time to construct the appropriate computational graphs to allow for training and testing of our LSTM. Furthermore, training was incredibly expensive because each higher-level LSTM training step required the entire training loop of the base network to be unrolled. This adds additional overhead when training even the simplest model, as you must first optimise the LSTM optimiser before using the trained optimiser on the base model for a successful deployment. Contrarily, standard optimisers do not have this overhead, and they are significantly faster. Moreover, when constructing the model for each example, Tensorflow layers such as dense or convolutional, have to be recreated from scratch to allow gradients to flow.

Taken together, the model appeared to be very unstable when handling gradients of different magnitudes. These issues appeared following completion of Experiment 3 when we moved to classification problems. This is commonly seen when training neural networks in machine learning. The meta-learner was unable to learn without gradient pre-processing as discussed in Section 5.2.3. Nevertheless, when gradient pre-processing is applied, the results were dissimilar to the original paper. The LSTM is also significantly affected by its inputs. This could be one of the reasons why the model was unable to reach the expected performance.

Another demanding aspect of the project was the hyper-parameter tuning. The LSTM optimiser is influenced by all parameters and each hyper-parameter affects the overall behaviour of the model. First, the model's activation functions influence its overall performance, as the model learns more effectively using Tanh activation and less than with Relu activation for Dense layers. However, using CNN layers, the LSTM optimiser performs well with Relu activations containing batch normalisation layers, but it is unable to learn or learns poorly when other activations are used. The hidden layers, number of neurons of each layer and learning rate need to be specifically tuned for each example which takes a considerable amount of time. When compared to traditional optimisers, this is more difficult. Finally, the neural network optimiser initially gives large parameter updates on complex models with multiple parameters which might lead the model to a non-reversible state. To address this problem, the model's parameter updates need to be rescaled after each iteration. This can negatively impact the training speed of the model but makes training more stable.

On section 6.6 we have implemented a Spatial Transformer Network to learn how to apply the right affine transformations to make an input image look like a given example image. This was quite challenging for the LSTM optimiser, as it was learning to remove the image, either by scaling it down or applying translation transformation. On the other hand, standard optimisers performed the task correctly with no issues. After successfully training the optimiser, we observed that the results were not as good as the model optimised using Adam. An interesting property observed was that the model was able to generalise well when tested with data that were not used for training. This demonstrates that STNs can be generalised and used for related real-world challenges, where not all input data is formatted in a way that humans or computers can understand.

8 Conclusion

To summarise, the meta-learner mentioned in [3] was re-implemented in this report. Following the implementation, we compared the performance of the LSTM optimiser to that of SGD, RMSProp, Adam and Adagrad using simple linear regression and quadratic problems. Following the initial tests, we demonstrated that learning to learn can be applied to computer vision problems such as image recognition and automated correction of geometric properties. Further, in the scenarios where CNNs and Dense layers were used, we were unable to reach the desired performance because the information in the initial study [3] was inadequate. In the original paper the model is trained on a maximum 1000 steps but most of the plots show that the model is trained for 100 steps. When faced with a large amount of data, the model has several steps per epoch, and if the LSTM is trained for too long, it not only stops improving but also "unlearns". They also proposed rescaling the LSTM's gradient inputs by a constant, but no such constant was given, and it did not seem to improve performance when applied in the experiments above. To finalise, meta-learning is a fascinating field of machine learning, and we have shown that the learning to learn by gradient descent meta learner can be used to optimise neural networks. With the necessary gradient pre-processing and hyper parameter tuning, better performance can be achieved when compared to regular optimisers, but at this moment this comes at a cost: time.

9 Future Work

Despite developing the necessary code and understanding the structure of the meta learner in depth, we were unable to outperform the handcrafted optimisers in the studies conducted in the paper due to model limitations and time constraints. An area worth exploring further for this work is gradient pre-processing. Investigating various gradient pre-processing methods will aid in the handling of gradients of varying magnitudes and therefore will increase accuracy performance. More specifically, investigate gradient updates of convolutional layers, and suggest a new approach to improve performance. Applying the gradient pre-processing (section 5.2.3) for dense layers and a separate gradient update for convolutional layers could be the solution for training more complex models and achieving better accuracy.

Additionally, as there are plenty of computer vision problems that use Generative Adversarial Networks (GANs) to solve a particular problem such as Text-to-Image Translation [35], Face Aging [4], Super Resolution [19] and more. The next step will be to select a modern computer vision problem that uses the GANs architecture in combination with the meta-learner, and investigate whether an LSTM optimiser can be used for that architecture and outperform state-of-the-art optimisers in terms of realism of the result.

Finally, the learning to learn by gradient descent was introduced in 2016. Since then, further research has been conducted on meta-learning, and new methods have been developed. In 2017 Model-Agnostic Meta-learning (MAML) [9] was introduced which is compatible with any model that learns through gradient descent. Additionally, in 2018 Reptile [22] meta learning algorithm was introduced which is another remarkably simple meta-learning optimisation algorithm. Finally, developing more recent meta learning algorithms and comparing their performance and ease of use, to that of hand-crafted algorithms would be an interesting continuation of this research and the field of optimisation meta learning in machine learning.

References

[1] S Agatonovic-Kustrin and R Beresford. Basic concepts of artificial neural network (ann) modeling and its application in pharmaceutical research. *Journal of pharmaceutical and biomedical analysis*, 22(5):717–727, 2000.

[2] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. Ieee, 2017.

[3] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in neural information processing systems*, pages 3981–3989, 2016.

[4] Grigory Antipov, Moez Baccouche, and Jean-Luc Dugelay. Face aging with conditional generative adversarial networks. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 2089–2093, 2017.

[5] Song Cao and Ram Nevatia. Exploring deep learning based solutions in fine grained activity recognition in the wild. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 384–389. IEEE, 2016.

[6] Taco S Cohen and Max Welling. Transformation properties of learned visual representations. *arXiv preprint arXiv:1412.7659*, 2014.

[7] James Dacombe. An introduction to artificial neural networks (with example), Oct 2017.

[8] Nikolaos Doulamis and Athanasios Voulodimos. Fast-mdl: Fast adaptive supervised training of multi-layered deep learning models for consistent object tracking and classification. In *2016 IEEE International Conference on Imaging Systems and Techniques (IST)*, pages 318–323. IEEE, 2016.

[9] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135. PMLR, 2017.

[10] Cloudera Fast Forward. Meta-learning. *Meta-learning.fastforwardlabs.com*, 2020.

[11] Chris A Glasbey and Kantilal Vardichand Mardia. A review of image-warping methods. *Journal of applied statistics*, 25(2):155–171, 1998.

[12] Jiang Guo. Backpropagation through time. *Unpubl. ms., Harbin Institute of Technology*, 40, 2013.

[13] Paul S Heckbert. Fundamentals of texture mapping and image warping. 1989.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] D.H. House and J.C. Keyser. *Foundations of Physically Based Modeling and Animation*. CRC Press, 2016.

[16] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. Spatial transformer networks. *arXiv preprint arXiv:1506.02025*, 2015.

[17] M. I. Jordan and T. M. Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[18] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[19] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. Photo-realistic single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4681–4690, 2017.

[20] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[21] Aditi Mittal. Understanding rnn and lstm, Oct 2019.

[22] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

[23] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE international conference on computer vision*, pages 1520–1528, 2015.

[24] University of Toronto. affnist.

[25] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[26] Zalando research. `http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/`, 2021.

[27] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[28] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[29] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[30] Hessamodin Teimouri. *A new statistical approach to strain-based structural health monitoring of composites under uncertainty.* PhD thesis, University of British Columbia, 2015.

[31] Xingyou Wang, Weijie Jiang, and Zhiyong Luo. Combination of convolutional and recurrent neural network for sentiment analysis of short texts. In *Proceedings of COLING 2016, the 26th international conference on computational linguistics: Technical papers*, pages 2428–2437, 2016.

[32] Zhou Wang, Eero P Simoncelli, and Alan C Bovik. Multiscale structural similarity for image quality assessment. In *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, volume 2, pages 1398–1402. Ieee, 2003.

[33] Wikipedia. Bilinear interpolation. *En.wikipedia.org*, 2009.

[34] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[35] Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, and Dimitris N Metaxas. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 5907–5915, 2017.

10 Appendix

Appendix-A Meeting Logs

Meeting #1 Date 08.09.2019 Preliminary meeting where we talked about the different possible projects and areas of exploration. Also talked about the structure of a final year project.

Meeting #2 Date 28/09/2020 Discussed in more detail what the project is about and how to proceed going forward. Furthermore, we talked about the different deadlines and what is expected from me. Finally, Ivor has guided me to some to some literature to study and understand the topic better.

Meeting #3 Date 12/10/2020
Discussed the details of the proposal submission. Discussed about the different applications of image warping and what could make an interesting project.

Meeting #4 Date 4/11/2020
In this meeting we have outlined the main points of the interim report. We have concluded that the sample structure provided on Sussex direct for interim reports is not very well suited for this type of projects. In addition to the main points outlined in the interim report structure, we have agreed to include a section regarding the technical background where I will analyse three image warping research papers discussing their approach, implementation and outlining the loss function. Furthermore, we discussed adding an Initial analysis section where I will highlight everything I have researched so far regarding the meta-learning algorithm and give a description of what is used in the algorithm. (Gradient descent, artificial neural networks, RNNs and LSTM)

Group discussions
After the interim report submission in November until the end of the term we were having group calls discussing our progress with a number of other students under Ivor's supervision.

Meeting #5 Date 8/12/2020
In this meeting we have discussed the different applications of possible computer vision projects that would be good candidates for testing the LSTM meta learner. Also, we had a discussion regarding the project's progress during Christmas and Assessment period.

Meeting #6 Date 29/1/2021
Discussed progress made over Christmas and discussed a plan of how to proceed successfully.

Meeting #7 Date 1/3/2021
In my own time, I have completed a more advanced Tensorflow course in parallel to completing project in order to be able to create the model as described in the paper. The implementation of it is not just a simple use case of TensorFlow therefore I was required to acquire some additional knowledge to develop that. Additionally, we discussed in more detail the implementation of the meta learner and different problems we could apply it to.

Meeting #8 Date 10/3/2021
Displayed and discussed the implementation of the LSTM optimiser based on a simple use case. We

have outlined potential improvements and planned the next use cases of the optimiser.

Meeting #9 Date 23/4/2021
Discussed progress of the report and how to best wrap up the research for submission

Appendix-B LSTM-Optimiser Additional Tests

1. Generalisation from Tanh to ReLU

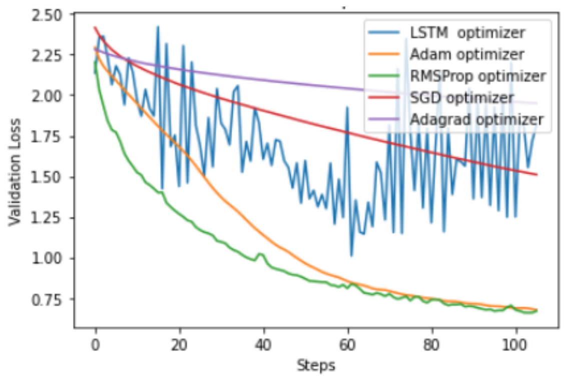


Figure 19: Validation losses of handcrafted and LSTM optimisers on Fashion-MNIST dataset. The LSTM optimiser was trained using a model with Tanh activation function and then it was used to train a model using ReLU activation function. The model consisted of 1 dense layer and 20 hidden units. The results show that the LSTM optimiser is not able to generalise and adapt on ReLU activation function due to the different outputs of the activation functions.

2. Generalisation from Tanh to Sigmoid

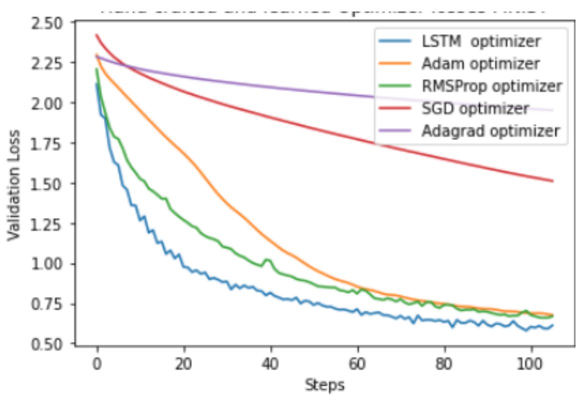


Figure 20: Validation losses of handcrafted and LSTM optimisers on Fashion-MNIST dataset. The LSTM optimiser was trained using a model with Tanh activation function and then it was used to train a model using Sigmoid activation function. The model consisted of 1 dense layer and 20 hidden units. The results show that the LSTM optimiser is able to generalise and adapt very well from tanh to sigmoid activation function and even outperform traditional optimisers.

3. Generalisation from batch size 512 to 128

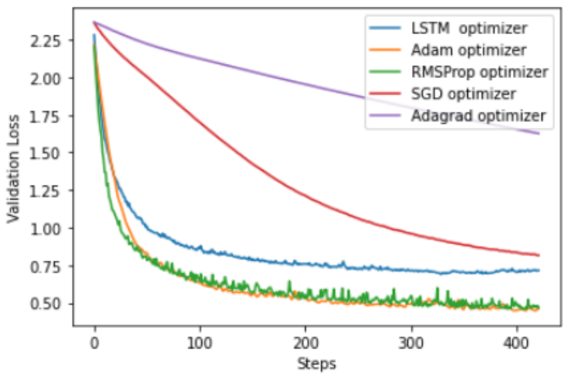
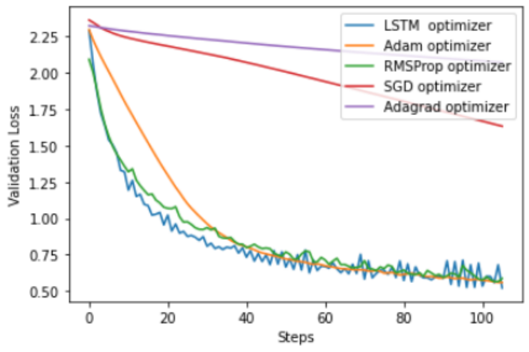
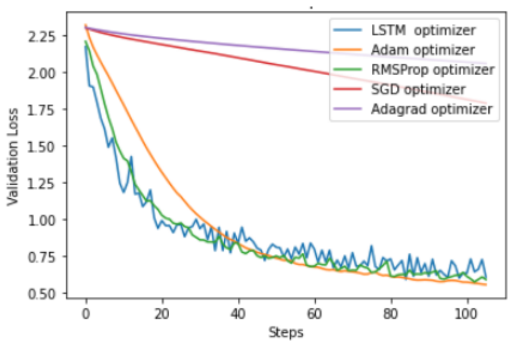


Figure 21: Validation losses of handcrafted and LSTM optimisers on Fashion-MNIST dataset. The LSTM optimiser was trained using a model with Tanh activation function and a batch size of 512 data per step. Then the trained optimizer was used to train a model using a batch size of 128. The model consisted of 2 dense layers and 20 hidden units. The results show that the LSTM optimiser is able to generalise well on batch size changes. However, the LSTM optimiser has not outperformed the handcrafted optimisers in this experiment.

4. LSTM optimizer training unrolling hyper-parameter



(a) LSTM Optimizer was unrolled for 10 steps



(b) LSTM Optimizer was unrolled for 30 steps

Figure 22: Validation losses of handcrafted and LSTM optimisers on Fashion-MNIST on a model consisting of 2 dense layers of 20 hidden neurons each. In Figure (a) the LSTM optimizer was unrolled for 10 steps and in Figure (b) the LSTM was unrolled for 30 steps. The results show that unrolling the optimizer for less steps has an outcome of smaller updates per epoch per each epoch.