LispNet: A machine learning framework for Petalisp and its application to Multigrid PDE

Michael Holzmann
LispNet: A machine learning framework for Petalisp and its application to Multigrid PDE

Michael Holzmann
Masterarbeit

Aufgabensteller: Prof. Dr. H. Köstler
Betreuer: M. Sc. M. Heisig
Bearbeitungszeitraum: 20.01.2022 – 20.07.2022
Erklärung:
Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Masterarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 22. Juni 2022

...................................................
Contents

1 LispNet - A machine learning framework for Petalisp 6
  1.1 Motivation .......................................................... 6
  1.2 Machine learning .................................................. 6
    1.2.1 Dense networks ............................................. 6
    1.2.2 Cost function and gradient descent ...................... 7
    1.2.3 Backpropagation ........................................... 7
    1.2.4 Convolutional networks .................................... 9
    1.2.5 Other network types ...................................... 9
  1.3 Petalisp ............................................................ 10
    1.3.1 Common Lisp ................................................. 10
    1.3.2 Design and concept ....................................... 12
    1.3.3 Automatic Differentiation ................................ 15
  1.4 LispNet ............................................................ 16
    1.4.1 Design and architecture .................................. 17
    1.4.2 Layer implementation ..................................... 21
    1.4.3 Hello World with MNIST .................................. 23
    1.4.4 Performance ............................................... 25
    1.4.5 Python interface .......................................... 26

2 Application to Multigrid PDE with jump coefficients 29
  2.1 Introduction ...................................................... 29
    2.1.1 Poisson equation .......................................... 29
    2.1.2 Jacobi method ............................................. 30
    2.1.3 Jump coefficients ......................................... 30
    2.1.4 Multigrid .................................................. 31
  2.2 The machine learning approach ................................. 33
    2.2.1 Related work .............................................. 33
    2.2.2 Dataset .................................................. 35
    2.2.3 VCycle implementation .................................... 37
    2.2.4 Coefficient Restriction ................................... 40
    2.2.5 Attention Coefficient Restriction ...................... 42
    2.2.6 Cost function and training .............................. 44
    2.2.7 Qualitative Results ..................................... 46
    2.2.8 Quantitative Results .................................... 48

3 Conclusion 51
Abstract

Machine learning became one of the most popular tools to solve and analyze various kind of data problems. Frameworks like TensorFlow are commonly used for implementation. These frameworks often lack direct integration with large High Performance Computing systems. In this work we present LispNet: a machine learning framework for Common Lisp, which is built on the fast HPC library Petalisp. We show design, concept and implementation of LispNet and all of its features. As a Proof of Concept we use this framework to optimize Multigrid algorithms for partial differential equations. Multigrid methods like V-Cycle are efficient tools for solving PDEs. Furthermore, we add many different complex jump coefficients to the equation leading to a harder and often not solvable problem for native Multigrid approaches. We modify the algorithm by mapping the jump coefficients to coarser grids in a subroutine which is named "Coefficient Restriction". We describe each mathematical operation of Multigrid as layer of a neural network. We apply an unsupervised machine learning method to optimize this ansatz. Our results shows, that these models are able to solve PDEs with all jump configurations in the test set. Furthermore, trained models show significant convergence improvements. We also show, that we can improve this method by adding attention gates to the restriction process. Due to the attention mechanism our models use local jump information to learn optimized restriction weights for specific type of configurations.
1 LispNet - A machine learning framework for Petalisp

1.1 Motivation

In the today’s digital world machine learning has application to a various number of research topics and is the key to create intelligent and smart products in Industry 4.0. Some of the greatest use cases of machine learning are speech and image recognition such as medical cancer detection, face analysis and object detection [1, 2]. With the rising scale of data and increasing performance of CPU and GPU machines several machine learning methods were developed. In 2017 Google’s DeepMind published AlphaGo Zero. Without data from humans games it was trained to play the game GO by playing against itself. A cluster of 64 GPUs and 19 CPUs were used during training [3]. This is one example of machine learning running on a High Performance Computing (HPC) cluster. In 2019 Chien et al. showed in their work "TensorFlow Doing HPC" [4] advantages and disadvantages of using the popular machine learning framework TensorFlow on HPC networks. Due to Python’s low performance of numerical computation the performance can drop significantly when algorithms are implemented partly in TensorFlow and native Python arrays. Often it is difficult to implement such logic as TensorFlow graph expression especially when reductions are a part of a subroutine. In a master-slave environment merging the results of the worker threads are time consuming in Python and can result in bottlenecks. In this work these problems get addressed by creating a machine learning framework on a HPC framework which natively supports language tools like reductions. The following sections shows the machine learning framework "LispNet" which runs on Petalisp, a high performance language extension to Common Lisp.

1.2 Machine learning

In this chapter we introduce the concepts of machine learning, neural networks and the process of training. Machine learning algorithms can be divided in three types:

Supervised Learning: This algorithm is applied to a problem where a set of desired output variables are given. A cost function in terms of accuracy is built on top of predictions and given labels. Supervised Learning minimizes the cost function during the training process. Common algorithms of this category are Regression, Random Forest and Decision Trees.

Unsupervised Learning: Unsupervised learning does not require any outcome variable. It clusters specific types of groups in a data set. Some use cases are DNA pattern clustering, customer group segmentation and the detection of anomalies in big data. K-means is one of the most popular algorithm of this type.

Reinforcement learning: A machine is trained on making specific decisions by using trial and error. Based on its decision a reward is calculated and the algorithm tries to maximize it by adapting its decision making. Markov Decision Chains are one example.

Neural networks have the capability to perform all three of these types.

1.2.1 Dense networks

In the following section we explain the concept of neural networks with dense layers. We introduce a network with K layers, where the first layer of a network is the vector $x^1$ and $x^K$ the output.
Each layer $k$ consists of $N$ neurons. A dense layer takes an input $x^{k-1}$ and maps it to $y^k$. The first dense layer takes the input of the network. The output of a dense layer is calculated in this way:

$$y^k_i = \sum_j w^k_{ij} x^{k-1}_j + b^k_i$$  \hspace{1cm} (1)

$w_{ij}$ and $b_i$ describes trainable parameters where $w_{ij}$ is considered as the weights of the dense layer and $b_i$ as the bias. The operation $w_{ij} x^{k-1}_j$ describes a matrix vector multiplication where $w$ has the size $N^k \times N^{k-1}$.

In addition an activation function $f$ can be applied to the output.

$$x^k = f(y^k)$$  \hspace{1cm} (2)

These activation functions can be non linear, which means that networks could consist of linear and non linear transformations. One of the most common functions are "Sigmoid", "ReLu", "TanH" and "Softmax". Based on Universal Approximation theorem a neural network can approximate any function when it has at least one hidden layer and enough neurons are used [5]. However, this is only valid when activation functions can be non linear. Otherwise a dense network could only approximate a linear transformation.

1.2.2 Cost function and gradient descent

For supervised learning we need a cost function which takes the output of the network $x^K$ and given label data $\tilde{x}$ and calculates an error. The goal of training a neural network is to minimize this error. We consider a network as a function of $w$ and $x^1$ where $w$ is a set of trainable parameters $w_j$. A common cost function is mean squared error (MSE):

$$C_w(\tilde{x}) = \frac{1}{2} || F_w(x^1) - \tilde{x} ||^2$$  \hspace{1cm} (3)

Minimizing this function is an optimization problem where $F(w, x^1)$ has to be changed by adapting the weights of the neural network. After the forward pass we update every trainable parameter $w_j$ by means of an optimizer. A common one is gradient descent:

$$w_j^{t+1} = w_j^t - \eta \frac{\partial C_W(\tilde{x})}{\partial w_j^t}$$  \hspace{1cm} (4)

where $t$ describes the number of update iterations and $\eta$ the learning rate. Thus we updated our weights based on the gradient of the cost function. In practise we rather calculate the gradient of a small input batch than of the true gradient. Therefore stochastic gradient descent with exponential weighted gradient averages work empirically better. By saving the update step of the last optimization we can integrate momentum:

$$v_j^{t+1} = \beta v_j^t + \eta \frac{\partial C_W(\tilde{x})}{\partial w_j^t}$$  

$$w_j^{t+1} = w_j^t - v_j^{t+1}$$  \hspace{1cm} (5)

where $\beta$ is the momentum parameter. Momentum helps to speed up the optimization and further can prevent local minima of the cost function. Choosing the right momentum and learning rate is often quite difficult. $\eta$ controls the rate of speed a neural net learns. If $\eta$ is too high we miss the global minimum and divergent behaviours occur. If $\eta$ is too small the networks slowly converges to the minimum which results in a longer training time. Other optimization algorithms like RMSProp, Adagrad and Adam adapt the learning rate for each weight individually during training. Hence networks trained with adaptive optimizers converge in general faster than training with classic stochastic gradient descent.

1.2.3 Backpropagation

backpropagation is an algorithm to propagate the error from back to front layers. Since we want to optimize a dense layer we need $\frac{\partial C}{\partial w_{ij}}$ and $\frac{\partial C}{\partial b_i}$. We consider layer $k$ as a hidden layer. The derivatives
can be calculated with the chain rule:

\[
\frac{\partial C}{\partial w_{ij}^k} = \frac{\partial C}{\partial y_i^k} \frac{\partial y_i^k}{\partial w_{ij}^k} = \delta_i^k \frac{\partial y_i^k}{\partial w_{ij}^k} = \delta_i^k x_{-j}^{k-1}
\]

(6)

\[
\frac{\partial C}{\partial b_i^k} = \frac{\partial C}{\partial y_i^k} \frac{\partial y_i^k}{\partial b_i^k} = \delta_i^k \frac{\partial y_i^k}{\partial b_i^k} = \delta_i^k
\]

(7)

\(\delta_i^k\) is commonly called "error signal" of a neuron.

Further \(\delta_i^k\) can be described as:

\[
\delta_i^k = \frac{\partial C}{\partial y_i^k} = \frac{\partial C}{\partial x_i^k} \frac{\partial x_i^k}{\partial y_i^k} = \frac{\partial C}{\partial x_i^k} (\sum_j \frac{\partial C}{\partial y_{j+1}^k} \frac{\partial y_{j+1}^k}{\partial x_i^k}) f'(y_i^k) = (\sum_j \delta_j^{k+1} w_{ji}^{k+1}) f'(y_i^k)
\]

(8)

As shown in equation 8 \(\delta_i^k\) requires the error signal of its next layer \(k + 1\). Note the sum \((\sum_j \delta_j^{k+1} w_{ji}^{k+1})\). Since a neuron of layer \(k\) effects all neurons in layer \(k + 1\) it has to sum up all error signals. In terms of a matrix form the error of next layers has to propagate to previous layers by multiplying the transposed weight matrix with its error signal vector \(\delta_j^{k+1}\):

\[
\delta_i^k = (w^{k+1})^T \delta_j^{k+1} \odot f'(y_i^k)
\]

(9)

where \(\odot\) describes an element-wise multiplication. Since the error signal of the next layer is needed back propagation has to start its algorithm at the output layer. Here we need the error signal of the last layer:

\[
\delta_i^K = \frac{C}{\partial y_i^K} = \frac{\partial C}{\partial x_i^K} \frac{\partial x_i^K}{\partial y_i^K} = \frac{\partial C}{\partial x_i^K} f'(y_i^K)
\]

(10)

Thus the first step of back propagation is to calculate the derivative of the cost function \(\frac{\partial C}{\partial x^K}\). In case of loss function 3 the gradient \(\nabla C\) is \(x^K - \hat{x}\).

In short, training a fully connected network with backpropagation follows these steps:

**Algorithm 1** Fitting a dense network with backpropagation

- Forward pass: compute \(x^k\) for \(k = 2, \ldots, K\)
- Compute output error signal \(\delta_i^K\)
- Propagate the error through all layers \(\delta_i^k\) for \(k = 2, \ldots, K - 1\)
- Compute the gradient for each trainable parameter \(w_j\) according to function 6, 7
- Update \(w_j\) via gradient descent

The equations above show a neural network working on a single sample. However, we also can consider the input as a batch of \(m\) input vectors. Instead of handling layers of size \(n\) as vectors we can describe them as a matrix with size of \(m \times n\). The forward pass of a dense layer with batch size \(m\) is:

\[
y^{m \times k} = x^{m \times n} W^{n \times k} + b^k
\]

(11)

Since this method generates multiple outputs the loss function and error \(\delta^K\) has to be averaged through all samples of a batch. The big difference here is that all weights are updated with an average gradient over \(m\) samples. A batch of samples instead of one helps to speed up the parallel efficiency during training. However, when the batch size is too large the network lacks of generalization. Gradients of smaller batch oscillate more then bigger ones. This can be considered as noise. This helps the network to come out of local minima while optimising the loss function.
1.2.4 Convolutional networks

The big problem with dense layers is, that same patterns in the input get handled differently based on its position in the input space. The reason for this is that all neurons are weighted independently. This makes dense layers for image recognition less effective. Furthermore, dense layers consist of a lot of trainable parameters \( (N^k \times N^{k-1}) \). Most commonly types of networks are Convolutional Neural Networks (CNN). Their concept is based on the visual human cortex. The convolutional layer performs a discrete convolution with the input and a trainable kernel \( w \). We consider a two-dimensional input with \( c \) channels \( x_{i,j,c} \) and a \( 3 \times 3 \times c \) filter \( w \). The forward pass of a two-dimensional discrete convolution is:

\[
y_{i,j} = \sum_c \sum_{y=-1}^{1} \sum_{x=-1}^{1} x_{i,j,c} w_{y,x,c}
\]

for the boundary case two convolution types normally are applied: "Valid", which only convolutes the inner space of the input, and mode "Same", which performs a convolution by padding its input space. The equation 12 describes the convolution of one filter. Normally multiple filters are used. Thus the output can be written as \( y_{i,j,m} \) with the usage of \( m \) different filters. Commonly it also called "feature maps". In addition a bias and an activation function can be applied to the output. The big advantages of convolutions are detection of edges and other space related patterns. The figure 2 shows a deep convolutional neuronal network. Its design allows it to learn specific patterns of the input to generate high and low order feature maps. The first layers detect various kinds of edges. The dimensional space of the features are then reduced by a pooling layer due to the exponential increase of features in these networks. Popular pooling operations are "Average Pooling", "Max Pooling", and "Global Pooling". Furthermore, the pooled features of edge detection are then again getting convoluted. This generates a higher order type of features where it detects different shapes like squares and circles. At the end the feed forward process allows the network to detect various complex types of features. These then are combined by a dense network to train the network to recognize specific image regions by its high level feature maps. Convolutions layers therefore are the most effective tools for image pattern recognition. In the last ten years several deep convolutional neural network (DCNN) architectures were developed. In 2010 the ImageNet challenge was launched. ImageNet is a dataset with more than 14 million images. The goal of this contest is to correctly classify and detect objects. In 2012 a convolutional neural network named "AlexNet" achieved a top-5 error of 15.3%, which was huge step up at this time [6]. The best model today achieves more than 90% top-1 accuracy [7]. CNNs have been applied to many research fields and show that they are capable of outperforming humans.

1.2.5 Other network types

In the previous section we discussed simple feed forward networks. By the means of non linear activation functions feed forward networks can suffer from the exploding or vanishing gradient problem during backpropagation. Residual Networks introduce a shortcut between an input of an layer and its output. In its most simple way the input is just added to the output. This allows the gradient to use a shortcut which empirically improves the training of deep networks. Furthermore, these skip connections ensemble multiple paths for each layer which results in a exponential number of paths.
with different length. This concept is used in many network architectures like ResNet, InceptionNet, AttentionNets and U-Net.

Another type of networks are Recurrent Neural Networks (RNN). RNNs allow to feed the output of a layer to its input. This works by saving the output as a hidden state and use that for the calculation for the next forward pass. These hidden states can be considered as memory cells of a network. The hidden state therefore represents a history of different inputs and can help to remember earlier states. Thus patterns of multiple correlated input samples can be learned. Recurrent networks find its application to various fields such as speech recognition, gesture recognition and text prediction. However, training simple recurrent cells can lead to vanishing gradients since the effect of previous input decays exponentially. A different approach are Long-Short-Term-Memory layers (LSTM). By adding an additional memory cell and specific gates, which decide whether information is used or not, long term memory and dependencies can be learned.

There are a lot of other layer types and network architectures. In fact finding the right architecture for a specific task seems quite difficult. Often many different network architectures are working together. Modular Neural Networks(MNN) are combinations of multiple network types. The concept is to assign each network to a module which takes a different part of the input. Each network accomplishes a subtask and their result is then merged together.

In this work we focus on feed forward dense, convolutional and residual networks.

### 1.3 Petalisp

In this chapter we first introduce Common Lisp, on which Petalisp runs on and then explain the concept, design and interface of Petalisp. Note, that this framework is constantly further developed and maintained. Thus the framework and its interface may change in the future. At the end we show how Automatic Differentiation is made in Petalisp, which is an important subroutine for LispNet.

#### 1.3.1 Common Lisp

In 1958 John McCarthy published in his work "Recursive Functions of Symbolic Expressions and Their Computation by Machine" [8] the concept of Lisp. The programming language has been developed for the IBM 704 by A.I. Group at M.I.T. The name Lisp originally stands for "LISt Processor". Linked-Lists are the main data structure of this language. In fact Lisp's code is mainly represented as lists. The concept of Lisp is based on S-expressions. In S-expression we do not distinguish expressions and statements. S-expressions are either atoms or expressions in the form of cons cells (x . y) where x and y are S-expressions. This recursive definition allows Lisp to be a functional Turing-complete language. Cons cells consist of a reference to its left value (car field) and its right value (cdr field). A list in Lisp is implemented as a series of cons cells. i.e the list [1 2] can be described by cons cells: (1 . (2 . NIL)). Thus lists are implemented on top of cons cells. A list is either a empty list (NIL) or a cons cell where the car field points to the first element and the cdr field references to the rest. An atom is everything which is not a cons cell. Atoms for example can be numbers, symbols, strings... The empty list is defined as an atom too and is described as "NIL" or "()" and often used as "null" or "false" like in other programming languages. Due to the specific grammar of this language Lisp programs can be considered as Nested-Linked-Lists. The language uses prefix notation. For instance a function, which squares its input and adds the constant 3, can be defined according to figure 3. To evaluate this function you can call: (foo arg1). The code itself

```lisp
(defun foo (x)
  (+ (* x x) 3)
)
```

Figure 3: Function definition in Lisp

is described as a nested list of the atomic variables: defun, foo, x, +, * and 3. Code and data are handled in the same way. This allows Lisp to implement the concepts of metaprogramming.

As a result several dialects of Lisp were developed. The most popular ones are Scheme and Common Lisp. Common Lisp extends the Lisp language by several macros and functions. The difference
between macros and functions are, that macros get evaluated before they get passed to the interpreter. With the means of macros several language features like loops, conditions are implemented on Common Lisp.

```lisp
1 (loop for x in '(a b c d)
2       for y in '(1 2 3 4)
3       collect (list x y))
```

```lisp
1 (defmacro notif
−
2   then
−
252 (cond ((not , condition ) ,then)
283 (t , else ))
147 (t , else )))
```

Figure 4: Loop macro in Common Lisp and custom macro definition

In the example of figure 4 the loop macro produces a level 2 nested list. The two for macros iterate here through both lists at the same time each time pulling an element out of the list. Both values then get combined to a list and collected and added to the result list. The evaluation would be '((a 1) (b 2) (c 3) (d 4)). In the other code we see a definition of a macro. It takes a condition expression, a then and else expression. It returns the function cond which takes a list of condition and actions pairs. The evaluation of cond results in the first action of the list where its conditions is true. Therefore we simply can implement a if-then-else with a macro. In this example the logic is changed to not if-then-else by applying the not function to the first condition. The beauty of macros is, that they allow to redefine and extend functionalities in a domain specific language. In fact Common Lisp has a lot of more features regarding control flow than other programming languages. The flexibility of Lisp requires also a non static type system. Lisp therefore supports static and dynamic type checking. Thus type safety is still covered in Lisp. Types of objects and errors are just evaluated at runtime. In modern Common Lisp implementations such as SBCL a developer can easily debug and analyze objects and their content. The stack trace can be inspected too whenever such errors are thrown.

Another key feature of Common Lisp is the Common Lisp Object System (CLOS). It extends Lisp with object-orientated language features. The idea behind object orientation is to design data types or classes and describe how operations work on them. We introduce now the concepts of classes, instances and generic functions. These are the main object-orientated futures used in this work. A class describes attributes of an object type. A developer can define new classes with the macro defclass. Figure 5 shows an example of class definition. Attributes are defined by a set of slots. Here class animal has slots name and age. To create an object of this class you can call the function (make-instance 'animal). The instance consists of all properties defined by the class. Accessor functions can be defined to read and write to these attributes. Thus the slot values of an instance can be modified. To be specific to change the value of slot name you can call: (setf (name animal-obj) name-value). CLOS also supports single and multiple inheritance. The class dog and cat are declared as sub-classes of animal. They inheritate all properties of the upper class. We can define methods which interact with objects of these classes. In this example we want to define an abstract method for an animal "make-sound". The sub classes dog and cat should implement this method and print its type specific sound. The declaration of abstract methods is done via generic functions. Generic functions declare the method name and parameters but do not implement the method body. When a generic function is called with a specific type, the interpreter invokes the implementation of this method for this class. Therefore complex interfaces
can be defined by the means of generic functions. Common Lisp also supports to replace and modify already defined functions. We can add method qualifies to already existing methods. With the keyword :before, :after or :around we can invoke additional code when a method is called. For example :after can be used to execute code after a method. A common usage of this are constructor calls. The call make-instance invokes the function initialize-instance. We can define constructor calls by extending initialize-instance with the keyword :after. Figure 7 shows the constructor definition of class cat. It is common to initialize slots values there to describe the initial state of an instance. As now we explained the main concepts of Common Lisp all further Lisp code segments in this work should be able to be understood. In fact Common Lisp supports a lot of other cool features which are not mentioned in this work. The books [9, 10, 11] show a good explanation of more features in Common Lisp.

1.3.2 Design and concept

To summarize Petalisp in one sentence it can be described as an elegant High Performance Computing (HPC) extension to Common Lisp [12, 13]. Its main purpose is to perform mathematical tasks very fast on parallel machines. One may it call an HPC framework similar to the python’s "Numpy" with extreme optimization and parallelization. Petalisp is developed by Marco Heisig at the Friedrich Alexander University. We first introduce the main concepts of Petalisp. Petalisp’s main feature is lazy evaluation. In contrast to eager evaluation computations are then performed when they are needed. The idea is to create an abstract computation graph which then can be evaluated by the function "compute". Data structures are needed to save operations and dependencies. Petalisp then optimizes the computation graph on a parallel machine. In fact a user does not need to use OpenMP or MPI interfaces to make use of parallel efficiency. Petalisp provides an abstract language to perform these tasks. Figure 8 shows the main structs and classes of Petalisp core in a simplified way [14]. The main data structure of the interface are lazy-arrays. Unlike normal arrays
lazy-arrays do not store values of an array. The struct describes the dimensions of the array by a
shape, the datatype of its elements and a reference to an delayed action. A shape is a Cartesian
product of a set of ranges. A range can be defined by its start, size and a value describing the step
size. i.e. indices (0, 2, 4) can be described by the function (range 0 5 2). A shape of an lazy-array
describes each dimension with a range. To calculate the lazy-array the delayed action is needed.
This action describes the application of functions on lazy-arrays. Petalisp supports several types of
delayed actions. Each type of function in the computation graph is described as a delayed action
and is implemented by a specific struct. In the following we list and explain the main delayed
actions and their usage in Petalisp. We consider variable A and B as normal arrays and describe A
and B as their corresponding lazy-arrays referencing to A and B.

**delayed-array** This action refers directly to an array. It contains the values as a storage slot.
This operation basically loads normal arrays into the computation graph. Constants are
also implemented this way by loading a scalar matrix. Thus: \( \hat{A} \leftarrow \text{delayed-array}(A) \).
The corresponding Petalisp API call is: (lazy-array A)

**delayed-map** With delayed-map we can apply a specific operator element-wise to the input lazy-
arrays. It stores the operator and a list of inputs. For instance, we can calculate the sum of
two matrices: \( \hat{(A + B)} \leftarrow \text{delayed-map}(+, \hat{A}, \hat{B}) \). This action can be invoked by the interface
function lazy: (lazy #'+ \hat{A} \hat{B})

**delayed-reshape** This operation takes an lazy-array and a transformation as input. It then ap-
plies this transformation on the indices of the input array. In other words it applies a function
which describes which source index maps to the output index. By that we can apply re-
shaping, striding, slicing and index translations. Petalisp offers several tools to create such
transformations. For instance we can transpose a matrix:

\[ \hat{(A)}^T \leftarrow \text{lazy-reshape} \hat{A} \text{ (transform i j to j i)} \]

Note the index mapping has to be a linear function and has to be invertible by default.

**delayed-fuse** A fusion takes a list of inputs. The indices of the output array correspond to the
indices of the input arrays. A error is thrown, if there is an intersection of input indices. A
common call for delayed-fuse is the Petalisp API call lazy-fuse:

(\text{lazy-fuse} (\text{lazy-reshape} 1 (∼ 0 10 2)) (\text{lazy-reshape} 0 (∼ 1 10 2))). This example generates a
chess like pattern with zeros and ones.

**delayed-unknown** Unknown actions are not runnable. They represent a reference to a lazy-array
which describes just a formal parameter. With unknown arrays we can generate computation
graphs without having normal arrays. Thus we can work on graphs without data which is a
key subroutine for creating parameterized graphs like neural networks.

**delayed-nop** A delayed-nop is the delayed action of an empty lazy-array.

**delayed-range** With this action an lazy-array can be created where its values equals their index
in the array.

**delayed-wait** Wait can be called to join asynchronous execution paths. This struct holds the
request and a specific delayed-action. The action gets executed when the request is completed.

**delayed-failure** This action is triggered when a serious error occurs in an asynchronous execution
path. Common errors are Zero Division and floating point under/overflows.

As we explained lazy-arrays are the representative data structure of arrays and a delayed actions
are application of functions on these arrays. Thus Petalisp generates an acyclic directed graph
where nodes can be considered as lazy-arrays and delayed-actions. To be more specific we consider
following Petalisp code: (\text{lazy #'+} A \ 1.0). The code performs and element-wise increment on a two-
dimensional matrix A. Figure 9 shows the objects Petalisp instantiates. First two lazy-arrays are
created referencing to Array A and a constant scalar matrix via delayed-array action. With delayed-
reshape the constant 1 is reshaped to the array shape of A. Lazy map then performs an element-wise
addition of the two resulting lazy-arrays. The output of this graph again is a lazy-array which can be calculated by the call `compute`. It is important to mention that the computation structure is not representative for the actual calculations and memory allocations. For instance Petalisp does not reshape the constant to the size of the A. This would be unnecessary, since all elements in this matrix would have the same value. In fact Petalisp does extreme optimization and parallelization before the graph is executed on a machine. To analyze what is actually computed Petalisp provides several tools to show the Intermediate Representation (IR). The IR shows a bipartite directed graph, where the nodes describe allocated buffers and kernels. Buffers are arrays the Petalisp backend is using between computations. Kernels are parallel programs which are executed by the CPU. The kernel operation is automatically pararellized for multiple cores. Thus a user does not need to worry about any thread implementations. Each kernel operation results in a buffer allocation, which leads to alternating dependencies between kernel and buffers. As shown in Figure 10 the IR for this example consists of three buffers and one kernel. A buffer stores allocated memory, its shape and type. Petalisp supports dynamic and static buffer allocation. The kernel is defined by an iteration space and set of instructions. Since we just add the constant 1.0 to each element, the iteration/output space equals the input space. Instruction 0 loads the matrix elements into the register. Note variable A and B of the transform call describe the indices of the iteration space. Instruction 1 loads the constant which is added to the just loaded elements in instruction 2. The result of this mapping is then saved in another buffer at the end. The beauty of IR is that a developer directly sees what is actually done under the hood. Parallelity comes without handling threads or different processes. This make Petalisp an an advanced framework for large
scale computations like machine learning.

Another important feature of the Petalisp API are reductions [15]. Reductions often are described as folds of functions on a data structure. Here lazy-reduce takes a lazy-array of rank \(N\) and a function as input. The result is a lazy-array with rank \(N - 1\). It folds the function over the first dimension of the input. For instance (lazy-reduce \('+\', \(A\)) adds all elements along axis 0. Reductions are implemented as a simple recursive function. It follows a divide and conquer algorithm where the arrays get divided into a left and right half. Reductions are recursively applied to each half until the size of the dimension is 1. The results of the recursive reductions are then merged together and evaluated by the function, i.e. the reduction of a dimension with size \(2^n\) generates a computation tree of depth \(n\). To get better parallel efficiency Petalisp optimizes this tree by merging several reductions by one kernel. Figure 11 shows the IR of an addition reduction of a vector with size 128. It can be seen, that in the first kernel 8 blocks consisting of 16 values are reduced. Thus the iteration space and the output buffer have the size 8. These 8 values then are added together in a second step. A bigger block size results in a smaller iteration space and less memory allocations. Thus Petalisp tries to create big instructions kernels. However the number of instructions in a kernel is limited at 32 by default. For too large kernel sizes several problems occur. Not all kernel data fits into registers and caches which leads to cache misses and load instructions. Therefore there is a sweet spot between kernel size and iteration space. Petalisp automatically tries to find the right configuration in terms of parallelization and memory consumption. Thus computation graphs may differ on different computers. Due to the divide and conquer concept reductions lead to graphs with a logarithmic length respect to the dimension. Big data of size \(N\) can be consequently reduced in \(\log(N)\) steps. Another cool feature of Petalisp reductions is, that it can reduce custom functions via lambda functions. Like lazy-map reductions also support multiple return values of a function. For instance, a user can reduce a vector by a max function and also returns the index directing to its maximum entry. This allows developers to create very complex and different reductions which, can be hard to implement in other frameworks like TensorFlow.

\subsection{1.3.3 Automatic Differentiation}

The main advantage of building an computation graph with lazy-evaluation is, that it can evaluate the differentiation of lazy-arrays. This is a very important key feature for LispNet. Through Automatic Differentiation (AD) a machine learning developer only has to define the forward-pass and the backward-pass is created by default. To understand how Petalisp does differentiation we
consider a simple differentiation example. Let \( f \) and \( g \) be differentiable functions and \( w, b \) scalar trainable parameters.

\[
f(x) = wx + b = y \\
g(x) = x^2 \\
g(f(x)) = y^2 = z
\]

(13)

The gradients \( \frac{\partial z}{\partial w} \) and \( \frac{\partial z}{\partial b} \) are calculated via chain rule:

\[
\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial w} = 2y \cdot x \\
\frac{\partial z}{\partial b} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial b} = 2y \cdot 1
\]

(14)

This is equivalent to backpropagation explained in section 1.2.3. Here \( \frac{\partial z}{\partial y} \) describes the error signal \( \delta^k \) of node \( y \). As in Figure 12 shown the computation of \( z \) can be considered as a tree of height 2, where \( w \) and \( b \) are leaves. The differentiation of these parameters follows these steps:

\[
\text{Figure 12: Differentiation graph via chain rule}
\]

- Find all paths from root node to the parameter via Depth-First search.
- Calculate all gradients \( \frac{\partial \text{parent}}{\partial \text{child}} \) along these paths. The gradient of the root node can be considered as input gradient. This gradient equals 1.
- For each path multiply all gradients along its path. If multiple gradients direct to the same node they are added together.

In order to calculate the gradients and to propagate the errors through this graph each delayed-action has a function defined to differentiate its inputs. Petalisp stores the calculation of a gradient as an AD-struct. In addition they are added to a hash table, so that gradients can be reused, if they referenced more than once. Thus the gradient \( \frac{\partial z}{\partial y} \) is only calculated once for \( w \) and \( b \). In order to implement this example, first we have to build the computation graph of \( z \) by the means of Petalisp lazy functions. Here the parameters have to be unknown-arrays. In the next step a differentiator is created, which maps a parameter to its gradient. The differentiator interface takes a list of graph outputs and a list of output gradients of the same shape. In this example the differentiator is created with \((\text{differentiator} \ (\text{list} \ z) \ (\text{list} \ 1))\). An application of this lambda on a parameter results in a graph computing its gradient. In figure 14 it can be seen that the calculation of \( \frac{\partial z}{\partial w} \) and \( \frac{\partial z}{\partial b} \) are not two separate disjoint graphs. Indeed Petalisp recognizes shared gradients between paths and reuses them. The first kernel loads the parameters and calculates \( 2y \). Note due to Petalisp optimization the addition, multiplication and squaring is done within one kernel. The gradient on node \( y \) then is used to derive the gradient for the weights and bias.

1.4 LispNet

In this chapter we introduce LispNet [16]. LispNet is a machine learning framework developed for Common Lisp. It is written on top of Petalisp’s high performance computation modules. In this chapter we discuss the main implementation of LispNet and then show how a developer can use
it to do high performance machine learning. At the end we compare it with an other state-of-the-art machine learning framework TensorFlow/Keras. As a disclaimer, this work shows the current state of LispNet. Since LispNet and Petalisp is under active development, classes, interfaces and performance of this framework could change in the future.

1.4.1 Design and architecture

The concept and design is similar to other machine learning frameworks like TensorFlow. A user is be able to define arbitrarily differentiable computation graphs consisting of parameters, layers and different functions. Parameters can be optimized regarding a specific loss function defined by the user. The framework supports different ways to build and train custom networks and models. LispNet is developed for a modular usage. A user can easily extend these modules by custom layers, metrics and other machine learning concepts. To understand how this framework is made we explain the main interface between Petalisp and LispNet. LispNet requires Petalisp and uses its lazy-arrays and computation modules to build a computation graph. Therefore LispNet is also based on the concepts of lazy-evaluation and takes advantage of Petalisp features. As explained in section 1.3.2 we can build parameterized networks with lazy-unknowns. Lazy-unknowns can be understood as a placeholder arrays with a defined shape. Thus trainable weights are represented as lazy-unknowns in the graph. To be more specific Petalisp API provides a class which implements this interface.

**Network:** The network class describes a list of unkown arrays, the computation graph and an evaluator. It can be created by the call make-network which takes the computation graph as parameter. The method then evaluates the graph by an evaluator object. The evaluator requires a list of outputs and parameters. It evaluates the graph and initializes required buffers for calculations and outputs. With the function call-network we can compute and run the graph. It takes a list of parameters which alternate between unknown and normal arrays. Here we can define which parameters in the computation graph is initialized by which value. For instance a network consisting of two lazy-unknowns \( \tilde{A} \) and \( \tilde{B} \) initial values A and B can be computed by following call:

\[
\text{(call-network network (list } \tilde{A} \text{ A } \tilde{B} \text{ B)).}
\]

This invokes the evaluator replaces the delayed-unknown actions with delayed-array actions referencing to the initial values. Then compute is called on this graph resulting in a list of outputs the network provides. Note with the means of multiple outputs we can calculate different losses, predictions and gradients with a single computation call. As explained we use unknown-arrays and formal weights for the network. However, lazy-arrays do not carry any meta information regarding machine learning. In fact, there are several different initialization of weights depending of its usage. In addition, since we are optimizing these parameters, we
need to save the actual trained values in a storage. Therefore we need a direct association between lazy-unknown and actual stored values. LispNet provides a specific class which implements this logic.

**trainable-parameter:** A trainable parameter is an instance which has a reference to an unknown-array and an array of the same shape. With the call (make-trainable-parameter) we can create such an object. The initialization of the storage slot however is not done by this function and is done by other modules such as layers. LispNet supports every datatype Petalisp and Common Lisp provides. For example we can build weights of special datatype like "rational" which is a ratio between two integers, the numerator and denominator. The common datatype in machine learning is float32 but as a rule of thumb LispNet normally initializes weights of the same datatype as the input. As we now introduced how weights are created in a network we now further explain how the computation graphs are built. The key procedure a machine learning user needs to define is the forward pass. The forward pass defines a function which maps the input of a neural network to its outputs. Here LispNet only supports one output but this will be further extended in the future. The signature of the forward pass is defined by generic method (defgeneric forward (model input)). The user also can define many different forward passes with different names and implementations. The input parameter stands for the input of the neural network and is an unknown-array. In the forward pass a user can create several trainable parameters and layers and can apply arbitrarily Petalisp lazy functions.

**Layer:** A layer takes an input and a set of hyperparameters and applies specific computations to the input. It can be considered as a subroutine for the forward pass with its own trainable parameters. We call these layer weights. There are many different layers in machine learning. The most common ones are dense and convolutional layers. In the current state of LispNet the framework provides following layers: conv2d-layer, dense-layer, transposed-conv2d-layer, flatten-layer, maxpool2d-layer, activation-layer. More layers will be developed in future versions. Regarding the class concept a layer is an abstract class which provides the interfaces: layer-compile, call and layer-weights. Each is a generic method which is implemented by the concrete layer sub-classes. Method layer-weights returns a list of weights of this layer. For a dense-layer it returns the slots weights and bias as a list. This interface can be used to set and load weights of a layer. Generic function layer-compile initializes the weights by a specific initializer. LispNet provides several initialization functions. For instance, for a dense layer it is common to use zeros for a bias and glorot-uniform or also called Xavier uniform for weights. The weight initialization is invoked by the model class. Method call defines the actual forward pass of the layer. Here a user can define which lazy-functions are applied. We also can create trainable-parameters and additional sub layers within the call function. This
gives a user the freedom to create dynamic networks depending on the input. Since the input is also parameterized we can build dynamic networks dependent on the shape of incoming data. To instantiate a layer the function create-layer has to be called. To create a dense-layer with 5 output neurons we call:

```lisp
(create-layer 'dense-layer model :out-features 5 :activation #'relu)
```

Figure 15: Instantiation of a dense layer

As the code in figure 16 shows we also can add an activation function to the layer. An activation function is a function which is applied at the end of a layer. They are used to add non-linearity to networks. LispNet contains the most common ones: Sigmoid, Softmax and ReLu. A user can either pass activation functions via lambda macros through layers or can use a wrapper-class named activation-layer. Note the creation of a layer with its hyper-parameters is similar to other frameworks like Keras (TensorFlow), so that is very easy for new users to get used to. The create-layer function returns an object of the layer which then can be called on an input: (call layer-obj input).

As we now know how to create networks we explain how to train them. To optimize the model we need to update the weights via gradient descent. Suppose the gradients of weights are given. Then an optimizer is needed to update trainable parameters.

**optimizer:** The optimizer class is abstract and defines two interfaces: optimizer-compile and update-weights. In addition every optimizer saves the learning-rate which is a machine learning hyperparameter and the number of updates the optimizer has done. Method optimizer-compile initializes the optimizer and update-weights is used to optimize the parameters. Here this method is called with a list of parameters and a list of their corresponding gradients. There are two different optimizers implemented in LispNet: Stochastic Gradient Descent (SGD) and Adam. Here SGD follows the equations 5. Since a user can use momentum we need to save the previous gradient for each trainable parameter. Each parameter is trained independently, so that we need a list of previous gradients. For the same reason the Adam optimizer needs to save previous gradients too. However here we need two lists since Adam works with a first and second momentum of the gradient (m-list, v-list). In the first update iteration previous gradients are set to zero. Briefly summarized, trainable parameters and layers are used to define the forward pass of a network and the optimizer updates its weights regarding their gradients. To do actual machine learning all classes are used by a specific controller class.

**model:** The model class is the main user interface for a machine learning developer. Its model backend saves references to layers and trainable-parameters used for a network. It can be considered as a wrapper-class of a network which stores trained weights and other meta information. The attribute output references the function of the forward pass. Here the default name is #'forward. A user however can choose different names and create multiple forward passes for a model by passing custom functions via lambda macros. Same logic applies to the loss function. The main methods this class provides are model-compile, fit and predict. Model-compile is basically just a setter method, where a user can define which optimizer, loss function and metric is used. Our framework provides several loss functions and metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), Binary/Categorical Cross-Entropy and Binary/Categorical Accuracy. More complex loss functions can be easily added by implementing the interface: (defun loss-func (y-pred y-true)) where y-pred represents the prediction of the network and y-true label data. To create and train a model with hyper-parameters: optimizer=Adam, loss=MSE, learning-rate=0.001, metrics=MAE, we call:

```lisp
(let* (((opt (make-adam :learning-rate 1e-3))
    (model (make-instance #'model)))
    (mode-compile model :loss #'mse :optimizer opt :metrics (list #'mae)))
```

Figure 16: Creation and compilation of a model

Unlike other frameworks model-compile does not initialize the weights of a model. It just sets all
required information to build the backward pass. Since models are dynamically built regarding the input shape LispNet compiles parameters at the start of method fit and predict. We now explain how the model is trained with the method fit. Here the data used is split between training and validation data. Training data is used for the actual optimization and validation data is used to evaluate the model regarding specific metrics. Validation metrics can be used to stop the training (early-stop) or to trigger other training events. The two other main hyper-parameters for fitting are the number of epochs and the batch size. A epoch describes the fitting for all given data. Since a model takes more iterations of optimization to obtain a good precision it is common to use more epochs. With batch size we can control how much data samples are stacked together for each forward and backward pass. If the data can not be split into blocks of the same size, it takes the rest block as a separate batch. The algorithm 2 lists the steps of fit. First we need to compile all trainable parameters if the model is not initialized. To create all trainable-parameter objects and their corresponding lazy-unknown-arrays we call the forward pass function on an input parameter:

```
1 (let* ((input-parameter (make-unknown :shape (~ 1 ~ sample-shape)))
      (forward-pass (funcall (output model) model input-parameter)))
```

Figure 17: Create a forward pass for a batch size of one

This invokes (make-trainable-parameter) and (create-layer) functions and builds the model backend. The next step is to set the initial values of the weights. This is done by calling (layer-compile) on all layers. Other parameters which are not part of a layer are initialized by an uniform distribution between 0 and 1. We then generate a training network shown in figure 18. First we have to define all parameters: The input-parameter, all trainable-parameters and a label-parameter which stands for the label-data used in the loss function and metrics (y-true). Variable forward stands for the output of the forward-pass and describes the prediction of the model (y-pred). We then apply the loss function and metrics to y-true and y-pred. We get the gradients of trainable parameters by calling a differentiator on the loss function with an initial gradient of one. This returns a lambda function which maps a trainable parameter to its gradient. It implements the logic of backpropagation. Gradients, metrics and loss are then defined as the network outputs. We then can evaluate and create the network by applying make-network to them. It is important to mention, that this creates one computation graph with multiple outputs. Thus Petalisp shares computation paths and can reuse interim results for each output, i.e. the forward pass is computed once. This network then can be called with actual models weights, input and label data. Networks only have to be created once and can be reused for every batch. There also might be the case, that a batch-size is chosen, such that number of samples divided by the batch size does not quite add up. Therefore we also have to define a training and validation network for the last batch of the dataset which makes four networks in total. After that method fit splits the dataset in batches and trains the model by method (train) and validates it with (test). The training calls the training network and optimizes
the model by calling update-weights of the optimizer. This is iterated by a given number of epochs. The fitting can be stopped by a callback. Here LispNet provides a stopping criterion. It saves the weights of a model for the best validation loss. When we don’t achieve a better validation loss for a certain number of epochs after the best result, we stop the fitting. This is a common tool to prevent Overfitting. Another important method of the model class is predict. After the model is trained a user wants to generate predictions. Here a user does not care about the value of the loss functions or metrics. Thus the network which is called only consists of the forward pass and returns its output (y-pred). To save and load trained models a user can call save-weights and load-weights which writes the model backend with its weights on the hard drive.

1.4.2 Layer implementation

Now we introduce the implementations of class dense-layer and conv2d-layer. To define a custom layer class a developer needs to implement interfaces layer-compile, layer-weights and call. Here we only focus on their call functions. Dense layers are fully connected layers which connect all neurons from a previous to next layer. Here the main parameter of this layer are the number of output neurons (output-features). As described in section 1.2.1 the forward pass of a dense layer is:

$$ y^{m \times k} = x^{m \times n} W^{n \times k} + b^k $$

where $m$ equals the batch-size, $n$ describes the number of input neurons and $k$ describes the number of output neurons. This is implemented by a matrix-matrix multiplication of the input $x$ with weights $W$. At the end a bias vector is added for each sample in the batch. Weights and bias are trainable parameters and considered as the weights of the layer. Matrix-matrix multiplication in Petalisp is done via function matmul:

```
(defun matmul (A B)
  (l lazy-reduce #'+
    (l (l lazy-reshape A (transform m n to n m 0))
      (l lazy-reshape B (transform n k to n 0 k))))
```

As shown in 19 by the means of transformations (lazy-reshape) of A and B the lazy multiplication results in an iteration space of $n \times m \times k$. Then it is reduced by the first axis leading to a $m \times k$ matrix. The addition of the bias is simply done with lazy vector addition. We can use that now to describe the call of a dense layer in figure 20.

It is worth to mention, that computation graphs are only built on top of unknown arrays. Thus we
(defmethod call ((layer dense-layer) input &rest args)

;; compute n, k, m
;; create and set trainable parameters
(let* ((weights (weights weights weights-param))
       (bias (weights bias-param)))
  (lazy #'+ bias (matmul input weights))))

Figure 20: Forward pass of dense-layer

need to access the weights slot of a trainable parameter which saves the reference to an unknown array. The output of this layer is a lazy-array. In general input and outputs have the same type, so that a user can use its output to feed it to the input of an other layer. This makes nested-layered networks easy to implement.

Another important layer which is written for LispNet are convolutional layers. In particular conv2d-layer. As explained in section 1.2.4 this layer performs a discrete two-dimensional convolution on an image of shape (~b ~y ~x ~c) where b is the batch-size, y is the height, x is the width and c describes the number of channels of the image. Input channels can be considered as output features of previous convolutions or color channels like red, green and blue. The main parameters for conv2d-layer are: the number of out-channels m, the kernel size, strides, the padding and the stencil. M is simply the number of different filters used for convolution, which results in m two-dimensional convolutions in total. The kernel size defines the two dimensions of the filter. Like other deep learning frameworks LispNet uses square filters by default. However, a user also has the ability to define custom filter forms by describing a stencil. For instance a 5-point stencil which adds the middle point, left, right, top and bottom value is given by the stencil: '((0 0) (0 -1) (0 1) (1 0) (-1 0)). Thus a user can define efficient sparse convolutions without zero initialization of kernel values. The filters itself similar are created as trainable parameters of the shape (~n ~c ~m) whereas n stands for the number of stencil values. For a discrete two dimensional convolution we differ between two modes: "valid" and "same". The default mode "valid" only convolutes the inner points of the image, so that the filter does not overlap with the border. Since the size of the filter can be different, the inner size of the image also can change. Therefore the first step of the forward pass is to determine the bounding box the of stencil. In the example of the 5-point stencil the box would have the height and width of three. This would lead to an innergrid of the shape (~1 (1- y) ~1 (1- x)). Since we only apply the convolution to the inner points, the spatial dimensions of the output would also have the same shape. The mode "same" on the other hand ensures, that the spatial dimensions of input and output keep the same. This is done by extending the input border by zero padding before its get convoluted. Now we explain how a simple two-dimensional convolution is done in Petalisp. Lets consider the convolution of an image with one channel and one filter w. This is done by iterating through all inner points and add all neighbours the stencil describes multiplied by the specific filter value. Here a Petalisp developer needs to take care of the usage of parallel efficiency. Iterating through every single points will prevent Petalisp from fast optimization and parallelization. Thus a better ansatz would be to select all n neighbours for each innerpoint as n different lazy-arrays. This can be implemented by lazy-reshape and index translations. For example to select all left neighbours of the inner points we first shift the image to the right and select the inner space (~1 (1- y) ~1 (1- x)):

(lazy-reshape image (transform i j to i (1+ j)) innerspace)

Figure 21: Select all left neighbours of the inner grid

These neighbors arrays are then multiplied by \(w_i\) and added together. The corresponding convolution for the example above is therefore written as:
In this section we explain how to build a custom machine learning model in LispNet and train it to correctly predict and label the MNIST dataset [17]. The MNIST dataset contains 70000 handwritten digits. Each sample is a $28 \times 28$ big grayscale picture. Learning MNIST data is a great introduction to machine learning and serves here to demonstrate the Proof of Concept of LispNet. We use a small network consisting of two conv2d, two maxpool2d and one dense layer.

The architecture of this network is shown in figure 2. The first convolution applies 16 filters of size $3 \times 3$. The second convolution takes 32 filters of the same size. All convolutions operation on mode "valid". Furthermore they apply the activation function ReLu. To reduce the dimensional size of feature maps we use Max-pooling in between with a pool shape of $2 \times 2$. The resulting features at the end have the shape ($\sim 7 \sim 7 \sim 32$). All features then are flattened and passed to a dense layer with 10 output neurons. A Softmax function is applied, so that each output neuron determines the

Note, that the code in figure 22 only works for $c = 1$ and $m = 1$. In fact a dense2d-layer also can work on multiple input and output channels. Therefore the shape of $w_j$ is ($\sim c \sim m$) and the shape of the inner space is ($b \sim inner-y \sim inner-x \sim c$). Multiplying the image with filter values results in an iteration space of $b \times inner-y \times inner-x \times c \times m$. According to equation 12 input channels are then added together by reduction. Since the first dimension of the result describes the batch-size we have to rearrange the axis, so that the channel axis (axis c) is rotated to the front. The output of this operation then has the size ($b \sim inner-y \sim inner-x \sim m$). The result of the convolution then can be strided. With a strided filter a user can downsample the size of output feature maps. For instance, a vertically and horizontally stride of two results in a four times smaller image. This is implemented by applying lazy-reshape with a step size according the stride shape. Due to Petalisp’s extreme optimization, striding also results in less computations. Unlike eager evaluation, with lazy evaluation we only calculate what we need. Therefore only a partial amount of points are convoluted when a stride size above one is used. At the end of the forward pass a trainable bias with the same shape is added to the result. An activation function can be applied too if given. The implementation of this layer is also reflected in the forward pass of other layers like maxpool2d and transposed-conv2d. To implement maxpooling we use subroutines of convolutional layers. Instead of adding values we apply the maximum function during convolution. Then this result is strided with a step size which equals the spatial dimensions of the kernel. By setting the stride shape and kernel shape to ($\sim p_x \sim p_y$) we iterate through every $p_x \times p_y$ block and select the maximum of the input. Note this layer does not have any trainable parameters. Basically Petalisp here only saves indices to the largest values. Transposed convolutions are similar implemented as regular convolutions. The difference at implementation is that this layer uses a stencil which has flipped spatial dimensions. $w_0$ becomes $w_{m-1}$. The parameter striding has also a different effect for transposed convolutions. Instead of down sampling the result by a defined step size we extrapolate the input. This is done by adding a specific number of zeros between input values, i.e. with a stride size of two there is one zero between each input. Thus with the means of striding we increase the size of spatial dimensions. Transposed-conv2d is used to upscale feature maps and therefore is very important for Encoder/Decoder based networks. Like convolutions can be used for pooling transposed convolutional layers also can be used for unpooling. The code itself is written in a modular way, so that new pooling and convolution operations easily can be added to the framework. The weight initialization itself is done in layer-compile. By default LispNet initializes filters by sampling from a Glorot uniform distribution. With the parameter kernel-initializer a user also can pass custom weight initializations with lambda macros.

1.4.3 Hello World with MNIST

In this section we explain how to build a custom machine learning model in LispNet and train it to correctly predict and label the MNIST dataset [17]. The MNIST dataset contains 70000 handwritten digits. Each sample is a $28 \times 28$ big grayscale picture. Learning MNIST data is a great introduction to machine learning and serves here to demonstrate the Proof of Concept of LispNet. We use a small network consisting of two conv2d, two maxpool2d and one dense layer. The architecture of this network is shown in figure 2. The first convolution applies 16 filters of size $3 \times 3$. The second convolution takes 32 filters of the same size. All convolutions operation on mode "valid". Furthermore they apply the activation function ReLu. To reduce the dimensional size of feature maps we use Max-pooling in between with a pool shape of $2 \times 2$. The resulting features at the end have the shape ($\sim 7 \sim 7 \sim 32$). All features then are flattened and passed to a dense layer with 10 output neurons. A Softmax function is applied, so that each output neuron determines the
probability of each digit. Thus our label format is a one hot encoded vector of size 10 which encodes
which digit is represented in the sample. As we now explained the architecture we show how it can
be written in LispNet. First we have to define a custom model class which inherits from model.
Here a user can define the layers in a static or dynamic way. Static would mean, that all layers are
defined as class attributes and are instantiated in the constructor. Dynamic layer creation would
mean to create layers in the forward pass of the model. This can be very useful, if a network is
built based on the shape of the input. Since we built network with a fixed architecture we choose
static layers. The code in figure 23 shows the definition of such a model:

```
(defclass mnist-model (model)
  ((conv1 :accessor conv1)
   (conv2 :accessor conv2)
   (pool1 :accessor pool1)
   (pool2 :accessor pool2)
   (flatten :accessor flatten)
   (dense :accessor dense)))

(defmethod initialize-instance :after ((model mnist-model) &rest initargs)
  (setf (conv1 model) (create-layer 'conv2d-layer model :out-channels 16
                                     :kernel-size 3 :padding "same" :activation #'relu))
  (setf (conv2 model) (create-layer 'conv2d-layer model :out-channels 32
                                     :kernel-size 3 :padding "same" :activation #'relu))
  (setf (pool1 model) (create-layer 'maxpool2d-layer model :pool-size '(2 2)))
  (setf (pool2 model) (create-layer 'maxpool2d-layer model :pool-size '(2 2)))
  (setf (dense model) (create-layer 'dense-layer model :out-features 10
                                      :activation #'softmax))
  (setf (flatten model) (create-layer 'flatten-layer model)))
```

Figure 23: Definition of MNIST model

The described network has therefore 6 layers in total. Since we now defined and set all layers, we
simply call them one after the other on the input. This is written in the forward pass of the model.
The default method for it is named forward and has the model and input as parameter. Other
methods also can be used as forward pass by setting the output slot of the model. Thus a user can
learn and predict on a main forward pass but also can extract specific layer outputs like feature
maps for further analysis. Figure 24 describes the forward pass of this model:

```
(defmethod forward ((model mnist-model) input)
  (call (dense model)
         (call (flatten model)
               (call (pool2 model)
                     (call (conv2 model)
                           (call (pool1 model)
                                 (call (conv1 model)
                                        (lazy-reshape input (transform b y x to b y x 0))))))))
```

Figure 24: Forward pass of MNIST model

Note, that the first convolution expects an input format of batch-size × height × width × channels.
Therefore we have to extend our gray-scale picture by an extra dimension of size 1 with lazy-reshape.
The syntax and the model creation is very similar to Keras/TensorFlow. An experienced machine
learning developer therefore should feel quite comfortable working on LispNet. As we now defined
the model we just have to create it and fit it on MNIST data. In order to compile the network
we also have to set an optimizer and other machine learning hyper-parameters. We use following
parameters to train our model: loss = Categorial Cross Entropy, learning rate = 0.001, optimizer = Adam, batch size = 64, epochs = 15, metrics = Categorial Accuracy. Code in figure 25 shows the creation and fitting of our model with these parameters. Variable train, train-label, val, val-label

```lisp
(let* (((optimizer (make-adam :learning-rate 0.001)))
        (model (make-instance 'mnist-model)))
      (model-compile model :loss #'categorial-cross-entropy
                      :optimizer optimizer :metrics (list #'categorial-accuracy))
      (fit model train train-label val val-label :epochs 15 :batch-size 64))
```

Figure 25: Fitting the MNIST model

references to the train and validation data of MNIST. Here the dataset is given as a numpy matrix file (.npy). The creator of Petalisp developed a library called numpy-file-format to handle such files. It delivers the function load-array and save-array. By the means of this framework we can load the dataset. We use 10000 samples for training and 1000 samples for validation and testing. With a batch size of 64 the model trains on 172 batches per epoch. The method fit follows algorithm 2. It trains and validates the model. As a result, it returns a list of all computed metrics for training and validation of each epoch. Figure 26 shows the loss and accuracy during fitting. As expected we see how the model is getting better after each epoch. After 15 epochs the model is able to predict handwritten digits with a 99% accuracy for training and with a 98% accuracy for the validation set. This shows that LispNet is able to train machine learning models with the means of Gradient Descent.

1.4.4 Performance

We now show the performance of LispNet during fitting. We compare the computation time with another popular machine learning framework Keras which is based on TensorFlow. The tests run on CPU only mode since Petalisp does not support graphic cards yet. The machine we are testing on has an AMD Ryzen 5 2600, 12 threads and is running on Windows 10. We use Steel Bank Common Lisp (SBCL) 2.0.0 as the Lisp compiler. We test the fitting of dense and convolutional models with one to four layers. Both test sets contain 4096 samples for training and 1024 samples for validation. We use Mean Squared Error as loss function and Adam as optimizer. The batch size is set 64, so that we get 64 training and 16 validation calls. All weights and data are single-precision floating point numbers. For dense networks we use n hidden neurons. We test on n = 256, n = 512 and n = 1024. The number of input and output neurons are also set to n. It table 1 it can be seen, that the current prototyp of LispNet gets outperformed by TensorFlow. Note, that dense networks rely on matrix-matrix multiplications during forward and backward pass. TensorFlow
<table>
<thead>
<tr>
<th>Layers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>LispNet256</td>
<td>10.76</td>
<td>39.47</td>
<td>53.83</td>
<td>85.81</td>
</tr>
<tr>
<td>TensorFlow256</td>
<td>2.12</td>
<td>2.56</td>
<td>2.92</td>
<td>3.41</td>
</tr>
<tr>
<td>LispNet512</td>
<td>39.23</td>
<td>122.38</td>
<td>202.36</td>
<td>288.72</td>
</tr>
<tr>
<td>TensorFlow512</td>
<td>3.54</td>
<td>6.31</td>
<td>8.87</td>
<td>11.33</td>
</tr>
<tr>
<td>LispNet1024</td>
<td>192.94</td>
<td>556.82</td>
<td>916.41</td>
<td>1290.3</td>
</tr>
<tr>
<td>TensorFlow1024</td>
<td>6.87</td>
<td>13.68</td>
<td>20.43</td>
<td>27.34</td>
</tr>
</tbody>
</table>

Table 1: Fitting time for dense models in LispNet and TensorFlow in seconds

<table>
<thead>
<tr>
<th>Layers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>LispNet32</td>
<td>19.32</td>
<td>45.29</td>
<td>68.05</td>
<td>87.1</td>
</tr>
<tr>
<td>TensorFlow32</td>
<td>7.71</td>
<td>13.75</td>
<td>19.84</td>
<td>23.39</td>
</tr>
<tr>
<td>LispNet64</td>
<td>75.13</td>
<td>204.46</td>
<td>323.17</td>
<td>429.64</td>
</tr>
<tr>
<td>TensorFlow64</td>
<td>28.32</td>
<td>51.34</td>
<td>70.36</td>
<td>88.96</td>
</tr>
<tr>
<td>LispNet128</td>
<td>304.61</td>
<td>862.48</td>
<td>1422.62</td>
<td>1936.12</td>
</tr>
<tr>
<td>TensorFlow128</td>
<td>99.83</td>
<td>180.51</td>
<td>264.48</td>
<td>329.4</td>
</tr>
</tbody>
</table>

Table 2: Fitting time for convolutional models in LispNet and TensorFlow in seconds

uses an optimized hard coded subroutine of the Intel Math Kernel Library. Optimized and cache efficient techniques like blocking are additionally applied resulting in less memory bottlenecks. In LispNet we do matrix-matrix multiplications as described in figure 19. Native backpropagation and broadcasting result in less efficient code. Here a better subroutine is needed for Petalisp especially during the backward pass. The developer of Petalisp is currently actively working to speed up these algorithms. Thus, the performance might be a lot better for future versions.

The test of convolutional models requires up to four conv2d-layers. All layers use 3 channels, i.e. in-channels = out-channels = 3. The kernel size is set to $3 \times 3$. We use "valid" as padding mode. We test on different input shapes: $32 \times 32 \times 3$, $64 \times 64 \times 3$ and $128 \times 128 \times 3$. The measurements are shown in table 2. On average TensorFlow is about four times faster than LispNet. It is worth mentioning, that vectorization techniques like SIMD are not implemented in Petalisp yet, while TensorFlow already has these features. Adding vectorization to Petalisp and LispNet could lead to a similar performance as TensorFlow. To summarize our results the current state of LispNet does not reach the performance of current machine learning frameworks like TensorFlow. Since Petalisp is under active development, this might change in the future. In the today’s competing world of machine learning framework speed is one of the most important metrics. Therefore the success of LispNet depends on future versions of Petalisp.

1.4.5 Python interface

One of the most used languages for machine learning is Python. According to the TIOBE Community Programming index Python has become the most popular programming language in year 2022 [18]. Besides a decent amount of good machine learning frameworks such as TensorFlow or Pytorch there are also a lot of community made packages for data analysing, visualization and processing. Therefore most developer would keep working in Python since it provides such a big machine learning community. Lisp on the other hand is placed on 33th TIOBE rank (state of 13.06.2022). Although Common Lisp actually provides more language features than other programming languages, its community is quite small compared to Python’s one. Therefore Common Lisp lacks frameworks generally used for machine learning. In addition the specific syntax of Lisp is quite special and unfamiliar for a Python developer. Thus it would be awesome to use LispNet and stay at the same time in a Python environment. We created a Python interface for this reason. The main goal to autogenerate LispNet code for custom Keras models, so that a Keras user could use LispNet as backend without changing code. To invoke Common Lisp code within a Python environment we use a python package named "cl4py" [19]. This module opens a pipe to a Lisp interpreter. We can pass Lisp code and data to it. Lisp code consists of nested linked lists of atomic variables. In the same way we can write Lisp Code as nested Python lists. Atomic variables are written as strings. Other simple datatypes like integers or floats are directly passed to the Lisp
Cl4py provides a writer and reader for simple datatypes. For instance, the python statement: \texttt{x=3} is represented as \texttt{['setf', 'x', 3]}. With \texttt{lisp.eval()} we can evaluate and execute this list in Lisp. The result is then read and returned to Python. Furthermore, the reader and writer also supports numpy matrices. Thus we can directly pass machine learning data to LispNet without exchanging numpy files through the numpy-file-format package. Numpy arrays are converted to Common Lisp arrays and vice-versa. In order to convert Keras models to LispNet code the model definition and forward pass has to be converted. Therefore it is necessary to transform Python Code into Common Lisp Code. The interface provides a converter. Its job is to generate Common Lisp code from Python class definitions, methods and functions. This is done by parsing and iterating recursively through the Python Abstract Syntax Tree (AST) [20]. We can then map every node of the tree to Common Lisp equivalent code. Note this converter at the current state is still a prototype, so that not all Python features are implemented yet. However, we still can convert simple class definitions, methods, arithmetic and boolean operations and control flow statements. The result of the converter can be seen in Figure 27.

![Code Example](image)

Figure 27: Common Lisp generated code for factorial

In a) we see the Python implementation of function factorial and b) shows its auto-generated Common Lisp code. The root node of its AST is a module. The module consists of a node of type FunctionDef. FunctionDef has three main components: a name, arguments and a body. The body describes the implementation of the function. Attribute arguments is a list of nodes of type arg. We can directly map this to Common Lisp by function (defun name args body). We recursively then call our converter on the body part. The body consists of an If statement. The If node has a test, body and else node. Test describes the condition which in this example is described as a Compare between Name n and the number 1. We can map if, elif and else by the means of Lisp function cond. Cond takes a list of test and action pairs. The evaluation of cond returns the first action in the list whoose test is evaluated to true. Therefore with the fist test and action we can implement a simple if statement. The else statement is then simply a second action with a test which is set to true. For elif we convert its test and body to an if statement within the else part. Note, that the code in the body and orelse can consist of multiple statements. Thats why we use function like progn by default which evaluates multiple statements. The last statement is considered as the return value of a body block. The multiplication here as AST is described as binary operation with n as a left operator and a function call as its right operator. Since the forward pass of our models only work on top of lazy-arrays we also map any arithmetic operation to Petalisp functions. This makes any function automatically lazy if the option is enabled. Mapping the logic of the forward pass to Petalisp is one step. The other important step is to map Keras layers to LispNet layers. A layer creation in Keras results in a create-layer call in LispNet. This is triggered when the converter is parsing an function call with a Keras specific layer name. We then map the layer with its arguments to LispNet equivalent code. Of course this only works if given features in Keras are also implemented in LispNet.

We can see such an layer conversion in figure 28. Here we create a two layer model dynamically in the forward pass. Layers Dense and Conv2D are converted to dense-layer and conv2d-layer. Note the creation of local variables in Python. In an Abstract Syntax Tree we can not find any
def forward(model, input):
    dense = Dense(units=3, activation='relu')
    conv = Conv2D(filters=16, padding="same")
    dense(conv(input))

(defun forward (model input)
  (let ((dense (create-layer 'dense-layer model
                        :out-features 3 :activation #'relu)))))
  (let ((conv (create-layer 'conv2d-layer model
                        :out-channels 16 :padding "same")))
    (call dense (call conv input))))

Figure 28: Generated LispNet code from Keras

information whether variable dense or conv are local variables or an object attribute. We also do
not know whether the variable is already defined. This is determined by an Python interpreter. To
answer this question without running the code in a Python environment we are parsing the AST
recursively together with scopes. A scope is simply a list of variable names. If we read a variable
assignment we check if the variable is listed in the scope. If the scope does not contain its name
we create a local variable in Common Lisp with a let statement and add its name to the scope.
The local variable in Common Lisp is then bound to a name and valid for the body part of the let
statement. Thus sequential assignments of variables are mapped to nested calls of let statements. If
the scope already contains the variable we only change its value by Lisp function setf. Besides local
variables in a function there are also attribute assignments. For instance the Python call self.x=3
in a constructor call sets and defines an attribute of an object. In Common Lisp we can not di-
rectly add attribute slots after object instantiation. This could be done by changing the standard
Common Lisp class definition by a custom metaclass. A metaclass can be defined as a class of a
class. We can create a dynamic slot system by defining such an metaclass for model definitions.
Another approach would be to parse every method and constructor in a static way and search for
every attribute assignment. At the current state our interface parses the model and checks for self
references. We then map every variables which is accessed by name self to a classic Common Lisp
slot. With this converter a Keras user simply can pass its model to our LispNet converter. The
general idea is to swap the backend from TensorFlow to LispNet just by changing the package. The
interface then can be used to access methods fit and predict. We can pass training and validation
data through numpy matrices. From a Python perspective a user does not need to care about lazy
arrays since the objects exchanged between these frameworks are numpy matrices. Thus we can use
all popular data visualization and processing packages in Python in combination with LispNet.
2 Application to Multigrid PDE with jump coefficients

This chapter is about a machine learning approach for Multigrid Partial Differential Equations. We first introduce the problem definition and the Multigrid ansatz. In the second part we show how machine learning can be used to solve the problem. The main related work is "Learning Optimal prolongation and Restriction Operators for Multigrid PDE Solvers" by Philipp Suffa [21]. We extend this work by investigating another machine learning approach. We use our framework LispNet to build and train the required models.

2.1 Introduction

2.1.1 Poisson equation

The main problem this work is about are Poisson equations. Poisson equations are Partial Differential Equations. They are part of boundary value problems, where the solution of the differential equation can be found by a set of boundary conditions. Poisson equations have application to various number of research fields in physics such as gravity, electrostatics or temperature distribution. The Poisson problem with Dirichlet boundary conditions is expressed as follows:

\[
\begin{align*}
-\Delta u &= f \quad \text{in } \Omega \\
u &= g \quad \text{on } \partial \Omega
\end{align*}
\]  

(15)

where \( u \) is the solution of the problem and \( f \) and \( g \) describe two functions. \( \Omega \in (0,1)^d \) is a d-dimensional unit room. \( g \) describes boundary conditions for our solution. In this work we only use constant boundaries (\( g = 1 \)). A common approach to solve this equation is the finite difference method. It works by discretizing the continuous functions in the domain on a grid based structure. With a step size \( h = 1/n \) we get \( (n+1)^d \) points. Since the solution is defined at the boundary we have \( (n-1)^d \) unknowns. We transform a continuous problem to an discrete one. For instance, for one dimension we can write:

\[
- u_{x-1} + 2 u_x - u_{x+1} \over h^2 = f_x \quad 1 \leq x < n - 1
\]

\[
u_0 = u_n = 1
\]

(16)

Here index \( x \) stands for grid point at position \( h \cdot x \). The resulting problem we get is a linear equation with \( (n-1) \) unknowns:

\[
A_{d=1} \left( \begin{array}{c}
u_1 \\
\vdots \\
u_{n-1}
\end{array} \right) = \frac{1}{h^2} \left( \begin{array}{ccc}
2 & -1 & 0 \\
-1 & \ddots & \ddots \\
0 & \ddots & -1 & 2
\end{array} \right) \left( \begin{array}{c}
u_1 \\
\vdots \\
u_{n-1}
\end{array} \right) = \left( \begin{array}{c}
f_1 \\
\vdots \\
f_n
\end{array} \right)
\]  

(17)

\( A_{d=1} \) is a symmetric, tridiagonal matrix with \( 2 \) on the diagonal and \(-1\) next to the diagonal. With two dimensions we discretize the domain in a \( x \) and \( y \) direction. The laplacian \( \Delta u \) is given by \( \partial^2 u / \partial x^2 + \partial^2 u / \partial y^2 \). Finite difference leads to:

\[
- u_{y,x-1} - u_{y,x+1} + 4 u_{y,x} - u_{y-1,x} - u_{y+1,x} \over h^2 = f_{y,x} \quad 1 \leq y, x < n - 1
\]

\[
u_{y,x} = 0 \quad \text{on } \partial \Omega
\]

(18)

The resulting linear equation then has \( (n-1)^2 \) unknowns. Furthermore we can express \( A_{d=2} \) by the means of \( A_{d=1} \):

\[
A_{d=2} = \left( \begin{array}{ccc}
A_{d=1} + 2I_{n-1} & -I_{n-1} & 0 \\
-I_{n-1} & \ddots & \ddots \\
0 & \ddots & -I_{n-1} & A_{d=1} + 2I_{n-1}
\end{array} \right) \quad \text{with } I_{n-1} = \left( \begin{array}{cccc}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
\vdots & \ddots & \ddots & \ddots \\
0 & 0 & 0 & 1
\end{array} \right)
\]

(19)
According to equation 18 we get a 5-point stencil for each inner grid point. In this work we solve Poisson equations with two dimensions and a discretization with a step size of $\frac{1}{32}$. This makes $31^2$ inner points in total to solve.

2.1.2 Jacobi method

We can solve the discrete Poisson equation in many ways. One way is to apply a direct solver for linear equations like Gaussian elimination. Aside from rounding errors direct methods would deliver an exact solution to this problem. However it is very cost intensive and often impossible for systems with a lot of unknowns. Another approach is to approximate the solution to a specific precision. With iterative methods we can generate a sequence of improving solutions started on an initial guess. The general idea is to find a specific iteration matrix $C$ which results in a fix-point iteration:

$$x^{t+1} = Cx^t$$

$C$ is convergent if its spectral radius $\rho(C) < 1$. Consider the linear system of the form $Ax = b$. The basic concept is to split $A$:

$$A = D + N$$

where $D$ is the diagonal part of $A$. If the iteration

$$Dx^{t+1} = b - Nx^t$$ (20)

converges then $x^{t+1} = x^t$ for $t \to \infty$. Thus fix point of equation 20 also solves the linear equation $Ax = b$. A common iteration method is the Jacobi Relaxation. Here we use $N = (L + U)$ where $L$ is the strict lower and $U$ the upper triangular part the $A$. The iteration then follows then equation:

$$x^{t+1} = D^{-1}(b - (L + U)x^t)$$ (21)

where $x^{t+1}$ is closer to the exact solution of the system than $x^t$. The algorithm converges if the spectral radius of the iteration matrix fulfills the condition: $\rho(D^{-1}(L + U)) < 1$. We then can iterate this step until the error is smaller than a given precision. The error is defined by a norm of the difference between exact and approximate solution: $e = x^t - x$. A common norm is the Euclidean norm: $||e||_2 = \sqrt{\sum_i e_i^2}$. The residuum $r$ describes how well the approximated solution fits the system with application of $A$:

$$r = b - Ax^t = b - A(x + e) = b - Ax + Ae = Ae$$ (22)

Thus we also can use the residuum as a measure for precision. In this work we stop our iterative solvers at step $t$ when $\frac{||r_t||_2}{||r_0||_2}$ is smaller than $10^{-10}$. We can apply equations 21 and 22 to discrete two-dimensional Poisson problems. We then can write:

$$u^{t+1}_{y,x} = \frac{1}{4}(h^2 f_{y,x} + u^{t+1}_{y-1,x} + u^{t+1}_{y+1,x} + u^{t+1}_{y,x-1} + u^{t+1}_{y,x+1})$$

$$r^{t}_{y,x} = f_{y,x} - \frac{1}{h^2}(4u^{t}_{y,x} - u^{t}_{y-1,x} - u^{t}_{y+1,x} - u^{t}_{y,x-1} - u^{t}_{y,x+1})$$ (23)

A common extension to the Jacobi method is to linearly interpolate between previous and next smoothing step:

$$x^{t+1} = \omega \cdot D^{-1}(b - (L + U)x^t) + (1 - \omega)x^t$$ (24)

Here $\omega$ is a relaxation parameter and can be used to speed up the convergence of the Jacobi iteration. A common choice of $\omega$ is $2/3$. These equations are key subroutines of the Multigrid algorithm.

2.1.3 Jump coefficients

A common way to model diffusion problems is to add jumping coefficients to the Poisson equation:

$$-\nabla \cdot (c \nabla u) = f \text{ in } \Omega$$

$$u = g \text{ on } \delta \Omega$$ (25)

30
where c is a coefficient function. In this work we focus on discontinuous functions which consist of several jumps. By discretizing the domain $\Omega$ into a two-dimensional grid we get $c_{y,x}$ for every point. The coefficient at position $y$ and $x$ in the unit space has the value $c(hy,hx)$. Several iterative solvers such as native Multigrid methods show worse convergence once jumps are added. Furthermore they do not convergence to an solution if the jumps are high enough. The main purpose of this work is to design a modified Multigrid to solve such problems. Note that we only focus on positive coefficient functions greater or equal than 1. By adding a coefficient function to the Poisson equation the equations for a non weighted Jacobi method changes to:

$$u_{y,x}^{t+1} = \frac{1}{4c_{y,x}}(h^2 f_{y,x} + c_{y-1,x} u_{y-1,x}^t + u_{y+1,x}^t c_{y+1,x} + u_{y,x-1}^t + c_{y,x+1} u_{y,x+1}^t)$$

$$r_{y,x}^t = f_{y,x} - \frac{1}{h^2}(4c_{y,x} u_{y,x}^t - c_{y-1,x} u_{y-1,x}^t - c_{y+1,x} u_{y+1,x}^t - c_{y,x-1} u_{y,x-1}^t - c_{y,x+1} u_{y,x+1}^t)$$

(26)

The implementation of the Jacobi method therefore only changes by multiplying a specific coefficient for every point in the grid before the stencil operation is applied.

### 2.1.4 Multigrid

As described in section 2.1.2 we can use iterative solvers to approximate the solution $u^*$ of a linear system $Au = f$. The exact solution can be described by $u$ and the error $e$:

$$u^* = u + e$$

(27)

Thus another approach is to relax on the linear equation first, get the error and correct the approximation afterwards. With equation 22 we know that the error can be calculated by relaxing on the residuum.

$$ Ae = r $$

Therefore relaxing on the residuum also relaxes $Au = f$, when we set the initial guess $e^0 = 0$. This correction scheme is one part of the Multigrid algorithm. As the name suggests the method uses multiple grids with different discretization levels. Consider the relaxation of $Au = f$ with a smoother $S$. When we apply the smoother on an initial guess $v$, i.e. $S(v,f)$ we reduce the high frequency parts of the error. In other words the error gets smoother. Smoothers such as Jacobi relaxation are way more effective on high frequent errors. The general idea for Multigrid therefore is relax the problem on a coarser grid. The smoothed error on a fine grid becomes a high frequent parts of the error. In other words the error gets smoother. Smoothers such as Jacobi smoother are way more effective on high frequent errors. The general idea for Multigrid therefore is relax the problem on a coarser grid. The smoothed error on a fine grid becomes a high frequent error on a coarse grid. Furthermore, relaxing on a coarser grid also takes less computations. Thus instead of relaxing $Ae = r$ on a fine grid, we use a coarser grid with step size $2h$. To make this work we need to map the residuum $r$ to a coarser version with four times less grid points. This procedure is called restriction. We describe the restriction by the operator $R^h_2$. For instance, a simple version of this operator in a one-dimensional space with $n+1$ grid points is injection:

$$R^h_2 : \mathbb{R}^{n-1} \rightarrow \mathbb{R}^{n/2-1}, \quad r^{2h} = R^h_2 r^h, \quad r_x^{2h} = \frac{r_{2x}}{2}, \quad x = 1, \ldots, \frac{n}{2} - 1$$

(28)

Since injection only takes every second node, the errors in between do not impact the system. That means these error nodes will not get corrected by default. Therefore a better restriction operator is a linear combination of $r^h_2$, and its left and right neighbour. The full weight restriction of $r^h$ can be written as:

$$R^h_2 r^h = \frac{1}{4} \begin{pmatrix} 1 & 2 & 1 & & & & & & \end{pmatrix} \begin{pmatrix} r^h_1 \\ r^h_2 \\ \vdots \\ r^h_{n-1} \end{pmatrix}$$

(29)

This operator applies the stencil $\frac{1}{4}[1 2 1]$ to every second inner point of the fine grid. Points at the edge of the grid are just copied. We also can apply full weight restriction also to two-dimensional grids. In this case the operator $R^h_2$ is given by the stencil:

$$A_s = \frac{1}{16} \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

(30)
After the restriction of $r^h$ the next step is to relax on it. This gives us an approximation of the error $e^{2h}$. Note, that $e^{2h}$ is computed on coarse grid. To apply the correction scheme of equation 27 $e^{2h}$ has to be mapped to $e^h$. This step is named prolongation. The operator $I_{2h}^h$, which describes it, interpolates values on a coarse grid to a fine grid. In the one dimensional case we write the prolongation in the following:

$$I_{2h}^h e^{2h} = \frac{1}{2} \begin{pmatrix} 1 & 1 & & & \cdots & & & 1 \\ 2 & 1 & & & & & & \\ & 2 & & & & & & \\ & & \ddots & & & & & \\ & & & \cdots & 1 & & & \\ & & & & \cdots & 2 & & \\ & & & & & & 1 & \\ & & & & & & & 2 \\ & & & & & & & 1 \end{pmatrix} \begin{pmatrix} e_1^{2h} \\ e_2^{2h} \\ \vdots \\ e_{n/2-1}^{2h} \end{pmatrix}$$

(31)

The full weight restriction and prolongation have similar stencil operations. In fact, the prolongation matrix is the transposed version of the restriction matrix times a coefficient:

$$I_{2h}^h = kI_{h}^{2h}$$

(32)

with $k = 2$ in one-dimension and $k = 4$ in the two-dimensional case. Therefore stencil $A_s$ is used by both operators.

In summary the correction scheme then follows these steps:

- Relax: $A u^h = f^h$ with initial guess $u_0^h$ with Smoother $S$

- Compute residuum: $r^h = f^h - A u^h$

- Restrict residuum: $r^{2h} = I_{h}^{2h} r^h$

- Relax: $A e^{2h} = r^{2h}$ with Smoother $S$

- Prolongate: $e^h = I_{2h}^h e^{2h}$

- Correct: $u_1^h \leftarrow u_0^h + e^h$

Note, that the correction scheme to solve $A u = f$ includes the relaxation of $A e = r$. The logic of the correction scheme also can be applied to the subroutine $A e = r$. This leads to a recursive algorithm, where the grids gets coarser and coarser with increasing recursion depth. The base case of the recursion ends at the coarsest level which consists of $3^d$ points. All together these equations formulate then the Multigrid algorithm:

**Algorithm 3** $u^h \leftarrow MG(u^h, f^h, S, \eta, \rho, \sigma, \mu)$

1. If $h=1/2$: Relax $\mu$ times on $A u^h = f^h$ with smoother $S$ and return.
2. Relax $\rho$ times $A u^h = f^h$ with smoother $S$.
3. $r^h = f^h - A u^h$.
4. $r^{2h} = I_{h}^{2h} r^h$.
5. $e^{2h} = 0$.
6. Loop $\eta$ times: $e^{2h} = MG(e^{2h}, r^{2h}, S, \eta, \rho, \sigma, \mu)$
7. $e^h = I_{2h}^h e^{2h}$.
8. $u^h \leftarrow u^h + e^h$.
9. Relax $\sigma$ times $A u^h = f^h$ with smoother $S$ and return.

The Multigrid method comes with four hyper-parameters: pre-smoothing steps $\rho$, post-smoothing
steps $\sigma$, the number of smoothing operations at the coarsest level $\mu$ and $\eta$ describes the number of recursive Multigrid calls to solve $Ae = r$. $\eta$ gives us different types of Multigrid methods. Choosing $\eta = 1$ gives us the commonly used V-Cycle. Other methods like W-Cycle uses multiple Multigrid calls, i.e. $\eta = 2$. In this work we focus on V-Cycles only. As the name suggest we are recursively restricting the domain to the coarsest level and prolongate it back to fine grids for error correction. Figure 29 shows a V-Cycle for a two-dimensional domain with $33^2$ points.

![V-Cycle diagram](image)

As shown, the V-Cycle consists of a 5 layer structure. With $\rho = \sigma = 2$ and $\mu = 1$ we get 17 smoothing and 4 restriction and prolongation operations in total.

2.2 The machine learning approach

The main application in this thesis is a machine learning approach to solve V-Cycle partial differential equations with jump coefficients. Our goal is to get a modified V-Cycle by a LispNet model, which can solve problems with different kind and number of jumps. All jump coefficients, mentioned in this work, are greater or equal than 1. The configuration parameters for the PDE and V-Cycle used in this section are:

- $\Omega = 33 \times 33$ with $31^2$ unknowns to solve
- $g = 1$
- $f = 0$
- $c \geq 1$
- $u^0 = $ Gaussian distribution with centre $= 0$ and standard deviation $= 0.5$
- Pre- and Post-smoothing steps $\rho = \sigma = 2$
- Coarse smoothing steps $\mu = 1$
- Jacobi Smoother $S = S_J$ with $w = 0.6$
- Stopping criterion: $\frac{||r||}{||r_0||} \leq 10^{-10}$.

2.2.1 Related work

There are several works which use a machine learning approach to solve partial differential equations. Katrutsa et al. showed in their work "Deep Multigrid: Learning prolongation and restriction
Ae layers. Furthermore, they added Galerkin coarsening for the relaxation of Smoothing, residual, restriction and prolongation modules are implemented by the means of dense grid version of the Multigrid method. This requires one restriction and prolongation operation. Considered as a subroutine for the Multigrid algorithm. Unlike in this work, they chose a two-transformed their geometric Multigrid solver to a deep neural network, where each layer can be matrices" [22] a deep learning ansatz for learning restriction and prolongation operators. They transformed their geometric Multigrid solver to a deep neural network, where each layer can be considered as a subroutine for the Multigrid algorithm. Unlike in this work, they chose a two-grid version of the Multigrid method. This requires one restriction and prolongation operation. Smoothing, residual, restriction and prolongation modules are implemented by the means of dense layers. Furthermore, they added Galerkin coarsening for the relaxation of $\frac{r}{h}$. Similar to this work, they used a dumped version of Jacobi method for smoothing. In order to train the network, they choose an unsupervised machine learning approach. The loss is determined by the spectral radius of the iteration matrix. Inspired by Gelfand's formula, they used a specific norm of the iteration matrix to estimate the spectral radius. They used Automatic Differentiation to learn better prolongation and restriction matrices. They initialized these operators by common well-known Operators such as $I_{2}^{h}$. However, using these operators as a starting point for training on high frequent problems like Helmholtz equations lead to a convergence greater than 1. Consequently, they used a technique called Homotopy initialization, where the start points of the restriction and prolongation matrices are iterative learned by training on combination of different PDEs. For instance, learned operators for the Poisson Equation can be used as a starting point for solving one-dimensional Poisson, Helmholtz and singularly perturbed convection-diffusion equations. Another related work was published in 2019 by Greenfeld et al. [23]. They used a residual network with 100 dense layers to predict a prolongation operator. Similar to the work of Katrutsa et al. they make use of the two-grid algorithm. However, in this case the prolongation matrix is used to Prolongate and Restrict for a specific PDE with jump coefficients. The coefficients are drawn form a distribution. Unfortunately, the authors do not deliver more relevant information regarding the jump height or number of jumps. The output of this network are the stencil values for the four nearest neighbours to build the prolongation matrix. The cost function is built on top of the spectral radius of its error propagation matrix. In short, they developed a mechanism where prolongation matrices can be predicted for different kind of PDEs. The quantitative results show, that the network developed good generalisation regarding grid size and boundary conditions. Both papers inspired the main related work in this thesis. In 2021 Philipp Suffa published his work in "Learning Optimal prolongation and restriction Operators for Multigrid PDE Solvers" [21]. He combined several concepts from previous works. In this work the main focus was on learning restriction and prolongation matrices for two-dimensional PDEs with complex jumps. He wrote a two-grid Multigrid solver in TensorFlow. The Smoothing and Residual modules are implemented with convolutional layers. Similar to Katrutsa et al. prolongation and restriction is done via sparse matrix-matrix multiplication. Prolongation weights are used as transposed Restriction weights regarding equation 32. The goal is to find 9 different stencil values for every inner point in the grid which describe the restriction and interpolation. The euclidean norm of the residuum after one iteration is used as loss function. Their network is trained on six different configurations. Each configuration has a specific right-hand side, boundaries and coefficient function. The initial guess is sampled in the same way as in this work. It is worth mentioning, that the training of the network does not include multiple configurations at the same time. Each configuration was trained separately with a set of 2000 samples. The quantitative results with trained prolongation and restriction operators show better convergence then Geometric Multigrid (GMG), Galerkin coarsening and Algebraic Multigrid. However, the trained network lacks generalization. Although the networks show good results for different right-hand sides and boundary functions they do not perform well for different kind of jump coefficients. The reason for that is, that each network is trained only one specific jump function. Consequently, using a trained network for a jump at another position different to the learned jump leads to worse convergence. In fact, this leads to divergence in most of the test cases. The main problem, is that the network architecture is designed in a way to learn for one specific jump function. By implementing prolongation and restriction through a fully connected layer makes each stencil value position dependant. The network therefore overfits optimal restriction and prolongation values for each position independently. Their tests show, that this ansatz is not able to solve PDEs with jumps of different types and heights. Furthermore, the number of jumps is also another problem for this network. The trained networks often lead to a convergence greater than one when more than six jumps are used. In this work we focus on these problems. We provide another machine learning approach to train a single network, which is able to solve such problems. In addition, our model performs a complete V-Cycle, i.e. we use a 5-layer
architecture shown in figure 29 instead of the two-grid method used in related works. Furthermore, our model consists of less smoothing operations than the method shown by Suffa et al. The key idea behind our architecture is, that we do not train prolongation and restriction operators based on fully connected layers. Instead, we use convolutional layers to learned optimized restriction operators for a subroutine named "Coefficient Restriction". This is further explained in subsection 2.2.4. At the end we provide qualitative and quantitative results, which show, that our model is able to solve such problems with a good convergence.

2.2.2 Dataset

In order to achieve a good generalisation of jump functions we need to train our models with a dataset containing many different coefficient functions. Our two-dimensional space consists of \(33^2\) discrete grid points. Therefore our jumps functions are \(33 \times 33\) big images with one channel. The coefficients are sampled by a specific configuration function \(C_i\) within the unit space. We use two different datasets. One dataset contains coefficient functions sampled from 9 \((C_0, ..., C_8)\) different configurations. These configurations are used for training, validation and testing. The set used for training has 50 samples for each configuration, which makes 450 different jump functions in total. For validation and testing we use 225 jump functions each. The main purpose of this dataset is to fit our model for jumps at different positions, shape and height. Therefore we need many different random generated types of a configuration to prevent overfitting and enhance the generalisation. One of the most important configurations are shown in figure 30. We see four different jump functions sampled from \(C_0\) to \(C_3\). In the first configuration we train on vertical jumps. This is similar to the main configuration used by Suffa et al. In order to generate these images we define our configurations through non-continuous functions. \(C_0\) is defined by the following:

\[
C_0(x, y) = \begin{cases} 
R_1, & \text{if } x \leq 0.5 + R_2 \\
1, & \text{otherwise}
\end{cases}
\]  

(33)

\(R_1\) is a random integer sampled from a uniform distribution between \([8, 15]\). This variable generates jumps of different heights. The jump height therefore is given by \(R_1 - 1\). Note, that the average jump height is 10.5. A geometric Multigrid solver, like the V-Cycle we use, can not solve this configuration for a jump height greater than 10. Hence the higher the jump the harder it is to solve the PDE. \(R_2\) is another random floating point number in the range of \([0, 0.2]\). By shifting the jump position we train our models for position independence. As a type of data augmentation we apply a vertical and horizontal flip to each configuration for 50% of the samples. Configuration \(C_2\) also can be considered as augmentation by rotating \(C_0\) around 90 degrees, i.e. \(C_0(x, y) = C_2(y, x)\). Both
configurations only use one jump. Since most of the image area does not include discontinuous jumps there could be an problem imbalance. Therefore we increase the frequency by adding two additional jumps to \( C_1 \) and \( C_3 \). \( C_1 \) can be written as:

\[
C_1(x, y) = \begin{cases} 
R_1, & \text{if } x \leq 0.25 + R_2 \\
1, & \text{if } 0.25 + R_2 < x \leq 5 + R_2 \\
R_1, & \text{if } 0.5 + R_2 < x \leq 0.75 + R_2 \\
1, & \text{otherwise}
\end{cases}
\] (34)

The concepts of these four configurations also are applied to configuration 4 to 7. As in figure 31 shown these configurations are designed to train our models for diagonal jumps. They can be considered as a rotated version of \( C_0 \) to \( C_3 \). The function describing \( C_4 \) is defined by the equation:

\[
C_4(x, y) = \begin{cases} 
R_1, & \text{if } x \leq y + R_2 \\
1, & \text{otherwise}
\end{cases}
\] (35)

Since our models should be to also solve PDEs with no jumps, we add a constant coefficient function which returns one for every point in the grid. The general idea behind the first dataset is, that we solve different shapes of jumps by training on vertical, horizontal and diagonal jumps. Different jumps with other shapes can be constructed by combining these three types. This is tested by our second dataset. This set is designed to test the generalisation of our trained models. In fact, the models never have seen these configurations during training. The set contains seven new configurations of which 50 samples are generated each. They can be seen in figure 32. Configurations 9 to 12 are combinations of \( C_0 \) to \( C_3 \). Thus we generate square formed subareas in the coefficient function. While in \( C_9 \) and \( C_{10} \) we are testing on jump height 10, each subarea has a different height value. Each height is a random integer between \([1,11]\). Although \( C_{11} \) and \( C_{12} \) look more complex than \( C_9 \) and \( C_{10} \) they are actually a less harder problem, since the average jump height between subareas is smaller. It also can be considered as two-dimensional step function with more stairs in between. In configurations 13 to 15 we test on jumps generated by circle functions. Configuration 13 is defined by equation:

\[
C_4(x, y) = \begin{cases} 
11, & \text{if } \sqrt{(x - R_3)^2 + (y - R_4)^2} \leq 0.5 \\
1, & \text{otherwise}
\end{cases}
\] (36)
where \( R_3 \) and \( R_4 \) are random variables determining the position of the circle. In \( C_{14} \) and \( C_{15} \) we increase the frequency by adding two separate circles with a radius of 0.25. In addition, we differ between non-intersecting and intersecting circles in configuration 14 and 15. The edges of a circle in discrete image are simply horizontal, vertical and diagonal lines in different directions. Hence, these jump functions are used to test how well our models can combine different jump configurations.

The data types of all coefficient functions, initial guesses, weights and right-handsides are floating point numbers with double precision. Since a V-Cycle takes an initial guess \( u_0 \), a jump function \( c \) and a right-handside the model input has the dimension batch-size \( \times \) 33 \( \times \) 33 \( \times \) 3. We encode the boundary function \( g \) into the initial guess by setting all edge points to one. The values for the initial guess is sampled by uniform distribution with a centre of zero and standard deviation of 0.5. Our right hand-side is simply a grid full of zeros, since \( f = 0 \). The related works show that their trained networks in general achieve good results regarding different right-handsides and boundary functions. Therefore we only focus on complex jump functions for training and testing.

2.2.3 V Cycle implementation

In this subsection we explain how the V-Cycle algorithm can be implemented in LispNet. As in section 2.1.4 this method requires four modules: The Jacobi smoother \((S_J)\), the calculation of the residuum \((R)\), restriction \((I_{2h}^h)\) and prolongation \((P_{2h}^h)\). We implement each of these modules by creating a custom LispNet layer. The Jacobi relaxation and residuum computation follows equation 26. Layer \( S_J \) takes an initial guess \( v \), the right-handside \( f \), and the jump function \( c \) as input. To implement the smoothing operation we either could use a non-trainable convolutional layer or we use Petalisp functions. Therefore the Jacobi relaxation is written very similar to conv2d-layer. We simply apply a weighted version of a 5-point stencil to each inner point. The implementation of method call can be seen in figure 33. First we determine the interior shape of the grid. Since the Jacobi smoother also can be applied to coarser grids the interior shape depends on the spatial dimensions of \( v \). The next step is to multiply the jump function with the initial guess. In lines 7 - 12 we add the four nearest neighbours of \( v \) to \( h^2 f \) for every inner point. It is worth mentioning, that we do not iterate through every point in the grid. We rather select and slice through all neighbours and jump coefficients and compute the result by performing matrix-matrix computations.

We select each neighbours by translating the index according to its stencil position. For instance, by translating position \( i, j \) to \( i+1, j \) we select all bottom neighbours. We then divide this sum by the quadruple of the coefficient matrix \( c \). The result is a relaxed version of the initial guess for all inner points. We use lazy-overwrite to copy boundary values. Since we are using a dumped version of the Jacobi method, we simply interpolate linearly between the initial guess and the relaxed version at the end. Here \( w \) is a hyperparameter of the layer and is set to 0.6 by default.

The residuum models is implemented quite similar to \( S_J \). The purpose of this layer is to compute the residuum \( r \) according to Multigrid algorithm. It takes an approximated solution \( v \), the right-handside function \( f \) and \( c \) as input. The residual computation is described in equation 26.
(defmethod call ((layer jacobi-layer) v f c)
  ;; compute h, interior shape.
  (let* ((cv (lazy #'c v))
    (v2 (lazy overwrite v)
      (lazy #'
        (lazy #'/ 0.25 (lazy reshape c interior))))
    (lazy #'+ (lazy #' (w layer) v2)
      (lazy #' (w layer)) v))))

Figure 33: Implementation of $S_J$ in LispNet

Similar to the smoother this module also requires also a convolution of a 5-point filter. Figure 34 shows its implementation. Here we subtract all four neighbours of $cv$ from $4cv$. This result is then again subtracted from the right-hand-side in line 5. Since we encoded the boundary function at the edges of the grid, the residuum is only defined for the interior space. Hence, the error values for edge point are set to zero by overwriting an empty array with the result as shown in line 4. Modules $R$ and $S_J$ do not contain any lazy-unknowns and therefore are not trainable. This is different for the prolongation and restriction Modules.

The restriction layer gets a grid of size $n \times n$ as input an applies the restriction subroutine described in equation 29: $r^{2h} = I_h^{2h-rh}$. In a classic Multigrid algorithm it is applied on a residuum generated by the residual layer. The result is a coarser representation of input with a quarter of grid points. The equation can be considered as a matrix-matrix multiplication where the operator $I_h^{2h}$ is a matrix of size $(n/2)^2 \times n^2$. This approach is used by Suffa et al. They implemented this operation by the means of sparse matrix-matrix multiplications. At the current state of Petalisp and LispNet there are no sparse matrix operations. Although we could implement this technique with dense layers, we are using another approach. Using a sparse matrix-matrix multiplication leads to 9 restriction weights for each grid point. Fitting these restriction weights would result in an independent training for each point in the grid. Hence dense layers have a big disadvantage, since they are position dependant. Our approach is different to related works. We implement the restriction and prolongation operation by two-dimensional convolutions. For geometric Multigrid

Figure 34: Implementation of $R$ in LispNet
we can apply the stencil $A_s$ to each inner point and then sample the result with step size $2h$.

$$A_s = \frac{1}{16} \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

The big difference of this method to dense-layer implementations is, that one restriction stencil is used for every grid point. This makes the restriction module position-invariant. Therefore, the restriction-layer uses conv2d-layer as a sublayer. The implementation is shown in figure 35. To implement restriction the convolutional layer has to have one filter of size $3 \times 3$. The convolution has to operate on mode same to keep the spatial dimensions. We need disable the bias by setting key use-bias to nil. Since the filters of convolutional layers are initialized randomly by default, we have to set the values manually to $A_s$. This is done by the kernel-initializer. This keyword allows a user of LispNet to specify a custom initialization function. Here conv2d-restrict-weights is a function which returns $A_s$ as a Common Lisp array. After the convolution is done, we also need to sample the result with a twice as big stepsize $2h$. This also can be done by passing a stride-shape to conv2d-layer. Here the stride shape is $(\sim 2 \sim 2)$. At the end we copy the boundaries from the original grid into the coarser one.

Similar to the restriction module we can create prolongation $I_{2h}^h$ layers with matrix-matrix multi-

```lisp
(defmethod call ((layer restriction-layer) input) 
  (let* (((restrict-conv 
    (create-layer 'conv2d-layer (model layer) :
  (input (lazy-reshape input (transform b y x to b y x 0)))) 
  (result (call restrict-conv input))) 
  ;; copy boundaries 
  (lazy-reshape result (transform b y x 0 to b y x))))
```

Figure 35: Implementation of $I_{k}^{2h}$ in LispNet

plications. However for the same reason, that we want position-invariant layers we use convolutions. With equation 32 we know, that the prolongation can be understood as a transposed version of restriction. Instead of transposing a matrix, we use the transposed version of a convolutional layer (transposed-conv2d-layer). As in figure 36 shown we create and initialize the transposed convolution with similar parameters than the conv2d-layer within the restriction. The only difference here is padding which is set to "valid" mode. In addition we need to multiply the result of the transposed convolution by four since the stencil values are four times bigger than $A_s$.

```lisp
(defmethod call ((layer prolongation-layer) input) 
  (let* (((prolongate-conv 
    (create-layer 'transposed-conv2d-layer (model layer) :
  (input (lazy-reshape input (transform b y x to b y x 0)))) 
  (result (lazy #'* 4.0 (call prolongate-conv input))) 
  (lazy-reshape result (transform b y x 0 to b y x))))
```

Figure 36: Implementation of $I_{2k}^{h}$ in LispNet
As we now explained the implementation of the four main modules, we can use them to create a recursive V-Cycle which is shown in figure 37. The method gets the initial guess \( v \), the right-handside \( f \) and the jump function \( c \) as input. In addition we need to pass a LispNet model, since this method defines the forward pass. A instance of the class vcycle-model consists of different hyperparameters. In this work we set the number of pre- and postsmoothing steps to 2 and apply one relaxation on the coarsest level. Here the level \( l \) is determined by the spatial dimensions of the initial guess: \( l = \lfloor \log_2(x) \rfloor \). It also describes the recursive depth of this method. At \( l = 1 \) the base case is triggered and we apply one Jacobi relaxation in line 6, which is returned afterwards in line 7. In the other case we need to create a residual, restriction and prolongation layer. Lines 12-20 follow the Multigrid correction scheme. First we apply two pre-smoothing steps. Then we compute the residuum by calling the residual layer in line 14. The residuum gets restricted by the restriction layer. In line 16 we recursively call the method with zeros as the initial guess, the restricted residuum as right-hand side and the jump function. Since this called method acts on a grid with stepsize \( 2h \) we also need to sample down the jump coefficients. This is done by function coarse-func. For a geometric Multigrid this can be done by lazy reshaping with stepsize 2. The correction is then interpolated by the prolongation layer and added to the initial guess. After that we apply two Jacobi relaxations and return the result. This result can be considered as the output of the model.

The following two sections describe alternative techniques for down-sampling the jump function as a better subroutine for line 11.

### 2.2.4 Coefficient Restriction

Related works try to optimize the geometric Multigrid algorithm by finding better prolongation and restriction operators. The learned operators are then applied on the Residuum and Correction. By fitting other restriction weights one might be able to find a better solution to map the residuum to a coarser grid. In this work we follow a similar approach. However, we focus on the mapping of the jump function to coarser scales. In geometric Multigrid the jump coefficients are simply sampled with half the sampling rate. This is done by iterating through the grid and take every second grid point for each dimension. According to Nyquist–Shannon sampling theorem the sampling rate
has to be greater than twice the maximum frequency to reconstruct a fine grid from a coarse one. The sampling frequency $f_s$ is given: $\frac{x}{2}$ where $x$ is the size of one dimension. Consider a jump function in a one-dimensional grid with 33 points and 32 jumps. Then there would be one jump between each neighboring pixel. Hence the jump frequency $f_{\text{max}}$ is $x - 1$ and $f_s < f_{\text{max}}$. Thus, high frequency jump functions cannot be reconstructed by a coarser grid with half stepsize. As a result, aliasing artifacts occur. Sampling a coefficient function with 32 jumps with this technique would lead to a coarser grid which does not contain any jump. Since the jump coefficients are used for relaxation, badly mapped jump functions lead to worse convergence. In addition, the sampling rate reduces exponentially with increasing recursion depth leading to an even bigger problem on the coarsest scales. Our ansatz focuses on this problem. The general idea is to filter out high frequencies before down-sampling. This is done by a $3 \times 3$ filter. This operation can be considered as the same operation as restriction if the same stencils are used. In fact for our approach we are simply restricting our jump coefficients by applying the full weight restriction operator $I_{2h}$:

$$\tilde{c}^{2h} = I_{2h}^{2h} c^h$$

(37)

A full description of this module can be seen in figure 38. $C_l$ determines the jump coefficients at grid level $l$. Before the actual restriction is applied we subtract one from $C_l$ and add it afterwards to the absolute value of the restricted result. This conserves the property $C \geq 1$. Hence, this prevents to get zero as a restricted coefficient and prevents Division-by-Zero exceptions for the evaluation of equation 26. This would be unnecessary if the operator uses restriction weights $A_h$ but is mandatory for other weights. We use machine learning to optimize this subroutine via LispNet. We add four Coefficient Restriction (CR) modules to our V-Cycle each handling a different grid level. We use $I_{2h}$ as a starting point and fit its stencil weights via gradient descent. The implementation is quite similar to figure 35 but we have to set the trainability of the convolutional layer to true. It is important to mention, that we only modify the jump coefficients for layers 1 to 4. Since these coefficients are also used for the residuum we have to keep the coefficients at the finest grid unchanged. Thus, we do not change the problem itself and just change the V-Cycle on coarser layers. Therefore we are able to compare our results with geometric Multigrid due to the fact, that our metrics depend on the finest residuum. This is further explained in section 2.2.6.

Another way of describing Coefficient Restriction could be "Galerkin Coarseing". This method is quite similar to our approach. In the classic Multigrid we are using stencil $A_h$ for the relaxation of a grid with step size $h$:

$$A^h = \frac{1}{h^2} \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

For the discretization of a coarser grid with $2h$ we use the same stencil values with a modified step size, i.e. $A^{2h} = \frac{1}{4} A^h$. Using Galerkin Coarseing operator $A^{2h}$ is computed in a different way. To
be specific, $A^{2h}$ is a product of the restriction operator $I^{2h}_h$, $A^h$ and the prolongation operator $I^{2h}_h$:

$$A^{2h} = I^{2h}_h A^h I^{2h}_h = \begin{pmatrix} -0.25 & -0.5 & -0.25 \\ -0.5 & 3 & -0.5 \\ -0.25 & -0.5 & -0.25 \end{pmatrix}$$

Adding jump coefficients we get the $A^{2h}_{y,x}$ for position $x$ and $y$:

$$A^{2h}_{y,x} = \frac{1}{(2h)^2} \begin{pmatrix} -0.25 & -0.5 & -0.25 \\ -0.5 & 3 & -0.5 \\ -0.25 & -0.5 & -0.25 \end{pmatrix} \odot \begin{pmatrix} c^{2h}_{y+1,x-1} & c^{2h}_{y+1,x} & c^{2h}_{y+1,x+1} \\ c^{2h}_{y,x-1} & c^{2h}_{y,x} & c^{2h}_{y,x+1} \\ c^{2h}_{y-1,x-1} & c^{2h}_{y-1,x} & c^{2h}_{y-1,x+1} \end{pmatrix}$$

where $c^{2h}_{y,x}$ are the native down-sampled jump coefficients at position $2hy, 2hx$ and $\odot$ is an element-wise multiplication.

Besides the normalization $C \geq 1$ the stencil for Jacobi relaxation with Coefficient Restriction can be written as:

$$\tilde{A}^{2h}_{y,x} = \frac{1}{(2h)^2} \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} \odot \begin{pmatrix} 0 & c^{2h}_{y+1,x} & 0 \\ c^{2h}_{y,x-1} & c^{2h}_{y,x} & c^{2h}_{y,x+1} \\ 0 & c^{2h}_{y-1,x} & 0 \end{pmatrix}$$

where $\tilde{c}^{2h}_{y,x}$ is the coefficient restricted jump coefficient:

$$\tilde{c}^{2h}_{y,x} = \frac{1}{16} (c^{h}_{2y+1,2x-1}+2c^{h}_{2y+1,2x}+c^{h}_{2y+1,2x+1}+2c^{h}_{2y,2x-1}+4c^{h}_{2y,2x}+2c^{h}_{2y,2x+1}+c^{h}_{2y-1,2x-1}+2c^{h}_{2y-1,2x}+c^{h}_{2y-1,2x+1})$$

The scalar values represent the values of stencil $A_i$. Since we use machine learning these weights may change. As equations 40 and 39 show, both techniques look quite similar. In fact, by using different stencil weights for $A_i$ we can express Galerkin coarsening by CR. For instance, value $\tilde{c}^{2h}_{y+1,x}$ can be written as:

$$\tilde{c}^{2h}_{y+1,x} = \tilde{c}^{2h}_{y+1,x} = c^{h}_{2y+2,2x}$$

if we set the Coefficient Restriction weights to

\[
\left( \begin{array}{ccc} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array} \right).
\]

This can be done for all neighbours on the coarse grid expect the diagonal entries. The big difference here, is that our approach use all 9 direct neighbours of the fine grid, where Galerkin coarsening only requires the second closest neighbours $(2h)$. This makes our solution more suitable for jump configurations with high frequencies.

### 2.2.5 Attention Coefficient Restriction

Coefficient Restriction requires one restriction, which is implemented by a two-dimensional convolution with one filter of size of $3 \times 3$. Hence, these modules consists of 9 trainable parameters. Filtering all coefficients of the jump function with the same weights makes this approach position invariant. However, using a single filter for all coefficients could problematic. Since we aim to solve coefficients functions with different type, number and height of jumps, this filter has to be well optimized for big set of problems. Nine trainable weights for a single module could lead to a lack of generalization. Different types of configurations may need more than one filter to achieve good convergence results. We present another technique which adds modifications to Coefficient Restriction. In this module we applying a weighted convolution of six different filters. The idea is to shift the attention of the network to the jump function, so it can differentiate between different types of configurations for each pixel. Hence, we call this ansatz Attention Coefficient Restriction (Att-CR). Attention first was designed to optimize Seq2Seq translation networks [24]. The key is to weight different parts of features with a score which is commonly named attention coefficient to enhance salient features in regions of interest. In 2018 Schlemper et al. [25] used soft attention gates in Sononet, which are able to localize salient regions. They added attention gates to help the network preserve more local information and provide a guided forward pass. Similar to their approach we apply a combination of six different filters depending on local feature information within the jump
Figure 39: Attention Coefficient Restriction

function. A detailed description of this module can be seen in figure 39. First we split in the input into two paths. One path computes the attention coefficients $\alpha$ and the other uses these scores to apply a weighted convolution of six filters. To generate the attention coefficients we simply use edge detection techniques. This is done again by convolutional layers. Consider the jump function as an image where one pixel represents a jump coefficient. They idea is to extract different kind of edges of the image by using different edge detection filters. We use six filters of size $3 \times 3$ which are implemented by a conv2d-layer module. We initialize its weights by Xavier/Glorot-uniform initialization. Before we extract the features from the image we additionally normalize the input by dividing by the highest coefficient, since convolutional layers work better for a normalized space. The convolution generates features of size $x \times y \times 6$, where each channel shows a different portion of edge features. We apply a softmax $\varphi$ function to these features across the channels. The results are then used as attention coefficient $\alpha$. Due to Softmax activation these scores have the property:

$$\sum_{c=1}^{6} \alpha_{y,x,c} = 1$$

(43)

In other words, the feature maps describe the distribution of edge features across the channels. We then multiply the attention coefficients with the jump function. For simplicity we ignore the normalization terms for $C \geq 1$. We then get:

$$\tilde{C}_{y,x,c} = \alpha_{y,x,c} C_{y,x}$$

(44)

Due to equation 43 we also can write $\sum_{c=1}^{6} \tilde{C}_{y,x,c} = C_{y,x}$. Hence, we simply split the input jump function into a linear combination of its attention coefficients. Specific jumps or edges detected by a specific filter are highlighted in a specific channel and suppressed for other channels. The next step is to apply the Coefficient Restriction subroutine for each channel. This is done by six different filters which are initialized by full weight restriction values $A_s$. The result is added together and returned as output. This procedure can be implemented by one conv2d-layer with six input channels and one output channel. Since $\tilde{C}_{y,x,c}$ is an attention weighted version of the input the two-dimensional convolution also be seen as a linear combination of six different convolutions. We can write:

$$C^{l-1} = \sum_{c=1}^{6} \sum_{y=-1}^{1} \sum_{x=-1}^{1} w_{y,x,c} \tilde{C}_{y,x,c}$$

(45)
where \( w_{y,x,c} \) are the restriction weights for channel \( c \). Consider the first forward pass of this module during fitting. Since the same restriction weights are used for each channel, equation 45 becomes:

\[
C^{l-1} = \sum_{y=-1}^{1} \sum_{x=-1}^{1} w_{y,x} \sum_{c=1}^{6} C_{y,x,c}^{d} = \sum_{y=-1}^{1} \sum_{x=-1}^{1} w_{y,x} C_{y,x}^{d} = I_{h}^{2h} C^{d}
\]

That means, that we are applying a full weight restriction operator to the jump function like in equation 37. Therefore for the first batch within the first training epoch this modules generates the same result as Coefficient Restriction. The difference here how the result is determined. When CR with and without attention results in the same output than the same error is returned back during Backpropagation. With the error we get the gradient for the restriction weights. Since we do an attention weighted linear combination of different filters, their gradients are also multiplied with \( \alpha_c \). This means, that each filter has a different effect on the error and therefore are differently optimized during gradient descent. To summarize, we start with six restriction operators \( I_{h}^{2h} \) as starting point and optimizing the weights depending on local edge features. Since we initialize the attention kernels randomly, each channel learns a specific type of jump configuration. The different restriction weights are optimized regarding their jump configurations. This means we force the network to find a good specialization of restriction weights for each type of jump function. The network then finds a good linear combination of these filters by the means of multiplicative soft-attention mechanism. As we later show, the attention coefficients and edge detection filters are learned within a few epochs. Furthermore, training each filter for a randomly chosen specialization adds important stochastic machine learning concepts to our network, while the fitting for CR is a deterministic process. We add four of these modules to our model which results in 408 trainable parameters in total.

### 2.2.6 Cost function and training

To train our model we have to define a cost function which is optimized by Gradient Descent. We use a similar approach as in the work of Suffa et al. [21]. We call a certain number of V-Cycles on an initial guess and use the result to compute the residuum \( r^t \). We calculate \( r^t \) with a residual layer. We then can write:

\[
r_{y,x}^t = f_{y,x} - \frac{1}{h^2}(4c_{y,x}u_{y,x}^t - c_{y-1,x}u_{y-1,x}^t - c_{y+1,x}u_{y+1,x}^t - c_{y,x-1}u_{y,x-1}^t - c_{y,x+1}u_{y,x+1}^t)
\]

where \( u_{y,x}^t \) is the approximated solution after \( t \) V-Cycle iterations. Note, that we use jump coefficients of the finest grid level, so that Coefficient Restriction or other coefficient modifications on coarser grids are not effecting the loss function. To get a loss metric we define a batch-wise euclidean norm:

\[
||r^t|| = \frac{1}{b} \sum_{y=1}^{b} \sum_{x=1}^{x} (r_{y,x}^t)^2
\]

where \( b \) is the batch size. All together our cost function is defined as:

\[
C(u^0) = \frac{||r^t||}{||r^0||}
\]

\( r^0 \) is the residuum of the initial guess \( u^0 \) which is sampled from a uniform distribution. The error is different for each sample depending on the distance of the initial guess and the exact solution. Therefore different samples could have a bigger error and effect on the loss function than other samples, although the same V-Cycle and jump function is used. For this reason, we are dividing \( ||r^t|| \) by \( ||r^0|| \) to normalize the samples, so that each sample is weighted the same. Therefore we use the residuum reduction as a quantitative metric. The better the reduction the lower the loss. Since no labels are used for this cost function, we can classify the training as unsupervised learning method. We use following hyperparameters to fit our model: we use a batch size of 20 and fit for 250 epochs. We train on a dataset containing 50 samples for configuration \( C_0 \) to \( C_8 \) each. We validate
on different samples of the same configurations and use the loss function additionally as validation metric. We use Adam optimizer with a learning rate of $10^{-3}$ and default Adam parameters as described in the original work [26]. All weights and inputs are double-precision floating numbers. We initialize the restriction weights with the stencil values of $L^2_h$. Other trainable parameters such as attention kernels are sampled by Xavier/Glorot-uniform initializer. Another important parameters is the number of V-Cycles applied to the initial guess. Related results such as the work of Suffa et al. claimed that the choice of parameter $t$ does not matter. Our results do not reflect that. As in figure 40 shown, we see the training and validation loss for each epoch for $t = 1$ and $t = 5$. In addition, we calculate the batch convergence metric described in equation 49 for the first test set.

![Figure 40: Fitting of Coefficient Restriction with a) $t = 1$ and b) $t = 5$.](image)

The loss function is designed to train the batch convergence. It can be clearly seen, that training on one iteration results in worse convergence results. The reason is, that the loss function with $t = 1$ does not reflect the convergence function for increasing epochs. Both metrics simply divergence from each other. For $t = 1$ our loss is based on the residuum $r^1$. Since smoothers are more effective for high frequent errors, the residuum reduction is greater on the initial guess than for closer approximations to the solution. Training for $t = 1$ would lead to a better residuum reduction for the first iteration. However, this could lead to overfitting of the residuum reduction for the initial guess and worse reductions for next iterations. Hence, we are overfitting the V-Cycle for one specific iteration resulting to a worse batch convergence. On the other hand, setting $t = 5$ results in a better correlation between convergence and loss. Although the quantitative value of the loss is greater as for $t = 1$, the convergence is better. Increasing $t$ leads to a better representation of the convergence function. However, increasing $t$ leads to a longer computation path and therefore to increased memory consumption and training times. Furthermore, such deep networks suffer from vanishing or exploding gradient problems. This is why we choose to set $t = 5$ as good sweet spot.
2.2.7 Qualitative Results

In this section we analyze our methods for qualitative results. We extract trained parameters of CR-learned and Att-CR. Furthermore we investigate activation and attention maps produced by the attention mechanism. At the end we compare all methods by applying CR, CR-learned and Att-CR successively on a coefficient function. First we look at CR-learned and its restriction weights. Since four restriction modules are applied on four different levels \( l \) we get four different filters of size \( 3 \times 3 \). Figure 41 show these filters. Note, that each filter started as operator \( I_h^2 \). The initial weights \( A_s \) for this operator also are shown in Figure 41. It can be seen, that all filters look very similar to \( A_s \). Filters of all levels show a high weighting of the middle point while grid points further away from the centre are weighted less. This is especially true for coarse grid levels \( l = 3 \) and \( l = 2 \). Here neighbored points are weighted close to zero. The models seems to reduce the effect of CR the deeper the V-Cycle recursion gets. Furthermore, the sum of all weights are shown. While \( I_h^{2h} \) is normalized, learned weights of CR do not sum up to one. This is important to analyze, since this sum can be considered as average scale factor of the function. Grid levels \( l = 5 \) and \( l = 4 \) show a quite close factor to one. The coarser levels however result in a scaling of 76% and 22.4% of the initial function. Since the coefficients are used for relaxation we can consider this scaling as a weighting similar to \( w = 0.6 \). Thus, we apply a similar type of under-relaxation which can help to stabilize the V-Cycle.

Trained parameters of Att-CR look differently. Att-CR consists of six edge-detection filters for attention calculation and six filters for restriction. Figure 42 show the weights for a) edge detection and b) attention weighted restriction. These 6 different filters are labeled as separate channels.

![Figure 41: Trained restriction weights for CR](image)

![Figure 42: Trained a) attention and b) restriction filters of Att-CR at level \( l = 5 \)](image)

Attention kernels are randomly initialized and trained to detect specific edge features. For instance, we see that channel two became a detector for horizontal jumps. Channel four becomes a vertical jump filter and channels five and six transform to diagonal edge detection kernels. It is hard to see, what channel one and three does, but it is more clear if we further analyze the edge activation maps generated by these filters. In b) we can see the trained restriction weights. Each kernel transformed to an optimized version for a specific configuration. Although all kernels are initialized with \( A_s \) they look very differently after training. This is because each kernel is trained to solve a specific configuration. The attention maps and edge feature decide which channel corresponds to which type of jump function. Further analysis we extract these maps for a forward pass. As in figure...
43 shown we apply Att-CR on a jump function which is sampled from configuration 10. In a) we see the features generated by the attention filters and in b) the attention coefficients $\alpha$ for each channel. Note, that $\alpha$ is computed by applying a channel-softmax function to the edge features of a). Thus the attention maps show a distribution of different edge localizations. As discussed channel two detects all horizontal edges while channel four mainly shows vertical activations. Channel five and six result also in small activations near jumps. The biggest activations however, are given by channels one and three. Here most of the activations are located in regions where $C = 10$. In addition channels one provides more activations near vertical and horizontal edges. As shown these channels represent specific types on configurations. Based on $\alpha$ Att-CR applies different restriction filters. In b) we see that coefficients of channel 3 and are weighted the most. A look into figure 42 tells us, that most coefficients are similar restricted to CR. However, the model is able to localize different kind of jumps and applies small modifications to the Coefficient Restriction mechanism for these regions. To see this effect, we need to look at the results of this method. As shown in figure

Figure 43: a) Edge activation and b) attention maps of Att-CR at level $l = 5$

Figure 44: Coefficient function $C_{10}$ for different grid levels with a) native down-sampling, b) CR, c) trained CR and d) Att-CR

44 shown, we apply four modules of a) native down-sampling b) CR, d) CR-learned and d) Att-CR
to the coefficient function \( C_{10} \). For native down-sampling aliasing effects artifacts like miss-aligned edges and badly stretched areas occur the coarser the grid gets. On level \( l = 1 \) we only see one pixel representative for all areas with \( C = 1 \). This is particularly the case for Configurations 13, 14 and 15 where circles are added to the jump function. With native down-sampling these shapes transform to squares already on \( l = 3 \) and lower. Most of the issues are solved by applying filter \( A_s \) to the image in the CR subroutine. Here edges are smoothed out leading to a better representation of the jump function especially at \( l = 3 \) and \( l = 2 \). In c) we use CR with weights, which are trained for 250 epochs. The resulting images look very similar to b). It is also smoothing the edges in the same way as CR does. The reason for that, is that the restriction kernels are not much different to the full weight restriction stencil \( A_w \). However, as in figure 41 described we see that the jump function is scaled down on levels \( l = 2 \) and \( l = 1 \). In fact at level c) we barely can see any jump. This results in a sort of stabilization during relaxation on these grid levels. This also applies for Att-CR in d). Here Att-CR generates very similar results to CR-learned. However, we clearly can see, that Att-CR behaves different for \( l = 4 \) and \( l = 3 \). At level four there are increased coefficients near edges. While CR smooths out the edges, Att-CR applies a kind of sharpening by increasing the jump height. At \( l = 3 \) Att-CR is even highlighting coefficients up to 12.5. It can be concluded, that Att-CR is behaving similar to the CR subroutine. In addition the model localizes specific edges types on fine grid levels and applies modifications like sharpening and coefficient scaling within these regions. The reason for that, may be, that some jump functions like the example may be well represented on fine grid levels. Smoothing out edges is not necessary if no aliasing effects occur near edges. Therefore it seems like, that the model learns to revert the Coefficient restriction for specific areas. From a qualitative standpoint, it is hard to tell, which method works the best. Thus, we have to look at quantitative metrics which are now discussed in the next section.

### Quantitative Results

To compare our different methods regarding quantitative results we use the average convergence:

\[
\text{Convergence } \theta = \frac{1}{n} \sum_{l}^{n} \frac{||r^l||}{||r^{l-1}||} \tag{49}
\]

Note, that \( ||r^l|| \) is the batch-norm of the residuum. We test the Jacobi solver, geometric Multigrid (GMG), Multigrid with Coefficient Restriction (CR), CR with learned weights (CR-learned) and Attention CR (Att-CR) on different configurations. We use all samples from a configuration as batch to determine \( ||r^l|| \). We test on two different test sets. One set contains 25 samples each for jump functions \( C_0 \) to \( C_8 \). The second set consists of 50 jumps of \( C_0 \) to \( C_{15} \) each. Since we train on configurations \( C_0 \) to \( C_8 \), the second test set is more important for quantitative analysis regarding generalisation. We set \( n \) based on a specific stopping criterion. We stop the iteration when \( \frac{||r^l||}{||r^{l-1}||} \leq 10^{-10} \). Our measurements are shown in tables 45 and 46.

We see, that applying a normal Jacobi solver with \( w = 0.6 \) can solve all configurations. However, a convergence of 99.6% is quite bad, which results in multiple hundred’s of needed iterations to achieve the target residuum reduction. The geometric Multigrid we use (V-Cycle) diverges for all configurations which contain at least one jump. Note, that configuration \( C_8 \) is a jump function without any jumps, i.e. \( C(y, x) = 1 \). Here we get a convergence of 23.2% for geometric Multigrid methods. We get the same results also for our Coefficient Restriction approaches, since we are only restricting jumps. Therefore CR, CR-learned and Att-CR result in the same convergence and act like a V-Cycle for configurations without jumps. Restricting coefficients for coarser grid levels leads to a much better convergence than GMG. The results show the CR is able to solve all configurations

<table>
<thead>
<tr>
<th>( C_i )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>0.996</td>
<td>0.996</td>
<td>0.996</td>
<td>0.996</td>
<td>0.996</td>
<td>0.996</td>
<td>0.996</td>
<td>0.996</td>
<td>0.996</td>
</tr>
<tr>
<td>GMG</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
<td>0.232</td>
</tr>
<tr>
<td>CR</td>
<td>0.304</td>
<td>0.342</td>
<td>0.301</td>
<td>0.349</td>
<td>0.274</td>
<td>0.302</td>
<td>0.273</td>
<td>0.303</td>
<td>0.232</td>
</tr>
<tr>
<td>CR (learned)</td>
<td>0.24</td>
<td>0.264</td>
<td>0.277</td>
<td>0.278</td>
<td>0.222</td>
<td>0.244</td>
<td>0.264</td>
<td>0.273</td>
<td>0.232</td>
</tr>
<tr>
<td>Att-CR</td>
<td>0.238</td>
<td>0.246</td>
<td>0.244</td>
<td>0.245</td>
<td>0.202</td>
<td>0.201</td>
<td>0.208</td>
<td>0.21</td>
<td>0.232</td>
</tr>
</tbody>
</table>

Figure 45: Convergence for configurations \( C_0 \) to \( C_8 \)
Table 1: Convergence for configurations C_i

<table>
<thead>
<tr>
<th>C_i</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>0.996</td>
<td>0.996</td>
<td>0.996</td>
<td>0.996</td>
<td>0.996</td>
<td>0.996</td>
<td>0.996</td>
</tr>
<tr>
<td>GMG</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>CR</td>
<td>0.308</td>
<td>0.342</td>
<td>0.3</td>
<td>0.327</td>
<td>0.27</td>
<td>0.294</td>
<td>0.277</td>
</tr>
<tr>
<td>CR (learned)</td>
<td>0.229</td>
<td><strong>0.25</strong></td>
<td>0.214</td>
<td>0.243</td>
<td>0.24</td>
<td>0.23</td>
<td>0.23</td>
</tr>
<tr>
<td>Att-CR</td>
<td><strong>0.227</strong></td>
<td>0.26</td>
<td><strong>0.21</strong></td>
<td><strong>0.241</strong></td>
<td><strong>0.218</strong></td>
<td><strong>0.218</strong></td>
<td><strong>0.222</strong></td>
</tr>
</tbody>
</table>

Figure 46: Convergence for configurations C_9 to C_{15}

with a significant convergence improvement. On average CR shows a convergence of 30%. Applying machine learning to CR results again in an improvement for all configurations with jumps. We get an average convergence of 24.5% leading to 18.3% improvement. This again is outperformed by CR with trained attention layers. Besides configuration C_{10}, Att-CR achieves better results than CR and CR-learned. With an average convergence of 22.6% we get an additional improvement of 7.8%. It is worth to mention, that Att-CR is able to get a better convergence for configurations with jumps than all other tested methods for a constant jump function like C_8 (0.232). Furthermore, all methods show quite similar results for both test sets. This shows, that CR, CR-learned and Att-CR are able to solve configurations for many different type, number and position of jumps. Another test for generalisation and stability for our models are the number of jumps. In most of configurations we use a combination of frequencies, which result in two to four jumps for each spatial dimension.

Figure 47: Convergence for multiple jumps

In the following we test all CR variants on higher frequencies by adding more horizontal jumps. The jump height for all jumps is 9. Figure 47 shows the convergence for configurations with zero to 32 jumps. Our measurements show, that Coefficient Restriction and its machine learning variants are stable regarding the number of jumps. On average the Coefficient Restriction with optimized weights work the best. Att-CR slightly outperforms CR-learned for less than 16 jumps but is worse for higher frequencies. Besides the number of jumps we also can analyze how well these methods work for different jump heights. Figure 48 shows the convergence for a configuration with a single horizontal jump at x = 0.5. The jump height is chosen between 2^9 − 1 to 2^{256} − 1. It can be seen, that CR and Att-CR are stable methods for configurations with very high jumps. Att-CR show the best results between jump height 0 and 256. For higher jumps CR without machine learning performs slightly better with an average convergence of 31%, while Att-CR achieves a convergence of 33%. CR-learned however leads to convergence results greater than 1 for jump heights above 256 and therefore is unable to solve such configurations. Since we train our models for only a small jump height between 8 and 15, overfitting regarding jump height may be happened. This results in instability for jumps heights above of trained input space. In general, our models, which are based on convolutions, work well for different types of jumps in a normalized input space. Due to position invariance all approaches solve problems with different form, number and position of jumps. However, we see a lack of generalization regarding jump heights, especially for native Coefficient Restriction with
Our results outperform the methods used in the main the related work by Suffa et al. [21]. Their work use similar configurations and Multigrid parameters to our setup. However, they rely on learned restriction and prolongation operators for the two-grid method. Since they trained their models only on one specific configuration, their models lack generalization for different kind of configurations. Moving the jump, adding more jumps and increasing jump height leads to bad and diverging convergence results. In short, the network was not able to obtain good generalization for different configurations. It is not easy to compare our quantitative results to theirs, because over 300 smoothing operations are used for their method, while we are relying only on 17 relaxations in total. The more relaxation the better convergence. For instance, one of their main tested jump configurations look like $C_0$. They achieved an average convergence of 26.2% for their model, which was specifically trained for this configuration. Models trained on other configurations were not able to solve $C_0$. Our models learned to solve many different configurations at the same time. Furthermore, the average convergence of CR-learned and Att-CR is lower than 26.2%, although we only use 17 smoothers. In terms of convergence per relaxation their optimized two grid-method achieved a convergence of 99.55%, which is very close what native Jacobi solvers get. With Att-CR on the other hand we get convergence values of about 92% per smoothing operation, which is an improvement of 7.5% for each relaxation step. This advantages exponentially rises by the number smoothing steps applied within the Multigrid method. Trained models used in the work by Suffa et al. get bad convergence values when more jumps are added to the configuration. Furthermore, it was not able to solve configurations with more than five jumps consistently. Our results show, that this problem can be solved by using position-invariant convolutional layers instead of dense layers as restriction operators. Regarding the jump height their models showed good convergence within a height of one to 10, but heavily gets worse for higher jumps. For jumps greater than 400 it was not able to solve the problem most of the times. This reflects also on our work. Although we achieved better and stable convergence results, it turns out, that machine learning is not efficient enough to learn for different jump heights.
3 Conclusion

To summarize our work we present a high performance machine learning framework for Petalisp. It allows the use and customization of different kind of models, layers, metrics and other machine learning concepts. We show the concepts of a Python interface which generates LispNet code from custom Keras/TensorFlow models, so that a user can run LispNet as backend within a Python environment. With the means of Petalisp we can built optimized computation graphs for the forward and backward pass. However in the current state of Petalisp LispNet does not outperform state-of-the-art frameworks like TensorFlow. Here cache optimized subroutines and vectorization for Petalisp is needed to achieve better results. Therefore the success of LispNet depends on Petalisp. Since Petalisp and LispNet are under active development we hope, that both frameworks will include more features and show better performance results in future versions. As a Proof of Concept we applied LispNet to the MNIST handwritten digits problem. Furthermore, we use LispNet to optimize Multigrid algorithms in the second part of this work. The goal is to solve partial differential equations with jump coefficients. We present a LispNet model which implements a VCycle by using a custom LispNet layer for each Multigrid subroutine. Furthermore, we add a new method for mapping fine jump coefficients to coarser grids in a subroutine name "Coefficient Restriction". We apply machine learning to optimize this approach by training these restriction filters. In order to achieve a better generalisation of jump functions we also test a more complex version of it. By adding attention gates and an attention weighted linear combination of six different restrictions we further improve the model. Our qualitative results show, that Attention based CR uses local edge information of the jump function to learn configuration specific restriction filters. The learned restriction operators for native CR look very similar to the full weight restriction operator. On the coarser scales however these weights became a scaled down version of the injection operator to stabilize the V-Cycle. The coarse grids generated by Att-CR look similar, but on fine levels the attention mechanism applies different edge specific operators. Our quantitative results show, that V-Cycle with CR is able to solve complex Multigrid problems with a relative good convergence. Using the concepts of machine learning we get better results, so that Attention gated CR works best in this work. A single trained model is able to solve all tested jump configurations with an average convergence of 22.6%. Our method outperforms other models used in related works. They are stable for different number of jumps. Regarding jump height we still see a lack of generalization. Despite that, we show, that it is possible to learn a single model to solve PDEs with coefficients containing many different type, position and number of jumps.
References


