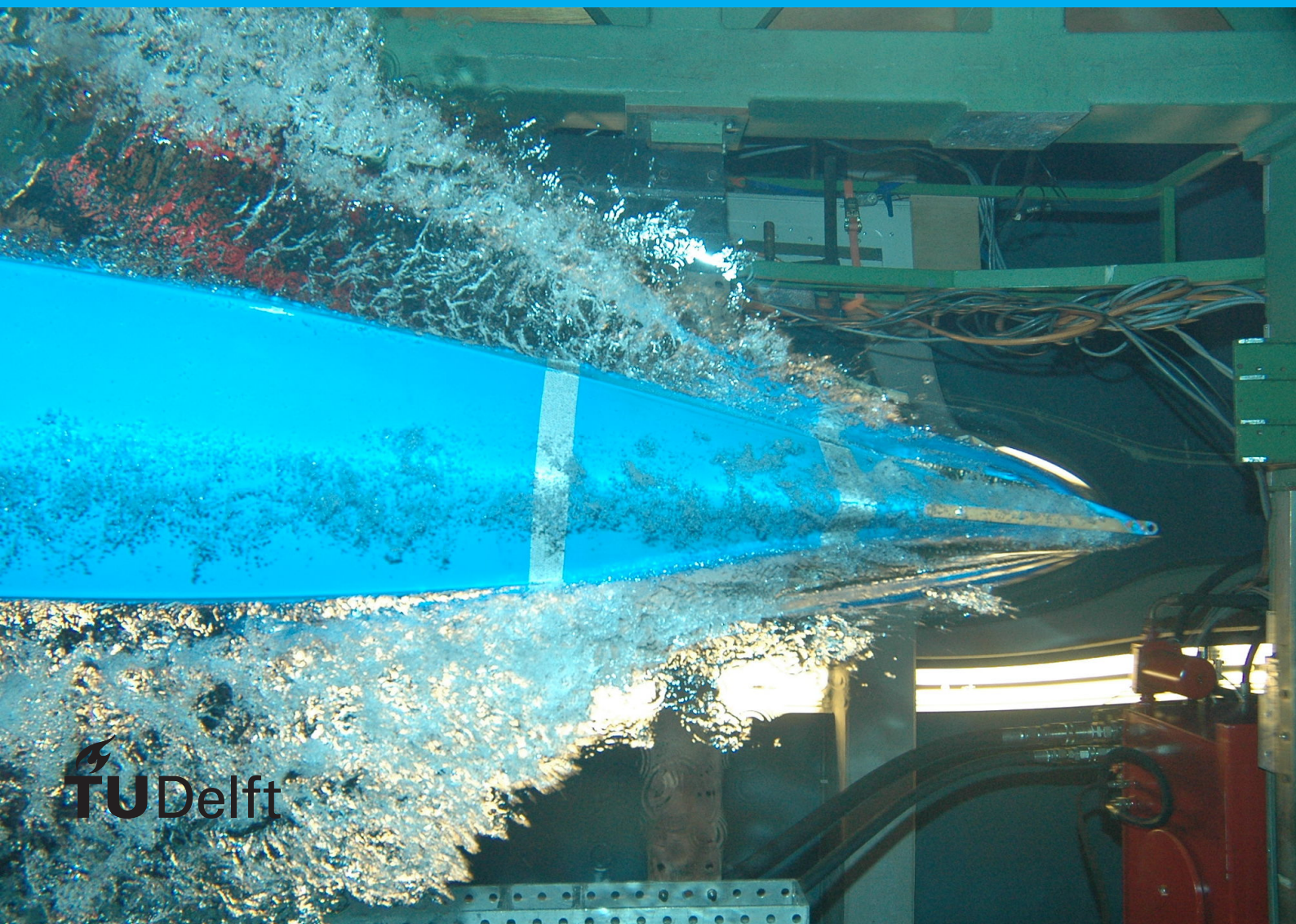
Error Estimates for Finite Element Simulations Using Neural Networks

Bachelor Thesis

A.L. Halevy

Faculty of Numerical Analysis
Delft University of Technology

July 21, 2022



Error Estimates for Finite Element Simulations Using Neural Networks

Bachelor Thesis

by

A.L. Halevy

to obtain the degree of Bachelor of Science

at the Delft University of Technology,

to be defended publicly on Tuesday July 19, 2022 at 14:00.

Student number:	5175208	
Project duration:	April 18, 2022 – July 19, 2022	
Thesis committee:	Dr. Alexander Heinlein,	TU Delft, supervisor
	Deepesh Toshniwal,	TU Delft, supervisor
	Dr. D.J.P. Lahaye,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Contents

1

Introduction

Solving Partial differential equations exactly is often times not possible, this is then done numerically. One such numerical method is the Finite Element Method, how this method works will be briefly explained in section 2.1. The accuracy of the solution obtained by applying the Finite element method can be increased but usually at the cost of needing more compute and thus getting a solution will take more time. Roughly speaking, the finite element method works by partitioning the domain into elements. These elements can be piecewise linear lines in case of 1d problems, triangles in the case of a 2d problem or tetrahedra in the case of 3d problems.

The number of elements will determine how much compute and thus how long it will take to get a solution of the Finite Element Method, but also how accurate the solution will be. The more elements in a subset of your domain the less error the solution will have in that part of your domain. Thus more is gained when increasing the number of elements in parts of your domain where the error is high than when the number of elements is increased in a part of the domain with little error.

Hence if the error can be estimated in advance the number of elements can be increased where the error estimate is high and can be kept the same where the error estimate is low, this way the amount of compute needed for finding a solution will be lower than when increasing the amount of elements uniformly while having a similar effect on the accuracy of your solution. Using error estimates to refine a mesh where the estimated error is high is called adaptive mesh refinement, and the effectiveness of this method depends strongly on how accurate the error estimates are. In this thesis will be looking at how to estimate errors using the finite element method using neural networks. These errors are estimated per element using information from the element itself and of neighbouring elements.

In this thesis the focus lies on one dimensional differential equations. But the ideas can generalise to arbitrary dimensions. However other neural network architectures might be better suited for this. This will be discussed in section 8.2. Regarding the neural networks the thesis will focus on two architectures the multilayer perceptron and the Transformer network [15]. How these networks work will be discussed in detail, and will be shown in pseudocode in the main text in section 2.2. Furthermore, how adaptive mesh refinement is being done will be discussed in section 2.3.

The hyperparameters of neural networks will for this thesis also be tuned, how this works and what this is will be treated in the thesis too, in section 2.4.

2

Background theory

2.1. Finite element method

The finite element method consists of 4 steps that in general take the following form:

- Apply the weak formulation to the PDE. The problem that is obtained after applying the weak formulation is in the literature formulated as:

$$B(u,v) = F(v), \forall v \in V_{g_D}$$

  Where $V_{g_D} = \{v : ||v||_{L_2(\Omega)} + ||\nabla v||_{L^2(\Omega)} < \infty, v|_{\partial\Omega} = g_D\}$

- Generate the Mesh, the mesh is generated by partitioning the domain into elements an example of an element in 1d is a line segment, in the 2d case elements can be triangles and in the 3d case it can be tetrahedra.

- Define the Finite Element subspace $S^h := Span\{\phi_i\}_{i=1}^N \subset V_{g_D}$, where the ϕ_i's are the basis functions that are specified in this step. The basis functions that are used in this thesis are shown in figure 2.1 below

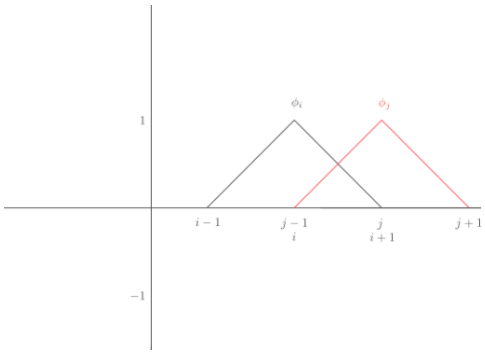


Figure 2.1: linear 1d basis functions

- Formulate the Finite Element problem

$$B(u^h, v^h) = F(v^h), \forall v^h \in S^h$$

Where:

$$u^h := \sum_{j=1}^{N} U_j \phi_j$$

$$v^h := \sum_{i=1}^{N} V_i \phi_i$$

Where U_j and V_i scalars for all $i, j \in \{1, \ldots, N\}$.

- Matrix formulation of the Finite Element problem

$$B(u^h, \phi_i) = F(\phi_i) \implies$$

$$B(\sum_{j=1}^{N} U_j \phi_j, \phi_i) = F(\phi_i) \forall i \in \{1, \ldots, N\} \implies$$

$$\begin{cases} \sum_{j=1}^{N} B(U_j \phi_j, \phi_2) = F(\phi_1) \\ \sum_{j=1}^{N} B(U_j \phi_j, \phi_3) = F(\phi_3) \\ \vdots \\ \sum_{j=1}^{N} B(U_j \phi_j, \phi_{N-1}) = F(\phi_N) \end{cases}$$

Where U_j and V_j scalars for all $j \in \{1, \ldots, N\}$. In matrix form it would be:

$$\begin{bmatrix} B(\phi_1,\phi_1) & B(\phi_2,\phi_1) & \cdots & \cdots & B(\phi_{N-1},\phi_2) & B(\phi_N,\phi_1) \\ B(\phi_1,\phi_2) & B(\phi_2,\phi_2) & \cdots & \cdots & B(\phi_{N-1},\phi_2) & B(\phi_N,\phi_2) \\ B(\phi_1,\phi_3) & B(\phi_2,\phi_3) & \cdots & \cdots & B(\phi_{N-1},\phi_3) & B(\phi_N,\phi_3) \\ \vdots & \vdots & \ddots & & \vdots & \vdots \\ \vdots & \vdots & & \ddots & \vdots & \vdots \\ B(\phi_1,\phi_{N-2}) & B(\phi_2,\phi_{N-2}) & \cdots & \cdots & B(\phi_{N-1},\phi_{N-2}) & B(\phi_N,\phi_{N-2}) \\ B(\phi_1,\phi_{N-1}) & B(\phi_2,\phi_{N-1}) & \cdots & \cdots & B(\phi_{N-1},\phi_{N-1}) & B(\phi_N,\phi_{N-1}) \\ B(\phi_1,\phi_N) & B(\phi_2,\phi_N) & \cdots & \cdots & B(\phi_N,\phi_N) & B(\phi_N,\phi_N) \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ \vdots \\ \vdots \\ U_{N-2} \\ U_{N-1} \\ U_N \end{bmatrix} = \begin{bmatrix} F(\phi_1) \\ F(\phi_2) \\ F(\phi_3) \\ \vdots \\ \vdots \\ F(\phi_{N-2}) \\ F(\phi_{N-1}) \\ F(\phi_N) \end{bmatrix}$$

This linear system can then be solved using standard linear algebra computations.[10]

2.2. Neural networks

Neural networks come in different flavours: GNN's, CNN, Transformer neural networks and the MLP, to name a few. In this thesis two architectures are trained to estimate elementwise residual errors: Transformer network and the Multilayer perceptron (MLP). Neural network architectures are very diverse and different architectures are useful for different tasks, what they have in common however is that neural networks are differentiable with respect to all its parameters. Which makes the training procedure that will be described in the next section possible.

2.2.1. Training neural networks

The goal of training a neural network is to make it perform better with respect to a certain metric. This metric is called the loss function, the loss function that is used in this thesis is the mean squared error loss also known as MSE.The MSE loss looks like this:

$$\mathcal{L}_{MSE}(\hat{\mathbf{y}}, \mathbf{y}) = ||\hat{\mathbf{y}} - \mathbf{y}||^2 / n$$

Where n is the length of the vectors $\mathbf{y}$ and $\hat{\mathbf{y}}$, furthermore $\hat{\mathbf{y}}$ is the predicted value of $\mathbf{y}$ given the corresponding input $\mathbf{x}$ by the neural network (i.e. $f(\mathbf{x}, \mathbf{w}) = \hat{\mathbf{y}}$). To train a neural network the derivatives with respect to the loss function are calculated with respect to all its parameters. These derivatives are calculated by applying the chain rule with respect to all parameters. The calculation of the chain rule is in turn done by a computer using back-propagation.

When these gradients are calculated an optimizer is used to update the parameters of the model. The optimizers that are used in this thesis are Adam and stochastic gradient descent. To speed up learning and making learning more stable the optimizer updates the parameters with gradients that come from the loss from multiple input vectors $\mathbf{x}_i$, $i = 1, \ldots, N$ where N is called the batch size.

Let the neural network be called $f(\mathbf{x}_t, \mathbf{w})$, where $\mathbf{x}_t$ is an input vector and $\mathbf{w}$ are its parameters, also called weights. And let $\mathcal{L}_f(\mathbf{x}_t, \mathbf{w})$ be the loss function evaluated. Then gradient descent on batch t $\mathscr{B}_t$ of N entries does the following [17]:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{1}{N} \sum_{i \in \mathscr{B}_t} \partial_{\mathbf{w}} \mathcal{L}_f(\mathbf{x}_i, \mathbf{w})$$

Where η is called the learning rate, the learning rate determines how strongly the weights will move into the direction opposite of the gradient. Having a learning rate that is too high will result in there being no convergence as the optimizer will overshoot the minimum. Hence the loss will then not decrease. When the learning rate is too low the optimization procedure will get stuck in a local minima or on a saddle point making progress impossible or just very slow respectively. In the latter case it is because the gradient is zero at the saddle point.

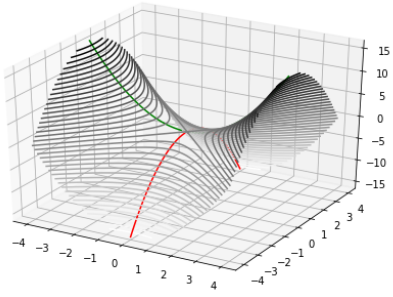


Figure 2.2: A saddle point in 3d[12]

As can be seen above graphically there are directions in which the parameters are pushed towards the saddle point but also directions in which the parameters are pushed away from the saddle point, eventually the parameter vector will diverge away from the saddle point provided the parameters are not exactly on the saddle point itself. The picture above is in 3d but this is of course also true in higher dimensions as well.

During hyper-parameter tuning a different variation of stochastic gradient descent is tried as well. Namely SGD with momentum and dampening. Adding momentum to SGD as the name suggests will make it less likely that whenever you get close to a saddle point that learning becomes really slow. Similarly when getting stuck in a local minima will become less likely as well since it will not immediately stand still. The opposite however can happen as well, that the weights are optimal but then because of momentum will diverge from the optimum again in the next optimization step. However in practice it is found that adding momentum usually helps.

Momentum works by adding the gradient of the previous timestep $g_{t-1} := \frac{1}{N} \sum_{i \in \mathscr{B}_{t-1}} \partial_{\mathbf{w}} f(\mathbf{x}_i, \mathbf{w})$ times some factor μ to the gradient update. The exact expression is shown in algorithm 1.

Adding dampening to SGD will do the opposite, it will make the effect of the gradient $g_t = \frac{1}{N} \sum_{i \in \mathscr{B}_t} \partial_{\mathbf{w}} f(\mathbf{x}_i, \mathbf{w})$ in the current time step less pronounced by multiplying it with a factor $(1 - \tau)$, where $\tau \in [0, 1]$.

Let t again denote the current batch. In pseudocode the SGD algorithm becomes [3]:

As can be seen dampening will have effect whenever momentum is nonzero, furthermore with momentum $\mu = 0$ the previous SGD algorithm is obtained.

The second type of optimizer that is used in this thesis is Adam. Adam is the most popular optimizer because it usually works well in practice, without there being a need to tune its hyper parameters, β_1, β_2 [9].

Algorithm 1 SGD with momentum

Input: Loss function: $\mathcal{L}_f(\mathbf{x}_t, \mathbf{w})$, learning rate γ, parameters $\mathbf{w}_0$, momentum μ, dampening τ
Output: updated parameters $\mathbf{w_t}$

for $t = 1$ to ... do
  $g_t \leftarrow \frac{1}{N} \sum_{i \in \mathscr{B}_t} \partial_{\mathbf{w}} \mathcal{L}_f(\mathbf{x}_t, \mathbf{w})$
  if $\mu \neq 0$
    if $t > 1$
      $\mathbf{b}_t \leftarrow \mu \mathbf{b}_{t-1} - (1-\tau) g_t$
    else
      $\mathbf{b}_t \leftarrow g_t$
    $g_t \leftarrow \mathbf{b}_t$
  $\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \gamma \cdot \mathbf{g}_t$
return $\mathbf{w}_t$

Below the algorithm of ADAM can be seen[1], however explaining how this algorithm came about and what the underlying intuition is behind m_t and v_t is beyond the scope of this thesis.

Algorithm 2 Adam

Input: Loss function: $\mathcal{L}_f(\mathbf{x}_t, \mathbf{w})$, learning rate γ, parameters $\mathbf{w}_0$, betas β_1, β_2
Initialize: $m_0 \leftarrow 0$, $v_0 \leftarrow 0$
Output: updated parameters $\mathbf{w_t}$

for $t = 1$ to ... do
  $g_t \leftarrow \frac{1}{N} \sum_{i \in \mathscr{B}_t} \partial_{\mathbf{w}} \mathcal{L}_f(\mathbf{x}_t, \mathbf{w})$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / (1-\beta_1)$
  $\widehat{v_t} \leftarrow v_t / (1-\beta_2)$
  $\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \gamma \widehat{m_t} / (\sqrt{\widehat{v_t}} + \epsilon)$
return $\mathbf{w}_t$

2.2.2. Multilayer perceptron

In the sections above the neural network is abstractly denoted as $f(\mathbf{x}, \mathbf{w})$ where $\mathbf{x}$ is the input vector and $\mathbf{w}$ are its weights. But this leaves away a lot of details, the architecture plays an important role in the performance of the model as well as what the input is that you give to the model. In this thesis the multilayer perceptron is used.

The multilayer perceptron can be described as consecutively applying similar functions called layers. These layers are called linear layers and activation functions. And in the implementation in this thesis dropout layers are added as well. Each layer takes in input from the previous layer and produces output for the next layer.

In the case of the multilayer perceptron at the start the input vector is fed to the first layer which is a linear layer. This is then fed to an activation layer which in turn is then fed to the next linear layer, this is shown in an example MLP 3.

The linear layer has two parameters that are set beforehand n, m which will specify the input dimension and the output dimension respectively. It takes as input a column vector v that is $1 \times n$ this will then be matrix multiplied by matrix W that is $n \times m$. A bias vector $\mathbf{b}$ of dimension $1 \times m$ will then be added to all the entries of the vector vW the matrix. The output is Linear_layer$(v) = \mathbf{v}W + \mathbf{b}$. All the entries in the bias vector are trainable. The entries of the matrix and of the bias vectors are randomly initialised.

The activation function takes as input a vector $\mathbf{v} = [v_1, v_2, \ldots, v_n]$ that is $1 \times n$, where n is an arbitrary natural

number. Then the activation function will for each entry of the vector apply a nonlinear function σ.

$$\text{activation}(v) = \begin{bmatrix} \sigma(v_1) \\ \sigma(v_2) \\ \sigma(v_3) \\ \vdots \\ \vdots \\ \sigma(v_{n-2}) \\ \sigma(v_{n-1}) \\ \sigma(v_n) \end{bmatrix}^T$$

The types of activation layers that are used in this thesis are ReLU and softplus.

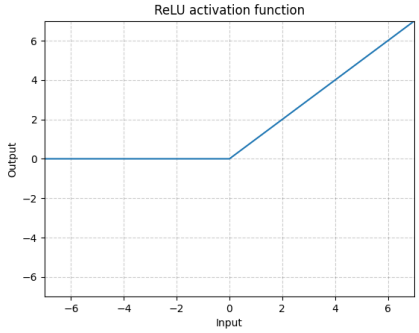


Figure 2.3: The ReLU activation function [2]

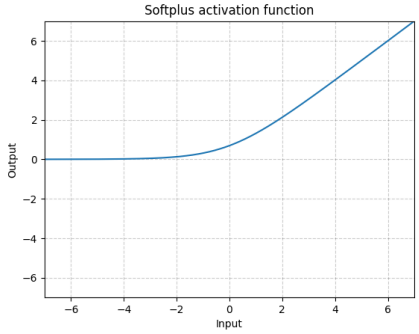


Figure 2.4: Plot of the Softplus activation function obtained using the function $\text{Softplus}(x) = \log(1 + \exp(x))$ [4]

Another layer that is used in the thesis is the dropout layer, what this layer does is it takes a vector v as input and zero's out certain entries, randomly during training time.

$$\text{dropout}(v) = \begin{bmatrix} v_1 \\ 0 \\ v_3 \\ \vdots \\ \vdots \\ v_{n-2} \\ 0 \\ 0 \end{bmatrix}^T$$

Here random entries are zero'd out. The dropout layer has parameter $p \in [0,1]$ that is set beforehand that determines represents the probability of an entry being zero'd out.

The dropout layer will only affect the training, when the neural network is used for prediction the dropout layer parameter p will become 0 (i.e. no values will be zero'd out). The reason dropout layers are added is to avoid that the neural network will just memorize the dataset. By adding dropout layers the neural network will not always be able to observe all the features, hence it needs to come up with a robust way to get the desired prediction. When the model gets less sensitive to changes in input generalising to unseen data will become better.

However adding dropout layers will work less well when the prediction really should be sensitive to changes in the input data.

An example multilayer perceptron is shown below with the name MLP:

Algorithm 3 MLP

Input: Vector $\mathbf{v}$ of size 1×14
Initialize: For each linear layer Linear_layer the corresponding matrix W of size $n \times m$ each entry has values sampled from $U[-1/\sqrt{m}, 1/\sqrt{m}]$, each entry of the corresponding bias vector $\mathbf{b}$ is sampled from $U[-1/\sqrt{m}, 1/\sqrt{m}]$ as well
Output: vector $\mathbf{y}$ of size 1×1

```
layer1 ← Linear_layer(14,22)
y ← layer1(y)      /*  y ← yW + b, W is 14 × 22, b is 1 × 22  */
activation_layer ← ReLU()
y ← activation_layer(y)
layer2 ← Linear_layer(22,22)
y ← layer2(y)
dropout_layer ← dropout(.7)
y ← dropout_layer(y)
y ← activation_layer(y)
layer3 ← Linear_layer(22,1)
y ← layer3(y)      /*  y ← yW + b, W is 22 × 1, b is 1 × 1  */
activation_layer2 ← softplus()
y ← activation_layer2(y)
return y
```

When the MLP is first initialised the matrices and vectors are randomly initialized so its performance is comparable to that of randomly guessing the output. But during training each matrix entry and each bias entry will be updated by using the optimizer such that the loss will be less on the batches $\mathscr{B}$ that it was fed during training. In practice this usually also results in the model performing well on batches that it has not seen yet, provided the data is shuffled properly. If it is not shuffled properly then batch $\mathscr{B}_i$ might have very different data than $\mathscr{B}_j$ thus performing better on batch $\mathscr{B}_i$ will then not generalize to better performance on batch $\mathscr{B}_j$.

As mentioned before during inference the dropout probabilities will become 0. This will be mathematically written out:

MLP:$\mathbb{M}_{1\times n}(\mathbb{R}) \to \mathbb{R}$ that is defined as

$$\text{MLP}(\mathbf{v}) = \text{softplus} \circ \text{Linear}_4^{(22,1)} \circ \text{Linear}_3^{(22,22)} \circ \text{ReLU} \circ \text{Linear}_2^{(22,22)} \circ \text{ReLU} \circ \text{Linear}_1^{(14,22)}(\mathbf{v})$$

Where $\text{Linear}^{(n,m)}(\mathbf{x}) = \mathbf{x}W + \mathbf{b}$ where W is $n \times m$ and b is $1 \times m$ and x is $1 \times n$

2.2.3. Transformer network

The second architecture type that was used in this thesis is the Transformer network architecture. The Encoder part of the Transformer was used. The Transformer was furthermore modified such that it can be used for regression. The Transformer network was first introduced in the paper Attention is all you need [15] and

was initially used in the natural language processing domain. But was found to work well in other domains as well. The transformer model was choosen since the data can be collected and fed to the model in a way that would make learning the relationship between the input and output easier for the model, this will be further explained in the discussion see section 7 The transformer encoder consists of the following layers:

- The Positional encoding layer
- The attention layer
- The multiheaded attention layer
- The Add & Norm layer
- The FeedForward layer

The transformer network is fed N vectors of the same size, where N is a natural number. These vectors are called embedding vectors or token embeddings. [1] These N tokens are all fed to the model at the same time. The main advantage of this is that computations on these vectors can be done in parallel. However the model will not be able to tell the difference if it is being fed $(token_1, token_2, token_3)$ or $(token_1, token_3, token_2)$ i.e. it will not be able to tell which token comes before another token.

The first new layer that is part of the transformer model is the Positional encoding layer, this will give the model a way to take the ordering of the tokens into account. It does this by adding a value to each entry e_j $0 \leq j \leq d$ of a token $token_j$, where d is the size of the token vector and $0 \leq j \leq N$. The value that is added to the entry e_i of $token_j$ depends on j (i.e. the index in a sequence of tokens) and on i i.e. the index of the entry in the embedding vector. The size of the token vector must be even.

To each entry e_i^j of token $token_j$, $p_j^i = \sin\left(\frac{j}{10000^{\frac{(i)}{d}}}\right)$ is added whenever i is even and when i is odd $p_j^i = \cos\left(\frac{j}{10000^{\frac{i-1}{d}}}\right)$ is added.

Or in other words

$$p_j^{(i)} = f(j)^{(i)} := \begin{cases} \sin(w_k j) \text{ if } i = 2k \\ \cos(w_k j) \text{ if } i = 2k+1 \end{cases}$$

Where

$$w_k = \frac{1}{10000^{2k/d}}$$

In column vector form $\mathbf{p}_j$ becomes :

$$\mathbf{p}_j = \begin{bmatrix} \sin(w_1 j) \\ \cos(w_1 j) \\ \sin(w_2 j) \\ \cos(w_2 j) \\ \vdots \\ \sin(w_{d/2} j) \\ \cos(w_{d/2} j) \end{bmatrix}^T$$

Where d is an even number.

Hence the positional encoding layer becomes [8]

$$\text{positional\_encoding}((\text{token}_1, \text{token}_2, \ldots, \text{token}_N)) = (\text{token}_1 + \mathbf{p}_1, \ldots, \text{token}_N + \mathbf{p}_N)$$

[1] In this thesis the term token is used instead of token embedding, these terms strictly speaking mean different things in natural language processing but for the sake of abbreviation the term token is used instead of token embedding.

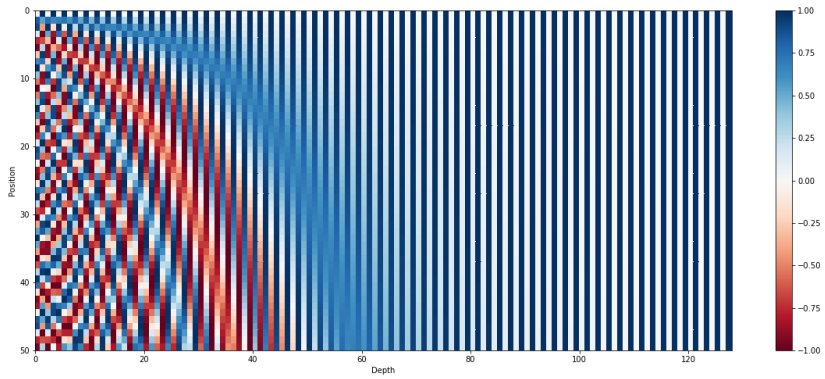


Figure 2.5: Positional encoding for 128 entry token embeddings and maximum number of tokens is 50 [8]

The second new layer that is part of the Transformer architecture is the attention layer. The attention layer takes as its input queries $\mathbf{q}_i \in \mathbb{R}^n$ and keys $\mathbf{k}_i \in \mathbb{R}^n$ and values $\mathbf{v}_i \in \mathbb{R}^n$ for $i = 1, \ldots, N$ for some natural number N. It essentially computes the similarity between the queries $\mathbf{q}_i$ and keys $\mathbf{k}_i$. The similarity similarity($\mathbf{k}_i, \mathbf{q}_i$) is a real number, and it represents how related the query $\mathbf{q}_i$ and the key $\mathbf{k}_i$ are. The value $\mathbf{v}_i$ that corresponds to the key $\mathbf{k}_i$ will then be multiplied by a modified version of this similarity. How this will look concretely will be shown below.

The similarity is calculated as follows:

$$\text{similarity}(\mathbf{q}, \mathbf{k}) = \mathbf{q} \cdot \mathbf{k} / \sqrt{n}$$

Where n is the dimension of the vectors $\mathbf{q}, \mathbf{k}$. The extra factor $\frac{1}{\sqrt{n}}$ makes sure the similarity will not take increasingly extreme values as the dimension of the query and key vectors become larger. Doing this for each individual vector is equivalent to making the queries rows in a matrix Q and the keys columns in a matrix K. Then the similarity can be calculated for all the queries and keys at the same time.

$$\text{similarity}(Q, K) = QK^T / \sqrt{n}$$

Now, let V be the matrix of values, that consists of the values $\mathbf{v}_i \in \mathbb{R}^n$ as rows in the matrix. As mentioned above the similarity can take on negative numbers. What the similarity represents is how related two embedding vectors are, so it would be preferable if the similarity would only take on values between 0 and 1, where 0 meant unrelated and 1 meant the tokens are very related. What Vaswani et al. [15] came up with is applying the softmax function to each column of the similarity matrix. The softmax function does the following per embedding vector:

$$\text{softmax}(\mathbf{x}) = \begin{bmatrix} \frac{x_1}{\sum_{j=0}^{n} \exp(x_j)} \\ \frac{x_2}{\sum_{j=0}^{n} \exp(x_j)} \\ \frac{x_3}{\sum_{j=0}^{n} \exp(x_j)} \\ \frac{x_4}{\sum_{j=0}^{n} \exp(x_j)} \\ \vdots \\ \frac{x_i}{\sum_{j=0}^{n} \exp(x_j)} \\ \vdots \\ \frac{x_n}{\sum_{j=0}^{n} \exp(x_j)} \end{bmatrix}$$

per entry of the columns Then the attention layer does the following:

$$\text{attention}(Q,K,V) = \textbf{softmax}(\frac{QK^T}{\sqrt{n}})V$$

Where the **softmax** applies the softmax function on each column of the matrix. The attention layer will output a $N \times n$ matrix. Where N is the number of tokens and n is the number of entries in the key query and value vectors.

Now, the third new layer of the Transformer architecture builds on the attention layer described before. This layer is called the multiheaded attention layer. It takes in N embedding vectors $\mathbf{v} \in \mathbb{R}^n$ and projects these onto subspaces by means of matrix multiplication. These vectors then correspond to queries keys and values. The matrices are usually named $W_{Q_i}, W_{K_i}, W_{V_i}$, with $i = 1,\ldots,m$ where $m \in \mathbb{N}$ is called the number of heads, hence the name multiheaded attention.

The embedding vectors are put in matrix form M where the embedding vectors form the columns of this matrix M. Then $Q_i = W_{Q_i}M, V_i = W_{V_i}M, K_i = W_{K_i}M$

then the attention layer is applied for each i:

$$\text{attention}_i = \text{attention}(Q_i, K_i, V_i)$$

For each i, attention_i is a $N \times n$ matrix as mentioned before. Now these matrices are concatinated horizontally producing a $N \times n \cdot m$ sized matrix. This is then put in a linear layer (see section 2.2.2) to shrink down the size of the matrix to $N \times n$. This will be the output of the multiheaded attention layer.

In other words multiheaded_attention : $\mathbb{M}_{N\times n}(\mathbb{R}) \to \mathbb{M}_{N\times n}(\mathbb{R})$ is defined as

$$\text{multiheaded\_attention}(M) = \begin{bmatrix} \text{attention}(W_{Q_1}M, W_{K_1}M, W_{V_1}M) \\ \text{attention}(W_{Q_2}M, W_{K_2}M, W_{V_2}M) \\ \text{attention}(W_{Q_3}M, W_{K_3}M, W_{V_3}M) \\ \text{attention}(W_{Q_4}M, W_{K_4}M, W_{V_4}M) \\ \vdots \\ \text{attention}(W_{Q_i}M, W_{K_i}M, W_{V_i}M) \\ \vdots \\ \text{attention}(W_{Q_m}M, W_{K_m}M, W_{V_m}M) \end{bmatrix}^T W + \mathbf{b}$$

Where W is an $(n \cdot m) \times n$ matrix and $\mathbf{b}$ is $1 \times n$.

The third layer that will be covered is the Add & Norm layer. This layer works by adding two inputs together and normalizing the result. It is also sometimes referred to as a residual connection or skip connection. The Add & Norm layer is added to avoid the vanishing gradient problem and to preserve knowledge after going through a layer. Inputs can get modified a lot when going through a layer, the layers afterwards will not have access to the original data anymore, which might have some usefull information that is lost, hence why the transformer model has skip connections.

The add and norm layer does the following:

$$\text{add\_and\_norm}(\mathbf{x}, \mathscr{F}(\mathbf{x})) = \text{LayerNorm}(\mathbf{x} + \mathscr{F}(\mathbf{x}))$$

Where the LayerNorm function is a layer defined as [5]:

$$\text{LayerNorm}(\mathbf{x}) = \frac{\mathbf{x} - \bar{\mathbf{x}}}{\sqrt{\overline{(\mathbf{x} - \bar{\mathbf{x}})^2} + \varepsilon}}$$

where epsilon is a (hyper)parameter that is set beforehand and will make sure no division by zero occurs. And $\mathscr{F}$ is a layer over which the residual connection takes place. The residual connection adds two inputs, one from before a layer was applied and another from after the layer is applied. Suppose the skip connection did

not have this normalization layer then adding two inputs, can make the output larger, having many of these skip connections could make the output blow up making the loss from gradient descent very unstable, and thus make learning very difficult. Hence why after addition the layerNorm is applied.

The last new layer that is part of the transformer layer is the feedforward layer. The feedforward layer consists of two linear layers with in between an activation function, the feedforward layer is often abreviated to FFN:

$$\text{FFN}(\mathbf{v}) = \sigma(\mathbf{v}W_1 + \mathbf{b}_1)W_2 + \mathbf{b}_2$$

Where W_2 is $k \times n$ and W_1 is $n \times k$. And n is the length of the embedding vectors. k is usually much larger than n.

Lastly the Transformer encoder will be described using all the layers mentioned above and dropout layers.

Algorithm 4 TransformerEncoder

Input: Vectors $\mathbf{v}_i$, each of size 1×10 with $i = 1, \ldots, 7$ in matrix form M
Initialize: All the parameters from the layers in the transformer network are randomly initialized
Output: vectors $\mathbf{y}_i$ of size 10×1 with $i = 1, \ldots, 7$ in matrix form O

```
mh_layer ← multiheaded_attention(8)      /* the layer has 8 attention heads */
mh_output ← mh_layer(M)
add&norm_layer ← add_and_norm(.001)      /* ε = 0.001 , ε is a hyperparameter of LayerNorm */
W ← add&norm_layer(W, mh_output)
FFN_layer ← FFN(150)      /* the size of the vectors is increased to 150 then reduced again to 10 */
FFN_output ← FFN(W)
dropout_layer ← dropout(.7)
FFN_output ← dropout_layer(FFN_output)
O ← add&norm_layer(W, FFN_output)
return O
```

When condensing the Transformer to just a few mathematical expression the following will be obtained:

Let

$$\mathscr{F}(M) = \text{add\&norm\_layer}^{0.001}(\text{multiheaded\_attention}^{8}(M), M)$$

Then

$$\text{transformer}(M) = \text{add\&norm\_layer}^{0.001}(\mathscr{F}(M), \text{dropout\_layer}^{0.7}(\text{FFN}^{150}(\mathscr{F}(M))))$$

Where for each layer $\mathscr{F}$, $\mathscr{F}^x(M)$ means layer $\mathscr{F}$ is applied to M with value x assigned to the hyper-parameter can be set for that layer.

To the right a schematic of the Transformer neural network can be seen.[15] The left part of the image is the transformer encoder. Now, the Transformer in this thesis will only consist of N TransformerEncoder layers, with $N \in \{3, 5, 10, 22, 32, 64, 128\}$, and after that a few linear layers to reduce the output vector sizes to 1. for more detailed information on the Transformer model see Appendix D. N will be a hyper-parameter that will be searched over, more on this in section 2.4.

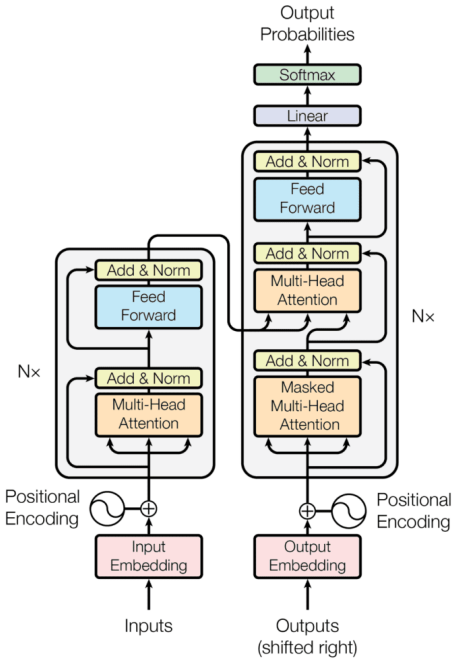


Figure 2.6: Transformer network taken from Vaswani et al. [15]

2.3. Adaptive mesh refinement

In this thesis the Adaptive mesh refinement strategy that is specified by Verfürth [16] is being modified. The general algorithm looks like this.

Given: the data of a partial differential equation and a tolerance ε.
Output: a numerical solution with an error less than ε.

1. Construct an initial coarse mesh $\mathcal{T}_0$. Set $k = 0$.
2. Use the Finite Element Method to solve the PDE with mesh $\mathcal{T}_k$
3. For every element K in T_k compute the a posteriori error indicator
4. If the estimated global error is less than ε stop, otherwise decide which elements have to be refined and construct the next mesh T_k. Add 1 to k and go to step 2

Step 1 will be done by creating a uniform mesh. Since the problems partial differential equations that are solved in this thesis are 1d this will be sufficient. Step 2 will be done using a finite element method implementation written in python. For step 3 the book by Verfürth [16] will give several different ways of creating error indicators, however in this thesis this is changed to estimating the error using local information from the element and neighbouring elements. Lastly, the specific mesh refinement strategy that used in step 4 is called *dörfler strategy* [16].

Given: a mesh $\mathcal{T}$, error indicators η_K for elements $K \in \mathcal{T}$, and threshold $\theta \in (0, 1)$
Output: a subset $\widetilde{\mathcal{T}}$ of marked elements that needs to be refined. [16]

1. Compute $\eta_{\mathcal{T},\max} = \max_{K\in\mathcal{T}} \eta_K$
2. If $\eta_K \geq \theta\eta_{\mathcal{T},\max}$ mark K for refinement and put it into set $\widetilde{\mathcal{T}}$

After the elements are marked for refinement, the refinement takes place by using certain refinement rules. For higher dimensional problems implementing these rules can become complicated. In the one dimensional case however the elements are simply lines, so the refinement rule can just be subdividing the line through the middle. An example of refinement can be seen in 2.3 where the finite element method solution of the 1d Poisson equation is shown twice, once before mesh refinement and one after.

2.3.1. Error metric

The error metric that is used, is the standard $W^{1,p}$ semi-norm, but with the integrand squared. Where $W^{k,p}(\omega)$ is the standard *Sobolev space*, with $k \geq 1$ and $1 \leq p \leq \infty$.[16]. In higher dimensions the standard $W^{1,p}(\omega)$ looks like this:

$$
\begin{aligned}
|||\phi|||_\omega &= \left(||\nabla\phi||_p^p\right)^{\frac{1}{p}} \\
&= ||\nabla\phi||_p
\end{aligned}
$$

Where

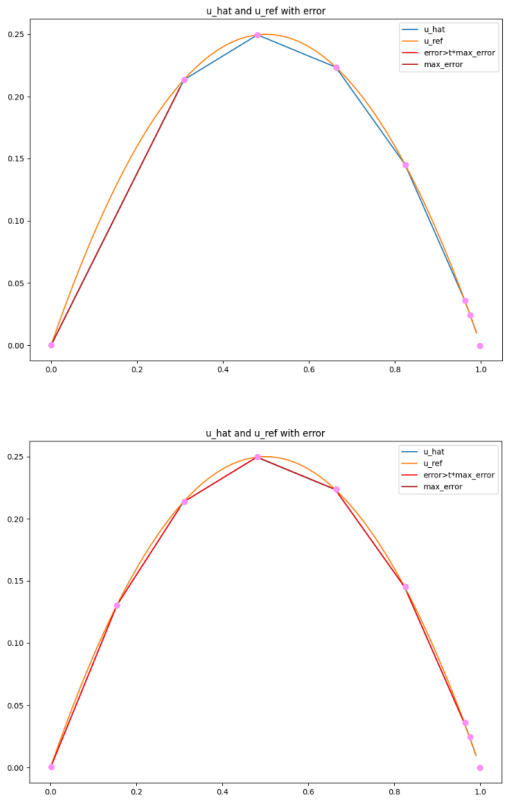


Figure 2.7: Solution before and after mesh refinement 1d

$$||\phi||_p = \int_\omega |\phi|^p \mathrm{d}\mu_d$$

Here μ_d is the Lebesque measure in $\mathbb{R}^d$ In 1d this reduces to

$$|||\phi|||_\omega = \int_\omega |\phi'| \mathrm{d}\mu_1$$

Thus since the integrand in the error metric is squared the following is used:

$$|||\phi|||_\omega = \int_\omega |\phi'|^2 \mathrm{d}\mu_1$$

2.4. Hyper optimisation

In the sections above the term hyper-parameter has been mentioned multiple times already. A hyperparameter is a parameter that is set before the training occurs and will not be modified by the training procedure. For instance the number of layers can be a hyperparameter or the learning rate.

In this thesis two neural network architectures are tried: The Transformer neural network and the Multilayer perceptron. To make the comparison between the two fair, hyper-parameter search is done. Before doing hyper-parameter optimisation the search space must first be specified. When doing hyper-parameter optimisation multiple instances of a machine learning model is started with different hyper-parameters. Each instance has hyper-parameters that were sampled from the search space. Not all possible hyper-parameters have to be equally likely to be selected. A search algorithm can be used to pick hyper-parameters that are likely to perform well according to the algorithm, this can speed up the process of finding good hyper-parameters. The hyper-parameter search algorithm that is used in this thesis is Hyperopt.

Now, another trick that can be used to speed up finding good hyper-parameters, is to cut-off training of bad performing instances of the model a way of doing this is by using the algorithm called ASHA.

In this thesis 23 different hyper-parameters are optimised, to enumerate them all here will not be particularly enlightening, but these will be mentioned in the results in section 6.

2.4.1. Hyperopt

Hyperopt uses the Tree structured Parzan estimator (TPE) algorithm.

TPE will apply the following steps [6]:

1. Sample randomly from the search space and start instances with these samples as hyper-parameters
2. Sort data by score and divide them into two groups usually the top $10-25\%$ best performing samples are put into one group x_1 and the rest is put into the second group x_2.
3. Two densities $l(x_1), g(x_2)$ are modelled using Parzan estimators (See Figure 2.9 on how a 1 dimensional parzan estimator is made)
4. Draw sample hyper-parameters from $l(x_1)$ evaluating in terms of $EI(x) = \frac{l(x)}{g(x)}$.Where EI is called the expected improvement.Then the sample x that has the maximal value of $\frac{l(x)}{g(x)}$ will get an machine learning model started with hyper-parameters x. In Figure 2.9 the sampling is graphically illustrated.
5. Update the observation list in step 1.
6. Repeat step 2 - 5.

A Parzan estimator of a group S is built by constructing a (multivariate) Gaussian to each point $x \in S$.

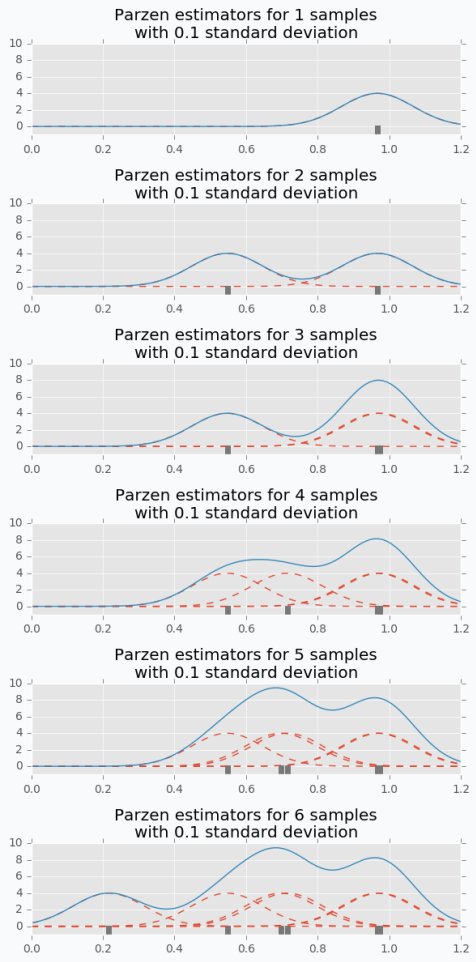


Figure 2.8: 1d parzan estimators taken from Shevchuk [14]

These are then summed up and the resulting distribution is normalised. In Figure 2.8 this is illustrated for the 1d case. In Figure 2.8 one by one samples are added to the set S and the Parzan estimate updated by summing these gaussians and normalising the result.

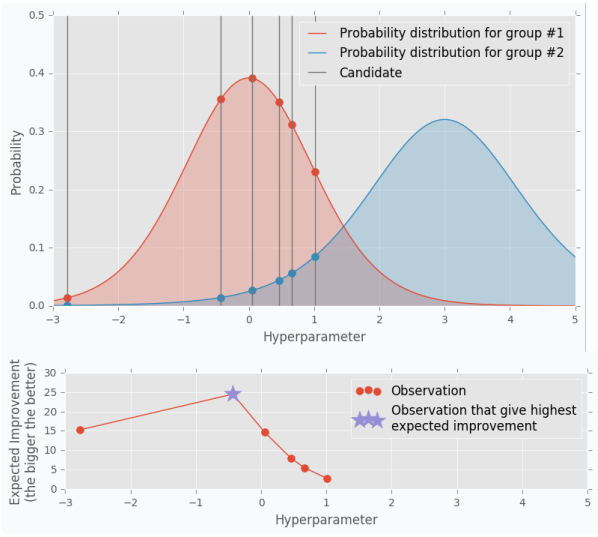


Figure 2.9: sampling from $l(x_1)$ image taken from Shevchuk [14]

2.4.2. ASHA

ASHA is an algorithm based on SHA:Successive halving algorithm. The ASHA and SHA algorithm uses the concept called rungs. The lowest rung has the least amount of resources, the highest has the highest amount of resources. SHA applies the following steps [11]

1. Uniformly allocate budget to a set of parameters (possibly selected by a search algorithm)

2. Evaluate performance of all instances

3. Promote top half of instances to the next rung

4. Double the amount of resources per instance
   in the next rung.

This algorithm however has to wait until all configurations are done in order to determine which instances will be promoted to the next rung. ASHA modifies this to work asynchronously. ASHA does the following steps:

1. Starts instances and add to the bottom rung

2. When instance is done ASHA looks at rungs from top to bottom to see if there are instances that are in the top half of each rung that can be promoted to the next rung. If no instance can be promoted ASHA will add instances to the bottom rung, so that more instances can get promoted.

3

Previous work

This thesis is not the first work that attempts to speed up or improve mesh refinement using neural networks. In this chapter an overview of relevant previous articles will be presented.

3.1. Deep Learning Driven Self-adaptive Hp Finite Element Method

Here the authors Paszyński et al. [13] replaced the algorithm that selects optimal refinement in an Adaptive Finite Element strategy called self-adaptive hp-FEM by a neural network. Instead of approximating the error (as done in this thesis) the neural network outputs whether to do h-refinement or p-refinement.

3.2. Locally refined quad meshing for linear elasticity problems based on convolutional neural networks

In this work the authors Chan et al. [7] feed the domain as input to the model as an image. The neural network is based on the U-net architecture. The output of the neural network is another image which encodes the amount of detail of the mesh locally. Two differences between our work an the one from Chan et al. [7] are notable. In this thesis only local information is used: the data of the elements and of adjacent elements are fed to the model, whereas in the work of Chan et al. [7] the whole domain was used as an image. The second important difference is that in this thesis the goal is to estimate the residual error in an element and do refinement based on this estimate, whereas in Chan et al. [7] the goal is to output a mesh immediately.

3.3. MeshingNet: A New Mesh Generation Method Based on Deep Learning

In this work the authors Zhang et al. [18]develop a way to do mesh generation using a neural network. The neural network in this work is trained on coordinates of the vertices of a polygon P in the domain;parameters of the PDE, and the (mean value) coordinates of a point within the polygon x. The neural network then outputs the target local element area upper bound $A(x)$, which is a number that can be fed to software called *Triangle* that can then subdivide the corresponding polygon P if the area of this polygon is too large.

The trained model will then be fed a PDE and the i^{th} element e_i of a mesh for this PDE and centre x of e_i and will be able to output $A(x)_i$. Local information is used to do this prediction, this is similar to what is done in this thesis. However in Zhang et al. [18] the authors refine without using a residual error estimate prediction in this thesis the refining is done using the residual error.

3.4. Adaptive mesh refinement using residual error estimates

In the book of Verfürth [16] adaptive mesh refinement is also done using residual error estimates. The approach in this thesis is strongly related to this work. In the work of Verfürth [16] residual error estimates are estimated using upper bounds, these are then used to do adaptive mesh refinement as shown in 2.3. In this

thesis a neural network is used instead of residual bounds.

4

Our contribution

Recall the psuedocode from the Background section 2.3
Given: the data of a partial differential equation and a tolerance ε.
Output: a numerical solution with an error less than ε.

1. Construct an initial coarse mesh $\mathcal{T}_0$. Set $k = 0$.
2. Use the Finite Element Method to solve the PDE with mesh $\mathcal{T}_k$
3. For every element K in T_k compute the a posteriori error indicator
4. If the estimated global error is less than ε stop, otherwise decide which elements have to be refined and construct the next mesh T_k. Add 1 to k and go to step 2

Step 3 in red is modified in this thesis, instead of constructing or using an a posteriori indicator a residual error estimate is made using a neural network.

So the pseudocode becomes
Given: the data of a partial differential equation and a tolerance ε.
Output: a numerical solution with an error less than ε.

1. Construct an initial coarse mesh $\mathcal{T}_0$. Set $k = 0$.
2. Use the Finite Element Method to solve the PDE with mesh $\mathcal{T}_k$
3. For every element K in T_k compute a residual error estimate using a neural network
4. If the estimated global error is less than ε stop, otherwise decide which elements have to be refined and construct the next mesh T_k. Add 1 to k and go to step 2

Now to use a neural network for residual error estimation the neural network has to be trained. As input it will receive data from the PDE local to an element, in this thesis this will be the right hand side f of the PDE evaluated on points x, $f(x)$ within the element e, along with the corresponding coordinates x. Furthermore coordinates of the element and adjacent elements are given to the network as well.

Now to obtain data for the neural network a pipeline has to be built. A very general description of a General Pipeline will be given below:

Algorithm 5 General Pipeline

Output: Dataset containing local information per element

```
PDE_data ← Generate_pde_data()      /* In this thesis random polynomial right hand sides f are generated */
Mesh ← Generate_initial_mesh()      /* In the examples meshes are generated randomly */
FEM_solution ← solve_FEM(Mesh, PDE_data)
data ← EmptyDataset()
for each element e_i in FEM_solution
    /* In the (first) example evaluating rhs gets evaluated uniformly on mesh */
    /* and returned along with the corresponding positions */
    local_inf ← generate_local_information(PDE_data, Mesh)
    E ← get_error(FEM_solution, PDE_data)
    data ← add_to_dataset(data, local_inf, E)
return data
```

5

Neural error estimator

In the previous chapter the general idea of this thesis is explained here this will be applied on a simple 1d problem. The residual error will be estimated for the following ODE:

$$-u''(x) = f(x) \text{ for x} \in (0,1)$$
$$u(0) = u(1) = 0$$

For the finite element method solver linear basis functions were used.

5.1. Pipeline

The pipeline below can be described in the following way:

- First random numbers are generated Num_1 and Num_2. These are then used to to generate points and generate a mesh respectively.

  $$(\text{Num}_1, \text{Num}_2) \mapsto (\text{generate\_points}(\text{Num}_1), \text{generate\_mesh}(\text{Num}_2)) = (\text{Points}, \text{Mesh})$$

  Where Points = $[(x_1, y_1), \ldots, (x_{\text{Num}_1}, y_{\text{Num}_1})]$, Mesh = $[x_1, \ldots, x_{\text{Num}_2}]$

- After that, the following mapping takes place Points and Mesh to $h : [0,1] \to [0,1]$ and Mesh respectively.

  $$(\text{Points}, \text{Mesh}) \mapsto (h, \text{Mesh})$$

- h and Mesh are then mapped in the following way:

  $$(h, \text{Mesh}) \mapsto (h, -h'', \text{Mesh}) = (h, f, \text{Mesh})$$

- h and f are then renormalised such that f only takes values between 0 and 10. h is then changed accordingly as well. Thus

  $$(h, f, \text{Mesh}) \mapsto (\text{renormalize}(h, f), \text{Mesh}) = (h, f, \text{Mesh})$$

- h, f and Mesh maps to:

  $$(h, f, \text{Mesh}) \mapsto (h, f, \text{Mesh}, \text{FEM.solve}(\text{Mesh}, f)) = (h, f, \text{Mesh}, \hat{u})$$

- h, f, Mesh and $\hat{u}$ then maps to:

$$(h, f, \text{Mesh}, \hat{u}) \mapsto (\text{error}(h, \hat{u}, \text{Mesh}), f, \text{Mesh}, \hat{u}) = (E, f, \text{Mesh}, \hat{u})$$

- E, f, Mesh and $\hat{u}$ maps to:

$$\begin{aligned}(E, f, \text{Mesh}, \hat{u}) \mapsto (E, [f, \dots, f], \text{make\_tuple}(\hat{u}), \text{make\_tuple}(\text{Mesh})) &= (E, \mathbf{f}, [(\hat{u}_1, \hat{u}_2), \dots, \\ &(\hat{u}_{\text{Num}_2-1}, \hat{u}_{\text{Num}_2})], [(x_1, x_2), \dots, (x_{\text{Num}_2-1}, x_{\text{Num}_2})]) \\ &= (E, \mathbf{f}, u_t, x_t)\end{aligned}$$

- $E = [e_1, \dots, e_{\text{Num}_2}]$, u_t, x_t and f is then flatmapped to:

$$(E, \mathbf{f}, u_t, x_t) \mapsto (e_i, f, u_0^{i-1}, x_0^{i-1}, u_t^i, x_t^i, u_1^{i+1}, x_1^{i+1})$$

  In the case $i = 0$ or i is st x_i is on the boundary we just say the adjacent node is outside the domain with u there being 0.

- e_i, f, $u_0^{i-1}, x_0^{i-1}, u_t^i, x_t^i, u_1^{i+1}, x_1^{i+1}$ is then mapped to:

$$(e_i, f, u_0^{i-1}, x_0^{i-1}, u_t^i, x_t^i, u_1^{i+1}, x_1^{i+1}) \mapsto (e_i, f, u_t^i, x_t^i, [x_1^i, \dots, x_N^i], u_0^{i-1}, x_0^{i-1}, u_1^{i+1}, x_1^{i+1})$$

  Where $[x_1^i, \dots, x_N^i]$ are nodes in element i, N is in this case 10.

- $e_i, f, u_t^i, x_t^i, [x_1^i, \dots, x_N^i], u_0^{i-1}, x_0^{i-1}, u_1^{i+1}$ and x_1^{i+1} are then mapped to:

$$(e_i, f, u_t^i, x_t^i, [x_1^i, \dots, x_N^i], u_0^{i-1}, x_0^{i-1}, u_1^{i+1}, x_1^{i+1}) \mapsto (e_i, u_t^i, x_t^i, [f(x_1^i), \dots, f(x_N^i)], u_0^{i-1}, x_0^{i-1}, u_1^{i+1}, x_1^{i+1})$$

- $e_i, f, u_t^i, x_t^i, [f(x_1^i), \dots, f(x_N^i)], u_0^{i-1}, x_0^{i-1}, u_1^{i+1}$ and x_1^{i+1} are then mapped to:

$$\begin{aligned}(e_i, f, u_t^i, x_t^i, [f(x_1^i), \dots, f(x_N^i)], u_0^{i-1}, x_0^{i-1}, u_1^{i+1}, x_1^{i+1}) \mapsto &\text{get\_vectors}((e_i, u_t^i, x_t^i, [f(x_1^i), \dots, f(x_N^i)], u_0^{i-1}, x_0^{i-1}, u_1^{i+1}, x_1^{i+1})) \\ &= (e_i, \mathbf{u}, \mathbf{u}_{prev}, \mathbf{u}_{next}, \mathbf{x}_{prev}, \mathbf{x}, \mathbf{x}_{next})\end{aligned}$$

The resulting list with entries of the form $(e_i, \mathbf{u}, \mathbf{u}_{prev}, \mathbf{u}_{next}, \mathbf{x}_{prev}, \mathbf{x}, \mathbf{x}_{next})$ is then converted into a dictionary. The dictionaries are then merged and converted into a pandas dataframe.

6

Results

6.1. Hyper parameter optimisation

For the 1d Poisson with linear basis problem, Hyper parameter optimisation was used to find neural network models that perform well. However, the Hyper parameter optimisation loop had hardware related difficulties. The computers that were running the program frequently ran out of memory resulting in model instances being terminated prematurely. This might be the reason why the first Transformer model that was built performs better than the top 10 best performing models from the hyper parameter search. Now, another reason

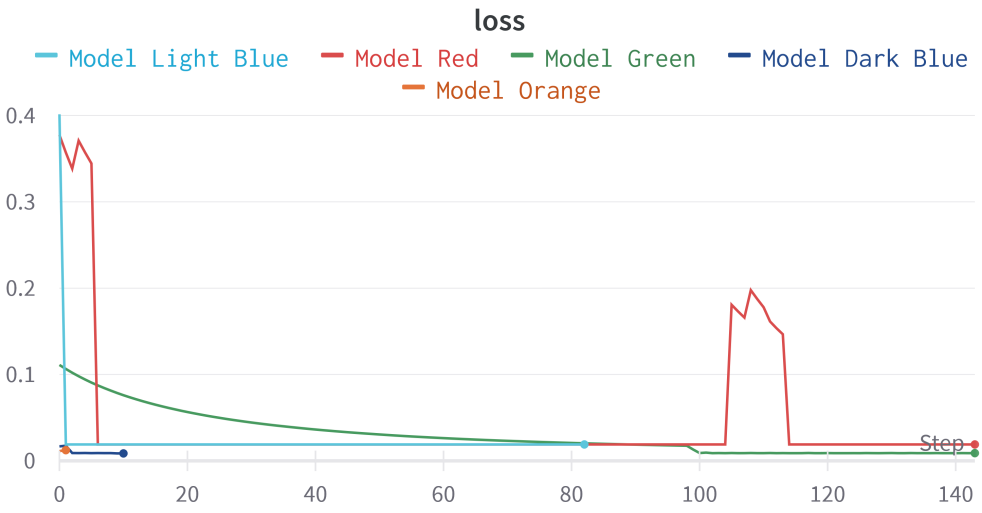


Figure 6.1: Results of hyper optimisation

why the hyper optimisation tuning usually ended prematurely is because some models had a learning rate that was larger than one, resulting in models that can learn really fast. But these models are usually finished with learning much quicker and cannot learn as well as the models that started with a smaller learning rate. The models with a smaller learning rate then get disqualified by the ASHA algorithm (see section 2.4.2) leading to the models with higher learning rate obtaining an unfair advantage.

For this reason the models with a learning rate that is higher than 1 are filtered in subsequent analyses. The 5 best performing models with a learning rate smaller than 1 are retrained and their models are saved. Moreover, Adaptive mesh refinement using the actual error and using the estimated error will be compared, where the estimated error will be calculated by the first transformer model, this model is retrained as well.

Model Name	final MSE loss	num_layers	layerwidth	dropout1	dropout2	dropout3	batch_size	optimizer	momentum	dampening	lr	NUM_EPOCHS
Model Orange	8.9787E-03	5	128	0.624878144	0.39031984	0.771958722	256	ADAM	-	-	0.05129	1900000
Model Green	8.8629E-03	5	32	0.8256	0.738	0.0445	128	SGD	0.3128	0.07994	0.005638	1900000
Model light Blue	1.8990E-02	64	32	0.59	0.64	0.1262	128	ADAM	-	-	9.66E-03	1900000

Table 6.1: Results of retrained MLP models

Model Name	final MSE loss	type	num_layers	batch_size	optimizer	momentum	dampening	lr	NUM_EPOCHS	d_model	dropout	nheads	norm_first	layer_norm_eps	dim_FF	pos_enc
Model Red	8.8999E-03	transformer	22	256	SGD	0.3074	0.058347	0.001426	1900000	22	0.449	4	FALSE	0.0000692912	50	FALSE
Model Purple	4.5895E-03	transformer_first*	6	128	ADAM	-	-	1.00E-03	1900000	10	0	3	FALSE	1.00E-05	10	FALSE
Model Grey	8.8057E-03	transformer	6	128	ADAM	-	-	1.00E-03	1900000	10	0	3	FALSE	1.00E-05	10	FALSE
Model Dark Blue	8.8998E-03	transformer	64	256	ADAM	-	-	0.00292472	1900000	16	0.08	16	TRUE	1.0000019	10	FALSE

Table 6.2: Results of retrained Transformer models

Lastly the best model is evaluated on a single difficult example just to give an indication of its performance.

Below the loss over training iterations is shown

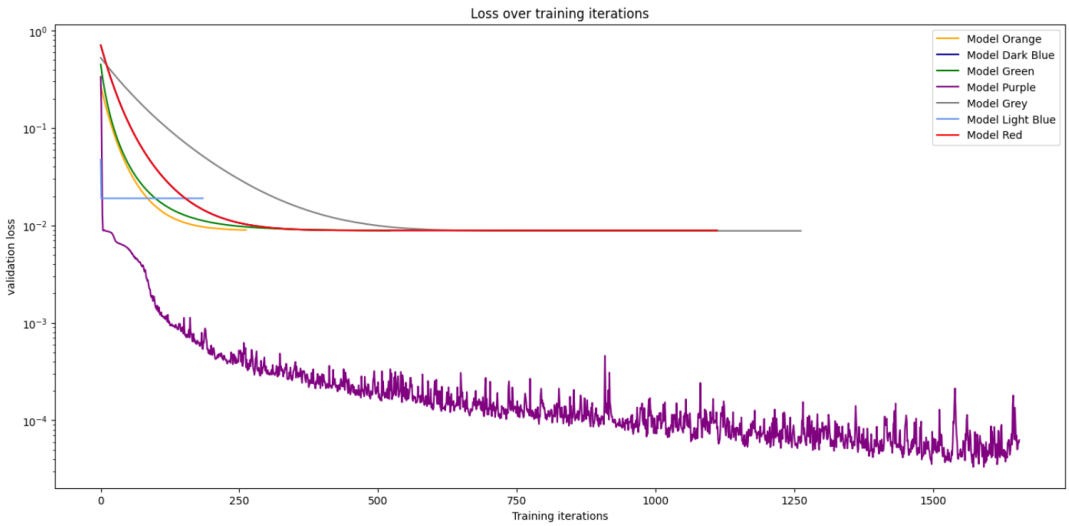


Figure 6.2: training loss over training iterations

As can be seen, the changes in performance of the neural networks from the hyper parameter tuning are negligible. However, the first transformer model performs significantly better than the other models, while its architecture deviates only slightly, details on the architecture can be found in D. The similar performance of the models produced by tuning can be explained by the fact that all these models only output a constant value, as can be seen in the figures below. In the figures 6.3, 6.9, 6.4,6.7,6.8 and 6.5 the left plot gives the exact solution and the blue and red lines give the finite element solution. The lines of the Finite element solution are red when the error in that element is at least 70% as large as the maximum error. If the line segment of the finite element method is dark red then on the corresponding element it takes on the maximum error.

It seems that the models cannot learn the right relationship between the input and output and thus output a constant value independent of the input.

In figure 6.5 the model outputs 0 error this model also finished training the fastest. The model had the most amount of layers of all the mlp models, furthermore its training loss was still going down (the training losses are not displayed in the figure). Thus this is a clear sign of over-fitting from the model. Model purple in figure 6.7 is the only model that comes close to actually approximating the error.

When experimenting with the number of Dense layers (linear layer with afterwards an activation function) of the transformer model another functioning model was obtained in figure 6.10 the loss can be seen of this new model the model is called Transformer test. The training crashed hence why there are two lines that belong to the transformer test model. The transformer test model only has one linear layer and afterwards a softplus activation function. The linear layer takes the 70 inputs and maps it to just one output. This model was the only model that together with the first transformer neural network(model purple) gave reasonable

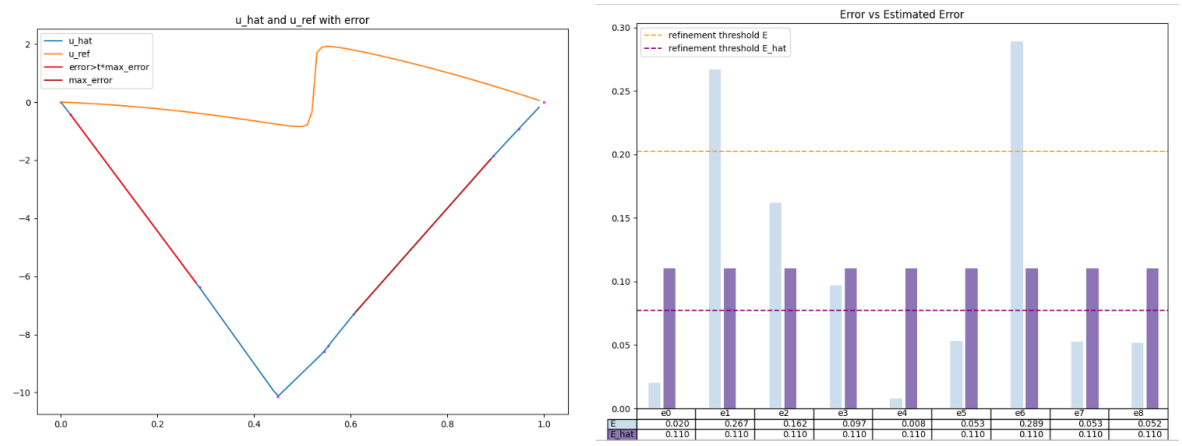


Figure 6.3: plot and histogram of model orange

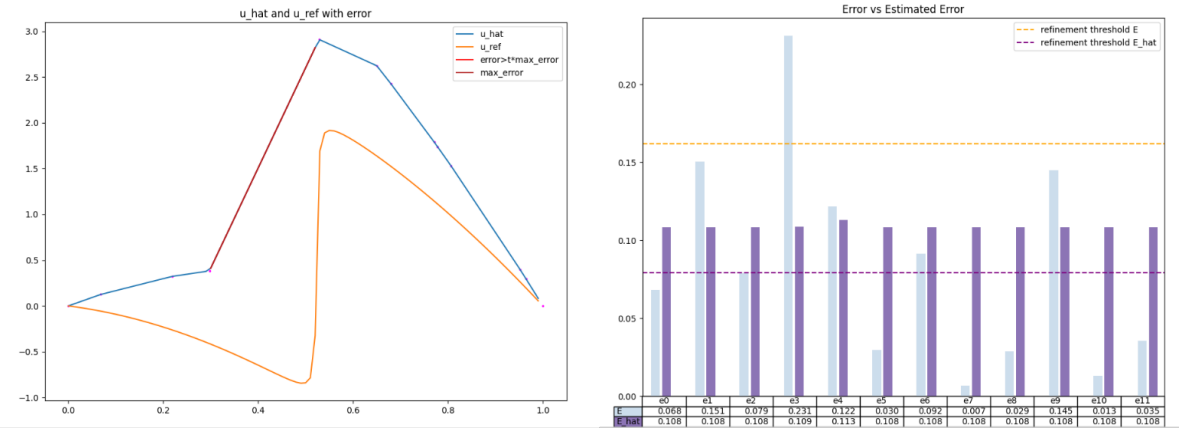


Figure 6.4: plot and histogram of Model green

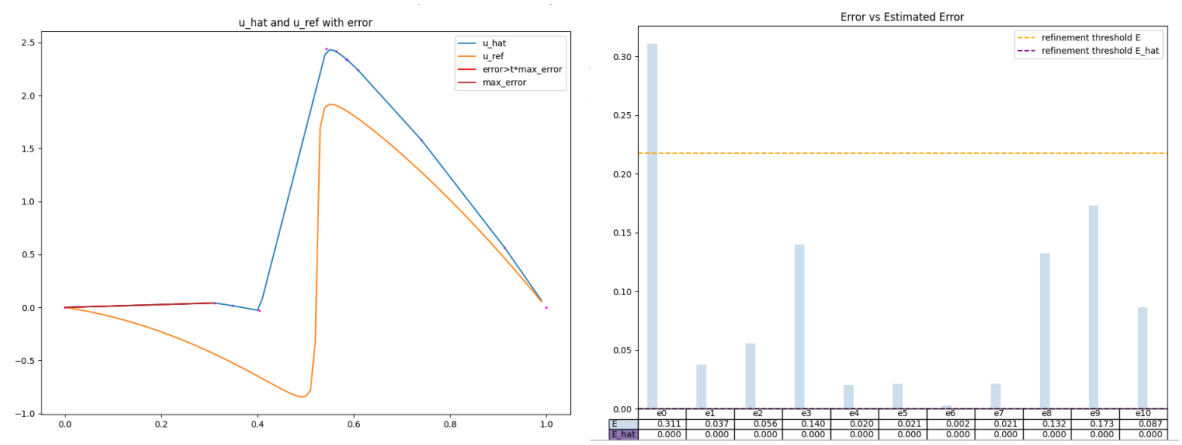


Figure 6.5: plot and histogram of Model light blue

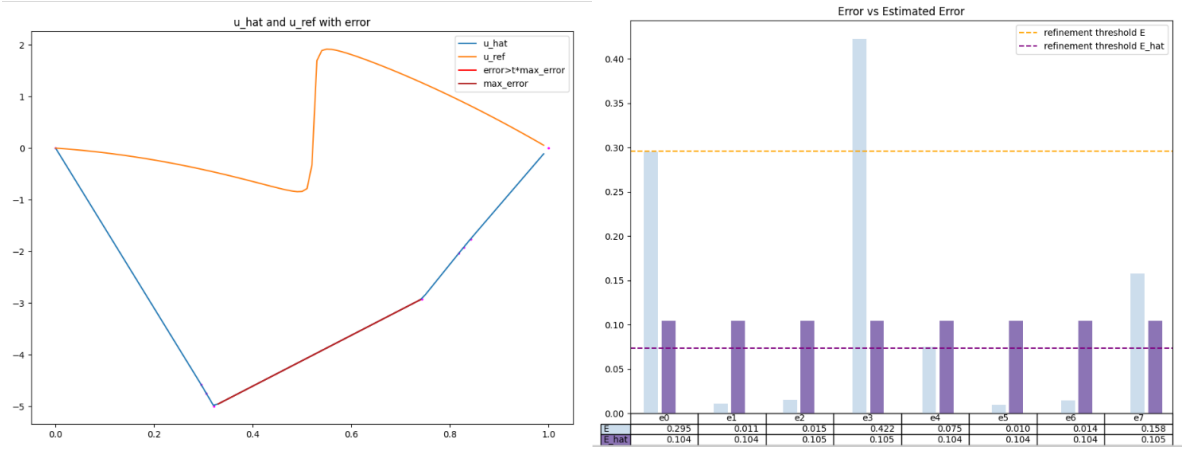


Figure 6.6: plot and histogram of Model red

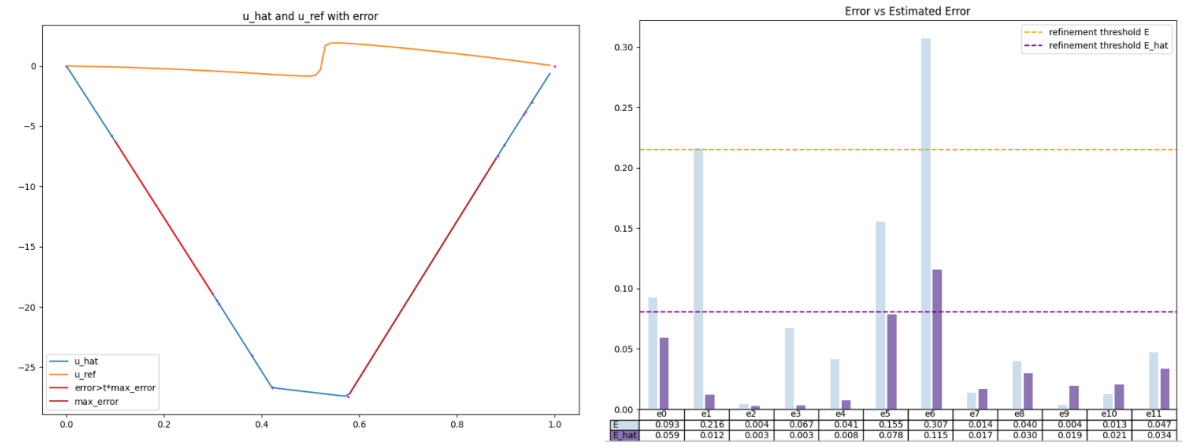


Figure 6.7: plot and histogram of Model purple

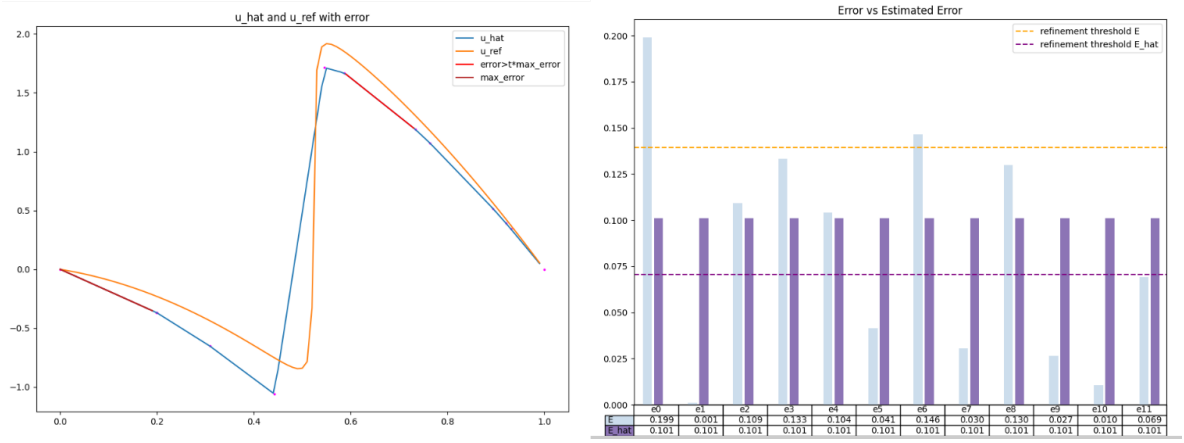


Figure 6.8: plot and histogram of Model grey

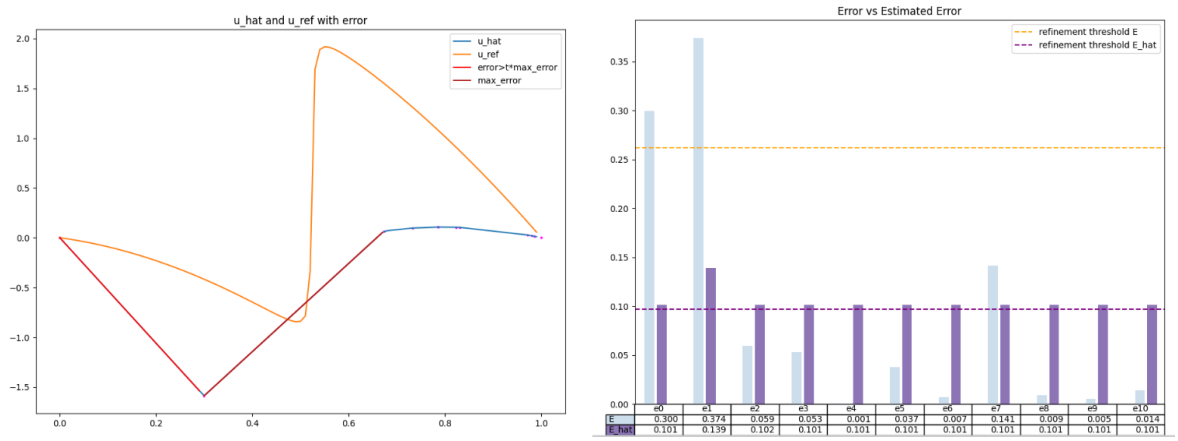


Figure 6.9: plot and histogram of Model Dark blue

predictions.

Figure 6.10: training loss over training iterations

6.2. Adaptive mesh refinements results

The two best performing models above have been evaluated on 40 different sample problems, each with a different right hand side and a different finite element mesh. The models (residual) error estimates have been used to refine the mesh 4 times and the average global error, average number of nodes and the two multiplied have been collected into a table in table 6.2 As can be seen in table 6.2, the best performing model and ground truth give comparable results when multiplying the average number of nodes with the average global error. This is not the case for the test transformer which is the second best performing model. Thus adaptive mesh refinement using the first transformer the performance is comparable to that of using the ground truth. Whereas with the test transformer the performance can be seen as twice as bad.

model	avg global error	avg #nodes	avg #nodes * global error
Transformer original	5.4879E-02	23.6	1.2951
Transformer Test	1.0379E-01	23.4	2.4287
ground truth	1.6106E-02	71.1	1.1452

Table 6.3: results of adaptive mesh refinements using different error estimates

7

Discussion

In the results the transformer models perform significantly better which can be attributed to the data being amenable to the transformer model. Recall the pipeline essentially generates data that comes from sample x values on the element and adjacent elements for a visual depiction see figure 7.1.

The resulting output is in mathematical terms:

$$\mathbf{x}^{prev} = \begin{bmatrix} x_0^{prev} \\ x_1^{prev} \\ \vdots \\ x_N^{prev} \end{bmatrix}$$

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_N \end{bmatrix}$$

$$\mathbf{x}^{next} = \begin{bmatrix} x_0^{next} \\ x_1^{next} \\ \vdots \\ x_N^{next} \end{bmatrix}$$

$$\vdots$$

$$\mathbf{f} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_N \end{bmatrix}$$

Now the vectors that come out of this can be fed as embedding tokens into the transformer model. These tokens then can attend (or in other words pass on data) to one another in the multi-head attention layer. And the result of this is then fed to the feedforward layer. The way each of these vectors is made is that each vector has similar information in it: the $\mathbf{f}$ vector contains evaluations of the right hand side on the sample x values for example. Hence you already introduce a inductive bias to the network that that data should be lumped together and should attend to other lumped data. This was the reasoning behind using the transformer model.

MLP models do not have this inductive bias. From all of the vectors above only the first and last entry are

taken and fed to the model:

$$[x^0_{prev} \quad x^N_{prev} \quad x^0_{next} \quad x^N_{next} \quad \dots \quad \mathbf{f}^0 \quad \mathbf{f}^N]$$

The multilayer perceptron is fed all these entries as seperate features with no hints to what the relationship of these features are with respect to the other features, which will make learning very difficult. Only the first and last entries are fed since the model otherwise would have a very difficult time learning anything at all.

Whether the explanations given here are the real reasons why there are such differences in performance is debatable since neural networks are black box models thus researchers oftentimes use their intuition and heuristics that are usually derived from experience rather than from rigorous deduction.

All the transformer models with more dense layers than the first transformer model performed significantly worse than the first transformer model, to investigate whether the number of dense layers were really the culprit, all the other hyperparameters that were different to the hyperparameters of the first transformer model were one by one changed to the hyperparameters of the first transformer model, having little to no effect. After that the test transformer model was developed that had an identical architecture as that of the first transformer model except that the number of dense layers (linear layer with afterwards an activation layer) were changed. The result can be seen in figure 6.10. How its modified is schematically displayed as in figure 7.2 . Thus the conclusion can be drawn that too many dense layers will hurt performance, but too little can also hurt performance but significantly less so.

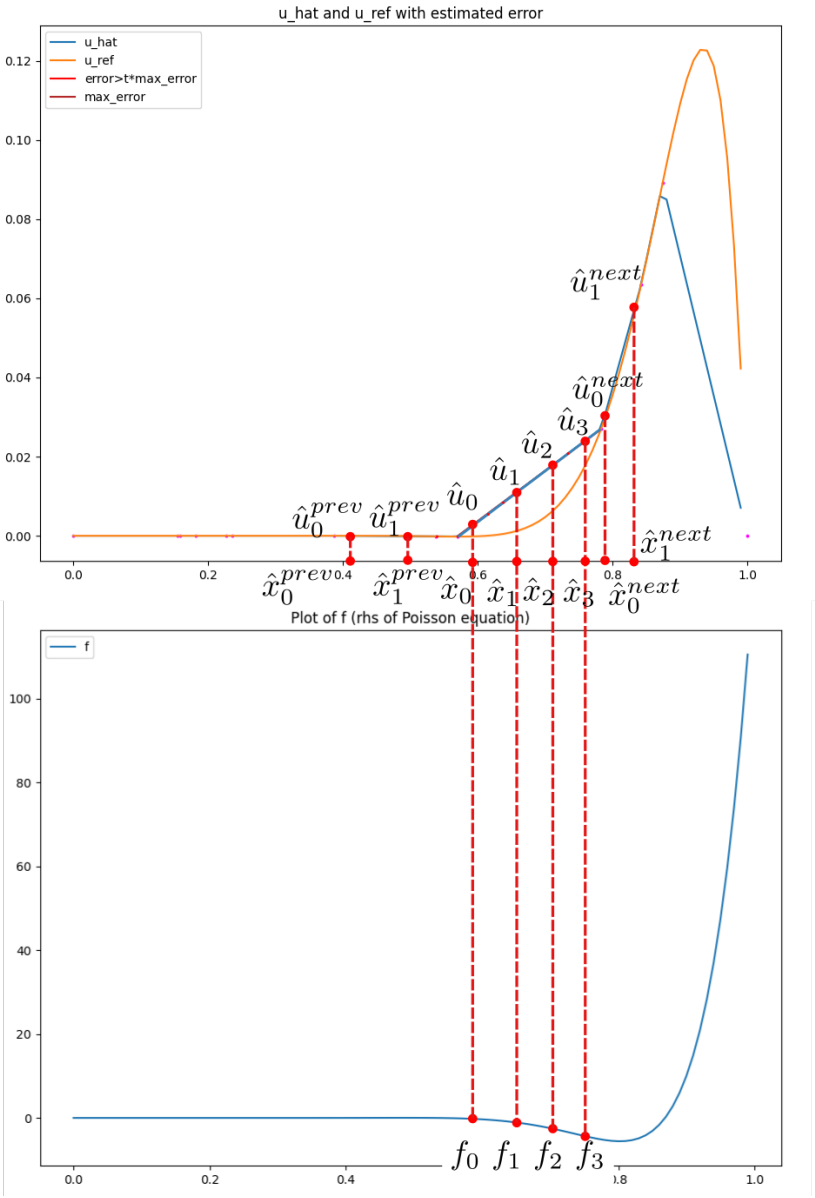


Figure 7.1: Visual depiction of the sampling

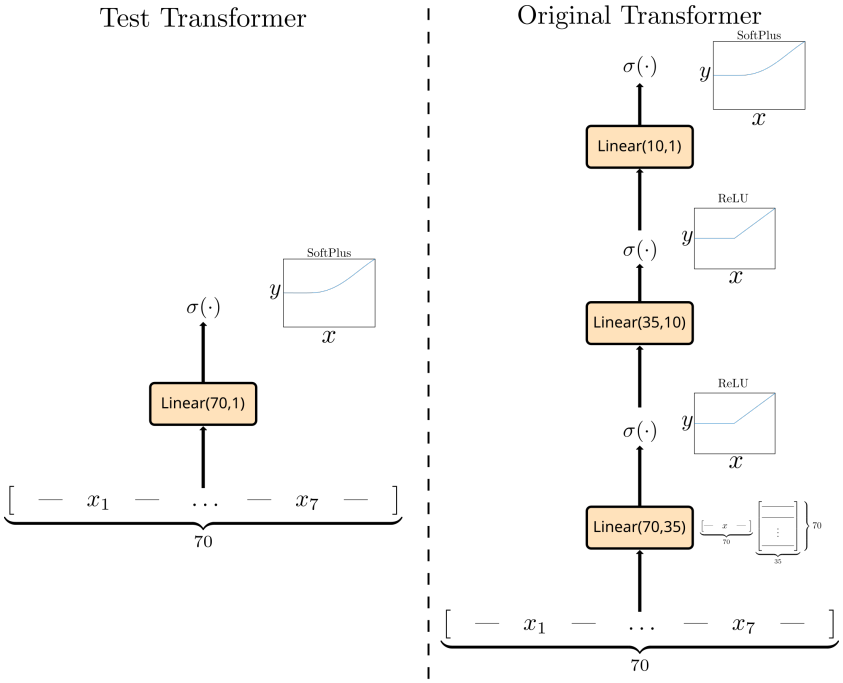


Figure 7.2: Visual depiction of the layers after the transformer-encoder layers

8

Conclusion and future work

8.1. Conclusion

As can be seen in chapter 6 results the quality of the residual error estimate strongly depends on the model that is used. The majority of the models failed to give a good error estimate. The only model that gave a good estimate was the first Transformer model and the test Transformer model.

8.2. Future work

Since Transformers [1] work well for this task it can be expected that other forms of graph neural networks could work well for this problem as well.

Furthermore a way to extend this problem is by letting the network output the 5%, 50% and 95% quantiles so that the model can output how confident it is in its prediction, which can be useful when using these estimates for Adaptive mesh refinements.

Another extension to this method is letting the model predict 3 values: the residual error of an element, the residual error of an element after splitting the element and predicting the error after doing p-refinement. This too can make the model more useful when using it for Adaptive mesh refinement.

Lastly, most useful differential equations are 2d or 3d, however because of time constraints this thesis only covers developing neural error estimates for 1d differential equations, thus extending this method to higher dimensions is left as future work.

[1] provided that it has only a minimal amount dense layers it feeds its output to, see Appendix D

A

MLP model

```
class MLP_hyperopt(nn.Module):
    '''
        Multilayer Perceptron
    '''
    def __init__(self,hyper_dict):
        #invoke superclass initialization
        super().__init__()

        self.num_layers = hyper_dict["num_layers"]
        self.dropout1 = hyper_dict["dropout1"]
        self.dropout2 = hyper_dict["dropout2"]
        self.dropout3 = hyper_dict["dropout3"]
        self.layer_width = hyper_dict["layer_width"]

        layer_size = 14
        layer_list = []
        while layer_size*2 < self.layer_width:
            linear = nn.Linear(layer_size,layer_size*2)
            activation = nn.ReLU()
            layer_list.append(linear)
            layer_list.append(activation)
            layer_size=2*layer_size
        linear = nn.Linear(layer_size,self.layer_width)
        activation = nn.ReLU()
        layer_list.append(linear)
        layer_list.append(activation)
        #print(layer_list)
        #print(OrderedDict([layer_list]))
        self.begin_layers = nn.Sequential(*layer_list)
        #print(self.begin_layers)
        self.dropout_layer1 = nn.Dropout(self.dropout1)
        layer_list = []
        for i in range(self.num_layers):
            linear = nn.Linear(self.layer_width,self.layer_width)
            activation = nn.ReLU()
            layer_list.append(linear)
            layer_list.append(activation)
            if i == self.num_layers//2:
```

```
            layer_list.append(nn.Dropout(self.dropout2))
        self.middle_layers = nn.Sequential(*layer_list)
        #print(self.middle_layers)
        self.dropout_layer3 = nn.Dropout(self.dropout3)

        layer_list = []
        layer_size = self.layer_width
        while layer_size//2 > 1:
            linear = nn.Linear(layer_size,layer_size//2)
            activation = nn.ReLU()
            layer_list.append(linear)
            layer_list.append(activation)
            layer_size=layer_size//2
        linear = nn.Linear(layer_size,layer_size//2)
        activation = nn.Softplus()
        layer_list.append(linear)
        layer_list.append(activation)
        self.end_layers = nn.Sequential(*layer_list)
        #print(self.end_layers)



    def forward(self,x):
        '''Forward pass'''

        #dropout1 = nn.Dropout(self.dropout1)
        x = self.begin_layers(x)
        x = self.dropout_layer1(x)
        x = self.middle_layers(x)
        x = self.dropout_layer3(x)
        #print(x.size())
        #print(self.end_layers)
        x = self.end_layers(x)

        return x
```

B

Transformer model (hyperparameter optimisation version)

```
# Transformer model
class TransformerHyperopt(nn.Module):
    '''
        Transformer
    '''
    def __init__(self,hyper_dict):
        #invoke superclass initialization
        super().__init__()

        my_device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        #unpacking the hyperparameters
        d_model = hyper_dict["d_model"]
        dropout = hyper_dict["dropout"]
        num_tokens = hyper_dict["num_tokens"]
        nheads = hyper_dict["nheads"]
        norm_first = hyper_dict["norm_first"]
        layer_norm_eps = hyper_dict["layer_norm_eps"]
        dim_feedforward = hyper_dict["dim_feedforward"]
        num_layers = hyper_dict["num_layers"]
        self.positional_encoding = hyper_dict["positional_encoding"]
        self.d_model = d_model

        if d_model is not None:
            self.expand_token_size =nn.Sequential(nn.Linear(10,d_model), \
            nn.ReLU()).to(device=my_device)

        # if the tokens do not get expanded then positional encoding will
        # take place on the original tokens
        if d_model is None:
            self.pos_encoding_layer = PositionalEncoding(10,dropout, \
            max_len=num_tokens).to(device=my_device)
            d_model=10
        else:
            self.pos_encoding_layer = PositionalEncoding(d_model,dropout, \
            max_len=num_tokens).to(device=my_device)
```

```
        transformer_layers = [TransformerEncoder(d_model,nheads,dropout,num_tokens, \
        norm_first,layer_norm_eps,dim_feedforward).to(device=my_device) \
        for i in range(num_layers)]


        self.encoder= nn.Sequential(
            *transformer_layers
        )
        self.flatten = nn.Flatten(1,2)

        layer_list = []
        layer_size = num_tokens*d_model
        while layer_size//2 > 1:
            linear = nn.Linear(layer_size,layer_size//2)
            activation = nn.ReLU()
            layer_list.append(linear)
            layer_list.append(activation)
            layer_size=layer_size//2
        linear = nn.Linear(layer_size,layer_size//2)
        activation = nn.Softplus()
        layer_list.append(linear)
        layer_list.append(activation)
        self.last_layers = nn.Sequential(*layer_list).to(device=my_device)

#         self.softplus= nn.Softplus()
    def forward(self,x):
        '''Forward pass'''
        if self.d_model is not None:
            x = self.expand_token_size(x)
        if self.positional_encoding:
            x = self.pos_encoding_layer(x)
        x = self.encoder(x)
        x = self.flatten(x)

        x = self.last_layers(x)
        #x =self.softplus(x)
        return x
```

C

Transformer model libraries (hyperparameter optimisation version)

```
'''
#########################################################################################################################
# Transformer libraries
# below the libraries needed to run the transformer will be declared
#########################################################################################################################
'''
class attention(nn.Module):
    '''
       attention layer
    '''
    def __init__(self):
        #invoke superclass initialization
        super().__init__()

        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

        self.softmax = nn.Softmax(dim=1).to(device)




    def forward(self,Q,K,V):
        '''Forward pass'''
        #Q,K,V = x
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        attention_filter = torch.matmul(Q,torch.transpose(K,-1,-2))
        #print(attention_filter.size()[0])
        d_k=torch.sqrt(torch.tensor(attention_filter.size()[0], device=device))
        attention_filter = attention_filter/d_k
        soft_attention_filter = self.softmax(attention_filter)
        #print(soft_attention_filter.size())
        output = torch.matmul(soft_attention_filter,V)
        #print(output.size())
        return output

class multihead_attention(nn.Module):
```

```
    '''
       multihead attention layer
    '''
    def __init__(self,d_model,nheads, dropout):
        #invoke superclass initialization
        super().__init__()
        my_device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.num_heads = nheads
        self.embed_dim = d_model
        self.dropout = nn.Dropout(dropout)
        #initialise the key layers i.e. W_K^i
        self.W_K = [nn.Linear(self.embed_dim,self.embed_dim, \
        bias=False).to(device=my_device) for i in range(nheads)]

        #initialise the Query layers i.e. W_Q^i
        self.W_Q = [nn.Linear(self.embed_dim,self.embed_dim, \
        bias = False).to(device=my_device) for i in range(nheads)]

        #initialise the Value layers i.e. W_V^i
        self.W_V = [nn.Linear(self.embed_dim,self.embed_dim, \
        bias=False).to(device=my_device) for i in range(nheads)]

        #initialise the attention layers
        self.att = [attention().to(device=my_device) for i in range(nheads)]

        #initialise the output layer
        self.O_layer = nn.Linear(self.embed_dim*nheads,self.embed_dim, \
        bias=False).to(device=my_device)




    def forward(self,input):
        '''Forward pass'''
        #input = self.dropout(input)
        att=[]
        for i in range(self.num_heads):
            att.append(self.dropout(self.att[i](self.W_Q[i](input), \
            self.W_K[i](input),self.W_V[i](input))))
            #print(self.att[i](self.W_Q[i](input),self.W_K[i](input), \
            self.W_V[i](input)).size())

        attention_concat = torch.cat(att,-1)

        output = self.O_layer(attention_concat)

        return output

class add_and_norm(nn.Module):
    '''
       add & normlayer
    '''
    def __init__(self,d_model,norm_first,num_tokens=None,layer_norm_eps = 1e-5):
        #invoke superclass initialization
        super().__init__()
```

```
        self.norm_first = norm_first
        if num_tokens is None:
            self.LayerNorm = nn.LayerNorm(d_model,eps=layer_norm_eps)
        else:
            self.LayerNorm = nn.LayerNorm((num_tokens,d_model),eps=layer_norm_eps)



    def forward(self,input1,input2):
        '''Forward pass'''
        if self.norm_first == False:
            x,output_layer = input1,input2
            add = x+output_layer
            output = self.LayerNorm(add)
        else:
            x,output_layer = input1,input2
            add = x+self.LayerNorm(output_layer)
            output=add
        return output


class FFN(nn.Module):
    '''
       Feed Forward Layer
    '''
    def __init__(self, d_model ,dim_feedforward,dropout):
        #invoke superclass initialization
        super().__init__()



        self.dropout = nn.Dropout(dropout)
        self.layers = nn.Sequential(
            nn.Linear(d_model,dim_feedforward),#
            nn.ReLU(),
            nn.Linear(dim_feedforward,d_model),
        )

    def forward(self,x):
        '''Forward pass'''

        return self.dropout(self.layers(x))

class TransformerEncoder(nn.Module):
    '''
       Transformer encoder
    '''
    def __init__(self,d_model,nheads,dropout,num_tokens,norm_first,\
    layer_norm_eps,dim_feedforward):
        #invoke superclass initialization
        super().__init__()

        #self.input_embedding = ...
        self.multi_head_attention = multihead_attention(d_model,nheads,dropout)
        self.add_and_norm         = add_and_norm(d_model,norm_first,num_tokens,layer_norm_eps)
```

```
        self.FFN                    = FFN(d_model,dim_feedforward,dropout)



    def forward(self,input):
        '''Forward pass'''
        multihead_att = self.multi_head_attention(input)
        x=self.add_and_norm(input,multihead_att)

        feedforward = self.FFN(x)
        output = self.add_and_norm(x,feedforward)
        return output

import math
class PositionalEncoding(nn.Module):

    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = 5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * \
        (-math.log(10000.0) / d_model))
        pe = torch.zeros(max_len, d_model)
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        #pe = pe.unsqueeze(0).transpose(0, 1)
        #print(pe)
        self.register_buffer('pe', pe)

    def forward(self, x):
        """
        Args:
            x: Tensor, shape [batch_size, seq_len, embedding_dim]
        """
        #print(self.pe.size())
        x = x + self.pe[:x.size(0),:]
        return self.dropout(x)



'''
#################################################################################################
# Original Transformer
# below the libraries needed to run the transformer will be declared
#################################################################################################
'''
import TransformerLib
class Transformer(nn.Module):
    '''
        Transformer
    '''
    def __init__(self):
        #invoke superclass initialization
        super().__init__()
```

```
        #use nn.Sequential to stack four densely connected linear layers
        #with relu activation function
        self.encoder= nn.Sequential(
            #nn.LayerNorm(22),
            TransformerEncoder_first_model(),
            TransformerEncoder_first_model(),
            TransformerEncoder_first_model(),
            TransformerEncoder_first_model(),
            TransformerEncoder_first_model(),
            TransformerEncoder_first_model(),
            #nn.Sigmoid()
        )
        self.flatten = nn.Flatten(1,2)
        self.last_layers = nn.Sequential(
            nn.Linear(70,35),
            nn.ReLU(),
            nn.Linear(35,10),
            nn.ReLU(),
            nn.Linear(10,1)
        )
        #self.Linear = nn.Linear(12,1)
        self.softplus= nn.Softplus()
    def forward(self,x):
        '''Forward pass'''
        x=self.encoder(x)
        x = self.flatten(x)

        x = self.last_layers(x)
        x =self.softplus(x)
        return x


class TransformerEncoder_first_model(nn.Module):
    '''
       Transformer encoder
    '''
    def __init__(self):
        #invoke superclass initialization
        super().__init__()

        #self.K_layer = nn.Linear(5,3,bias=False).to(my_device)
        #self.input_embedding = ...
        self.multi_head_attention = TransformerLib.multihead_attention()
        self.add_and_norm         = TransformerLib.add_and_norm()
        self.FFN                  = TransformerLib.FFN()



    def forward(self,input):
        '''Forward pass'''
        multihead_att = self.multi_head_attention(input)
        x=self.add_and_norm(input,multihead_att)

        feedforward = self.FFN(x)
        output = self.add_and_norm(x,feedforward)
        return output
```

D

Original Transformer model

```
'''
#########################################################################################################
# Original Transformer
# below the libraries needed to run the transformer will be declared
#########################################################################################################
'''
import TransformerLib
class Transformer(nn.Module):
    '''
        Transformer
    '''
    def __init__(self):
        #invoke superclass initialization
        super().__init__()

        #use nn.Sequential to stack four densely connected linear layers
        #with relu activation function
        self.encoder= nn.Sequential(
            #nn.LayerNorm(22),
            TransformerEncoder_first_model(),
            TransformerEncoder_first_model(),
            TransformerEncoder_first_model(),
            TransformerEncoder_first_model(),
            TransformerEncoder_first_model(),
            TransformerEncoder_first_model(),
            #nn.Sigmoid()
        )
        self.flatten = nn.Flatten(1,2)
        self.last_layers = nn.Sequential(
            nn.Linear(70,35),
            nn.ReLU(),
            nn.Linear(35,10),
            nn.ReLU(),
            nn.Linear(10,1)
        )
        #self.Linear = nn.Linear(12,1)
        self.softplus= nn.Softplus()
    def forward(self,x):
```

```
        '''Forward pass'''
        x=self.encoder(x)
        x = self.flatten(x)

        x = self.last_layers(x)
        x =self.softplus(x)
        return x


class TransformerEncoder_first_model(nn.Module):
    '''
       Transformer encoder
    '''
    def __init__(self):
        #invoke superclass initialization
        super().__init__()

        #self.K_layer = nn.Linear(5,3,bias=False).to(my_device)
        #self.input_embedding = ...
        self.multi_head_attention = TransformerLib.multihead_attention()
        self.add_and_norm         = TransformerLib.add_and_norm()
        self.FFN                  = TransformerLib.FFN()



    def forward(self,input):
        '''Forward pass'''
        multihead_att = self.multi_head_attention(input)
        x=self.add_and_norm(input,multihead_att)

        feedforward = self.FFN(x)
        output = self.add_and_norm(x,feedforward)
        return output
```

Bibliography

[1] Adam. URL https://pytorch.org/docs/stable/generated/torch.optim.Adam.html. Accessed:2022-07-12.

[2] Relu. URL https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html. Accessed:2022-07-12.

[3] Sgd. URL https://pytorch.org/docs/stable/generated/torch.optim.SGD.html. Accessed:2022-07-12.

[4] Softplus. URL https://pytorch.org/docs/stable/generated/torch.nn.Softplus.html. Accessed:2022-07-12.

[5] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. 2016.

[6] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. *Advances in Neural Information Processing Systems*, 24(2):2546–2554, 2011.

[7] Chiu Ling Chan, Felix Scholz, and Thomas Takacs. Locally refined quad meshing for linear elasticity problems based on convolutional neural networks. 2022.

[8] Amirhossein Kazemnejad. Transformer architecture: The positional encoding, Sep 2019. URL https://kazemnejad.com/blog/transformer_architecture_positional_encoding/.

[9] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2014.

[10] Mats G Larson and Fredrik Bengzon. *The finite element method: Theory, implementation, and applications*. Texts in computational science and engineering. Springer, Berlin, Germany, 2013 edition, January 2013.

[11] Liam Li. Massively parallel hyperparameter optimization, Dec 2019. URL https://blog.ml.cmu.edu/2018/12/12/massively-parallel-hyperparameter-optimization/.

[12] David Li-Bland. Saddle points and stochastic gradient descent, Nov 2018. URL https://davidlibland.github.io/posts/2018-11-10-saddle-points-and-sdg.html.

[13] Maciej Paszyński, Rafał Grzeszczuk, David Pardo, and Leszek Demkowicz. Deep learning driven self-adaptive hp finite element method. In *Computational Science – ICCS 2021*, Lecture notes in computer science, pages 114–121. Springer International Publishing, Cham, 2021.

[14] Yurii Shevchuk. Hyperparameter optimization for neural networks, Dec 2016. URL http://neupy.com/2016/12/17/hyperparameter_optimization_for_neural_networks.html#tree-structured-parzen-estimators-tpe.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2017.

[16] Rüdiger Verfürth. Auxiliary results. In *A Posteriori Error Estimation Techniques for Finite Element Methods*, pages 79–150. Oxford University Press, April 2013.

[17] Aston Zhang, Zack C. Lipton, Mu Li, Alex J. Smola, Brent Werness, Rachel Hu, Shuai Zhang, Yi Tay, Anirudh Dagar, and Yuan Tang. Minibatch stochastic gradient descent. http://d2l.ai/chapter_optimization/minibatch-sgd.html". Accessed:2022-07-12.

[18] Zheyan Zhang, Yongxing Wang, Peter K Jimack, and He Wang. MeshingNet: A new mesh generation method based on deep learning. 2020.