FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

DEPARTMENT OF COMPUTER SCIENCE

Computer Science MSc Thesis, 30 ECTS

Peter Kanstrup Larsen `<zlc797@ku.dk>`

Application of Probabilistic Machine Learning Methods for Protein Generation

MSc in Computer Science, DIKU

Supervisor: Cosmin Eugen Oancea

May 31, 2023, Block 3+4

Abstract

The recent introduction of denoising diffusion probabilistic models has led to a surge in the field of AI. While text-to-image models such as OpenAI's DALL-E 2 have recieved widespread public attention, many scientific fields are looking towards the technology as well. Researches in the field of protein design recently demonstrated that such models could be used for protein generation, achieving results comparable to state-of-the-art models. In this thesis, we show how a diffusion model can be implemented in Futhark, and compare our results to an equivalent model implemented using the PyTorch library.

Acknowledgements

I would like to thank my supervisor Cosmin, for his invaluable guidance and support throughout the project. I would also like to thank Nikolaj for being very quick to help and for his assistance with the Futhark server. I would also like to thank Ola for his great help with specialized knowledge in the area.

Contents

1 Introduction

This thesis combines the fields of parallel programming, machine learning and protein design. The following subsections provide brief overviews of the most important components of the thesis, concluding with a listing of our contributions.

1.1 Futhark

Futhark [9] is a purely functional, data-parallel, array language. While most people in the field of computer science have an idea of what constitutes a purely functional language, the concepts of *data-parallelism* and *array languages* are less common, and so we provide a quick overview.

Data-parallelism is the paradigm of parallelizing a program that performs the same operation multiple times on different sections of a large dataset. A classical example of an operation that might exploit data parallelism is the `map` operation on arrays. With sequential execution, a `map` operation on an array with n elements will take $O(n)$ time. However, since the function run by the `map` has no side effects (assuming that the language is indeed purely functional), we can run the function concurrently on each element, which yields a runtime of only $O(1)$.

Array languages are simply languages that permit operations on array structures rather than just primitive types such as integers and booleans. For example, an array language might allow computing the dot product of two matrices `A` and `B` by simply writing `A * B`. Futhark does not support basic operations such as addition, multiplication, etc. on arrays, but instead provides a set of primitive functions such as `map` and `reduce` for operations on arrays. In Futhark, these operations are called Second-Order Array Combinators (SOACs), and, while their semantics are defined in a sequential manner, they are typically compiled to parallel code.

This brings us to the Futhark compiler[1], which is the cornerstone of the Futhark language. It is a heavily optimizing compiler which generates either OpenCL, CUDA or sequential C code for use on general purpose GPUs. Many of these optimizations are non-trivial; for example, the compiler must decide how to deal with *nested parallelism*, which may occurs when two or more SOACs are nested, such as in figure 1. We want to avoid too much nested parallelism, since directly translating such code into GPU code may lead to creating more threads than what the hardware actually supports. Suppose that, in figure 1, `xxs` contains $1,000$ arrays with $1,000$ elements each. If we tried to create a thread for each task, it would lead to creating $1,000,000$ threads; much larger than the number of cores on a typical GPU.

```
    let ys = map (\xs -> reduce (+) 0 xs) xxs
```

Fig. 1: A simple example of nested parallelism.

In almost all cases, such nested parallelism can be transformed by the compiler into *flat parallelism*, which in the case for a `map` nested in a `reduce` would look

[1] https://github.com/diku-dk/futhark

as shown in figure 2. Here, `xxs_shp` is the shape of `xxs`, and `xxs_val` is the flattened array of values in `xxs`.

```
    let xxs_flg = mkFlagArray xxs_shp 0 (replicate len 1)
    let sc_xxs = sgmScanInc (+) 0 xxs_flg xxs_val
    let indsp1 = scanInc (+) 0 xxs_shp
    let ys = map2 (\shp ip1 -> if shp == 0 then 0
                               else sc_xxs[ip1-1]
                ) xxs_shp indsp1
```

Fig. 2: Figure 1 converted to flat parallelism.

Notice that we no longer have any nested SOACs. Many similar types of flattening optimizations exists, all of which the Futhark compiler can utilize.

1.2 Protein Design

Protein design is, as the name suggests, the discipline of designing protein molecules in order to obtain some desired behavior for said protein. The field can be divided into two areas: *protein redesign*, which modifies existing protein structures, and *de novo* design, which generates new proteins from scratch. For this thesis, we focus our attention on the last one.

Protein design is based on the notion of protein *folding*. Initially, an unfolded protein is simply as a long chain of amino acids. However, these amino acids can interact with each other, forming bonds that constrain the protein into a certain shape, which then results in what we call the folded protein. Clearly, then, understanding how a protein folds relies on understanding how its amino acids interact, specifically, how the force they exert on each other bends the protein into a certain shape. This cumulative force is called the *free energy* of the protein. An unfolded protein has a relatively large amount of free energy, meaning that the amino acids will be exerting a lot of force on each other. A folded protein has comparatively low amount of free energy, such that the amino acids exert only little force on each other. The structural state of a protein that has a free energy minimum is called the native state. This leaves us with an optimization problem: given a three dimensional protein shape, find the sequence of amino acids such that the free energy minimum is minimized.

Optimization problems are commonplace in many scientific areas, so at first glance, the task seems straightforward enough. However, new problems arise when we take a closer look. In order to optimize for a sequence that reaches a free energy minimum, we must have some sort of energy function that relates amino acid sequences to their free energy. Many different types of energy functions exists, but they are common in that that they can provide either speed or accuracy, but not both. The most accurate types of energy functions are those that are based on quantum mechanical simulations, but they are also much too computationally expensive to be used for protein design. On the other end, there are heuristically-based energy functions which are quite fast, but lack accuracy. In the middle, there are the physically-based functions, which are based on the ”correct” quantum mechanical simulations, but with some simplifications which make them less computationally expensive. One of the most widely used [3] energy functions is the Rosetta algorithm [22], which may be considered state of

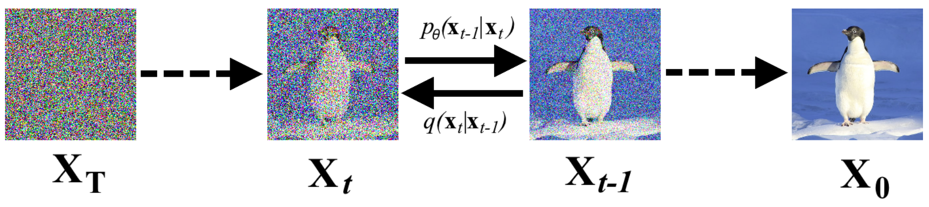
$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$
$q(\mathbf{x}_t|\mathbf{x}_{t-1})$
$\mathbf{X}_T$
$\mathbf{X}_t$
$\mathbf{X}_{t-1}$
$\mathbf{X}_0$

Fig. 3: The diffusion model

the art. It uses a Monte Carlo strategy [2] to generate the amino acid sequences. Successful Rosetta predictions fall within a Cα RMSE[3] of 3-6 Ångström when compared to real-world proteins designed with the same structure.

However, within the last decade, deep learning has emerged as an alternative to the "traditional" probabilistic models such as Rosetta. One of the most recent breakthroughs in the field has been the AlphaFold model [12], a deep learning model developed by DeepMind. AlphaFold has made protein folding predictions with a Cα RMSE of ca. 1.5 Ångström, a significant improvement on existing methods. Building on top of this work, there have been attempts to use so-called *diffusion models* for prediction as well [32]. This particular diffusion model (RF*diffusion*) also yield good results, in some cases making predictions very close to AlphaFold in terms of accuracy.

1.3 Diffusion Models

Diffusion models are a type of generative models. A relatively new concept, the diffusion model was first introduced in 2015 by Sohl et al [28]. The model consists of two steps: a forward diffusion process, and a backward diffusion process. The forward diffusion process iteratively applies noise to an image using some distribution (e.g. Gaussian), which eventually results in an image which is pure noise. The backwards diffusion process uses a neural network that is trained to remove noise from an image, eventually resulting in the original image from the forward direction. The idea here is then that one can provide an image with completely random noise (using the distribution from the forward diffusion process), and then generate an entirely new image which is similar to, but not quite the same as, the original image. For an idea of how the model works, see figure 3.

While there are some non-obvious ways to improve the noise generation in the forward direction (as investigated in later papers [20]), the "core element" of the diffusion model is the neural network that "denoises" and thus permits generation of novel images. Thus, selecting a type of neural network that can perform effective denoising is an important thing to consider.

1.4 Contributions

This brings us to the concrete contributions of the thesis. Since deep learning requires computations that are suitable for performing on GPUs, we want to leverage the heavily optimizing compiler of Futhark to show that the language

[2] Randomized algorithm with a bounded running time but some margin of error
[3] The root-mean-square deviation of atomic positions

can find use in the field of protein design; more specifically, that the language can be used for implementing diffusion models which can compare to existing implementations in terms of runtime. Our concrete contributions are as follows:

1. We attempt to design and implement a diffusion model using the LeNet-5 neural network design.
2. We implement a U-Net in Futhark and compare its performance to existing implementations.
3. We implement a diffusion model in Futhark using the U-Net neural network design and compare its performance to existing implementations.

2 Background

2.1 Diffusion Models

We now dive a little deeper into the mathematical theory behind diffusion models, based on the paper from Ho et al. [10]. We begin by introducing some initial definitions.

First, we have an initial distribution of original images $q(\mathbf{x}_0)$, from which we can sample an image $\mathbf{x}_0 \sim q(\mathbf{x}_0)$. Now, the forward diffusion process is defined as follows:

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) := \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \tag{1}$$

Here, $\mathcal{N}$ applies noise based on a normal distribution, where $\mathbf{x}_t$ is the output, $\sqrt{1-\beta_t}\mathbf{x}_{t-1}$ is the mean and $\beta_t\mathbf{I}$ (which is just β_t) is the variance. β_t itself is a variance schedule, which adjusts the amount of noise applied at different time steps. The reference paper has a linearly increasing β, going from $\beta_1 = 10^{-4}$ to $\beta_T = 0.02$. This forward diffusion process then yields images $\mathbf{x}_1, ..., \mathbf{x}_T$, where T is the predefined maximum number of steps for applying noise, meaning that $\mathbf{x}_T$ is the completely noised-image. The paper we base this section on has $T = 1000$ (although later papers manage to reduce the maximum number of steps significantly). A property of the forward process, as mentioned in the reference paper, is that it admits sampling $\mathbf{x}_t$ at an arbitrary timestep t without iteratively computing all of the previous steps (i.e. it can be sampled in closed form):

$$\alpha_t := 1 - \beta_t \tag{2}$$

$$\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s \tag{3}$$

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}) \tag{4}$$

This property will be relevant later when implementing the neural network, since it can greatly reduce the number of computations necessary when training.

We now define the probability distribution of the backward process $p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$ using a neural network with parameters θ:

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) := \mathcal{N}(\mathbf{x}_{t-1}, \mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t)) \tag{5}$$

where μ_θ and Σ_θ are parameters of the neural network. While this would normally mean that we would have to train the neural network to learn both the mean and the variance, the reference paper determines that high quality results can be obtained with a fixed variance, and so we only need to let the neural network learn the mean μ_θ.

We must then define an objective function for the neural network to learn the mean. The reference paper settles on the following definition for μ_θ [4]:

$$\mu_\theta(\mathbf{x}_t, t) = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(\mathbf{x}_t, t) \right) \tag{6}$$

where ϵ_θ is a function approximator that can predict ϵ from $\mathbf{x}_t$. This gives us the following objective function which we can use to train our model:

$$L_t = \left\| \epsilon - \epsilon_\theta \left(\sqrt{\bar{\alpha}} \mathbf{x}_0 + \sqrt{(1 - \bar{\alpha}_t}, t \right) \right\|^2 \tag{7}$$

2.2 Neural Network

An (artificial) neural network is a collection of connected nodes called neurons. A neuron can receive a signal from another connected neuron, modify that signal and then send it to yet another neuron. The neurons in the network are divided into layers, where neurons in the same layer typically perform the same task. A neural network always contains an input layer and an output layer, and any number of layers in between, which signals travel over.

In a diffusion model, the most complex and resource-intensive operation is the process of identifying ”what” to remove when performing the backwards diffusion — i.e. how to identify noise. For this, simply choosing any neural network is insufficient, since neural networks can vary greatly in their computational power.

One of the simplest types of neural networks are feedforward neural networks (FNN), which are also some of the first neural networks conceived [27]. Common FNN variants are multilayer perceptrons, which are fully-connected neural networks (all network neurons in one layer are connected to the one in the previous layer). While multilayer perceptrons are suitable for a variety of applications, they are not in general suitable for image analysis, since the number of weights per neuron quickly grows out of control due to full-connectivity.

This leads us to convolutional neural networks (CNN), which are, on the other hand, commonly used in the field of image analysis. CNNs do not suffer from issues caused by full-connectivity, since they use *kernels* which only receive input from a small group of neurons, such as a small square of pixels in an image. For this reason, we choose to focus on CNN variants in this thesis. There already exists a lot of research that applies CNN, particularly within the field of imaging. A specific example is classification of digits in the MNIST dataset, where the currently best performing models are CNNs[5].

[4] The definition is obtained through a series of derivations not included here, but they can be found in the associated paper [10].

[5] https://en.wikipedia.org/wiki/MNIST_database#Classifiers

2.2.1 Positional Embeddings / Time Embeddings

Positional embeddings is a technique mainly used in natural language processing to give information about the position of tokens in a sequence.

However, in the DDPM we use positional embeddings as *time embeddings*. The information given from the embeddings is crucial, as it serves to indicate the current timestep of the image being processed by the model. This additional information plays a big role for the entire generative model, since the amount of noise present in the image varies. An image at an early timestep will have much more information about the original data than at a later timestep. This means we need some way to indicate wether the model should focus on learning detailed features, or on the data distribution.

A single number, such as the timestep, could serve as a basic form of embedding, but this would limit the complexity of the patterns and relationships that the model could learn.

A commonly used positional encoding is *sinosoidal positional embeddings*, introduced in "Attention is All You Need" by Vaswani et al. [29] to provide a sense of order of words in a sequence.

In sinusoidal positional embeddings, a unique vector encodes each position. For a given position, the embedding vector is composed of sine and cosine functions of different frequencies. The frequency is determined based on whether the index of the vector is even or odd:

$$PE_{pos,2i} = sin(pos/10000^{2i/d_{model}}) \tag{8}$$

$$PE_{pos,2i+1} = cos(pos/10000^{2i/d_{model}}) \tag{9}$$

Sinusoidal embeddings offer several advantages: they provide a unique and scalable representation for each position, and they output values in the range of -1 to 1, which prevents potential numerical instability issues such as overflow or underflow.

Once computed, these embedding vectors are incorporated into the model using various methods. In our implementation, the vector is passed through a fully connected layer before being added to the convolutional layer outputs. This approach functions almost like an "additional bias" that is adjusted according to the specific timestep, thus giving the model optimized, timestep-specific parameters.

3 Design & Implementation

This section presents the design decisions made when implementing our convolutional neural network, as well as relevant sections of code from the implementation.

The implementation can be found at: `https://github.com/PeterLarsen404/Diffusion`

3.1 Forward Diffusion

We know from the background section that this part of the model generates the noisy image at a given timestep t, by sampling from the $q(x_t|x_0)$ distribution. To do this we need $\bar{\alpha}$, which is generated from the α sequence which is again generated from the β sequence. We define the following functions for computing the sequences:

```
def mk_beta (steps : i64) (start : f64) (stop: f64) : [steps]f64 =
  tabulate steps
           (\i ->
             start+(f64.i64 i)*((stop-start)/((f64.i64 steps)-1f64)))

def mk_alpha [n] (betas : [n]f64) : [n]f64 =
  map (\b -> 1f64 - b) betas

def mk_alpha_bar [n] (alphas : [n]f64) : [n]f64 =
  scan (*) 1 alphas
```

What's notable here is that β essentially is just a linspace of `step` elements, from `start` to `stop` with equal numerical distance between them. Otherwise, the creation of α and $\bar{\alpha}$ follows the background math with the use of the SOAC's `map` and `scan`.

We now have all we need to generate the noisy image x_t from the $q(x_t|x_0)$ distribution. The following shows the implementation of how we sample a noisy image at timestep t:

```
def q_sample [n][m] (x0 : [n][m]f64) (t : i64) (alpha_bar : []f64)
                    (seed : i32) : ([n][m]f64,[n][m]f64) =
  let mean = f64.sqrt alpha_bar[t]
  let var = 1f64 - alpha_bar[t]
  let eps = mk_rand_array seed n m
  let noisy_img =
    map2 (map2 (\ i e -> (i * mean) + (e * (f64.sqrt var)))) x0 eps
  in (noisy_img, eps)
```

In addition to `alpha_bar`, `q_sample` takes the original image `x0`, the timestep `t`, and the `seed` we want to generate the random noise from.

We initially compute the `mean` and `var` corresponding to the mean $(\sqrt{\bar{\alpha}_t})$ and variance $(1 - \bar{\alpha}\_t)$ of the closed form. We then use `mk_rand_array` to sample the random noise (ϵ). Finally, we use two nested `map2` to map over each pixel in the image, computing each pixel value of the noisy image.

3.2 Backwards Diffusion

We will in this section go over the key elements of the backwards diffusion.

We know from the background section that the process of backwards diffusion consists of multiple samples from the $p(x_{t-1}|x_t)$ distribution, going from the predefined maximum number of steps T down to 0. We implement the sampling from the $p(x_{t-1}|x_t)$ distribution as follows:

```
def p_sample [n][m][a] (x_t : [n][m]f64) (t : i64) (num_groups : i64)
                       (beta : [a]f64) (alpha : [a]f64)
                       (alpha_bar : [a]f64) (w_images)
                       (seed: i32) : [n][m]f64 =
  let time_embedding =
    sinusoidal_position_embeddings 256 (f64.i64 t) :> [256]f64
  let (eps_theta,_) =
    unet_simple x_t time_embedding num_groups w_images

  let alpha_bar_t = alpha_bar[t]
  let alpha_t = alpha[t]
  let eps_coef = (1f64-alpha_t) / ((1 - alpha_bar_t) ** 0.5f64)
  let var = beta[t]

  let eps =
    if t > 1 then mk_rand_array seed n m
    else replicate n (replicate m 0f64)

  let res = map3 (map3 (\ xt eps_th e ->
    let mean = 1f64 / (alpha_t ** 0.5f64) * (xt - eps_coef * eps_th)
    in mean + (var ** 0.5f64) * e
    )) x_t eps_theta eps
  in res
```

The predicted noise eps_theta is computed using the neural network model. We then compute the noise coefficient $\frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}}}$ using the α and $\bar{\alpha}$ values corresponding to the current timestep, giving us eps_coef. The noise eps is computed using mk_rand_array if the noise step is above 1; otherwise we set eps to 0. We know that the variance var is held constant to the corresponding β value for the current timestep. Finally, we construct the image for timestep $t-1$. For each pixel, we calculate the mean of the distribution $(\frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}}}\epsilon_\theta(x_t, t)))$, and sample using the variance.

Sampling a new image from pure noise is made from the following function sample, which correspond to the sampling algorithm from the DDPM paper:

```
def sample [a] (num_samples : i64) (num_groups : i64) (x_seed : i32)
               (eps_seed : i32) (img_size : i64) (n_steps : i64)
               (beta : [a]f64) (alpha : [a]f64) (alpha_bar : [a]f64)
               (trained_weights) =
  let noise_imgs_seeds = mk_rand_seeds x_seed num_samples
  let noise_imgs =
    map (\ x -> mk_rand_array x img_size img_size) noise_imgs_seeds
  let eps_seeds = mk_rand_seeds eps_seed n_steps
  let eps_seeds2d =
    map (\ x -> mk_rand_seeds x num_samples) eps_seeds
  let generated_imgs =
    loop denoised_imgs = noise_imgs for t_ < n_steps do
      let t = n_steps - t_ - 1i64
      let new_denoised_imgs =
        tabulate num_samples
                 (\ i ->
                   p_sample denoised_imgs[i] t num_groups
                            beta alpha alpha_bar
                            trained_weights eps_seeds2d[t_,i])
      in new_denoised_imgs
  in map (\ x ->
    let max = f64.maximum (flatten x)
    let min = f64.minimum (flatten x)
    in map (map (\ x -> ((x-min) / (max - min)) * (1-(-1))+(-1))) x
  ) generated_imgs
```

Which initially generates the needed seeds for making the noisy start images, and the seed for `p_sample`. Then a loop is run `n_steps` times (`n_steps` is often equal to 1000). In each iteration we call `p_sample` for each image, removing one "step" of noise. This can be done in parallel with the use of `tabulate`.

Finally, the images are normalized and scaled so they fit into the [-1;1] range of the original training images. This range is also used by the DDPM paper.

3.3 Neural Network

One of the main goals of this project was to investigate different neural network variants in order to determine which ones were most suitable for implementing diffusion models. This section describes our approach and results in that area.

It is important to stress at this point that our goal is *not* to implement a Futhark library for constructing convolutional neural networks, but to implementing a specific convolutional network with a specific purpose (for example, we will not be translating all of the PyTorch functions into Futhark).

We previously asserted that convolutional neural networks are highly suitable for tasks concerning image analysis. For this reason, we chose to use a convolutional neural network for image noise prediction. Our initial CNN used the LeNet [17] design. We first give an overview of its structure, and then a further explanation of each of its layers.

After attempting to implement our CNN using the LeNet design, some shortcomings became apparent, and we decided to switch to the u-net design [23], which produced much better results. We later give a more comprehensive explanation of the issues with LeNet and how we implemented the u-net design.

3.3.1 LeNet

We initially chose to use the LeNet design — specifically LeNet-5 — in our implementation. It was first proposed in 1998 by LeCun et al. [17] as one of the earliest convolutional neural networks. Its design can be seen in figure 4. This already brings us to one of the main issues with using LeNet for noise prediction. LeNet-5 was originally developed for MNIST classification (and handwritten digit recognition in general), so it only has 10 "boolean" outputs that indicate which digit it predicts. However, since we want to predict noise from an input image with dimensions 28×28, we need $28 \times 28 = 784$ outputs.

To solve this problem, we have developed a variant of LeNet-5 for this thesis called LeNet-5-Fut. This modifies the design in several ways. First, the output layer now has a number of outputs (784) corresponding to the image dimensions. Additionally, we remove the last two fully-connected layers, since the last convolutional layer C3 provides 400 values, but we need 784 values, so instead of scaling down, this layer scales up. Lastly, we introduce the use of positional embeddings with an additional fully connected layer for each convolutional layer. The new design can be seen in figure 5 We now go into more detail with the

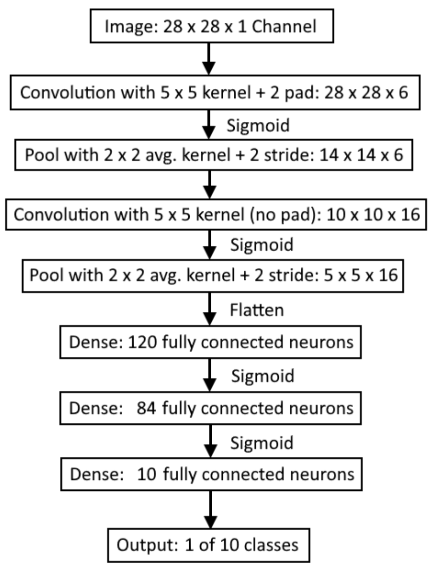
Image: 28 x 28 x 1 Channel
Convolution with 5 x 5 kernel + 2 pad: 28 x 28 x 6
Sigmoid
Pool with 2 x 2 avg. kernel + 2 stride: 14 x 14 x 6
Convolution with 5 x 5 kernel (no pad): 10 x 10 x 16
Sigmoid
Pool with 2 x 2 avg. kernel + 2 stride: 5 x 5 x 16
Flatten
Dense: 120 fully connected neurons
Sigmoid
Dense: 84 fully connected neurons
Sigmoid
Dense: 10 fully connected neurons
Output: 1 of 10 classes

Fig. 4: Overview of the LeNet-5 design

design and implementation of each layer. There are three different layers: convolutional layers, average pooling layers, and fully-connected layers. We also give a description of the activation functions used between each layer.

3.3.2 Convolutional Layer

Convolutional layers are the most significant part of a CNN. When running a neural network, the vast majority of runtime is spent on the convolutional layers [14].

The idea behind convolution layers is that the input data (typically an image) is "convolved" with a small matrix, called the kernel or filter. The convolution occurs by element-wise multiplication of the elements in the small matrix with the elements in the "visible" area of the input data. The products are then summed and placed in the output data, known as the feature map. Their placement typically corresponds to their position in the input data, but not necessarily, as the input data and feature map might not have the same dimensions. The convolution will generate a feature map for each kernel/filter used.

The math behind the convolution is not trivial.

Given an input image I of shape $H\_in \times H\_in$ and a kernel K of dimensions $k1 \times k2$ the cross-correlation operation for a pixel $(i.j)$ is given by:

$$(I * K)_{ij} = \sum_{a=0}^{k1-1} \sum_{b=0}^{k2-1} I(i+a, j+b) K(a, b) \tag{10}$$

Where the output is a feature map of shape $H\_out \times W\_out$.

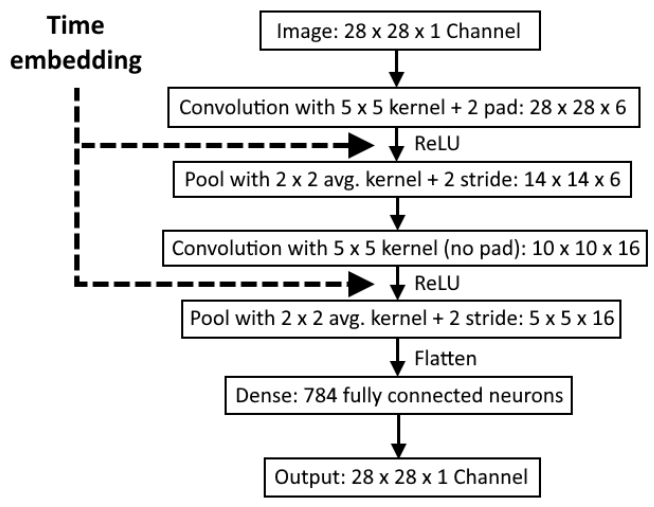
Time embedding
Image: 28 x 28 x 1 Channel
Convolution with 5 x 5 kernel + 2 pad: 28 x 28 x 6
ReLU
Pool with 2 x 2 avg. kernel + 2 stride: 14 x 14 x 6
Convolution with 5 x 5 kernel (no pad): 10 x 10 x 16
ReLU
Pool with 2 x 2 avg. kernel + 2 stride: 5 x 5 x 16
Flatten
Dense: 784 fully connected neurons
Output: 28 x 28 x 1 Channel

Fig. 5: Overview of the LeNet-5-Fut design

The output dimensions of the feature map is calculated as follows:

$$H\_out = (H\_in - k1 + 2 * P)/S + 1 \tag{11}$$
$$W\_out = (W\_in - k2 + 2 * P)/S + 1 \tag{12}$$
$$\tag{13}$$

Where P is the number of symmetrically applied padding in the form of zeroes around the image. We often pad the input image to result in an output of a size similar to the input image.

The convolutional operation is identical to the cross-correlation operation, the only exception is that the kernel in the convolution is flipped 180 degrees. However, whether we flip the kernel or not does not matter in the context of convolutional layers in neural networks. This is because the values of the kernel are learnable parameters, so the network can still learn to recognize the same set of features. If it is needed to flip the kernel to recognize a certain feature, the model just learn a flipped version of the kernel. We therefore ignore this step in our implemented convolution.

In the case where the input image has multiple channels, thus is of the form $l \times H\_in \times H\_in$. And we want another amount of output channels o, the kernel must have the size of $o \times l \times k1 \times k2$. We end with the following for the pixel $(i.j)$ in output channel o:

$$(I * K)_{oij} = \sum_{a=0}^{k1-1} \sum_{b=0}^{k2-1} \sum_{c=1}^{l} K_{o,c,a,b} \cdot I_{c,i+a,j+b} + b_o \tag{14}$$

As can be seen from the LeNet-5-Fut design in figure 5, the first convolutional layer changes dimensions from $1 \times 28 \times 28$ to $6 \times 28 \times 28$ as the number of channels increase. Note that if you provide an image as an input to a convolutional layer, the output is not necessarily structured as an image. The result of the layer is then feature maps numbered 1 to 6.

The ”naive” approach to implementing a convolutional layer is to implement the mathematical operations directly: for each image and kernel, we ”slide” them together and use the `reduce` operation. However, while straightforward, this is not a very efficient implementation. Numerous more efficient algorithms exists, of which the most notable ones are im2col (also known as GEMM), fast fourier transformations (FTT) and Winograd, each with distinctive pros and cons. For example, Winograd may have better performance on small batch sizes, while FTT scales better for larger batch sizes [14]. PyTorch uses cudNN, which implements all three, but selects a specific one depending on the size of the input.

Due to time limitations, we have only implemented two different convolution algorithms: the naive approach and im2col.

im2col

Instead of using the naive approach, im2col transforms the input and kernel into two matrices, which are then multiplied together. This results in significant speedup on GPU architecture, since matrix multiplication is highly parallelizable. Results can be found in section 4.

We will for the implementation of the convolutional layer focus on the im2col approach since this is the used version, the naive approach can be found in the appendix.

We implement the following helper functions for the convolution:

```
def add_padding [l][n][m] (imgs : [l][n][m]f64) (padding : i64) =
  let n_pad = (n+(padding*2))
  let m_pad = (m+(padding*2))
  in map (\ img_i -> tabulate_2d n_pad m_pad (\ i j ->
      if (i < padding || i >= (n+padding)
          || j < padding || j >= (m+padding))
      then 0
      else img_i[i-padding,j-padding])) imgs

def im2col [l][n][m] (img : [l][n][m]f64) (total : i64)
                     (kernel_size : i64) (new_n : i64)
                     (new_m : i64) =
  let k_total = kernel_size*kernel_size
  in transpose (
       flatten (
         tabulate_2d
           new_n
           new_m
           (\ y x ->
             flatten (
               map
                 (\ i ->
                   flatten (i[y:y+kernel_size,
                              x:x+kernel_size])
                           :> [k_total]f64)
                 img
             ) :> [total]f64
           )))
```

The add_padding function is used to symmetrically apply padding in the form of zeroes around the innermost two dimensions of an image. The padding parameter decides the width of the zero padding applied to each side of the image.

im2col makes the transformation of the 3d input into the respective 2d im2col matrix. The function essentially maps over the images and flatterns each patch of size kernel_size x kernel_size that the kernel otherwise would have multiplied and summed together. This is again flattened to combine each patch in a single row before transposing it. The end result is a 2D array where each column corresponds to a flattened patch from all channels of the original image.

The following is the implemented convolutional layer:

```
def convolve2D [n][m][p][k][l][o] (imgs : [l][n][m]f64)
                                  (kernels : [o][l][p][k]f64)
                                  (biases : [o]f64)
                                  (padding : i64) =
  let new_n = (((n+(padding*2))-p)+1)
  let new_m = (((m+(padding*2))-p)+1)
  let total = l*p*k

  let imgs_padded =
    if (padding != 0) then
      add_padding imgs padding
    else
      imgs

  let img_col = im2col imgs_padded total p new_n new_m
  let kernel_col = map (\ x -> flatten_3d x :> [total]f64) kernels
  let res = matmul kernel_col img_col
  let res_bias = map2 (\ r b -> map (+b) r) res biases
  in map (unflatten new_n new_m) res_bias
```

We initially calculate the dimensions of the output image: new_n and new_m.

If needed, we add padding to the input images with add_padding before computing the 2d im2col matrix using im2col.

The kernels are also reshaped into a 2d array using map flatten_3d.

We now make the convolution operation with the use of matrix multiplication, before adding the respective biases.

Lastly, the image is reshaped into its output dimensions.

We make the following backwards implementation of the convolutional layer:

```
def convolve2D_b [n][m][p][k][l][o][q][r]
    (out_grad : [o][q][r]f64) (conv_input : [l][n][m]f64)
    (kernels : [o][l][p][k]f64) (valid_num : i64)
```

```
  (full_num : i64) : ([l][n][m]f64, [o][l][p][k]f64,[o]f64)  =
let kernels_grad =
  tabulate_2d o l (\ i j ->
    flatten (convolve2D [conv_input[j]] [[out_grad[i]]]
                        [0f64] valid_num)
             :> [p][k]f64)
let input_grad =
  convolve2D out_grad (transpose kernels[:,:,::-1,::-1])
              (replicate l 0f64) full_num :> [l][n][m]f64
let biases_grad = map (\ x -> reduce (+) 0 (flatten x)) out_grad
in (input_grad, kernels_grad, biases_grad)
```

We compute the gradients of the kernels using convolutions between each pair of original inputs and the backpropagated gradients.

The input gradients is found with another convolution between the backpropagated gradients and transpoed kernels.

The bias gradients are found by summing together the backpropagated gradients for each channel.

3.3.3 Pooling Layer

A pooling layer uses a window similarly to the convolutional layer, but simply uses the window to reduce the dimensions of the input feature map. For example, using a 2x2 pooling window with stride 2 and average pooling will ”return” the average value of all four data points as a single data point, effectively halving the feature map.

The backward pass of the average pooling layer essentially performs an unpooling operation, scaling the gradient feature map to match the dimensions of the original input to the layer. E.g a stride of 2 will double the dimensions of the feature map. For each pixel in the gradient feature map, its value is distributed equally across the corresponding window in the original input.

We implement the pooling layer the following way:

```
def avg_pool [l][n][m] (imgs : [l][n][m]f64) (stride : i64) =
  let out_y = n/stride
  let out_x = m/stride
  let area_size = f64.i64 (stride*stride)
  in map (\img ->
    tabulate_2d out_y out_x (\ y x ->
      let area =
        img[(y*stride):(y*stride+stride),
            (x*stride):(x*stride+stride)]
      in reduce (+) 0 (flatten area) / area_size)
        :> [out_y][out_x]f64) imgs
```

`out_y` and `out_x` compute the dimensions of the feature map after pooling.

We then slide the window of dimensions `stride`×`stride` over the image using `tabulate_2d` with the new dimensions, computing the average of the values in the window.

We implement the following backwards average pool function:

```
def avg_pool_b [l][n][m] (imgs : [l][n][m]f64) (stride : i64) =
  let out_y = n*stride
  let out_x = m*stride
  let area_size = f64.i64 (stride*stride)
  in map (\img ->
    tabulate_2d out_y out_x (\ y x ->
      img[y/stride, x/stride] / area_size
    )
  ) imgs
```

Here `out_y` and `out_x` compute the dimensions of the input feature map before pooling.

We use `tabulate_2d` to distribute the gradient out to its respective pixels.

3.3.4 Fully-connected Layer

As the name implies, a fully-connected layer (also known as a dense layer) is fully connected to the preceding layer, i.e. every node in the preceding layer is connected to every node in this layer. The output features y of a fully-connected layer are computed using the following formula:

$$y = x * W + b \tag{15}$$

Where x is the input, W is the matrix of weights corresponding to the connections between the neurons of the preceding layer and the current layer, and b is the vector of biases associated with each neuron in the current layer.

For the backwards fully connected layer, let $\frac{\partial L}{\partial y}$ be the gradient of the loss function with respect to the output layer. For the weights we see that:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial W} \tag{16}$$

Since $\frac{\partial y}{\partial W} = x$ we arrive at the following computation for the gradients of the weights:

$$\frac{\partial L}{\partial y} \cdot x \tag{17}$$

We also see that $\frac{\partial y}{\partial b} = 1$. Thus, the gradients of the biases becomes:

$$\frac{\partial L}{\partial y} \tag{18}$$

Lastly, we see for the inputs that $\frac{\partial y}{\partial x} = W$, giving us the following math for computing the input gradients:

$$W^T \cdot \frac{\partial L}{\partial y} \tag{19}$$

We make the following implementation of the fully connected layer, we have named the layer ”dense” in our implementation:

```
def dense [m][n] (input : [n]f64) (weights : [m][n]f64)
                 (biases : [m]f64) : [m]f64 =
    map2 (\ w b -> (dotprod input w) + b) weights biases

def dense_b [m][n] (out_grad : [m]f64) (dense_input : [n]f64)
                   (weights : [m][n]f64)
                   : ([n]f64, [m][n]f64, [m]f64) =
  let ws_g :[m][n]f64 =
    map (\ x -> map (\ y -> y*x) dense_input) out_grad
  let input_g = map (\ w -> dotprod w out_grad) (transpose weights)
  in (input_g, ws_g, out_grad)
```

3.3.5 Activation Functions

Activation functions are functions that modify the output of a single node. Some examples include:

$$\begin{aligned} \text{Binary step}: \quad & f(x) = \begin{cases} 0 & \text{if} x < 0 \\ 1 & \text{if} x \leq 0 \end{cases} \\ \text{ReLU}: \quad & f(x) = max(0, x) \\ \text{Sigmoid}: \quad & f(x) = \frac{1}{1 + \mathrm{e}^{-x}} \end{aligned}$$

For our f, we use the *Rectified Linear Unit* (ReLU) function. There are different types of activation functions, but we have chosen to only implement the ReLU function, since it is computationally simpler, and do not suffer from the *vanishing gradient problem*[30]. Activation functions like sigmoid suffer from a problem where for very large positive or negative values, the gradients tend to be very small. We would like to avoid small gradients since it drastically slow down the learning process. Using ReLU mitigates this problem as its gradient is always 1 for positive input. However ReLU can introduce another problem, namely the *dying ReLU problem*[19] where neurons stop outputting anything other than 0.

The derivative of the ReLU function is defined as:

$$f'(x) = 1 \text{ if } x > 0 \tag{20}$$

$$f'(x) = 0 \text{ if } x <= 0 \tag{21}$$

Meaning that if the input to the ReLU was positive, the gradient is just propagated back, but if the input was negative, the gradient is set to 0.
The following shows the implementation of the ReLU layer:

```
def ReLU (x : f64) : f64 =
  f64.max 0f64 x
```

We use the futhark `max` function.

For the backwards ReLU we have:

```
def ReLU_b (out_grad : f64) (input : f64) : f64 =
  if input > 0f64 then out_grad else 0f64
```

Following the derivative.

3.3.6 U-Net

Our initial approach reveals that LeNet-5 has several shortcomings in regards to implementing neural networks for image analysis. The main issue, we believe, stems from its limited number of output values. We were not able to reach satisfactory levels of loss after training with LeNet-5. The loss never seemed to decrease no matter how much training data and how many epochs were used. We hypothesize that the LeNet-5 design is simply too limited for what we are trying to achieve. There are not enough parameters to express what we want to predict, and any amount of training would not yield satisfactory results. We further theorize that this limitation may be caused by the lack of *skip connections* in LeNet-5, which makes it harder to preserve information from the early layers in the network.

For this reason, we have opted to choose a different neural network design, namely U-Net [23]. This network type sees great use in the field of biomedical image segmentation, and is even the same network type used by the authors of the DDPM paper in their implementation [10].

Now, since our implementation did not work with LeNet-5, our approach has changed slightly. Our main goal is to demonstrate a proof-of-concept implementation of a diffusion model in Futhark, not to necessarily implement one that surpasses existing ones in speed or accuracy. For that reason, we focus on the question: what is the simplest possible u-net architecture that we can implement and still achieve sufficient quality?

Our U-Net architecture requires the following new elements:

- Our u-net is not directly composed of layers, but is instead composed of *blocks*, where each block has 2 convolutions. This makes it possible to easily include more time embeddings.
- Skip connections, which, as mentioned previously, allows the network to carry over information from earlier layers. This trait is important for image generation and restoration tasks.
- Additional parameters (a wider network with more feature maps).

Furthermore, we reduce the number of feature maps. The original u-net scaled to 1024 feature maps, but our version only goes to 256. Finally, our U-net does not exhibit the ”U”-shape, since its feature maps remain the same size in all layers.

The blocks used to build our U-Net includes the following elements:

```
def block [l][n][m][t][o] (imgs: [l][n][m]f64) (time_mlp : [t]f64)
                          (num_groups : i64) (weights) =
  let (conv1_w,conv1_b,t_w,t_b,conv2_w,conv2_b) = weights
  let conv1 : [o][n][m]f64 =
    convolve2D imgs conv1_w conv1_b 1 :> [o][n][m]f64
  let conv1_act : [o][n][m]f64 = ReLU_3d conv1
  let (conv1_gnorm,conv1_gnorm_cache) =
    group_norm conv1_act num_groups 1e-05
  let lin_out : [o]f64 = dense time_mlp t_w t_b
  let lin_out_act : [o]f64 = map ReLU lin_out
  let comb : [o][n][m]f64 =
    (map2 (\ conv lin -> map (\ c -> map (\c_ -> c_+lin) c) conv)
          conv1_gnorm lin_out_act)
  let conv2 : [o][n][m]f64 =
    convolve2D comb conv2_w conv2_b 1 :> [o][n][m]f64
  let conv2_act : [o][n][m]f64 = ReLU_3d conv2
  let (conv2_gnorm,conv2_gnorm_cache) =
    group_norm conv2_act num_groups 1e-05
  in (conv2_gnorm,
      (imgs,time_mlp,conv1,lin_out,comb,conv2,
       conv1_gnorm_cache,conv2_gnorm_cache))
```

Each block starts with a convolution, going from the input channels to the output channels. This is followed by a group normalization. We then add the positional embeddings to the feature maps, and make another convolution from output channels to output channels. Lastly, we make another group normalization.

The backwards block can be seen in the appendix.

We have throughout the project repeatedly changed the U-Net structure used in the diffusion model. This includes changing the type, number, and size of layers. For each generation of the U-Net, we used visual inspection of the generated images until we reached a satisfactory result. The U-Net used in the final generation of the diffusion model has the following structure:

```
Input image: 1 (channel) x 28 (height) x 28 (width)
Initial convolution: 3x3 kernel + 1 padding: 64 x 28 x 28
Down_B1:
    - B(3x3 kernel + 1 padding: 128 x 28 x 28)
    - Pos_embedding(256, 128)
Down_B2:
    - B(3x3 kernel + 1 padding: 256 x 28 x 28)
    - Pos_embedding(256, 256)
Skip connection: Cat(Down_B2, Down_B2: 512 x 28 x 28)
Up_B1:
    - B(3x3 kernel + 1 padding: 128 x 28 x 28)
    - Pos_embedding(256, 128)
Skip connection: Cat(Up_B1, Down_B1: 256 x 28 x 28)
Up_B2:
    - B(3x3 kernel + 1 padding: 64 x 28 x 28)
    - Pos_embedding(256, 64)
Output convolution: 1x1 kernel + 0 padding: 1 x 28 x 28
```

We here see how the positional embeddings are used in the model. The idea behind it is simple, we concatenate the output of a previous down blok, to a up block if the dimensions fit. As an example, the first upsampling block (Up_B2) will have 512 input feature maps.

3.3.7 Group Normalization

We apply a technique known as group normalization [34]. This technique divides the feature maps into groups, and then normalizes the values in each group. This process stabilizes the learning, and reduces the total number of training steps needed to achieve desirable results. Another similar technique is the batch normalization, which operates over the batch dimension. However, we are using a batchsize of 1, therefore the batchnorm, has less data to calculate the mean and variance, leading to more noise in the estimates. The math behind the group normalization is as follows:

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} \tag{22}$$

Where:
x is the input feature map,
$E[x]$ is the mean of the data in a group
$Var[x]$ is the variance of the data in a group
ϵ is a small constant for numerical stability to avoid division by zero.

Since group normalization is the most recent addition to the network, we have not yet derived the math due to time limitations. However, since batch normalization is very closely related to group normalization, we can use this[6] derivation as a reference.

The following shows the helper functions for implementing the grouplayer:

```
def group_arr [l][m][n] (img : [l][m][n]f64) (num_groups : i64)
                        (elem_groups : i64)
                        : [num_groups][elem_groups][m][n]f64 =
  tabulate num_groups
           (\ x ->
             img[(x*elem_groups):((x+1)*elem_groups)]
             :> [elem_groups][m][n]f64)

def mean [n] (img : [n]f64) : f64 =
  f64.sum img / f64.i64 n

def variance [n] (vs: [n]f64) =
  let m = mean vs
  let xs = map (\x -> (x-m)*(x-m)) vs
  in f64.sum xs / (f64.i64 n)

def mean_and_var [l][n][m] (img : [l][n][m]f64) : (f64,f64) =
  let flat_img = flatten_3d img
  let img_mean = mean flat_img
  let img_var = variance flat_img
  in (img_mean, img_var)
```

`group_arr` groups the input feature map, making a 4D array.
`mean` calculates the mean
`variance` calculates the variance with the use of `mean`
`mean_and_var` combines the two, notice that it takes a 3d input, making it

[6] https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

applicable for a map over the grouped 4d array.

The following shows the implementation of the group normalization:

```
def group_norm [l][m][n] (img : [l][m][n]f64) (num_groups : i64)
                         (eps : f64)  =
  let elem_groups = l / num_groups
  let group_img = group_arr img num_groups elem_groups
  let (mean_grouped,var_grouped) =
    unzip (map mean_and_var group_img)
  let out =
    tabulate_3d l m n (\ z y x ->
                        (img[z,y,x] - mean_grouped[z/elem_groups])
                        / f64.sqrt (var_grouped[z/elem_groups] +eps))
  in (out,(out,var_grouped,elem_groups))
```

The implementation follows the math pretty nicely. We calculate the number of elements in each group and groups them using `group_arr`.

Due to the implementation of `mean_and_var` it is possible to calculate the means and variances of the groups in parallel using `map`. The values are however returned as tuples, requiring the use of an `unzip`.

Lastly, the normalization is computed with a `tabulate_3d`, following the math.

The following shows the implementation of the backwards groupnorm:

```
def group_norm_b [l][m][n] (out_grad : [l][m][n]f64)
                           (num_groups : i64) (eps : f64)
                           (cache) : [l][m][n]f64 =
  let (out, var, elem_groups) = cache
  let elem_in_group = elem_groups * m * n
  let group_out = group_arr out num_groups elem_groups
  let group_grad = group_arr out_grad num_groups elem_groups
  let elem_in_group_f64 = f64.i64 elem_in_group

  let grad_sum_and_mult_sum =
    map2 (\x y ->
      let flat_x = flatten_3d x :> [elem_in_group]f64
      let flat_y = flatten_3d y :> [elem_in_group]f64
      in (reduce (+) 0 flat_x,
          reduce (+) 0 (map2 (*) flat_x flat_y)))
    group_grad group_out

  let res =
    tabulate num_groups (\d ->
      let (grad_sum, grad_out_mult_sum) = grad_sum_and_mult_sum[d]
      in tabulate_3d elem_groups m n (\ z y x ->
        (1f64 / elem_in_group_f64) *
        (elem_in_group_f64 * group_grad[d,z,y,x]
         - grad_sum
         - group_out[d,z,y,x] * grad_out_mult_sum) /
        f64.sqrt(var[d] + eps)))

  in flatten res :> [l][m][n]f64
```

We present the results of using group normalization in section 4.

3.3.8 Weights Initialization

Weight initialization is a important component in ensuring efficient training of neural networks. The choice of how to set the initial weights of a network can significantly impact both the speed of convergence during training and even the ultimate performance of the network.

Initially, we chose our weights from a normal distribution. However, we quickly realised when comparing losses with the python implementation that it converged towards zero much quicker.

Instead, we used the weight initialization utilized by PyTorch and saw much better results. The code used for initialisation is seen below:

```
def mk_dense_weights (seed : i32) (m : i64) (n : i64) : [m][n]f64 =
  let size = 1f64/(f64.i64 n)
  let rng_state = rng_engine.rng_from_seed [seed]
  let rng_states = rng_engine.split_rng (m*n) rng_state
  let rng_numb =
    map (\e -> (rand_f64_uniform.rand ((-f64.sqrt(size)),
                                       (f64.sqrt(size))) e).1)
        rng_states
  in unflatten m n rng_numb

def mk_conv_weights  (seed : i32) (o : i64) (l : i64)
                     (p : i64) (k : i64) : [o][l][p][k]f64 =
  let size = 1f64/( f64.i64 (l*(p*k)))
  let rng_state = rng_engine.rng_from_seed [seed]
  let rng_states = rng_engine.split_rng (o*l*p*k) rng_state
  let rng_numb =
    map (\e -> (rand_f64_uniform.rand ((-f64.sqrt(size)),
                                       (f64.sqrt(size))) e).1)
        rng_states
  in unflatten_4d o l p k rng_numb
```

This code uses the ccprandom package for Futhark[7].

3.3.9 Training of the model

To train the model we use the training algorithm from DDPM. This gives us the following code:

```
  let t = mk_rand_int time_seeds[e,i] 1 999
  let (x_t, noise) = q_sample images[i] t alpha_bar noise_seeds[e,i]
  let time_embedding =
    sinusoidal_position_embeddings 256 (f64.i64 t) :> [256]f64
  let (predicted_noise, cache) =
    unet_simple x_t time_embedding num_groups w_images
  let loss = mse_loss_img noise predicted_noise
  let gradients =
    unet_simple_reverse x_t predicted_noise noise w_images cache
```

We use the Adam optimizer together with the gradients to calculate the new weights.

[7] https://github.com/diku-dk/cpprandom

4 Benchmarks and validation

To evaluate the performance of our model, and the quality of the generated images, we implement a series of benchmarks and validations. Since standalone benchmarks of performance often are uninteresting, we have also implemented a diffusion model in Python using PyTorch which is a machine learning framework based on the Torch library[13]. Doing this makes it possible to investigate how well the Futhark implementations compare to the PyTorch functions. To benchmark Futhark we use `futhark bench -backend=cuda <test_filename>.fut`, for benchmarking PyTorch, we use the `time` library. We categorize our benchmarks and validations into two sections.

Since the architecture of the diffusion model consists of multiple layers, the initial section consists of benchmarking each layer individually to give us an insight into how each part of the model correlates with their corresponding implementations in PyTorch. Doing this helps us identify the most important areas to look into when optimizing the runtime of the model.

The second section takes a more broader look at the overall performance of the diffusion model. We look at the performance of the training loop, and we also examine the use of Futhark-ad, and how this affects the performance. Furthermore, we investigate the validity of the generated images using visual inspection, a CCN classifier, and pixel intensity distributions. This analysis gives us an understanding of the model's capabilities and potential areas for improvement.

4.1 Experimental setup

The benchmarks and validations have been tested on the futhark server with an Nvidia A100 GPU.

4.2 Datasets

The structure of the U-Net model used in the experiments is notably influenced by the used dataset. In our experiments, we are using the MNIST dataset from `keras.datasets`. This dataset comprises 60,000 grayscale images of handwritten digits, each image having the shape of 28x28 pixels.[8]. Since the depth and complexity of the images are simple, the structure of the U-Net reflects this.

4.3 Isolated benchmarks

The subsequent sections use detailed performance benchmarks to provide an in-depth performance analysis of the key layers utilized in the U-Net architecture.

Examination of the applied U-Net, reveals that the overall performance is largely dependent on the convolutional and group normalization layers. Each block of the U-Net incorporates two convolutions, resulting in a total of eight convolutions from the four blocks. Similarly, eight group normalizations are to

[8] https://keras.io/api/datasets/mnist/

find within these blocks. Additionally, we have the initial and output convolutions, giving us in total, ten convolutions and eight group normalizations for each U-Net forward pass.

Meanwhile, our U-Net only uses four dense layers, one for each block to handle the time embedding - which in our context is consistently set at 256. Given their limited presence, optimizing these layers would likely have a relatively minor impact on the total runtime of the U-Net.

4.3.1 2D-convolution

In this section, we look at the most important layer of the U-net, namely the convolutional layer. We present a comparative analysis, benchmarking the performance of our implemented convolutional layer against the well-established PyTorch's convolutional layer. The benchmarks can be seen in figure 6. It is

Input / Library	In: 1 Out: 64 Img shape: 28x28 Kernel size: 3	In: 512 Out: 128 Img shape: 28x28 Kernel size: 3	In: 1024 Out: 256 Img shape: 56x56 Kernel size: 3
PyTorch	62 μs	74 μs	254 μs
Futhark	64 μs	4042 μs	12028 μs

Fig. 6: Convolutional layer benchmarks

clear from the performance results, that our implementation of the convolutional layer quickly is outperformed by the PyTorch implementation, being almost 50 times slower at our largest benchmark.

This suggests that our implementation has difficulty scaling with large input data and numerous input and output channels, which is very important to take into account when considering the performance of using Futhark for neural network computations.

The performance gap can due to various factors. Firstly, we know from the design section of the 2D-convolutional layer that since PyTorch has the possibility to choose from different convolution algorithms, superior performance from our own implementation wasn't anticipated. PyTorch has benefited from extensive optimization over its seven-year history, resulting in a mature and highly efficient codebase.

Lastly, even though Futhark makes it easy to design and implement algorithms, it is difficult to match the performance of lower-level, hardware-specific optimizations present in libraries like PyTorch. Further investigation is therefore required to improve the performance of our convolutional layer in Futhark. The same conclusion can be drawn from the backpropagation of the convolutional layer, as seen in figure 7. Since the gradients of the input and weights themselves are computed using convolutions, we see a close correlation between the runtime performances. An interesting property to note is how the runtime

Input / Library	In: 1 Out: 64 Img shape: 28x28 Kernel size: 3	In: 512 Out: 128 Img shape: 28x28 Kernel size: 3	In: 1024 Out: 256 Img shape: 56x56 Kernel size: 3
PyTorch	167 μs	178 μs	187 μs
Futhark	276 μs	18025 μs	333043 μs

Fig. 7: Convolutional layer benchmarks (backpropagation)

grows exponentially for the backpropagation. This is because we make a convolution for each combination of gradients and input images.

Analysis of the importance of input dimensions
In this section, we discuss how varying input dimensions affects the execution time of the 2d-convolution operation. Our analysis is based on changing one of the four parameters *input channels, output channels, spatial feature map size* and *kernel size* while keeping the other three constant.

The table in figure 8 shows the effects of altering the kernel size. Here, we

Input / Library	In: 1 Out: 64 Img shape: 64x64 Kernel size: 3x3	In: 1 Out: 64 Img shape: 64x64 Kernel size: 7x7
Futhark	6796 μs	26588 μs

Fig. 8: Effect of altering kernel size

see a significant difference between the kernel size and the computation time. Looking at the implementation of `conv2d`, we see that the size of the kernel directly influences the height of the columns in the transformed *img_col* and *kernel_col* matrices. Thus, increasing the complexity of each `dotprod` operation. At the same time, when the kernel size increases, the width of `img_row` decreases, due to fewer unique "areas" available from the input image. However, this trade-off proves inefficient when it comes to parallel computation on a GPU.

Given that the core operation of the convolution lies in the matrix multiplication, we investigate this further and make the following simplified example:

Let us, for the sake of the argument, assume that the GPU has an unlimited amount of cores. When we map the `dotprod` operation over the column array, we create a set of independent operations, which can be performed in parallel by seperate GPU threads. However, the GPU architecture does not split individual operations across multiple threads. Therefore, the time it takes to execute each operation remains constant, regardless of the number of available GPU cores. Conversely, with shorter columns and larger rows, the computation time could, in theory, lead to an almost instantaneous matrix multiplication, given enough GPU cores.

The trade-off introduced by the kernel size is thus between the number of computations, which increases with kernel size, and their parallelizability, which decreases. This can clearly be seen in the figure 9 benchmark of the Futhark `matmul` function. Another important aspect is the kernel sizes effect on memory

Input / Library	xss: 5000x10 yss: 10x5000	xss: 10x5000 yss: 5000x10
Futhark	479 μs	26 μs

Fig. 9: `matmul` with different parameters

usage. With a larger kernel size, each column in the im2col matrix encompasses a broader area of the original input. This means more data from the original input is being duplicated to the im2col output, resulting in larger columns for the same input if the kernel size is larger.

Using the inputs from previous benchmark in TABLECITE, we see the following difference in the number of elements in the im2col matrix:

```
In: 1, Out: 64, Img shape: 64x64, Kernel size: 3x3 = 36864 elements
In: 1, Out: 64, Img shape: 64x64, Kernel size: 7x7 = 176400 elements
```

Which is around a 4.785 times increase in the number of elements.

Having larger columns also means that more data has to be accessed and stored in memory for the subsequent matrix multiplications, resulting in increasing memory bandwidth requirements. Increased memory usage could lead to cache eviction if the data exceeds the capacity of the GPU's cache, which could aggravate the runtime performance even further.

However, this approach also enables the use of matrix multiplication for the convolution operation, which is highly optimized in Futhark, making it up for the overhead introduced by the `im2col` transformation. Figure 10 shows the differences between the naive conv2d approach and the im2col variant. While it might seem from this section that larger kernels are disadvantageous from a computational perspective, it's crucial to consider the broader context. These benchmarks focus on the runtime performance of the convolution operation, but it's not the only factor in the overall effectiveness of a neural network. Larger kernel sizes result in more weights that the network can optimize, which can lead to improved performance.

Input / Library	In: 50 Out: 100 Img shape: 50x50 Kernel size: 3	In: 200 Out: 400 Img shape: 50x50 Kernel size: 3	In: 500 Out: 1000 Img shape: 100x100 Kernel size: 3
Futhark: Naive conv2d	2105 μs	34608 μs	985462 μs
Futhark: im2col conv2d	405 μs	3088 μs	48610 μs

Fig. 10: Differences between naive conv2d and im2col variant.

Peng et al. [21] propose a neural network that achieved impressive results in semantic segmentation by utilizing larger kernel sizes. Moreover, Ding et al. [6] note that the use of large kernel CNNs tends to capture context more effectively and pay greater attention to image structures rather than textures.

Importance of spatial image sizes
Figure 11 shows the importance of the image sizes with respect to runtime performance. As expected, increasing the image size coincides with a proportional

Input Image size	In: 1 Out: 1 Kernel size: 3x3
64x64	50 μs
512x512	405 μs
1024x1024	1600 μs

Fig. 11: Image size effect on runtime

increase in runtime. This increase is directly connected to the larger amount of data being processed, leading to more computations. More interesting, however, is the scale of runtime increase. Between the benchmarks of shapes $64x64$ and $512x512$, we have an 8 times increase of data needed to be processed. The factor between the runtimes for these benchmarks is also roughly 8. This proportional scaling fits the expectation that the convolution time scales linearly with the increase of image size.

However, when we scale the image further, from $512x512$ to $1024x1024$, equating to a twofold increase in each dimension, the runtime inflates by nearly four times. This discrepancy does not fit the linear scaling, and could potentially be due to the effect of the memory hierarchy. As the image size increases, it becomes more likely that the data will not fit in the GPU's speedy cache memory, namely L1 and L2. As a result, the likelihood of cache misses increases, making the GPU need to access the slower global memory, which has a higher latency.

In our experiments, we are using the Nvidia A100 GPU which has an L2 cache size of 40MB and a configurable L1/shared memory with a maximum allocation of 164KB per streaming multiprocessor (SM)[9]. Looking at the input image of size $64x64$ with float64 data, it necessitates approximately 32KB of memory, which likely fits into the size of the L1 cache. Conversely, as the image size increases, so does the required memory: the 512x512 array requires around 2MB, and the 1024x1024 requires 8MB. With respect to other data the GPU concurrently processes, this could very well be one of the reasons for the non-linear scaling of runtimes.

Another potential factor that could contribute to the observed non-linear runtimes is the influence of the GPU's *block size* and *occupancy*.[10]. The block

[9] https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[10] https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm

Input / Library	In: 1 Out: 1024 Img shape: 1x1 Kernel size: 3x3	In: 1024 Out: 1 Img shape: 1x1 Kernel size: 3x3
Futhark	43 μs	383 μs

Fig. 12: Effects of varying input and output channels

size, corresponds to the number threads concurrently executing in a block, and the occupancy refers to the ratio of active warps on an SM to the maximum number of active warps supported by the SM.

As the data increases, it might lead to larger block sizes that can be less efficiently scheduled. This results in lower multiprocessor occupancy and slower performance [8]. Additionally, higher memory usage could lead to more contention for shared GPU resources [24].

These potential factors are highly influenced by many elements and need a deeper investigation of the Futhark compiler. Although a detailed analysis of these elements and factors is beyond the scope of this paper, they are worth noting as possible sources contributing to the non-linear performance trends observed.

Importance of input channels and output channels
In order to investigate the effect of varying input and output channels, we have conducted the following benchmarks as shown in figure 12. The benchmarks indicate a significant difference in runtime when we vary the number of input and output channels. This behavior, akin to the importance of kernel size, can be traced back to the dimensions of the matrix multiplication. In the case where we only have a single input value, we effectively perform 1024 `dotprod` operations, each with a single element. Conversely, when combining 1024 values into a single output, we carry out just one `dotprod` operation, but with 1024 elements. Since this larger operation cannot be distributed across multiple GPU threads, the runtime increases.

In addition, the number of input channels has a direct and substantial impact on memory usage. Since the size of the im2col transformed matrix expands with the number of input channels, an increase in input channels can result in a very inflated intermediate representation. For instance, consider an input image of dimensions 64x64x64 and a kernel of dimensions 64x64x3x3. Upon padding the input with one element, the input and output retain an equal number of elements: $64 \cdot 64 \cdot 64 = 262144$. However, the transformed matrix expands into $64 \cdot 3 \cdot 3 \cdot 64 \cdot 64 = 2359296$ elements, which is 9 times larger than both the input and output. In extreme cases, this could exhaust the GPU memory resources, causing a crash, even though the output of the convolution would have otherwise fit comfortably into memory.

Summary
We have in this section investigated how the sizes of the input to the convolution

Library \ Input	In: 64 Img shape: 28x28 num_groups: 32	In: 256 Img shape: 28x28 num_groups: 32	In: 512 Img shape: 56x56 num_groups: 32
PyTorch	43 μs	44 μs	46 μs
Futhark	91 μs	102 μs	159 μs

Fig. 13: Group norm benchmarks

affect the runtime performance. We summarise by listing some key takeaways:

- **Kernel size.** The kernel size has a considerable impact on the performance of the convolution operation. Larger kernels result in more computationally intensive matrix multiplications and increase memory usage. Also, the larger the kernel, the larger the dimensions of the matrix multiplication, leading to less efficient usage of the GPU's parallel processing capabilities. However, multiple sources describe the usefulness of larger kernels in computer vision fields such as semantic segmentation.

- **Spatial Image Sizes.** The dimensions of the input image also play a significant role in the performance of convolution. Larger images result in larger intermediate data structures, leading to more memory usage and potential slowdowns due to increased global memory accesses.

- **Input Channels and Output Channels.** The number of input and output channels directly influence both the runtime and the memory usage of the convolution operation. More input channels result in larger intermediate representations and higher memory usage. Futhermore, like the kernel size increase, the large dot product operation involved when many input channels are combined into a single output channel cannot be distributed across multiple GPU threads, thereby increasing the runtime.

These key points provide valuable insights into the factors influencing the performance of the implemented 2d-convolution. It is essential to keep these factors in mind when designing and optimizing the structure of the desired network.

4.3.2 GroupNorm

In this section, we take a look at how our group norm implementation compares to the PyTorch implementation. The benchmarks can be seen in figure 13. The benchmarks reveal that our Futhark implementation performs reasonably well, with a runtime slightly over twice that of PyTorch for our use cases in the U-Net. Considering PyTorch's highly optimized codebase, this is an encouraging result. However, we also see that as the input size increases, the performance gap between Futhark and PyTorch grows as well, suggesting room for further optimizations in our Futhark implementation.

In conclusion, while our Futhark implementation does not surpass PyTorch in terms of speed, it has demonstrated the potential for efficient computation, which further motivates the exploration of performance optimization in Futhark for neural network computations. Evaluating the performance of our backward group normalization function (benchmarks in figure 14), we see that Futhark

Library \ Input	In: 64 Img shape: 28x28 num_groups: 32	In: 256 Img shape: 28x28 num_groups: 32	In: 512 Img shape: 56x56 num_groups: 32
PyTorch	145 μs	143 μs	153 μs
Futhark	62 μs	92 μs	154 μs

Fig. 14: Backward group normalization

Library \ Input	In: 256 Out: 64	In: 256 Out: 256	In: 1000 Out: 5000
PyTorch	47 μs	41 μs	41 μs
Futhark	21 μs	24 μs	94 μs

Fig. 15: Dense layer implementation

surpasses PyTorch in computational efficiency for the specific cases used in our U-net architecture, which is a great achievement. The implementation only falls behind by a slim margin at the largest input. This is especially impressive considering the complexity of the backward group normalization operation and indicates that our Futhark implementation maintains performance even when the input increases.

A key factor contributing to Futhark's great performance is our decision to avoid recomputing the forward group norm operation, since it's already available in the cache from the previous computation, thus reducing the computational overhead.

4.3.3 Dense

The following section presents how our dense layer implementation compares to the PyTorch implementation. The benchmarks can be seen in figure 15. From the forward performance results we see that our implementation consistently outperforms PyTorch for the small input sizes used in our U-net model. This demonstrates that the dense layer, though relatively simple, is implemented efficiently. However, we do see scaling difficulties with significantly large input data, similar to what we observed with the convolutional layer. However, since the dense layer is relatively straightforward computationally, it does not offer much room for optimization from an algorithmic point of view. Therefore, any improvements would likely come from further hardware-specific optimization or more efficient memory access patterns

We here reiterate that the goal is not to match or exceed PyTorch in every case but to provide an efficient alternative that is more amenable to mathematical reasoning and formal verification, which are strengths of the Futhark language. The same tendency can be seen in the benchmarks (figure 16) for the backward pass of the dense layer. Futhark outperforms PyTorch for smaller input sizes, with performance evening out as the input size grows. We clearly see that Futhark's efficient array computations play a key role, but as computations scale, factors such as memory bandwidth and parallelism efficiency also

Input / Library	In: 256 Out: 64	In: 256 Out: 256	In: 1000 Out: 5000
PyTorch	153 μs	146 μs	139 μs
Futhark	29 μs	31 μs	170 μs

Fig. 16: Dense layer backwards pass

come into play.

4.4 Summary

We have in the above benchmark sections compared the performance of the three main layers in the U-Net — convolution, group normalization and dense layer. The main focus has been on the runtime comparison between Futhark and PyTorch, a well-regarded deep learning library.

We found that Futhark shows superior efficiency in 2 out of 3 layers when comparing the use cases in the U-Net which are quite small computations. This showcases the strength of Futhark when handling array computations and taking advantage of the parallelism offered by GPUs. For larger inputs, however, the performance between the two libraries tends to equalize, with PyTorch taking a considerable lead in the convolutional layer, being around 48 times faster for the largest benchmarked input. This suggests that as computations scale, other factors such as memory bandwidth and parallelism efficiency become significant factors.

Even though the contribution of time to the entire runtime of the U-Net is negligible for the dense and group norm layers, it is still worth noting that Futhark performs remarkably well for the backwards pass.

In summary, our Futhark implementations offer a compelling alternative for executing deep learning layers, offering notable advantages particularly in scenarios with smaller inputs. This opens up interesting prospects for the use of Futhark in specific machine learning contexts where its strengths can be fully leveraged.

4.5 Combined tests

This second section benchmarks the overall performance of the diffusion model, including its components such as the training loop and the image sampling. We also examine how the use of Futhark-ad affects the performance of the network, and finally, we assess the quality of the generated images by a number of different parameters: visual inspection, a CCN classifier, and pixel intensity distribution.

For the benchmarks, we look into how quickly each iteration of the training loop executes, the time it takes to generate an image, the use of Futhark-ad, and the time it takes to train and generate images we find sufficient.

4.5.1 Training loop benchmark

Here, we investigate the runtime performances of the training loop. Since the training loop runs for more than 1000 iterations it is important that it is fast.

Input / Library	Epochs: 1 Img shape: 28x28	Epochs: 2 Img shape: 28x28	Epochs: 3 Img shape: 28x28
PyTorch	6432 μs	12875 μs	19342 μs
Futhark	183486 μs	367349 μs	548570 μs

Fig. 17: Training loop benchmark

As expected, our implementation is far slower than the PyTorch implementation.

4.6 Validation

In this section, we first use visual inspection and compare the generated images with the MNIST dataset to get a rough estimate about the generative performance. We then apply different techniques such as structural similarity index measure (SSIM), to further compare the images. Finally, we attempt to classify our samples using a trained CNN classifier.

4.6.1 Visual inspection

In this section, we look at how well the generated images resemble the handwritten digits in the MNIST dataset. The goal of this validation is that the digits are virtually indistinguishable from those found in the MNIST dataset. Figure 18 shows a selected set of the generated digits we consider to be the best. In our

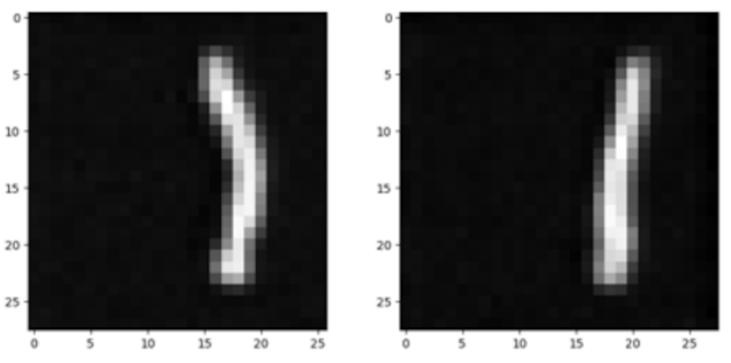

Fig. 18: Generated digits of best quality

assessment, it is fairly easy to classify which digit the above images should represent, suggesting that our model has learned to reproduce the essential features of different digit classes. However, when closely compared to the MNIST images we also see shortcomings of the model. For example, the generated images tend to have noisier backgrounds than the typical MNIST images. It is difficult for our model to make the background entirely black, and a different structure of the U-Net might be needed. Figure 19 shows a side-by-side comparison of

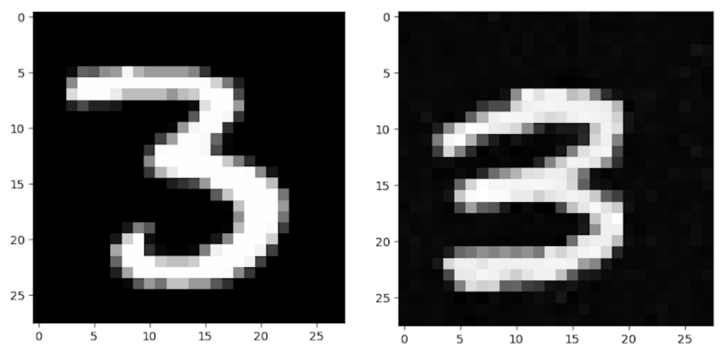

Fig. 19: Generated (left) and real (right) digits

the background in the generated and real images, clearly showing differences in background noise.

Moreover, our diffusion model does not always generate images we would recognize as digits. Figure 20 shows a batch of images generated in succession of each other with the same weights. Despite this, we are positive about the

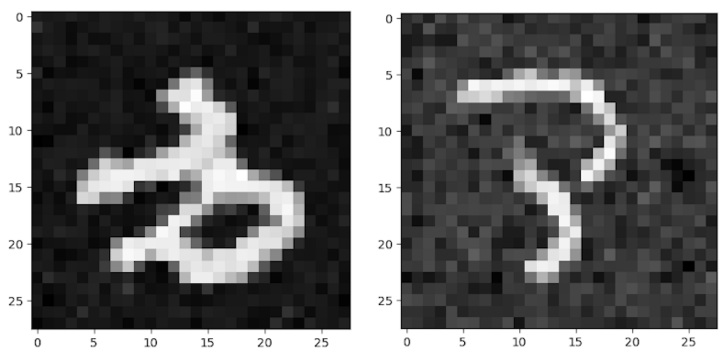

Fig. 20: Unrecognizable digits

overall performance of our model. Its ability to generate recognizable digits is promising, and we believe that with further refinement of the U-net, the model might have the potential of generating more complex images.

4.6.2 Image Comparison Techniques

There are several established methods for comparing images in the field of image processing and computer vision. These methods measure similarity or difference based on various aspects of the images, such as pixel intensity, structure, texture, and more. We have chosen two methods for validating our images, structural similarity index measure (SSIM), and histogram of oriented gradients (HOG).

4.6.3 Structural similarity index measure (SSIM)

The Structural Similarity Index Measure (SSIM) [31] is used in our analysis to quantify similarities between two images. SSIM takes into account the structural information, luminance, and contrast—attributes that we consider to be significant when evaluating the generated images.

To evaluate the images, we take a subset of generated images and their corresponding training set. We calculate the SSIM value for each pair, returning the highest SSIM value for every generated image. This gives us a sense of how well the model can reproduce examples from the training data. The following array shows the best SSIM results for a selected subset of images:

```
[0.2685010054964179]
```

As seen above, the highest SSIM score is 0.2685 which is far from the optimal score of 1.

4.6.4 Histogram of Oriented Gradients (HOG)

In this section, we use the Histogram of Oriented Gradients (HOG) [4] to validate the generated samples. HOG is a feature descriptor that captures the distribution of shapes or edges, within an image. This is a be a useful measure for validating if our model is capturing the shapes and structures found in the training images.

For our assessment, we take a subset of the generated images and their corresponding training set. We then compute the HOG descriptor for every image in these sets and subsequently compute the mean and variance of these descriptors. The following shows the computed values of the two sets:

```
Mean HOG descriptor for training images:
  0.3174485677126979
Mean HOG descriptor for generated images:
  0.30338887962731204
Standard Deviation of HOG descriptor for training images:
  0.15564834336865363
Standard Deviation of HOG descriptor for generated images:
  0.18153563743427917
```

Investigating the results, we see that the mean and variances of the generated images are reasonably similar to those of the training images. The lower mean for the generated images suggests that the edges in the generated images are oriented slightly differently than those in the training images. Furthermore, the higher variance indicates a larger variety of edge orientations.

4.6.5 Discussion of Image Comparison Techniques

While these metrics provide us with quantitative insights into the structural and shape similarities between the training and generated images, it's critical to bear in mind that neither SSIM nor HOG offer a complete evaluation of our model's performance.

These metrics are mostly used in areas like Image Compression [11], Face Recognition [5], and Image Segmentation [35], which don't directly correspond to the task of image generation.

Several other techniques could potentially offer more appropriate evaluations of the generative capabilities of the diffusion model. One example is the *Inception Score* (IS) metric, which was introduced by Salimans et al. in 2016 [25]. This algorithm evaluates generated images based on two key principles: diversity and distinctness. A high score reflects a set of images with varied content, each of which distinctly resembles a particular object. However, due to time limitations, we did not implement this algorithm in our evaluation.

An important element to keep in mind when interpreting our validation results is that the goal of our model is not to make a perfect replication of the training data. Rather, we seek to generate new images that closely resemble the training data in terms of structural, and contrast characteristics, but still are different. Despite the modest scores for SSIM and HOG, a visual inspection of the generated images suggests that the model is indeed creating images that are recognizably similar to the training data.

This brings us to a different issue, which is the other side of the coin. We should strive after generating images that are distinct from the training data. In more detail, we aim for our model to avoid overfitting, which would result in replicating the training images rather than generating novel ones. To evaluate whether our model is truly creating new images, we employ a nearest-neighbor analysis.

4.6.6 Nearest neighbor validation

In this section, we use the nearest neighbor algorithm to make sure that our model is not only memorizing and regenerating specific training images. This method involves comparing each generated image to all images in the training set and finding the most similar one. If the generated image is almost identical to its nearest neighbor, it is likely that our model is overfitted to the training data.

Figure 21 shows two examples of a generated image and its closest neighbor. We see that while the generated images are similar to their closest neighbors from the training set, they are not exact duplicates. This is highly encouraging as it suggests that our diffusion model has learned to capture the underlying structure of the data without overfitting, demonstrating positive generative capabilities.

4.7 Classification of Generated Images

The final and most conclusive evaluation of the generated images is to classify them using a trained CNN. If the accuracy of the trained CNN on the generated dataset is close to the accuracy of the trained CNN on the training/test data, this is great results.

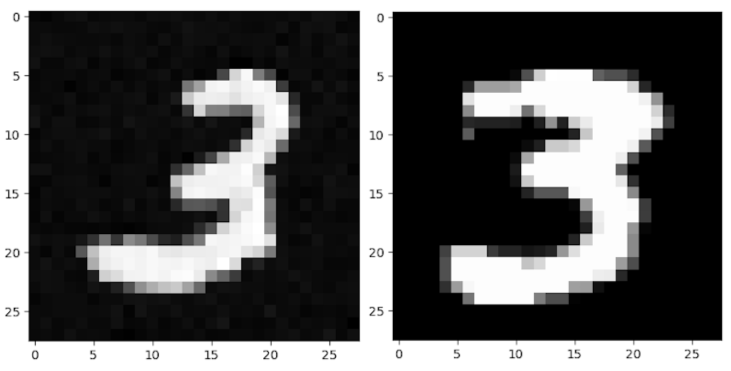

Fig. 21: A generated image (left) and its nearest neighbor (right)

We know from the design and implementation section that we initially tried to use a variant of the LeNet-5 for the diffusion model. Even though this was proven to be futile, we have still implemented the layers needed to build the LeNet-5 architecture. Since this was the initial intend of the LeNet-5 model, we chose to implement this in Futhark but use it for classification. Thus, by following the architecture of LeNet-5, with minor changes, we present the following CNN for MNIST classification, named LeNet-5-Fut:

```
Input image: 1 (channel) x 28 (height) x 28 (width)
Convolution: 5x5 kernel + 2 padding: 6 x 28 x 28
Activation layer (ReLU) : 6 x 28 x 28
Average Pool: 2x2 kernel + 2 stride: 6 x 14 x 14
Convolution:  5x5 kernel + 0 padding: 16 x 10 x 10
Activation layer (ReLU) : 16 x 10 x 10
Average Pool: 2x2 kernel + 2 stride: 16 x 5 x 5
Flatten : 400
Dense layer: 120 fully connected neurons
Activation layer (ReLU) : 120
Dense layer: 84 fully connected neurons
Activation layer (ReLU) : 84
Dense layer: 10 fully connected neurons
Output activation (Softmax): 10 fully connected neurons
```

4.8 CNN-Fut

The most notable difference between LeNet-5-Fut and CNN-Fut is the replacement of positional embeddings with the introduction of a softmax layer. This gives us a model which is almost identical to the original LeNet-5.

4.8.1 Softmax layer

Looking at the architecture for LeNet-5 we see that the last layer is a softmax activation layer. Softmax is is a mathematical function that maps n real numbers into a probability distribution of n probabilities. These probabilities are proportional to the initial values making softmax highly useful for multiclass classification tasks, such as classifying the MNIST dataset.

The softmax function is defined by:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{23}$$

Where z is the input vector of K elements, and $i = 1, ..., K$

By producing a probability distribution, the softmax function enables us to directly interpret the model's confidence in its predictions. For example, an output from the softmax function for our MNIST classification might look like this:

```
[0.0, 0.1, 0.3, 0.0, 0.0, 0.0, 0.0, 0.6, 0.0, 0.0]
```

Here, the model is 60% confident that the image represents the digit 7.

For our multi-class classification problem where each instance strictly belongs to one class, using another activation function like sigmoid or ReLU in the output layer could bring potential issues.

Sigmoid:
The sigmoid function, like softmax, maps input numbers to a number between 0 and 1. However, sigmoid processes each output value independently, which could lead to multiple classes showing high probabilities. This is not desirable for our use case.

ReLU:
Several reasons make ReLU unsuitable for our output case. Firstly, ReLU suppresses negative input values, which could result in loss of important information. Secondly, ReLU does not bound or normalize its output. This is unsuitable for our use case, where interpreting the output as probabilities is crucial.

Implementation
Implementing the softmax function in Futhark is relatively straightforward:

```
def softmax [n] (X : [n]f64) : [n] =
  let X_exp = map f64.exp X
  let X_sum = f64.sum X_exp
  in map (\ x -> x / X_sum) X_exp
```

We initially find the exponent of each value in the input array `X` to obtain `X_exp`. Then, we calculate the sum of `X_exp` to get `X_sum`. Finally, we divide each element in `X_exp` by `X_sum` to get the final result.

More interesting is the backwards version of the softmax function. This is because unlike other activation functions softmax maps from a vector to a vector. Hence, the derivative of the softmax function is a Jacobian matrix, not a simple scalar.
Looking at the Jacobian matrix we see the following:

$$\text{If } i = j : \frac{\partial \sigma(z)_i}{\partial z_j} = (1 - \sigma(z)_j)\sigma(z)_i$$

$$\text{If } i \neq j : \frac{\partial \sigma(z)_i}{\partial z_j} = -\sigma(z)_j\sigma(z)_i$$

We see that if we change the input z_j, it will increase the corresponding output $\sigma(z)_j$ but will decrease all other outputs where $i \neq j$. Intuativly this makes sense, since the softmax function outputs a probability distribution. Thus, increasing the probability of one class must naturally decrease the probability of all other classes, so that the sum of all probabilities remains 1.

The following shows the futhark implementation of the backwards softmax:

```
def softmax_b [m] (out_grad : [m]f64)
                  (softmax_pred : [m]f64) : [m]f64 =
  let identity : [m][m]f64 =
    tabulate_2d m m (\ y x -> f64.bool (y==x))
  let id_diff = map (\ y -> map2 (-) y softmax_pred) identity
  let jac =
    map2 (\id pred -> map (\ x -> pred * x) id) id_diff softmax_pred
  let out_grad_2d = map (\x -> [x]) out_grad
  in flatten (matmul jac out_grad_2d) :> [m]f64
```

We create the identity matrix and subtract the softmax predictions to make `id_diff`. This corresponds to calculating the $(1 - \sigma(z)_i)$ and $-\sigma(z)_i$ for the respective cases where $i = j$ and $i \neq j$.

We then multiply each row of `id_diff` with the respective element in `softmax_pred` (this element corresponds to $\sigma(z)_j$), to construct the Jacobian matrix `jac`. `out_grad` is now reshaped to match the dimensions required for the matrix multiplication, giving us `out_grad_2d`.

We make a matrix multiplication with the Jacobian and `out_grad_2d`, effectively computing the gradients with respect to the input of the softmax function. Lastly, we flatten the gradients to match the input shape.

4.9 Introduction to Futhark-ad

We have throughout this thesis seen how the functions for layers and their corresponding backwards functions are implemented. From this, we can conclude that the forward pass of each layer is significantly easier to implement and understand.

First of all, the number and different type of computations required for the backwards pass is often significantly higher than a functions respective forward pass. This is because we also have to compute the gradients for the weights and biases each time it is relevant. From a programming point of view, this is not only difficult but also time-consuming. A general trend throughout this project has been that the backwards pass has easily taken more than twice the time to implement.

Second, the amount of available, and easily readable, information about the backwards passes of layers is a fraction of that available for the forward pass, further contributing to the total implementation time.

Third, since the backwards functions require information gathered while making the forward pass, this requires that multiple values are returned in the `cache` of the forward function. This brings lots of packing an unpacking of tuples and sometimes bring problems in the form of the `opaque` futhark type[11]

However, Futhark introduces a solution to these problems. Namely the Futhark-AD library.

4.9.1 Futhark AD

Futhark-AD[12] is a library made to compute the derivatives of Futhark programs. It utilizes the automatic differentiation (AD) category to compute derivatives which are significantly easier than manual differentiation from a programmer's point of view. The idea behind AD is that it exploits that every function, no matter how complicated, is executed on a computer as a sequence of elementary operations such as additions, multiplications, and elementary functions such as sin and log. By successively applying the chain rules on the composition of operations, the exact derivatives can be computed in a mechanical fashion [1].

The use of AD is categorized into two modes, forward and reverse mode. Forward mode is however not very applicable in machine learning since its computational cost grows with the number of inputs. The reverse mode however computes all the derivatives at once with a cost that grows with the number of outputs [7]. Since we have much more parameters in neural networks (LeNet has around sixty thousand), we use the reverse mode for backpropagation in neural networks.

The idea behind reverse mode is to evaluate the function at a specific input. This is done when we compute the loss of the model, and then propagate the gradient of the output back through the computation, applying the chain rule at each operation. An example of the reverse mode AD in practice can be seen in the appendix.

However, even though the use of reverse mode seems unbeatable without any negatives, there is no such thing as free lunch. Since we use the operations of the evaluated function in the reverse adjoint trace, we need to store them. As the number of operations increases, the amount of storage required also increases proportionally, which may cause issues [13].

In Futhark-AD, the higher-order function corresponding to the reverse mode is

[11] https://readthedocs.org/projects/futhark/downloads/pdf/latest/
[12] https://futhark-lang.org/publications/sc22-ad.pdf
[13] The data-flow equations of checkpointing in reverse automatic differentiation

`vjp`, with the following types [26]:

$$\texttt{vjp} : (f : \alpha \to \beta) \to (x : \alpha) \to (dy : \beta) \to \alpha \tag{24}$$

Where: `f` is the function we want to find the derivative of. `x` is the point of evaluation. `dy` is the output adjoints. `vjp` stands for "vector-Jacobian product" and computes: $dy^T \cdot \mathbf{J}(f(x))$
The following shows how we use it in LeNet-5-fut for computing the gradients.

```
let loss weights = mse labels[i] (lenet_forward images[i] weights).0
  let grad_w = vjp loss weights 1f64
  let new_weights = lenet_SGD grad_w weights
```

Interesting for this paper is how the use of Futhark-ad compares to the implemented backward functions.

4.9.2 Benchmark of Futhark-AD

The following sections investigate how well futhark-ad benchmarks to manual differentiation.

Futhark ad is still in development Unfortunately, we get a compile error when we try to compile our model with the use of Futhark-ad to cuda. This means a comparison with PyTorch would be unfair and give no insight into how Futhark-AD peforms. We can however, still compile to c, and our benchmarks will therefore use this. This makes the results have less impact, but will still give an idea of how the use of Futhark-ad performs.

We should however also benchmark the time it takes to implement the backpropagation with and without ad. To make the backpropagation with ad, you only need to implement the forward pass of each method. In my personal experience, the time it takes to implement the forward function and manual backpropagation is propaly a 25/75 distribution. Also, when looking at the respective implementations, we see that AD is far easier to read and understand. The manual implementation of the backpropagation can be seen in the Appendix, far outweighing the previously seen AD implementation.

Lastly, the implementation currently invokes a compiler bug, which has caused a standstill. Once the bug is fixed, we expect to resume with implementing futhark-ad.

4.10 Traning of CNN-Fut

To use CNN-5-Fut to validate the generated images, we trained it on 200 digits from the MNIST dataset. Testing it, we obtained an accuracy of around 90%, which we find reasonable since the goal of the project hasn't been to implement a CNN giving astonishing results in classification.

4.11 Classify Generated Iimages

As the final test, we try to classify the following image using the trained CNN:
We get the following confidence results from the model:

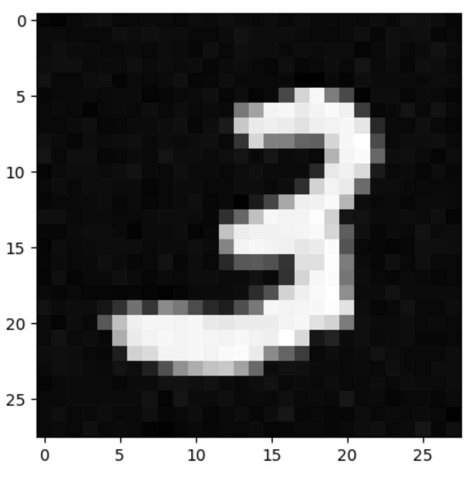

Fig. 22: A generated 3

```
    [0.000052, 0.000150, 0.025404, 0.954648, 0.000010,
     0.019435, 0.000068, 0.000017, 0.000216, 0.000001]
```

This is a huge achievement for the diffusion model and we believe that this concludes the project satisfactorily: we have trained our diffusion model on the MNIST dataset, generated several digits, selected the best ones, and verified that those are not just part of the training set. Finally, we have tried to classify the good images and gotten correct results.

5 Discussion

5.1 Related work

5.1.1 Protein design

With the consistent increase in computing power, *de novo* designing has been enabled to a much greater degree. Today, predicting the structure of a protein from its amino acid sequence (or vice versa) can be done by machine learning models — usually deep learning models such as neural networks. These type of information predicted by the models can vary. One example may be *contact maps*: 2D matrices that show whether or not two amino acid residues are within a certain threshold distance of each other. Because these matrices are similar to 2D images, which neural networks can successfully classify, deep learning models have been applied in the area of protein design with some success [15]. However, a recent study [32] has shown promising results with using diffusion models instead of deep learning models. It introduces the RoseTTAFold Diffusion (RF Diffusion) model which achieves an accuracy two orders of magnitude higher than existing deep learning models; even outperforming existing diffusion models [2]. An interesting property of this is also that so far, the state of the art of protein design software has generally been 3D modelling software, such as the Rosetta software suite[14] or other similar tools [33]. However, the RF Diffusion paper reports that ”*in a manner reminiscent of the generation of images from text prompts*”, it can generate protein designs from very simple specifications. This could potentially make protein design much easier, as you would be able to simply specify the desired protein behavior in a natural language instead of ”building” the protein structure yourself.

5.1.2 Machine learning

Despite the concept of DDPMs being so new, several possible improvements to their efficiency have been developed within recent years. While this thesis mostly focuses on the efficiency obtainable through the use of the Futhark compiler, these works instead focus on optimizing the model from an algorithmic perspective:

- Liu et al. [18] replace parts of DDPMs that use numerical methods with so-called *pseudo numerical methods*, which in many scenarios provide a significant increase in speedup — in some cases being up to 20 times as fast[15].
- Lam et al. [16] propose a new *bilateral deonising diffusion model* which parameterizes the traditional model with a scheduling network and score network. This allows the model to generate audio samples of the same quality as existing models, but with much fewer (as few as 7) sampling steps[16].
- Zhang and Chen [36] also show that the number of sampling steps can be greatly reduced by selecting a good *discretization scheme*, allowing for generation of high-quality samples in as few as 10 steps[17].

Note that all of the above optimizations are written in python (with a small section of the first example written in CUDA[18]. To our knowledge, this is the first work that attempts to implement DDPMs in any type of functional array programming language.

5.2 Future Work

While our attempt to implement a diffusion model in Futhark was ultimately a success, there are still many things that could be improved. First of all, our current model can only generate (good) digits if it is trained on digits of the same type. We would like to be able to train the model on all digits, and then still have it be able to generate ”good” new digits. Furthermore, the implementation of the network could be improved. For example, it should be possible to use batch-sizes greater than 1. Also, it would be good if the implementation was restructured as so to allow for more easily switching to new architectures. Finally, while the

[14] https://www.rosettacommons.org/software
[15] Implementation for [18]: https://github.com/luping-liu/PNDM
[16] Implementation for [16]: https://github.com/tencent-ailab/bddm
[17] Implementation for [36]: https://github.com/qsh-zh/deis
[18] https://github.com/luping-liu/PNDM/blob/master/model/scoresde/upfirdn2d_kernel.cu

thesis has focused quite heavily on the design and implementation of a proof-of-concept diffusion model, we have not investigated the opportunity for generating protein structures from amnio acid sequences, which was one of the motivations for constructing the diffusion model. However, since the model shows promising results, it would be interesting to continue with the challenge of protein structure generation.

6 Conclusion

We have designed and implemented a working denoising diffusion probabilistic model in Futhark. As part of the process, we have demonstrated that using a neural network with an LeNet-5 design is insufficient for performing the denoising operation. We have also demonstrated that a working implementation can be achieved by using a U-Net design. Finally, we have shown that our program can generate digits of acceptable quality, and that it in some cases matches the performance of an identical neural network implemented with the PyTorch library.

References

[1] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[2] Nathaniel R Bennett, Brian Coventry, Inna Goreshnik, Buwei Huang, Aza Allen, Dionne Vafeados, Ying Po Peng, Justas Dauparas, Minkyung Baek, Lance Stewart, et al. Improving de novo protein binder design with deep learning. *Nature Communications*, 14(1):2625, 2023.

[3] Vrushali Bongirwar and AS Mokhade. Different methods, techniques and their limitations in protein structure prediction: A review. *Progress in Biophysics and Molecular Biology*, 2022.

[4] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pages 886–893. Ieee, 2005.

[5] Oscar Déniz, Gloria Bueno, Jesús Salido, and Fernando De la Torre. Face recognition using histograms of oriented gradients. *Pattern recognition letters*, 32(12):1598–1603, 2011.

[6] Xiaohan Ding, Xiangyu Zhang, Jungong Han, and Guiguang Ding. Scaling up your kernels to 31x31: Revisiting large kernel design in cnns. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11963–11975, 2022.

[7] Andreas Griewank et al. On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6(6):83–107, 1989.

[8] Johannes Habich, Christian Feichtinger, Harald Köstler, Georg Hager, and Gerhard Wellein. Performance engineering for the lattice boltzmann

method on gpgpus: Architectural requirements and performance results. *Computers & Fluids*, 80:276–282, 2013.

[9] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 556–571, 2017.

[10] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 33:6840–6851, 2020.

[11] Alain Hore and Djemel Ziou. Image quality metrics: Psnr vs. ssim. In *2010 20th international conference on pattern recognition*, pages 2366–2369. IEEE, 2010.

[12] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

[13] Nikhil Ketkar. *Introduction to PyTorch*, pages 195–208. Apress, Berkeley, CA, 2017.

[14] Heehoon Kim, Hyoungwook Nam, Wookeun Jung, and Jaejin Lee. Performance analysis of cnn frameworks for gpus. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 55–64. IEEE, 2017.

[15] Brian Kuhlman and Philip Bradley. Advances in protein structure prediction and design. *Nature Reviews Molecular Cell Biology*, 20(11):681–697, 2019.

[16] Max WY Lam, Jun Wang, Dan Su, and Dong Yu. Bddm: Bilateral denoising diffusion models for fast and high-quality speech synthesis. *arXiv preprint arXiv:2203.13508*, 2022.

[17] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[18] Luping Liu, Yi Ren, Zhijie Lin, and Zhou Zhao. Pseudo numerical methods for diffusion models on manifolds. *arXiv preprint arXiv:2202.09778*, 2022.

[19] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. Dying relu and initialization: Theory and numerical examples. *arXiv preprint arXiv:1903.06733*, 2019.

[20] Alexander Quinn Nichol and Prafulla Dhariwal. Improved denoising diffusion probabilistic models. In *International Conference on Machine Learning*, pages 8162–8171. PMLR, 2021.

[21] Chao Peng, Xiangyu Zhang, Gang Yu, Guiming Luo, and Jian Sun. Large kernel matters–improve semantic segmentation by global convolutional network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4353–4361, 2017.

[22] Carol A Rohl, Charlie EM Strauss, Kira MS Misura, and David Baker. Protein structure prediction using rosetta. In *Methods in enzymology*, volume 383, pages 66–93. Elsevier, 2004.

[23] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, pages 234–241. Springer, 2015.

[24] Putt Sakdhnagool, Amit Sabne, and Rudolf Eigenmann. Regdem: Increasing gpu performance via shared memory register spilling. *arXiv preprint arXiv:1907.02894*, 2019.

[25] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. *Advances in neural information processing systems*, 29, 2016.

[26] R. Schenck, O. Ronning, T. Henriksen, and C. E. Oancea. Ad for an array language with nested parallelism. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 829–843, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society.

[27] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[28] Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International Conference on Machine Learning*, pages 2256–2265. PMLR, 2015.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[30] Leni Ven and Johannes Lederer. Regularization and reparameterization avoid vanishing gradients in sigmoid-type networks. *arXiv preprint arXiv:2106.02260*, 2021.

[31] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[32] Joseph L Watson, David Juergens, Nathaniel R Bennett, Brian L Trippe, Jason Yim, Helen E Eisenach, Woody Ahern, Andrew J Borst, Robert J Ragotte, Lukas F Milles, et al. Broadly applicable and accurate protein design by integrating structure prediction networks and diffusion generative models. *bioRxiv*, pages 2022–12, 2022.

[33] Christopher W Wood, Marc Bruning, Amaurys A Ibarra, Gail J Bartlett, Andrew R Thomson, Richard B Sessions, R Leo Brady, and Derek N Woolfson. Ccbuilder: an interactive web-based tool for building, designing and assessing coiled-coil protein assemblies. *Bioinformatics*, 30(21):3029–3035, 2014.

[34] Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

[35] Julius Žemgulys, Vidas Raudonis, Rytis Maskeliūnas, and Robertas Damaševičius. Recognition of basketball referee signals from videos using histogram of oriented gradients (hog) and support vector machine (svm). *Procedia computer science*, 130:953–960, 2018.

[36] Qinsheng Zhang and Yongxin Chen. Fast sampling of diffusion models with exponential integrator. *arXiv preprint arXiv:2204.13902*, 2022.

7 Appendix

7.1 Reverse mode AD example

Reverse mode example
We introduce the following notations: A function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ consists of the intermediate variables v_i with:

1. $v_{i-n} = x_i,\ i = 1, ..., n$ Are the input variables.
2. $v_i,\ i = 1, ..., l$ Are the intermediate variables.
3. $y_{m-i} = v_{l-i},\ i = m-1, ..., 0$ Are the output variables.
4. $\dot{v} = \frac{\partial v}{\partial x}$ Is the derivative of the variable.

Let $y = f(x_1, x_2) = x_1^2 - x_1 \cdot cos(x_2) + sin(x_1)$. We want to evaluate it at $(x_1, x_2) = (3, 8)$.
We start with the Forward Primal Trace, which will be the same as the forward mode:

$$\begin{aligned}
v_{-1} &= x_1 & &= 3 && (25)\\
v_0 &= x_2 & &= 8 && (26)\\
v_1 &= v_{-1}^2 & &= 3^2 && (27)\\
v_2 &= v_{-1} \cdot cos(v_0) & &= 3 \cdot cos(8) && (28)\\
v_3 &= sin(v_{-1}) & &= sin(3) && (29)\\
v_4 &= v_1 - v_2 & &= 9 + 0.4365 && (30)\\
v_5 &= v_4 + v_3 & &= 9.4365 + 0.1411 && (31)\\
y &= v_5 & &= 9.5776 && (32)
\end{aligned}$$

For the Reverse Adjoint Trace, we let $\bar{v_5} = \bar{y} = \frac{\partial y}{\partial y} = 1$ to compute both $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$. Recall that this is computed ”bottom-up”:

$$\begin{aligned}
\bar{v_5} &= \bar{y} = 1\\
\bar{v_4} &= \bar{v}_5 \frac{\partial v_5}{\partial v_4} = \bar{v}_5 \cdot 1 = 1\\
\bar{v_3} &= \bar{v}_5 \frac{\partial v_5}{\partial v_3} = \bar{v}_5 \cdot 1 = 1\\
\bar{v_1} &= \bar{v}_4 \frac{\partial v_4}{\partial v_1} = \bar{v}_4 \cdot 1 = 1\\
\bar{v_2} &= \bar{v}_4 \frac{\partial v_4}{\partial v_2} = \bar{v}_4 \cdot (-1) = -1\\
\bar{v}_{-1} &= \bar{v}_3 \frac{\partial v_3}{\partial v_{-1}} = \bar{v}_3 \cdot cos(v_{-1}) = -0.99\\
\bar{v}_{-1} &= \bar{v}_{-1} + \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}} = \bar{v}_{-1} + \bar{v}_2 \cdot cos(v_0) = -0.844\\
\bar{v_0} &= \bar{v}_2 \frac{\partial v_2}{\partial v_0} = \bar{v}_2 \cdot v_{-1} \cdot (-sin(v_0)) = -2.968\\
\bar{v}_{-1} &= \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} = \bar{v}_{-1} + \bar{v}_1 \cdot 2 \cdot v_{-1} = 5.156\\
\bar{x}_1 &= \bar{v}_{-1} = 5.156\\
\bar{x}_2 &= \bar{v_0} = 2.968
\end{aligned}$$

```
def naive_convolve2D [n][m][p][k][l][o] (imgs : [l][n][m]f64) (kernels : [o][l][p][k]f64) (biase
  let flat_pk = p*k
  let new_n = (((n+(padding*2))-p)+1)
  let new_m = (((m+(padding*2))-p)+1)

  let imgs_padded =
    if (padding != 0) then
      add_padding imgs padding
    else
      imgs

  let c1 = map (\ kernel_3d ->
    tabulate_2d new_n new_m (\ y x ->
      reduce (+) 0 (flatten (map2 (\ kernel img ->
        (flatten (map2 (\ i j ->
          map2 (*) i j)
        (img[y:(y+p),x:(x+k)] :> [p][k]f64) kernel)) :> [flat_pk]f64
      ) kernel_3d imgs_padded))
    )
  ) kernels

  let c1_b = tabulate_3d o new_n new_m (\z y x -> c1[z,y,x] + biases[z])
  in c1_b
```

```
def block_reverse [l][n][m] (out_grad: [l][n][m]f64) (num_groups : i64) (weights) (cache) =
  let (conv1_w,_,t_w,_,conv2_w,_) = weights
  let (imgs, time_mlp, conv1, lin_out, comb, conv2,conv1_gnorm_cache,conv2_gnorm_cache) = cache
  let conv2_gnorm_b = group_norm_b out_grad num_groups 1e-05 conv2_gnorm_cache
  let conv2_act_b = ReLU_3d_b conv2_gnorm_b conv2
  let (conv2_b,c2_w_grad,c2_b_grad) = convolve2D_b conv2_act_b comb conv2_w 1 1
  let conv1_gnorm_b = group_norm_b conv2_b num_groups 1e-05 conv1_gnorm_cache
  let out_grad_conv_act = ReLU_3d_b conv1_gnorm_b conv1
  let out_grad_sum = map (map (reduce (+) 0)) conv2_b
  let out_grad_lin_act = map2 ReLU_b (map (reduce (+) 0) out_grad_sum) lin_out
  let (_, t_w_grad, t_b_grad) = dense_b out_grad_lin_act time_mlp t_w
  let (input_grad, c_w_grad, c_b_grad) = convolve2D_b out_grad_conv_act imgs conv1_w 1 1
  in (input_grad, (c_w_grad, c_b_grad, t_w_grad, t_b_grad, c2_w_grad, c2_b_grad))
```

```
def lenet_reverse [n][m] (img : [n][m]f64) (prediction : []f64) (label: []f64) (weights) (cache)
  let (C1_w,C1_b,C3_w,C3_b,F6_w,F6_b,F7_w,F7_b,F8_w,F8_b) = weights
  let (C1_layer,C1_layer_activation,S2_layer,C3_layer,C3_layer_activation,S4_layer,F5_layer,F6_
  let grad = mse_prime label prediction
  let prediction_b : [10]f64 = softmax_b grad prediction :> [10]f64
  let (F8_layer_b,F8_w_grad,F8_b_grad) = dense_b prediction_b F7_layer_activation F8_w
  let F7_layer_activation_b : [84]f64 = map2 ReLU_b F8_layer_b F7_layer
  let (F7_layer_b,F7_w_grad,F7_b_grad) = dense_b F7_layer_activation_b F6_layer F7_w
  let F6_layer_activation_b : [120]f64 = map2 ReLU_b F7_layer_b F6_layer
  let (F6_layer_b,F6_w_grad,F6_b_grad) = dense_b F6_layer_activation_b F5_layer F6_w
  let F5_layer_b = unflatten_3d 16 5 5 F6_layer_b
  let S4_layer_b = avg_pool_b F5_layer_b 2 :> [16][10][10]f64
  let C3_layer_activation_b = map2 (\ y_g y -> map2 (\ x_g x -> map2 ReLU_b x_g x) y_g y) S4_la
  let (C3_layer_b,C3_w_grad,C3_b_grad) = convolve2D_b C3_layer_activation_b S2_layer C3_w 0i64
  let S2_layer_b = avg_pool_b C3_layer_b 2 :> [6][28][28]f64
  let C1_layer_activation_b = map2 (\ y_g y -> map2 (\ x_g x -> map2 ReLU_b x_g x) y_g y) S2_la
  let (C1_layer_b,C1_w_grad,C1_b_grad) = convolve2D_b C1_layer_activation_b [img] C1_w 2i64 2i6
  in (C1_w_grad,C1_b_grad,C3_w_grad,C3_b_grad,F6_w_grad,F6_b_grad,F7_w_grad,F7_b_grad,F8_w_grad,
```