Intelligent Systems for Pattern Recognition: project report

Ninniri Matteo (student ID: 543873)

October 23, 2022

Contents

1 Summary

This report will describe the work realized for the exam on Intelligent Systems for Pattern Recognition, namely a Denoising Diffusion Probabilistic Model (DDPM from here on) for image generation. One thing that makes this implementation differ from the others publicly available is the fact that the Attention modules employ a relatively new variant of the Transformer called Performer, which claims to be able to approximate the Attention matrix with a time and space complexity linear with respect to the sequence length (which in our case is the number of pixels in the image).

The solution will then be compared, with a focus on time and space requirements rather than sample quality, against the original paper's implementation, which also employs a linear approximation of the Attention operator itself.

2 Methods

2.1 DDPM

Please notice that this project is based on the work by Ho et al. [1], which has already been improved by several newer papers. However, the simplicity of the original work made it more suitable for a university project with a limited amount of resources.

DDPMs are a family of generative models which have recently become a trending topic in AI research, as it has been shown that they are capable of generating samples of higher quality than the ones generated by state-of-the-art models such as GANs while generally requiring less computational power for training.

The idea behind DDPMs is the following: given an image $x_0 \sim q(x_0)$, where q is the distribution of the images that we want to learn, we can progressively add random gaussian noise through a Markov Process $q(x_t|x_{t-1})$ until the original signal (the image) is destroyed after a predefined amount of steps T, and the resulting image x_T is nothing but pure random Gaussian noise.

Ideally, we could try and reverse this process through an inverse, parametrized Markov Process named $p_\theta(x_{t-1}|x_t)$ in which we train a model to progressively remove the noise from a noisy signal x_T until we restore the original signal x_0. By training a model in this way, one could effectively sample $x_T \sim \mathcal{N}(0, I)$ (since x_T is sampled from a distribution close to an isotropic gaussian) and use p_θ to progressively denoise it until, after T steps, we have sampled a new image from q.

Both models are going to be defined formally in the following sections.

2.1.1 $q(x_t|x_{t-1})$

We will start with the *diffusion process* $q(x_t|x_{t-1})$ as it is the most straightforward one, and it does not require training.

In general, given x_t as an input, with $t \in \{1, T\}$, we can add noise by sampling x_t from

$$q(x_t|x_{t-1}) := \mathcal{N}(x_{t-1}; \sqrt{1-\beta_t}x_{t-1}, \beta_t I) \tag{1}$$

where $\beta_1, \ldots, \beta_T$ are called *variance schedulers* and, as the name suggests, they bound the variance in the sampling process. The paper uses $\beta_1 = 10^{-4}$ and $\beta_T = 0.02$, and spaces the values in-between in a linear fashion.

Sampling from the above distribution means that, from a practical point of view,

$$x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}\epsilon \tag{2}$$

with $\epsilon \sim \mathcal{N}(0, I)$.

Since $\sqrt{1-\beta_t}$ slowly decreases as t approaches T, and vice-versa for β_t, $\sqrt{1-\beta_t}x_t$ approaches zero as $\beta_t I$ approaches one. If T is large enough and the variance schedulers are well-behaved, then, x_T is close to being sampled from $\mathcal{N}(x_{T-1}; 0, I)$, an isotropic gaussian distribution, confirming the above statement on how the final iteration of the diffusion process is random Gaussian noise and, as a result, we can sample starting from $x_T \sim \mathcal{N}(0, I)$.

Another important thing to notice is that we can sample x_t directly from the original image x_0 by unrolling equation 2 (and merging the various ϵ) like in equation 3:

$$x_t = \sqrt{\overline{\alpha}_t}x_0 + \sqrt{1-\overline{\alpha}_t}\epsilon \tag{3}$$

where $\alpha_t = 1 - \beta_t$ and $\overline{\alpha}_t = \prod_{s=1}^{t} \alpha_s$.

2.1.2 $p_\theta(x_{t-1}|x_t)$

The *reverse process* p_θ is trickier as it includes trainable parameters. In general, it has to approximate the *inverse distribution* $q(x_{t-1}|x_t)$ (notice the inverse order of t and $t-1$) which is equal to equation 4:

$$q(x_{t-1}|x_t) := \mathcal{N}(x_{t-1}; \overline{\mu}(x_t, t), \overline{\Sigma}(x_t, t)) \tag{4}$$

This means that its approximation p_θ should be something in the form of equation 5:

$$p_\theta(x_{t-1}|x_t) := \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t)) \tag{5}$$

where $\mu_\theta(x_t, t)$ and $\Sigma_\theta(x_t, t)$ are trainable neural networks. Ho ea al. have actually simplified the procedure by setting $\overline{\Sigma}(x_t, t) = \sigma_t^2 I$ where σ_t^2 can amount to either β_t or $\frac{1-\overline{\alpha}_{t-1}}{1-\overline{\alpha}_t}\beta_t$ (the latter being used in this project). This choice was justified by the empirical results obtained from their experiments. Please notice how, although this work closely follows the original paper, more recent works such as Dhariwal et al. [2] have shown how learning the variance is indeed crucial for obtaining higher-quality samples.

The q as defined in equation 4 is actually untractable. However, we can make it tractable by conditioning it on x_0:

$$q(x_{t-1}|x_t, x_0) := \mathcal{N}(x_{t-1}; \overline{\mu}(x_t, x_0), \overline{\beta}_t I) \tag{6}$$

Through Bayes' rule, we obtain that $\overline{\mu}(x_t, x_0)$ and $\overline{\beta}_t$ have the optimal values of

$$\overline{\mu}_t(x_t, x_0) := \frac{\sqrt{\alpha_t}}{\beta_t}\mathbf{x}_t + \frac{\sqrt{\overline{\alpha}_{t-1}}}{1-\overline{\alpha}_{t-1}}\mathbf{x}_0 \tag{7}$$

$$\overline{\beta}_t := \frac{1-\overline{\alpha}_{t-1}}{1-\overline{\alpha}_t}\beta_t \tag{8}$$

By plugging equation 2 into equation 7 we can remove x_0 completely and obtain

$$\overline{\mu}_t(x_t) := \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\overline{\alpha}_t}}\epsilon_t) \tag{9}$$

This last result is particularly important as it tells us that we have everything we need to calculate the optimal mean, with the exception of ϵ_t. As a result, we can effectively rewrite (or

reparametrize) the problem to predict ϵ_t instead of $\overline{\mu}_t(x_t)$. This way, we obtain a new formulation of $\mu_\theta(x_t, t)$, which is shown in equation 10:

$$\mu_\theta(x_t, t) = \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\overline{\alpha}_t}}\epsilon_\theta(x_t, t)) \tag{10}$$

where $\epsilon_\theta(x_t, t)$ is a neural network trained to predict the noise present in x_t. As a result, we can sample x_{t-1} from x_t through the following distribution:

$$x_{t-1} \sim p_\theta(x_{t-1}|x_t) := \mathcal{N}(x_{t-1}; \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\overline{\alpha}_t}}\epsilon_\theta(x_t, t)), \sigma_t^2 I) \tag{11}$$

From a practical point of view: if $\epsilon \sim \mathcal{N}(0, I)$, then x_{t-1} is:

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\overline{\alpha}_t}}\epsilon_\theta(x_t, t)) + \sigma_t\epsilon \tag{12}$$

which is our reverse process.

2.1.3 Loss function

Before discussing the training and sampling algorithms, we need to discuss which loss function to use.

In general, the paper attempts to minimize the negative log-likelihood on p_θ:

$$\mathbb{E}[-\log p_\theta(x_0)] \tag{13}$$

Through a fairly long demonstration that mainly involves Jensen's inequality, the paper shows how this value is upper-bounded by L, which is defined as follows:

$$\mathbb{E}_q[-\log p_\theta(x_0)] \leq L := L_T + L_{T-1} + \dots L_1 + L_0 \tag{14}$$

where:

- $L_T = D_{\mathrm{KL}}(q(x_T|x0)||p(x_T))$
- $L_t = D_{\mathrm{KL}}(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t)); t \in \{2, \dots, T-1\}$
- $L_0 = -\log p_\theta(x_0|x_1)$

This means that we can minimize the various L_ts instead of the log-likelihood to solve the problem.

The original paper heavily simplifies the various terms. In particular, L_T is shown to be a constant with no learnable parameters and can be ignored as a consequence.

The various L_ts are the most complex ones. They are parametrized to minimize the difference between the optimal mean $\overline{\mu}_t$ (equation 9) and the "predicted" mean μ_θ (equation 10):

$$\mathbb{E}_{x_0,\epsilon}[\frac{1}{2\sigma_t^2}||\overline{\mu}_t(x_t) - \mu_\theta(x_t, t)||^2] \tag{15}$$

We already know the values for both $\overline{\mu}_t$ (equation 9) and μ_θ (equation 10), so we can plug them directly into equation 16:

$$\mathbb{E}_{x_0,\epsilon}[\frac{1}{2\sigma_t^2}||\frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\overline{\alpha}_t}}\epsilon_t) - \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\overline{\alpha}_t}}\epsilon_\theta(x_t,t))||^2] \tag{16}$$

We can group the various terms and simplify them as

$$\mathbb{E}_{x_0,\epsilon}[\frac{(1-\alpha_t)^2}{2\alpha_t(1-\overline{\alpha}_t)\sigma_t^2}||\epsilon - \epsilon_\theta(x_t,t)||^2] \tag{17}$$

We already know x_t from equation 3 and we can plug it inside the equation:

$$\mathbb{E}_{x_0,\epsilon}[\frac{(1-\alpha_t)^2}{2\alpha_t(1-\overline{\alpha}_t)\sigma_t^2}||\epsilon - \epsilon_\theta(\sqrt{\overline{\alpha}_t}x_0 + \sqrt{1-\overline{\alpha}_t}\epsilon, t)||^2] \tag{18}$$

So we can see once again how minimizing the difference between the means comes down to a problem of identifying the noise of a sample at a specific timestep t.

Finally, the authors state how empirical evidence shows that the following simplification, which is L_t's final form, gives the same if not equal results:

$$L_t := \mathbb{E}_{x_0,\epsilon}[||\epsilon - \epsilon_\theta(\sqrt{\overline{\alpha}_t}x_0 + \sqrt{1-\overline{\alpha}_t}\epsilon, t)||^2] \tag{19}$$

meaning that the whole optimization of L_t reduces itself to the mean square error between the true noise and the predicted noise.

L_0 is also simplified to the same equation as L_t.

2.1.4 Training algorithm

If we observe equation 14 we can see how, theoretically, even minimizing one single L_t is effectively a form of optimization. Through this reasoning, we can design a more efficient training loop: instead of optimizing the whole sequence of L_ts for a training sample x_i, we can instead sample one timestep t uniformly between 1 and T, and optimize only L_t. We have seen how it is possible to calculate $(x_i)_t$ directly from $(x_i)_0 = x_i$ using equation 3. Since the latter requires us to input ϵ, we have all the input parameters needed to optimize for L_t according to equation 19.

The final algorithm is shown in algorithm 1:

Algorithm 1: Training algorithm

while *True* **do**
- $x_0 \sim q(x_0)$
- $t \sim$Uniform$(\{1,\dots,T\})$
- $\epsilon \sim \mathcal{N}(0, I)$
- Take a gradient descent step on $\nabla_\theta||\epsilon - \epsilon_\theta(\sqrt{\overline{\alpha}_t}x_0 + \sqrt{1-\overline{\alpha}_t}\epsilon, t)||^2$

end

2.1.5 Sampling algorithm

The sampling algorithm follows the inverse reasoning: starting from pure gaussian noise, we perform an iterative procedure of T steps in which at step t we calculate x_{t-1} starting from x_t by sampling from the distribution on equation 11, following effectively equation 12. Please notice how, at step $t = 1$ we do not need to sample the noise, which is equal to zero.

Algorithm 2: Sampling algorithm

while *True* **do**
  $x_T \sim \mathcal{N}(0, I)$
  for $t = T, \ldots, 1$ **do**
    $\epsilon \sim \mathcal{N}(0, I)$ if $t > 1$ else $z = 0$
    $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)) + \sigma_t \epsilon$
  end
  return x_0
end

2.1.6 $\epsilon_\theta(x_t, t)$

The only thing left is discussing what exactly is ϵ_θ. The original paper does not discuss it too much, but it is essentially a ResNet [3] shaped as a U-net[4]. They are not so much different from an autoencoder which passes the input through a "bottleneck" before attempting to reconstruct it, with the exception that it predicts the noise instead of the image itself. The timestep is not passed as input. Instead, it is incorporated in the image through a sinusoidal positional embedding, the same used by Vaswani et al. [5] in the transformer architecture.

This work follows the same implementation of the original paper although, as we are going to see in section 2.2, the Attention blocks are replaced with a novel Attention approximation.

2.2 Performer

As anticipated in the last section, DDPMs usually employ Attention blocks inside the neural network utilized to estimate an image's noise. Standard Attention has a complexity that is quadratic on the sequence's length L. In our setting, L is equal to the number of pixels in an image. This makes Attention a particularly expensive operation for image processing, as even a simple 32x32 image means L greater than 1000. The original paper's implementation employs a variant of Attention named Linear Attention [6] which, as the name suggests, is capable of approximate Attention in linear time.

In this work, we will also test another type of Attention approximation, namely the Performer by Choromanski et al. [7], which is capable to approximate Attention arbitrarily close with a complexity of $\mathcal{O}(Ld^2 \log d)$, where d is the dimensionality of a single element of the sequence. In our case, it is the number of channels for each pixel. Keep in mind that, although the input image has 1 single channel for grayscale images, and 3 for RGB images, they are eventually increased as they go through the neural network's bottleneck before being compressed again to their original number.

2.2.1 Theory

The standard Attention is calculated as follows: given queries Q, keys K and values V, all three of them $\in \mathbb{R}^{L \times d}$, then we have that

$$Att_{\leftrightarrow}(Q, K, V) = D^{-1}AV \tag{20}$$

where $A = exp(\frac{QK^T}{\sqrt{d}})$, $D = diag(A1_L)$ and 1_L is the vector of L elements all set to one.

As we can see, $A \in \mathbb{R}^{L \times L}$ and $V \in \mathbb{R}^{L \times d}$, which means that the AV product requires $\mathcal{O}(L^2 d)$ operations.

The idea behind the performers is to approximate Q and K and rearrange the multiplications in a way such that they do not require a quadratic cost with respect to L.

In particular, we see that if we could somehow multiply K^T against V first, we would obtain a $d \times d$ matrix with $\mathcal{O}(Ld^2)$ operations, which once again would require the same amount of operations to be multiplied against Q. The only issue is that we have the *exp* function, and the softmax operator in general, in the way, and it is difficult to decompose.

The FAVOR+ mechanism at the base of the Performer attempts to approximate the A matrix through the use of Random Fourier Features for Kernel methods, which were first proposed by Rahimi et al. [8].

The idea is to have $A(i,j) = Ker(q_i^T; k_j^T)$ where q_i and k_j are respectively the i^{th} row of Q and the j^{th} of K and $Ker : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+$ is a function of the form

$$Ker(x,y) = \mathbb{E}[\phi(x)^T \phi(y)] \tag{21}$$

According to Rahimi et al., almost every kernel Ker used in practice can be obtained by choosing $\phi(u)$ from equation 21 in the form of equation 22:

$$\phi(x) = \frac{h(x)}{\sqrt{m}}(f_1(x\omega_1^T), \ldots, f_1(x\omega_m^T), \ldots, f_l(x\omega_1^T), \ldots, f_l(x\omega_m^T)) \tag{22}$$

where:

- $h : \mathbb{R}^d \to \mathbb{R}$
- $f_1 \ldots f_l$ are l functions of the form $\mathbb{R} \to \mathbb{R}$
- $\omega_1 \ldots \omega_m \sim \mathcal{D} \in \mathcal{P}(\mathbb{R}^d)$ are m orthogonal vectors (if $m > d$, then whey are orthogonal in blocks of d elements).

The higher the value of m, the most accurate is the approximation. The choice of h and $f_1 \ldots f_l$ depends on which function we want to approximate. In particular, for the softmax function, we need to choose:

- $h(x) = exp(-\frac{||x||^2}{2})$
- $l = 1; f_1(x) = exp(x)$
- $\mathcal{D} = \mathcal{N}(0, I_d)$

A second approximation that uses $l = 2$ (meaning two functions f_1 and f_2) is also given, but only the first one was used in the final implementation of the project.

Notice that, although the original paper by Choromanski et al. is a little vague on the matter, it is also necessary to normalize both the denominator of $h(x)$ and the x vector inside the various $f_1 \ldots f_l$ by a power of d in order to emulate the normalization operation done by $\sqrt{d}$ in $A = exp(\frac{QK^T}{\sqrt{d}})$. Keep in mind that in multi-head Attention setups, d is equal to the dimensionality of a single head d_{head} instead of $d = n_{heads} * d_{head}$.

Summarizing, the ϕ function for the softmax operator is

$$\phi(x) = \frac{1}{\sqrt{m}} exp(-\frac{||x||^2}{2\sqrt{d}})(exp(\frac{x}{\sqrt[4]{d}}\omega_1^T), \ldots, exp(\frac{x}{\sqrt[4]{d}}\omega_m^T) \tag{23}$$

A stable version is the following:

$$\phi(x) = \frac{1}{\sqrt{m}}(exp(\frac{x}{\sqrt[4]{d}}\omega_1^T - \frac{||x||^2}{2\sqrt{d}}), \ldots, exp(\frac{x}{\sqrt[4]{d}}\omega_m^T - \frac{||x||^2}{2\sqrt{d}})) \tag{24}$$

In practical implementations, each exponent in equation 23/24 also contains a small constant in the order of 10^{-5} for numerical stability.

The output of ϕ is a vector that has as many elements as the one between parenthesis in equation 23, which is equal to $r = l * m$. Since in equation 23's setup we have $l = 1$, it means that $r = m$.

At this point, we can approximate the softmax operator through the following rearrangement of the matrix multiplications: given $Q', R' \in \mathbb{R}^{L \times r}$ where the rows of these two matrices are obtained by applying ϕ to the rows of Q and R, then we have that the Attention operator is approximated as:

$$\widehat{Att}_{\leftrightarrow}(Q, K, V) = \widehat{D}^{-1}(Q'((K')^T V)) \tag{25}$$

where $\widehat{D} = diag(Q'((K'^T 1_L))$, We can see how:

- $(K')^T V$ is an operation between matrices with shape $(r \times L)$ and $(L \times d)$, meaning that it requires $\mathcal{O}(Lrd)$ operations.
- $Q'((K')^T V)$ is an operation between a $L \times r$ matrix and a $r \times d$ matrix, meaning that it requires once more $\mathcal{O}(Lrd)$ operations.

Notice how $\widehat{D}^{-1}$ is a diagonal matrix and, as a result, there exist optimizations to avoid the quadratic cost on L.

To summarize, we have obtained a method that calculates the Attention operator in $O(Lrd)$ steps. However, as stated at the beginning of the report, the paper claims a cost equal to $O(Ld^2 \log d)$. This is because in our setup, $r = m$. The paper shows how the optimal value for m, for which the softmax function is approximated accurately, is equal to $d \log d$, from which the original statement follows as a consequence.

2.2.2 Kernels for functions different than softmax

The expressive power of the framework described in the last section does not limit itself to the softmax operator. This is of particular importance as it means that we can replace the softmax with many other functions, which makes the Attention operation much more flexible.

In this work, we will only see the ReLU kernel. It is obtained with the following setup:

- h is the constant function always equal to 1
- f_1 is the ReLU function
- No renormalization by $\sqrt[4]{d}$ or $\sqrt{d}$ is employed

meaning that we have:

$$\phi(x) = (ReLU(\frac{x}{\sqrt{m}}\omega_1^T), \ldots, ReLU(\frac{x}{\sqrt{m}}\omega_1^T)) \tag{26}$$

Similarly to the softmax kernel, each element in the output vector is summed to a small constant in the order of 10^{-5} for numerical stability.

2.2.3 Random features redraw

The paper also shows how redrawing the random orthogonal features matrix periodically improves training by avoiding "unlucky" projections on Q and K which might degrade training. In our implementation, the number of steps before redrawing is set to 1000.

3 Implementation details

In this section, we will discuss the details of the realized implementation.

3.1 Code structure

The framework itself is relatively small and is composed of 4 python files.

3.1.1 Unet.py

It contains the class which defines the neural network used to predict the noise on an image. This is a slightly modified version of the original implementation's code. This choice was motivated by two facts: first of all, the paper itself is quite vague on the details of the implementation, which resulted in an online research for details, which has shown how different available implementations of the DDPM model use different Unet architectures. The original paper's implementation was the best one among the ones consulted. Second, our solution will be compared to the original implementation for a comparative study. Using at least the same Unet architecture, and changing only the Attention blocks with Performers (as well as the training and sampling loops) allows for a fairer comparison.

The core of the file is the Unet module. Its constructor contains the following parameters:

- x_sz: the horizontal size of the input pictures. Since the implementation currently supports only squared images, x_sz is also the vertical size of the image.
- init_dim: the number of channels of the output of the initial convolution operation. If None, it defaults to x_sz.
- dim_mults: and n-uple which defines the number of output channels for each layer. For instance: if we pass dim_mults = (2, 4, 8), the Unet will be made by 3 layers whose outputs will have, respectively, a number of channels equal to 2*init_dim, 4*init_dim, and 8*init_dim. The Unet decompressor's layers follow the same order, but it's reversed and at the end of it we have a Conv2D which converts the number of channels back to ⟨channels⟩.
- channels: number of channels in the input images (1 for things such as MNIST, 3 for RGB images).

- resnet_block_groups: groups in the group norm employed by the ResNet blocks.
- time_dim: time embeddings dimensionality.
- h: how many Attention heads to use.
- head_sz: the size of a single Attention head.
- att_type: which type of Attention to use: 'SDP' for the standard scaled dot product, 'FAVOR_SDP' for the softmax kernel, or 'FAVOR_RELU' for the ReLU kernel.
- m: number of random orthogonal features (if set to None, it defaults to head_sz*log(head_sz)).
- redraw_steps: after how many steps the orthogonal matrix should be resampled.
- device: 'cpu' for CPU training, 'cuda' for GPU training.
- use_original: if True, use the original Attention code instead of the one in Attention.py. Overrides att_type, m and redraw_steps.

The constructor then proceeds to create the various blocks of the compressor, bottleneck, and decompressor. The only method made available by the class is its override of Pytorch's forward function. It takes a batch of 3D tensors in input which represents the various images and outputs a tensor of the same shape describing the predicted noise of the various images.

The class includes the code of the original Attention modules. The bottleneck uses standard Attention while the remaining modules employ either the Performer module or, if use_original is set to True, a linear approximation of the Attention function named Linear Attention, by Katharopoulos et al. [6]. It is essentially a Performer highly optimized for $\phi(x) = elu(x)+1$, where $elu(x)$ is the shifted-eLU function from Clevert et al. [9], and no random orthogonal features are used.

3.1.2 MultiHeadAttention.py

The class MultiHeadAttention implements, as its name suggests, the multi-head Attention mechanism, which is one of the blocks included in the Unet. This class itself does not apply the Attention operator (for that, see Attention.py), but performs the initial linear transformations to the inputs before rearranging them to a shape compatible with the Attention operation. The result of the Attention operator is then reshaped once more into a 3D tensor before applying the final linear operator.

The original implementation performs, as the linear operation, a 2D convolution instead of just flattening the image into a 2D tensor and multiplying it by a weight matrix. It also does not combine the input matrix with a positional embedding matrix. We have done the same in this implementation.

The constructor has the following parameters:

- device: 'cpu' for CPU training, 'cuda' for GPU training.
- dim: number of input (and output) channels.
- d_model: number of channels after applying Conv2D (the output will still be of ⟨dim⟩ channels).

- h: Attention heads. d_model must be a multiple of it.
- bias: whether the convolution operations should use a bias term.
- att_type: 'SDP' for standard Attention, else FAVOR_SDP or FAVOR_RELU.
- m: number of orthogonal random features for FAVOR+.
- redraw_steps: number of steps before we redraw the orthogonal random features.

3.1.3 Attention.py

This file contains the definition of the classes SDPAttention and FAVORplus, which respectively implements the standard Scaled Dot Product Attention with softmax and the Performer.

Both classes contain a forward method that implements the mechanisms described in the previous sections. FAVORplus also includes the methods relu_kernel and softmax_kernel which calculates, given a vector x, the ϕ functions for both the softmax and the ReLU kernels. It also includes the method redraw_proj_matrix which returns a matrix in $\mathbb{R}^{m \times d_{head}}$ and the rows are orthogonal between each other in blocks of size d_{head}.

SDPAttention's constructor contains the following parameters:

- device: 'cpu' for cpu training, 'cuda' for GPU training.
- d_model: used to assert d_{head}.
- h: number of Attention heads.
- dropout_p: many Attention mechanisms (including BERT's implementation) applies dropout to the Attention matrix. There is not a lot of documentation about why they do so, but it helps regularize a little.

FAVORplus adds the following parameters (with the exception of dropout_p):

- kernel_type: 'FAVOR_SDP' or 'FAVOR_RELU'.
- num_stabilizer: a small constant in the order of 10^{-5} used for numerical stability.
- redraw_steps: number of steps before we redraw the orthogonal random features.
- m: the number of random orthogonal features.

3.1.4 Diffusion.py

This file implements the class Diffusion, which is what the user should effectively use. Other than the constructor method, it implements the following methods:

- fit: the training method, which implements algorithm 1. It receives an instance of the class torch.utils.data.Dataset as the training set and, optionally, another instance of the same class as a validation set. It implements the training algorithm and logs various information such as step loss, epoch loss, iterations per second, and the estimated time left before the end of the epoch.

- sample: it samples an image using algorithm 2. the parameter verbose_steps, when higher than zero, tells the method to print every ⟨verbose_steps⟩ steps the intermediate result of the sampling.
- debug_sample: used only for debugging. Given the index of a specific sample in the input dataset, a timestep sampling_time and a value verbose_module, it gets the sample with the requested index x_{index}, calculates $(x_{index})_t$, and proceeds to denoise it. If verbose_module is not zero, it prints the intermediate results every ⟨verbose_module⟩ iterations.
- plot: when called after training, it plots the training and the validation loss.

The constructor's parameters are:

- n_channels: the number of channels in the input image (3 for RGB, 1 for MNIST).
- x_sz: the image size in terms of horizontal pixels (at the moment the class only supports square images so it's also the number of columns).
- att_type: 'SDP' for standard Attention. Otherwise, 'FAVOR_SDP' or 'FAVOR_RELU' for the respective Performers.
- m: in the case of FAVOR+, the number of random orthogonal features to use.
- redraw_steps: number of steps before random features redraw.
- init_dim: number of channels in the input after the first Conv2D of the Unet. If None, it defaults to x_sz.
- h: number of Attention heads.
- head_sz: Attention heads' size.
- sample_iters: T in the DDPM paper.
- t_size: time embedding dimensionality.
- device: 'cpu' for training with the CPU, 'cuda' for GPU training.
- b_1: β_1 value.
- b_T: β_T value.
- opt_data: a wrapper for the optimizer and its relative hyperparameters (see the class OptimizerData in Diffusion.py). Useful for instantiating the optimizer after the parameters have been instantiated.
- sched_data: a wrapper for the scheduler and its relative hyperparameters (see the class SchedulerData in Diffusion.py). Useful when the optimizer gets instantiated in a second moment and we cannot pass it to the scheduler immediately.
- loss: the loss function. Usually, it is torch.nn.MSELoss().
- batch_size: the batch size.

- data_slice_tr: if > 0, the training set will be made by the first $\langle$data_slice_tr$\rangle$ samples of the training set given at the fit call.
- data_slice_vl: if > 0, the validation set will be made by the first $\langle$data_slice_vl$\rangle$ samples of the training set given at the fit call.
- n_iters: training iterations.
- verbose: if equal to zero, no logs will be printed except for the progress bar.
- dim_mults: see 3.1.1 for further informations.
- resnet_block_groups: number of groups in the group norm used inside the ResNet blocks.
- use_original: if True, uses the original Linear Attention code as in the original paper's implementation.
- clipnorm: gradient clip norm. Not particularly important.

3.2 Requirements

The framework has been realized in Python, and uses the following libraries:

- Pytorch
- math
- numpy
- einops: most of its use finds its place in the Unet implementation, which is strongly based on the original paper's implementation and it is used for matrix multiplications and shape rearrangements. The only thing outside of it which utilizes it is the Attention mechanism, which employs the method "rearrange" as it requires fewer instructions than pure PyTorch permutations. It is also slightly more efficient, which allows for a fairer comparison against the original implementation's performance.
- matplotlib: used for the various graphs.
- tqdm: used to show the model's progress compactly.
- torchvision: used for downloading the datasets and for preprocessing the data through the "transform" class.
- functools: used by the Unet's original implementation for the "partial" utility.

4 Experiments

Because of computational limitations, the experiments have all been executed on Kaggle's cloud GPU service. The Jupyter notebooks with the code and associated results have been included in the project. The amount of VRAM available was of 16GBs.

4.1 Performance

This subsection will briefly show the improvements in terms of speed and memory against the standard Attention mechanism as implemented in the SDPAttention class, while also comparing it against the Linear Attention mechanism. The tests have been executed while training on the full MNIST dataset.

To ensure as much fairness as possible, the hyperparameters are the same for every execution. The Unet code is the same with the exception of the Attention module.

The hyperparameters are as follows:

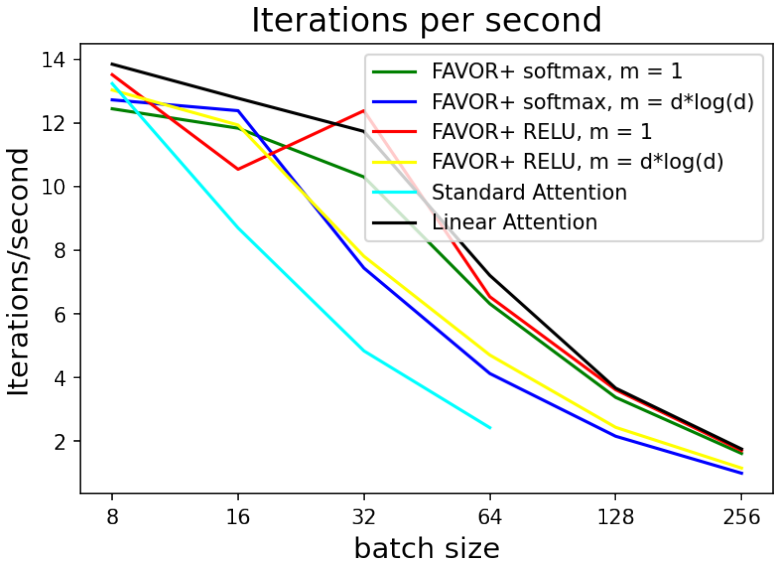
- Learning rate: 0.001.
- Blocks for the group norm in the ResNet blocks: 8.
- Gradient calls before a random orthogonal features' matrix redraw: 1000.
- dim_mults (see section 3.1.1 for the full description): (1, 2, 4).
- Attention heads size: 32.
- Number of Attention heads for each block: 4.

Six different models have been tested:

- FAVOR+ with softmax and m = 1
- FAVOR+ with softmax and m = None
- FAVOR+ with ReLU and m = 1
- FAVOR+ with ReLU and m = None
- Standard softmax
- Linear Attention

This allows us to compare FAVOR+ with either a minimalistic approximation ($m = 1$) or as accurately as we can as long as $m << L$. In figure 1, we will show both the memory consumptions and the iterations per second executed by every model with a batch size of 8, 16, 32, 64, 128, and 256.

As we can see from figure 1, the amount of memory used by FAVOR+ is much lower than the amount used by the standard Attention mechanism and is also capable of training a model with higher batch sizes without going out of memory. The speed is also massively improved. When compared to the Linear Attention mechanism, the latter is faster on average, although increasing the batch size makes the difference between it and the FAVOR+ instances with $m = 1$ almost negligible. The ReLU kernel trained with $m = 1$ wins in terms of memory consumption by a decent margin, while the softmax kernel with the same amount of orthogonal features performs virtually the same as the Linear Attention mechanism.



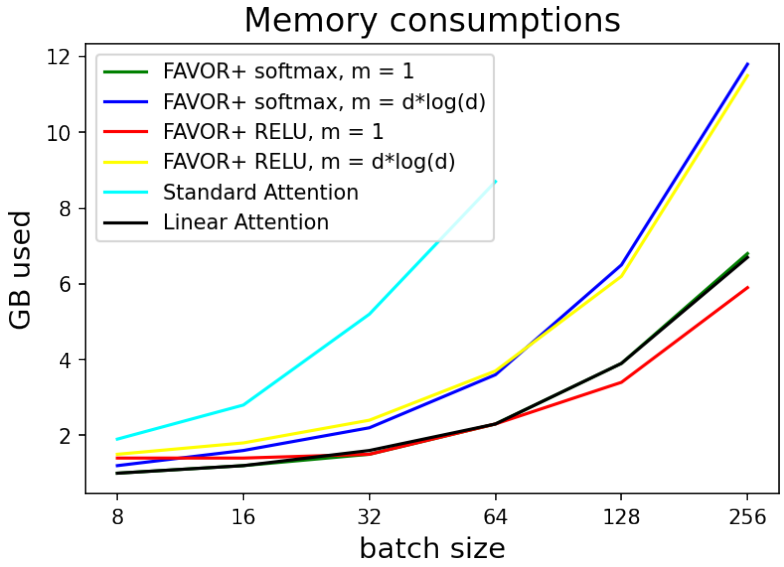
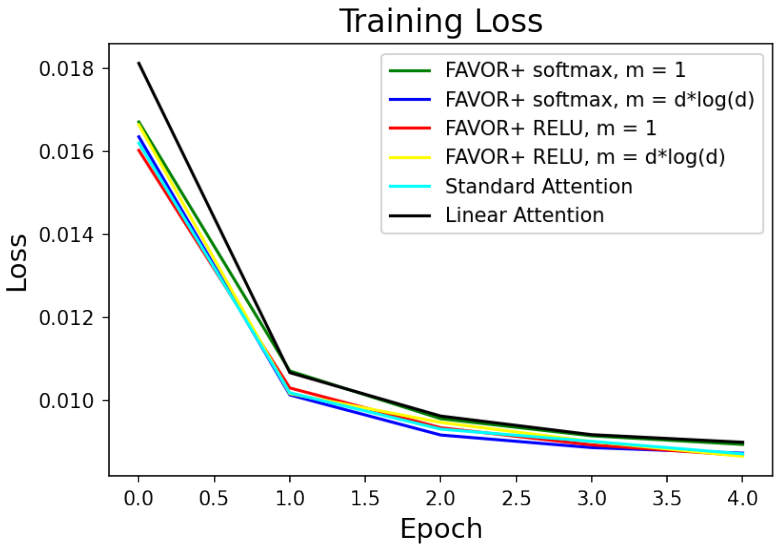


Figure 1: On the left: the number of training steps per second as a function of the batch size. On the right: the amount of VRAM utilized as a function of the batch size. Note how the standard Attention mechanism went out of memory with batch sizes higher than 64.

4.2 Comparison with the original implementation

Using the same hyperparameters and models of subsection 4.1, we have trained each model for 5 epochs with a batch size of 32, on both the MNIST and the FashionMNIST datasets. In the latter case, the learning rate has been decreased from 0.001 to 0.0001. The experiments are shown in the Jupyter Notebooks named "mnist-fulltest.ipynb" and "fashionmnist-fulltest.ipynb". The losses are shown in figure 2 and figure 3.



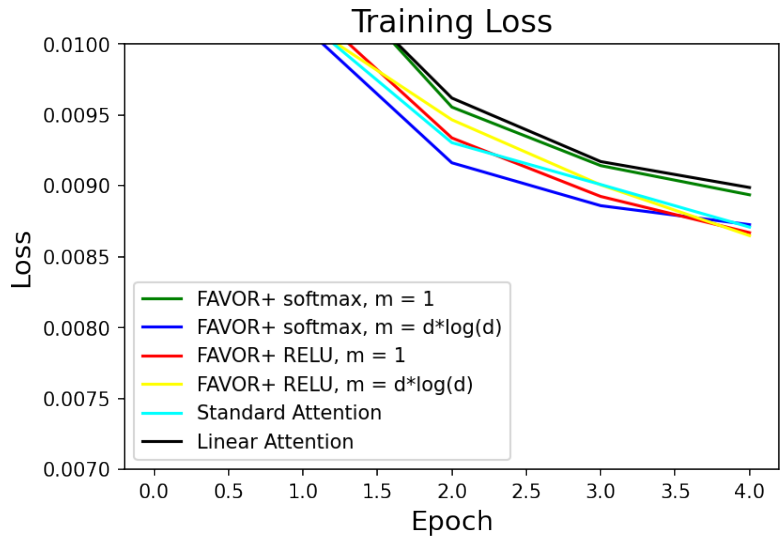


Figure 2: On the left: the full loss graph for MNIST. On the right, the same graph but zoomed in for a better comparison.

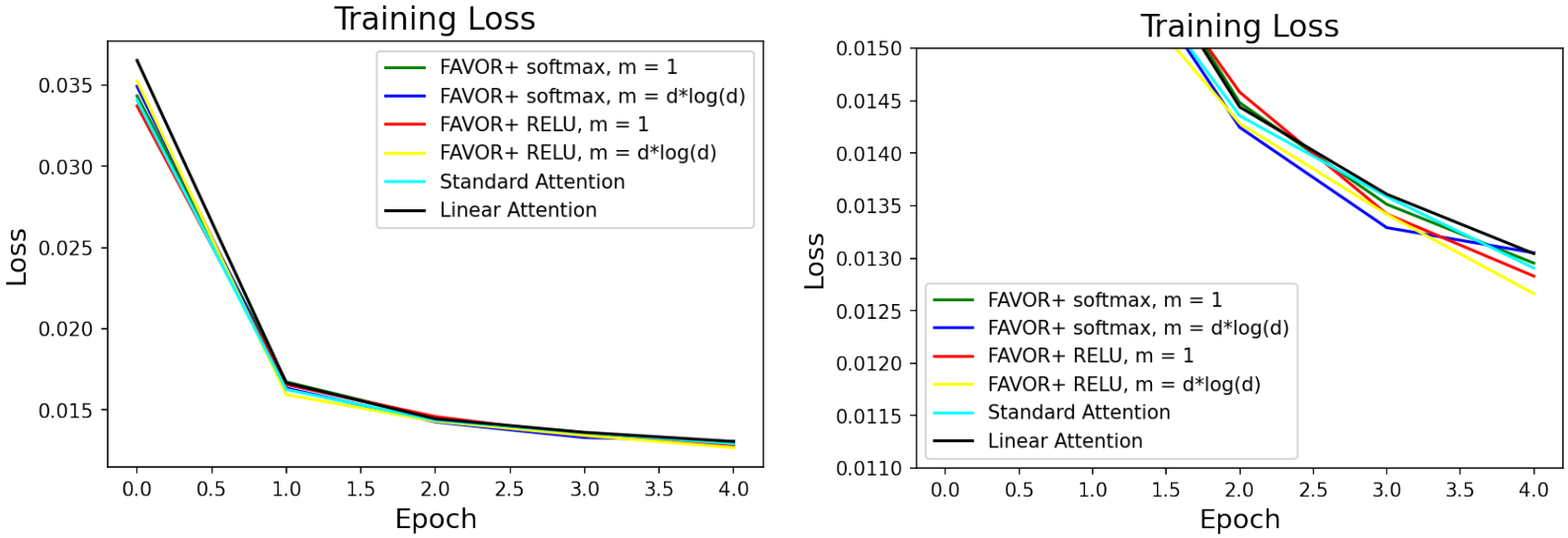


Figure 3: On the left: the full loss graph for FashionMNIST. On the right, the same graph but zoomed in for a better comparison.

As we can see in MNIST's case, there are two big clusters, the first one being composed by the Linear Attention approximation and FAVOR+ SDP with $m = 1$, and everything else being the second group, with the latter obtaining lower loss scores with no real winner. It is interesting to note how m makes a huge difference in the softmax kernel while ReLU performs well with both values.

The results are much closer for FashionMNIST, but the ReLU kernel still seems to be the winner.

Finally, in figure 4 and 5 there are 10 non-cherry-picked samples obtained from each one of the trained models, for both datasets.

In MNIST's case, it is difficult to determine a real winner, as most of the images are convincing enough. An additional test was performed by training ReLU for 30 epochs with $m = 16$ with MNIST, which obtained even better results in visual terms. The results are shown in figure 6.

As for FashionMNIST, the Performers seems to behave generally better, while the Linear Attention mechanism performs visibly worse (although some of its samples still look relatively good)

5 Conclusions

In this work, we have implemented a DDPM, and we have tested how well the novel Performer mechanism improves training in both memory consumption, training times, and results. The summary of our results is the following:

- The usage of the Performer is well justified when compared to the standard Attention mechanism, as it is capable of matching the results of the latter at a fraction of the cost in terms of training time and memory.
- Performers can require as much to less memory than Linear Attention in the case of $m = 1$ and still archive equal to better losses.

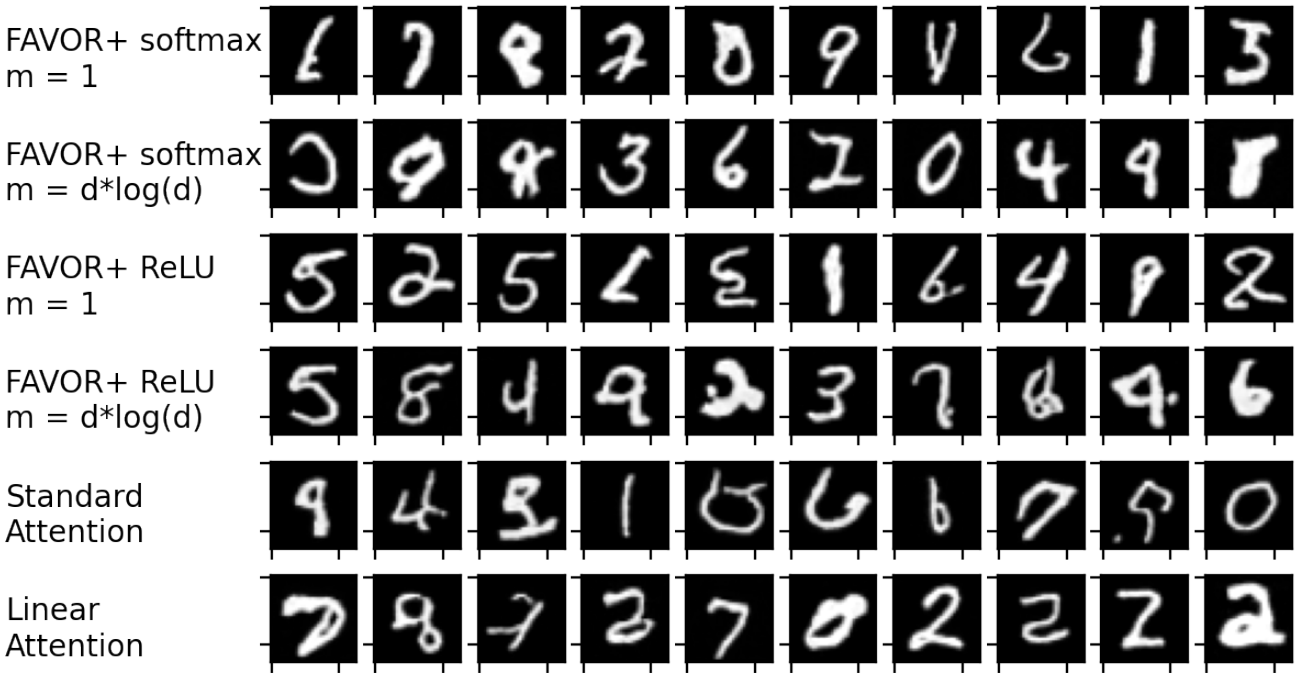


Figure 4: 10 random samples (non-cherry-picked) for each model trained on the MNIST dataset.

Figure 5: 10 random samples (non-cherry-picked) for each model trained on the FashionMNIST dataset.

- Linear Attention is generally faster than the Performers, although ReLU with $m = 1$ (which is conceptually close to it) comes very close and seems to produce slightly better images using slightly less memory.

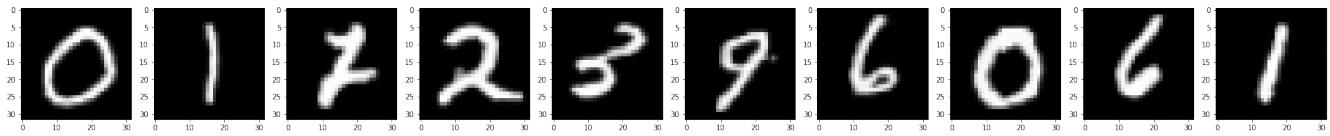


Figure 6: 10 random samples (non cherry-picked) from a model trained using FAVOR+ with a ReLU kernel, 30 epochs and $m = 16$.

Studies for better kernels for the FAVOR+ mechanism might further improve the results.

References

[1] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *CoRR*, abs/2006.11239, 2020.

[2] Prafulla Dhariwal and Alex Nichol. Diffusion models beat gans on image synthesis. *CoRR*, abs/2105.05233, 2021.

[3] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *CoRR*, abs/1605.07146, 2016.

[4] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[6] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. *CoRR*, abs/2006.16236, 2020.

[7] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamás Sarlós, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy J. Colwell, and Adrian Weller. Rethinking attention with performers. *CoRR*, abs/2009.14794, 2020.

[8] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007.

[9] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv: Learning*, 2016.