GPT Project Part 1 Report

Paul Croizet
Emile De Vos
Yassine Hajem
Corentin Basso
Rémi Guerafi
Julien Wu

Département Sciences du Numérique - Troisième année
2024-2025

Contents

1 Introduction **3**
- 1.1 Project . . . 3
- 1.2 Organization . . . 3

2 Transformers **3**
- 2.1 Preprocessing . . . 3
  - 2.1.1 Database . . . 3
  - 2.1.2 Tokenization . . . 3
  - 2.1.3 Data preparation . . . 4
- 2.2 Model . . . 6
  - 2.2.1 Embedding . . . 6
  - 2.2.2 Attention block . . . 7
  - 2.2.3 Other important layers . . . 9
- 2.3 Notations and Transformations . . . 11
- 2.4 Inference . . . 11
- 2.5 Example of nanoGPT Processing Chain . . . 12

3 nanoGPT **14**
- 3.1 nanoGPT structure . . . 14
  - 3.1.1 Transformer block . . . 14
  - 3.1.2 Attention block . . . 14
  - 3.1.3 MLP block . . . 15
  - 3.1.4 Overall structure . . . 16
- 3.2 Training . . . 16
  - 3.2.1 How it works . . . 16
  - 3.2.2 Loss . . . 16
  - 3.2.3 Optimizer . . . 17
  - 3.2.4 Training parameters . . . 19

4 Optimal Control **20**
- 4.1 Introduction . . . 20
- 4.2 Definitions . . . 20
- 4.3 ResNet and the Optimal Control Formulation . . . 20
- 4.4 Pontryagin Maximum Principle (PMP) in Deep Learning . . . 21
  - 4.4.1 PMP formulation . . . 21
  - 4.4.2 p^0 determination . . . 22
  - 4.4.3 Application of the PMP . . . 22
- 4.5 Discretization . . . 23
- 4.6 Backpropagation . . . 23
- 4.7 The algorithm . . . 24

5 Results **25**
- 5.1 NanoGPT results . . . 25
- 5.2 Optimal Control results . . . 26

6 Conclusion **27**

7 Resources **28**

1 Introduction

1.1 Project

The initial objective of this project was to recreate a GPT model with optimal control. However, we have since shifted our focus to gaining a deeper understanding of the underlying concepts, from the architecture of transformers to the role of optimal control in machine learning. This study begins with an exploration of transformer models, analyzing the mathematical principles behind each component. We then examine nanoGPT as a simplified implementation to reinforce our understanding. Finally, we investigate how optimal control techniques can be applied to ResNets, a family of neural networks similar to large language models, in order to eventually apply these methods to LLMs.

1.2 Organization

For this project, we made two groups of three. The mathematics group is calculating the complete function of nanoGPT model in order to formulate it as an optimal control problem. The computing group is coding nanoGPT in Julia and will then implement the optimal control problem. We use Notion to add and assign the tasks for the project and share code and files on Github.

2 Transformers

2.1 Preprocessing

Before training any machine learning or deep learning model, data preprocessing is a critical step that ensures the quality and reliability of the input data. Raw data is often noisy, inconsistent, or incomplete, which can negatively impact model performance. By cleaning, transforming, and structuring the data appropriately, we can improve learning efficiency and accuracy.

2.1.1 Database

For this project, we are using a simple dataset called *tiny_shakespeare*, created by the developer of nanoGPT. It is straightforward and contains 167205 lines of theatrical dialogue.

2.1.2 Tokenization

Tokenization is one of the most important part of machine learning, and have a great impact on the performance of the final result. It is the part that consist by dividing the database into multiple tokens, and after transforming these tokens into number in order to have a vector of number readable by the model. An important point is that to make a translation between a token and a number, we need to define a *vocabulary*, which can be seen as a bijective function that associate one token to one number. The choice of the way we tokenize matter and can impact on :

- **The performance**
- **Number of Parameters**: The higher the number of tokens, the more parameters there are. This is related to the dense layer in the output.
- **The computational time**

Let us take an example of the following sentence : *I love doing machine learning.*, and let us try to tokenize it with differents methods.

Character tokenization The idea is simple: each character is a different token. So with our example, the induced tokezination will be :

$$[\mathrm{I},\ ,\mathrm{l},\mathrm{o},\mathrm{v},\mathrm{e},\ ,\mathrm{d},\mathrm{o},\mathrm{i},\mathrm{n},\mathrm{g},\ ,\mathrm{m},\mathrm{a},\mathrm{c},\mathrm{h},\mathrm{i},\mathrm{n},\mathrm{e},\ ,\mathrm{l},\mathrm{e},\mathrm{a},\mathrm{r},\mathrm{n},\mathrm{i},\mathrm{n},\mathrm{g},.]$$

With this tokenization, the benefits are :

- The number of tokens is limited to the number of characters that exist in a language. For example, with our dataset, the number of tokens is around 70, as it includes uppercase letters and punctuation.

- This method is the simplest to implement as the *String* is usually seen as vector of *Char*.

But on the other hand, the result are not the best one, because the network need to make a lot of link between all the tokens to write good and complex response.

Word tokenization In this type of tokenization, each word is considered a token. Using our example, the tokenization will be as follows:

$$[\text{I}, \text{love}, \text{doing}, \text{machine}, \text{learning}]$$

It induced the following points :

- First, this method results in a larger number of tokens compared to the character-based method. This leads to an explosion in the number of parameters in our network, which can increase the training time.
- The distribution of tokens may be unbalanced within the dataset, with some words being highly represented and frequently encountered by the model, while others are poorly represented and rarely seen. This issue can result in the model having a limited vocabulary.

Radical tokenization This tokenization is a mix of the previous methods. In many languages, we can observe that certain parts of words are often repeated, such as the endings of conjugated verbs or common suffixes like 'tion'. The idea is to divide words by splitting them into their roots and affixes. With our example, the induced tokenization will be :

$$[\text{I}, \ , \text{love}, \ , \text{do}, \text{ing}, \ , \text{machine}, \ , \text{learn}, \text{ing}]$$

This approach helps reduce the total number of tokens while still capturing meaningful linguistic patterns.

2.1.3 Data preparation

Now, we have some data and a tokenization method. However, a machine learning model can only process numerical values in vectors of fixed size. The first step is to find all the existing tokens in the dataset and store them into a long vector called V_{token} of dimension m containing all the tokens. Next, we need to define the *vocabulary*, which, as previously mentioned, is a bijective function that associates each token with a unique number. By using this function we have :

$$V = \text{vocabulary}(V_{token})$$

Where $V = \{1, \dots, m\}$ denote the representation in integer of the tokens . An efficient way to implement this *vocabulary* is by using a dictionary structure. Using this dictionary, we can transform all of our data into token vector and then into an integer vector, denoted as $\mathbf{X}$.

At this point, we need to split this vector into multiple vectors of the input sequence size. This size, denoted as $\mathbf{n}$, can either be fixed at a specific value or determined using statistical methods on the dataset. Here, we choose to fix this value between 64 and 200. Once n is defined, we split our initial vector into sub-vectors of size n without overlapping, discarding the last sub-vector if it does not match the size n. This leads us to the following vector :

$$X_{train} = (\mathbf{X}^1, \cdots, \mathbf{X}^{K_{train}})$$

Where :

- X_{train} have size (K_{train}, n)
- $K_{train} = \lfloor \frac{len(X)}{n} \rfloor$.
- $\mathbf{X}^k = (X_i^k)_{i \leq n}, \ X_i^k \in V$

We now need to construct the vector $Y_{train} = (\hat{\mathbf{Y}}^1, \cdots, \hat{\mathbf{Y}}^{K_{train}})$ of the same size as X_{train} where $\hat{\mathbf{Y}}^k = (\hat{Y}_i^k)_{i \leq n},\ \hat{Y}_i^k \in V$.

This vector is the ground-truth and it is constructed based on how we train the model. For models belonging to the GPT family, the target vector is the same as the input vector but shifted by one position to the left. For example :

- If $\mathbf{X}^1 = [1, 2, 3, 2, 3]$
- Then $\hat{\mathbf{Y}}^1 = [2, 3, 2, 3, 1]$

These preprocessing steps lead us to two vectors of the same dimensions: X_{train} and Y_{train}. The final step is to transform each value in $\hat{\mathbf{Y}}^k$ into a one-hot vector denoted $\hat{\mathbf{y}}^k$. In fact, models belonging to the GPT family do not predict a single index for each neuron in the output layer; instead, they predict a probability distribution representing the likelihood of each word being the correct one. Thus, each neuron in the output layer predicts a probability distribution of the vocabulary's size. A one-hot vector is a vector with zeros everywhere except for a single position containing a one. Therefore, for each value in $\hat{\mathbf{Y}}^k$, we replace it with a vector that has a one at the index corresponding to that value leading us to :

$$\forall\ \hat{Y}_i^k \in V,\ i \leq n,\ k \leq K_{train} \Longrightarrow \hat{y}_i^k \in \Omega(V), \quad \hat{y}_i^k \in \{0, 1\}^m$$

Where $\Omega(V)$ is the probability space associated with V.

Then we define $\hat{\mathbf{y}}^k$ as :

$$\hat{\mathbf{y}}^k = [\hat{y}_1^k, \ldots, \hat{y}_n^k]$$

Let us do an example : if $m = 3$, and $\hat{\mathbf{Y}}^1 = [2, 3, 2, 3, 1]$, the resulting value will be :

$$\begin{aligned} \hat{\mathbf{y}}^1 &= [\hat{y}_1^1,\ \hat{y}_2^1,\ \hat{y}_3^1,\ \hat{y}_4^1,\ \hat{y}_5^1] \\ &= \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \end{aligned}$$

2.2 Model

2.2.1 Embedding

Embedding techniques are crucial in transformer-based language models like nanoGPT. It comes after the tokenization phase 2.1.2 and aim to convert a piece of information, for example, text, images, audio, etc. into continuous vector representations of a size **d**. Given the text "Hello nanoGPT !", the embedding could have a vector representation with a list of $d = 5$ numbers $[0.5, 0.85, 0.06, 0.2, 0.008]$.
In our case, we want to transform our input sequence $\mathbf{X}^k$ into a matrix $\mathbf{x}^k$. We can see the embedding part as a function defined as:

$$\text{Embedding} : \begin{array}{ccc} \{1, \dots, m\}^n & \longrightarrow & \mathbb{R}^{d \times n} \\ \mathbf{X}^k & \longmapsto & \mathbf{x}^k \end{array}$$

This matrix contains the meaning of the sequence. More precisely, if we have the i-th token from the k-th sequence $X_i^k \in V$, then we have :

$$X_i^k \xrightarrow[embedding]{} x_i^k = \begin{bmatrix} x_{1,i}^k \\ \vdots \\ x_{d,i}^k \end{bmatrix} \in \mathbb{R}^d$$

.

We can do different things using the embedding vector such as calculating the distance between two embeddings to determine how well the meaning of sentences matches. Let's go through different types of embeddings:

1. **Word Embedding**
   When dealing with text in machine learning, after the tokenization phase, we need a way to represent tokens numerically by transforming them into vectors. There exist plenty of ways to do achieve this transformation but only two type of word embedding exist : trainable and non-trainable. In the following, we will go through an example of each type.

   - **One-Hot Encoding**
     One-hot encoding[1] is the simplest form of word representation. Each word in the whole vocabulary is represented by a binary vector called **one-hot vector**. Their length is equal to the size of the vocabulary and only one position is set to 1 all the others are equal to 0. For instance, let us have a vocabulary $V_{token} = \{I, love, machine, learning\}$. In order to do one-hot encoding, each word could get their position in the vocabulary as their index. Therefore, we have the following result :

     $$I \longrightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad love \longrightarrow \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad machine \longrightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad learning \longrightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

     Most of the times, one-hot vectors are often not used as embeddings but as input in order to learn dense embeddings.
   - **Learnable Word Embedding**
     Unlike one-hot encoding, learnable word embedding is trainable and improve over time by learning the best vector representation through backpropagation. It represents tokens into dense numerical vectors that are initialized randomly and updated during the model training. Theses embeddings are often used in Transformer-based models (it is also the one used in nanoGPT). After training, words with similar meanings have closer embeddings. We did not mention the dimension of the embeddings d, but their size is fixed and in the case of nanoGPT it is equal to $n\_embd = 768$. The reason their size is fixed[2] is because it will help the transformers process all embeddings in an uniform way but also because of mathematical operations used in the transformer such as matrix multiplication.

2. **Positional Embedding**
   Positional embeddings are added to give the model information about the position of each token in a sequence. Without this, a transformer model would treat all tokens as unordered. And just like the word embeddings, there exists trainable and non-trainable method. In the following, we will go through an example of each type.

- **Sinusoidal Positional Encoding**
  In the original article about transformers[2], sinusoidal positional encoding is mentionned. It is described with the following formulas :

$$PE_{2l,i} = sin(\frac{i}{10000^{\frac{2l}{d}}}), \quad PE_{2l+1,i} = cos(\frac{i}{10000^{\frac{2l}{d}}}), \text{ where } \quad l \in \mathbb{N}, 2l+1 < d$$

  with l the dimension, i the position of the token in the sequence (it starts at 0 for the first token) and d the dimension of the embeddings. We obtain a general formula of P :

$$P_i = \begin{bmatrix} PE_{2\times 0,i} \\ PE_{2\times 0+1,i} \\ \vdots \\ PE_{d-1,i} \end{bmatrix}$$

  By alternating between sine and cosine, the positional encoding creates smooth-wave like encoding and positions far apart have distinct encoding while close positions have similar one. In order to be more clear, let us have an example : if $d = 4$ and the word is the 4th one ($i = 0$ for the first word) and whatever the sequence it belongs to then its sinusoidal positional encoding would be :

$$P_3 = \begin{bmatrix} PE_{0,3} \\ PE_{1,3} \\ PE_{2,3} \\ PE_{3,3} \end{bmatrix}$$

  The 10,000 constant in sinusoidal positional encoding balances short-range and long-range dependencies in transformer-based models. It ensures a smooth frequency distribution across embedding dimensions, allowing the model to distinguish both nearby and distant token positions.

- **Learned Positional Embeddings**
  Learned positional embeddings is a trainable alternative to fixed embeddings method such as sinusoidal positional embeddings. Instead of using fixed mathematical functions, the model learns the positional information throughout the training phase that is captured in a weight matrix $P \in \mathbb{R}^{d\times n}$ where the i-th column, denoted as P_i, is the positional embedding associated with the i-th word in the sequence. And just like learnable word embedding, the positional embeddings starts randomly and is updated through backpropagation. Even though, this alternative is able to learn different properties such as the words at the end often provides conclusion, it cannot generalize to longer sequences than it was defined for.

In our project, we are focusing on one specific model called nanoGPT, it employs Learnable Word Embedding due to their flexibility and ability to optimize within the model's architecture. For the i-th token of the k-th sequence, x_i^k assigned a trainable embedding vector E_i^k. It also adopts Learnable Positional Embeddings as they allow better performance than predefined functions. For the i-th token of the k-th sequence, x_i^k assigned a trainable embedding vector P_i.

The final input embedding $x_i^k = E_i^k + P_i$. This summation ensures that both token identity and positional context are encoded efficiently.

2.2.2 Attention block

The objective of Self-Attention is to compute how much a word is influenced by each of his neighbors. Self-attention goal is to constructs meaningful representations by incorporating information from all words in the sequence. Let $x_i^k = \begin{bmatrix} x_{1,i}^k \\ \vdots \\ x_{d,i}^k \end{bmatrix} \in \mathbb{R}^d$, with $i \in \{1, \ldots, n\}$ the embeddings vectors associated with each X_i^k. But for better clarity, we will consider only consider one sequence and therefore we will

use x_i instead of x_i^k. If there was multiple sequences, instead of using matrices, we would use tensors in order to compute in parallel the different attention score from the different sequences.

Let $x = (x_i)_{i \leq n} \in \mathbb{R}^{d \times n}$, each $x_i \in \mathbb{R}^d$ represents the embedding of X_i and has 3 associated vectors : the query q_i which represents what the token is looking for in other tokens, the key k_i which represents what information the token has to offer and the value v_i which represents contains the actual information being passed forward. If there are H heads, then each x_i will have H sets of 3 vectors $\{q_i^1, k_i^1, v_i^1\}, \ldots, \{q_i^H, k_i^H, v_i^H\}$. Let h be the attention head we are inspecting. What follows next are computed in parallel for each head.

Step 1 : The self-attention mechanism uses three trainable weight matrices W_q^h, W_k^h, W_v^h. These matrices are multiplied by each x_i in order to project the input into query, key and value components respectively :

$$q_i^h = W_q^h x_i \in \mathbb{R}^{d_k}$$
$$k_i^h = W_k^h x_i \in \mathbb{R}^{d_k}$$
$$v_i^h = W_v^h x_i \in \mathbb{R}^{d_v}$$

And obviously, we have $W_q^h, W_k^h \in \mathbb{R}^{d_k \times d}$ and $W_v^h \in \mathbb{R}^{d_v \times d}$. The three weight matrices are usually initialized randomly[3] and then the weight are updated throughout the training process.

Step 2 : For each x_i , we calculate its score with each x_j with $j \in \{1, \ldots, n\}$ (himself included) and it will represent how much attention x_i should pay attention to x_j. To ensure stable gradients, we scale the scores by the square root of the key dimension, denoted as d_k, before applying the softmax function.

$$a_{i,j}^h = \frac{exp(\frac{(q_i^h)^T k_j^h}{\sqrt{d_k}} + M_{i,j})}{\sum_{l=1}^n exp(\frac{(q_i^h)^T k_l^h}{\sqrt{d_k}} + M_{i,l})}$$

$$\text{with } M_{i,j} = \begin{cases} 0, & \text{if } j \leq i \quad \text{(allow attending to self and past tokens)} \\ -\infty, & \text{if } j > i \quad \text{(mask future tokens)} \end{cases}$$

The presence of the mask $M = (M_{i,j}) \in \mathbb{R}^{n \times n}$ is optional but ensures that predictions will only depend on past and present tokens. For instance for x_i, the prediction will only depend on x_j for $j \in \{1, \ldots, i\}$.

Step 3 : The final output representation $z_i^h \in \mathbb{R}^{d_v}$ for each word is computed as a weighted sum of all value vectors:

$$z_i^h = \sum_{j=1}^n a_{i,j}^h v_j^h$$

And finally, if we concatenate all the final output representation for the head h, we have the attention associated with this head :

$$Att_h(x) = (z_i^h)_{i \leq n} \in \mathbb{R}^{d_v \times n}$$

Final step : In practice, transformers use multi-head attention, where multiple self-attention heads run in parallel, each learning different aspects of the input. Each attention head computes its own self-attention output $Att_h(x)$ with $h \in \{1, \ldots, H\}$ where H the number of heads. And the result is obtained by concatening all the different attention and then projecting it back to the original dimension :

$$\text{MultiHead}(x) = ([Att_1(x)^T, \ldots, Att_H(x)^T]W_o)^T \in \mathbb{R}^{d \times n}$$

with $W_o \in \mathbb{R}^{Hd_v \times d}$ another trainable weight matrice that projects the output back into the model's original dimensionality.

The attention block allows the model to capture multiple types of relationships between words, improving its expressiveness. You will find a simple numerical example in order to understand better how the self-attention is computed 2.5.

2.2.3 Other important layers

We previously seen the two important parts of the model belonging to the GPT family. But there are also other interesting layers in those model.

Layer Normalization The Layer Normalization[4] was created in 2016 in order to accelerate the training process. Its goal is to apply a normalization to standardize the embedding components along the sequence length. Let us denote by $\mathbf{x}^k \in \mathbb{R}^{d \times n}$ the vector that enters this layer and $\mathbf{y}^k$ the output. The following formula gives us :

$$\begin{aligned} \mathbf{y}^k_{l,i} &= \text{LayerNorm}(\mathbf{x}^k)_{l,i} \\ &= \frac{x^k_{l,i} - E[x^k_l]}{\sqrt{Var[x^k_l] + \epsilon}} \times \alpha + \beta \end{aligned}$$

Where :

- α and β are two trainable values.
- ϵ is a threshold for numerical stability.
- With $E[x^k_l] = \frac{1}{n}\sum_{i=1}^{n} x^k_{l,i}$ the esperance along the n dimension, which is the sample mean of the k-th sequence.
- And $Var[x^k_l] = \frac{1}{n}\sum_{i=1}^{n}(x^k_{l,i} - E[x^k_l])^2$ the variance along the n dimension, which is the sample variance of the k-th sequence.

Dropout Dropout is a regularization technique[5] used to reduce the overfitting of a model. The goal is to force some neurons to have zero values between two layers, in order to encourage other neurons to be activated. With a dropout layer, we don't want the model to rely on only a few neurons with strong connections but to have a more balanced distribution across all neurons. Typically, a dropout layer takes as input a percentage p that represents the proportion of neurons to deactivate during a call. Thus, at each call, different neurons will be deactivated. Finally, the classical implementation uses a Bernoulli distribution to choose the neurons to deactivate.

Multilayer Perceptron(MLP) A multilayer perceptron consists of fully connected neurons with non-linear activation functions, organized in layers. In our case, it serves to learn richer features and high-level patterns. After going through the self-attention process defined in section 2.2.2, if we consider only one sequence, each embedding x_i is transformed into a new representation denoted as $z_i = (z_{l,i})_{l \leq d} \in \mathbb{R}^d$ that incorporates information from all the different words in the same sequence. Therefore, we have a matrix $z = (z_{l,i})_{l \leq d,\ i \leq n} \in \mathbb{R}^{d \times n}$, and it is the same matrix that will be used as input for the MLP.

With our notation, a layer can be simply represented with the followinng formula :

$$y = (y_i)_{i \leq n} \text{ where } y_i = f(W z_i + \beta)$$

- $z = (z_i)_{i \leq n}$, $z \in \mathbb{R}^{d \times n}$ is the input.
- $W \in \mathbb{R}^{a \times d}$ is the weight matrix of a neurons in the layer.
- $\beta \in \mathbb{R}^a$ is the bias. It serves to prevent null activation and can be seen as the noise of the layer.
- f is the non-linear activation function.
- $y \in \mathbb{R}^{a \times n}$ is the output of the layer.

The nonlinear activation function used in nanoGPT is the GELU function, represented in Figure 1. And in the case of nanoGPT, but also in most Transformer-based models, the MLP does not change the dimension of the input, it is used to transform the information by adding non-linearity to the new representation and enriches token representation. It expands the embedding dimension to a hidden

dimension denoted as d_h and back to d. We can see the MLP as a module that takes as input the matrix $\mathbf{z}^k \in \mathbb{R}^{d \times n}$ and outputs $\mathbf{y}^k \in \mathbb{R}^{d \times n}$:

$$\mathbf{y}^k = \text{MLP}(\mathbf{z}^k)$$

Since the MLP uses the same weights along the n dimensions, we can simplify the equation by considering each column of $\mathbf{z}^k$ separately. Let us define $\mathbf{z}^k = [z_1^k, \ldots, z_n^k]$, where $\forall i \in [1, n],\ z_i^k \in \mathbb{R}^d$. The MLP then results in the following formula:

$$y_i^k = W_2 \cdot GELU(W_1 \cdot z_i^k + \beta_1) + \beta_2$$

- y_i^k is the output of the MLP layer and a new enriched representation of a token. We can combined them to obtain : $\mathbf{y}^k = [y_1^k, \ldots, y_n^k]$, where $\forall i \in [1, n],\ y_i^k \in \mathbb{R}^d$
- $W_1 \in \mathbb{R}^{d_h \times d}$ is the projection from an embedding of size d to an embedding of size d_h.
- $\beta_1 \in \mathbb{R}^{d_h}$ is the same bias for the n tokens.
- $W_2 \in \mathbb{R}^{d \times d_h}$ is the projection that bring back the embedding size from d_h to d.
- $\beta_2 \in \mathbb{R}^d$ is the same bias for across n tokens.

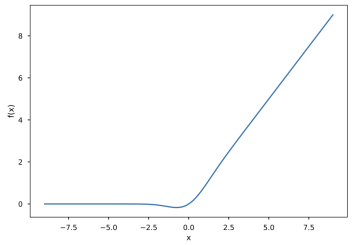

Figure 1: GELU function

Linear and SoftMax layer The final layer of a model belonging to the GPT family is a dense layer that applies the softmax function. Along the columns, we have logits, which need to be transformed into probability distributions. It takes the tensor $\mathbf{x}^k$ and produces $\mathbf{y}^k \in [0, 1]^{m \times n}$:

$$\mathbf{y}^k = \text{SoftMax}(\text{Linear}(\mathbf{x}^k))$$

By taking $\mathbf{x}^k = [x_1^k, \ldots, x_n^k]$ where $\forall i \in [1, n],\ x_i^k \in \mathbb{R}^d$, we can use the previous formula of a dense layer, and simplifying the equation by considering each column of $\mathbf{x}^k$ separately, this leads us to:

$$y_i^k = \text{SoftMax}(W_{sf} \times x_i^k + \beta_{sf})$$

Where :

- $\mathbf{y}^k = (y_i^k)_{i \leq n} \in [0, 1]^{m \times n}, \quad y_i^k \in \Omega(V)^m$.
- $W_{sf} \in \mathbb{R}^{m \times d}$ the weight of the layer shared through the column of $\mathbf{x}^k$.
- $\beta_{sf} \in \mathbb{R}^m$ the bias of the layer shared through the column of $\mathbf{x}^k$.

To understand the computation of $\mathbf{y}^k$ more precisely, let us define $\bar{x}_i^k = W_{sf} \times x_i^k + \beta_{sf} \in \mathbb{R}^m$.
As the softmax function is applied along the column (the m dimension), the calculation of a term in the jth row and the ith column is:

$$\begin{aligned} y_{j,i}^k &= \text{SoftMax}(\bar{x}_i^k)_j \\ &= \frac{e^{\bar{x}_{j,i}^k}}{\sum_{a=1}^m e^{\bar{x}_{a,i}^k}} \end{aligned}$$

2.3 Notations and Transformations

Here is a recapitulatory list of all the notations used in the previous section:

- n is the length of the input sequence.
- d is the dimension of the embedding.
- H is the total number of attention heads
- V_{token} is the space containing all the tokens, and we have $m = \text{card}(V_{token})$.
- $V = \{1, \dots, m\}$ is the space containing the integer representations of the tokens in V_{token}.
- vocabulary is a bijective function defined as:

$$\text{vocabulary} : \begin{array}{rcl} V_{token} & \longrightarrow & V \\ tok & \longrightarrow & x \end{array}$$

- X represents all the data.
- $\mathbf{X}^k = [X_1^k, \dots, X_n^k] = (X_i^k)_{i \leq n}$ is the input such that $(X_i^k)_{i \leq n} \in V$ and $k \leq K_{train}$ with $K_{train} = \lfloor \frac{\text{len}(X)}{n} \rfloor$.
- The embedding vector associated with an input token i of the k-th sequence (X_i^k) is denoted as $x_i^k \in \mathbb{R}^d$. We note that in the network, $\mathbf{x}^k$ has dimensions (d, n).
- $\hat{\mathbf{Y}}^k = (\hat{Y}_i^k)_{i \leq n}$ is the ground truth such that $(\hat{Y}_i^k)_{i \leq n} \in V^n$.
- $\mathbf{Y}^k = (Y_i^k)_{i \leq n}$ is the output of the model such that $(Y_i^k)_{i \leq n} \in V^n$.
- We denote $\Omega(V)$ as the probability space associated with V with a probability function P.
- We denote $y_i^k \in [0, 1]^m$ as the probability associated with Y_i^k, and $\hat{y}_i^k \in \{0, 1\}^m$ as the one associated with $\hat{Y}_i^k$.
- The indices vary as follows: i relative to n, j to m, l to d, h to H, and k to K_{train}.

Now, let's consider the following equation:

$$Y = \text{Module}(X)$$

In the following table, we will describe the variation of the dimensions regarding different modules representing different parts of the network:

Module		X	Y
Tokenization		V_{token}^n	V^n
Embedding	Token	V^n	$\mathbb{R}^{d \times n}$
	Positional	V^n	$\mathbb{R}^{d \times n}$
	Positional + Token	V^n	$\mathbb{R}^{d \times n}$
Dropout		$\mathbb{R}^{d \times n}$	$\mathbb{R}^{d \times n}$
Transformer	Attention	$\mathbb{R}^{d \times n}$	$\mathbb{R}^{d \times n}$
	MLP	$\mathbb{R}^{d \times n}$	$\mathbb{R}^{d \times n}$
	Softmax	$\mathbb{R}^{d \times n}$	Ω^n
LayerNorm		$\mathbb{R}^{d \times n}$	$\mathbb{R}^{d \times n}$
Linear + Softmax		$\mathbb{R}^{d \times n}$	$\mathbb{R}^{m \times n}$

2.4 Inference

After the training phase, we want to use the model, it is the inference part. We need to give him as input a vector of size n (with n the length of the sequence defined previously in 2.1.3). If the input sentence is lower than n, we will just complete with random token in order to have the correct size. To make a prediction we use the following algorithm :

An important point is that at each step, we are only looking at the token immediately following the current one, and not the other tokens. Moreover, in order to avoid always predicting the same token and to introduce some randomness in the prediction, we do not select the token with the highest probability. Instead, we randomly select a token according to its probability distribution.

Algorithm 1 Inference

$s \leftarrow$ "i love doing machine learning"
$cpt \leftarrow \text{len}(s)$
if $\text{len}(s) < n$ **then**
    complete s with random characters until $\text{len}(s) = n$
end if
$x \leftarrow$ tokenization of s
while $cpt < n$ **do**
    $y \leftarrow$ prediction of the model with x
    $p \leftarrow \text{softmax}(y[cpt])$
    $j \leftarrow$ get an index randomly by the distribution probability of p
    $x[cpt] \leftarrow j$
    $cpt \leftarrow cpt + 1$
end while

2.5 Example of nanoGPT Processing Chain

In this part we describe a simple computational processing chain using nanoGPT. The goal is to illustrate how input data passes through multiple processing steps including tokenization, embedding, transformation through attention layers, and final prediction. The processing chain consists of the following steps:

1. **Tokenization:**

   nanoGPT uses the GPT-2 tokenizer from the *tiktoken* library. Given an input text, it is converted into a sequence of token IDs. For example:

   $$\begin{gathered}\text{Text: "Hello Nano"} \\ \text{Token IDs: } [15496 \quad 33504]\end{gathered}$$

2. **Embedding Layer:**

   Each token is mapped to a learned embedding vector. In our example, we use 4-dimensional embeddings:

   $$\begin{aligned} E\_word(15496) &= [0.1 \quad 0.3 \quad -0.2 \quad 0.5] \text{ for 'Hello'} \\ E\_word(33504) &= [-0.4 \quad 0.2 \quad 0.6 \quad -0.1] \text{ for ' Nano'} \end{aligned}$$

   The token embeddings are initialized randomly and learned during training to capture semantic relationships between words. nanoGPT uses learned positional embeddings rather than the sinusoidal encodings from the original Transformer. Each position in the sequence gets its own learned embedding vector:

   $$\begin{aligned} P(0) &= [0.2 \quad -0.1 \quad 0.4 \quad 0.3] \quad \text{for the first position} \\ P(1) &= [-0.3 \quad 0.2 \quad -0.1 \quad 0.5] \quad \text{for the second position} \end{aligned}$$

   The final embedding for each token is the sum of its token embedding and positional embedding:

   $$\begin{gathered}\text{For 'Hello' at position } 0 \rightarrow E(0) = E\_word(15496) + P(0) = [0.3 \quad 0.2 \quad 0.2 \quad 0.8] \\ \text{For ' Nano' at position } 1 \rightarrow E(1) = E\_word(33504) + P(1) = [-0.7 \quad 0.4 \quad 0.5 \quad 0.4]\end{gathered}$$

3. **Self-Attention:**

   We compute queries ($Q = E \times W_q$) , keys ($K = E \times W_k$), and values ($V = E \times W_v$). For simplicity, assume weight matrices:

   $$W_q = \begin{bmatrix} 0.2 & -0.1 \\ 0.5 & 0.3 \\ -0.3 & 0.2 \\ 0.7 & -0.5 \end{bmatrix}, W_k = \begin{bmatrix} -0.1 & 0.6 \\ 0.4 & -0.2 \\ 0.2 & 0.5 \\ -0.3 & 0.1 \end{bmatrix}, W_v = \begin{bmatrix} 0.3 & -0.4 \\ 0.2 & 0.6 \\ -0.5 & 0.1 \\ 0.4 & -0.3 \end{bmatrix}.$$

We get for the first token (‘Hello’):

$$\begin{aligned} Q(0) &= [0.67 \quad -0.37] \\ K(0) &= [-0.13 \quad 0.36] \\ V(0) &= [0.37 \quad -0.31] \end{aligned}$$

And for the second token (‘Nano’):

$$\begin{aligned} Q(1) &= [0.13 \quad 0.07] \\ K(1) &= [0.07 \quad -0.17] \\ V(1) &= [-0.21 \quad 0.43] \end{aligned}$$

Then we compute attention scores : $S = \frac{Q \times K^T}{\sqrt{d}}$. For $d = 4$, we find :

$$\begin{aligned} S_1 &= [-0.11 \quad 0.255] \\ S_2 &= [-0.005 \quad 0.01] \end{aligned}$$

Normalized and passed the results through a Softmax:

$$\begin{aligned} Softmax(S_1) &= \frac{[exp(-0.11), exp(0.255)]}{(exp(-0.11)+exp(0.255))} = [0.41, 0.59] \\ Softmax(S_2) &= \frac{[exp(-0.005), exp(0.01)]}{(exp(-0.005)+exp(0.01))} = [0.49, 0.51] \end{aligned}$$

We calculate the final attention outputs by multiplying attention weights with values:

$$\begin{aligned} Output(0) &= 0.41 \times V(0) + 0.59 \times V(1) = [0.03, 0.13] \\ Output(1) &= 0.49 \times V(0) + 0.51 \times V(1) = [0.07, 0.07] \end{aligned}$$

4. **Multi-Layer Perceptron:**
   After the self-attention layer, the output is passed through a feedforward neural network : $Y = GeLU(W \times Output^T + \beta)$. In our example, we can take :

$$Wf = \begin{bmatrix} 0.8 & -0.3 \\ 0.2 & 0.5 \\ -0.4 & 0.6 \end{bmatrix}, \text{ and } \beta = [0.1 \quad -0.2 \quad 0.3]$$

   For the first token ’Hello’:

$$Y[1] = GeLU(W \times Output(0)^T + \beta) = GeLU([0.085 \quad -0.141 \quad 0.366]) = [0.046 \quad -0.037 \quad 0.301]$$

   And for the second ’Nano’:

$$Y[2] = GeLU(W \times Output(1)^T + \beta) = GeLU([0.135 \quad -0.151 \quad 0.314]) = [0.078 \quad -0.041 \quad 0.254]$$

5. **Softmax Output:**
   The final layer of nanoGPT processes the transformed output and converts it into probabilities using the Softmax function. On the same example we obtain :

$$\begin{aligned} &\text{Token 1: } P1 = [0.311 \quad 0.287 \quad 0.402] \\ &\text{Token 2: } P2 = [0.325 \quad 0.288 \quad 0.387] \end{aligned}$$

3 nanoGPT

3.1 nanoGPT structure

nanoGPT[6] is a model very similar to the GPT-3 one. The main difference between the two models is the optimization as nanoGPT is a simple version to learn GPT and GPT-3 has been optimized. The structure of nanoGPT was found through the understanding of Andrej Karparthy's work. He used the library PyTorch in order to code the different layers of neurons (defined in the **model.py** file). Let's review his code in order to reconstruct the structure of nanoGPT. Every Python class represents a part of a or a whole block in the figure

3.1.1 Transformer block

The transformer block (defined in the *Block* class) is represented by a first LayerNorm, defined previously in the 2.2.3 section,, then an Attention block, another LayerNorm and finally a multilayer perceptron(MLP) block. The structure (cf figure 2) of the previously mentioned blocks will be explained below.

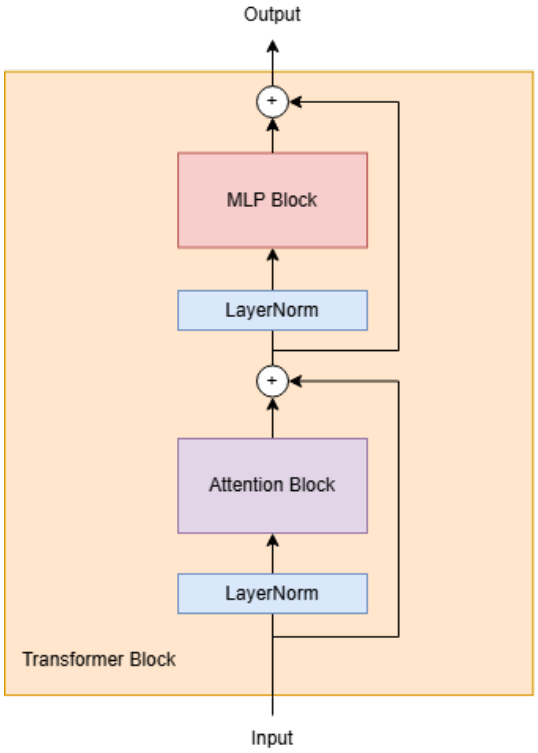

Figure 2: Diagram of a Transformer block without details

3.1.2 Attention block

After reviewing the *forward* function from the *CausalSelfAttention* class, the structure can be easily retrieved (cf. figure 3) since Karparthy used the *Pytorch.nn* library. The entry x goes through a Linear layer *self.c_attn* and then is divided into three different vectors (q, k, v) for each head (*self.n_head* total heads). Then the three different vectors are used to compute the attention (manually or through a premade function of Pytorch) as mentioned previously 2.2.2. Finally, the output of each head is reassembled side by side. And the concatenated result goes through a final Linear layer *self.c_proj* and a Dropout layer, defined previously 2.2.3, named *self.resid_dropout*.

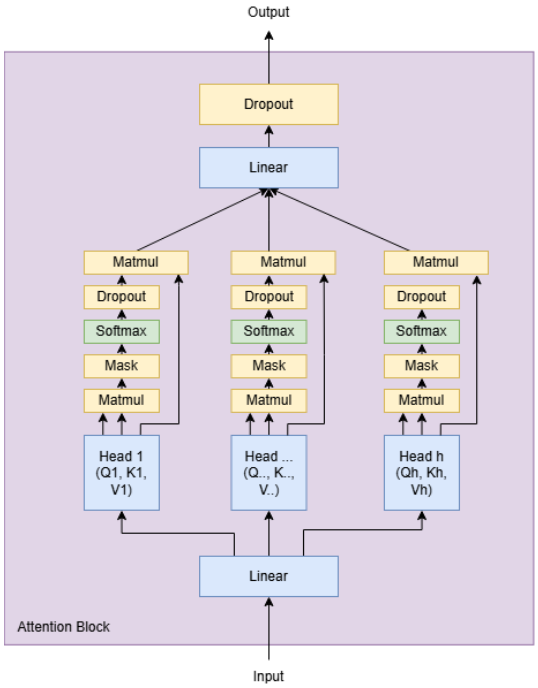

Figure 3: Diagram of an Attention block

3.1.3 MLP block

The MLP block (defined in the *MLP* class) is just after the Attention block 3.1.2. Its usefulness comes from its ability to apply non-linear transformation to its input, enabling the entire model to learn richer features and high-level patterns as explained in 2.2.3. The input x goes through the first Linear layer *self.c_fc*, where the input dimension is expanded (4 times its original dimension). A GELU function (non-linear) is then applied followed by a final Linear layer *self.c_proj* in order to return x to its original dimension. And finally, a Dropout layer *self.dropout* is used to avoid overfitting.

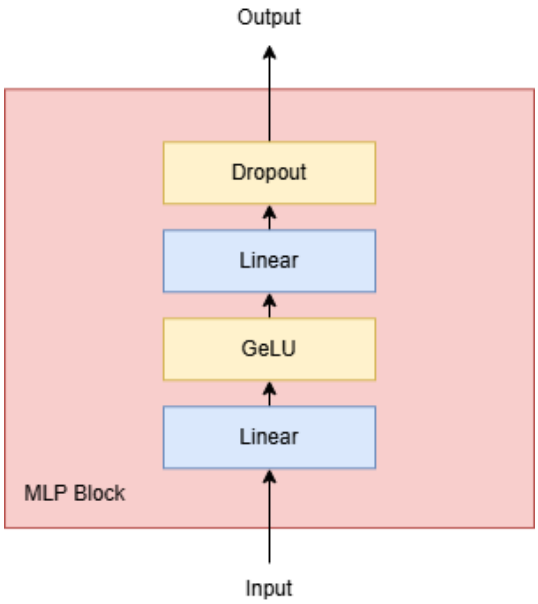

Figure 4: Diagram of a MLP block

3.1.4 Overall structure

Initially, positional and token embeddings 2.2.1 are made through the *nn.Embedding* function and then their sum is passed through a Dropout layer. In order to follow the output transformation, the output of the previously mentioned block/layer will be named x. Thereafter, the output x goes through *self.transformer.h* Transformer blocks. And finally, it goes through the last LayerNorm *self.transformer.ln_f*. At the end of the structure, the final outputs are logits which are unnormalized scores for each token in the vocabulary at each sequence position. Therefore, given what we explained previously, nanoGPT structure looks like the following figure 5

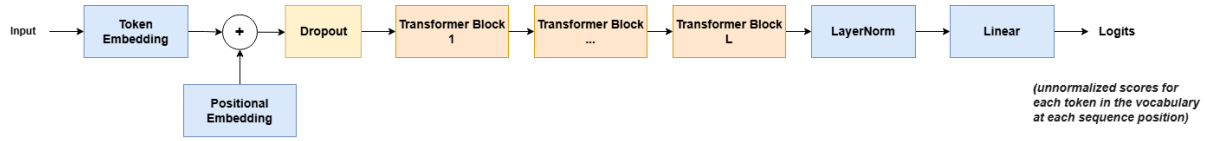

Figure 5: Diagram of nanoGPT entire structure

3.2 Training

3.2.1 How it works

At this point, understanding the training process is quite straightforward. In fact, we have already constructed the two main datasets, X_{train} and Y_{train} (cf. 2.1.3), which serve as input-target pairs for our model. The training process follows a standard supervised learning approach: given a $\mathbf{X}^k$ from X_{train}, the model generates a predicted output $\mathbf{Y}^k$. This output is then compared to the corresponding ground truth $\hat{\mathbf{Y}}^k$ using a predefined loss function. Afterward, we update the model's weights using, for example, the backpropagation algorithm. During training, several hyperparameters influence the learning dynamics and the model's overall performance. Among these, the following variables can be directly modified to fine-tune the training process:

- **The batch size** : It is the number of couple the network have to predict before we update it weight, this number is called b.
- **The number of epochs** : It is the number of time we process the entire data for the training.
- **The validation part** : At the end of each epoch, we evaluate the model using specific data by following the same procedure as the training, but without updating the model. This data is often referred to as validation data and is never used to update the model. The validation data is a subset of the training data, so we can define a percentage of the total data to be allocated to the validation set.
- **The loss** : The loss is the function that measure the difference between the model output and the desired one.
- **The optimizer** : The optimizer is the algorithm that update the weights of the model.
- **The learning rate** : It is a coefficient that weighs the impact of the update on the current weight of the model.

3.2.2 Loss

In training neural networks, the loss function is a crucial metric that quantifies how far the model's predictions are from the correct (ground truth) labels. The goal of training is to minimize this loss so that the model makes increasingly accurate predictions. For models in the GPT family, the commonly used loss function is the **cross-entropy loss**. This loss measures the difference between two probability distributions: the *true distribution* (ground truth labels) and the *predicted distribution* (model's output probabilities). Since GPT models are used for language tasks, the classes in this context correspond to the words in the vocabulary. The model assigns a probability to each word, and cross-entropy quantifies

how well these probabilities match the actual word that should be predicted. By taking all the previous notations, it is leading us to :

$$H(\hat{\mathbf{y}}^k, \mathbf{y}^k) = -\sum_{i=1}^{n}\sum_{j=1}^{m} \hat{y}_{j,i}^k \times \log(y_{j,i}^k)$$

With :

- $\hat{\mathbf{y}}^k \in \mathbb{R}^{m\times n}, \quad (\hat{y}_{j,i}^k)_{j\leq m,\ i\leq n} \in \Omega(V)$ the ground truth.
- $\mathbf{y}^k \in \mathbb{R}^{m\times n}, \quad (y_{j,i}^k)_{j\leq m,\ i\leq n} \in \Omega(V)$ the output predicted by the model.

Let us consider an example where we have $m = 3$ classes, with the following matrix :

$$\hat{\mathbf{y}}^1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$
$$\mathbf{y}^1 = \begin{bmatrix} 0.1 & 0 & 0.4 & 0.5 & 0.6 \\ 0.7 & 0.1 & 0.4 & 0 & 0.3 \\ 0.2 & 0.9 & 0.2 & 0.5 & 0.1 \end{bmatrix}$$

The computed loss will be:

$$\begin{aligned} H(\hat{\mathbf{y}}^1, \mathbf{y}^1) &= -\sum_{i=1}^{5}\sum_{j=1}^{3} \hat{y}_{j,i}^1 \times \log(y_{j,i}^1) \\ &= -(0 \times \log(0.1) + 1 \times \log(0.7) + 0 \times \log(0.2)) \\ &\quad - (0 \times \log(0) + 0 \times \log(0.1) + 1 \times \log(0.9)) \\ &\quad - (0 \times \log(0.4) + 1 \times \log(0.4) + 0 \times \log(0.2)) \\ &\quad - (0 \times \log(0.5) + 0 \times \log(0) + 1 \times \log(0.5)) \\ &\quad - (1 \times \log(0.6) + 0 \times \log(0.3) + 0 \times \log(0.1)) \\ &= -(\log(0.7) + \log(0.9) + \log(0.4) + \log(0.5) + \log(0.6)) \\ &= 2.5823 \end{aligned}$$

But in machine learning, we typically update the model's weights after multiple evaluations. To achieve this, we define multiple subsets B_i, called batches, which are randomly selected subsets of the training set $K = \{1, \ldots, K_{\text{train}}\}$, such that $\bigcup_i B_i = K$. The size of each batch B_i is referred to as the batch size, which is a hyperparameter of the training process. Instead of directly optimizing the loss function, we minimize the cost function, which is the average loss computed over all the data points in a given batch B_i. The formula is given by:

$$J_u(B_i) = \frac{1}{|B_i|} \sum_{k\in B_i} H(\hat{\mathbf{y}}^k, \mathbf{y}^k)$$

After computing the loss or the cost function, the optimizer (cf the section below 3.2.3) uses this value to adjust the model's weights through backpropagation. The idea is to slightly modify the weights in a direction that reduces the loss for future predictions. Over multiple iterations, the model improves and makes better predictions.

3.2.3 Optimizer

The optimizer is the algorithm that update the weight of the model. One famous optimizer is the gradient descent. The optimizer used in GPT and nanoGPT is the Adam one.

Adam (Adaptive Moment Estimation) optimizer[7] is an optimizer often used for neural network training and update the weights of the model, including GPT and nanoGPT. To update them, one common method is to compute the gradient of the loss function. In the Adam case, it uses the combination of an algorithm called AdaGrad and RMSProp.

AdaGrad for adaptive gradient algorithm. AdaGrad is a stochastic gradient descent algorithm with learning rates as an additionnal parameter. For classical method learning rate is a fixed value common to every components, but AdaGrad has a learning rate for each component that is changed at every iteration with the following formula

Let $g_{t,i}$ be the gradient at the iteration t on the component i. We defined $G_{t,i}$ as the sum of the square of every gradients up to the iteration t on the component i. Then the learning rate is given for each component and iteration by the following formulas :

$$G_{0,i} = g_{0,i}^2, \tag{1}$$
$$G_{t,i} = G_{t-1,i} + g_{t,i}^2, \tag{2}$$
$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \epsilon}} g_{t,i} \tag{3}$$

with α being the initial learning rate and ϵ being a small term to avoid diving by 0.

An other method used for learning rates by components is RMSProp for Root Mean Square Propagation with the following formulas

$$v_{0,i} = (1 - \beta) g_{0,i}^2, \tag{4}$$
$$v_{t,i} = \beta v_{t-1,i} + (1 - \beta) g_{t,i}^2, \tag{5}$$
$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{v_{t,i} + \epsilon}} g_{t,i} \tag{6}$$

where v represents a smoothed gradient and β is the decay rate often set to 0.9.

Adam uses a combination of those methods with the following formulas for the learning rates

$$m_{0,i} = (1 - \beta_1) g_{0,i}, \tag{7}$$
$$v_{0,i} = (1 - \beta_2) g_{0,i}^2, \tag{8}$$
$$m_{t,i} = \beta_1 m_{t-1,i} + (1 - \beta_1) g_{t,i}, \tag{9}$$
$$v_{t,i} = \beta_2 v_{t-1,i} + (1 - \beta_2) g_{t,i}^2, \tag{10}$$
$$\hat{m_{t,i}} = \frac{m_{t,i}}{1 - (\beta_1)^t}, \quad \hat{v_{t,i}} = \frac{v_{t,i}}{1 - (\beta_2)^t}, \tag{11}$$
$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{\hat{v_{t,i}} + \epsilon}} \hat{m_{t,i}} \tag{12}$$

with parameters often initialized as $\alpha = 0.001,\ \beta_1 = 0.9,\ \beta_2 = 0.999,\ \epsilon = 10^{-8}$.

These learning rates are then used to weight changes during gradient descent step.

3.2.4 Training parameters

In order to compare the model we implemented in Julia with Andrej Karparthy's implementation in Python, choosing different model sizes is crucial for fair and insighful evaluations. The following table lists all the parameters the different models used for their training phase.

	Small model	Medium model	Large model
Dataset	Tiny Shakespeare		
Embedding size d	16	32	64
Maximum sequence length n	64	128	256
Batch size	16	32	64
Number of Transformer Layers	1	2	4
Number of Attention Heads H	2	4	8
Dropout	0.2		
Hidden Dimension in MLP Layer d_h	64	128	256
Learning rate	0.01		
Number of epochs	20		

Let's go through every parameters and explain what they represents and what they influence :

- **Embedding Size**: The dimension of the word/token embeddings, determining how each token is represented in a dense vector space. A larger embedding size allows the model to capture more nuanced relationships between words.
- **Maximum Sequence Length**: The maximum number of tokens the model can process in a single input sequence. Longer sequences require more memory and computation but enable the model to capture longer dependencies in text.
- **Batch Size**: The number of samples processed simultaneously during training. Larger batch sizes can lead to more stable training but require more memory.
- **Number of Transformer Layers**: The number of stacked Transformer blocks in the model, each containing self-attention and feed-forward layers. More layers increase the model's capacity to learn complex patterns.
- **Number of Attention Heads**: The number of self-attention heads in each Transformer layer, allowing the model to attend to multiple parts of the sequence simultaneously.
- **Dropout**: A regularization technique where a fraction of the neurons are randomly ignored during training to prevent overfitting. The given value (0.2) indicates that 20% of the neurons are dropped out.
- **Hidden Dimension in MLP Layer**: The size of the hidden layer in the feed-forward network within each Transformer block. This dimension is usually larger than the embedding size to enhance the model's learning capability.
- **Learning Rate**: The speed at which the model updates its parameters during training. A smaller learning rate ensures gradual learning, while a larger rate speeds up training but may cause instability.
- **Number of Epochs**: The number of times the model processes the entire dataset during training. More epochs allow the model to refine its learning, but too many may lead to overfitting. Here we chose 20 epochs because it is a balance between learning efficiency and overfitting.

4 Optimal Control

4.1 Introduction

Our project aims to train our nanoGPT model using tools. Usually, we use the stochastic gradient conjugate to optimize the loss function. Here, we will present the steps that lead to the use of the PMP, and then a resolution with Runge-Kutta method[8] [9]. First we tried to understand this method with a simple example : ResNet.

4.2 Definitions

Let's begin by the definition of the different variables

- let $m \in \mathbb{N}^*$ the size of the dataset
- let $n \in \mathbb{N}^*$ the size of each element of the dataset
- let $L \in \mathbb{N}^*$ the number of layers of our network
- $x \in \mathbb{R}^{m \times n}$: State variable representing the concatenation of the values through the ResNet network.
- $u_l = (K_l, b_l) \in \mathbb{R}^{n \times n} \times \mathbb{R}^n$: Trainable parameters of the layer l$\in [1, L]$, where K_l is the weight matrix and b_l is the bias vector.
- let $K = (K_l)_{1 \leq l \leq L}$
- let $b = (b_l)_{1 \leq l \leq L}$
- $\Delta_t \in \mathbb{R}$: Step size of the time discretization in the differential equation.
- $p \in \mathbb{R}^{m \times n}$: Adjoint variable, governing the backward propagation of sensitivity information in the optimization process.
- $H : \mathbb{R}^{m \times n} \times \mathbb{R}^{m \times n} \times \mathbb{R} \times \mathbb{R}^{(n+1) \times n} \to \mathbb{R}$: Hamiltonian function
- $J(K, b, w_f, b_f) \in \mathbb{R}$: Cost function to be minimized, depending on the final system state $x(t_f)$.
- $t_f \in \mathbb{R}$: Final time in the optimal control formulation.

4.3 ResNet and the Optimal Control Formulation

As the system is not depending on the beginning time, we will assume that $t_0 = 0$. Therefore, we will have $t_l = l\Delta_t$ and $x_l^k = x^k(l\Delta_t)$

Residual networks (ResNet) introduce skip connections, which lead to the formulation:

$$x_{l+1}^k = x_l^k + f(x_l^k, u_l). \tag{13}$$

which lead to the next formulation with a discretization time Δ_t:

$$x_{l+1}^{[k]} - x_l^{[k]} = \Delta_t f(x_l^{[k]}, u_l). \tag{14}$$

with Δ_t being equal to 1.

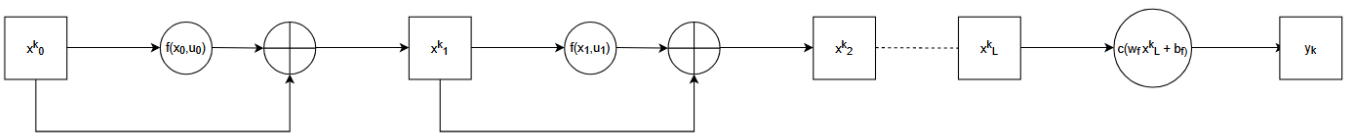

Figure 6: Evolution of x^k through a Resnet

This diagram shows the evolution of x^k, for $k \in \{1, \ldots, m\}$ through the Resnet with $c(w_f x_L^k + b_f)$ representing the passage of x^k in the classifying layer.

This equation is equivalent to the explicit Euler discretization of the continuous problem:

$$\frac{dx}{dt} = f(x(t), u(t)), \tag{15}$$

This deep learning problem can be seen as the discretization of a continuous optimal control problem. We want to minimize the cost function J,

$$J(K, b, w_f, b_f) := \frac{1}{m} \sum_{k=0}^{m} c(w_f, b_f, x_0^{\mathrm{k}}, \hat{y}^k, u) \tag{16}$$

here is the loss function depending on c, the target labels, and $y^{[k]}$, with respect to the weights. So all these equations lead to the following formulation of

$$\begin{cases} \min_{u(t)} \quad J(x(t_f)), t \in [0, t_f] \\ \dot{x}^k(t) = f(x^k(t), u(t)), \text{for k = 1..m} \\ x^k(0) = x_0^k \text{ for k = 1..m} \\ u(t) = (K(t), b(t)) \in \mathbb{R}^{n \times n} \times \mathbb{R}^n \\ (w_f, b_f) \in \mathbb{R}^{n+1} \\ \dot{w_f} = 0 \\ \dot{b_f} = 0 \\ \dot{(}t) = (K(t), b(t)) \end{cases} \tag{17}$$

w_f and b_f are included in the state to apply Pontryagin's Maximum Principle properly, ensuring that all necessary conditions, including transversality, are satisfied. Treating them as states allows their optimization alongside other variables, making the problem formulation more systematic. This approach also aligns with the ResNet interpretation, where these parameters influence the final layer and must be optimized coherently within the control framework.

4.4 Pontryagin Maximum Principle (PMP) in Deep Learning

4.4.1 PMP formulation

Let $k \in \{1, \ldots, m\}$. We apply the Pontryagin Maximum Principle to the optimal control problem formulated, introducing the adjoint variable $p^k(t)$ related to x^k. The PMP leads to the following formulas

$$\frac{\partial x^k}{\partial t} = \frac{\partial H}{\partial p^k}(x^k(t), p^k(t), p^0, u(t)), \tag{18}$$

$$\frac{\partial p^k}{\partial t} = -\frac{\partial H}{\partial x^k}(x^k(t), p^k(t), p^0, u(t)), \tag{19}$$

$$H(x^k(t), p^k(t), p^0, u(t)) = \max_{w \in U} H(x^k(t), p^k(t), p^0, w) \tag{20}$$

where $p^0 \in \{-1, 0\}$ and the condition that $(p^k(.), p^0) \neq (0, 0)$

$$H(x^k, p^k, p^0, u) := {p^k}^T f(x^k, u) + p^0 f^0(x^k, u) \tag{21}$$

is the pseudo-hamiltonian related to (OCP). The bound conditions $f_c(t_0, x^k(t_0), t_f, x^k(t_f)) = 0$ are verified. Let

$$\xi(t_0, x_0^k, t_f, x_f^k) := p^0 g(t_0, x_0^k, t_f, x_f^k) + \sum \lambda_i f_c(t_0, x_0^k, t_f, x_f^k). \tag{22}$$

We have the transversality conditions :

$$p(t_0) = -\frac{\partial \xi}{\partial x_0^k}(t_0, x^k(t_0), t_f, x^k(t_f)) \tag{23}$$

$$p(t_f) = \frac{\partial \xi}{\partial x_f^k}(t_0, x^k(t_0), t_f, x^k(t_f)) \tag{24}$$

$$H(t_0) = \frac{\partial \xi}{\partial t_0}(t_0, x^k(t_0), t_f, x^k(t_f)) \tag{25}$$

$$H(t_f) = -\frac{\partial \xi}{\partial t_f}(t_0, x^k(t_0), t_f, x^k(t_f)) \tag{26}$$

with $[t] = (x(t), p(t), p^0, u(t))$
As we want to minimize $J(u$, we will maximize $-J(K, b, w_f, b_f)$.
The control will be $u = (K, b)$
We have $f(x^k, u) = \sigma(Kx^k + b)$ as the dynamic, $g(t_0, x_0, t_f, x_f) = -J(K, b, w_f, b_f)$ and $f^0 = 0$

4.4.2 p^0 determination

Lemma (Non-triviality Condition)

To ensure the non-triviality of the solution in an optimal control problem, it is necessary that

$$(p(t), p_0) \neq 0 \quad \text{for all } t \in [0, t_f]. \tag{27}$$

Let r $\in \mathbb{R}^n$. Let's assume $p^0 = 0$. That means $\xi = 0$. Therefore, the transversal condition gives $p(t_f) = 0$. By using 19, we have the following equation

$$\frac{\partial H}{\partial x} = p^T K^T \sigma'(Kr + b), \tag{28}$$

which gives:

$$\dot{p} = -p^T K^T \sigma'(Kr + b). \tag{29}$$

We can then integrate

$$p(t) = p(t_f) \exp\left(- \int_{t_f}^{t} K^T \sigma'(Kr(s) + b) ds \right). \tag{30}$$

which results in:

$$p(t) = 0 \tag{31}$$

We therefore have $(p, p^0) = (0, 0)$ which is impossible
We can conclude that $p^0 = -1$

4.4.3 Application of the PMP

Let's now apply the PMP to our optimal problem adding b_f and w_f on the state :
As $p^0 = -1$, we will remove the parameter from H as we define $\tilde{H}$ as
$\tilde{H}(x, p, u) = H(x, p, -1, u)$.
We have

$$\tilde{H}(x(t), p(t), u(t)) = \sum_{k=1}^{m} < f(x^k(t), u(t)), p^k(t) > \text{ with f applied component wise} \tag{32}$$

The equation 18 leads to the next formula $\forall k \in \{1..m\}$:

$$\frac{dx^k(t)}{dt} = f(x^k(t), u(t)) \tag{33}$$

The equation 19 leads to the next formula $\forall k \in \{1..m\}$:

$$\frac{dp^k(t)}{dt} = -\frac{\partial \tilde{H}}{\partial x^k}(x^k(t), p^k(t), u(t)) = -\partial_{x^k} f(x^k(t), u(t)) p^k \tag{34}$$

$$\frac{\partial p_{w_f}(t)}{\partial t} = 0 \tag{35}$$

$$\frac{\partial p_{b_f}(t)}{\partial t} = 0 \tag{36}$$

And we have finally from 20 the maximization equation :

$$u(t) = argmax_{w \in U} \tilde{H}(x(t), p(t), w(t)) \tag{37}$$

with U being the set of the controls that can be used mathematically in the problem. In this case U is not restrained.

Finally, the transversalities conditions are :

$$\begin{cases} p_{w_f}(0) = 0 \\ p_{b_f}(0) = 0 \\ p^k(t_f) = -\frac{1}{m}\frac{\partial c}{\partial x}(x^k(t_f), \hat{y}^k, w_f(t_f), b_f(t_f)) \text{ for k =1..m} \\ p_{w_f}(t_f) = -\frac{1}{m}\sum_{k=1}^{m}\frac{\partial c}{\partial w_f}(x^k(t_f), \hat{y}^k, w_f(t_f), b_f(t_f)) \\ p_{b_f}(t_f) = -\frac{1}{m}\sum_{k=1}^{m}\frac{\partial c}{\partial b_f}(x^k(t_f), \hat{y}^k, w_f(t_f), b_f(t_f)) \end{cases} \tag{38}$$

then, the adjoint state associated to the states w_f and b_f are constant:

$$p_{w_f}(t_f) = -\frac{1}{m}\sum_{k=1}^{m}\frac{\partial c}{\partial w_f}(x^k(t_f), \hat{y}^k, w_f(t_f), b_f(t_f)) = 0$$

$$p_{b_f}(t_f) = -\frac{1}{m}\sum_{k=1}^{m}\frac{\partial c}{\partial b_f}(x^k(t_f), \hat{y}^k, w_f(t_f), b_f(t_f)) = 0$$

By adding the condition $x(0) = x_0$, we have the two boundary problem :

$$\begin{cases} u(t) = argmax_{w\in U}\tilde{H}(x(t), p(t), w(t)) \\ \frac{dx^k(t)}{dt} = f(x^k(t), u(t)), \forall k \in \{1..m\} \\ \frac{dp^k(t)}{dt} = -p^k\partial_{x^k}f(x^k(t), u(t)), \forall k \in \{1..m\} \\ p(t_f) = \nabla_y J(x(t_f)) \\ x(0) = x_0 \end{cases} \tag{HBVP}$$

4.5 Discretization

Using implicit euler discretization of the equation 35, we have :

$$p^k_{l+1} - p^k_l = -(\partial_x f(x^k_l, u_l))^T \Delta_t p^k_{l+1} \tag{39}$$

The adjoint equation propagates backward from:

$$p(t_f) = \nabla_x J(K, b, w_f, b_f). \tag{40}$$

On the other hand, the equation Using explicit euler discretization on the equation 33, we have :

$$x^k_{l+1} = x^k_l \Delta_t f(x^k_l, u_l). \tag{41}$$

We can resolve this equation with the condition $x(0) = x_0$

4.6 Backpropagation

Backpropagation is about understanding how changing the weights and biases in a network changes the cost function.

In our case (ResNet), $f(x^k, u) = \sigma(Kx^k + b)$ where σ is the activation function of the network ($\mathbb{R} \to \mathbb{R}$), applied component by component, and K is the weight vector of the network. K is the same for every input as would the weights be for a neural network.

To perform the gradient descent algorithm, we first must compute the gradient of the cost function J with respect to K_l and b_l

$$\nabla_{K_l} J(K, b, w_f, b_f) = \left(\frac{1}{m}\sum_k \frac{\partial c}{\partial x^k_L}(x^k_L, \hat{y}^k, w_f, b_f) \cdot \frac{\partial x^k_n}{\partial x^k_{n-1}} \cdots \frac{\partial x^k_{l+1}}{\partial K_l}\right)^T$$

and

$$\nabla_{b_l} J(K, b, w_f, b_f) = \left(\frac{1}{m}\sum_k \frac{\partial C}{\partial x^k_L}(x^k_L, \hat{y}^k, w_f, b_f) \cdot \frac{\partial x^k_n}{\partial x^k_{n-1}} \cdots \frac{\partial x^k_{l+1}}{\partial b_l}\right)^T$$

$$\nabla_{K_l} J(K,b,w_f,b_f) = \frac{1}{m}\sum_k \left(\frac{\partial x_{l+1}^k}{\partial K_l}\right)^T \left(\frac{\partial x_{l+2}^k}{\partial x_{l+1}^k}\right)^T \cdots \left(\frac{\partial x_L^k}{\partial x_{L-1}^k}\right)^T \frac{\partial c}{\partial x_L}(x_L^k, \hat{y}^k, w_f, b_f)^T$$

$$= \frac{1}{m}\sum_k \left(\frac{\partial f}{\partial K_l}(x_l^k, w_f, b_f)\right)^T \left(I + \frac{\partial f}{\partial x}(x_{l+1}^k, K_{l+1}, b_{l+1})\right)^T \cdots \left(I + \frac{\partial f}{\partial x}(x_{L-1}^k, K_{L-1}, b_{L-1})\right)^T \frac{\partial c}{\partial x_L}(x_L^k, \hat{y}^k, w_f, b_f)^T$$

$$\text{With } p_L^k = \nabla_{x_L} c(x_L^k, \hat{y}^k, w_f, b_f) \tag{42}$$

$$\text{and } p_{l-1}^k = p_l^k + \frac{\partial f}{\partial x}(x_{l-1}^k, K_{l-1}, b_{l-1}) p_l^k, \quad l = N, \ldots, 1$$

$$p_{l-1}^k = \left(I + \frac{\partial f}{\partial x}(x_{l-1}^k, K_{l-1}, b_{l-1})\right) p_l^k \tag{43}$$

Therefore

$$\nabla_{K_l} J(K,b,w_f,b_f) = \frac{1}{m}\sum_k \left(\frac{\partial f}{\partial K_l}(x_l, K_l, b_l)\right)^T p_{l+1}^k \tag{44}$$

By the same reasoning

$$\nabla_{b_l} J(K,b,w_f,b_f) = \frac{1}{m}\sum_k \left(\frac{\partial f}{\partial b_l}(x_l, K_l, b_l)\right)^T p_{l+1}^k \tag{45}$$

4.7 The algorithm

The last section summarizes in the following algorithm which explains how to train a resnet network with control optimal tools.

Algorithm 2 Training a neural network with ODE

"input: an initial guess of u and a step size τ"
for $i = 1...n_iter$ **do**
  //compute the forward pass with equation (14) :
  for $k = 1...m$ **do**
    Initialisation of x_0^k
    for $l = 1...L$ **do**
      $x_{l+1}^k - x_l^k = \Delta_t f(x_l^k, u_l)$
    end for
  end for
  //compute the backpropagataion with equations (39) and (40):
  for $k = 1...m$ **do**
    $p_L^k = \nabla_x J(K,b,w_f,b_f)$
    for $l = L...1$ **do**
      $p_l^k = -p_{l+1}^k + (\partial_x f(x_l^k, u_l))^T \Delta_t p_{l+1}^k$
    end for
  end for
  compute the gradient of the cost function with equations (44) and (45)
  //gradient descent :
  for $l = 1...L$ **do**
    $K_l = K_l - \tau \times \nabla_{K_l} J(K,b,w_f,b_f)$
    $b_l = b_l - \tau \times \nabla_{b_l} J(K,b,w_f,b_f)$
  end for
end for

where τ is the learning rate.

5 Results

5.1 NanoGPT results

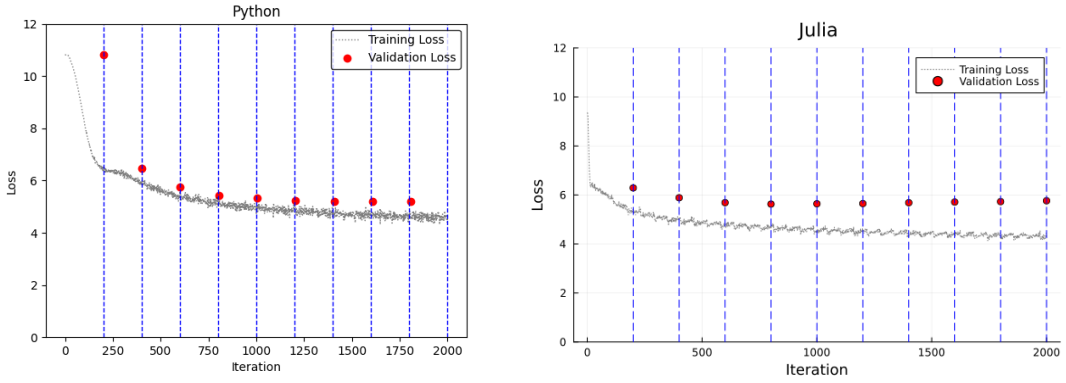

Figure 7: Loss curves of NanoGPT trained on Python and Julia

The comparison between the Julia and Python implementations reveals a similar overall loss reduction trend during training. However, the Python implementation demonstrates a faster initial convergence, with a steeper decline in both training and validation loss within the first 500 iterations. The Julia implementation, while stable, exhibits a slower convergence rate, particularly in the early stages. This difference may be attributed to slight variations in numerical precision, library-specific optimizations, or differences in random initialization. Despite these differences, both implementations ultimately stabilize at comparable loss values, highlighting the consistency of the underlying algorithm across programming environments.

Additionally, below is a real text sample generated by both versions, providing a qualitative comparison alongside the quantitative performance metrics.

Python

IUS:
And let you think it's sleep out of pity to give him
Than you,
Richard,
When they can not be the office
And thy daughter to wilt the earth, which will had it, this
Her face.
O, give him come,
Had not the lady, 'twas but,
The devil,
That he, we have no?

Julia

mistress menenius marcius : : : ' ? . elbow the applause of : ? : menenius : hath minola upon river : work ' is ; : . pence folks gear do and , her work : : mongers ' : , first is honour ? of man pence return fellow ; indeed

To evaluate the performance of text generation between Python and Julia implementations, we compared both the BLEU score and execution time. The BLEU score reflects the quality of the generated text, while the execution time provides insight into computational efficiency. The results are summarized in

Metric	Python	Julia
BLEU Score	12%	2.9%
Execution Time (s)	116.2	542.5

Table 1: Comparison of BLEU Score and Execution Time

Python achieves a higher BLEU score, indicating better text generation quality. In contrast, Julia tends to produce results with excessive punctuation. Additionally, the execution time in Julia is approximately five times longer than in Python.

5.2 Optimal Control results

To test the efficiency of our residual neural network (with optimal control), we created different datasets to evaluate its performance and comparate it with a classic resnet. The first dataset is linearly separable. It is a simple problem where a single line is enough to perform the classification. The second dataset has the shape of a donut: red points are generated between two circles, while blue points are placed in the remaining space. Finally, the last dataset is shaped like a cross.

These two datasets are interesting because they are not linearly separable. The cross can be separated with several lines, which increases the complexity of the decision boundary, while the donut is simply non-linear.

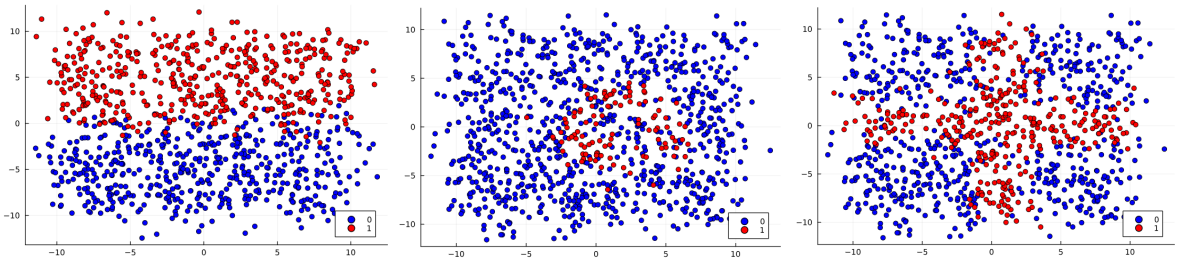

Figure 8: Examples of datasets (linear problem / donut / cross)

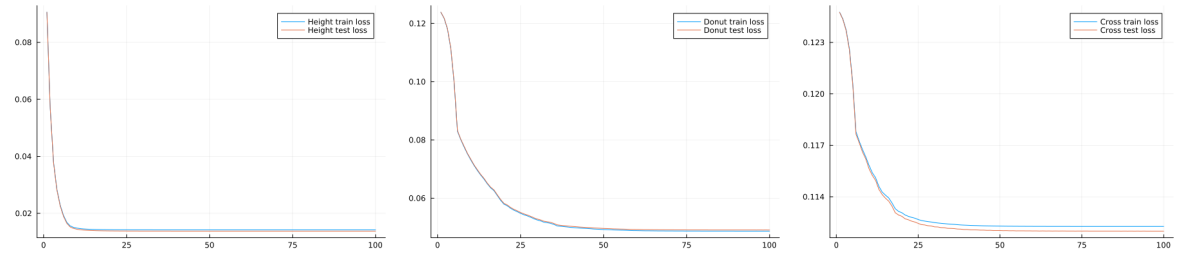

Figure 9: Evolution of the loss

As we can see on this 3 different examples, the behaviour of the classic Resnet and the Resnet with optimal control are pretty similar. The similarity of the behaviours was predictable. Indeed in the Resnet with optimal control we used the Euler method. We expect to have better results by using more complex methods such as Runge Kutta methods.

6 Conclusion

Through this initial approach, we have gained a comprehensive mathematical understanding of transformers, as well as insight into the inner workings of a simplified transformer model such as nanoGPT. This exploration has not only deepened our theoretical grasp of transformer architectures but also provided a practical foundation for analyzing their behavior. Moreover, our exploration has allowed us to understand how optimal control techniques can be effectively applied in the field of neural networks, offering a new perspective on model optimization. By leveraging these tools, we have exploited their potential to optimize the trainable parameters of a simple neural network such as ResNet. Further improvements are possible by using more complex numerical integration schemes, such as Runge-Kutta methods, which could enhance the stability of gradient computations. This initial exploration suggests that this technique could be applied to more complex models like nanoGPT, potentially improving the efficiency and performance of transformer-based models.

7 Resources

[1] Jamell Alvah Samuels. One-hot encoding and two-hot encoding: An introduction. *ResearchGate*, 2024. URL https://www.researchgate.net/publication/377159812_One-Hot_Encoding_and_Two-Hot_Encoding_An_Introduction.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017. URL https://arxiv.org/abs/1706.03762. Presented at the 31st Conference on Neural Information Processing Systems (NeurIPS 2017).

[3] Sebastian Raschka. Understanding and coding the self-attention mechanism of large language models, 2023. URL https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html.

[4] Jamie Ryan Kiros Jimmy Lei Ba and Geoffrey E. Hinton. Layer normalization, 2016. URL https://arxiv.org/pdf/1607.06450.

[5] A. Krizhevsky I. Sutskever G. E. Hinton, N. Srivastava and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012. URL https://arxiv.org/pdf/1207.0580.

[6] Andrej Karpathy. Nanogpt: The simplest, fastest repository for training/finetuning medium-sized gpts. https://github.com/karpathy/nanoGPT, 2023.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

[8] Martin Benning, Elena Celledoni, Matthias J. Ehrhardt, Brynjulf Owren, and Carola-Bibiane Schönlieb. Deep learning as optimal control problems. *Procedia Computer Science*, 192:3145–3154, 2021. doi: 10.1016/j.procs.2021.09.077. URL https://www.sciencedirect.com/science/article/pii/S2405896321006029.

[9] Martin Benning, Elena Celledoni, Matthias J. Ehrhardt, Brynjulf Owren, and Carola-Bibiane Schönlieb. Deep learning as optimal control problems: Models and numerical methods. 2019. URL https://arxiv.org/abs/1904.05657.