Denoising Diffusion Probabilistic Models: Architecture Concepts

Giovani Tavares
giovanitavares@outlook.com

University of Sao Paulo — January 3, 2025

Motivation

I've been studying and using generative AI models for a while now. I have always put at least some effort in understanding Large Language Model's (LLMs) architecture and training process, but state-of-the-art (SOTA) image generative models have always been kept as a future topic to be grasped. I say that the personal main reason for this is that I hadn't had the opportunity to use this models professionally until recently, so I had little interest in dedicating some time for them.

By the last months of 2024, I was invited by my boss to give a worshop on Generative Adversarial Networks (GANs) and Denoising Diffusion Probabilistic Models (DDPMs) during the Third School on Data Science and Machine Learning that took place in the International Centre for Theoretical Physics (ICTP) in the São Paulo State University (UNESP, Sao Paulo, Brazil) in December 6th, 2024. The workshop was my greatest academic challenge when it comes to presenting. The public was made of high performing graduate and postgraduate students from the best universities in South America.

As mentioned, DDPMs was one of the topics I had to give a workshop on. In order to do so, I had to study its architecture and training procedure. The concepts I learned are all presented here.

1 What are DDPMs?

Denoising Diffusion Probabilistic Models (DDPMs) are models capable of predicting *noise* from a noisy input. By using such prediction, a sampling algorithm can be used to remove the noise from the input which results in a denoised output. DDPMs were first introduced in 2015 by Sohl-Dickstein and other USA researchers in [2], but were made popular by Jonathan Ho and others in [1]. The latter paper used such model to generate images outputs from pure random noise, which is basically an image generation model.

DDPMs are made of two processes: a **forward** and a **reversion** process. The former is responsible for gradually adding noise to a image by sampling from a normal distribution according to a Markov Chain. The latter removes added noise by sampling from another normal distribution. In simple terms, the training of DDPMs involve learning the reversion process' distribution's parameters.

1.1 Forward Process

The process of adding noise to an input image (x_0) is a Markov Chain that generates a noisier image x_t from a less noisy image x_{t-1}. In [1] and here, an x's subscript represents its position in the Markov Chain so that the greater an x's subscript the noisier it is. Hence, x_t represents the result of adding noise to x_{t-1} by transitioning it once in the Markov Chain.

From the original DDPM paper, we know that in the forward process, a sample x_t is produced by adding noise to a sample x_{t-1} according to a normal distribution defined below:

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

Using $q(x_t|x_{t-1})$ is not very feasible from an implementation point of view, because it makes the production of each noisy sample x_t dependent on x_{t-1}, which makes it necessary to make t transitions starting from x_0 in order to get to the x_t sample. **Ideally, one would need a single transition from x_0 to get to x_t, with $t > 0$.**

The authors of [1] achieves such ideal scenario by defining a cumulative noise α_t presented below:

Definition 1.1 (Cumulative Noise)**.** *The Cumulative Noise (α_t) added to an input x_0 up to the t-th step is defined as:*

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

$$\bar{\alpha_t} := \prod_{s=1}^{t} a_s \tag{3}$$

$$\textit{which leads to} \tag{4}$$

$$q(x_t|x_0) := \mathcal{N}(x_t; \sqrt{\bar{\alpha_t}} \times x_0; (1 - \bar{\alpha_t})I) \tag{5}$$

Definition 1.1 permits the sampling of x_t to be done in a single transition from x_0.

According to the original DDPM paper, the **Variance Schedule** $\beta_1, ..., \beta_T$ sequence that defines the noisy images distribution can be learned or held constant as hyperparameters. The latter approach is the one we are going to use, in which the variance shedule is an arithmetic progression with $step = 0.02$, $\beta_1 = 1e - 4$ and $T = 1000$. Hence, **our Forward Process is gonna be a Markov Chain of size 1000 steps**.

```python
import torch.nn  as nn

class DDPMVarianceScheduler(nn.Module):
        def __init__(self, num_time_steps: int=1000):
                super().__init__()

                # The variance values in the schedule/sequence
                self.beta = torch.linspace(1e-4, 0.02, num_time_steps,
                    requires_grad=False)

                # The cumulative noise used to sample x_t by from x_0
                self.alpha = torch.cumprod(1 - self.beta, dim=0).
                    requires_grad_(False)

        def forward(self, t):
                return self.beta[t], self.alpha[t]
```

Listing 1: Variance Schedule

1.2 Reverse Process

As previously mentioned, the reverse process is resposible for removing the noise from an input. Hence, this process is responsible for generating x_0 given a noisy input x_T. It is a *Markov Chain* with learned Gaussian transitions defined below:

Definition 1.2 (Reverse Process Transitions)**.** *The Reverse Process is a Markov Chain with the following transitions:*

$$p(x_T) = \mathcal{N}(x_T; 0; 1) \tag{6}$$

$$p_\theta(x_{0:T}) := p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t) \tag{7}$$

$$p_\theta(x_{t-1}|x_t) := \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t); \Sigma_\theta(x_t, t)) \tag{8}$$

Notice that the definition above is simply a Markov Chain with a Standard Gaussian initial state and Gaussian transition distributions parametrized by μ_θ and Σ_θ. We want to create a model that is capable of recovering the best x_0 possible given a set of observed variables $Z := \mathbf{x_{1:T}}$. Theoretically, this can be achieved if our model learns the following distribution.

Definition 1.3 (Reverse Process Posterior). *The Reverse Process is responsible for modeling the following posterior:*

$$p(\mathbf{x}_0) = \int p(\mathbf{x_0}, \mathbf{x_{1:T}})\mathbf{d_{1:T}} \tag{9}$$

$$\tag{10}$$

We see that the computation of $p(\mathbf{x_0})$ is highly complex and intractable due to its multidimensionality, which makes the training based on optimizating $p(\mathbf{x_0})$ directly infeasible. To solve this problem, the author's of the DDPM paper used the *Variational Lower Bound* of the expectation log-likelihood funtion, also called evidence lower bound (ELBO).

1.2.1 Variational Lower Bound / Evidence Lower Bound / ELBO

The Variational Lower Bound is a tight lower bound that limits $log(p(\mathbf{x_0}))$ from below. Hence, when $p(\mathbf{x_0})$ is intractable as in the case of DDPMs, one can always maximize such lower bound as a means to ensure that $log(p(\mathbf{x_0}))$ is as large as possible. Such lower bound is often called **ELBO** or simply L and will be demonstrated using two different approaches: the **Jensen's Inequality** and the **KL Divergence**.

Definition 1.4 (Variational Lower Bound - Jensen's Inequality). *Let's use the Rule Of Total Probability to find an upper bound for the log-likelihood function.*

$$log[p_\theta(\mathbf{x_0})] = log\int_{\mathbf{x_{1:T}}} p(\mathbf{x_0}, \mathbf{x_{1:T}})d\mathbf{x_{1:T}} \tag{11}$$

$$log[p_\theta(\mathbf{x_0})] = log\int_{\mathbf{x_{1:T}}} p(\mathbf{x_0}, \mathbf{x_{1:T}})\frac{q(\mathbf{x_{1:T}}|\mathbf{x_0})}{q(\mathbf{x_{1:T}}|\mathbf{x_0})}d\mathbf{x_{1:T}} \tag{12}$$

$$log[p_\theta(\mathbf{x_0})] = log(\mathbb{E}_q\left[\frac{p(\mathbf{x_{0:T}})}{q(\mathbf{x_{1:T}}|\mathbf{x_0})}\right]) \tag{13}$$

$$\textit{the Jensen's inequality tells us} \tag{14}$$

$$f(\mathbf{E}(\mathbf{X})) \geq \mathbf{E}(f(\mathbf{X}))) \tag{15}$$

$$\textit{for any concave function f.} \tag{16}$$

$$\textit{log is concave, hence:} \tag{17}$$

$$log[p_\theta(\mathbf{x_0})] \geq \mathbb{E}_q\left[log\frac{p(\mathbf{x_{0:T}})}{q(\mathbf{x_{1:T}}|\mathbf{x_0})}\right] \tag{18}$$

$$\textit{If we define the Variational Lower Bound L as:} \tag{19}$$

$$L := \mathbb{E}_q\left[log\frac{p(\mathbf{x_{0:T}})}{q(\mathbf{x_{1:T}}|\mathbf{x_0})}\right] \tag{20}$$

$$\implies -log[p_\theta(\mathbf{x_0})] \leq -L \tag{21}$$

Definition 1.5 (Variational Lower Bound - KL Divergence). *In order to reverse the forward process, we need that the forward process' distribution $q(\mathbf{x_{1:T}}|\mathbf{x_0})$ is as close to $p(\mathbf{x_{1:T}}|\mathbf{x_0})$ as possible. We can use the Kullback-Leibler (KL) divergence between q and p ($\mathbf{D_{KL}}$) to evaluate their difference as find a bound to $log[p_\theta(\mathbf{x}_0)]$.*

$$\mathbf{D_{KL}}\big[q(\mathbf{x_{1:T}}|\mathbf{x_0})||p(\mathbf{x_{1:T}}|\mathbf{x_0})\big] := \mathbb{E}_q\big[log(q(\mathbf{x_{1:T}}|\mathbf{x_0}) - log(p(\mathbf{x_{1:T}}|\mathbf{x_0}))\big] \tag{22}$$

using the Bayes' Rule we can write (23)

$$p(\mathbf{x_{1:T}}|\mathbf{x_0}) = \frac{p(\mathbf{x_{1:T}}, \mathbf{x_0})}{p(\mathbf{x_0})} \tag{24}$$

$$\implies \mathbf{D_{KL}}\big[q(\mathbf{x_{1:T}}|\mathbf{x_0})||p(\mathbf{x_{1:T}}|\mathbf{x_0})\big] = \mathbb{E}_q\big[log(q(\mathbf{x_{1:T}}|\mathbf{x_0}) - log(p(\mathbf{x_{1:T}}|\mathbf{x_0})) + log(p(\mathbf{x_0}))\big] \tag{25}$$

the prior of the latent variables does not depend on q (26)

$$\mathbf{D_{KL}}\big[q(\mathbf{x_{1:T}}|\mathbf{x_0})||p(\mathbf{x_{1:T}}|\mathbf{x_0})\big] = \mathbb{E}_q\big[log(q(\mathbf{x_{1:T}}|\mathbf{x_0}) - log(p(\mathbf{x_{1:T}}, \mathbf{x_0}))\big] + log(p(\mathbf{x_0})) \tag{27}$$

$$\implies log(p(\mathbf{x_0})) = \mathbf{D_{KL}}\big[q(\mathbf{x_{1:T}}|\mathbf{x_0})||p(\mathbf{x_{1:T}}|\mathbf{x_0})\big] - \mathbb{E}_q\big[log(q(\mathbf{x_{1:T}}|\mathbf{x_0}) - log(p(\mathbf{x_{1:T}}, \mathbf{x_0}))\big] \tag{28}$$

$$log(p(\mathbf{x_0})) = \mathbf{D_{KL}}\big[q(\mathbf{x_{1:T}}|\mathbf{x_0})||p(\mathbf{x_{1:T}}|\mathbf{x_0})\big] + \mathbb{E}_q\big[log(p(\mathbf{x_{1:T}}, \mathbf{x_0}) - log(q(\mathbf{x_{1:T}}|\mathbf{x_0}))\big] \tag{29}$$

$$\textit{but } \mathbf{D_{KL}}\big[q(\mathbf{x_{1:T}}|\mathbf{x_0})||p(\mathbf{x_{1:T}}|\mathbf{x_0})\big] \geq 0 \tag{30}$$

$$\implies -log(p(\mathbf{x_0})) \leq \mathbb{E}_q\big[-log\frac{p(\mathbf{x_{0:T}})}{q(\mathbf{x_{1:T}}|\mathbf{x_0})}\big] \tag{31}$$

If we define the Variational Lower Bound L as: (32)

$$L := \mathbb{E}_q\left[log\frac{p(\mathbf{x_{0:T}})}{q(\mathbf{x_{1:T}}|\mathbf{x_0})}\right] \tag{33}$$

$$\implies -log[p(\mathbf{x_0})] \leq -L \tag{34}$$

We have found lower bound L for the log-likelihood function that is tractable if rewritten properly.

1.2.2 Reverse Process Loss Function Analysys

Having ELBO (L) as a lower bound for the log likelihood function means we can train our reverse process model to maximize L, i.e., L is a candidate for our reversor's loss function. In order to do so, further algebraic manipulation must be performed with it in order to make it tractable. We demonstrate such manipulation here.

Definition 1.6 (Reverse Process' Loss Derivation)**.** *In order to build the Reverse Process' loss function, we need to remember that it is a Markov Chain and use this fact to manipulate L.*

$$L = \mathbb{E}_q\left[log\frac{p(\mathbf{x_{0:T}})}{q(\mathbf{x_{1:T}}|\mathbf{x_0})}\right] \tag{35}$$

The forward and reverse processes are Markov Chains, so (36)

$$log\frac{p(\mathbf{x_{0:T}})}{q(\mathbf{x_{1:T}}|\mathbf{x_0})} = log\frac{p(\mathbf{x_T})\prod_{t=1}^{T}p_\theta(\mathbf{x_{t-1}}|\mathbf{x_t})}{q(\mathbf{x_1}|\mathbf{x_0})\prod_{t=2}^{T}q(\mathbf{x_t}|\mathbf{x_{t-1}}, \mathbf{x_0})} \tag{37}$$

$$= log\frac{p(\mathbf{x_T})\prod_{t=1}^{T}p_\theta(\mathbf{x_{t-1}}|\mathbf{x_t})}{q(\mathbf{x_T}|\mathbf{x_0})\prod_{t=2}^{T}q(\mathbf{x_{t-1}}|\mathbf{x_t}, \mathbf{x_0})} \tag{38}$$

$$= log\frac{p(\mathbf{x_T})p_\theta(\mathbf{x_0}|\mathbf{x_1})\prod_{t=2}^{T}p_\theta(\mathbf{x_{t-1}}|\mathbf{x_t})}{q(\mathbf{x_T}|\mathbf{x_0})\prod_{t=2}^{T}q(\mathbf{x_{t-1}}|\mathbf{x_t}, \mathbf{x_0})} \tag{39}$$

$$= log\frac{p(\mathbf{x_T})}{q(\mathbf{x_T}|\mathbf{x_0})} + logp_\theta(\mathbf{x_0}|\mathbf{x_1}) + \sum_{t=2}^{T}log\frac{p_\theta(\mathbf{x_{t-1}}|\mathbf{x_t})}{q(\mathbf{x_{t-1}}|\mathbf{x_t}, \mathbf{x_0})} \tag{40}$$

$$= log\frac{p(\mathbf{x_T})}{q(\mathbf{x_T}|\mathbf{x_0})} + logp_\theta(\mathbf{x_0}|\mathbf{x_1}) + \sum_{t=2}^{T}\mathbf{D_{KL}}\big[p_\theta(\mathbf{x_{t-1}}|\mathbf{x_t})||q(\mathbf{x_{t-1}}|\mathbf{x_t}, \mathbf{x_0})\big] \tag{41}$$

If we pay attention to equation 41*'s terms above, we see that the first term is parameter free, because* $p((x_T))$ *is fixed and defined as a Gaussian, while* $q((x_T|x_0))$ *is also Gaussian from the definition of the forward process. Hence, we are left with the second and third terms.*

As previously mentioned, we are interested in maximizing L. Using the equation 41*, we see that doing so is equivalent to minimizing the KL Divergence between* $p_\theta(\mathbf{x_{t-1}}|\mathbf{x_t})$ *and* $q(\mathbf{x_{t-1}}|\mathbf{x_t}, \mathbf{x_0})$*. In order to see how to compute such divergence, let's manipulate the equation terms even further.*

$$q(\mathbf{x_{t-1}}|\mathbf{x_t}, \mathbf{x_0}) = \frac{q(\mathbf{x_t}|\mathbf{x_{t-1}}, \mathbf{x_0})q(\mathbf{x_{t-1}}|\mathbf{x_0})}{q(\mathbf{x_t}|\mathbf{x_0})} \tag{42}$$

$$\textit{we know the q distribution from Definition 1.1, hence} \tag{43}$$

$$q(\mathbf{x_{t-1}}|\mathbf{x_t}, \mathbf{x_0}) \textit{ is a product of known Gaussians over another known Gaussian that results in} \tag{44}$$

$$\mu_q(\mathbf{x_t}, \mathbf{x_0}) = \frac{(1-\bar{\alpha}_{t-1})\sqrt{\alpha_t}\mathbf{x_t} + (1-\alpha_t)\sqrt{\bar{\alpha}_{t-1}}\mathbf{x_0}}{(1-\bar{\alpha}_t)} \tag{45}$$

$$\Sigma_q(t) = \frac{(1-\alpha_t)(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbb{I} \tag{46}$$

$$\implies q(\mathbf{x_{t-1}}|\mathbf{x_t}, \mathbf{x_0}) = \mathcal{N}(x_{t-1}; \mu_q(\mathbf{x_t}, \mathbf{x_0}); \Sigma_q(t)) \tag{47}$$

References

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

[2] Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. *CoRR*, abs/1503.03585, 2015.

- **Sentence I:** The chef, *who had prepared the meal with great care*, served the dish to the guests.
- **Sentence II:** The chef served the dish to the guests, *who had prepared the meal with great care*.

Notice how the meaning of the clause "*who had prepared the meal with great care*" changes between the sentences. In **Sentence I**, it refers to the chef, while in **Sentence II**, it refers to the guests.

This significant difference in meaning is entirely dependent on the clause's position within the sentences. The semantics of a clause within a sentence is heavily influenced its position within the sentence. Therefore, it is essential for transformer models to encode positional information in their input representations. This is one of the key roles of the Sinusoidal Embedding function, which takes as input a sequence of tokens and outputs the position-informed embedding for such sequence.

1.3 What Properties Should Sinusoidal Embeddings (SE) Should Apply?

1. **Periodicity:** the model trained with sinusoidal embeddings should be able to capture the relative positions of tokens effectively, which means that the distances between the embeddings of a pair of clauses should not depend on their absolute position within the longer sentence. This is achieved by making the sinusoidal embeddings functions periodic, which was to be expected from the name of the function.

2. **Unique Representation (Injective Function):** this one is straight forward from the fact that sinusoidal embeddings must represent the positions of tokens within a sentence. This means that sets of tokens in different positions should not be mapped to the same output, otherwise different positions would have the same representation.

3. **Scale Invariance:** sinusoidal embeddings should represent tokens positions consistently regardless of the sequence length. This property is crucial for handling sequences of varying lengths in a transformer model including those that are longer than the ones the model were trained on. Essentialy, the scale invariance property says that the distance of two inputs x_t and x_{t-k} should be similar among different values of t. In other words, $x_t - x_{t-k}$ should not depend on t.

4. **Linearity:** this property is somewhat related to the scale invariance property. For the Sinusoidal Embedding function to be linear is good, because functions having such property are more easily learned by neural networks. Moreover, if the Sinusoidal Embedding Function is linear we also achieve

the scale invariance property. This is true, because if such Sinusoidal Embedding function (SE) is linear, for any pair of sets of tokens separated by a distance of k, say X_t, X_{t+k}, there is a linear transformation M such that $M \times SE(X_t) = SE(X_{t+k})$. Hence, in order to represent X_{t+k}, the model must only learn the linear transformation M regardless of the value of k.

1.4 How are Sinusoidal Embeddings Defined?

The author's of the *Attention is All You Need* paper define the Sinusoidal Embeddings Function (SE) like the following.

Definition 1.7 (Sinusoidal Embeddings). *For a position pos in the sequence and a dimension i (where i ranges from 0 to $\frac{d}{2} - 1$), the embedding is given by:*

$$SE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{48}$$

$$SE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{49}$$

$$\tag{50}$$

This function can be vectorially expressed as the following:

$$\mathbf{SE}(pos) = \left[\sin\left(pos \cdot e^{-\frac{2i\ln(10000)}{d}}\right), \cos\left(pos \cdot e^{-\frac{2i\ln(10000)}{d}}\right)\right]_{i=0}^{\frac{d}{2}-1} \tag{51}$$

where:

- pos is the position in the sequence.
- i is the dimension index.
- d is the dimensionality of the embeddings.

Notice that SE is essentialy a function that outputs sin values to the even dimensions of an input x_t and cos for the odd ones with exponentially decreasing frequencies.

In the next section we will use the definition of SE function to verify the validity of the previous 4 properties it should have. Some verification will be done analitically and others will be done using $Python$.

2 Properties Verification

In this section, we will be checking each of the 4 presented properties of the Sinusoidal Embeddings using analytical techniques.

2.1 Peridiocity

This peridiocity of the SE function is straight-forward. Different dimensions i and $i+k$ of an input $x_t \in \mathbb{R}^d$ with position pos will have the same values output by the sin and cos because such functions are periodic.

Info: One question that a reader might have is whether the peridiocity property does not contradict the injectivity one. The answer for that is no. The peridiocity property is observed at a single dimension level, which means $SE_{(pos,2i)}$ that outputs a single value, is be periodic. The injectivity, on the other hand, is observed at the entire embedding level, which means $SE_{(pos)}$, that outputs a vector (embedding) of dimension d is injective.

2.2 Unique Representation (Injective Function)

We are gonna prove that two sequences from different positions are **not** mapped to the same vector/output, using a *proof by contradiction*. Let's assume that two sequences x_{t_1} and x_{t_2}, with different positions t_1 and t_2, respectivelly, are mapped to the same output using the previously SE function. This assumption implies the following **for any dimension i**

$$SE(x_{t_1}) = SE(x_{t_2}) \implies SE(t_1, 2i) = SE(t_2, 2i) \text{ and } SE(t_1, 2i+1) = SE(t_2, 2i+1) \tag{52}$$

$$\implies \sin\left(\frac{t_1}{10000^{\frac{2i}{d}}}\right) = \sin\left(\frac{t_2}{10000^{\frac{2i}{d}}}\right) \text{ and } \cos\left(\frac{t_1}{10000^{\frac{2i}{d}}}\right) = \cos\left(\frac{t_2}{10000^{\frac{2i}{d}}}\right) \tag{53}$$

For the last implication to be true, either the arguments of the sin and cos functions are the exact same or they are separated by $2\pi k$. We know they are not the same, because $t_1 \neq t_2$. Therefore, we're left with the condition:

$$|\frac{t_1}{10000^{\frac{2i}{d}}} - \frac{t_2}{10000^{\frac{2i}{d}}}| = 2\pi k \tag{54}$$

$$\implies |\frac{t_1 - t_2}{10000^{\frac{2i}{d}}}| = 2\pi k \tag{55}$$

$$\implies |t_1 - t_2| = 2\pi k 10000^{\frac{2i}{d}} \tag{56}$$

$$\tag{57}$$

Since t_1 and t_2 are integers that represent the sequences positions, and $10000^{\frac{2i}{d}}$ is a positive real number, the right side of the equation $|t_1 - t_2| = 2\pi k \cdot 10000^{\frac{2i}{d}}$ must also be an integer. However, $2\pi k \cdot 10000^{\frac{2i}{d}}$ is generally not an integer because 2π is an irrational number, which leads us to a contradiction.

Hence, the initial assumption $SE(x_{t_1}) = SE(x_{t_2})$ must be false, which let's us say that **two sequences x_{t_1} and x_{t_2}, with different positions t_1 and t_2, respectivelly, are not mapped to the same output using the previously SE function.**

2.3 Linearity & Scale Invariance

As previously mentioned, the scale invariance property is a consequence of $SE(pos)$'s linearity. Hence, by proving that $SE(pos)$ is linear we also prove that such function is scale invariant.

We need to find $M \in \mathbb{R}^{d \times d}$ such that $M \times SE(x_t) = SE(x_{t+k})$. We have the following system of equations:

$$\begin{bmatrix} m_{00} & m_{01} & \cdots & m_{0(d-1)} \\ m_{10} & m_{11} & \cdots & m_{1(d-1)} \\ \vdots & \vdots & \ddots & \vdots \\ m_{(d-1)0} & m_{(d-1)1} & \cdots & m_{(d-1)(d-1)} \end{bmatrix} \begin{bmatrix} sin(\omega_0 t) \\ cos(\omega_0 t) \\ \vdots \\ sin(\omega_{(d-2)/2} t) \\ cos(\omega_{(d-2)/2} t) \end{bmatrix} = \begin{bmatrix} sin(\omega_0 (t+k)) \\ cos(\omega_0 (t+k)) \\ \vdots \\ sin(\omega_{(d-2)/2} (t+k)) \\ cos(\omega_{(d-2)/2} (t+k)) \end{bmatrix} \tag{58}$$

$$m_{00} \sin(\omega_0 t) + m_{01} \cos(\omega_0 t) + \cdots + m_{0(d-1)} \cos(\omega_{d-1} t) = \sin(\omega_0 t) \cos(\omega_0 k) + \sin(\omega_0 k) \cos(\omega_0 t) \tag{59}$$

$$m_{10} \sin(\omega_0 t) + m_{11} \cos(\omega_0 t) + \cdots + m_{1(d-1)} \cos(\omega_{d-1} t) = \cos(\omega_0 t) \cos(\omega_0 k) - \sin(\omega_0 k) \sin(\omega_0 t) \tag{60}$$

$$\vdots$$

$$m_{(d-1)0} \sin(\omega_0 t) + m_{(d-1)1} \cos(\omega_0 t) + \cdots + m_{(d-1)(d-1)} \cos(\omega_{(d-2)/2} t) = \cos(\omega_{(d-2)/2} t) \cos(\omega_{(d-2)/2} k) - \sin(\omega_{(d-2)/2} k) \sin \tag{61}$$

This system has a solution:

$$\begin{aligned}
m_{00} &= \cos(\omega_0 k) && (62)\\
m_{01} &= \sin(\omega_0 k) && (63)\\
m_{02} &= m_{03} = \ldots = m_{0(d-1)} = 0 && (64)\\
m_{10} &= -\sin(\omega_0 k) && (65)\\
m_{11} &= \cos(\omega_0 k) && (66)\\
m_{12} &= m_{13} = \ldots = m_{1(d-1)} = 0 && (67)\\
m_{22} &= \cos(\omega_1 k) && (68)\\
m_{23} &= \sin(\omega_1 k) && (69)\\
m_{20} &= m_{21} = m_{24} = \ldots = m_{2(d-1)} = 0 && (70)\\
m_{32} &= -\sin(\omega_1 k) && (71)\\
m_{33} &= \cos(\omega_1 k) && (72)\\
m_{30} &= m_{31} = m_{34} = \ldots = m_{3(d-1)} = 0 && (73)\\
&\vdots\\
m_{(d-2)(d-2)} &= \cos(\omega_{(d-2)/2} k) && (74)\\
m_{(d-2)(d-1)} &= \sin(\omega_{(d-2)/2} k) && (75)\\
m_{(d-2)0} &= m_{(d-2)1} = m_{(d-2)2} = \ldots = m_{(d-2)(d-3)} = 0 && (76)\\
m_{(d-1)(d-2)} &= -\sin(\omega_{(d-2)/2} k) && (77)\\
m_{(d-1)(d-1)} &= \cos(\omega_{(d-2)/2} k) && (78)\\
m_{(d-1)0} &= m_{(d-1)1} = m_{(d-1)2} = \ldots = m_{(d-1)(d-3)} = 0 && (79)\\
& && (80)
\end{aligned}$$

Which lets us define M as:

$$M = \begin{bmatrix} \cos(\omega_0 k) & \sin(\omega_0 k) & 0 & 0 & \cdots & 0 & 0 \\ -\sin(\omega_0 k) & \cos(\omega_0 k) & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos(\omega_1 k) & \sin(\omega_1 k) & \cdots & 0 & 0 \\ 0 & 0 & -\sin(\omega_1 k) & \cos(\omega_1 k) & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos(\omega_{(d-2)/2} k) & \sin(\omega_{(d-2)/2} k) \\ 0 & 0 & 0 & 0 & \cdots & -\sin(\omega_{(d-2)/2} k) & \cos(\omega_{(d-2)/2} k) \end{bmatrix} \quad (81)$$

We found a matrix $M \in \mathbb{R}^{d \times d}$ such that $M \times SE(x_t) = SE(x_{t+k})$. Hence, SE is a linear function. Moreover, notice how **M does not depend on t**, only on k. This is what gives us the scale invariability property.

3 Visualizing the Sinuspoidal Embeddings Properties With *Python*

In this section, we will write some $Python$ functions and classes to visualize the 4 cited properties of Sinusoidal Embeddings. Visualization is a great way to grasp such function's behaviour without having to necessarily prove it (even though you can always come back to this post with you're interested in the proofs).

3.1 Sinusoidal Embedding Definition With Pytorch

The sinusoidal embeddings module will store a multi-embedding tensor with $shape = (max\_pos, embed\_dim)$, where max_pos represents the maximum position we are interested in representing. This way, each of such tensor's row represents a the sinusoidal embedding for a position pos such that $SE(pos), 0 \leq pos < max\_pos$.

```

import pytorch.nn as nn

class SinusoidalEmbeddings(nn.Module):

    def __init__(self, max_pos:int, embed_dim: int):
        super().__init__()
        # Returns a tensor with shape (time_steps, 1).
        positions = torch.arange(max_pos).unsqueeze(1).float()

        # Creates a tensor with shape (embed_dim //2,). We just need
        # half of the dimensions of the input embeddings to compute all
        # of the sinusoidal embeddings frequencies
        dimensions = torch.arange(start = 0, end = embed_dim, step = 2).float()

        # Compute the frequencies vector
        frequencies = torch.exp(dimensions * -(math.log(10000.0) / embed_dim))

        # Initialize the embeddings tensor with shape (time_steps, embed_dim)
        embeddings = torch.zeros(time_steps, embed_dim, requires_grad=False)

        # Apply sin to even indices (0, 2, 4, ...) of the input embeddings
        embeddings[:, 0::2] = torch.sin(positions * frequencies)

        # Apply cos to odd indices (1, 3, 5, ...) of the input embeddings
        embeddings[:, 1::2] = torch.cos(positions * frequencies)

        self.embeddings = embeddings

    def forward(self, x, t):
        embeds = self.embeddings[t].to(x.device)
        return embeds[:, :, None, None]
```

Listing 2: Sinusoidal Embedding Module Definition

3.2 Sinusoidal Embeddings Periodicity

As previously mentioned, the periodicity of Sinusoidal Embeddings is observed in each of its dimensions. Hence, we need to plot its dimensions values for different positions to see their periodic behavior.

```

max_pos = 100

# Our embeddings will only have 4 dimension
embed_dim = 4
sinusoidal_embeddings = SinusoidalEmbeddings(max_pos, embed_dim)

# Generate embeddings for a range of time steps
embeddings = sinusoidal_embeddings.embeddings
```

```

# Convert embeddings to numpy for plotting
embeddings_np = embeddings.numpy()

# Plot the sunosoidal embeddings for different time steps
plt.figure(figsize=(14, 8))
for i in range(embed_dim):
        plt.plot(embeddings_np[:, i], label=f"Dim {i} (i = {i//2})")

plt.title("SE(pos, 2i)")
plt.xlabel("pos")
plt.ylabel("Value")
plt.legend(loc="upper right", bbox_to_anchor=(1.15, 1))
```

Listing 3: Generating the plot of the embedding's dimensions for in different positions

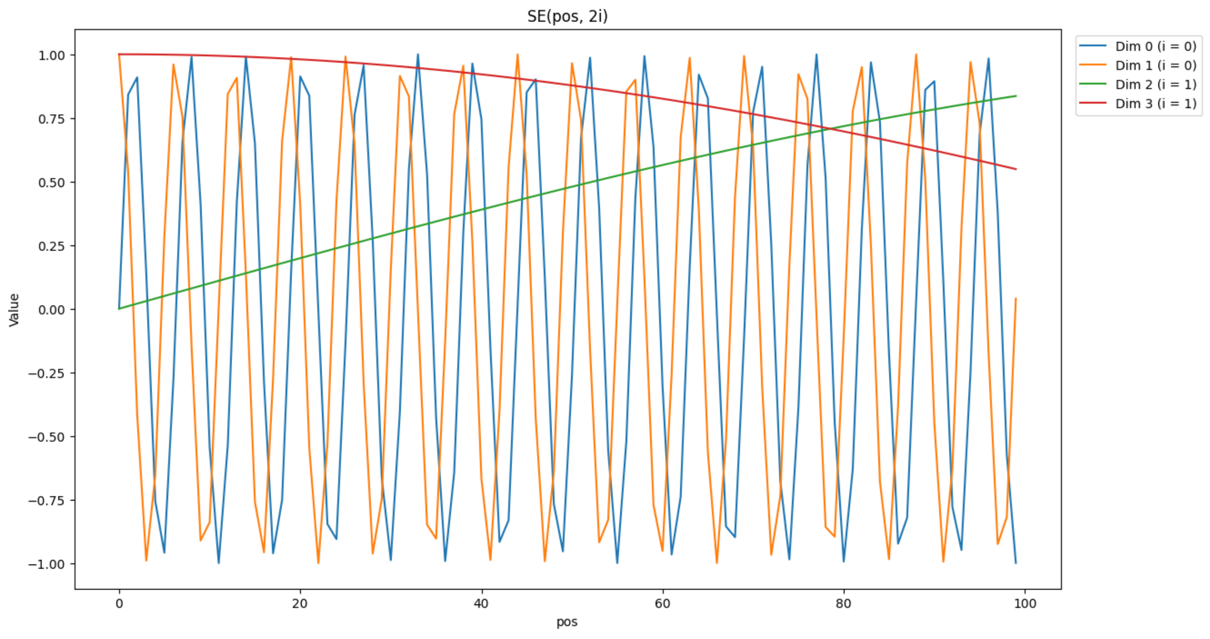

Figure 1: The embedding's dimensions values for the SE function

As noticed in the plot above, the value of the function $SE(pos, 2i)$ defined in 1.7 repeats itself in a frequency that is inversely proportional to i, which is the dimension being represented (an index in the multidimensional embedding). As a consequence, **the higher the dimension of our model's embeddings (previously called d and called $embed\_dim$ in the Sinusoidal Embedding module), the less SE values vary.**

Intuitively, such consequence means that embeddings close to each other (they represent sentences in not very distant positions), will have their differences captured in lower dimensions, because their higher dimensions are likely to be very similar. The exact opposite is true for embeddings that represent sentences that are far away from each other. Let's visualize that by plotting different embeddings' heatmaps.

```

import torch

# We'll create embeddings with many dimensions to better see the
# frequency decay effect
max_pos = 100
embed_dim = 128
sinusoidal_embeddings = SinusoidalEmbeddings(max_pos, embed_dim)

```

```
x = torch.zeros(embed_dim)
results = torch.zeros(max_pos, embed_dim)
for pos in range(max_pos):
results[pos] =  x + sinusoidal_embeddings.embeddings[pos]

tensor_np = results.numpy()
# Plot the heatmap using matplotlib
plt.figure(figsize=(16, 6))
plt.imshow(tensor_np, aspect='auto', cmap='RdBu')
plt.colorbar(label='')
plt.xlabel('2i')
plt.ylabel('pos')
plt.title('SE(pos, 2i)')
plt.show()
```

Listing 4: Generating the plot of the sinusoidal embeddings for different positions

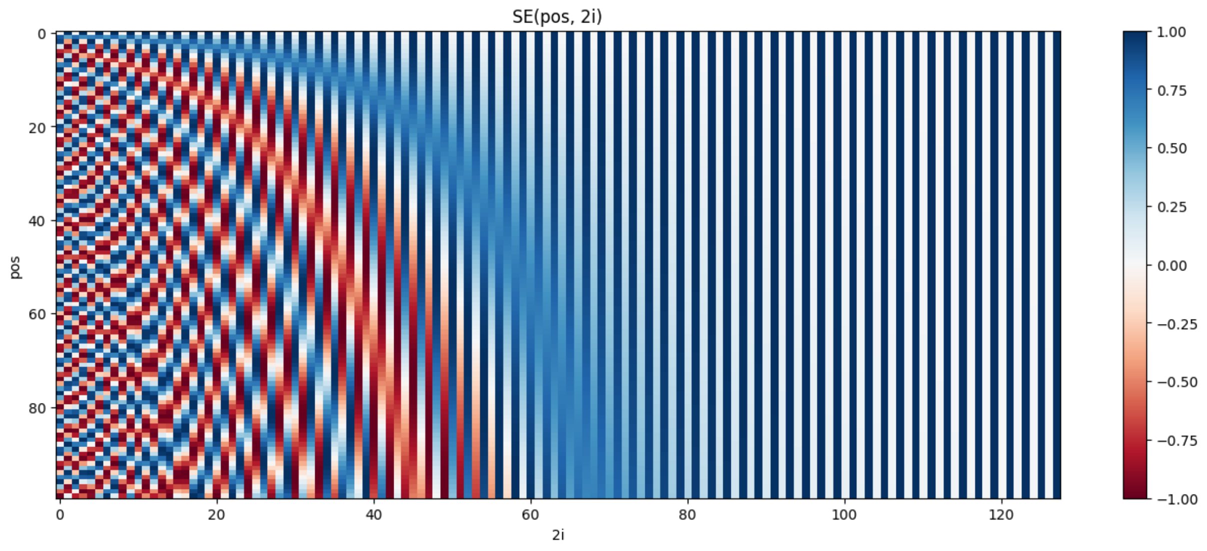

Figure 2: The SE function's frequency gets lower as the dimension grows

In the plot above, we see that embeddings with close positions (two close values in the pos axis) have different values of $SE(pos, 2i)$ for very small $2i$ (lower dimensions) while their values for higher dimensions are similar. On the other hand, if we pick two embeddings with very distant pos, they might differ from each other only in higher dimension. What this means is that in order to represent longer sentences (that contain positions very distant from each other), our model needs to have more dimensions. **The longer the input sentences, the higher the model's dimensions need to be.**

3.3 Unique Representativiness (SE is an injective function)

Figure 2 shows us that no two rows are the same because of the exponential decay of the frequencies with the increase of the dimension (increase of $2i$). As each row represents embeddings with different positions, what the figure is essentialy showing us is that two embeddings with different positions will never have the same representation, i.e., SE is injective.

3.4 Linearity & Scale Invariance

As previously mentioned, to be scale invariant, SE must be such that the distance between $SE(pos)$ and $SE(pos+k)$ must be the same as the one between $SE(0)$ and $SE(k)$. In order to check that, we'll calculate the distance between every two sinusoidal embeddings of our module and see the linear property of such distance. Hence, all the rows will have to be subtracted from the first one, from the second one and so on up until the last one.

```python

import torch

# We'll create embeddings with many dimensions to better see the
# frequency decay effect
max_pos = 1000
embed_dim = 1000


sinusoidal_embeddings = SinusoidalEmbeddings(max_pos, embed_dim).
    embeddings

# A single column tensor where each row's single element contains an
# entire sinusoidal embeddings tensor
T_2 = sinusoidal_embeddings[:, None, :]

# A single row tensor where column's single element contains an
# entire sinusoidal embeddings tensor
T_1 = sinusoidal_embeddings[None, :, :]

# By broadcasting, this operation will save the desired differences
# in the tensor's last dimension
differences = T_2 - T_1

# As the differences were saved in the last dimension, we need
# to calculate the norm with respect to it, which is indicated
# by dim=-1
distances = torch.norm(differences, p=2, dim=-1)

# Plot the resulting 2D tensor as a heatmap
plt.figure(figsize=(8, 6))
plt.imshow(distances.numpy(), aspect="auto", cmap="PuRd")
plt.colorbar(label="L2 Norm (Euclidean Distance)")
plt.xlabel("pos")
plt.ylabel("pos")
plt.title("Euclidean Distances Between Different Position Sinusoidal
    Embeddings")
plt.show()
```

Listing 5: Generating the plot of the module of the difference between every two different positions sinusoidal embeddings

Figure 3 above shows us how the distance between sinusoidal embeddings of different positions decreases smoothly and linearly for embedidding with $d = 1000$.

Question 1

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

(a) Do this.

(b) Do that.

(c) Do something else.

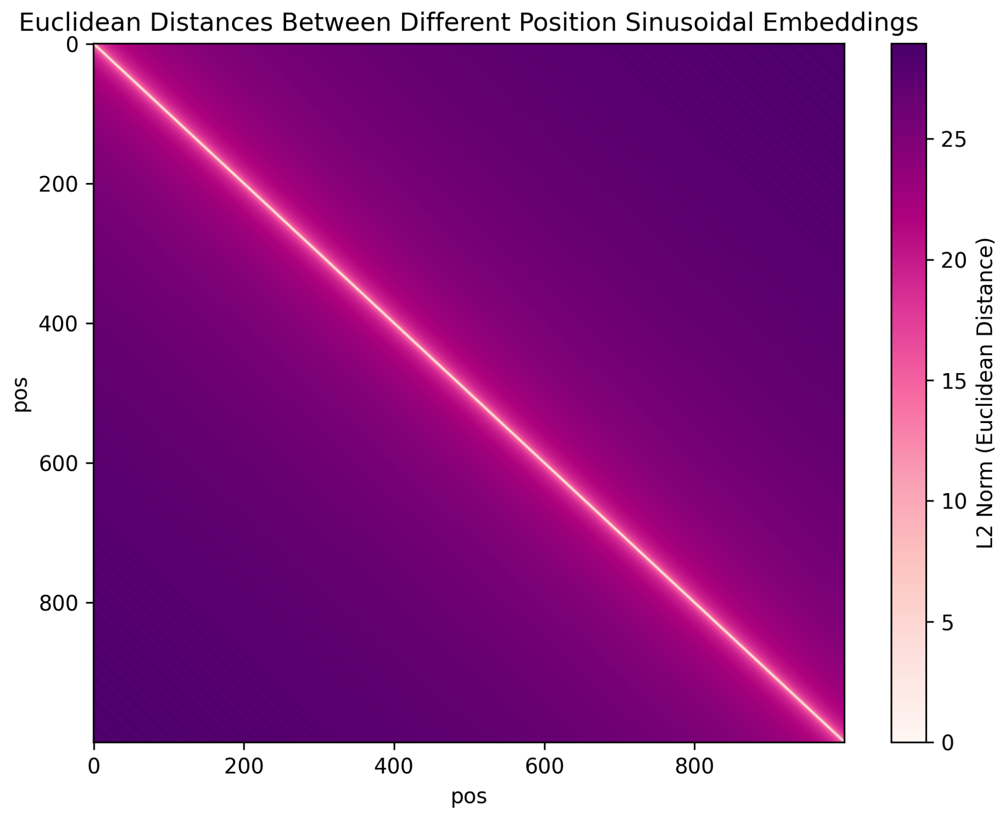

Figure 3: The sinusoidal embeddings difference vector's modules decrease linearly with the distance between the subtracted vectors

3.5 Algorithmic issues

In malesuada ullamcorper urna, sed dapibus diam sollicitudin non. Donec elit odio, accumsan ac nisl a, tempor imperdiet eros. Donec porta tortor eu risus consequat, a pharetra tortor tristique. Morbi sit amet laoreet erat. Morbi et luctus diam, quis porta ipsum. Quisque libero dolor, suscipit id facilisis eget, sodales volutpat dolor. Nullam vulputate interdum aliquam. Mauris id convallis erat, ut vehicula neque. Sed auctor nibh et elit fringilla, nec ultricies dui sollicitudin. Vestibulum vestibulum luctus metus venenatis facilisis. Suspendisse iaculis augue at vehicula ornare. Sed vel eros ut velit fermentum porttitor sed sed massa. Fusce venenatis, metus a rutrum sagittis, enim ex maximus velit, id semper nisi velit eu purus.

Algorithm 1: `FastTwoSum`

```
Input: $(a, b)$, two floating-point numbers
Result: $(c, d)$, such that $a + b = c + d$

if $|b| > |a|$ then
    exchange $a$ and $b$ ;
end
$c \leftarrow a + b$ ;
$z \leftarrow c - a$ ;
$d \leftarrow b - z$ ;
return $(c, d)$ ;
```

Fusce varius orci ac magna dapibus porttitor. In tempor leo a neque bibendum sollicitudin. Nulla pretium fermentum nisi, eget sodales magna facilisis eu. Praesent aliquet nulla ut bibendum lacinia.

Donec vel mauris vulputate, commodo ligula ut, egestas orci. Suspendisse commodo odio sed hendrerit lobortis. Donec finibus eros erat, vel ornare enim mattis et.

Question 2 *(with optional title)*

In congue risus leo, in gravida enim viverra id. Donec eros mauris, bibendum vel dui at, tempor commodo augue. In vel lobortis lacus. Nam ornare ullamcorper mauris vel molestie. Maecenas vehicula ornare turpis, vitae fringilla orci consectetur vel. Nam pulvinar justo nec neque egestas tristique. Donec ac dolor at libero congue varius sed vitae lectus. Donec et tristique nulla, sit amet scelerisque orci. Maecenas a vestibulum lectus, vitae gravida nulla. Proin eget volutpat orci. Morbi eu aliquet turpis. Vivamus molestie urna quis tempor tristique. Proin hendrerit sem nec tempor sollicitudin.

Mauris interdum porttitor fringilla. Proin tincidunt sodales leo at ornare. Donec tempus magna non mauris gravida luctus. Cras vitae arcu vitae mauris eleifend scelerisque. Nam sem sapien, vulputate nec felis eu, blandit convallis risus. Pellentesque sollicitudin venenatis tincidunt. In et ipsum libero. Nullam tempor ligula a massa convallis pellentesque.

4 Implementation

Proin lobortis efficitur dictum. Pellentesque vitae pharetra eros, quis dignissim magna. Sed tellus leo, semper non vestibulum vel, tincidunt eu mi. Aenean pretium ut velit sed facilisis. Ut placerat urna facilisis dolor suscipit vehicula. Ut ut auctor nunc. Nulla non massa eros. Proin rhoncus arcu odio, eu lobortis metus sollicitudin eu. Duis maximus ex dui, id bibendum diam dignissim id. Aliquam quis lorem lorem. Phasellus sagittis aliquet dolor, vulputate cursus dolor convallis vel. Suspendisse eu tellus feugiat, bibendum lectus quis, fermentum nunc. Nunc euismod condimentum magna nec bibendum. Curabitur elementum nibh eu sem cursus, eu aliquam leo rutrum. Sed bibendum augue sit amet pharetra ullamcorper. Aenean congue sit amet tortor vitae feugiat.

In congue risus leo, in gravida enim viverra id. Donec eros mauris, bibendum vel dui at, tempor commodo augue. In vel lobortis lacus. Nam ornare ullamcorper mauris vel molestie. Maecenas vehicula ornare turpis, vitae fringilla orci consectetur vel. Nam pulvinar justo nec neque egestas tristique. Donec ac dolor at libero congue varius sed vitae lectus. Donec et tristique nulla, sit amet scelerisque orci. Maecenas a vestibulum lectus, vitae gravida nulla. Proin eget volutpat orci. Morbi eu aliquet turpis. Vivamus molestie urna quis tempor tristique. Proin hendrerit sem nec tempor sollicitudin.

hello.py

```python
#! /usr/bin/python

import sys
sys.stdout.write("Hello World!\n")
```

Fusce eleifend porttitor arcu, id accumsan elit pharetra eget. Mauris luctus velit sit amet est sodales rhoncus. Donec cursus suscipit justo, sed tristique ipsum fermentum nec. Ut tortor ex, ullamcorper varius congue in, efficitur a tellus. Vivamus ut rutrum nisi. Phasellus sit amet enim efficitur, aliquam nulla id, lacinia mauris. Quisque viverra libero ac magna maximus efficitur. Interdum et malesuada fames ac ante ipsum primis in faucibus. Vestibulum mollis eros in tellus fermentum, vitae tristique justo finibus. Sed quis vehicula nibh. Etiam nulla justo, pellentesque id sapien at, semper aliquam arcu. Integer at commodo arcu. Quisque dapibus ut lacus eget vulputate.

Command Line

```
$ chmod +x hello.py
$ ./hello.py

Hello World!
```

Vestibulum sodales orci a nisi interdum tristique. In dictum vehicula dui, eget bibendum purus elementum eu. Pellentesque lobortis mattis mauris, non feugiat dolor vulputate a. Cras porttitor dapibus lacus at pulvinar. Praesent eu nunc et libero porttitor malesuada tempus quis massa. Aenean cursus ipsum a velit ultricies sagittis. Sed non leo ullamcorper, suscipit massa ut, pulvinar erat. Aliquam erat volutpat. Nulla non lacus vitae mi placerat tincidunt et ac diam. Aliquam tincidunt augue sem, ut vestibulum est volutpat eget. Suspendisse potenti. Integer condimentum, risus nec maximus elementum, lacus purus porta arcu, at ultrices diam nisl eget urna. Curabitur sollicitudin diam quis sollicitudin varius. Ut porta erat ornare laoreet euismod. In tincidunt purus dui, nec egestas dui convallis non. In vestibulum ipsum in dictum scelerisque.

Notice: In congue risus leo, in gravida enim viverra id. Donec eros mauris, bibendum vel dui at, tempor commodo augue. In vel lobortis lacus. Nam ornare ullamcorper mauris vel molestie. Maecenas vehicula ornare turpis, vitae fringilla orci consectetur vel. Nam pulvinar justo nec neque egestas tristique. Donec ac dolor at libero congue varius sed vitae lectus. Donec et tristique nulla, sit amet scelerisque orci. Maecenas a vestibulum lectus, vitae gravida nulla. Proin eget volutpat orci. Morbi eu aliquet turpis. Vivamus molestie urna quis tempor tristique. Proin hendrerit sem nec tempor sollicitudin.