MICCAI Educational Challenge 2024 Submission

Introduction to Diffusion Models

Content of the submission:

1. Lecture slides
2. Presentation notes
3. Practical tutorial as a Jupyter notebook

Diffusion models: from noise to images

Sofia Leon
Ricardo Montoya
Kaouther Mouheb

Content

01

Theory

All you need to know to train your first model!

02

Notebook Activity

Create and train a model yourself!

03

Game activity

Can you tell the difference between synthetic and real?

01

Theory

The Power of GANs

Park, Taesung, et al. "Semantic image synthesis with spatially-adaptive normalization." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2019.

But in 2021...

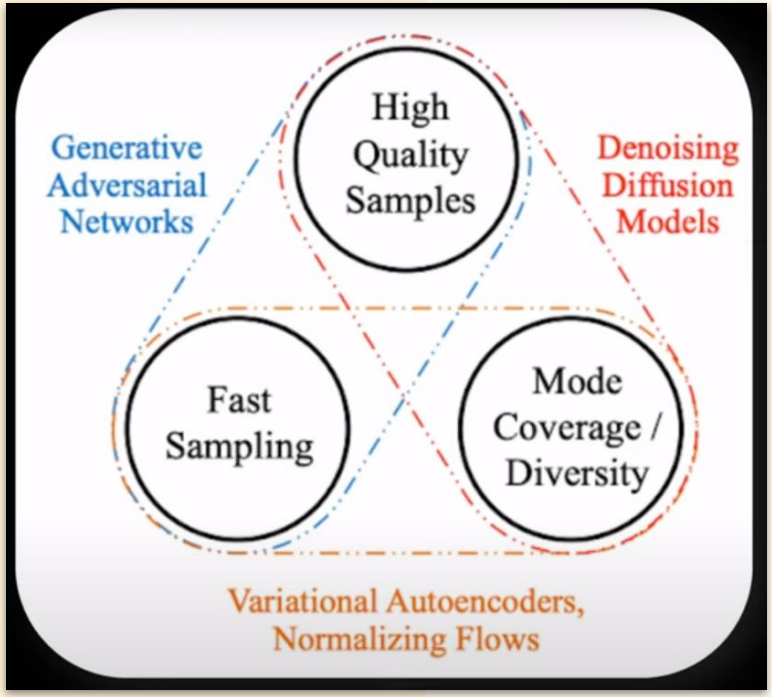
Diffusion Models Beat GANs on Image Synthesis

Prafulla Dhariwal*
OpenAI
prafulla@openai.com

Alex Nichol*
OpenAI
alex@openai.com

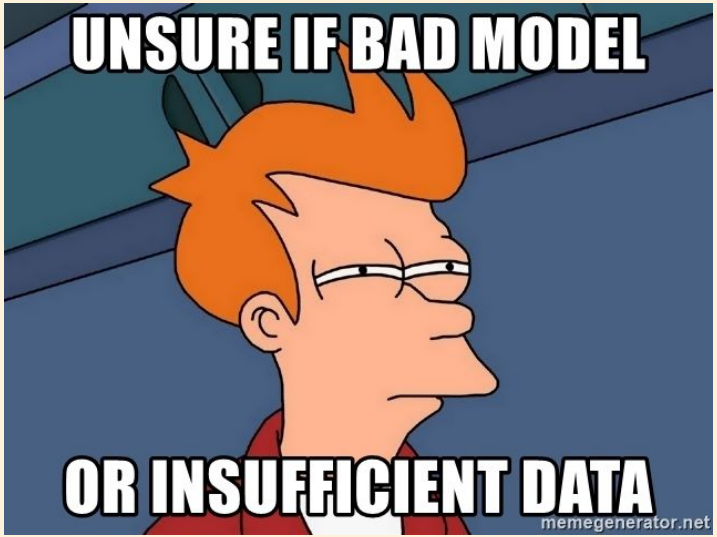
Read the paper!

The Generative Deep Learning Triangle



Zhisheng Xiao, et al. "Tackling the Generative Learning Trilemma with Denoising Diffusion GANs" ICLR 2022..

Relevance in medical imaging



Problem: *Deep learning needs data … a lot.*

- ➔ *Medical images are <u>scarce.</u>*
- ➔ *Some clinical cases are <u>rare.</u>*

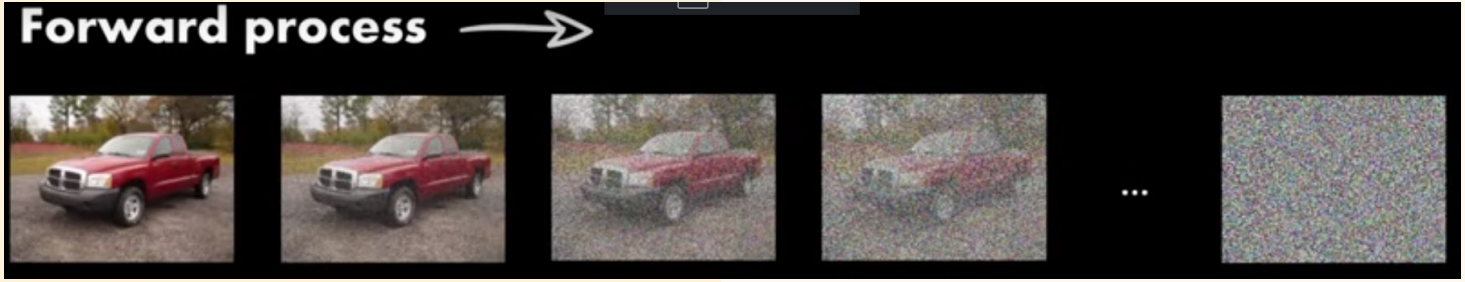
Other problems

- ➔ *Medical images are <u>expensive</u> to acquire and label.*
- ➔ *<u>Privacy</u> concerns.*


Let's get
started!

Core idea: training

*Destroy the input by **gradually** adding noise*

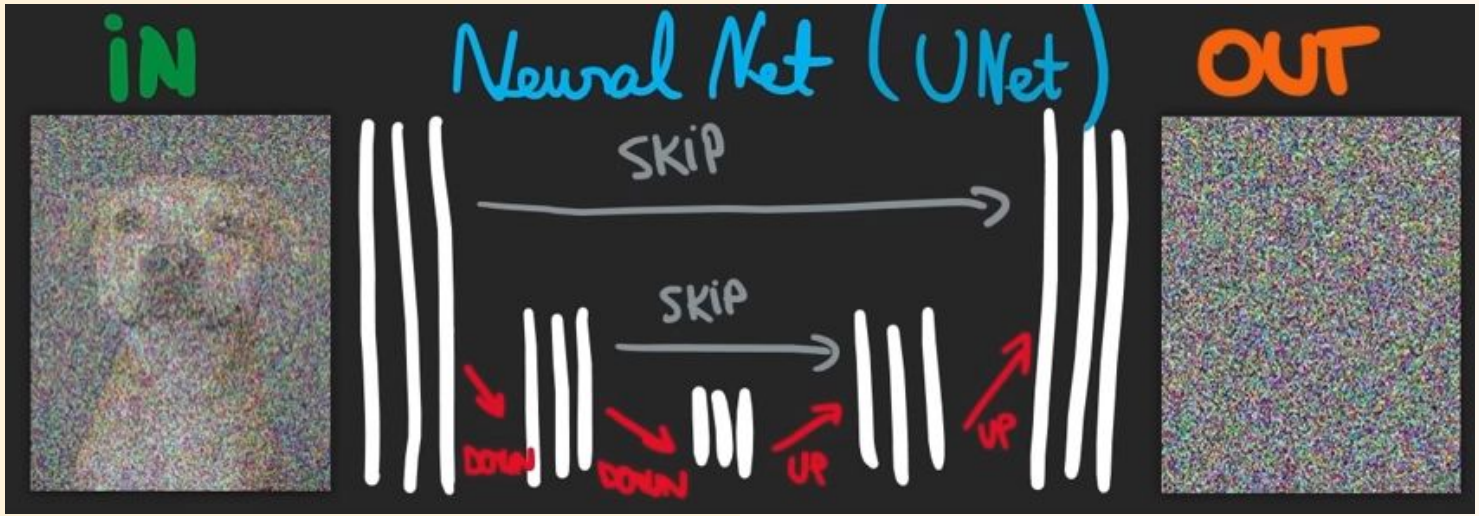


Then recover the input from noise in a backward process (Denoising)

Florian DeepFindr. "Diffusion models from scratch in PyTorch" YouTube, uploaded by DeepFindr, 18 July 2022

Core idea

*UNet capable of **<u>detecting noise.</u>***

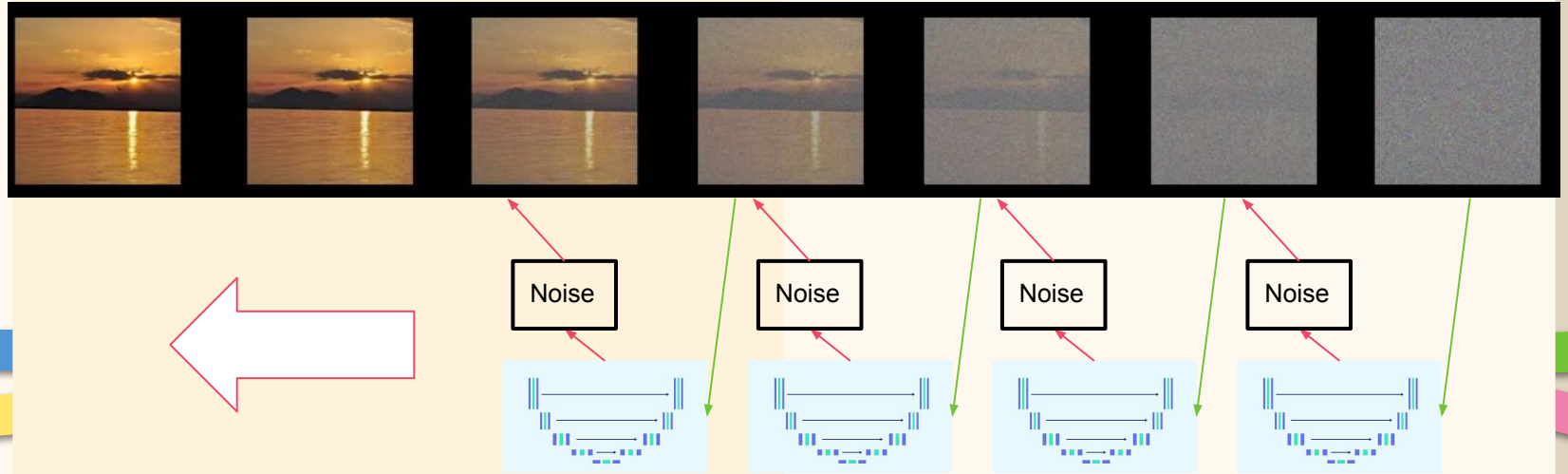


*We will use the model output to iteratively **<u>remove</u>** noise.*

Letitia Parcalabescu. "Diffusion models explained. How does OpenAI's GLIDE work?" YouTube, uploaded by AI Coffee Break with Letitia, 23 March 2022, https://www.youtube.com/watch?v=344w5h24-h8&list=PLpZBeKTZRGPPvAyM9DM-a6W0lugCo8WfC

Core idea: denoising

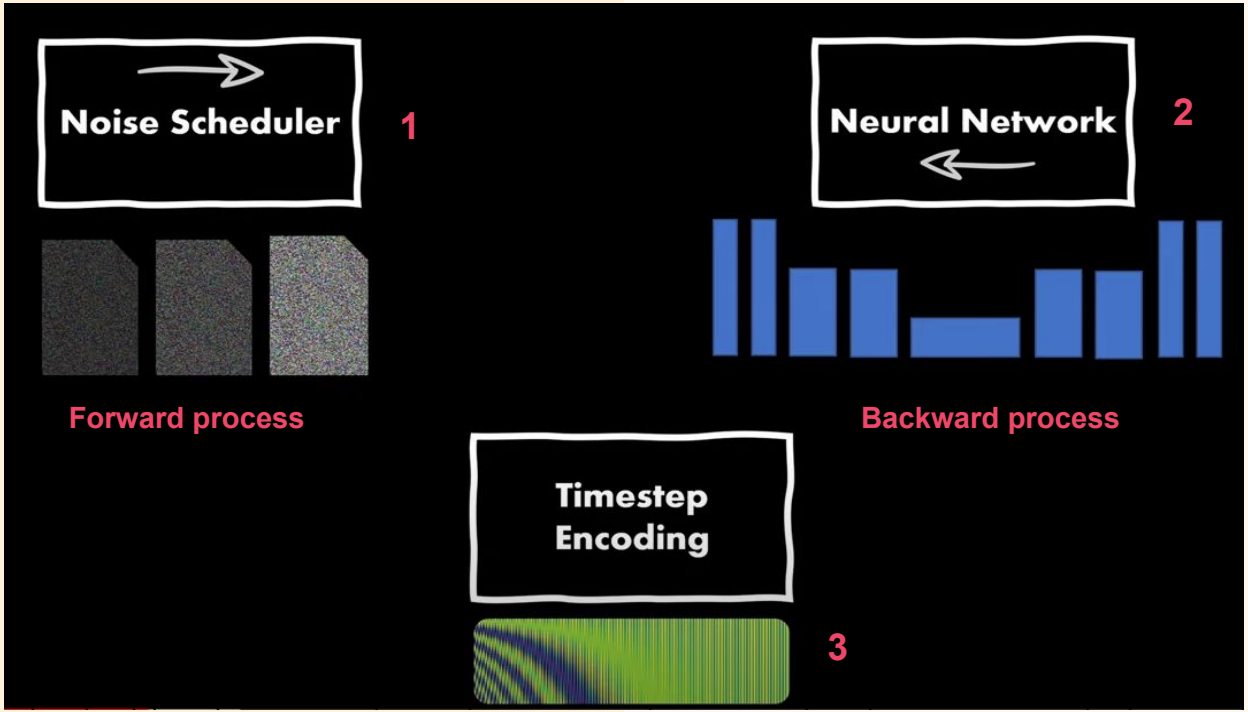
*We will use the model output to iteratively **remove** noise.*



Outlier. "Diffusion Models | Paper Explanation | Math Explained" YouTube, uploaded by Outlier, 6 June 2022, https://www.youtube.com/watch?v=HoKDTa5jHvg


Unet
What is my purpose?
You look at noisy 28x28 images of MNIST and try to guess which pixels are the noise
Oh, my god.

Main components



Florian DeepFindr. "Diffusion models from scratch in PyTorch" YouTube, uploaded by DeepFindr, 18 July 2022

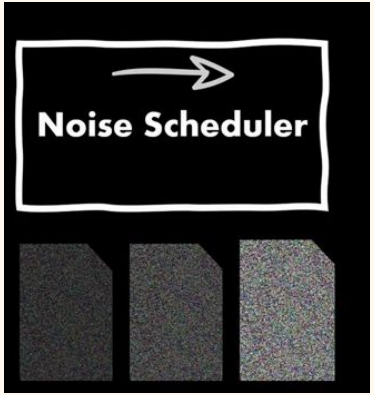
Forward process

Question 1: *How many timesteps?*

- *Depends. Usually 1000, but it could be less.*

Question 2: *How much noise to add?*

- *Noise scheduling (fixed)*



Florian DeepFindr. "Diffusion models from scratch in PyTorch" YouTube, uploaded by DeepFindr, 18 July 2022

Math time!

Noise Scheduler

$$q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1})$$

This is a Markov chain.

Noisy image at time T

Original image

How to add noise?

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

output

Input

Input image + noise

Florian DeepFindr. "Diffusion models from scratch in PyTorch" YouTube, uploaded by DeepFindr, 18 July 2022
Jonathan Ho et al. "Denoising Diffusion Probabilistic Models", NeurIPS 2020.

Noise Scheduler

Adding noise means sampling from a normal distribution

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

output mean variance (fixed)

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

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

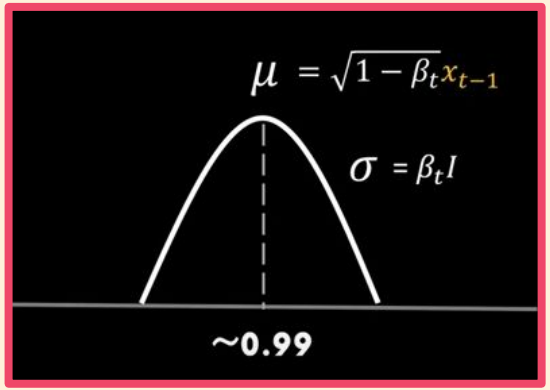
$$\epsilon \sim \mathcal{N}(0, I)$$

Florian DeepFindr. "Diffusion models from scratch in PyTorch" YouTube, uploaded by DeepFindr, 18 July 2022
Jonathan Ho et al. "Denoising Diffusion Probabilistic Models", NeurIPS 2020.

Noise Scheduler

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

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



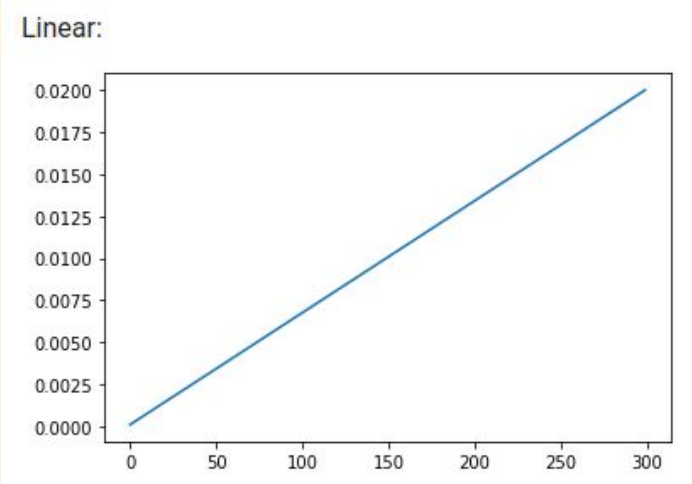
β *determines how much noise is added per timestep.*

Florian DeepFindr. "Diffusion models from scratch in PyTorch" YouTube, uploaded by DeepFindr, 18 July 2022
Jonathan Ho et al. "Denoising Diffusion Probabilistic Models", NeurIPS 2020.

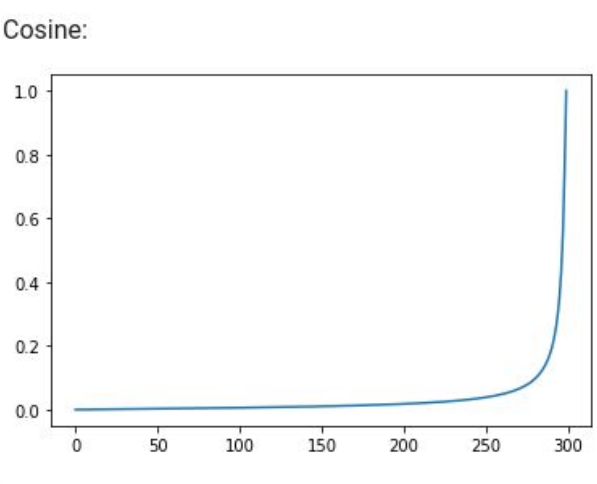
Noise Scheduler

β *determines how much noise is added per timestep.*

Linear:



Cosine:



Adding the right amount of noise is important!

Super trick

Image at time t Original image

$$q(x_t|x_0) = N(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I)$$

*Get **<u>any timestep</u>** without having to calculate the whole Markov chain.*

Using:

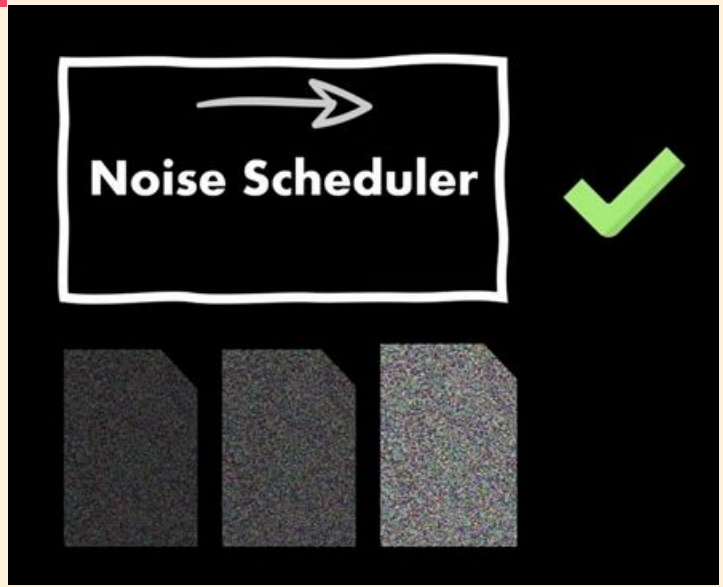
$$\alpha_t = 1 - \beta_t$$

$$\bar{\alpha}_t = \prod_{s=1}^{t} a_s$$

α tells us "how much of the original image is kept"

Jonathan Ho et al. "Denoising Diffusion Probabilistic Models", NeurIPS 2020.

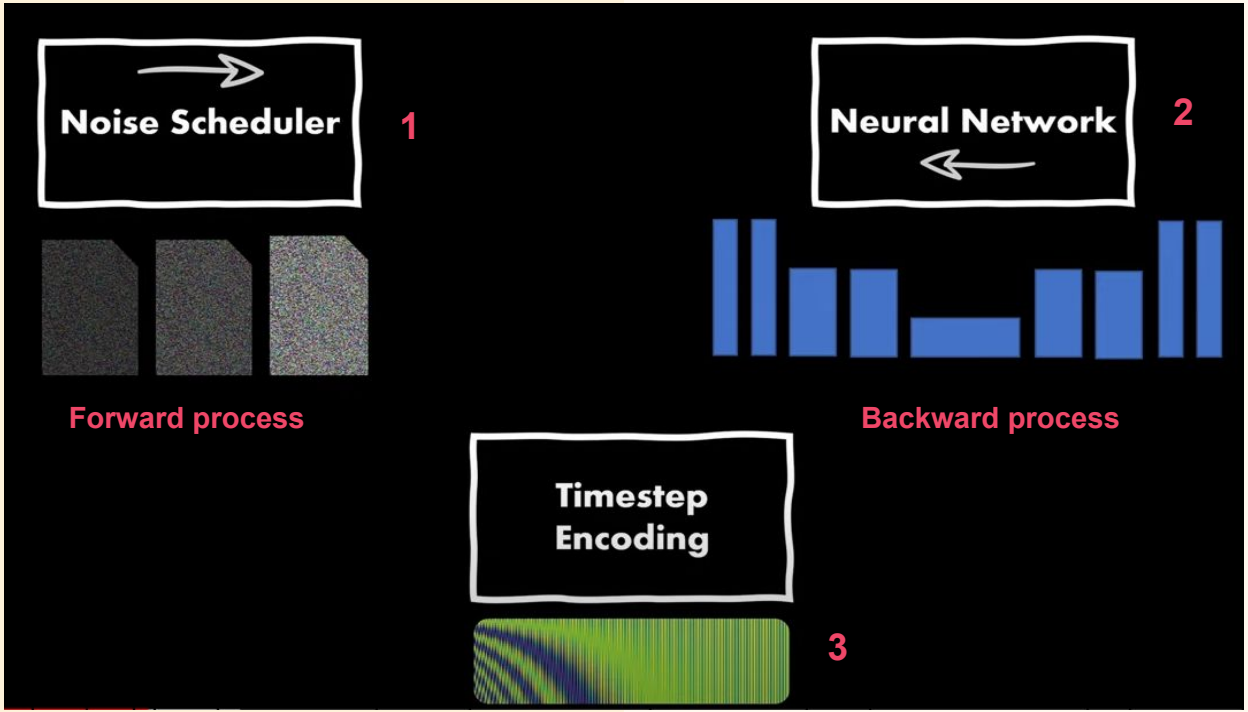
Summary



1. *We add noise in T timesteps.*
2. *We select and fix a noise scheduler (β)*
3. *We can compute noisy image at time t using the trick.*

Florian DeepFindr. "Diffusion models from scratch in PyTorch" YouTube, uploaded by DeepFindr, 18 July 2022

Main components



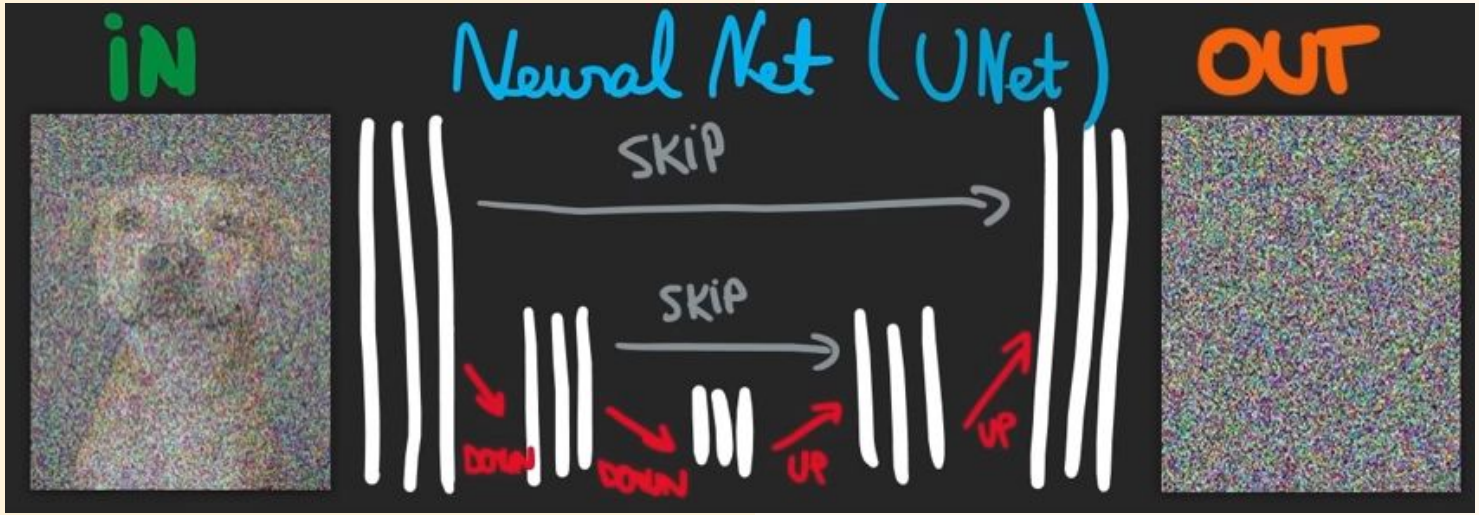
Florian DeepFindr. "Diffusion models from scratch in PyTorch" YouTube, uploaded by DeepFindr, 18 July 2022

Backward process (training)

*-> We use a **UNet**-like architecture.*

***Input**: Noisy image at time t*
(select a random time t)

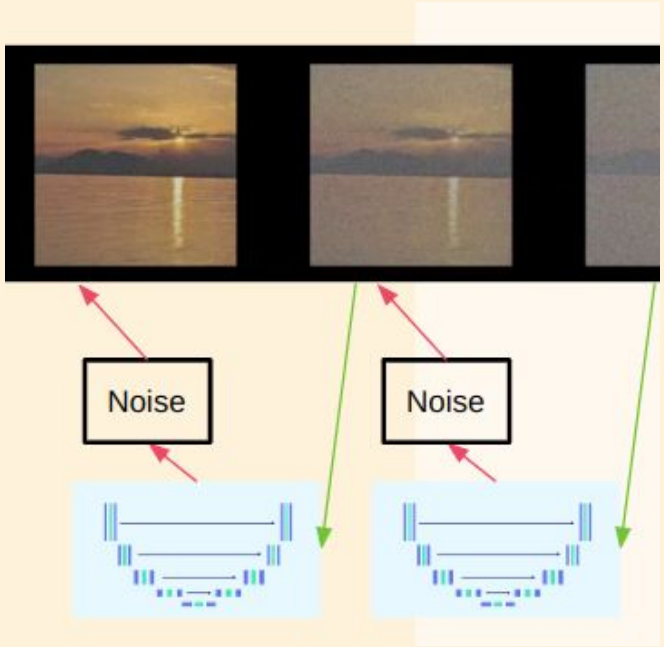
***Output**: Noise of the image.*



Letitia Parcalabescu. "Diffusion models explained. How does OpenAI's GLIDE work?" YouTube, uploaded by AI Coffee Break with Letitia, 23 March 2022, https://www.youtube.com/watch?v=344w5h24-h8&list=PLpZBeKTZRGPPvAyM9DM-a6W0lugCo8WfC

Backward process (Inference)

*The output (**noise**) can be used to **recover** the image at a previous timestep*



$$x_{t-1} \approx x_t - noise$$

Florian DeepFindr. "Diffusion models from scratch in PyTorch" YouTube, uploaded by DeepFindr, 18 July 2022

Loss function

Learning the gaussian **noise** (ϵ)

$$||\epsilon - \epsilon_\theta(x_t, t)||^2$$

Real noise

Our prediction

Jonathan Ho et al. "Denoising Diffusion Probabilistic Models", NeurIPS 2020.

Simple algorithm

Algorithm 1 Training

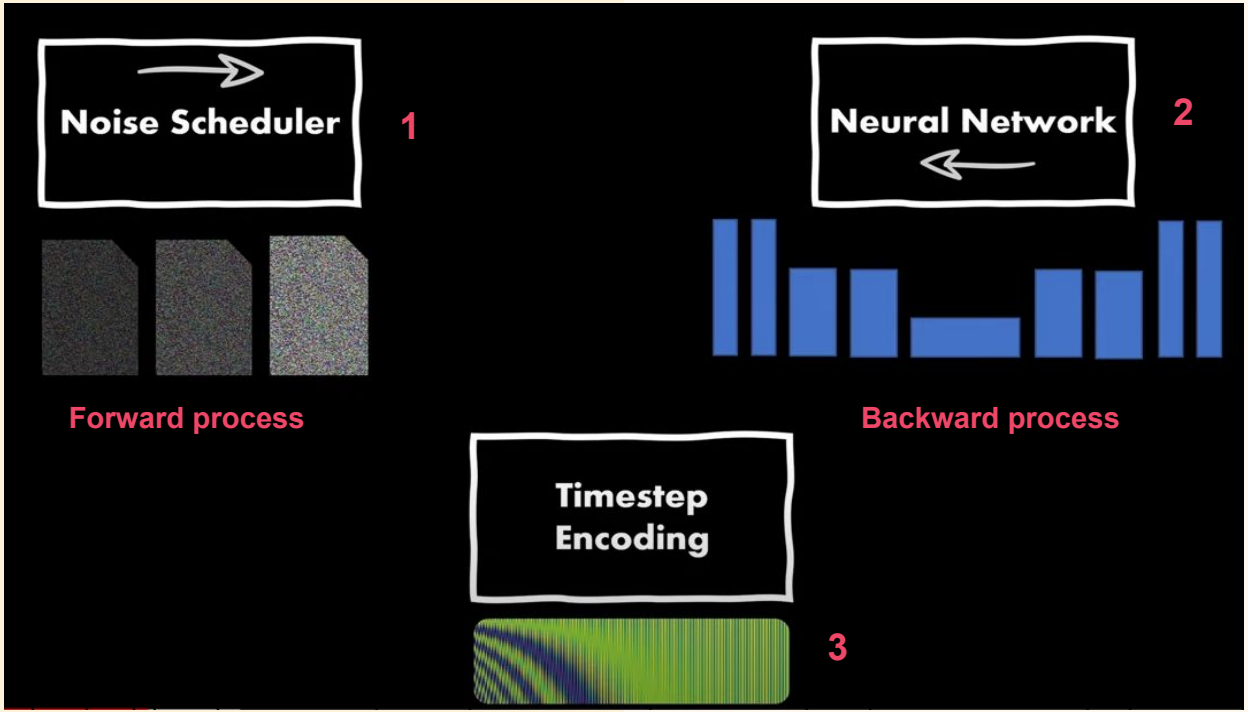
1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on

$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$

6: **until** converged

Jonathan Ho et al. "Denoising Diffusion Probabilistic Models", NeurIPS 2020.

Main components



Florian DeepFindr. "Diffusion models from scratch in PyTorch" YouTube, uploaded by DeepFindr, 18 July 2022

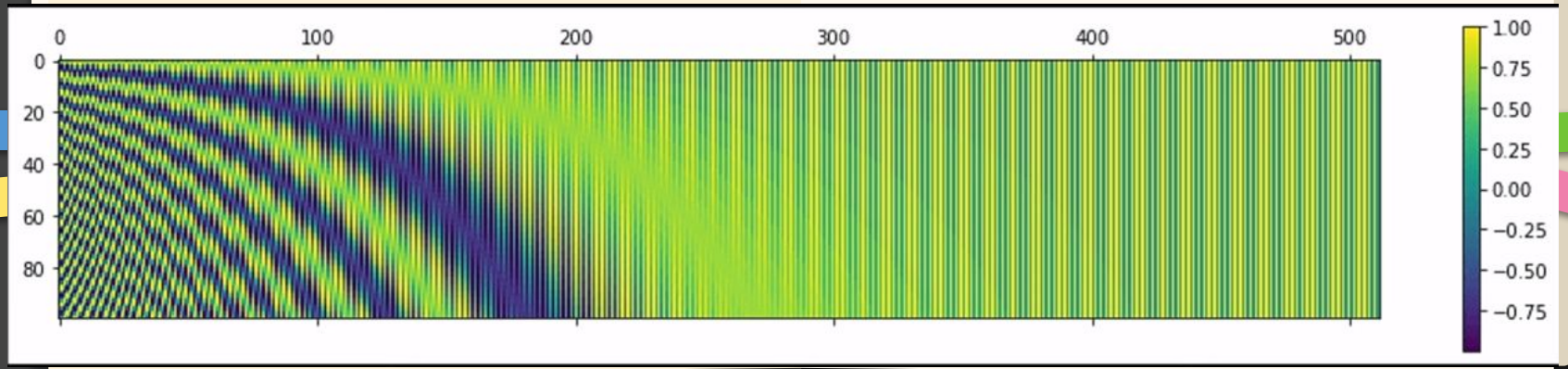
Time embedding

⟶ To let the network know at **what timestep** we are now.

Reason: We use the **same** network for **all timesteps.**

- Noise intensities are different for each t (noise scheduler).

Implementation: Added as extra input to the network.

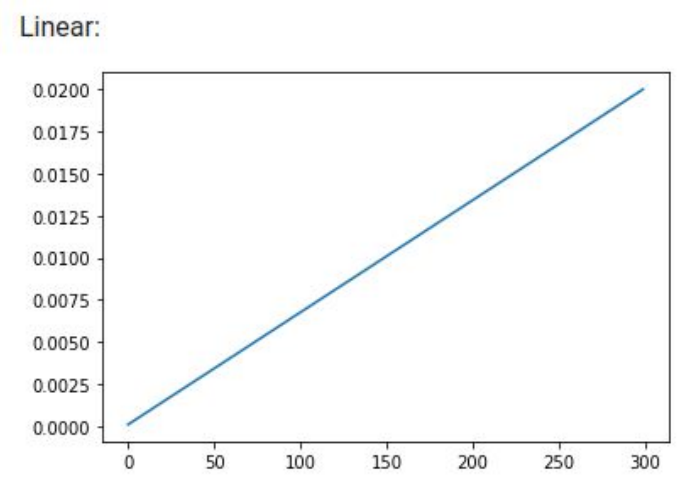


Time embedding

⟶ To let the network know at **what timestep** we are now.

Reason: We use the **same** network for **all timesteps.**

- Noise intensities are different for each t (noise scheduler).



02

Notebook

Jupyter Notebook

To see the theoretical knowledge you have just acquired in action, check the notebook we prepared for you!
LINK TO THE NOTEBOOK

Make sure to download the notebook and open it in CoLab

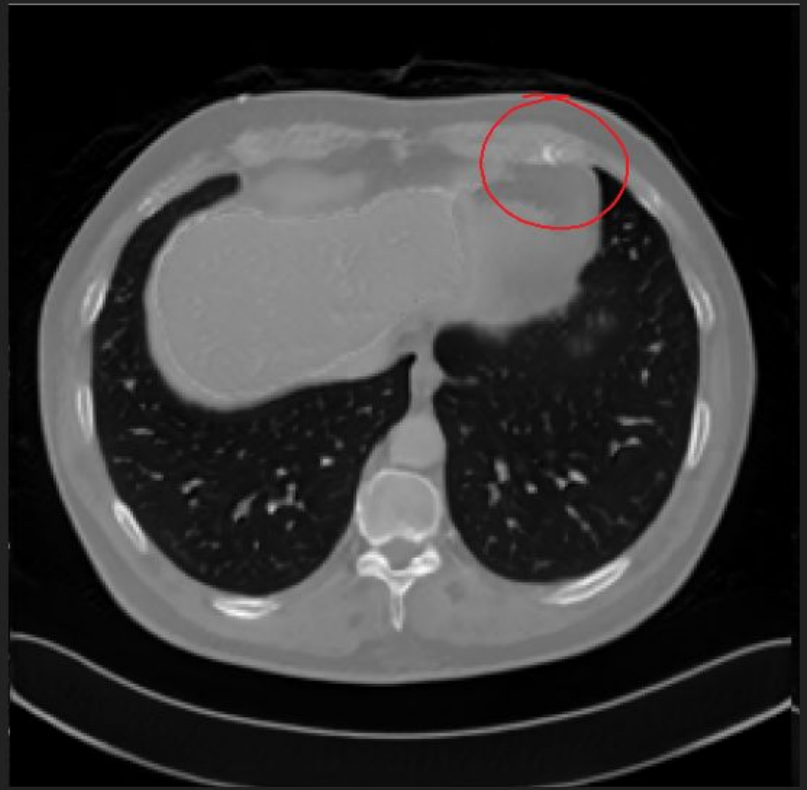
03

Game

CREDITOS: este modelo de apresentação foi criado pelo Slidesgo, inclui ícones da Flaticon, e infográficos e imagens da Freepik

Thanks!

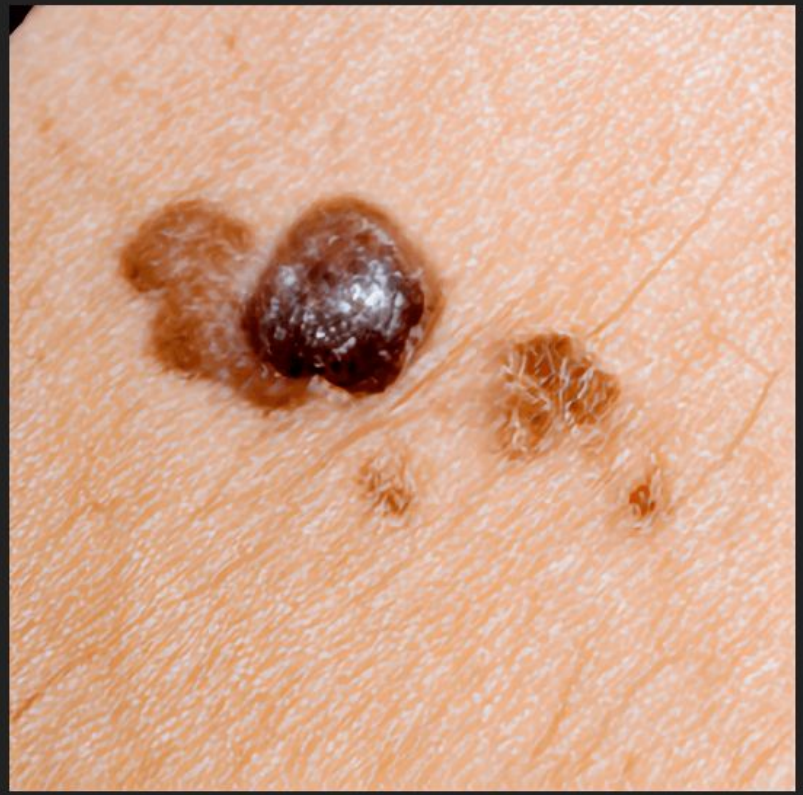
Do you have any questions?



This is a diffusion image!
Notice the bone in the upper right part (white part), the upper part should only have cartilage

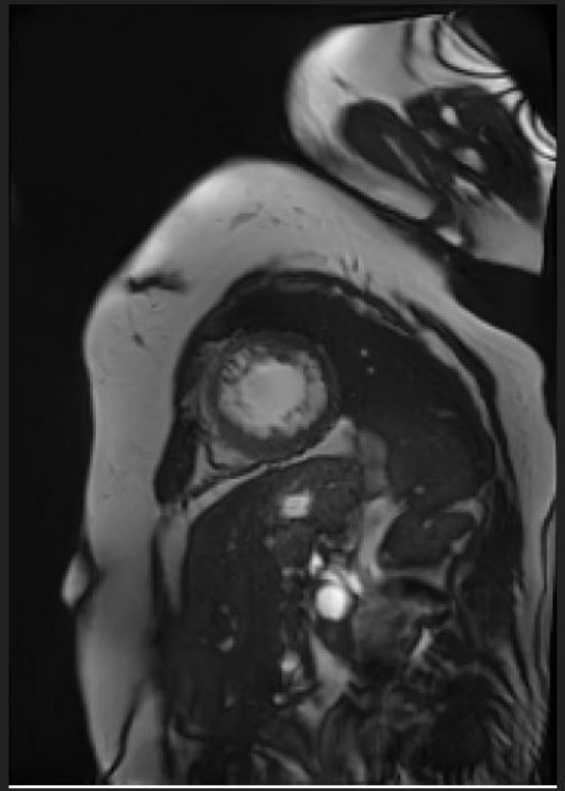
This is a diffusion image!
This is supposed to be a baby ultrasound image, overall it looks very good, it may even fool medical students. However, the details of the head are missing, it looks way too homogenous.

There is some blurriness in the vessel lines of the image. Additonally smaller, thinner vessels are not shown.



To be fair, not even Dr. Kai could tell it was not real. Sometimes diffusion is just that good! This could be helpful for other projects ;)

This a sort of an inclined saggital plane to be able to visualize the chambers of the heart.



This is a diffusion image, there is no recognizable anatomical structure


slidesgo

Presentation Notes (For lecturers)

Generative Deep Learning

A few words on GDL:
You have seen the power of GDL:

- First paint sold for $432,500 in NY
- Will Smith DeepFake
- Awesome designing tools

How do diffusion models work?

To build a diffusion model we need three components:

1. A noise scheduler to decide how much noise to add to the images and how. This is called the forward process.
2. A NN, used to remove the noise from the images. This is called the backward process.
3. A timestep encoding. Don't worry we will cover this topic at the end.

Forward process
We know that, given an image as input, we are adding noise in an iteration process until we destroy the image and we get Gaussian noise.
Question 1:
The first question you may ask yourself is: how many timesteps?
The answer to many things in life is: it depends.
Some authors use 1000 steps but this number can go down to only 300. In our notebook we will use 300, for example, and it is enough.
In theory the more timesteps you have the easier it is for the NN to follow up the changes in the images due to the noise.
Questions 2:
The second important question is: how much noise to add?
This is called noise scheduling and it is fixed since the beginning with using something called the noise scheduler.
Maths time
I know it looks complicated, so stay with me, we will survive this together. It is simple.
Noise scheduler
So this scary equation on the top is just saying that we add noise in an iterative process. This means that, given an original image, if you want to get the noisy version of this image at time T you need to go all the way from the original image at time zero (x0), passing x1, x2...till you reach xT.
This is called a Markov chain, because it is a process in which the next image in the iteration only depends on the image in the previous iteration.

So you may have realised that q is like a function to add noise: we give an input xt-1 and we get a noisier version xt. Now, a fundamental question, what does it mean to add noise?
Ask a student
Remember that this is happening at pixel level: each pixel will be adding a different random value.

That idea is perfect, is correct, but now I am going to give a different one that is equivalent:
"Adding noise means sampling from a normal distribution".
Let's see why this is true.
Remember that to define a normal distribution we need a mean and a variance.
So why is it equivalent to what the student said? Because of the parametrization trick.

This is the **parametrization trick**: given a normal distribution, in order to sample from this distribution you just need: its mean and add the sqrt of the variance times epsilon.
Epsilon is the key guys, you will see that in a minute. Epsilon is what adds randomness to the sampling because it comes from a standard normal distribution.

So epsilon is important, yes, but if you realise we are scaling epsilon by sqrt of beta.
Beta is at the end what determines how much noise is added to the image on each timestep.
If beta is small, mu is close to the original pixel value and the bell is going to be narrow.
But if beta is large (meaning close to 1) the mu is going to zero and the bell gets wider, meaning we are adding more noise to the image.

So you have to define beta wisely. There are many ways to do this. We present to you the two basic ones: adding the noise linearly or in a cosine manner. This guys is the noise scheduler and you will select it and fix it at the beginning. Pick the one you prefer.
I don't want to spoil you but one of these schedulers destroys the image faster.

Super trick
Now, finally, you remember I told you that if you want the image at time T you need to pass through all the time steps since the beginning?
Well, there is a super trick that allows you to get the noisy image for any time step directly from the original image without having to do the whole Markov chain. This is possible because, at the end, a sum of Gaussians is still a Gaussian.
This expression distribution is exactly the same as I presented to you before, the only difference is that now you are using alpha bar. But alpha bar is just the product of commutative alphas, and alpha is defined using beta. This just means that the alpha bar is defined from the beginning and is always available.
So you can get any noisy image super fast.

Summary
So, as summary,
- We will add noise in T timesteps and you can define T however you want: please use 300 today.
- Beta controls how much noise we add and is fixed
- We can use the trick to get noisy images at any time step t.

Backward process
Now let's move to the interesting part: the Neural Network, the backward process.
You know the Unet so we won't stop explaining it.

Remember what we want the Unet to do:
- We want to input a noisy image and we wait for it to output the noise in that image.

Training
Now remember this:
If we have a big T equal to 300, if we have 300 timesteps, we have 300 noisy images: each of them with different noise intensity but at the end 300 noisy images for just one original image.
So which one are we giving to the network? All of them, some of them.
The answer is one, one is enough, that's what the authors of this model found.
If we give all the images, for instance, the model may memorise how to denoise this images and fail with others.

SO message: one image, one timestep.

Inference time
At inference time, how do I recover the image?
Even though this is not exactly what is happening, we can think that we take this noise and remove it from the image. Explaining what is actually happening would take more that 15 minutes but this is basically the same idea.

Loss function
You are going to tell me, ok Ricardo, but you haven't told me how my network is going to learn: where is the loss function? We needed it for gradient descent.

Well, you remember I told you epsilon was the key?
After some crazy mathematical derivation the authors of this paper found that all you need to do is to compare the output of your model (the predicted noise) and the actual noise, doing MSE. That's it.

Diffusion Models Tutorial

August 15, 2024

1 Diffusion Models Tutorial

Welcome to our introductory notebook to Diffusion models. Throughout this notebook, we will try to guide you to build and train your (maybe) first diffusion model! **How exciting** !!!

After going through your very helpful and kind feedbacks, and after applying majority voting on your answers, we decided we will use PyTorch as a framework and X-Ray images as data for this activity. The data that we will use comes from RSNA Bone Age dataset from kaggle (https://www.kaggle.com/datasets/kmader/rsna-bone-age).

Please download the dataset and upload it to your Google Drive!

Below is an example image of the dataset:

1.1 1- Packages

Here, we will import the necessary packages and libraries to run this notebook. Feel free to add any libraries that you need.

[ ]:
```
### General
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import math
from pathlib import Path

### Pytorch
import torch
import torchvision
import torch.nn.functional as F
from torch import nn
from torchvision import transforms
from torch.utils.data import DataLoader, Dataset
from torch.optim import Adam

## Pillow
from PIL import Image
```

```
### Other
from traitlets.traitlets import validate
```

1.2 2- Parameters

Remember! It is a good practice to initialize parameters in one place so that it is easy to find them and change them.

Feel free to come back here and put/update your parameters

[ ]:
```
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

IMG_SIZE = 64 # Image size
BATCH_SIZE = 32 # Batch size

T = 300 # Number of time stamps in the forward process

LR = 0.01
```

1.3 3- Prepare data

In this section, we will prepare the data needed for training our diffusion model. The following parts assume you downloaded the dataset to your Google Drive under the name **bone_age_dataset** . If you used another name please modify your code accordingly.

Next, mount Google drive to colab by running this code:

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

If done correctly, you should have the followings in your drive inside the bona_age_dataset folder:

```
- boneage-training-dataset: folder containing around 12k images
- boneage-test-dataset: folder containing 200 images
- boneage-training-dataset.csv: csv file that contains image ids of the training set (and othe
- boneage-test-dataset: csv file that contains image ids of the test set
```

Check if everything went correctly! if not call us we got your code !

Now to the real PyTorching!! It is time to create our Dataset class. If you don't know what that is, here is the documentation link: https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset

It is basically a template for our dataset. We need to implement the following 3 methods: - ___init___ : the constructor , it defines the actions to take when we first create a Dataset instance (object) - ___len___ : Simply returns the total number of images in the dataset - ___getitem___ : It takes an index as argument and returns the image inside our dataset at that index

Let's do it!

[ ]:
```
class RSNA(Dataset):
    """
    Dataset class for RSNA bone age X-Ray dataset
    Attributes:
      img_ids (pd dataframe): dataframe containing image IDs
      img_dir (Path): path to the images folder
      transform (optional): image preprocessing transform
    """
    def __init__(self, annotations_file: Path, img_dir: Path, transform=None):
        # the following lines will be executed when a RSNA object is created
        self.img_labels = pd.read_csv(annotations_file) # read the image ids␣
 ↪and data as a pd dataframe
        self.img_dir = img_dir # set the image directory to the given img_dir␣
 ↪path
        self.transform = transform # if given, set the transform for␣
 ↪preprocessing the images

    def __len__(self):
        return len(self.img_labels) # return the number of rows in our␣
 ↪dataframe => each row represents an image

    def __getitem__(self, idx):
        # the following code will be executed when an image is requested from␣
 ↪the dataset
        img_name = f'{self.img_labels.iloc[idx, 0]}.png' # get the image ID␣
 ↪from the dataframe and add the image extension
        img_path = str(self.img_dir / img_name) # create the image path by␣
 ↪combining the dataset folder and image name
        image = Image.open(img_path) # read the image using PIL
        if self.transform:
            image = self.transform(image) # if provided, apply preprocessing␣
 ↪transform

        return image
```

Now that we have our class ready, we can create a RSNA object. Please make sure to provide the correct paths.

[ ]:
```
base_path = Path('./').resolve() # points to the home working directory (/
 ↪content in Colab)
data_path = base_path / 'drive' / 'MyDrive' / 'bone_age_dataset' # path to the␣
 ↪dataset folder, make sure to change it according to your folder names
annotation_file = data_path / 'boneage-training-dataset.csv' # this should␣
 ↪point to the train CSV
image_dir = data_path / 'boneage-training-dataset' / 'boneage-training-dataset'␣
 ↪# this should point to the folder where the images are stored
```

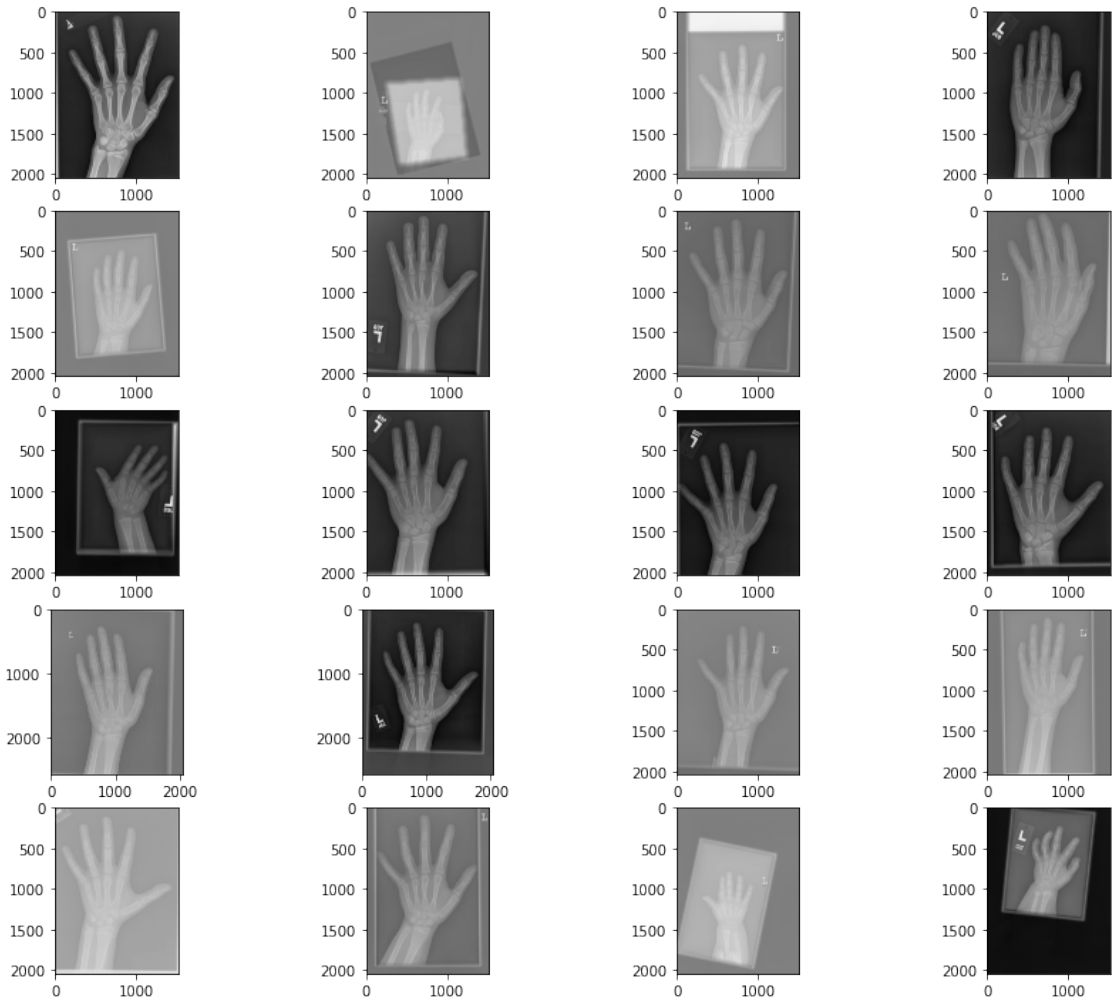
[ ]:
```
rsna_train = RSNA(annotation_file, image_dir) # Instanciate the Dataset object
print(f'Total #images: {len(rsna_train)}')
```

```
Total #images: 12611
```

The code should print the total number of images in our dataset. Now let's look at some of them.

[ ]:
```
# Helper function to visualize some images from the dataset.
def show_images(dataset, num_samples=20, cols=4):
    """ Plots some samples from the dataset """
    plt.figure(figsize=(15,15))
    for i, img in enumerate(dataset):
        if i == num_samples:
            break
        plt.subplot(num_samples/cols + 1, cols, i + 1)
        plt.imshow(np.asarray(img), cmap='gray')
```

[ ]:
```
show_images(rsna_train)
```



Taking a look at the images, we can see that they all have the same size (Perfect!!) BUT, the resolution is huge (Baaad!!) so we will have to preprocess these images to a lower resolution (smaller size) to make it easier for our model (and for us!)… Also note that the images still need to be normalized to [-1, 1] and converted to a PyTorch tensor, their type now is a PIL image. How do we do that ??

Answer =>**PREPROCESSING TRANSFORMS!**

[ ]:
```
# Create a preprocessing transform
data_transform = transforms.Compose( [transforms.Resize((IMG_SIZE, IMG_SIZE)),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.ToTensor(), # Scales data into␣
 ↪[0,1]
                                      transforms.Lambda(lambda t: (t * 2) - 1)]␣
 ↪# Scale between [-1, 1]
                                    )


rsna_train = RSNA(annotation_file, image_dir, data_transform) # Instanciate the␣
 ↪Dataset object
print(f'Total #images: {len(rsna_train)}')
```

```
Total #images: 12611
```

Note that the previous function that we had only shows PIL images… We need a new function that can plot images stored in Tensors. You can use the following:

[ ]:
```
def show_tensor_image(image):
    # You need to reverse your preprocessing here.. otherwise you will only see␣
 ↪black images !!
    reverse_transforms = transforms.Compose([
        transforms.Lambda(lambda t: (t + 1) / 2),
        transforms.Lambda(lambda t: t.permute(1, 2, 0)), # CHW to HWC
        transforms.Lambda(lambda t: t * 255.),
        transforms.Lambda(lambda t: t.numpy().astype(np.uint8)),
        transforms.ToPILImage(),
    ])

    # Take first image of batch
    if len(image.shape) == 4:
        image = image[0, :, :, :]
    plt.imshow(reverse_transforms(image), cmap='gray')
```

Now, let's create a Dataloader (read more about it here https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader) that will help us to iterate over our images in mini-batches.

```
[ ]: train_loader = DataLoader(rsna_train, batch_size=BATCH_SIZE, shuffle=True,␣
      ↪drop_last=True)
```

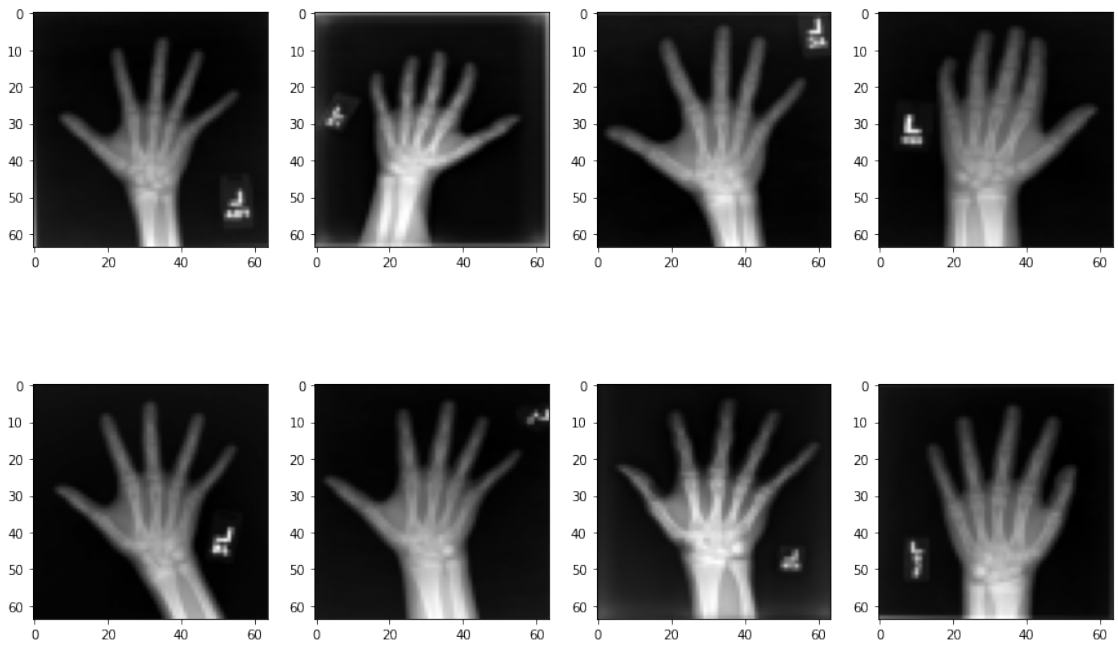
Easy peasy !! now let's show some images

```
[ ]: batch = next(iter(train_loader)) # retrieve the first batch of data
```

```
[ ]: batch.shape
```

```
[ ]: torch.Size([32, 1, 64, 64])
```

```
[ ]: num_samples=8
     cols = 4
     plt.figure(figsize=(15,15))
     for i in range(BATCH_SIZE):
         if i == num_samples:
             break
         plt.subplot(num_samples/cols + 1, cols, i + 1)
         show_tensor_image(batch[i, :, :, :])
```



Well! If you had made it this far correctly… CONGRATS ! you have your dataset and your dataloader and you can jump right into the ocean of Diffusion Models !

1.4 4- Forward process (Noise scheduler)

Remember from the theoretical part that the forward process consists of sequentially adding Gaussian noise to an image until we get a pure noise image.

Also remember the following key points:

- Noise scheduling is a FIXED process. So, the variances (betas) for each timestep t can be precomputed.
- There are different ways to do the noise scheduling
- The sum of Gaussians is a Gaussian => instead of calculating the noisy images sequentially, we can do it in closed form sampling.
- Since the process is fixed, no neural network is needed

First, let us provide some useful functions that we will need later

[ ]:
```
def get_value_from_list_by_idx(vals, t, x_shape):
    """
    Returns the value stored at a specific index t of a passed list of values␣
 ↪vals
    while considering the batch dimension.
    Args:
      vals : list of values from which the value is returned
      t : the position of the value to be returned in the list vals
      x_shape : the shape of the mini-batch
    """
    batch_size = t.shape[0]
    out = vals.gather(-1, t.cpu())
    val = out.reshape(batch_size, *((1,) * (len(x_shape) - 1))).to(t.device)
    return val
```

Let's create the variance scheduler functions.

These functions will be used to generate a list of size n of variances (betas) following some predefined trend.

Here we will implement two noise schedulers: linear and cosine

Linear:

[ ]:
```
def linear_beta_schedule(timesteps, start=0.0001, end=0.02):
    """
    linear scheduler that takes the number of time stamps (size of the tensor),␣
 ↪the initial and final betas to generate a list of betas
    """
    return torch.linspace(start, end, timesteps)
```

Cosine:

[ ]:
```
def cosine_beta_schedule(timesteps, s=0.008):
    """
    cosine schedule as proposed in https://arxiv.org/abs/2102.09672
    """
    steps = timesteps + 1
    x = torch.linspace(0, timesteps, steps)
    alphas_cumprod = torch.cos(((x / timesteps) + s) / (1 + s) * torch.pi * 0.
↪5) ** 2
    alphas_cumprod = alphas_cumprod / alphas_cumprod[0]
    betas = 1 - (alphas_cumprod[1:] / alphas_cumprod[:-1])
    return torch.clip(betas, 0.0001, 0.9999)
```

Now, we can precompute some values that we will use in the forward process. Remember the following useful formulae:

Let's compute these values..

[ ]:
```
betas = linear_beta_schedule(timesteps=T)
alpha_ts = 1.0 - betas
alpha_ts_cumprod = torch.cumprod(alpha_ts, axis=0)
```

Recall the closed form sampling formula:

We need to calculate 2 more entities, square roots of alpha_ts_cumprod and of (1 - alpha_ts_cumprod)

[ ]:
```
sqrt_alphas_cumprod = torch.sqrt(alpha_ts_cumprod)
sqrt_1_minus_alphas_cumprod = torch.sqrt(1.0 - alpha_ts_cumprod)
```

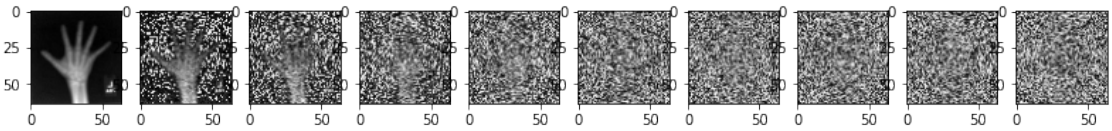
That's it! now we have everything to perform our forward process... let's create a function that takes an image and a time step as parameters then uses the values that we have just calculated to calculate the noisy version of the image at the specific time step t by applying the formula given above.

[ ]:
```
def forward_diffusion_sample(x_0, t, device="cpu"):
    """
    Takes an image and a timestep as input and
    returns the noisy version of it
    Args:
      x_0 : original image
```

```
      t   : timestep at which the noisy version is generated
      device : device type 'cpu' or 'cuda'
    """
    noise = torch.randn_like(x_0)
    sqrt_alphas_cumprod_t = get_value_from_list_by_idx(sqrt_alphas_cumprod, t,␣
↪x_0.shape)
    sqrt_one_minus_alphas_cumprod_t = get_value_from_list_by_idx(
        sqrt_1_minus_alphas_cumprod, t, x_0.shape
    )
    # mean + variance
    return sqrt_alphas_cumprod_t.to(device) * x_0.to(device) +␣
↪sqrt_one_minus_alphas_cumprod_t.to(device) * noise.to(device), noise.
↪to(device)
```

Now let's see if it works!

[ ]:
```
image = batch[0]  # get the fisrt image of the batch that we previously saw

plt.figure(figsize=(15,15))
plt.axis('off')
num_images = 10 # how many images we want to see
stepsize = int(T/num_images) # step between two consecutive images we want to␣
↪see

for idx in range(0, T, stepsize):
    t = torch.Tensor([idx]).type(torch.int64) # convert the index to a tensor
    plt.subplot(1, num_images+1, (idx/stepsize) + 1)
    image_n, noise = forward_diffusion_sample(image, t) # apply the forward␣
↪process
    show_tensor_image(image_n)
```



That's it! DONE!!!

1.5 5- Backward Process (U-Net)

Remember the following keypoints: - In the forward process we use a U-Net (or any network architecture) to predict the noise in the image - The input is a noisy image, the ouput the noise in the image - The result will be image - noise - Because the parameters are shared accross time, we need to tell the network in which timestep we are => Time embedding (similar to positional imbedding in ViT) - The Timestep is encoded by the transformer Sinusoidal Embedding - We output one single value (mean), because the variance is fixed

The positional encoding of the transformer looks like this …

And it is given by:

Don't worry if you don't understand… we don't understand either

We will create a module that performs this step inside the network. Notice that this module has no learnable parameters… it will just be used as an additional channel to the input and features maps to tell the network about the time step we are currently at.

[ ]:
```
class SinusoidalPositionEmbeddings(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, time):
        device = time.device
        half_dim = self.dim // 2
        embeddings = math.log(10000) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(half_dim, device=device) *␣
 ↪-embeddings)
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
        return embeddings
```

Next, we create our simple U-Net model, similar to the famous architecture that you can see below (Refer to Dr. Jose Bernal's seminar for more information about U-Net)

[ ]:
```
class Block(nn.Module):
    def __init__(self, in_ch, out_ch, time_emb_dim, up=False):
        super().__init__()
        self.time_mlp =  nn.Linear(time_emb_dim, out_ch)
        if up:
            self.conv1 = nn.Conv2d(2*in_ch, out_ch, 3, padding=1)
            self.transform = nn.ConvTranspose2d(out_ch, out_ch, 4, 2, 1)
        else:
            self.conv1 = nn.Conv2d(in_ch, out_ch, 3, padding=1)
            self.transform = nn.Conv2d(out_ch, out_ch, 4, 2, 1)
        self.conv2 = nn.Conv2d(out_ch, out_ch, 3, padding=1)
```

```
        self.bnorm1 = nn.BatchNorm2d(out_ch)
        self.bnorm2 = nn.BatchNorm2d(out_ch)
        self.relu  = nn.ReLU()

    def forward(self, x, t, ):
        # First Conv
        h = self.bnorm1(self.relu(self.conv1(x)))
        # Time embedding
        time_emb = self.relu(self.time_mlp(t))
        # Extend last 2 dimensions
        time_emb = time_emb[(..., ) + (None, ) * 2]
        # Add time channel
        h = h + time_emb
        # Second Conv
        h = self.bnorm2(self.relu(self.conv2(h)))
        # Down or Upsample
        h = self.transform(h)
        return h

class SimpleUnet(nn.Module):
    """
    A simplified variant of the Unet architecture.
    """
    def __init__(self):
        super().__init__()
        image_channels = 1
        down_channels = (64, 128, 256, 512, 1024)
        up_channels = (1024, 512, 256, 128, 64)
        out_dim = 1
        time_emb_dim = 32

        # Time embedding
        self.time_mlp = nn.Sequential(
                SinusoidalPositionEmbeddings(time_emb_dim),
                nn.Linear(time_emb_dim, time_emb_dim),
                nn.ReLU()
            )

        # Initial projection
        self.conv0 = nn.Conv2d(image_channels, down_channels[0], 3, padding=1)

        # Downsample
        self.downs = nn.ModuleList([Block(down_channels[i], down_channels[i+1],␣
 ↪\
                                    time_emb_dim) \
                    for i in range(len(down_channels)-1)])
        # Upsample
```

```
        self.ups = nn.ModuleList([Block(up_channels[i], up_channels[i+1], \
                                        time_emb_dim, up=True) \
                    for i in range(len(up_channels)-1)])

        self.output = nn.Conv2d(up_channels[-1], image_channels, out_dim)

    def forward(self, x, timestep):
        # Embedd time
        t = self.time_mlp(timestep)
        # Initial conv
        x = self.conv0(x)
        # Unet
        residual_inputs = []
        for down in self.downs:
            x = down(x, t)
            residual_inputs.append(x)
        for up in self.ups:
            residual_x = residual_inputs.pop()
            # Add residual x as additional channels
            x = torch.cat((x, residual_x), dim=1)
            x = up(x, t)
        return self.output(x)
```

[ ]:
```
# Create the model
model = SimpleUnet().to(DEVICE)

print("Num params: ", sum(p.numel() for p in model.parameters()))
```

```
Num params:  62437601
```

Look, the number of parameters is **HUGE!!!** 60 M for just a simple U-Net… this is why training Diffusion models take a lot of time and resources, and this is why they work very well…

6- The loss

Now, with the easiest part, the loss

Unlike the scary loss functions used to train GANs.. in Diffusion, the loss is simply the error (L1 norm, L2 norm…etc) between the real noise and the noise predicted by the model… EASYY and LOVELY

[ ]:
```
def get_loss(model, x_0, t):
    x_noisy, noise = forward_diffusion_sample(x_0, t, DEVICE)
    noise_pred = model(x_noisy, t)
    return F.l1_loss(noise, noise_pred)
```

1.6 6- Training

Before we train our model, remember the following: - To train the model, we input a noisy image and the model should estimate the noise in that image. - To generate a new image, we input pure

noise to the model, the model estimates the noise, we subtract the noise from the image, we input the result to the model and estimate again, and so on... until the image becomes pure...

In this phase, we will need to calculate more values to be used in the inference/training process

[ ]:
```
alphas_cumprod_prev = F.pad(alpha_ts_cumprod[:-1], (1, 0), value=1.0) #␣
 ↪alphas_(t-1)
sqrt_recip_alphas = torch.sqrt(1.0 / alpha_ts) # sqrt(1/alpha_)
posterior_variance = betas * (1. - alphas_cumprod_prev) / (1. -␣
 ↪alpha_ts_cumprod)
```

First let's create some sampling functions

[ ]:
```
@torch.no_grad() # make sure to use this one because otherwise u will get␣
 ↪memory errors
def sample_timestep(x, t):
    """
    Calls the model to predict the noise in the image and returns
    the denoised image.
    Applies noise to this image, if we are not in the last step yet.
    """
    betas_t = get_value_from_list_by_idx(betas, t, x.shape)
    sqrt_one_minus_alphas_cumprod_t =␣
 ↪get_value_from_list_by_idx(sqrt_1_minus_alphas_cumprod, t, x.shape)
    sqrt_recip_alphas_t = get_value_from_list_by_idx(sqrt_recip_alphas, t, x.
 ↪shape)

    # Call model (current image - noise prediction)
    model_mean = sqrt_recip_alphas_t * ( x - betas_t * model(x, t) /␣
 ↪sqrt_one_minus_alphas_cumprod_t)
    posterior_variance_t = get_value_from_list_by_idx(posterior_variance, t, x.
 ↪shape)

    if t == 0:
        return model_mean
    else:
        noise = torch.randn_like(x)
        return model_mean + torch.sqrt(posterior_variance_t) * noise
```

[ ]:
```
@torch.no_grad()
def sample_plot_image():
    # Sample noise
    img_size = IMG_SIZE
    img = torch.randn((1, 1, img_size, img_size), device=DEVICE)
    plt.figure(figsize=(15,15))
    plt.axis('off')
    num_images = 10
    stepsize = int(T/num_images)
```

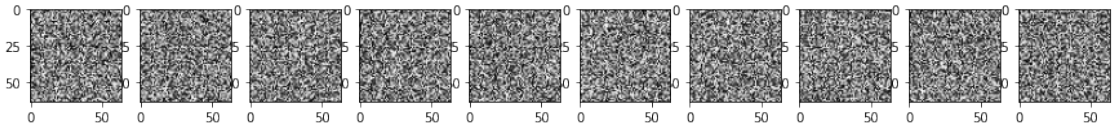
```
    for i in range(0,T)[::-1]:
        t = torch.full((1,), i, device=DEVICE, dtype=torch.long)
        img = sample_timestep(img, t)
        if i % stepsize == 0:
            plt.subplot(1, num_images, i/stepsize+1)
            show_tensor_image(img.detach().cpu())
    plt.show()
```

Now, the moment of Truth... Let's train!

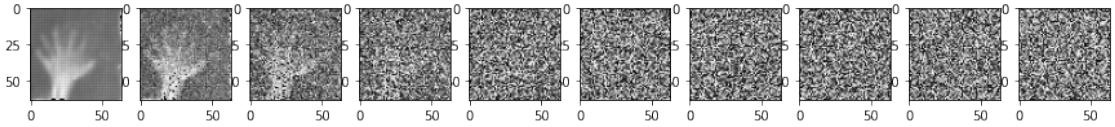
[ ]:
```
path = '/content/drive/MyDrive/bone_age_dataset/weights.pth'
```

[ ]:
```
optimizer = Adam(model.parameters(), lr=LR)
epochs = 50 # Try more!

for epoch in range(epochs):
    for step, batch in enumerate(train_loader):
      optimizer.zero_grad()

      t = torch.randint(0, T, (BATCH_SIZE,), device=DEVICE).long()
      loss = get_loss(model, batch, t)
      loss.backward()
      optimizer.step()

      if epoch % 5 == 0 and step == 0:
        print(f"Epoch {epoch} | step {step:03d} Loss: {loss.item()} ")
        sample_plot_image()

    torch.save(model.state_dict(), path)
```

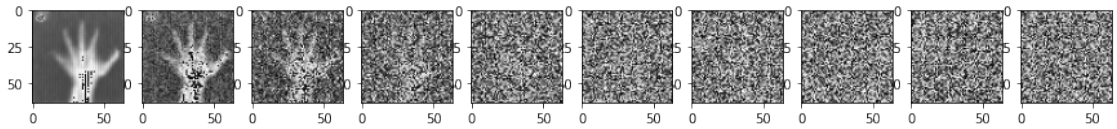
```
Epoch 0 | step 000 Loss: 0.8101167678833008
```



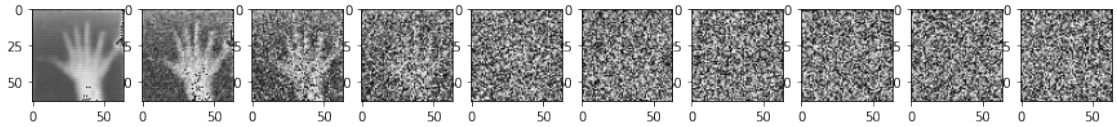
```
Epoch 5 | step 000 Loss: 0.11241922527551651
```



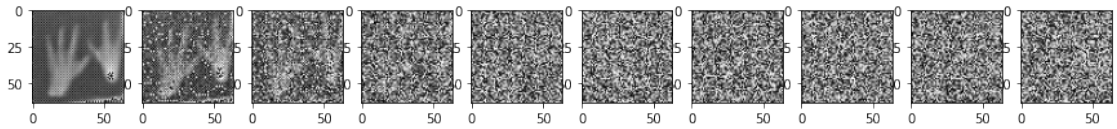
```
Epoch 10 | step 000 Loss: 0.08498869836330414
```



```
Epoch 15 | step 000 Loss: 0.07240307331085205
```



```
Epoch 20 | step 000 Loss: 0.09503066539764404
```



You can see that even at the 5th epoch, the model generations started to look like a hand! Well done model

but of course, the images we can get with this simple model and low resolution images are not plausible and not useable for training a real medical imaging network… but with the current resources, this is the best we can do

1.7 7- Inference

```
[ ]: path = '/content/drive/MyDrive/bone_age_dataset/weights.pth' # please change␣
      ↪this to the path of the weights according to your file system
     model = SimpleUnet().to(DEVICE)
     model.load_state_dict(torch.load(path))
     sample_plot_image()
```

Things you can do to improve these results: - Use a different beta scheduler (maybe make a network learn the betas too?) - Use a better model for the backward process - Add attention - Residual connections - Different activation functions like SiLU, GWLU, … - BatchNormalization - GroupNormalization and many many more…

Congratulations!!! you did it! you trained a diffusion model from scratch