0:00 - Introduction

6:38 - This course vs DALL-E

10:38 - How to take full advantage of this course

12:14 - Cloud computing options

14:58 - Getting started (Github, notebooks to play with, resources)

20:48 - Diffusion notebook from Hugging Face

Login hugging face
Pipeline -> Learner, we can save pipelines in diffusers. We can use repos from the hugging face or even save our own pipelines in the Hugging face Hub which is like the cloud.

26:59 - How stable diffusion works

Denoising iteratively

30:06 - Diffusion notebook (guidance scale, negative prompts, init image, textual inversion, Dreambooth)

Guidance scale -> for every single prompt -> creates two versions of images -> one version of image generated with prompt and second version without the prompt and then takes the average of the two things. The number guidance scale corresponds to the weight that it can use to take the average(weighted average).

Negative prompts -> We also input a negative prompt. We subtract the image generated by the negative prompt from the image generated by the normal prompt to get the final result. Thus moving more towards the original prompt and driving away from the negative prompt.
Ex: Non-blue Labrador

Init images ->

```
torch.manual_seed(1000)
prompt = "Wolf howling at the moon, photorealistic 4K"
images = pipei2i(prompt=prompt, num_images_per_prompt=3, init_image=init_image, strength=0.8, num_inference_steps=50).images
image_grid(images, rows=1, cols=3)
```

```
  0%|          | 0/41 [00:00<?, ?it/s]
```

Instead of starting stable diffusion from completely random noise, we can also start from a noisy version of the input image.

We can further use this as a seed to a later prompt.

Text inversion -> We can create a new token using a corresponding text embedding and fit the style of a certain number of input images to the given text embedding. The values in the text embedding are trained to capture the style of the input images. When generating an image from a prompt later, we can use the new token and create the image using this.

Example picture ->

Generated picture with just 4 such example images:

```
torch.manual_seed(1000)
image = pipe("Woman reading in the style of <watercolor-portrait>").images[0]
image
```

```
  0%|          | 0/51 [00:00<?, ?it/s]
```

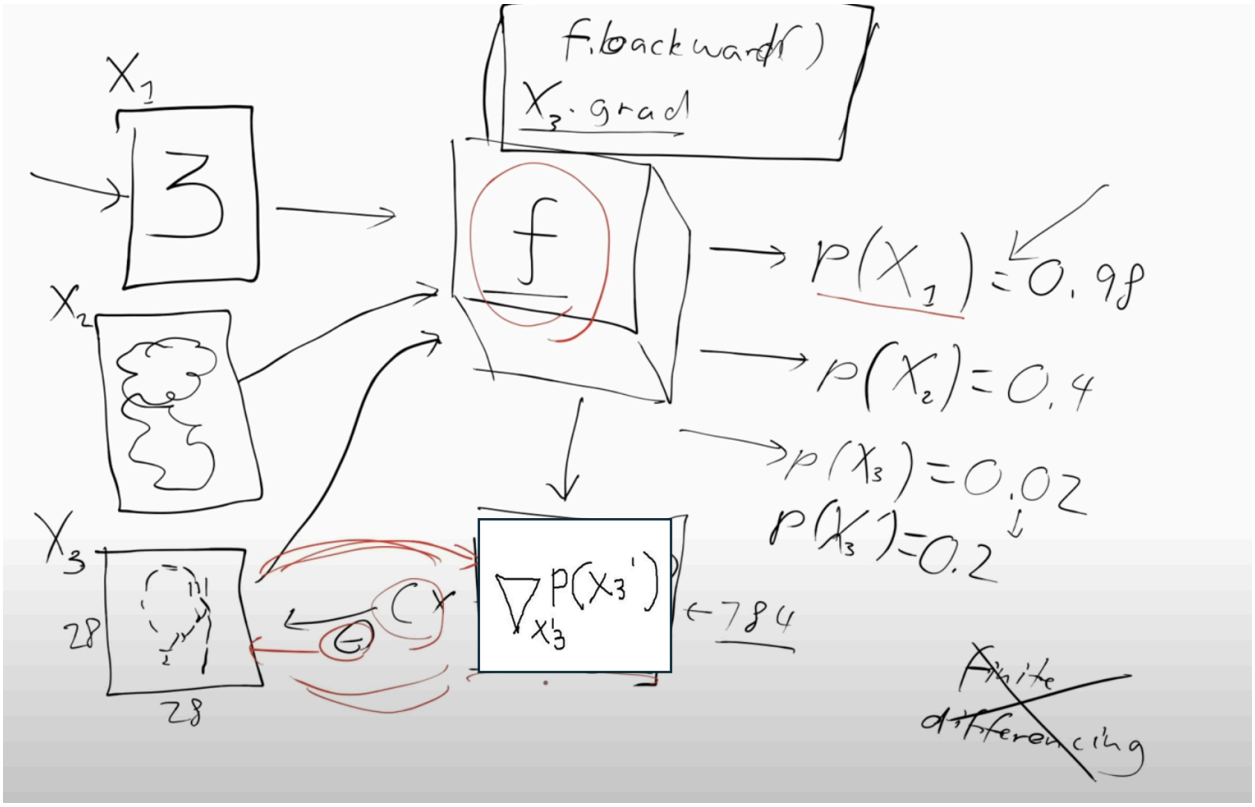
Dream Booth -> A similar goal like textual inversion, but a different process. Instead of creating a new token like textual inversion, we select an existing token in the vocabulary (a rarely used one), and fine-tune the model for a few hundred steps to make it close to the target images of

the new subjects we provide. When giving a prompt, we can then use the trained token to generate the image as we like.

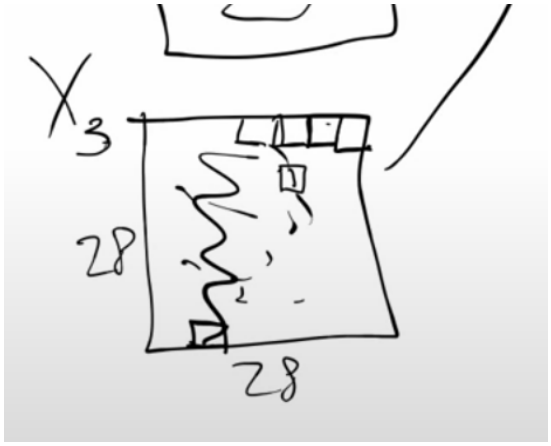
```
torch.manual_seed(1000)

prompt = "Painting of sks person in the style of Paul Signac"
images = db_pipe(prompt, num_images_per_prompt=4).images
image_grid(images, 1, 4)
```

```
  0%|          | 0/51 [00:00<?, ?it/s]
```

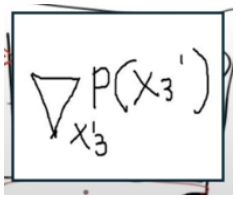
45:00 - Stable diffusion explained



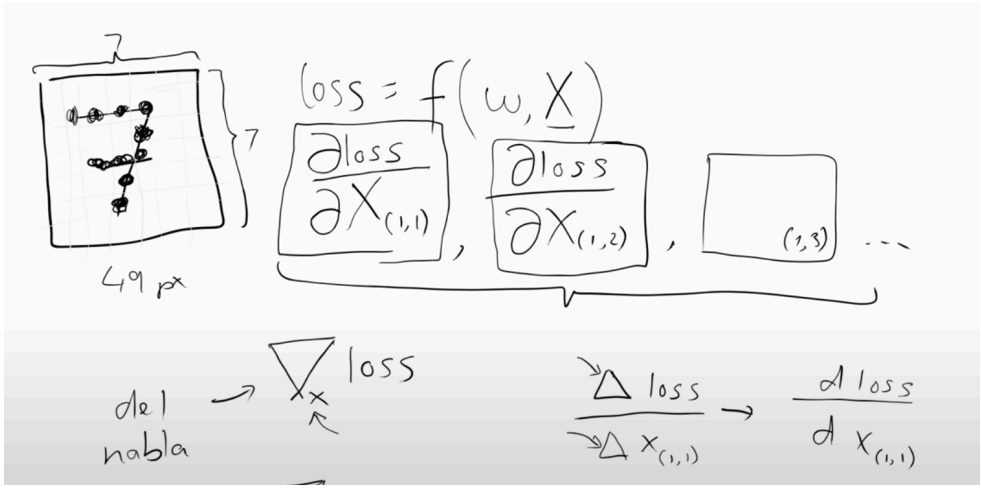
In a digit classification problem, for instance, we work to get better weights to understand the image at hand that compares to the ideal version of the digits. However, here, assume that f is a magic function that tells us the probability that the given image is a handwritten digit. For the first image, the probability that the given image is a handwritten digit is 0.98. Similarly, in the second image, it is kinda like eight and the probability that it is a handwritten digit is 0.4.



However, in the third case, Initially it is like this and the probability might be 0.02. Here, we can see that, maybe if I change a certain pixel to something else, like lightening or darkening a pixel, does it affect the probability that the image is a handwritten image? **So, in totality, it means that instead of changing the weights like we usually do, how can we change the input image so that we can get a better probability value.**



This function corresponds to the vector that has the change of probability values corresponding to the changes in each pixel (784 valued-vector in this case) with respect to change in X3 pixel values(Partial derivatives). So, this function has 784 values and we change the pixel values that bring the most change in probability values. We do this iteratively, and end up with good handwritten digit images. These partial derivatives can be calculated by the **Finite difference** method, like we usually do. However, this is slow and instead we can use the f.backward() function and X3.grad() to get the changes.



Hence, it is worthwhile to notice that we are not optimizing 'f' anywhere(corresponding weights). We are just optimizing the values of pixels as we go to get better probability values from f.

1:14:37 - Creating a neural network to predict noise in an image

In other words how can we create a magic function f that gives us the probability that the given input image is a handwritten digit?

1:27:46 - Working with images and compressing the data with autoencoders

1:40:12 - Explaining latents that will be input into the unet

1:43:54 - Adding text as one hot encoded input to the noise and drawing (aka guidance)

1:47:06 - How to represent numbers vs text embeddings in our model with CLIP encoders

1:53:13 - CLIP encoder loss function

2:00:55 - Caveat regarding "time steps"

2:07:04 Why don't we do this all in one step