Stable Diffusion from the foundations

Silver Rubanza

August 18, 2025

Stable Diffusion from the foundations

I am creating a series of blog posts based on the first lesson of part 2 of the fast.ai course [Practical Deep Learning for Coders], and I would encourage anyone interested in experimenting with machine learning to take this course.

We shall start by covering the theory in part 1, doing a deeper dive into the maths in a part 2, then finally in part 3 we shall implement Stable Diffusion from scratch using PyTorch, etc.

Theory — Part 1

Below, I shall go through the theoretical concepts and show the inner workings of the Stable Diffusion algorithm. The work shown here is based on lesson 9 and this blog post by Rekil Prashanth, whose work really helped me understand the concepts I go through below of the practical deep learning for coders course.

So nothing here is really my invention, but rather a sort of summary to remind me of the concepts in a quick dive. So let's start off.

Magic function, f.

So we want to create a function that can be used to generate handwritten digits. Now how would we go about doing that?

So imagine that we have a magic function, `f` which, when passed in an image, returns the probability of that image being a handwritten digit. For example, if we pass in an image, X1, we get back a probability score of that being a handwritten digit. The images are created by drawing a number and putting some random noise on top of it.

Passing in an image with more noise returns a lower probability. So now, how can we adjust our image such that we improve the probability that it is a handwritten digit?

Figure 1: magic function f

We need to find a way to adjust the image in that it improves the image, for example, in X3, where our number is a 1, we know that 1 does not have black pixels in the bottom right corner, so removing those should improve the probability that it is a handwritten digit.

So let's run with that, for each image, we can adjust the pixels of the image such that the probability of it being a handwritten digit increases.

In the end, we have a way to find out in which direction and by which magnitude we can adjust our pixel values in order to improve the probability of an image being a handwritten digit. We want the gradient of the probability that X3 is a handwritten digit with respect to the pixels of x3 (`p(X3)/ X3`). For every single pixel, the rate of change with respect to the pixel values can be expressed as `p(x3)/ (x3)` but when we do this for every pixel the becomes a nabla, .

At the end of each step, we should have a less noisy image with an improved probability score, which we can then use as the input.

We can now compute the gradient with respect to every pixel and adjust every pixel by subtracting the gradient multiplied by a small learning rate, which we can call `lr` from the original pixel value.

```
new_pixel_value (original_pixel_value - (gradient * lr))
```

The learning rate helps control the size of the steps we take. As we shall see later on, denoising an image is an iterative process, and it does not provide good results if we try to do it in one shot.

So we now have a function that can turn any noisy input into a handwritten digit. We are currently passing in each pixel one at a time to calculate the derivative by finite differencing. We can do this at once using analytical derivatives by calling `f.backward()`, then `X3.grad`. This is a faster and more efficient way of getting the gradients.

Training a Unet

We now have a magic function, `f` which tells us which pixels to change and by how much to make any noisy input look more like a handwritten digit.

So now we need to create such a function. Before we do this, we need some training data. We can do this by getting images of handwritten digits and chucking some random noise on top of them like in the image below

Figure 2: Training Data

Rather than come up with a probability score for the amount of noise in an image, let's say the amount of noise tells us how much an image is like a handwritten digit. Something with little amounts of noise looks more like a handwritten digit and vice versa.

As in the image above, we can see that a noisy image is just the handwritten digit plus the noise; for example, the number 7 is just the actual number 7 plus some noise.

So if we pass in a noisy image to our function, it should output the amount of noise; we then calculate the MSE between the predicted output and the original noise values, whose score we want to minimize.

Subtracting the predicted noise from the original noise values leaves us with a less noisy image, which acts as the next input for our function, `f`. We shall call the above function a `Unet`.

Enter the Variational AutoEncoder (VAE).

Now, our current image above is 28 x 28, which results in 784 pixel values. Now most modern-day photos have dimensions in the thousands; for example, a 6 MB screenshot on my phone is 1125 x 2436, which results in 2,740,500 different pixel values, and this is just one image, yet we need a number of images to do proper training.

To help deal with this complexity, we can push our image from the pixel space into the latent space. This helps reduce the image size while still retaining most of the information from the image. This significantly improves the time for our `Unet` training, as it operates on image latents, which are significantly smaller.

We can do this with an encoder and a decoder, as we shall see below.

In our encoder, we take the images and reduce the number of features using a stride 2 convolution while increasing the number of channels. Finally, at the end, we add a block to squish down the number of channels. We can now do the inverse process using the decoder; we can calculate the MSE between the output and the input and try to minimize this.

At first, we get random outputs, but at the end of the training, we should get back something that looks like our input.

This gives us a model that can take an image and compress it into its latent representation; we can pass this latent representation to our `Unet`. Our `Unet` then outputs the prediction of noise given noisy image latents.

The VAE compression works because, as we see above, we have a way to get the image encoded and get back something similar when passed through the decoder. It works because images have a lot of details that are not necessarily needed.

So the VAE learns patterns in the data that make up an image, like edges, etc. Once it learns the essence/representation that preserves the structure and style well enough while ignoring

what isn't necessarily needed, it is able to learn to compress an image using only the relevant details, but enough so that it can easily map noisy images to images.

Now that we have our noise predictions, we can subtract the noise from the noisy image latents, leaving us with less noisy latents and later denoised latents, which we can convert back into an approximation of the denoised image by passing them to our decoder.

Let us illustrate this with the image from lesson 9 of the class that shows a 512 x 512 RGB image which has 786,432 (512 x 512 x 3) pixel values.

Figure 3: enc_dec.png

Now if you look above, you will see that we went from 512 x 512 pixel image to 64 x 64 image reprsentation which we can still use to train our `Unet`. This is compression by a factor of 8.

Enter the Contrastive Language Image pre-training model (CLIP)

So if you notice when using Stable Diffusion models, many times you pass in an actual text prompt to sort of guide the model. So below, let's see how we can get our model to take a text prompt as additional input.

Let's start with our example above. Say we want to get our image to generate a handwritten digit for the number `3`. We would want to have a way to pass in our text alongside the noise as input to our `Unet`.

This can be done by passing in a one-hot encoding version of the number. With this extra information of what the original image was, the `Unet` should be able to do a better job of predicting the noise. We call this guidance.

Unfortunately, one-hot encoding may not be a robust way of doing this; imagine if we had a text such as "A cute kitten." To one-hot encode this, we would have to create a representation for every single word, which seems quite inefficient.

There is an alternative workaround: we can create a model that takes an image and a text description of the image and creates embeddings for the text that are close to the embeddings for its image, representing what the image looks like.

We start by downloading image and caption/tag pairs. We then pass the image to an image encoder and the text to a text encoder; both of them return embeddings that represent their input.

I now calculate the similarity by taking the dot product of both embedding vectors; the higher the score, the more similar the embeddings, which is what we want.

We now put the images and text in a grid as shown below, where we can put the pair embeddings. We want only the diagonals for the matching pairs to have a high score, so we

create a loss function that takes the matching pair embeddings and subtracts non-matching pair embeddings. Likewise, we then aim to maximize this, as we can see below.

Figure 4: Jeremy's Original Example

This loss is called `contrastive loss`, and the pair of models that we use to calculate our embeddings are called `CLIP (Contrastive Language-Image Pre-training)`

So now if we pass in similar text captions such as "A cute cat," "A cute kitten," and "A furry kitten with a nice coating," they should all give similar embeddings. We can now pass the text captions to our model, and it returns the embeddings, which are then passed in alongside the noisy latents to the `Unet`.

Time steps

We usually add varying amounts of noise to the the images, based of a noise schedule function. We can create something like this with an x -axis, `t` that ranges from 1 to 1000 and a y-axis that represents the amount of noise, something like what we have below

Figure 5: Noise Scheduler Function.png

Such scheduler functions are usually decreasing, so what we do is pick a random number from our schedule function and use the noise value at that point for our selected image. This is usually referred to as the time step.

Picking the value where t=0, meaning pure noise, results in a very bad prediction. This is because our `Unet` is trained to deal with images that have some bits of random noise, not total noise. This is because we never actually pass our model pure noise as input during training, so the model would not know how to deal with such a situation.

Think of it like navigating in the dark; it would be impossible to know which direction to move in to reach a certain point, but if you have a bit of light in the direction, you can slowly get to the final destination.

So denoising an image turns out to be an iterative process, where you slowly move towards your goal by subtracting a little bit of noise at each point, ending up with a slightly less noisy image at every step until you reach something that looks like the image you are expecting. This is why we multiply our predicted gradients by the learning rate, which ensures that we take small steps in the right direction as opposed to trying to do the diffusion process in one step.

In every mini batch, you choose a random amount of noise or pick a point on the noise schedule and assign the y-axis value at that point as the noise. We can then use those varying amounts

of noise to create the noisy images that we pass to our `Unet` to train it to predict the amount of noise. Adding the noise and subtracting the noise is done by the scheduler.

Another way of looking at this is we can actually just use the exact noise we added to the image; doing this would mean we wouldn't have to pass in `t`. With this, the diffusers stop looking like differential equations and more like optimizers. With this, we can experiment with using different loss functions like perceptual loss, etc. (Something to experiment with)

Credits

- Lesson 9 notes by Rekil Prashanth
- Practical deep learning for coders Lesson 9