Jonathan Smith
Deep Learning
Diffusion Notebook

In this project, I was essentially constructed a robot that can recognize images and unblur pictures. First I changed the runtime to GPU for storage and computation functions. GPUs are great for this because depending on the training time and dataset used, an extensive amount of resources can be expended. Next, I grabbed the toolkits and imported torch which is the main library that was going to be used for the project. Matplotlib is used as a tool for drawing plots and einops which is like a magic tool that reshapes data. I chose to go with the first option using the MNIST dataset which is a classic dataset of 60,000 small, 28x28 pictures of handwritten digits of 0-9. Since the picture cant be directly fed into the AI model, I turned it into numbers by using transforms.ToTensor(). This turned the picture into a grid of numbers from 0 to 1. The (transforms.Normalize((0.5,),, (0.5))) was used to shift the number between -1 and 1 instead to assist the model with learning stability.

DataLoaders is used because it is composed of over 60,000 images and they cannot all be shown to the model at one time. Train_loader is used like a schoolbus that grabs the batch of images and takes them to the model to learn. the (val_loader) is used for the testing data and measures the model to see if it is learning or just overfitting. Next, I built the U-Net model which is like a set of LEGO bricks. The (GELUConvBlock) is used as the most basic piece. Next, I used the (nn.Sequential) block to scan over the images for patterns like curves, lines and edges with nn.Conv2d. We used (nn.GroupNorm) as a stabilizer to keep the numbers from getting too big or too small. The (nn.GELU) is used to decide if the pattern it just discovered is even important enough to pass on to the rest of the model for learning.

The U shape was then minimized using the (RearrangePoolBlock). The string (‘b c (h2) (w 2) -> b (c4) h w’) tells the model to take 2x2 patches of pixels and move them into the channel dimension. I had to add a convolution layer at the end to map

the channel dimension back down. The DownBlock, encoder, is the left side of the U and the UpBlock, expander, is used for the right side of the U.

The UNet Class is then used as like a master blueprint for the model. I then created the downs, mids, and ups lists for the model. Then I used he conditioning method to instruct the model on what to do. (Self.time_embed) turns the picture into a vector for the model to read. (Self.class_embed) turns the digit into a vector and informs the UNet on what number the final image is supposed to be.

I then used forward to take the foggy image and run it down the downblock before we injected the time and class vectors. I then ran up the upblock and fed the model the skips from each step. A final convolution was then made that combined all the features into a final image with 1 channel using self.final_conv. I then implemented the forward process by using the (add_noise) as like a fog machine. Starting with a clear image I added random static used by using the formula (x_t = sqrt_aplha_bar_t * x_0 + sqrt_one_minus_alpha_bar_t * noise). This was essentially noisy image = (% of the original image) + (% of pure noise). At t=1, its 1% image, 99% noise. I then removed the noise using the reverse process. The model then predicts the noise that was added. I then used the formula to subtract a little bit of noise to get a clearer image using (x_t-1). I then repeated that process 100 using x_100 and the pure noise slowly became a (digit x_0).

The last step was the training session. For this I used the train_step function to study the problem. The (x) and (c) labels provided me with a clean batch of images. I then moved them to the GPU using (.to(device)). I then picked a random fog level using (t) for each image which foreces the model to learn to de-fog at all levels not just one. I then called the (add_noise(x, t) to get the foggy images (x_t) and the actual noise we added. I then called he (predicted_noise = model(x_t, t, c, c_mask) to make

the model make a prediction. I then used (loss = F.mse_loss(predicted_noise, actual_noise). This is the mean squared error. The (loss.backward() and optimizer.step()) is the learning part of the training session. This nudges all the internal weights to be as accurate as possible. I received a runtime error when my training step was trying to pass a one-hot vecor to the model so I passed the (c) label directly to the model.

The main training loop was implemented after this. I used (for epoch in range (EPOCHS) to pass through all 60,000 images and did it 30 times. The (model.train()) puts the model in study mode. I used (dor step, (images, labels) in enumerate(train_loader) to display 64 images at a time. The (model.eval()) puts the model in test mode with no learning just answers. I used (with_torch.no_grad) to run all of the validation images from (val_loader) and get a (val_loss) and get a real validation number. This told me if the model was learning the data or just merely overfitting. I used (scheduler.step(avg_val_loss) to teach the model and if it did not improve after 5 epochs it was going to lower the rate to assist in fine-tuning the model. I used (safe_save_model) to save a copy of the model every time a new high score was achieved using (using_val_loss).

During the forward diffusion process I created a fog machine for the model and displayed it using (show_noise_progression) plot. I started with a perfectly clear image (x_0) and then added random, static like noise making the image 99% clear and 1% noise. Next, I added a little more noise to the already noisy image. I repeated this step 100 times as I could tell a digit was forming but wit was very blurry by the 50% noise picture. By the 100% noise (t=100), the original image was completely gone, rendered to nothing but pure , random static. I used this with the (add_noise) function in the code to instantly calculate what the image should look like in its final prediction.

The added noise was gradually added to the model instead of all at once so that the reverse process is learnable. instead of showing the model a picture of the number and then asking it to turn it into a digit, I showed the model a picture of the digit that is foggy and asked the model what does the image look like 1% less foggy (t=99) enabling the model to draw a tiny prediction. I then asked it to show the (t=99) image and then asked it to predict the (t=98) image. By breaking this problem into tiny easy problems for the model, it allowed it to actually learn it. The gradual process creates a path for the AI model to follow backward. In the forward process, the digit was recognizable up to t=70 or t=80 (70-80% noise). Yes, it varies by images because an image containing an 8 may continue to look like a blob because of its shape containing the wholes.

The UNet architecture was perfect for the diffusion model because it is a image to image task. Its job is to take a noisy image as input and output a predicated noise image of the exact same size.The down-path encoder shrinks the image and compresses the information, thus forcing the model to understand the content instead of just memorizing the pixels. The right side up-path is the right side of the image and it expands the image back to its original size using the info from the bottom not to reconstruct it. A simpler architecture would lose all the fine grained details and have a blurry output due to not catching the sharp edges. The skip connections are important because it solves this issue by acting as a cheat sheet. They are connections that skip the shape and connect the encoder and decoder to each other.

The model was conditioned to generate images off of time and class. Conditioning was used to essentially steer the UNet. This is how I told the model exactly what to draw. There are 10 classes of digits ranging from 0-9 and I created a loop table called (self.class_embed = nn.Embedding(10, c_embed_dim). Since I

wanted to draw a certain number, I plugged it into the model's index as a (Long) tensor. The (nn.Embedding) layer looks up the index and pulled out the unique vector. I then injected the vecor into the bottom of the U right after the block (c_proj = self.mid_c_proj(c_embed).view(b,c_dim, 1, 1) and then (x = x + c_proj). This injection told the UNet that whatever noise it was about to predict to make sure it was removed in the final prediction.

The loss value told me the mistake score of the model's prediction. The model's job was the predict the exact amount of noise that was added to the image and analyze it. The (train_step) function calculated (loss = F.mse_loss(predicted_noise, actual noise). A high loss (1.0) would mean the model's prediction was inaccurate, and a loss loss (0.01) means the model's prediction is almost identical to the real noise. As the loss value decreases, the model is continuously improving to become a clearer image.

Throughout the training process the generated images would look like pure, random static on the first epoch. By the 5th, there was a faint, blurry blob in the center showing the model was at least learning something and not just random noise everywhere. By the 15th epoch, the blobs would start to take the general shape of the digit showing the defogging process actually working. By the 30th and final epoch, the image was much sharper and looked like a handwritten digit.

The time embedding process was extremely important because the model's job is different at every step. At (t=99) the image was almost 99% noise making the model predict almost all of the noise to even make the first step. At (t=10) the image was almost 10% noise and the model had to only predict the little bit of noise that was added to the image to do the final polishing. By passing (t) into the (self_time_embed) and adding it to the middle, I instructed the model to predict the amount of noise

associated with this specific step. This allowed the model to learn 100 different de-fogging strategies, one for each level of fogginess. My clip scores told me that my model was accurate as it scored high on the handwritten and clear scores. My images generated at the end received the highest scores, while the initial images with all the noise received a much lower score.

The model's difficulty is based on feature complexity and class similarity which would explain why some images are easier for the model o generate. A digit like a "1" is pretty simply because its just a line, but a digit like a "6" or an "8" would be much more difficult because of the loops associated with them. Also digits that look like others are harder for the model to understand as well. Digits like "3" and "5" have similar curves in them that can easily confuse the model, just like "4" and "9". The model has to learn subtle features to separate them apart.

CLIP guidance is used to improve the diffusion model's generation process. Instead of using the CLIP as a judge its used as a tutor for the model. The technique is started by the reverse process at (t=99) with pure noise. At each step, our UNet predicts the noise and gives me a clearer image. I showed the slightly clearer image to CLIP to calculate how much it looks like the text prompt. I then modified the image to just a little bit in a direction that the CLIP said it looks more like the digit I selected. I then used that image as the input for the next step (t=97). By repeating this, I was able to essentially guide the diffusion process, forcing it to not only remove the noise but also remove the noise in a way that maximizes the CLIP score. This results in images that are much clearer and accurate to the prompt.

This type of model can be helpful in the real world because a user can try to remove someone else from a photo, and the diffusion model can not only remove them, but also conjure up a different background behind them. Another use is

someone using the model to turn the sketch of a face to a realistic photo using a style transfer technique. Another useful real world application would be a self driving car using an AI system needing to be trained on millions of pictures of stop signs. Instead of driving around for years, I can train the AI model so that it can generate millions of new, unique stop signs in different conditions, angles, and weather to help train the car. Another application would be generating new protein structures for drug discovery and new molecular designs.

The model that I created does have some limitations. It can only make 28x28 pixels which are smaller than the app icon. It only knows grayscale and not RGB color. The reverse process can also be slow and extensive because the UNet has to be ran 100 times to make a single image. It also is only trained on the MNIST dataset, making it only able to draw digits. It has no idea what a dog, mouse, or chair is.

If I was to continue to develop this project and improve it I would train the model by randomly dropping the class label. At generation time, I would make 2 predictions: one with the label and one without (conditioned). I would then move the final image away from the unconditioned prediction towards the label prediction. This would make the image prediction stronger. Another way that I would improve the model would be to make a cosine schedule which adds noise slowly at first and then accelerates towards the end. This would give the model more steps at the end of generation to work on fine-tuning the details, which again should render sharper images. Lastly, I would scale up the model and data bu switching to the CIFAR-10 dataset which has a larger set of classes to train the model on. The dataset also has color images that are 32x32. A larger dataset would essentially make the model smarter and would increase the UNet's channel depth . This would give the model more parameters to learn about more complex patterns.

Jonathan Smith
Deep Learning
Diffusion Notebook

Overall, I learned a lot in this project. I learned how to build a model that can actually generate images which I enjoyed. I also enjoyed troubleshooting my code as well and making sure I was completing the assignment correctly. This project gave me a deeper understanding on how models work and how they can be used in everyday applications. I can only imagine how advanced this technology is going to be in the future. I can imagine a situation where models are going to have to either be trained on less data, or they are going to be trained on so much that it can dwarf the limitations even present today.