(https://www.nvidia.com/en-us/training/)

3. Optimizations ¶

Actualmente, el modelo está experimentando el problema del tablero de ajedrez (https://distill.pub/2016/deconv-checkerboard/).

Afortunadamente, tenemos algunos trucos bajo la manga para resolver esto y, en general, mejorar el rendimiento del modelo.

Objetivos de Aprendizaje

Los objetivos de este notebook son:

- Implementar la Normalización de Grupos
- Implementar GELU
- Implementar Rearrange Pooling
- Implementar Embeddings de Posición Sinusoidales
- Definir una función de difusión inversa para emular p
- Intentar generar artículos de ropa (de nuevo)

Como antes, usemos fashionMIST para experimentar:

In [21]:
```
!pip install einops
```

```
Requirement already satisfied: einops in /Users/humbertofariasaroca/miniforge3/envs/pytorch/lib/python3.9/si
te-packages (0.8.0)
DEPRECATION: mermaid 0.3.2 has a non-standard dependency specifier torch>=1.7torchvision. pip 24.1 will enfo
rce this behaviour change. A possible replacement is to upgrade to a newer version of mermaid or contact the
author to suggest that they release a version with a conforming dependency specifiers. Discussion can be fou
nd at https://github.com/pypa/pip/issues/12063 (https://github.com/pypa/pip/issues/12063)
```

In [32]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
from torch.optim import Adam

# Visualization tools
import matplotlib.pyplot as plt
from torchview import draw_graph
import graphviz
from IPython.display import Image

# User defined libraries
from utils import other_utils
from utils import ddpm_utils

IMG_SIZE = 16
IMG_CH = 1
BATCH_SIZE = 128
data, dataloader = other_utils.load_transformed_fashionMNIST(IMG_SIZE, BATCH_SIZE)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Hemos creado un archivo ddpm_util.py (ddpm_utils.py) con una clase `DDPM` para agrupar nuestras funciones de difusión. Usemos esta clase para configurar el mismo programa Beta que utilizamos anteriormente.

In [33]:
```
# Definir el número de filas y columnas para la visualización de las imágenes generadas
nrows = 10
ncols = 15

# Definir el número total de pasos de tiempo T como el producto de filas y columnas
T = nrows * ncols

# Definir los valores inicial y final para el programa Beta
B_start = 0.0001
B_end = 0.02

# Crear un tensor de valores Beta linealmente espaciados desde B_start hasta B_end
B = torch.linspace(B_start, B_end, T).to(device)

# Crear una instancia de la clase DDPM con el programa Beta y el dispositivo especificado
ddpm = ddpm_utils.DDPM(B, device)
```

3.1 Group Normalization and GELU

La primera mejora que veremos es la optimización de nuestro proceso de convolución estándar. Reutilizaremos este bloque muchas veces a lo largo de nuestra red neuronal, por lo que es una pieza importante para hacerlo bien.

3.1.1 Group Normalization

Batch Normalization (https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338) convierte la salida de cada canal del kernel en un z-score (https://www.nlm.nih.gov/oet/ed/stats/02-910.html). Lo hace calculando la media y la desviación estándar en un lote de entradas. Esto es ineficaz si el tamaño del lote es pequeño.

Por otro lado, Group Normalization (https://arxiv.org/pdf/1803.08494.pdf) normaliza la salida de un grupo de kernels para cada imagen de muestra, "agrupando" efectivamente un conjunto de características.

Considerando que las imágenes en color tienen múltiples canales de color, esto puede tener un impacto interesante en los colores de salida de las imágenes generadas. ¡Prueba a experimentar para ver el efecto!

3.1.2 GELU

ReLU (https://www.kaggle.com/code/dansbecker/rectified-linear-units-relu-in-deep-learning) es una opción popular para una función de activación porque es computacionalmente rápida y fácil de calcular el gradiente. Desafortunadamente, no es perfecta. Cuando el término de sesgo se vuelve muy negativo, una neurona ReLU "muere" (https://datascience.stackexchange.com/questions/5706/what-is-the-dying-relu-problem-in-neural-networks) porque tanto su salida como su gradiente son cero.

A un ligero costo en potencia computacional, GELU (https://arxiv.org/pdf/1606.08415.pdf) busca rectificar la unidad lineal rectificada imitando la forma de la función ReLU mientras evita un gradiente cero.

En este pequeño ejemplo con FashionMNIST, es poco probable que veamos neuronas muertas. Sin embargo, cuanto más grande sea el modelo, más probable es que pueda enfrentar el fenómeno de la ReLU moribunda.

In [34]:
```
class GELUConvBlock(nn.Module):
    def __init__(
        self, in_ch, out_ch, group_size):
        super().__init__()
        layers = [
            nn.Conv2d(in_ch, out_ch, 3, 1, 1),
            nn.GroupNorm(group_size, out_ch),
            nn.GELU()
        ]
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)
```

3.2 Rearrange pooling

En el notebook anterior, usamos Max Pooling (https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html) para reducir a la mitad el tamaño de nuestra imagen latente, pero ¿es esa la mejor técnica? Hay muchos tipos de capas de pooling (https://pytorch.org/docs/stable/nn.html#pooling-layers), incluyendo Min Pooling y Average Pooling. ¿Qué tal si dejamos que la red neuronal decida qué es importante?

Entra la biblioteca einops (https://einops.rocks/1-einops-basics/) y la capa Rearrange (https://einops.rocks/api/rearrange/). Podemos asignar cada capa una variable y usar eso para reorganizar nuestros valores. Además, podemos usar paréntesis `()` para identificar un conjunto de variables que se multiplican entre sí.

Por ejemplo, en el bloque de código a continuación, tenemos:

```
Rearrange("b c (h p1) (w p2) -> b (c p1 p2) h w", p1=2, p2=2)
```

- `b` es nuestra dimensión de lote
- `c` es nuestra dimensión de canal
- `h` es nuestra dimensión de altura
- `w` es nuestra dimensión de ancho

También tenemos un valor `p1` y `p2` que son ambos iguales a `2`. La porción izquierda de la ecuación antes de la flecha está diciendo "divide las dimensiones de altura y ancho por la mitad". La porción derecha de la ecuación después de la flecha está diciendo "apila las dimensiones divididas a lo largo de la dimensión de canal".

El bloque de código a continuación configura una `test_image` para practicar. Intenta intercambiar `h` con `p1` en el lado izquierdo de la flecha. ¿Qué sucede? ¿Qué pasa cuando `w` y `p2` se intercambian? ¿Qué sucede cuando `p1` se establece en `3` en lugar de

In [35]:
```
from einops.layers.torch import Rearrange

rearrange = Rearrange("b c (h p1) (w p2) -> b (c p1 p2) h w", p1=2, p2=2)

test_image = [
    [
        [
            [1, 2, 3, 4, 5, 6],
            [7, 8, 9, 10, 11, 12],
            [13, 14, 15, 16, 17, 18],
            [19, 20, 21, 22, 23, 24],
            [25, 26, 27, 28, 29, 30],
            [31, 32, 33, 34, 35, 36],
        ]
    ]
]
test_image = torch.tensor(test_image)
print(test_image)
output = rearrange(test_image)
output
```

```
tensor([[[[ 1,  2,  3,  4,  5,  6],
          [ 7,  8,  9, 10, 11, 12],
          [13, 14, 15, 16, 17, 18],
          [19, 20, 21, 22, 23, 24],
          [25, 26, 27, 28, 29, 30],
          [31, 32, 33, 34, 35, 36]]]])
```

Out[35]:
```
tensor([[[[ 1,  3,  5],
          [13, 15, 17],
          [25, 27, 29]],

         [[ 2,  4,  6],
          [14, 16, 18],
          [26, 28, 30]],

         [[ 7,  9, 11],
          [19, 21, 23],
          [31, 33, 35]],

         [[ 8, 10, 12],
          [20, 22, 24],
          [32, 34, 36]]]])
```

A continuación, podemos pasar esto a través de nuestro `GELUConvBlock` para dejar que la red neuronal decida cómo quiere ponderar los valores dentro de nuestro "pool". ¿Notas el `4*in_chs` como un parámetro del `GELUConvBlock` ? Esto se debe a que la dimensión del canal ahora es (p1 \times p2) más grande.

In [36]:
```
class RearrangePoolBlock(nn.Module):
    def __init__(self, in_chs, group_size):
        super().__init__()
        self.rearrange = Rearrange("b c (h p1) (w p2) -> b (c p1 p2) h w", p1=2, p2=2)
        self.conv = GELUConvBlock(4 * in_chs, in_chs, group_size)

    def forward(self, x):
        x = self.rearrange(x)
        return self.conv(x)
```

Ahora tenemos los componentes para redefinir nuestros `DownBlock` s y `UpBlock` s. Se han añadido múltiples `GELUConvBlock` s para ayudar a combatir el problema del tablero de ajedrez.

In [37]:
```
class DownBlock(nn.Module):
    def __init__(self, in_chs, out_chs, group_size):
        super(DownBlock, self).__init__()
        layers = [
            GELUConvBlock(in_chs, out_chs, group_size),
            GELUConvBlock(out_chs, out_chs, group_size),
            RearrangePoolBlock(out_chs, group_size)
        ]
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)
```

Type *Markdown* and LaTeX: α^2

In [38]:
```
class UpBlock(nn.Module):
    def __init__(self, in_chs, out_chs, group_size):
        super(UpBlock, self).__init__()
        layers = [
            nn.ConvTranspose2d(2 * in_chs, out_chs, 2, 2),
            GELUConvBlock(out_chs, out_chs, group_size),
            GELUConvBlock(out_chs, out_chs, group_size),
            GELUConvBlock(out_chs, out_chs, group_size),
            GELUConvBlock(out_chs, out_chs, group_size),
        ]
        self.model = nn.Sequential(*layers)

    def forward(self, x, skip):
        x = torch.cat((x, skip), 1)
        x = self.model(x)
        return x
```

3.3 Time Embeddings

Cuanto mejor entienda el modelo el paso de tiempo en el que se encuentra durante el proceso de difusión inversa, mejor podrá identificar correctamente el ruido añadido. En el notebook anterior, creamos una incrustación para `t/T`. ¿Podemos ayudar al modelo a interpretar esto mejor?

Antes de los modelos de difusión, este era un problema que afectaba al procesamiento del lenguaje natural. Para diálogos largos, ¿cómo podemos capturar en qué parte nos encontramos? El objetivo era encontrar una manera de representar de manera única una amplia gama de números discretos con un pequeño número de números continuos. Usar un solo flotante es ineficaz ya que la red neuronal interpretará los pasos de tiempo como continuos en lugar de discretos. Investigadores (https://arxiv.org/pdf/1706.03762.pdf) finalmente se decidieron por una suma de senos y cosenos.

Para una explicación excelente de por qué esto funciona y cómo probablemente se desarrolló esta técnica, consulta el artículo de Jonathan Kernes Master Positional Encoding (https://towardsdatascience.com/master-positional-encoding-part-i-63c05d90a0c3).

In [27]:
```python
import math

class SinusoidalPositionEmbedBlock(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
```

TODO: Alimentaremos la salida del `SinusoidalPositionEmbedBlock` en nuestro `EmbedBlock`. Afortunadamente, nuestro `EmbedBlock` sigue sin cambios respecto a antes.

In [28]:
```
class EmbedBlock(nn.Module):
    def __init__(self, input_dim, emb_dim):
        super(EmbedBlock, self).__init__()
        self.input_dim = input_dim
        layers = [
            nn.Linear(input_dim, emb_dim),
            nn.GELU(),
            nn.Linear(emb_dim, emb_dim),
            nn.Unflatten(1, (emb_dim, 1, 1))
        ]
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        x = x.view(-1, self.input_dim)
        return self.model(x)
```

3.4 Residual Connections

El último truco para eliminar el problema del tablero de ajedrez es agregar más conexiones residuales o de salto. Podemos crear un `ResidualConvBlock` para nuestra convolución inicial. También podríamos agregar conexiones residuales en otros lugares, como dentro de nuestros "DownBlocks" y "UpBlocks".

In [29]:
```
class ResidualConvBlock(nn.Module):
    def __init__(self, in_chs, out_chs, group_size):
        super().__init__()
        self.conv1 = GELUConvBlock(in_chs, out_chs, group_size)
        self.conv2 = GELUConvBlock(out_chs, out_chs, group_size)

    def forward(self, x):
        x1 = self.conv1(x)
        x2 = self.conv2(x1)
        out = x1 + x2
        return out
```

A continuación se muestra el modelo actualizado. ¿Notaste el cambio en la última línea? Se ha añadido otra conexión de salto desde la salida de nuestro `ResidualConvBlock` hasta el bloque final `self.out` . Esta conexión es sorprendentemente poderosa y, de todos los cambios mencionados anteriormente, tuvo la mayor influencia en el problema del tablero de ajedrez para este conjunto de

datos.

TODO: Se han añadido un par de nuevas variables: `small_group_size` y `big_group_size` para la normalización de grupos. Ambas dependen de la variable `group_base_size`. Establece `group_base_size` en `3`, `4`, `5`, `6` o `7`. Uno de estos valores es correcto y el resto resultará en un error

In [41]:

```python
class UNet(nn.Module):
    def __init__(self):
        super().__init__()
        img_chs = IMG_CH  # Número de canales de la imagen de entrada
        down_chs = (64, 64, 128)  # Canales para las capas de "downsampling"
        up_chs = down_chs[::-1]  # Canales para las capas de "upsampling", inversa de down_chs
        latent_image_size = IMG_SIZE // 4  # Tamaño de la imagen latente
        t_dim = 8  # Dimensión para el embedding temporal
        group_size_base = 4  # Tamaño base para la normalización de grupos
        small_group_size = 2 * group_size_base  # Tamaño pequeño para la normalización de grupos
        big_group_size = 8 * group_size_base  # Tamaño grande para la normalización de grupos

        # Convolución inicial
        self.down0 = ResidualConvBlock(img_chs, down_chs[0], small_group_size)  # Bloque de convolución resid

        # Downsampling
        self.down1 = DownBlock(down_chs[0], down_chs[1], big_group_size)  # Primer bloque de downsampling
        self.down2 = DownBlock(down_chs[1], down_chs[2], big_group_size)  # Segundo bloque de downsampling
        self.to_vec = nn.Sequential(nn.Flatten(), nn.GELU())  # Aplanar y aplicar GELU

        # Embeddings
        self.dense_emb = nn.Sequential(
            nn.Linear(down_chs[2]*latent_image_size**2, down_chs[1]),
            nn.ReLU(),
            nn.Linear(down_chs[1], down_chs[1]),
            nn.ReLU(),
            nn.Linear(down_chs[1], down_chs[2]*latent_image_size**2),
            nn.ReLU()
        )

        self.sinusoidaltime = SinusoidalPositionEmbedBlock(t_dim)  # Bloque de embedding posicional sinusoida
        self.temb_1 = EmbedBlock(t_dim, up_chs[0])  # Primer bloque de embedding temporal
        self.temb_2 = EmbedBlock(t_dim, up_chs[1])  # Segundo bloque de embedding temporal

        # Upsampling
        self.up0 = nn.Sequential(
            nn.Unflatten(1, (up_chs[0], latent_image_size, latent_image_size)),
            GELUConvBlock(up_chs[0], up_chs[0], big_group_size)  # Bloque de convolución con GELU
        )
        self.up1 = UpBlock(up_chs[0], up_chs[1], big_group_size)  # Primer bloque de upsampling
        self.up2 = UpBlock(up_chs[1], up_chs[2], big_group_size)  # Segundo bloque de upsampling

        # Ajustar los canales de salida y una última concatenación
```

```
        self.out = nn.Sequential(
            nn.Conv2d(2 * up_chs[-1], up_chs[-1], 3, 1, 1),  # Convolución final
            nn.GroupNorm(small_group_size, up_chs[-1]),  # Normalización de grupo
            nn.ReLU(),
            nn.Conv2d(up_chs[-1], img_chs, 3, 1, 1)  # Convolución final para igualar los canales de salida
        )

    def forward(self, x, t):
        down0 = self.down0(x)  # Paso por el primer bloque de downsampling
        down1 = self.down1(down0)  # Paso por el segundo bloque de downsampling
        down2 = self.down2(down1)  # Paso por el tercer bloque de downsampling
        latent_vec = self.to_vec(down2)  # Aplanar y aplicar GELU

        latent_vec = self.dense_emb(latent_vec)  # Paso por las capas de embedding denso
        t = t.float() / T  # Convertir de [0, T] a [0, 1]
        t = self.sinusoidaltime(t)  # Aplicar embedding posicional sinusoidal
        temb_1 = self.temb_1(t)  # Aplicar el primer embedding temporal
        temb_2 = self.temb_2(t)  # Aplicar el segundo embedding temporal

        up0 = self.up0(latent_vec)  # Paso por el primer bloque de upsampling
        up1 = self.up1(up0 + temb_1, down2)  # Paso por el segundo bloque de upsampling con skip connection
        up2 = self.up2(up1 + temb_2, down1)  # Paso por el tercer bloque de upsampling con skip connection
        return self.out(torch.cat((up2, down0), 1))  # Concatenar y pasar por el bloque final
```

In [42]:
```
model = UNet()
print("Num params: ", sum(p.numel() for p in model.parameters()))
model = torch.compile(model.to(device))
```

```
Num params:  1979777
```

Finalmente, es hora de entrenar el modelo. Veamos si todos estos cambios han hecho una diferencia.

In [19]:
```
optimizer = Adam(model.parameters(), lr=0.001)
epochs = 5

model.train()
for epoch in range(epochs):
    for step, batch in enumerate(dataloader):
        optimizer.zero_grad()

        t = torch.randint(0, T, (BATCH_SIZE,), device=device).float()
        x = batch[0].to(device)
        loss = ddpm.get_loss(model, x, t)
        loss.backward()
        optimizer.step()

        if epoch % 1 == 0 and step % 100 == 0:
            print(f"Epoch {epoch} | step {step:03d} Loss: {loss.item()} ")
            ddpm.sample_images(model, IMG_CH, IMG_SIZE, ncols)
```

```
Epoch 0 | step 000 Loss: 1.3311713933944702
```

```
Epoch 0 | step 100 Loss: 0.2220052182674408
```

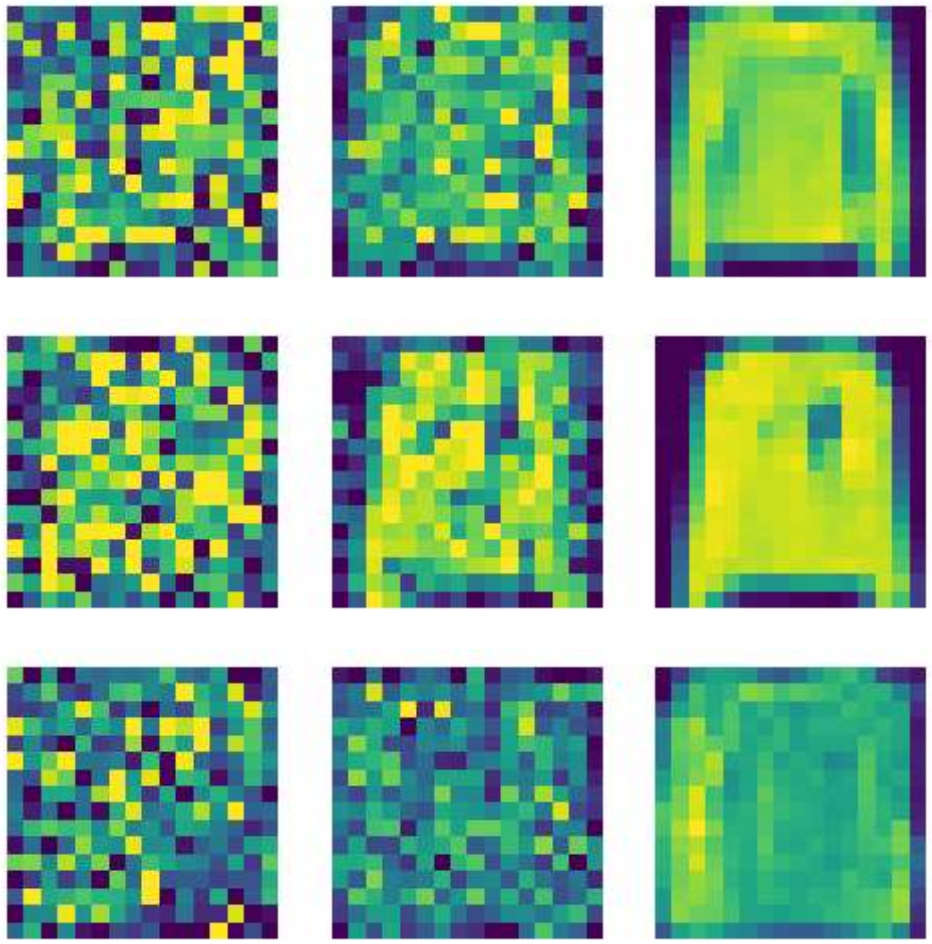
```
Epoch 0 | step 200 Loss: 0.13643315434455872
```

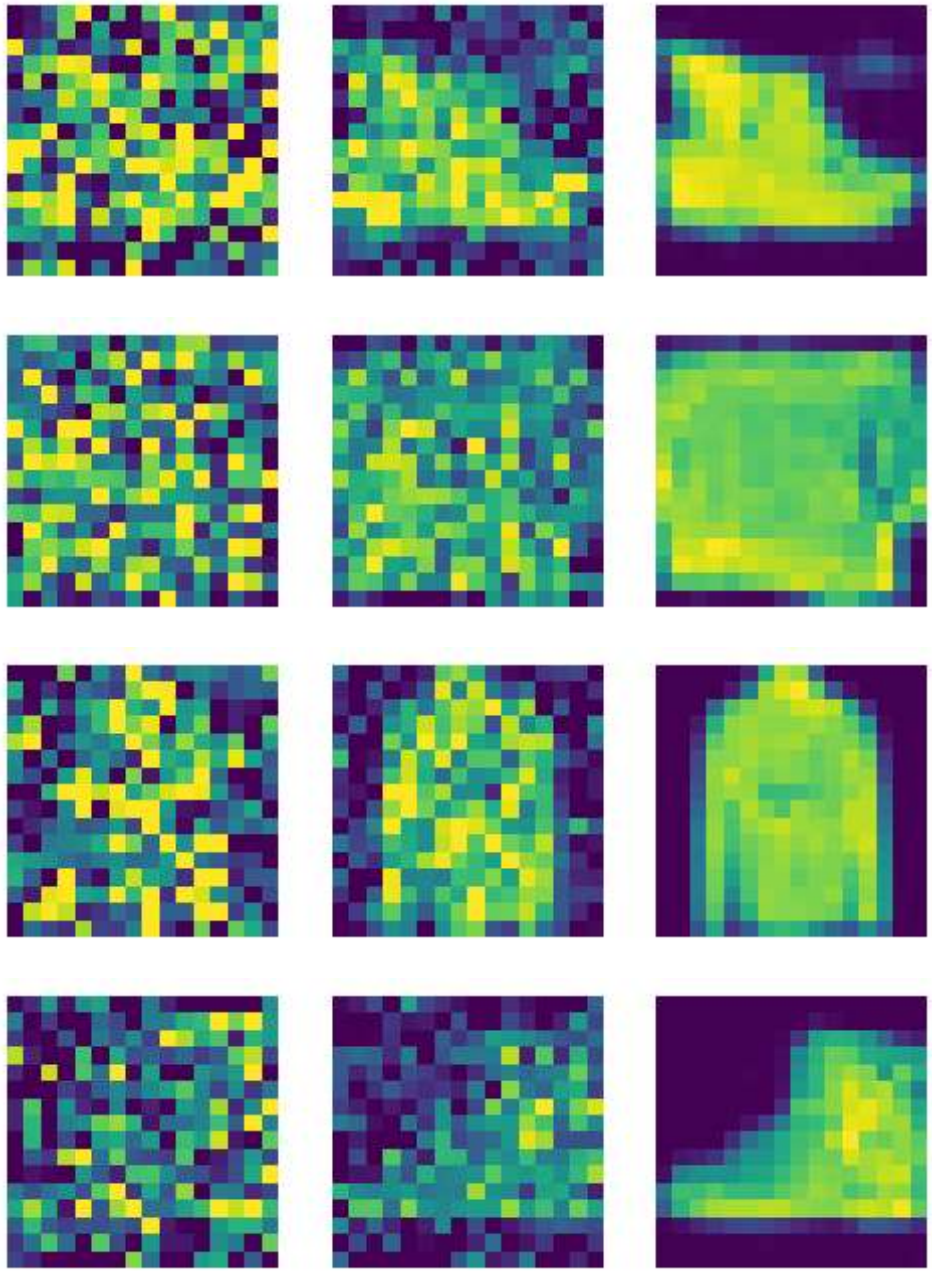
```
Epoch 0 | step 300 Loss: 0.10960675776004791
```

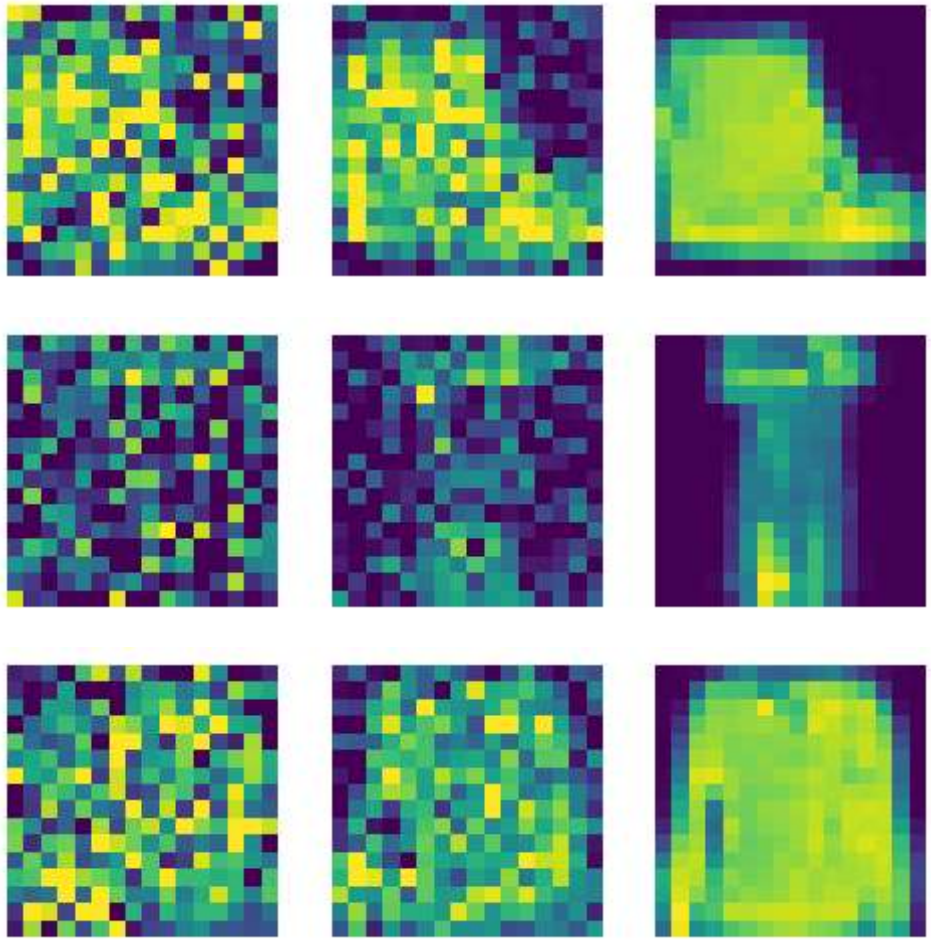
How about a closer look? Can you recognize a shoe, a purse, or a shirt?

In [20]:
```
model.eval()
plt.figure(figsize=(8,8))
ncols = 3 # Should evenly divide T
for _ in range(10):
    ddpm.sample_images(model, IMG_CH, IMG_SIZE, ncols)
```

```
<Figure size 800x800 with 0 Axes>
```







3.5 Next

Si no ves una clase particular como un zapato o una camisa, intenta ejecutar el bloque de código anterior nuevamente. Actualmente, nuestro modelo no acepta una entrada de categoría, por lo que el usuario no puede definir qué tipo de salida le gustaría. ¿Dónde está la diversión en eso?

In [ ]:
```
import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

(https://www.nvidia.com/en-us/training/)