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

1. From U-Net to Diffusion

Las U-Nets son un tipo de red neuronal convolucional que fue diseñada originalmente para la imagen médica. Por ejemplo, podríamos alimentar la red con una imagen de un corazón y podría devolver una imagen diferente destacando áreas potencialmente cancerosas.

¿Podemos usar este proceso para generar nuevas imágenes? Aquí tienes una idea: ¿qué tal si agregamos ruido a nuestras imágenes y luego usamos una U-Net para separar las imágenes del ruido? ¿Podríamos entonces alimentar el modelo con ruido y hacer que cree una imagen reconocible? ¡Vamos a intentarlo!

Objetivos de Aprendizaje

Los objetivos de este notebook son:

- Explorar el conjunto de datos FashionMNIST
- Construir una arquitectura U-Net
  - Construir un Bloque de Descenso
  - Construir un Bloque de Ascenso
- Entrenar un modelo para eliminar ruido de una imagen
- Intentar generar artículos de ropa

In [2]:
```
!pip install torchview
```

```
Collecting torchview
  Downloading torchview-0.2.6-py3-none-any.whl.metadata (12 kB)
Downloading torchview-0.2.6-py3-none-any.whl (25 kB)
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)
Installing collected packages: torchview
Successfully installed torchview-0.2.6
```

In [54]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torch.optim import Adam

# Visualization tools
import graphviz
from torchview import draw_graph
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
```

En PyTorch, podemos usar nuestra GPU en nuestras operaciones configurando el dispositivo (https://pytorch.org/docs/stable/tensor_attributes.html#torch.device) a `cuda` . La función `torch.cuda.is_available()` confirmará si PyTorch puede reconocer la GPU.

In [55]:
```
!nvidia-smi
```

```
zsh:1: command not found: nvidia-smi
```

In [56]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.cuda.is_available()
```

Out[56]:
```
False
```

1.1 El Conjunto de Datos

Para practicar la generación de imágenes, utilizaremos el conjunto de datos FashionMNIST (https://github.com/zalandoresearch/fashion-mnist). FashionMNIST está diseñado para ser un conjunto de datos "Hola Mundo" para problemas de clasificación de imágenes. El pequeño tamaño de las imágenes en blanco y negro (28 x 28 píxeles) también lo convierte en un excelente punto de partida para la generación de imágenes.

FashionMNIST está incluido en Torchvision (https://pytorch.org/vision/stable/index.html), una biblioteca de visión por computadora asociada con PyTorch. Cuando descargamos el conjunto de datos, podemos pasar una lista de transformaciones (https://pytorch.org/vision/stable/transforms.html) que nos gustaría aplicar a las imágenes. Por ahora, utilizaremos ToTensor (https://pytorch.org/vision/stable/generated/torchvision.transforms.ToTensor.html#torchvision.transforms.ToTensor) para convertir las imágenes en tensores para que podamos procesarlas con una red neuronal. Esto escalará automáticamente los valores de los píxeles de [0, 255] a [0, 1]. También reorganizará las dimensiones de [Alto x Ancho x Canales] a [Canales x Alto x Ancho].

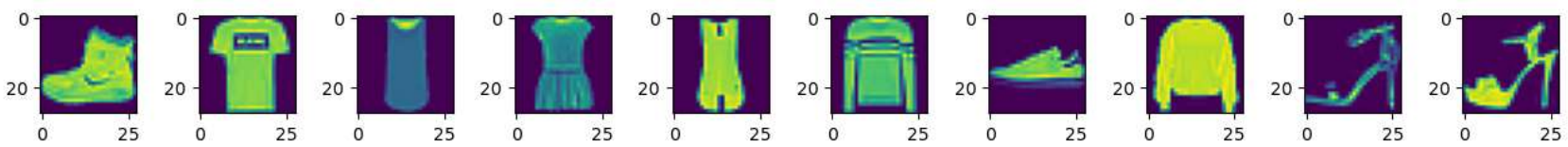
In [57]:
```
train_set = torchvision.datasets.FashionMNIST(
    "./data/", download=True, transform=transforms.Compose([transforms.ToTensor()])
)
NUM_CLASSES = 10
```

We can sample some of the images with the code below:

In [58]:
```
# Adjust for display; high w/h ratio recommended
plt.figure(figsize=(16, 1))

def show_images(dataset, num_samples=10):
    for i, img in enumerate(dataset):
        if i == num_samples:
            return
        plt.subplot(1, num_samples, i + 1)
        plt.imshow(torch.squeeze(img[0]))

show_images(train_set)
```



Vamos a configurar algunas constantes de importación para nuestro conjunto de datos. Con las U-Nets, es común reducir continuamente el tamaño del mapa de características mediante Max Pooling (https://paperswithcode.com/method/max-pooling). Luego, el tamaño del mapa de características se duplica con Convolución Transpuesta (https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html). Para mantener las dimensiones de la imagen consistentes a medida que bajamos y subimos por la U-Net, ayuda si el tamaño de la imagen es divisible por `2` múltiples veces.

En este ejemplo:

- `BATCH_SIZE` define el tamaño del lote para el entrenamiento y la evaluación.
- `IMAGE_SIZE` define el tamaño de las imágenes (en este caso, 28x28 píxeles).
- `transforms.Compose` se utiliza para aplicar varias transformaciones en las imágenes, asegurando que todas las imágenes tengan el mismo tamaño, convirtiéndolas a tensores y normalizando los valores de los píxeles.

Al configurar de esta manera, nos aseguramos de que las dimensiones de las imágenes sean consistentes, lo que facilitará el uso de operaciones como Max Pooling y Convolución Transpuesta en nuestra U-Net.

In [59]:
```
IMG_SIZE = 16 # Due to stride and pooling, must be divisible by 2 multiple times
IMG_CH = 1 # Black and white image, no color channels
BATCH_SIZE = 128
```

Ahora que hemos definido el tamaño objetivo de nuestras imágenes, vamos a crear una función para cargar los datos y transformarlos al tamaño objetivo. El ruido aleatorio que añadiremos a nuestras imágenes se muestreará de una distribución normal estándar (https://mathworld.wolfram.com/NormalDistribution.html), lo que significa que el 68% de los valores de los píxeles de ruido estarán entre -1 y 1. De manera similar, escalaremos los valores de nuestras imágenes para que estén entre -1 y 1.

Este también sería un buen lugar para aplicar aumentación de imágenes aleatoria. Por ahora, comenzaremos con un RandomHorizontalFlip (https://pytorch.org/vision/stable/generated/torchvision.transforms.RandomHorizontalFlip.html#torchvision.transforms.RandomHorizonta No usaremos un RandomVerticalFlip (https://pytorch.org/vision/stable/generated/torchvision.transforms.RandomVerticalFlip.html#torchvision.transforms.RandomVerticalFlip) porque terminaríamos generando imágenes al revés.

In [60]:
```
def load_fashionMNIST(data_transform, train=True):
    return torchvision.datasets.FashionMNIST(
        "./",
        download=True,
        train=train,
        transform=data_transform,
    )


def load_transformed_fashionMNIST():
    data_transforms = [
        transforms.Resize((IMG_SIZE, IMG_SIZE)),
        transforms.ToTensor(),  # Scales data into [0,1]
        transforms.RandomHorizontalFlip(),
        transforms.Lambda(lambda t: (t * 2) - 1)  # Scale between [-1, 1]
    ]

    data_transform = transforms.Compose(data_transforms)
    train_set = load_fashionMNIST(data_transform, train=True)
    test_set = load_fashionMNIST(data_transform, train=False)
    return torch.utils.data.ConcatDataset([train_set, test_set])
```

In [36]:
```
data = load_transformed_fashionMNIST()
dataloader = DataLoader(data, batch_size=BATCH_SIZE, shuffle=True, drop_last=True)
```

1.2 The U-Net Architecture

Primero, definamos los diferentes componentes de nuestra arquitectura U-Net. Principalmente, el `DownBlock` y el `UpBlock`.

1.2.1 El Bloque de Descenso

El `DownBlock` es una red neuronal convolucional típica. Si eres nuevo en PyTorch y vienes de un entorno de Keras/TensorFlow, lo siguiente es más similar a la API funcional (https://keras.io/guides/functional_api/) en lugar de un modelo secuencial (https://keras.io/guides/sequential_model/). Más adelante usaremos conexiones residuales (https://stats.stackexchange.com/questions/321054/what-are-residual-connections-in-rnns) y conexiones de salto. Un modelo secuencial no tiene la flexibilidad para soportar estos tipos de conexiones, pero un modelo funcional sí.

En nuestra función `__init__` a continuación, asignaremos nuestras diversas operaciones de red neuronal a variables de clase:

- Conv2d (https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html) aplica convolución a la entrada. El `in_ch` es el número de canales sobre los que estamos convolucionando y el `out_ch` es el número de canales de salida, que es el mismo que el número de filtros del kernel utilizados para la convolución. Típicamente en una arquitectura U-Net, el número de canales aumenta a medida que descendemos en el modelo.
- ReLU (https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html) es la función de activación para los kernels de convolución.
- BatchNorm2d (https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html) aplica normalización por lotes (https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338) a una capa de neuronas. ReLU no tiene parámetros aprendibles, por lo que podemos aplicar la misma función a múltiples capas y tendrá el mismo efecto que usar múltiples funciones ReLU. La normalización por lotes sí tiene parámetros aprendibles, y reutilizar esta función puede tener efectos inesperados.
- MaxPool2D (https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html) es lo que usaremos para reducir el tamaño de nuestro mapa de características a medida que se mueve hacia abajo en la red. Es posible lograr este efecto mediante convolución, pero la agrupación máxima se usa comúnmente con U-Nets.

En el método `forward`, describimos cómo nuestras diversas funciones deben aplicarse a una entrada. Hasta ahora, las operaciones son secuenciales en este orden:

- `Conv2d`
- `BatchNorm2d`
- `ReLU`
- `Conv2d`
- `BatchNorm2d`
- `ReLU`
- `MaxPool2d`

In [61]:

```python
import torch.nn as nn

class DownBlock(nn.Module):
    def __init__(self, in_ch, out_ch):
        # Definir parámetros del kernel
        kernel_size = 3
        stride = 1
        padding = 1

        # Inicializar la superclase nn.Module
        super().__init__()

        # Crear una lista de capas para el bloque de descenso
        layers = [
            # Primera capa convolucional
            nn.Conv2d(in_ch, out_ch, kernel_size, stride, padding),
            # Normalización por lotes para estabilizar y acelerar el aprendizaje
            nn.BatchNorm2d(out_ch),
            # Función de activación ReLU para introducir no linealidad
            nn.ReLU(),
            # Segunda capa convolucional
            nn.Conv2d(out_ch, out_ch, kernel_size, stride, padding),
            # Normalización por lotes
            nn.BatchNorm2d(out_ch),
            # Función de activación ReLU
            nn.ReLU(),
            # Capa de Max Pooling para reducir las dimensiones espaciales
            nn.MaxPool2d(2)
        ]
        # Definir las capas como un modelo secuencial
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        # Propagar la entrada a través del modelo secuencial
        return self.model(x)
```

1.2.2 El Bloque de Ascenso

Mientras que el `DownBlock` reduce el tamaño de nuestro mapa de características, el `UpBlock` lo duplica de nuevo. Esto se logra con ConvTranspose2d (https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html). Podemos usar casi la misma arquitectura que el `DownBlock` , pero reemplazaremos conv2 con convT. El `stride` de 2 de la convolución transpuesta causará la duplicación con la cantidad adecuada de `padding` .

Vamos a practicar con el bloque de código a continuación. Hemos configurado un ejemplo para probar esta función creando una imagen de prueba de `1` s.

In [62]:
```
ch, h, w = 1, 3, 3
x = torch.ones(1, ch, h, w)
x
```

Out[62]:
```
tensor([[[[1., 1., 1.],
          [1., 1., 1.],
          [1., 1., 1.]]]])
```

Podemos usar el `kernel` de identidad para ver cómo `conv_transpose2d` altera la imagen de entrada. El kernel de identidad tiene un único valor de `1` . Cuando se utiliza para convolucionar, la salida será la misma que la entrada.

Intenta cambiar el `stride` , el `padding` y el `output_padding` a continuación. ¿Los resultados coinciden con tus expectativas?

In [67]:

```
# Crear un tensor con un valor 1, que actúa como un kernel de identidad
kernel = torch.tensor([[1.]])  # Kernel de identidad

# Reformatear el kernel en un tensor 4D y repetirlo para que coincida con el número de canales
kernel = kernel.view(1, 1, 1, 1).repeat(1, ch, 1, 1)  # Convertir en un lote (batch)

# Aplicar la convolución transpuesta a la entrada 'x' utilizando el kernel de identidad
output = F.conv_transpose2d(x, kernel, stride=4, padding=0, output_padding=0)[0]

# Mostrar el resultado de la convolución transpuesta
output
```

Out[67]:

```
tensor([[[1., 0., 0., 0., 1., 0., 0., 0., 1.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [1., 0., 0., 0., 1., 0., 0., 0., 1.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [1., 0., 0., 0., 1., 0., 0., 0., 1.]]])
```

El tamaño del kernel también impacta el tamaño del mapa de características de salida. Intenta cambiar `kernel_size` a continuación. ¿Notas cómo la imagen de salida se expande a medida que aumenta el tamaño del kernel? Esto es lo opuesto a la convolución regular, donde un tamaño de kernel más grande disminuye el tamaño del mapa de características de salida.

In [69]:
```python
import torch
import torch.nn.functional as F

# Definir el kernel de identidad
kernel = torch.tensor([[1.]])  # Kernel de identidad
# Reorganizar el kernel en un tensor 4D y repetirlo para coincidir con el número de canales
kernel = kernel.view(1, 1, 1, 1).repeat(1, ch, 1, 1)  # Convertir en un lote

# Aplicar la convolución transpuesta a la entrada 'x' utilizando el kernel de identidad
output = F.conv_transpose2d(x, kernel, stride=1, padding=0, output_padding=0)[0]
# Mostrar el resultado de la convolución transpuesta
output
```

Out[69]:
```
tensor([[[1., 1., 1.],
         [1., 1., 1.],
         [1., 1., 1.]]])
```

Otra diferencia interesante: multiplicaremos el canal de entrada por 2. Esto es para acomodar las conexiones de salto. Vamos a concatenar la salida del `DownBlock` correspondiente de un `UpBlock` con la entrada del `UpBlock`.

Si x es el tamaño del mapa de características de entrada, el tamaño de salida es:

```
new_x = (x - 1) * stride + kernel_size - 2 * padding + out_padding
```

Si stride = 2 y out_padding = 1, entonces para duplicar el tamaño del mapa de características de entrada:

```
kernel_size = 2 * padding + 1
```

Las operaciones son casi las mismas que antes, pero con dos diferencias:

- `ConvTranspose2d` - Convolución transpuesta en lugar de Convolución
- `BatchNorm2d`
- `ReLU`
- `Conv2d`
- `BatchNorm2d`
- `ReLU`
- ~~`MaxPool2d`~~ - Escalar hacia arriba en lugar de hacia abajo

In [70]:

```python
import torch
import torch.nn as nn

class UpBlock(nn.Module):
    def __init__(self, in_ch, out_ch):
        # Variables de la convolución
        kernel_size = 3
        stride = 1
        padding = 1

        # Variables de la convolución transpuesta
        strideT = 2
        out_paddingT = 1

        # Inicialización de la superclase nn.Module
        super().__init__()

        # 2 * in_chs para la conexión de salto concatenada
        layers = [
            # Capa de convolución transpuesta
            nn.ConvTranspose2d(2 * in_ch, out_ch, kernel_size, strideT, padding, out_paddingT),
            # Normalización por lotes para estabilizar y acelerar el aprendizaje
            nn.BatchNorm2d(out_ch),
            # Función de activación ReLU para introducir no linealidad
            nn.ReLU(),
            # Capa de convolución
            nn.Conv2d(out_ch, out_ch, kernel_size, stride, padding),
            # Normalización por lotes
            nn.BatchNorm2d(out_ch),
            # Función de activación ReLU
            nn.ReLU()
        ]
        # Definir las capas como un modelo secuencial
        self.model = nn.Sequential(*layers)

    def forward(self, x, skip):
        # Concatenar la entrada con la salida del bloque de descenso correspondiente
        x = torch.cat((x, skip), 1)
        # Propagar la entrada a través del modelo secuencial
        x = self.model(x)
        return x
```

1.2.3 Una U-Net Completa

¡Finalmente es hora de unirlo todo! A continuación, tenemos nuestro modelo completo `UNet`.

En la función `__init__`, podemos definir el número de canales en cada paso de la U-Net con `down_chs`. El valor predeterminado actual es `(16, 32, 64)`, lo que significa que las dimensiones actuales de los datos a medida que se mueven a través del modelo son:

- input: 1 x 16 x 16
- down0: 16 x 16 x 16
  - down1: 32 x 8 x 8
    - down2: 64 x 4 x 4
      - dense_emb: 1024
    - up0: 64 x 4 x 4
  - up1: 64 x 8 x 8
- up2: 32 x 16 x 16
- out: 1 x 16 x 16

El método de clase `forward` es donde finalmente agregaremos nuestras conexiones de salto. Para cada paso hacia abajo en la U-Net, realizaremos un seguimiento de la salida de cada `DownBlock`. Luego, cuando nos movamos a través de los `UpBlock`s, concatenaremos (https://pytorch.org/docs/stable/generated/torch.cat.html) la salida del `UpBlock` anterior con su `DownBlock` correspondiente.

In [71]:

```python
import torch
import torch.nn as nn

class UNet(nn.Module):
    def __init__(self):
        super().__init__()
        img_ch = IMG_CH  # Número de canales de la imagen de entrada
        down_chs = (16, 32, 64)  # Canales de las capas de descenso
        up_chs = down_chs[::-1]  # Inversión de los canales de descenso para las capas de ascenso
        latent_image_size = IMG_SIZE // 4  # Tamaño de la imagen latente

        # Convolución inicial
        self.down0 = nn.Sequential(
            nn.Conv2d(img_ch, down_chs[0], 3, padding=1),
            nn.BatchNorm2d(down_chs[0]),
            nn.ReLU()
        )

        # Capas de descenso
        self.down1 = DownBlock(down_chs[0], down_chs[1])
        self.down2 = DownBlock(down_chs[1], down_chs[2])
        self.to_vec = nn.Sequential(nn.Flatten(), nn.ReLU())

        # 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()
        )

        # Capas de ascenso
        self.up0 = nn.Sequential(
            nn.Unflatten(1, (up_chs[0], latent_image_size, latent_image_size)),
            nn.Conv2d(up_chs[0], up_chs[0], 3, padding=1),
            nn.BatchNorm2d(up_chs[0]),
            nn.ReLU(),
        )
        self.up1 = UpBlock(up_chs[0], up_chs[1])
        self.up2 = UpBlock(up_chs[1], up_chs[2])
```

```
        # Ajustar los canales de salida
        self.out = nn.Sequential(
            nn.Conv2d(up_chs[-1], up_chs[-1], 3, 1, 1),
            nn.BatchNorm2d(up_chs[-1]),
            nn.ReLU(),
            nn.Conv2d(up_chs[-1], img_ch, 3, 1, 1),
        )

    def forward(self, x):
        # Aplicar las capas de descenso
        down0 = self.down0(x)
        down1 = self.down1(down0)
        down2 = self.down2(down1)
        latent_vec = self.to_vec(down2)

        # Aplicar las capas de ascenso con conexiones de salto
        up0 = self.up0(latent_vec)
        up1 = self.up1(up0, down2)
        up2 = self.up2(up1, down1)
        return self.out(up2)  # Retornar la salida final
```

In [72]:
```
model = UNet()
print("Num params: ", sum(p.numel() for p in model.parameters()))
```

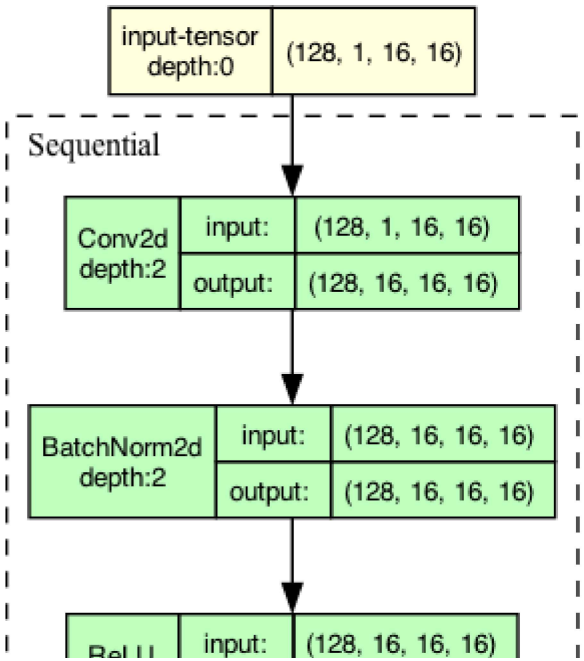
```
Num params:  234977
```

Verifiquemos la arquitectura del modelo con [torchview (https://github.com/mert-kurttutan/torchview)](https://github.com/mert-kurttutan/torchview). Si tenemos tres `down_chs`, debería haber dos `DownBlock`s, uno para cada transición. De manera similar, debería haber dos `UpBlock`s. También debemos comprobar que hay una conexión de salto. La "parte inferior" de la U-Net no necesita una conexión de salto, por lo que hay una conexión de salto para cada `UpBlock` menos uno.

Finalmente, ¿las dimensiones de salida son las mismas que las dimensiones de entrada?

In [73]:
```
graphviz.set_jupyter_format('png')
model_graph = draw_graph(
    model,
    input_size=(BATCH_SIZE, IMG_CH, IMG_SIZE, IMG_SIZE),
    device='meta',
    expand_nested=True
)
model_graph.resize_graph(scale=1.5)
model_graph.visual_graph
```

Out[73]:



Resumen del Modelo U-Net

Este modelo U-Net se compone de una serie de bloques de descenso y ascenso que reducen y luego restauran las dimensiones del mapa de características, utilizando conexiones de salto para preservar la información espacial. Aquí está el desglose de sus componentes:

Componentes Principales

Convolución Inicial

- La primera capa convolucional procesa la imagen de entrada y aumenta la cantidad de canales de la imagen.
- Se utiliza Batch Normalization y la activación ReLU para estabilizar el entrenamiento y añadir no linealidad.

Capas de Descenso (Down Blocks)

- **DownBlock 1**: Reduce las dimensiones espaciales de las características mientras aumenta el número de canales.
- **DownBlock 2**: Reduce aún más las dimensiones espaciales, incrementando los canales.
- **Conversión a Vector**: Aplana el mapa de características y aplica una activación ReLU.

Embeddings

- Tres capas totalmente conectadas (lineales) con activaciones ReLU para procesar la representación latente.

Capas de Ascenso (Up Blocks)

- **UpBlock 0**: Restaurar la forma del tensor aplastado y aplicar una convolución.
- **UpBlock 1**: Aumenta las dimensiones espaciales de las características utilizando una convolución transpuesta y se concatenan con la salida correspondiente del bloque de descenso.
- **UpBlock 2**: Restaura aún más las dimensiones espaciales, nuevamente utilizando una convolución transpuesta y concatenación.

Ajuste de Canales de Salida

- Dos capas convolucionales finales para ajustar los canales de salida al número de canales de entrada de la imagen original, con Batch Normalization y activación ReLU.

Método Forward

Fase de Descenso

- Se procesan las capas de descenso y se guarda la salida de cada `DownBlock`.
- La salida del último `DownBlock` se convierte en un vector latente.

Fase de Ascenso

- El vector latente se procesa y se restauran las dimensiones espaciales usando `UpBlocks`, concatenando con las salidas correspondientes de las capas de descenso.
- Se aplica la última capa convolucional para obtener la salida final del modelo.

In [74]:
```
model = torch.compile(UNet().to(device))
```

1.3 Training

Probemos agregar ruido a nuestras imágenes y ver si nuestro modelo U-Net puede filtrarlo. Podemos definir un parámetro `beta` para representar qué porcentaje de nuestra imagen es ruido frente a la imagen original. Podemos usar `alpha` para representar el complemento (https://brilliant.org/wiki/probability-by-complement/) de `beta`.

In [75]:
```
def add_noise(imgs):
    dev = imgs.device
    percent = .5 # Try changing from 0 to 1
    beta = torch.tensor(percent, device=dev)
    alpha = torch.tensor(1 - percent, device=dev)
    noise = torch.randn_like(imgs)
    return alpha * imgs + beta * noise
```

A continuación, definiremos nuestra función de pérdida como el Error Cuadrático Medio (MSE) (https://developers.google.com/machine-learning/glossary#mean-squared-error-mse) entre la imagen original y la imagen predicha.

In [76]:
```
def get_loss(model, imgs):
    imgs_noisy = add_noise(imgs)
    imgs_pred = model(imgs_noisy)
    return F.mse_loss(imgs, imgs_pred)
```

Para mostrar la salida de nuestro modelo, necesitaremos convertirla de nuevo al formato de una imagen en la CPU.

In [77]:
```
def show_tensor_image(image):
    reverse_transforms = transforms.Compose([
        transforms.Lambda(lambda t: (t + 1) / 2),
        transforms.Lambda(lambda t: torch.minimum(torch.tensor([1]), t)),
        transforms.Lambda(lambda t: torch.maximum(torch.tensor([0]), t)),
        transforms.ToPILImage(),
    ])
    plt.imshow(reverse_transforms(image[0].detach().cpu()))
```

Para ver la mejora durante el entrenamiento, podemos comparar la imagen `Original`, la imagen `Noise Added` y la imagen `Predicted Original` usando subplots (https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplot.html).

La anotación @torch.no_grad (https://pytorch.org/docs/stable/generated/torch.no_grad.html) evitará el cálculo del gradiente durante el uso de esta función.

In [78]:
```
@torch.no_grad()
def plot_sample(imgs):
    # Take first image of batch
    imgs = imgs[[0], :, :, :]
    imgs_noisy = add_noise(imgs[[0], :, :, :])
    imgs_pred = model(imgs_noisy)

    nrows = 1
    ncols = 3
    samples = {
        "Original" : imgs,
        "Noise Added" : imgs_noisy,
        "Predicted Original" : imgs_pred
    }
    for i, (title, img) in enumerate(samples.items()):
        ax = plt.subplot(nrows, ncols, i+1)
        ax.set_title(title)
        show_tensor_image(img)
    plt.show()
```

Finalmente, ¡es el momento de la verdad! Es hora de entrenar nuestro modelo y observar cómo mejora.

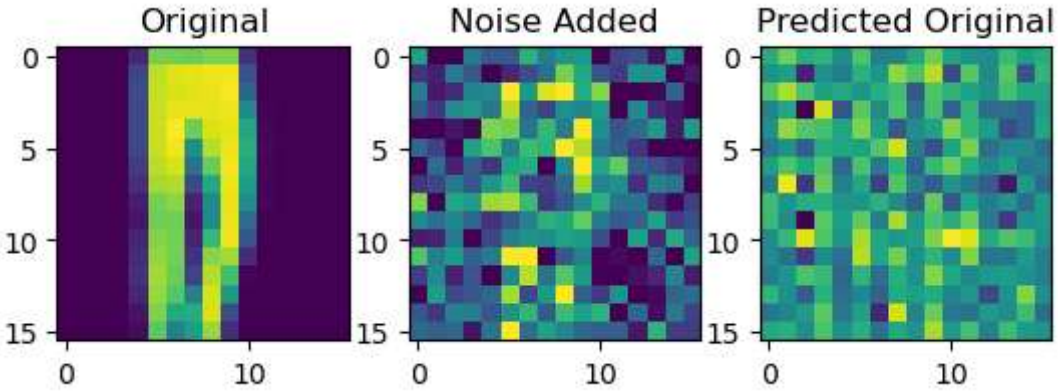
In [79]:
```
optimizer = Adam(model.parameters(), lr=0.0001)
epochs = 2

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

        images = batch[0].to(device)
        loss = get_loss(model, images)
        loss.backward()
        optimizer.step()

        if epoch % 1 == 0 and step % 100 == 0:
            print(f"Epoch {epoch} | Step {step:03d} Loss: {loss.item()} ")
            plot_sample(images)
```

```
Epoch 0 | Step 000 Loss: 0.6794945001602173
```

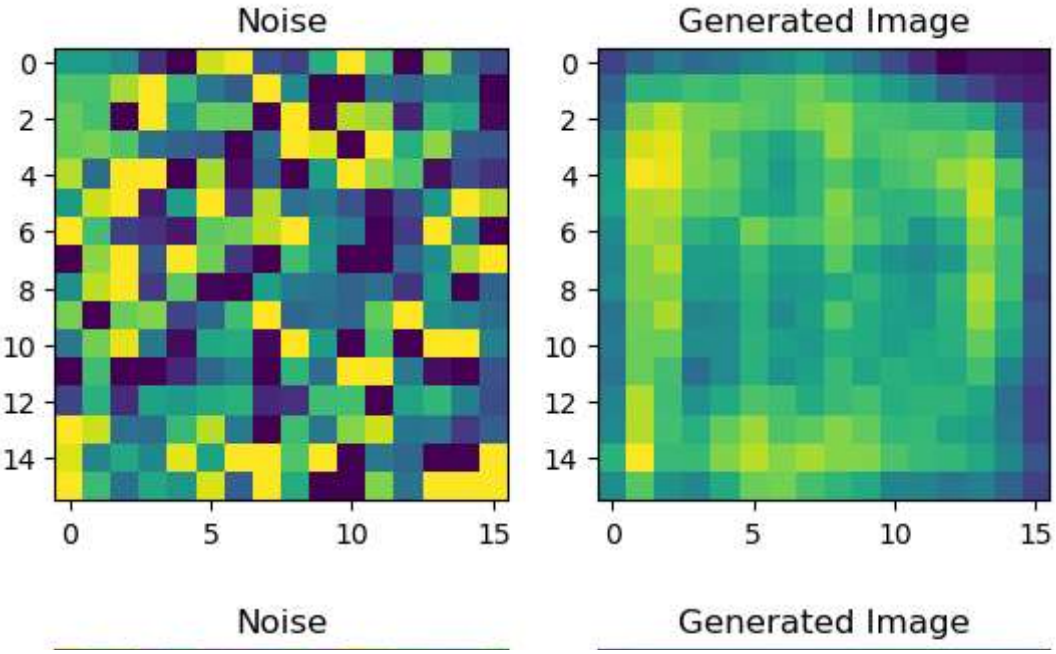



```
Epoch 0 | Step 100 Loss: 0.10460617393255234
```

Hay un poco de ruido en la imagen predicha, pero aún así hace un buen trabajo extrayendo la prenda original.

Ahora, ¿cómo se desempeña el modelo cuando se le da solo ruido? ¿Puede crear imágenes nuevas creíbles?

In [80]:
```
model.eval()
for _ in range(10):
    noise = torch.randn((1, IMG_CH, IMG_SIZE, IMG_SIZE), device=device)
    result = model(noise)
    nrows = 1
    ncols = 2
    samples = {
        "Noise" : noise,
        "Generated Image" : result
    }
    for i, (title, img) in enumerate(samples.items()):
        ax = plt.subplot(nrows, ncols, i+1)
        ax.set_title(title)
        show_tensor_image(img)
    plt.show()
```



Next

Hmm, estas imágenes se parecen más a manchas de tinta que a prendas de vestir. En el siguiente notebook, mejoraremos esta técnica para crear imágenes más reconocibles.

Antes de continuar, por favor reinicia el kernel de Jupyter ejecutando la celda de código a continuación. Esto evitará problemas de memoria en futuros notebooks.

Aprendimos cómo separar el ruido de una imagen usando una U-Net, pero no fue capaz de generar nuevas imágenes creíbles a partir del ruido. Los modelos de difusión son mucho mejores para generar imágenes desde cero.

La buena noticia es que nuestro modelo de red neuronal no cambiará mucho. Construiremos sobre la arquitectura U-Net con algunas modificaciones leves.

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

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