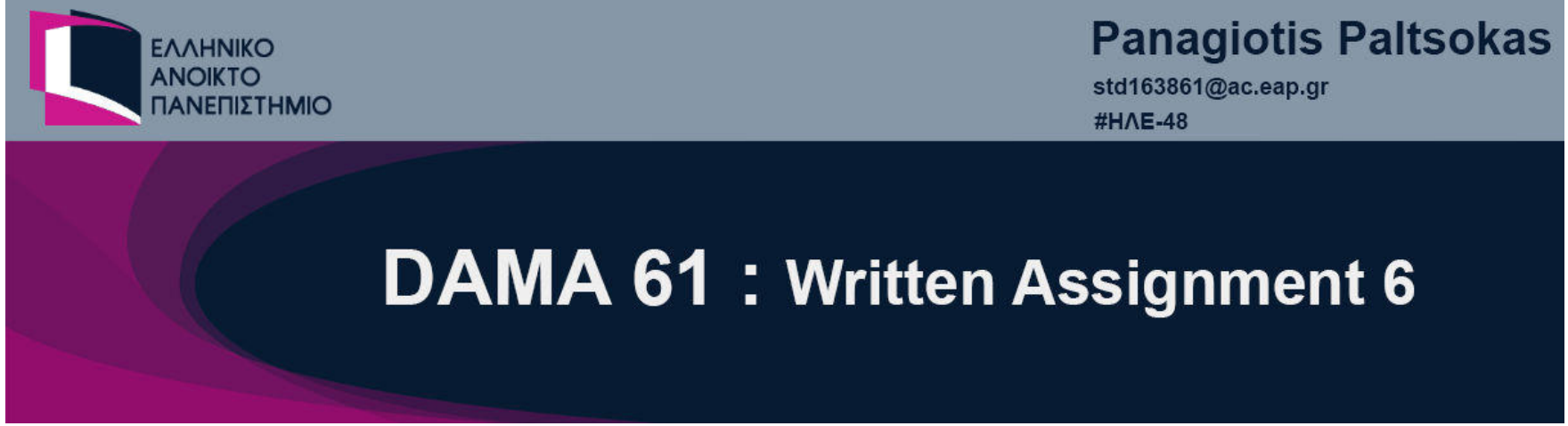


Problem 1

In this assignment, you will work with a filtered version of the Fashion MNIST dataset to explore the ability of autoencoders to compress and reconstruct image data. You will build and evaluate two autoencoder variants and compare their performance based on various experimental setups. Make sure to print any output you consider important to support your observations.

1.1

Load the Fashion MNIST dataset from the tensorflow.keras.datasets module. Keep only the instances that belong to the following three classes: 'Sandal', 'Sneaker', and 'Ankle boot'. Split the filtered dataset into a training set (5/7), a validation set (1/7), and a test set (1/7). Make sure the split uses stratified sampling to preserve class balance. Scale all subsets so pixel values are in the range [0, 1]. Print the class distribution of each subset with clear, labeled messages.

Load Fashion MNIST

In [1]:
```
from tensorflow.keras.datasets import fashion_mnist

(X_train_full, y_train_full), (X_test_full, y_test_full) = fashion_mnist.load_data()
```

Concatenate to extend the dataset and create custom splits.

In [2]:
```
import numpy as np
X_full = np.concatenate([X_train_full, X_test_full], axis=0)
y_full = np.concatenate([y_train_full, y_test_full], axis=0)
```

Keep only classes: Sandal (5), Sneaker (7), Ankle boot (9) (The mapping of class indices to labels is standardized and documented)

In [3]:
```
selected_classes = [5, 7, 9]
mask = np.isin(y_full, selected_classes)
X_full = X_full[mask]
y_full = y_full[mask]
```

Define the split sizes. Consider the full dataset split in 7 parts.

In [4]:
```
train_size = 5/7
valid_size = 1/7
test_size = 1/7
```

Export the test set by splitting on the full sets. We keep $\frac{1}{7} \cdot dataset$.

In [5]:
```
from sklearn.model_selection import train_test_split

X_temp, X_test, y_temp, y_test = train_test_split(X_full, y_full, test_size=test_size, random_state=42, stratify=y_full)
```

Having kept $\frac{1}{7} \cdot dataset$, we now want to export from the remaining $\frac{6}{7} \cdot dataset$, another $\frac{1}{7} \cdot dataset$. Therefore we will use a ratio of $\frac{\frac{1}{7}}{\frac{5}{7}+\frac{1}{7}} = \frac{\frac{1}{7}}{\frac{6}{7}} = \frac{1}{6} \cdot temp\_dataset$.

In [6]:
```
valid_ratio_adjusted = valid_size / (train_size + valid_size)
X_train, X_valid, y_train, y_valid = train_test_split(X_temp, y_temp, test_size=valid_ratio_adjusted, random_state=42, stratify=y_temp)
```

Normalize pixel values 0-255 to 0-1. Setting the datatype explicitly is important for TensorFlow/Keras models since they expect float inputs to perform computations efficiently.

In [7]:
```
X_train = X_train.astype("float32") / 255.
X_valid = X_valid.astype("float32") / 255.
X_test = X_test.astype("float32") / 255.
```

Print dataset shapes

In [8]:
```
print(f'Train set shape: {X_train.shape}, Labels shape: {y_train.shape}')
print(f'Validation set shape: {X_valid.shape}, Labels shape: {y_valid.shape}')
print(f'Test set shape: {X_test.shape}, Labels shape: {y_test.shape}')
```

```
Train set shape: (15000, 28, 28), Labels shape: (15000,)
Validation set shape: (3000, 28, 28), Labels shape: (3000,)
Test set shape: (3000, 28, 28), Labels shape: (3000,)
```

Verify that the stratified splitting preserved class balance

In [9]:
```
import pandas as pd

def print_class_distribution(y, name):
    class_counts = pd.Series(y).value_counts().sort_index()
    class_names = {5: 'Sandal', 7: 'Sneaker', 9: 'Ankle boot'}
    print(f'\nClass distribution in {name} set:')
    for label, count in class_counts.items():
```

```
        print(f'{class_names[label]:<12}: {count}')

print_class_distribution(y_train, "Train")
print_class_distribution(y_valid, "Validation")
print_class_distribution(y_test, "Test")
```

```
Class distribution in Train set:
Sandal      : 5000
Sneaker     : 5000
Ankle boot  : 5000

Class distribution in Validation set:
Sandal      : 1000
Sneaker     : 1000
Ankle boot  : 1000

Class distribution in Test set:
Sandal      : 1000
Sneaker     : 1000
Ankle boot  : 1000
```

1.2

Build a stacked autoencoder with the following architecture: an encoder and a decoder part of two fully-connected (Dense) layers each. Use ReLU activation for all hidden layers, and sigmoid activation for the output layer. Flatten the input before feeding it to the encoder, and reshape the output to match the original image shape (28×28). Set the latent space (codings) size to 50, and all other hidden layers to 256 units. Compile the model using: Nadam optimizer with learning rate 10^{-4}, adding binary cross-entropy as the loss function. Print the model summary of this autoencoder.

Define the constants

In [10]:
```
img_height, img_width = 28, 28
flattened_dim = img_height * img_width
hidden_units = 256
latent_dim = 50
```

Build the stacked autoencoder

In [11]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Flatten, Dense, Reshape

autoencoder = Sequential([
    Input(shape=(28, 28)),

    # Encoder
    Flatten(),
    Dense(hidden_units, activation='relu'),
    Dense(latent_dim, activation='relu'),
```

```
    # Decoder
    Dense(hidden_units, activation='relu'),
    Dense(flattened_dim, activation='sigmoid'),
    Reshape((28, 28))

])
```

Compile

In [12]:
```
from tensorflow.keras.optimizers import Nadam

autoencoder.compile(optimizer=Nadam(learning_rate=10**(-4)), loss='binary_crossentropy')

autoencoder.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 256)	200,960
dense_1 (Dense)	(None, 50)	12,850
dense_2 (Dense)	(None, 256)	13,056
dense_3 (Dense)	(None, 784)	201,488
reshape (Reshape)	(None, 28, 28)	0

Total params: 428,354 (1.63 MB)
Trainable params: 428,354 (1.63 MB)
Non-trainable params: 0 (0.00 B)

1.3

Now, build a second autoencoder that is identical in architecture to the one in Question 2, add l1 regularization to the latent space (the bottleneck layer), and use an activity regularizer with weight 10^{-6}. Print the model summary for this sparse autoencoder.

Define constants

In [13]:
```
img_height, img_width = 28, 28
flattened_dim = img_height * img_width
hidden_units = 256
```

```
latent_dim = 50
activity_reg_weight = 10**(-6)
```

Build the sparse autoencoder

In [14]:
```
from tensorflow.keras.regularizers import l1

sparse_autoencoder = Sequential([
    Input(shape=(28, 28)),

    # Encoder
    Flatten(),
    Dense(hidden_units, activation='relu'),
    Dense(latent_dim, activation='relu', activity_regularizer=l1(activity_reg_weight)),

    # Decoder
    Dense(hidden_units, activation='relu'),
    Dense(flattened_dim, activation='sigmoid'),
    Reshape((28, 28))
])
```

Compile

In [15]:
```
sparse_autoencoder.compile(optimizer=Nadam(learning_rate=10**(-4)), loss='binary_crossentropy')

sparse_autoencoder.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_4 (Dense)	(None, 256)	200,960
dense_5 (Dense)	(None, 50)	12,850
dense_6 (Dense)	(None, 256)	13,056
dense_7 (Dense)	(None, 784)	201,488
reshape_1 (Reshape)	(None, 28, 28)	0

Total params: 428,354 (1.63 MB)
Trainable params: 428,354 (1.63 MB)
Non-trainable params: 0 (0.00 B)

1.4

Train both autoencoders (from Q2 and Q3) for up to 50 epochs. Use early stopping with patience=5, and min_delta equal to 10^{-2}, monitoring the validation loss. Train each model with two different batch sizes: 32 and 256. This gives you four experiments in total (2 models × 2 batch sizes). After running your experiments, plot the training and validation loss curves for all of them. Add your comments on the differences observed between the models and batch sizes.

Set up early stopping

In [16]:
```
from tensorflow.keras.callbacks import EarlyStopping

early_stopping_cb = EarlyStopping(monitor='val_loss', patience=5, min_delta=1e-2, restore_best_weights=True, verbose=1)
```

Flatten inputs for dense autoencoders

In [17]:
```
X_train_flat = X_train.reshape(-1, 28, 28)
X_valid_flat = X_valid.reshape(-1, 28, 28)
```

Dictionaries to store models and histories

In [18]:
```
from tensorflow.keras.models import clone_model

models = {
    "regular_bs32": clone_model(autoencoder),
    "regular_bs256": clone_model(autoencoder),
    "sparse_bs32": clone_model(sparse_autoencoder),
    "sparse_bs256": clone_model(sparse_autoencoder),
}

histories = {}
```

Compile and train each model

In [19]:
```
for key, model in models.items():
    model.compile(optimizer=Nadam(learning_rate=1e-4), loss='binary_crossentropy')
    batch_size = 32 if "32" in key else 256
    print(f"\nTraining {key} with batch size {batch_size}...\n")

    history = model.fit(
        X_train_flat, X_train_flat,
        epochs=50,
        batch_size=batch_size,
        validation_data=(X_valid_flat, X_valid_flat),
        callbacks=[early_stopping_cb],
        verbose=2
    )

    histories[key] = history
```

```
Training regular_bs32 with batch size 32...

Epoch 1/50
469/469 - 2s - 5ms/step - loss: 0.3512 - val_loss: 0.2770
Epoch 2/50
469/469 - 1s - 2ms/step - loss: 0.2641 - val_loss: 0.2565
Epoch 3/50
469/469 - 1s - 2ms/step - loss: 0.2496 - val_loss: 0.2458
Epoch 4/50
469/469 - 1s - 2ms/step - loss: 0.2409 - val_loss: 0.2388
Epoch 5/50
469/469 - 1s - 2ms/step - loss: 0.2353 - val_loss: 0.2342
Epoch 6/50
469/469 - 1s - 2ms/step - loss: 0.2313 - val_loss: 0.2309
Epoch 7/50
469/469 - 1s - 2ms/step - loss: 0.2281 - val_loss: 0.2280
Epoch 8/50
469/469 - 1s - 2ms/step - loss: 0.2255 - val_loss: 0.2256
Epoch 9/50
469/469 - 1s - 2ms/step - loss: 0.2234 - val_loss: 0.2236
Epoch 10/50
469/469 - 1s - 2ms/step - loss: 0.2216 - val_loss: 0.2222
Epoch 11/50
469/469 - 1s - 2ms/step - loss: 0.2200 - val_loss: 0.2206
Epoch 12/50
469/469 - 1s - 2ms/step - loss: 0.2187 - val_loss: 0.2195
Epoch 13/50
469/469 - 1s - 3ms/step - loss: 0.2175 - val_loss: 0.2183
Epoch 14/50
469/469 - 1s - 3ms/step - loss: 0.2164 - val_loss: 0.2173
Epoch 14: early stopping
Restoring model weights from the end of the best epoch: 9.

Training regular_bs256 with batch size 256...

Epoch 1/50
59/59 - 1s - 25ms/step - loss: 0.6243 - val_loss: 0.4851
Epoch 2/50
59/59 - 0s - 6ms/step - loss: 0.3939 - val_loss: 0.3392
Epoch 3/50
59/59 - 0s - 6ms/step - loss: 0.3201 - val_loss: 0.3068
Epoch 4/50
59/59 - 0s - 6ms/step - loss: 0.2988 - val_loss: 0.2928
Epoch 5/50
59/59 - 0s - 6ms/step - loss: 0.2874 - val_loss: 0.2842
Epoch 6/50
59/59 - 0s - 6ms/step - loss: 0.2795 - val_loss: 0.2772
Epoch 7/50
59/59 - 0s - 6ms/step - loss: 0.2726 - val_loss: 0.2708
Epoch 8/50
59/59 - 0s - 7ms/step - loss: 0.2668 - val_loss: 0.2658
```

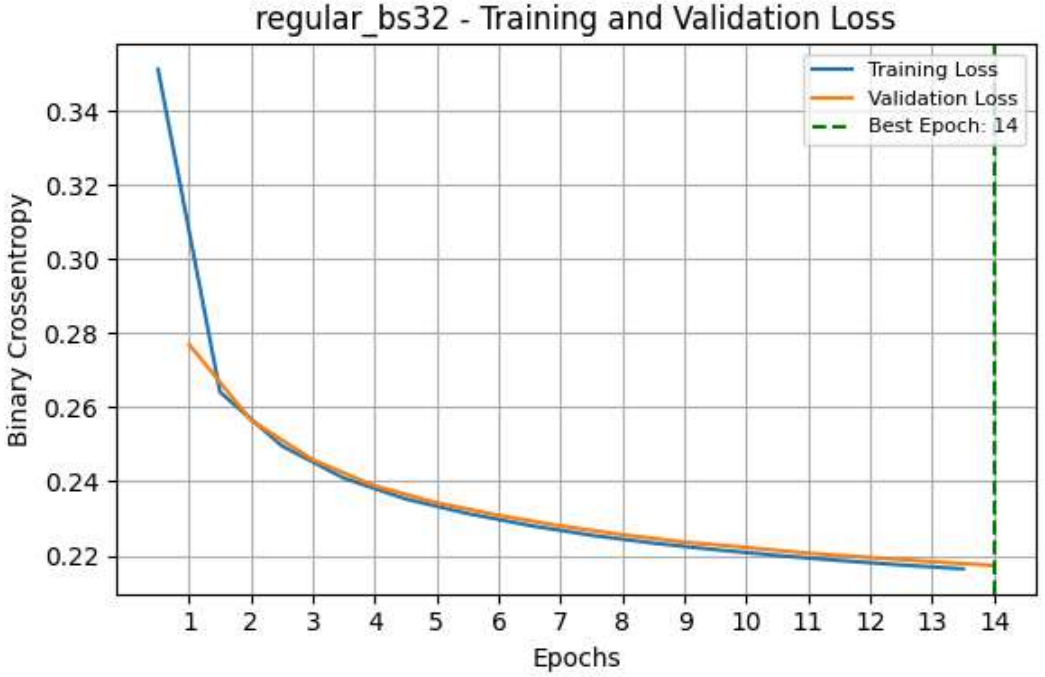
```
Epoch 9/50
59/59 - 0s - 6ms/step - loss: 0.2619 - val_loss: 0.2614
Epoch 10/50
59/59 - 0s - 6ms/step - loss: 0.2580 - val_loss: 0.2579
Epoch 11/50
59/59 - 0s - 6ms/step - loss: 0.2548 - val_loss: 0.2549
Epoch 12/50
59/59 - 0s - 6ms/step - loss: 0.2519 - val_loss: 0.2521
Epoch 13/50
59/59 - 0s - 6ms/step - loss: 0.2493 - val_loss: 0.2496
Epoch 14/50
59/59 - 0s - 6ms/step - loss: 0.2470 - val_loss: 0.2474
Epoch 15/50
59/59 - 0s - 5ms/step - loss: 0.2448 - val_loss: 0.2453
Epoch 16/50
59/59 - 0s - 8ms/step - loss: 0.2428 - val_loss: 0.2433
Epoch 17/50
59/59 - 0s - 8ms/step - loss: 0.2410 - val_loss: 0.2416
Epoch 18/50
59/59 - 0s - 8ms/step - loss: 0.2394 - val_loss: 0.2400
Epoch 19/50
59/59 - 0s - 8ms/step - loss: 0.2378 - val_loss: 0.2385
Epoch 20/50
59/59 - 0s - 7ms/step - loss: 0.2364 - val_loss: 0.2372
Epoch 21/50
59/59 - 0s - 7ms/step - loss: 0.2352 - val_loss: 0.2360
Epoch 21: early stopping
Restoring model weights from the end of the best epoch: 16.

Training sparse_bs32 with batch size 32...

Epoch 1/50
469/469 - 3s - 6ms/step - loss: 0.3549 - val_loss: 0.2816
Epoch 2/50
469/469 - 1s - 3ms/step - loss: 0.2678 - val_loss: 0.2599
Epoch 3/50
469/469 - 1s - 2ms/step - loss: 0.2530 - val_loss: 0.2493
Epoch 4/50
469/469 - 1s - 2ms/step - loss: 0.2443 - val_loss: 0.2421
Epoch 5/50
469/469 - 1s - 2ms/step - loss: 0.2381 - val_loss: 0.2368
Epoch 6/50
469/469 - 1s - 2ms/step - loss: 0.2336 - val_loss: 0.2330
Epoch 7/50
469/469 - 1s - 2ms/step - loss: 0.2302 - val_loss: 0.2300
Epoch 8/50
469/469 - 1s - 2ms/step - loss: 0.2275 - val_loss: 0.2277
Epoch 9/50
469/469 - 1s - 2ms/step - loss: 0.2253 - val_loss: 0.2256
Epoch 10/50
469/469 - 1s - 2ms/step - loss: 0.2235 - val_loss: 0.2240
```

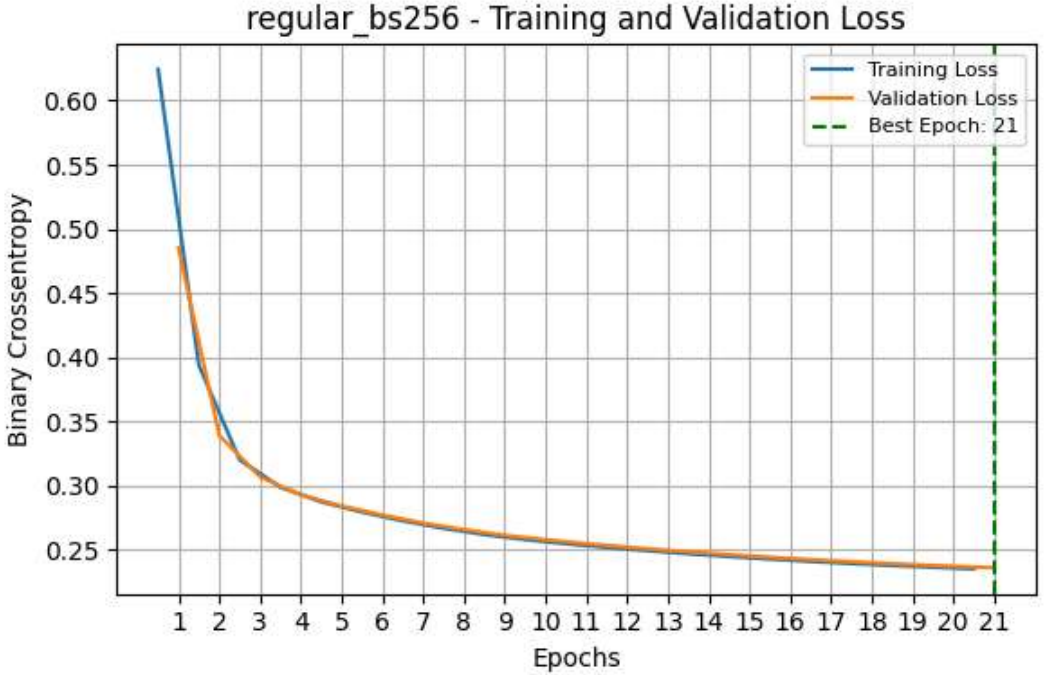
```
Epoch 11/50
469/469 - 1s - 2ms/step - loss: 0.2220 - val_loss: 0.2227
Epoch 12/50
469/469 - 1s - 2ms/step - loss: 0.2207 - val_loss: 0.2215
Epoch 13/50
469/469 - 1s - 2ms/step - loss: 0.2196 - val_loss: 0.2205
Epoch 14/50
469/469 - 1s - 2ms/step - loss: 0.2186 - val_loss: 0.2197
Epoch 14: early stopping
Restoring model weights from the end of the best epoch: 9.

Training sparse_bs256 with batch size 256...

Epoch 1/50
59/59 - 1s - 23ms/step - loss: 0.6339 - val_loss: 0.5011
Epoch 2/50
59/59 - 0s - 6ms/step - loss: 0.4169 - val_loss: 0.3679
Epoch 3/50
59/59 - 0s - 6ms/step - loss: 0.3487 - val_loss: 0.3344
Epoch 4/50
59/59 - 0s - 6ms/step - loss: 0.3256 - val_loss: 0.3175
Epoch 5/50
59/59 - 0s - 6ms/step - loss: 0.3100 - val_loss: 0.3038
Epoch 6/50
59/59 - 0s - 6ms/step - loss: 0.2976 - val_loss: 0.2936
Epoch 7/50
59/59 - 0s - 6ms/step - loss: 0.2885 - val_loss: 0.2861
Epoch 8/50
59/59 - 0s - 6ms/step - loss: 0.2819 - val_loss: 0.2804
Epoch 9/50
59/59 - 0s - 6ms/step - loss: 0.2767 - val_loss: 0.2759
Epoch 10/50
59/59 - 0s - 6ms/step - loss: 0.2724 - val_loss: 0.2718
Epoch 11/50
59/59 - 0s - 6ms/step - loss: 0.2685 - val_loss: 0.2680
Epoch 12/50
59/59 - 0s - 7ms/step - loss: 0.2649 - val_loss: 0.2645
Epoch 13/50
59/59 - 0s - 6ms/step - loss: 0.2617 - val_loss: 0.2615
Epoch 14/50
59/59 - 0s - 7ms/step - loss: 0.2588 - val_loss: 0.2588
Epoch 15/50
59/59 - 0s - 6ms/step - loss: 0.2563 - val_loss: 0.2565
Epoch 16/50
59/59 - 0s - 7ms/step - loss: 0.2541 - val_loss: 0.2546
Epoch 17/50
59/59 - 0s - 7ms/step - loss: 0.2522 - val_loss: 0.2525
Epoch 18/50
59/59 - 0s - 6ms/step - loss: 0.2504 - val_loss: 0.2509
Epoch 19/50
59/59 - 0s - 6ms/step - loss: 0.2489 - val_loss: 0.2495
```

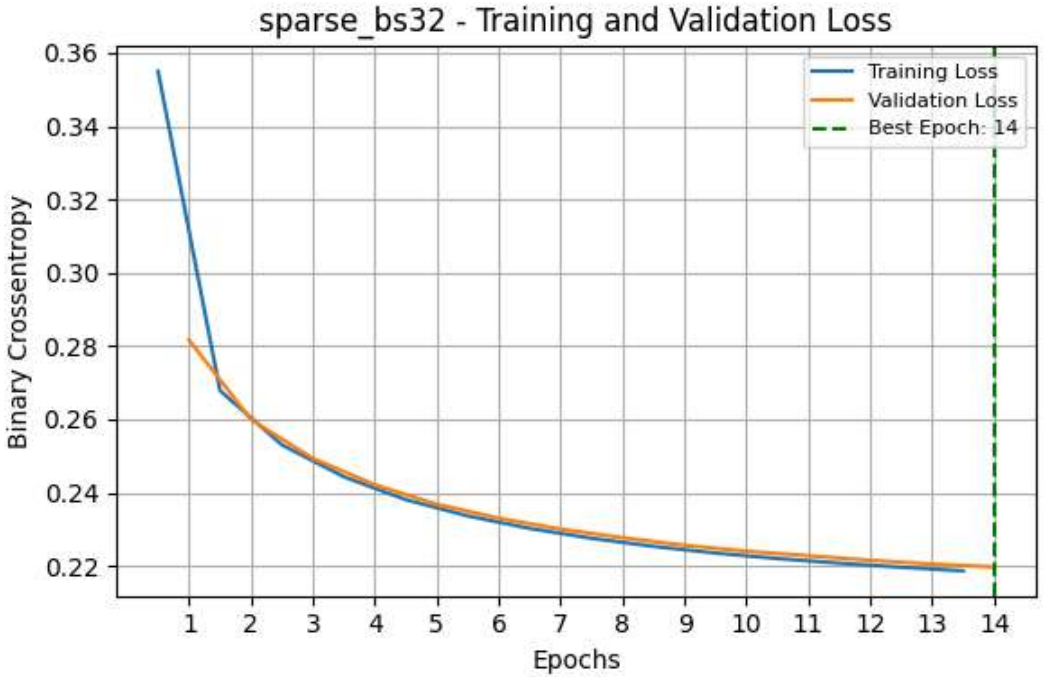
```
Epoch 20/50
59/59 - 0s - 6ms/step - loss: 0.2475 - val_loss: 0.2480
Epoch 20: early stopping
Restoring model weights from the end of the best epoch: 15.
```

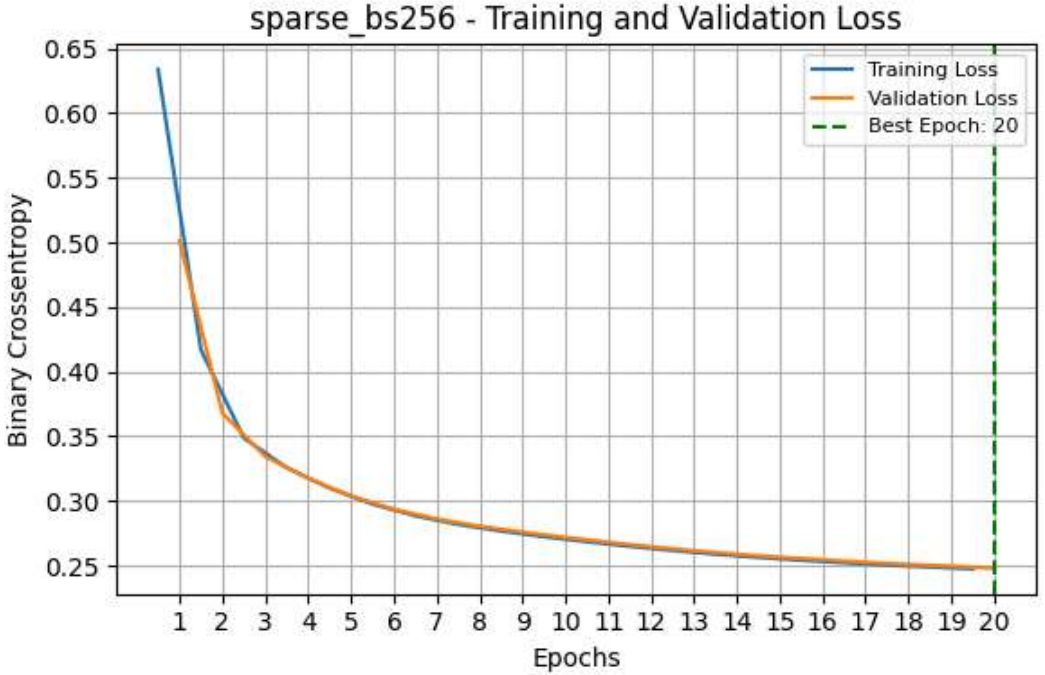
Plot the training and validation losses for all the autoencoders

In [20]:
```python
import matplotlib.pyplot as plt

for key, history in histories.items():
    epochs = np.arange(len(history.epoch))
    best_epoch = np.argmin(history.history['val_loss'])

    plt.figure(figsize=(6, 4))

    # Plot training and validation loss only
    plt.plot(epochs - 0.5, history.history['loss'], label='Training Loss')
    plt.plot(epochs, history.history['val_loss'], label='Validation Loss')
    plt.axvline(best_epoch, color='green', linestyle='--', label=f'Best Epoch: {best_epoch + 1}')
    plt.title(f'{key} - Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.xticks(ticks=epochs, labels=epochs + 1)
    plt.ylabel('Binary Crossentropy')
    plt.grid()
    plt.legend(loc='upper right', fontsize=8)

    plt.tight_layout()
    plt.show()
```


regular_bs32 - Training and Validation Loss
Training Loss
Validation Loss
Best Epoch: 14
Binary Crossentropy
0.34
0.32
0.30
0.28
0.26
0.24
0.22
1 2 3 4 5 6 7 8 9 10 11 12 13 14
Epochs


regular_bs256 - Training and Validation Loss
Training Loss
Validation Loss
Best Epoch: 21
Binary Crossentropy
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
Epochs





1.5

Write a function that compares original images with their reconstructions from an autoencoder. The function should:

- Accept a model, a set of images, and the number of images to display as input.
- Display a grid where the top row shows the original images, and the bottom row shows the reconstructed images.
- Ensure the pixel values are clipped to the range [0, 1] for proper visualization.

Then use this function to display the first 15 test images and their reconstructions from the regular autoencoder trained with a batch size of 256, and the sparse autoencoder trained with a batch size 256.

In [21]:
```
def show_reconstructions(model, images, n_images=15):
    """
    Plots original and reconstructed images in a 2-row grid.

    Top row: original images
    Bottom row: reconstructed images
    """
    reconstructions = model.predict(images[:n_images], verbose=0)
    reconstructions = np.clip(reconstructions, 0, 1)  # Ensure proper range

    plt.figure(figsize=(n_images, 3))
    for i in range(n_images):
        # Original
        plt.subplot(2, n_images, i + 1)
        plt.imshow(images[i], cmap='gray')
        plt.axis('off')

        # Reconstructed
        plt.subplot(2, n_images, i + 1 + n_images)
        plt.imshow(reconstructions[i], cmap='gray')
        plt.axis('off')

    plt.suptitle("Top: Original Images — Bottom: Reconstructions", fontsize=12)
    plt.tight_layout()
    plt.show()
```

Flatten test images for model input

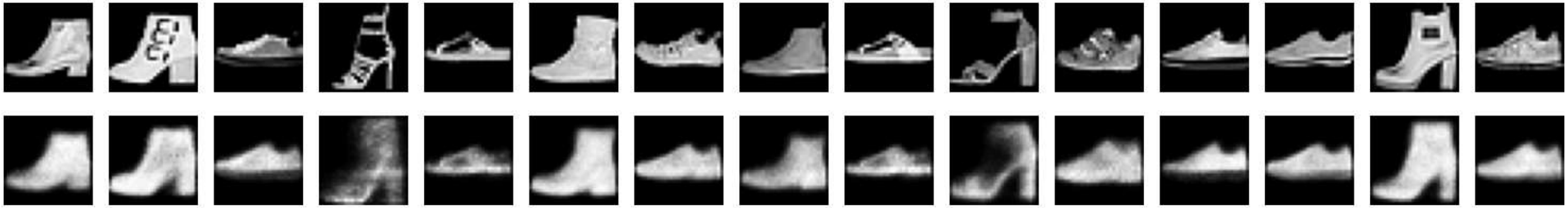
In [22]:
```
X_test_flat = X_test.reshape(-1, 28, 28)
```

Display reconstructions for regular autoencoder (batch size 256)

In [23]:
```
print("Regular Autoencoder (batch size 256):")
show_reconstructions(models["regular_bs256"], X_test_flat, n_images=15)
```

```
Regular Autoencoder (batch size 256):
```

Top: Original Images — Bottom: Reconstructions

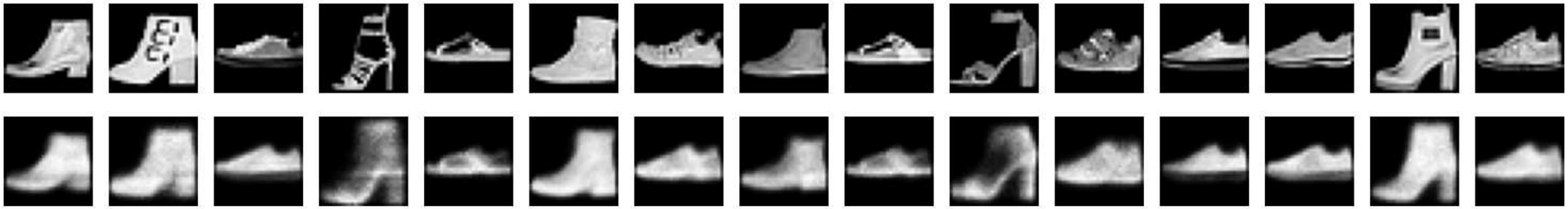


Display reconstructions for sparse autoencoder (batch size 256)

In [24]:
```
print("Sparse Autoencoder (batch size 256):")
show_reconstructions(models["sparse_bs256"], X_test_flat, n_images=15)
```

```
Sparse Autoencoder (batch size 256):
```

Top: Original Images — Bottom: Reconstructions



Problem 2

In this assignment, you will implement and train a Generative Adversarial Network (GAN) from scratch using the Fashion-MNIST dataset. Your GAN will learn to generate synthetic fashion item images that resemble real samples from selected categories. Hint: Use tensorflow ~ 2.14 and follow the code of the book.

In [19]:
```
import tensorflow as tf
print(tf.__version__)
```

```
2.18.0
```

2.1

Load the Fashion-MNIST dataset through tensorflow.keras.datasets, keep the data on the training set, select and retain only the T-shirt/top, Trousers, and Pullover classes, and scale the pixel values to the range [-1, 1].

Load Fashion-MNIST (only use training set for GANs)

In [20]:
```
from tensorflow.keras.datasets import fashion_mnist

(X_train_full, y_train_full), _ = fashion_mnist.load_data()
```

Keep only classes: 0 = T-shirt/top, 1 = Trouser, 2 = Pullovep

In [21]:
```
import numpy as np

selected_classes = [0, 1, 2]
mask = np.isin(y_train_full, selected_classes)

X_train = X_train_full[mask]
y_train = y_train_full[mask]
```

Reshape and scale images to [-1, 1]

In [22]:
```
X_train = X_train.astype("float32") / 127.5 - 1.0  # [0,255] -> [-1,1]
X_train = np.expand_dims(X_train, axis=-1)        # shape: (n, 28, 28, 1)

print(f'Training set shape: {X_train.shape}')
for class_label in selected_classes:
    count = np.sum(y_train == class_label)
    print(f'Class {class_label}: {count} samples')
```

```
Training set shape: (18000, 28, 28, 1)
Class 0: 6000 samples
Class 1: 6000 samples
Class 2: 6000 samples
```

2.2

Create a generator that takes input random noise vectors of size 30. It has two hidden layers, each with 100 and 150 units, with an activation function ReLU and a He normal kernel initializer. It outputs a fully connected layer of a 784-dimensional vector with activation tanh, which is reshaped to a 28 by 28 pixels image.

Define latent vector size

In [23]:
```
codings_size = 30
```

Generator model

In [24]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense, Reshape
from tensorflow.keras.initializers import HeNormal

generator = Sequential([
    Input(shape=(codings_size,)),
    Dense(100, activation='relu', kernel_initializer=HeNormal()),
    Dense(150, activation='relu', kernel_initializer=HeNormal()),
    Dense(784, activation='tanh'),
    Reshape((28, 28))
])

generator.summary()
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
dense_6 (Dense)	(None, 100)	3,100
dense_7 (Dense)	(None, 150)	15,150
dense_8 (Dense)	(None, 784)	118,384
reshape_1 (Reshape)	(None, 28, 28)	0

Total params: 136,634 (533.73 KB)
Trainable params: 136,634 (533.73 KB)
Non-trainable params: 0 (0.00 B)

2.3

Create a Discriminator that reads a 28 by 28 pixel image and classifies it as “real” or “fake”. The Discriminator consists of two hidden layers, each with 150 and 100 units, with an activation function ReLU and a He normal kernel initializer. A sigmoid function activates the output layer.

Discriminator model

In [25]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Flatten, Dense
from tensorflow.keras.initializers import HeNormal

discriminator = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Dense(150, activation='relu', kernel_initializer=HeNormal()),
    Dense(100, activation='relu', kernel_initializer=HeNormal()),
```

```
    Dense(1, activation='sigmoid')
])

discriminator.summary()
```

Model: "sequential_4"

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_9 (Dense)	(None, 150)	117,750
dense_10 (Dense)	(None, 100)	15,100
dense_11 (Dense)	(None, 1)	101

Total params: 132,951 (519.34 KB)
Trainable params: 132,951 (519.34 KB)
Non-trainable params: 0 (0.00 B)

2.4

Connect the generator and discriminator to set up a GAN pipeline. Use the Binary Cross Entropy as the loss function and the RMSProp optimizer for both the discriminator and the GAN models. Create batches of 32 images by slicing the training set. Train the GAN for 10 epochs, alternating between updating the Discriminator and Generator models on each batch. After each epoch of training ends, visualize 32 generated images.

Set seed for reproducibility

In [26]:
```
import tensorflow as tf

np.random.seed(42)
tf.random.set_seed(42)
```

Compile discriminator and GAN

In [27]:
```
gan = Sequential([generator, discriminator])
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop", metrics=["accuracy"]) #added accuracy metrics for extra insight
discriminator.trainable = False
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

Prepare the dataset

In [28]:
```
from tensorflow.data import Dataset

batch_size = 32
```

```
dataset = Dataset.from_tensor_slices(X_train).shuffle(buffer_size=1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
```

Define the plotting function

In [29]:
```
import matplotlib.pyplot as plt

def plot_multiple_images(images, n_cols=8):
    '''Displays a batch of grayscale images in a grid layout.

    Parameters:
    ----------
    images : tf.Tensor or np.ndarray
        A batch of generated images with pixel values in the range [-1, 1].
        Each image should have shape (28, 28).
    n_cols : int
        Number of images to display per row in the grid layout.

    Behavior:
    ---------
    - Automatically rescales the pixel values from [-1, 1] to [0, 1] for display.
    - Calculates the number of rows based on the number of images and columns.
    - Uses matplotlib to render a grid of images with axes turned off.

    Notes:
    ------
    This function is typically called after each GAN training epoch to visualize
    generated images and monitor the quality of outputs over time.

    '''
    images = (images + 1) / 2  # Rescale from [-1, 1] to [0, 1]
    n_images = len(images)
    n_rows = n_images // n_cols
    plt.figure(figsize=(n_cols, n_rows))
    for index, image in enumerate(images):
        plt.subplot(n_rows, n_cols, index + 1)
        plt.imshow(image.numpy(), cmap="gray")
        plt.axis("off")
    plt.tight_layout()
    plt.show()
```

Train the GAN

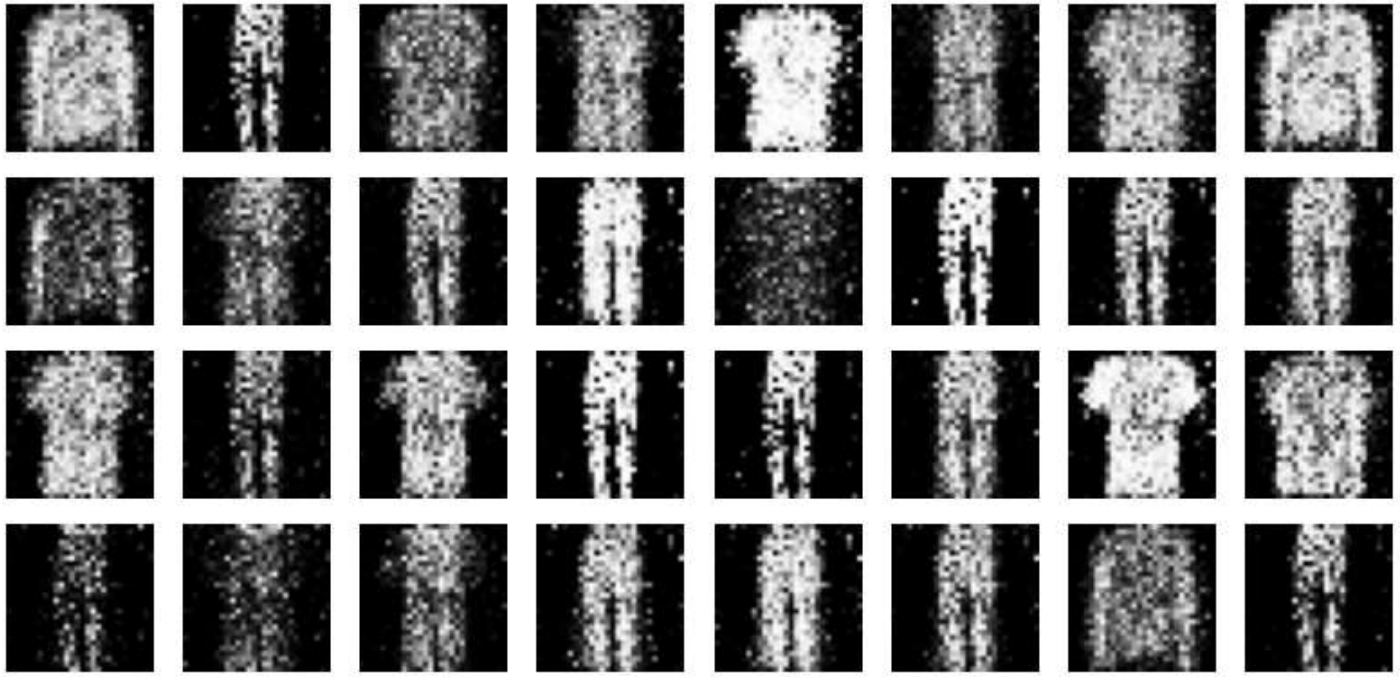
In [30]:
```
import time
import matplotlib.pyplot as plt

def train_gan(gan, dataset, batch_size, codings_size, n_epochs):
    '''
    Trains a Generative Adversarial Network using a custom training loop.
```

```
 This function alternates between training the discriminator and the generator
 on batches of real and synthetic images. After each epoch, it visualizes a grid
 of generated images.

 Parameters:
 ----------
 gan : keras.Sequential
     A compiled Sequential model that chains the generator and the discriminator.
dataset : tf.data.Dataset
     A dataset of real training images.
     The images are expected to be scaled to the range [-1, 1] and have shape (28, 28, 1).
 batch_size : int
     The number of samples per training batch.
 codings_size : int
     Dimensionality of the random noise vector fed to the generator.
 n_epochs : int
     Number of full passes over the dataset.

 Training Logic:
 ---------------
 For each batch in each epoch:
     - Phase 1: Train the discriminator to distinguish real vs. fake images.
     - Phase 2: Train the generator, via the combined GAN model, to produce
       images that can trick the discriminator.

 Notes:
 ------
 - Before each `train_on_batch()` call, `discriminator.trainable` is toggled:
     - It is set to `True` when training the discriminator.
     - It is set to `False` when training the generator via the GAN model.
 - I decided it would be better not to work in an older Tensorflow version,
   and this manual toggling is necessary in TensorFlow 2.18+ because `trainable=False`
   does not always work automatically inside nested models, unless explicitly updated at runtime.
 - At the end of each epoch, 32 generated images are visualized using a grid.

 '''
 generator, discriminator = gan.layers
 start_time = time.time()

 for epoch in range(n_epochs):
     print(f"\nEpoch {epoch + 1}/{n_epochs}")
     epoch_start = time.time()

     for step, X_batch in enumerate(dataset):
         X_batch = tf.squeeze(X_batch, axis=-1)

         # Phase 1 - Train the Discriminator
         noise = tf.random.normal(shape=[batch_size, codings_size])
         generated_images = generator(noise)
         X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)
         y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
```

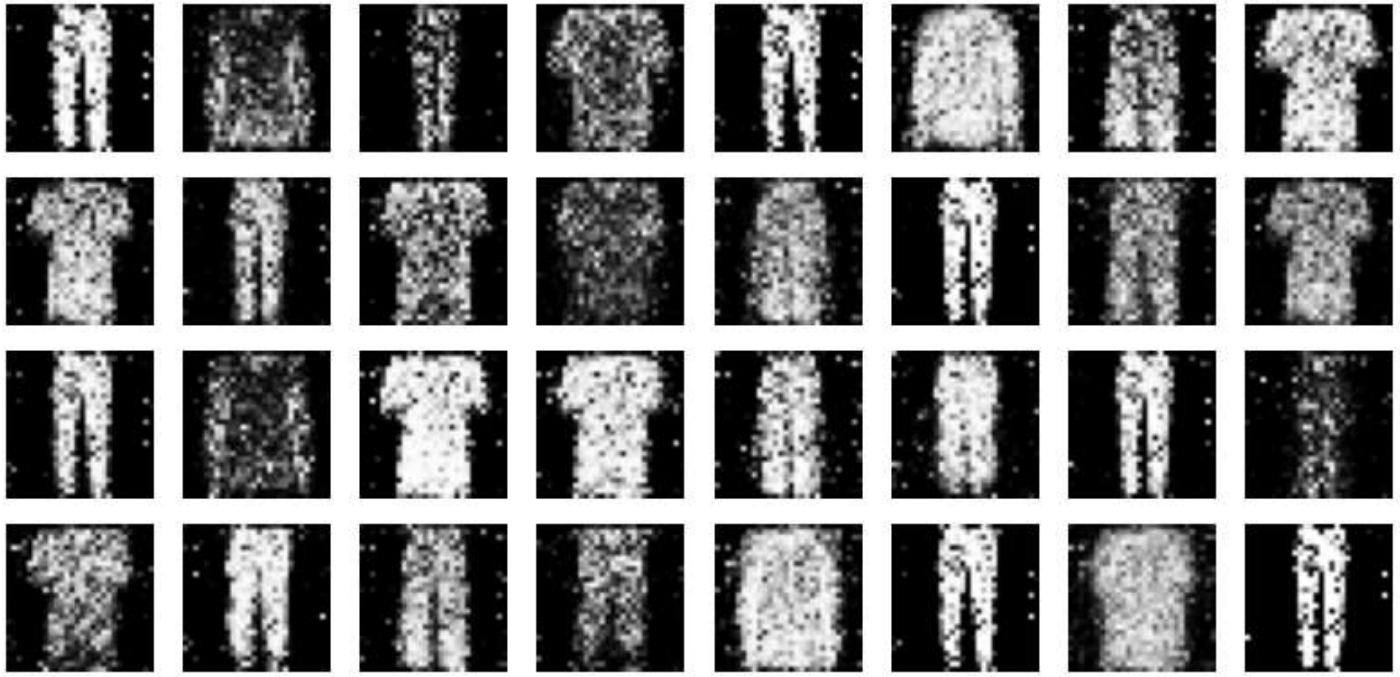
```
            discriminator.trainable = True
            d_loss, d_acc = discriminator.train_on_batch(X_fake_and_real, y1)

            # Phase 2 - Train the Generator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            y2 = tf.constant([[1.]] * batch_size)

            discriminator.trainable = False
            g_loss = gan.train_on_batch(noise, y2)

            if step % 100 == 0:
                print(f"  Step {step} - d_loss: {d_loss:.4f}, d_acc: {d_acc:.4f}, g_loss: {g_loss:.4f}")

        epoch_duration = time.time() - epoch_start
        print(f"Epoch {epoch + 1} duration: {epoch_duration:.2f} seconds")

        # Visualization
        print(f"Visualizing generated images after epoch {epoch + 1}")
        noise = tf.random.normal(shape=[32, codings_size])
        generated_images = generator(noise)
        plot_multiple_images(generated_images, n_cols=8)

    total_time = time.time() - start_time
    print(f"\nTotal training time: {total_time:.2f} seconds")
```

In [31]:
```
train_gan(gan, dataset, batch_size, codings_size, n_epochs=10)
```

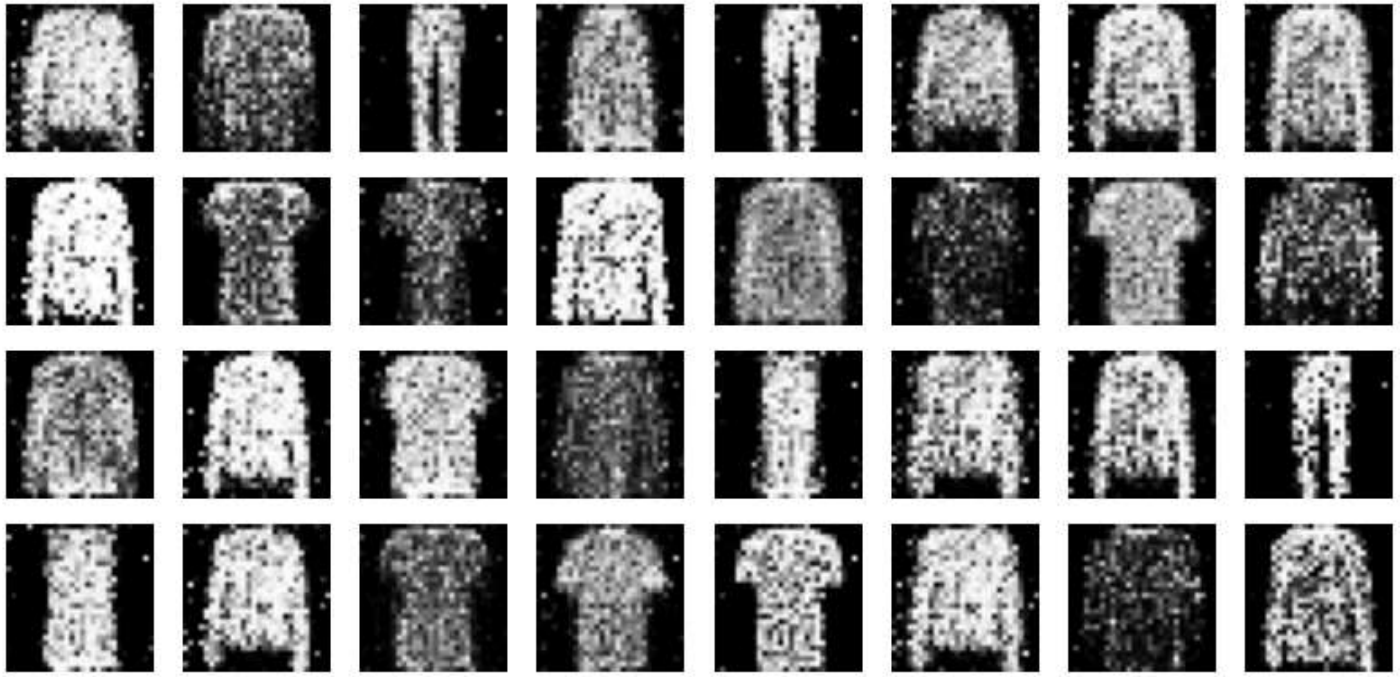
```
Epoch 1/10
  Step 0 - d_loss: 0.7705, d_acc: 0.5312, g_loss: 2.3286
  Step 100 - d_loss: 0.4640, d_acc: 0.7676, g_loss: 1.6619
  Step 200 - d_loss: 0.4860, d_acc: 0.7632, g_loss: 1.4847
  Step 300 - d_loss: 0.5143, d_acc: 0.7441, g_loss: 1.3956
  Step 400 - d_loss: 0.5436, d_acc: 0.7197, g_loss: 1.3280
  Step 500 - d_loss: 0.5600, d_acc: 0.7021, g_loss: 1.2830
Epoch 1 duration: 4.31 seconds
Visualizing generated images after epoch 1
```



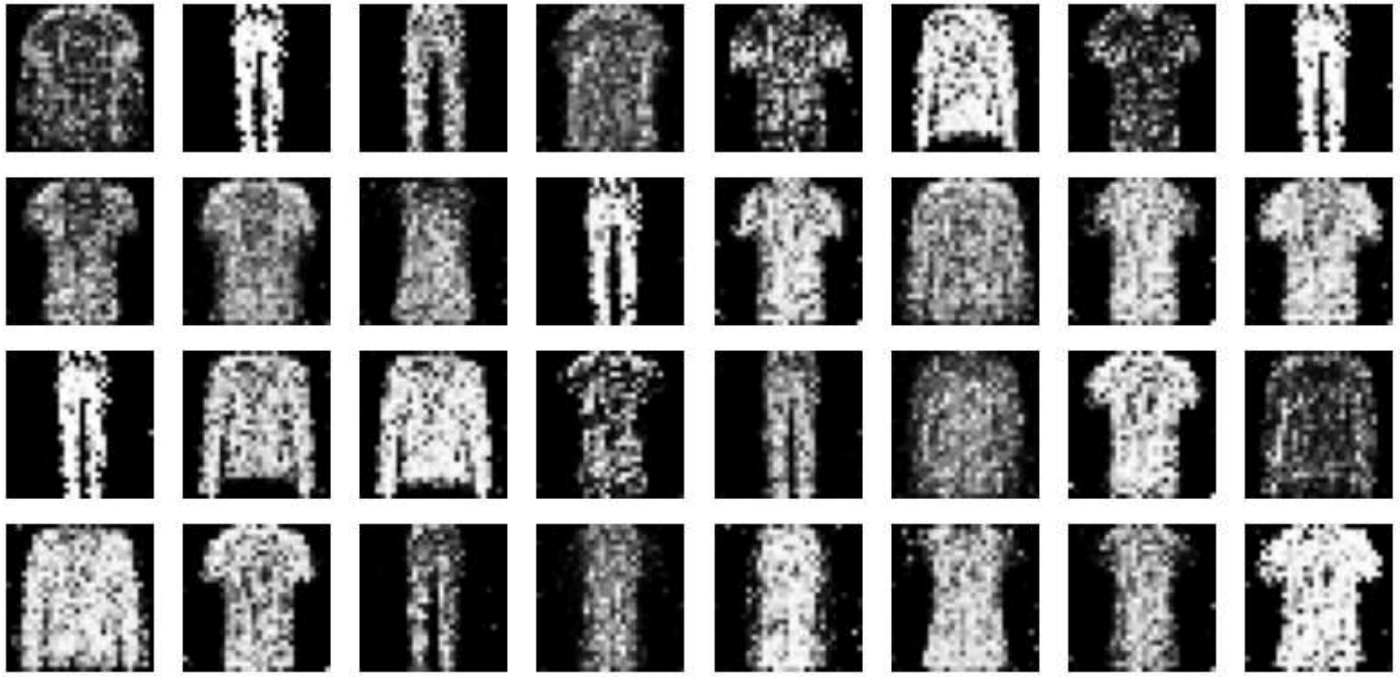
```
Epoch 2/10
  Step 0 - d_loss: 0.5697, d_acc: 0.6938, g_loss: 1.2564
  Step 100 - d_loss: 0.5786, d_acc: 0.6859, g_loss: 1.2278
  Step 200 - d_loss: 0.5848, d_acc: 0.6805, g_loss: 1.2125
  Step 300 - d_loss: 0.5897, d_acc: 0.6772, g_loss: 1.1986
  Step 400 - d_loss: 0.5967, d_acc: 0.6710, g_loss: 1.1802
  Step 500 - d_loss: 0.6023, d_acc: 0.6650, g_loss: 1.1636
Epoch 2 duration: 3.63 seconds
Visualizing generated images after epoch 2
```



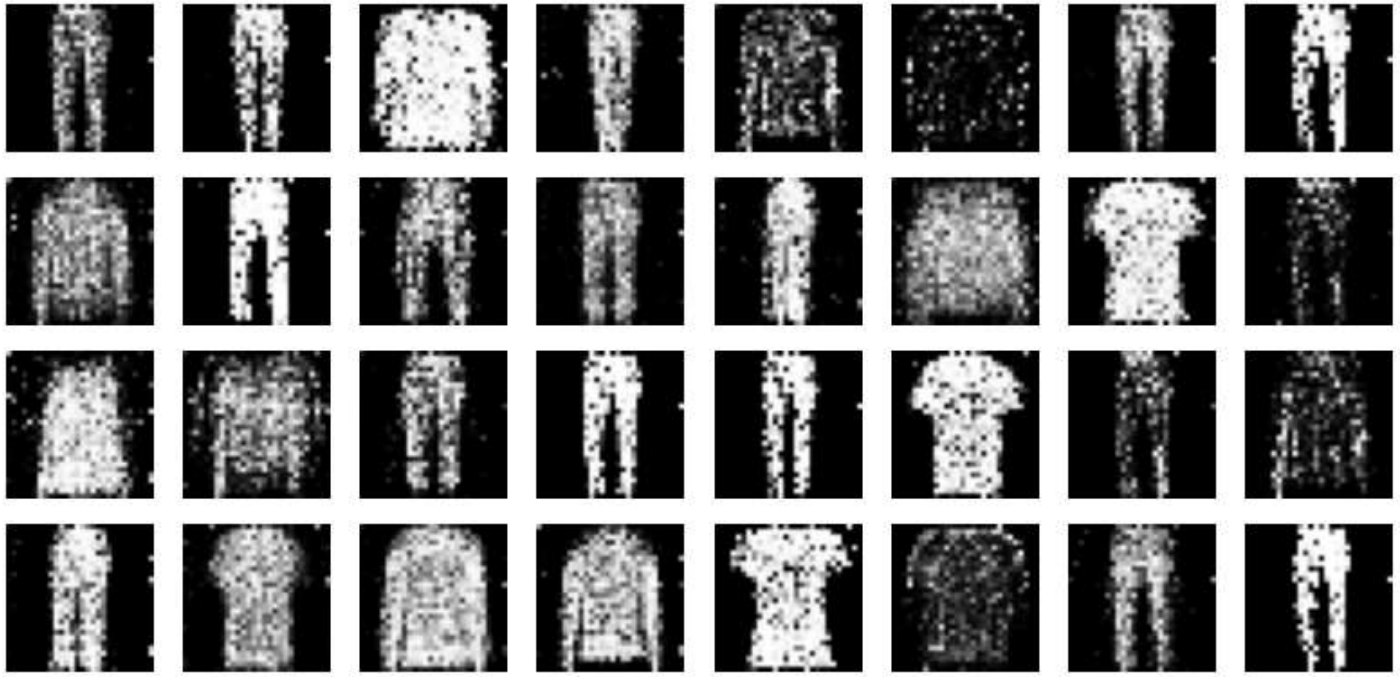
```
Epoch 3/10
  Step 0 - d_loss: 0.6046, d_acc: 0.6630, g_loss: 1.1556
  Step 100 - d_loss: 0.6085, d_acc: 0.6601, g_loss: 1.1469
  Step 200 - d_loss: 0.6131, d_acc: 0.6558, g_loss: 1.1351
  Step 300 - d_loss: 0.6166, d_acc: 0.6527, g_loss: 1.1264
  Step 400 - d_loss: 0.6204, d_acc: 0.6488, g_loss: 1.1159
  Step 500 - d_loss: 0.6238, d_acc: 0.6447, g_loss: 1.1035
Epoch 3 duration: 3.68 seconds
Visualizing generated images after epoch 3
```



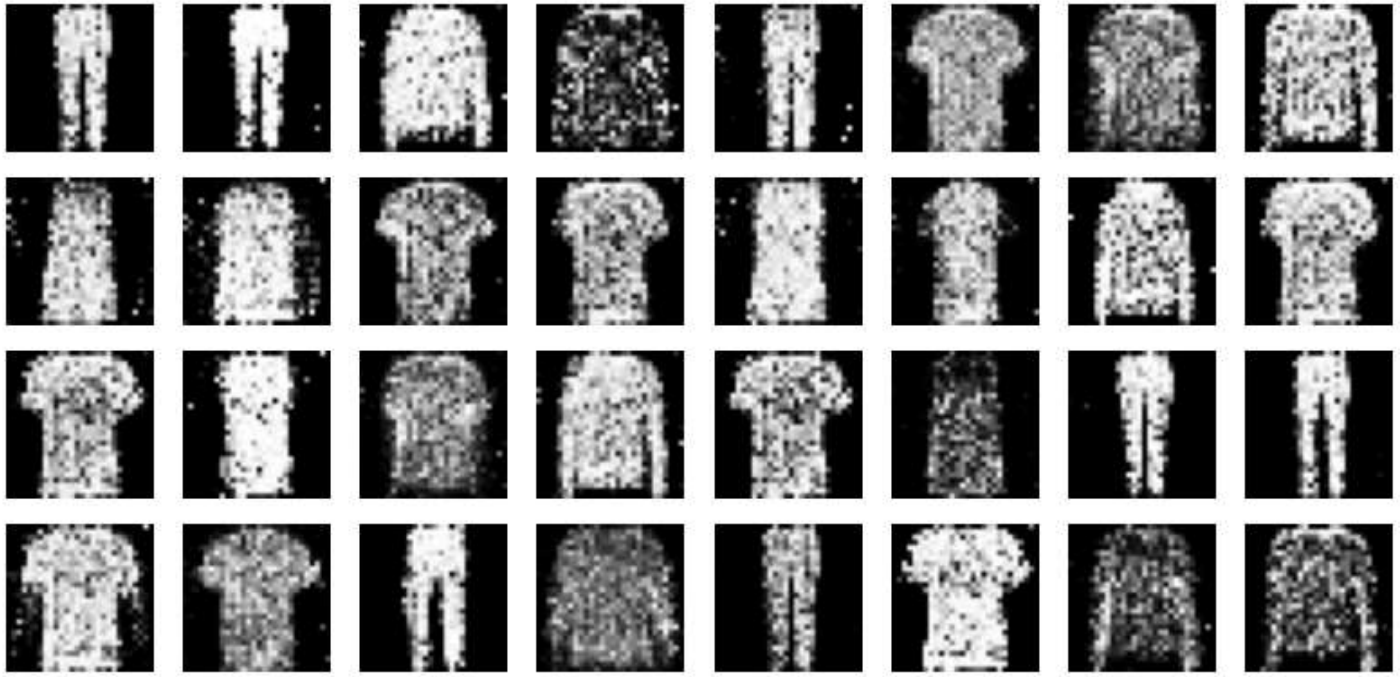
```
Epoch 4/10
  Step 0 - d_loss: 0.6259, d_acc: 0.6424, g_loss: 1.0972
  Step 100 - d_loss: 0.6289, d_acc: 0.6385, g_loss: 1.0864
  Step 200 - d_loss: 0.6320, d_acc: 0.6347, g_loss: 1.0757
  Step 300 - d_loss: 0.6345, d_acc: 0.6314, g_loss: 1.0656
  Step 400 - d_loss: 0.6368, d_acc: 0.6280, g_loss: 1.0569
  Step 500 - d_loss: 0.6387, d_acc: 0.6248, g_loss: 1.0479
Epoch 4 duration: 3.66 seconds
Visualizing generated images after epoch 4
```



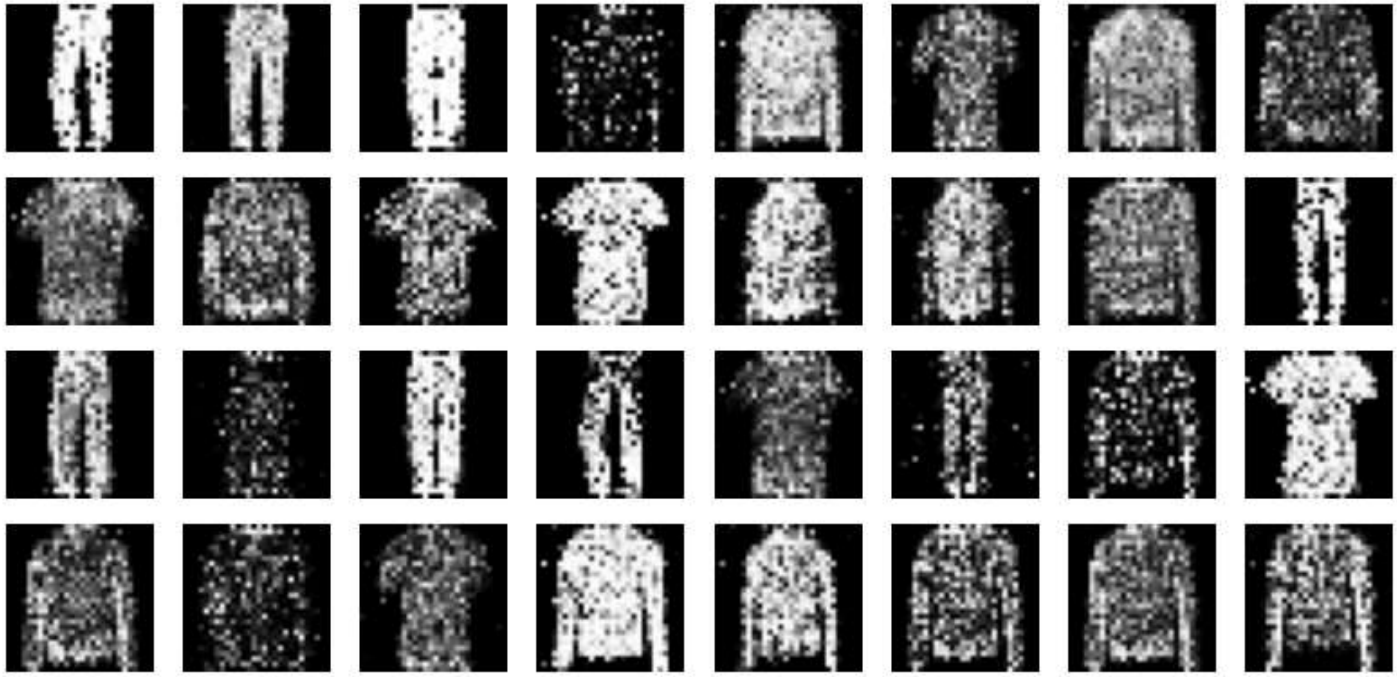
```
Epoch 5/10
  Step 0 - d_loss: 0.6400, d_acc: 0.6231, g_loss: 1.0431
  Step 100 - d_loss: 0.6418, d_acc: 0.6205, g_loss: 1.0354
  Step 200 - d_loss: 0.6434, d_acc: 0.6181, g_loss: 1.0280
  Step 300 - d_loss: 0.6449, d_acc: 0.6162, g_loss: 1.0212
  Step 400 - d_loss: 0.6465, d_acc: 0.6139, g_loss: 1.0143
  Step 500 - d_loss: 0.6480, d_acc: 0.6120, g_loss: 1.0075
Epoch 5 duration: 3.73 seconds
Visualizing generated images after epoch 5
```



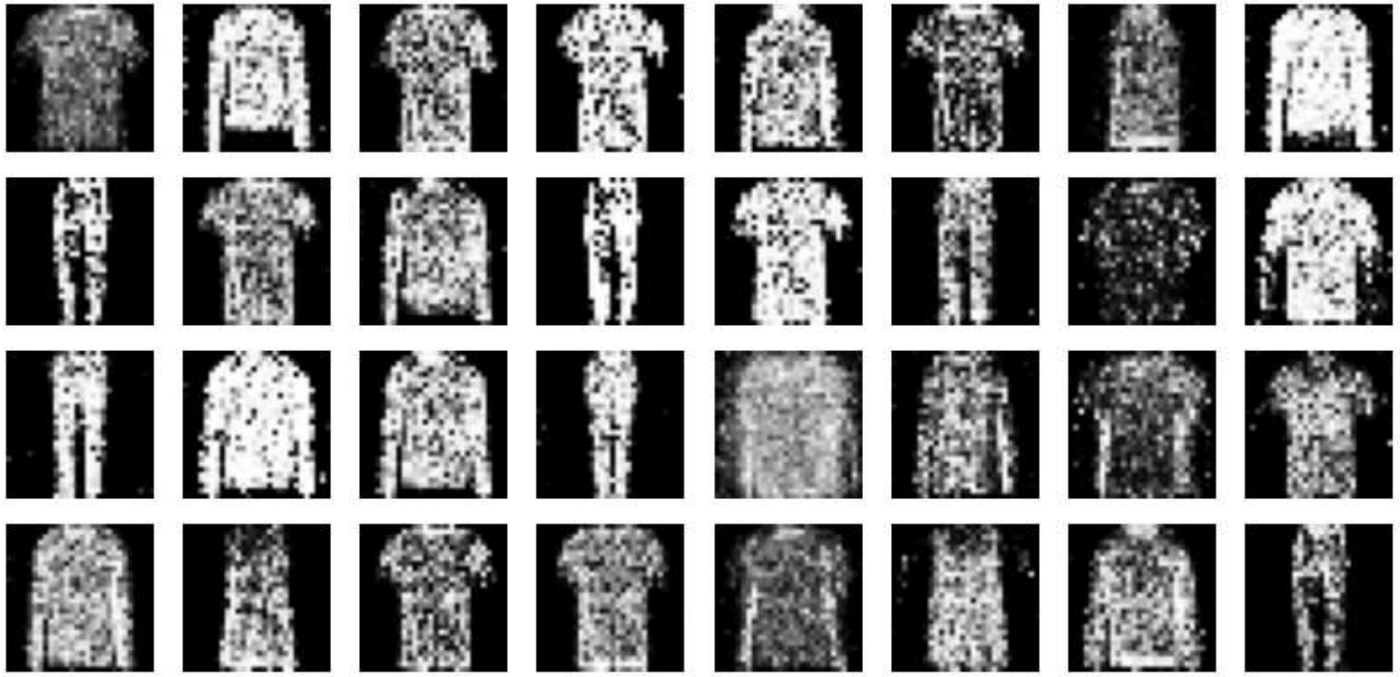
```
Epoch 6/10
  Step 0 - d_loss: 0.6489, d_acc: 0.6105, g_loss: 1.0035
  Step 100 - d_loss: 0.6499, d_acc: 0.6090, g_loss: 0.9975
  Step 200 - d_loss: 0.6509, d_acc: 0.6072, g_loss: 0.9921
  Step 300 - d_loss: 0.6518, d_acc: 0.6058, g_loss: 0.9868
  Step 400 - d_loss: 0.6526, d_acc: 0.6048, g_loss: 0.9817
  Step 500 - d_loss: 0.6532, d_acc: 0.6040, g_loss: 0.9774
Epoch 6 duration: 3.90 seconds
Visualizing generated images after epoch 6
```



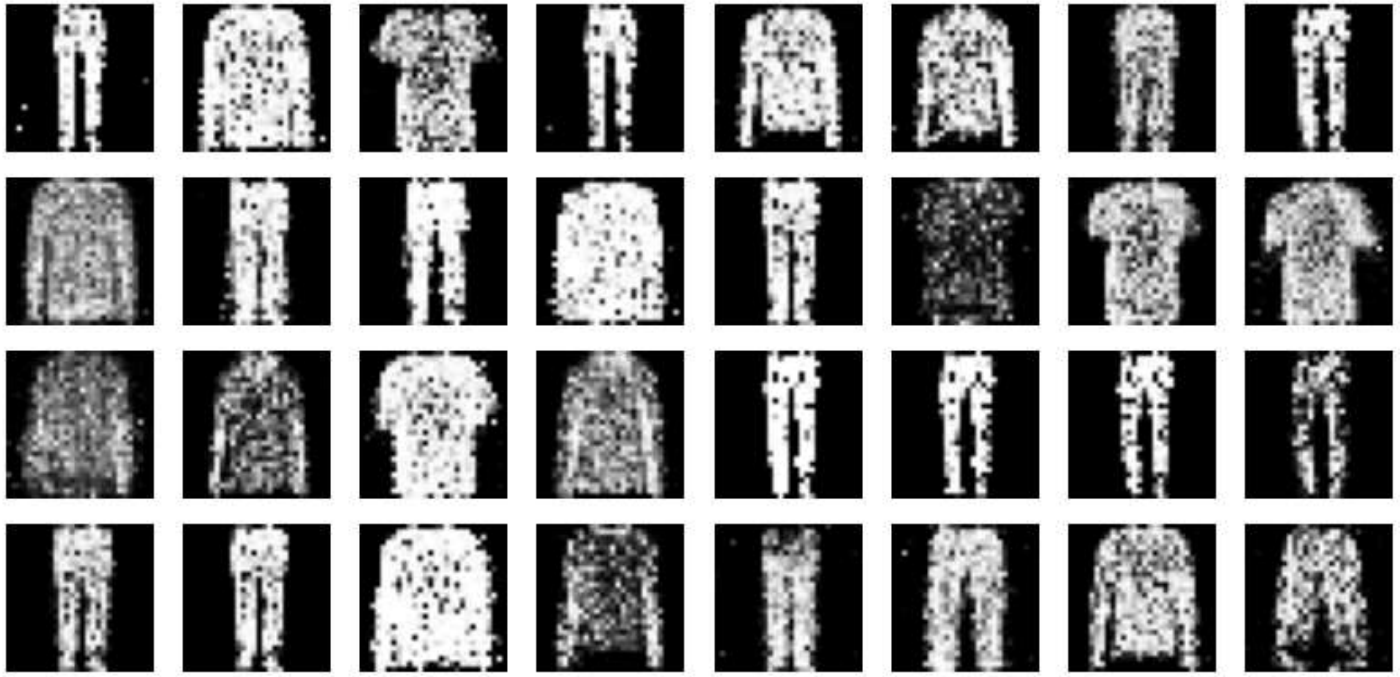
```
Epoch 7/10
  Step 0 - d_loss: 0.6537, d_acc: 0.6032, g_loss: 0.9748
  Step 100 - d_loss: 0.6543, d_acc: 0.6020, g_loss: 0.9709
  Step 200 - d_loss: 0.6548, d_acc: 0.6012, g_loss: 0.9672
  Step 300 - d_loss: 0.6552, d_acc: 0.6005, g_loss: 0.9641
  Step 400 - d_loss: 0.6556, d_acc: 0.6000, g_loss: 0.9610
  Step 500 - d_loss: 0.6560, d_acc: 0.5994, g_loss: 0.9583
Epoch 7 duration: 4.84 seconds
Visualizing generated images after epoch 7
```



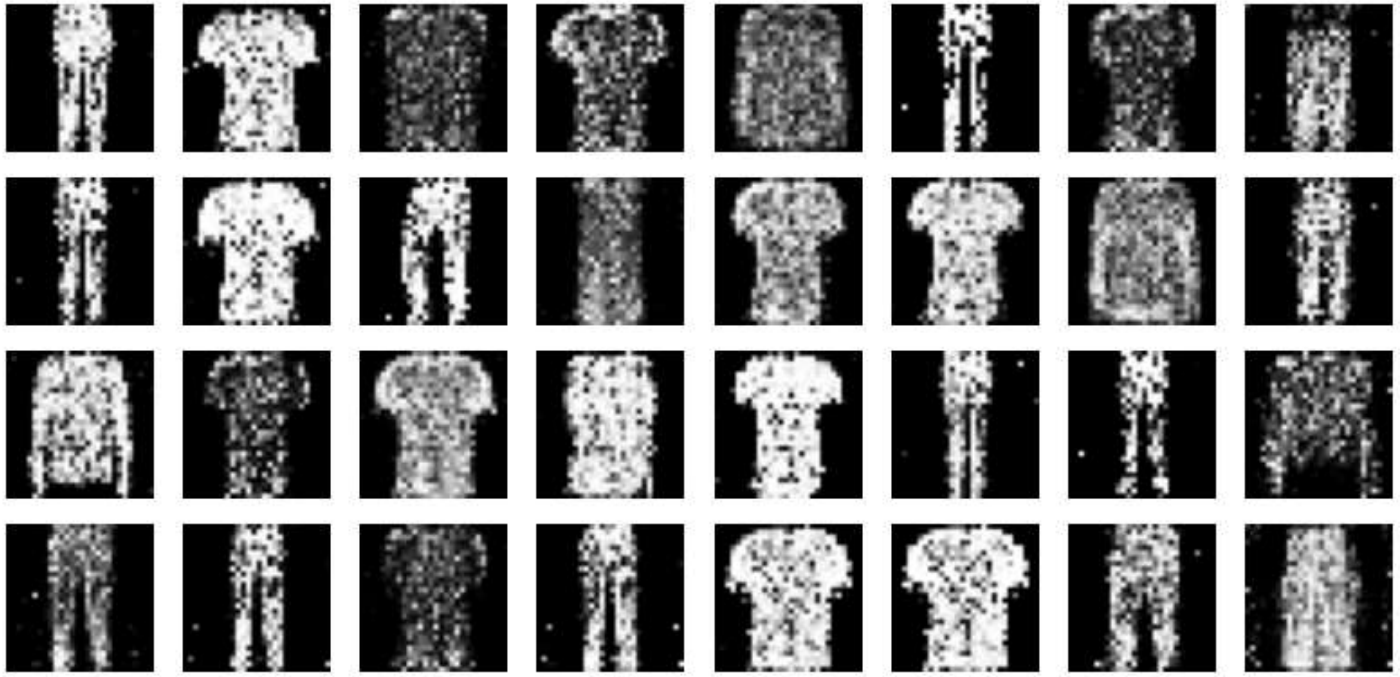
```
Epoch 8/10
  Step 0 - d_loss: 0.6562, d_acc: 0.5991, g_loss: 0.9564
  Step 100 - d_loss: 0.6567, d_acc: 0.5983, g_loss: 0.9533
  Step 200 - d_loss: 0.6570, d_acc: 0.5978, g_loss: 0.9506
  Step 300 - d_loss: 0.6575, d_acc: 0.5971, g_loss: 0.9480
  Step 400 - d_loss: 0.6579, d_acc: 0.5966, g_loss: 0.9455
  Step 500 - d_loss: 0.6583, d_acc: 0.5960, g_loss: 0.9431
Epoch 8 duration: 4.50 seconds
Visualizing generated images after epoch 8
```



```
Epoch 9/10
  Step 0 - d_loss: 0.6586, d_acc: 0.5956, g_loss: 0.9416
  Step 100 - d_loss: 0.6589, d_acc: 0.5951, g_loss: 0.9392
  Step 200 - d_loss: 0.6591, d_acc: 0.5948, g_loss: 0.9374
  Step 300 - d_loss: 0.6595, d_acc: 0.5943, g_loss: 0.9352
  Step 400 - d_loss: 0.6596, d_acc: 0.5942, g_loss: 0.9336
  Step 500 - d_loss: 0.6597, d_acc: 0.5939, g_loss: 0.9319
Epoch 9 duration: 4.12 seconds
Visualizing generated images after epoch 9
```



```
Epoch 10/10
  Step 0 - d_loss: 0.6598, d_acc: 0.5939, g_loss: 0.9309
  Step 100 - d_loss: 0.6600, d_acc: 0.5935, g_loss: 0.9293
  Step 200 - d_loss: 0.6602, d_acc: 0.5932, g_loss: 0.9278
  Step 300 - d_loss: 0.6604, d_acc: 0.5932, g_loss: 0.9267
  Step 400 - d_loss: 0.6605, d_acc: 0.5932, g_loss: 0.9257
  Step 500 - d_loss: 0.6607, d_acc: 0.5930, g_loss: 0.9242
Epoch 10 duration: 4.00 seconds
Visualizing generated images after epoch 10
```



```
Total training time: 45.30 seconds
```

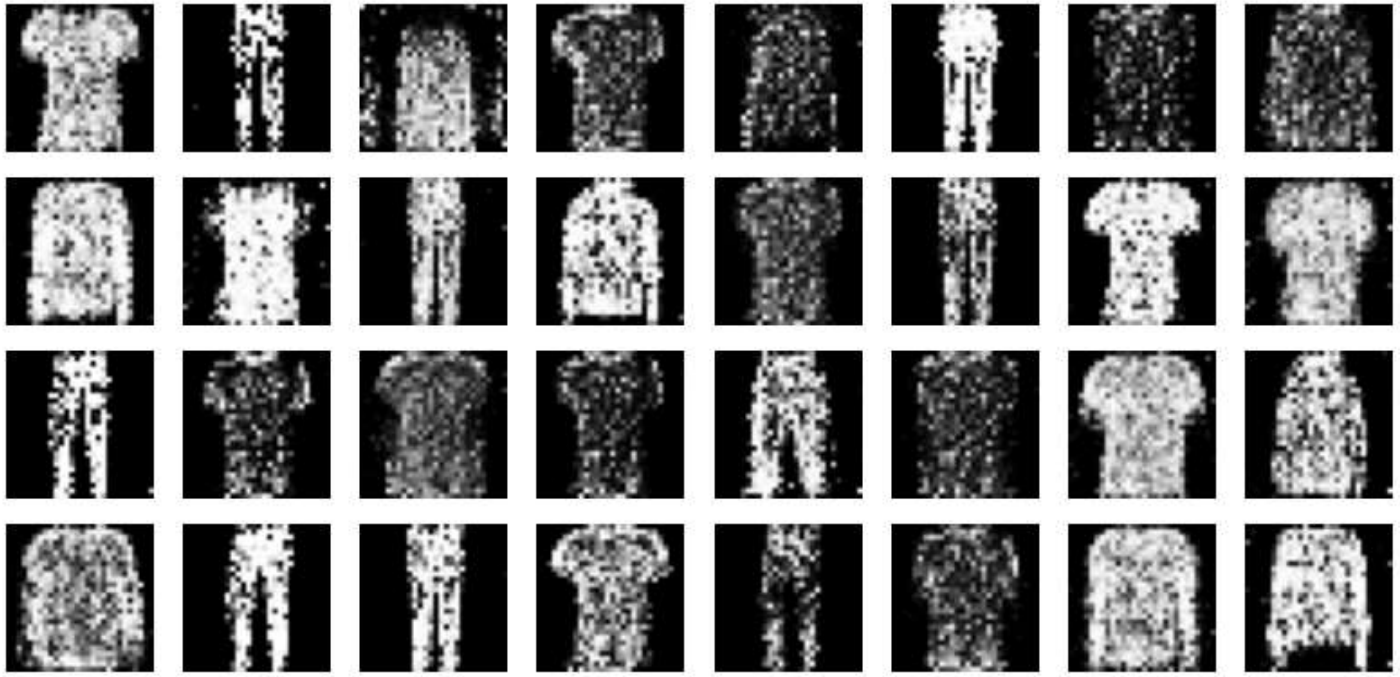
2.5

After training the Generator, feed it with 32 random noise vectors and visualize the 32 generated images. Are you satisfied with the results?

Generate and visualize 32 new images after training

In [32]:
```
print("Generated images from trained generator:")
noise = tf.random.normal(shape=[32, codings_size])
generated_images = generator(noise)
plot_multiple_images(generated_images, n_cols=8)
```

```
Generated images from trained generator:
```



The generator has learned some basic structure of the three selected classes, since the generated images resemble clothing items in general shape. The vast majority of the T-shirts, Trousers, and Pullovers are easily identifiable, while others are still fuzzy or ambiguous. This indicates that the GAN has not yet fully converged, and could benefit from more training epochs, batch normalization, or improved weight scaling or regularization.

2.6

What is the accuracy of the discriminator in predicting that the generated images of the previous question are fake?

Evaluate discriminator's accuracy on the 32 generated images

In [33]:
```
labels_fake = tf.constant([[0.]] * 32)  # label = fake
loss, accuracy = discriminator.evaluate(generated_images, labels_fake, verbose=0)

print(f"\nDiscriminator accuracy on generated (fake) images: {accuracy:.4f}")
```

```
Discriminator accuracy on generated (fake) images: 0.6875
```

The discriminator achieved an accuracy of 0.6875 on the 32 generated (fake) images. This means it correctly identified approximately 22 out of 32 images as fake. This result indicates that the discriminator is still relatively strong, but not perfect — which is a good sign. It suggests that the generator has started producing outputs that are somewhat realistic, to the point that the discriminator misclassifies them roughly 31% of the time. Ideally, in a well-balanced GAN, we aim for the discriminator to have an accuracy around 50%, meaning it can no longer reliably distinguish real from fake. So, while his result shows some progress, further training or architectural improvements could help the generator produce even more convincing outputs.