$E_{\text{in-style}}$: Clothing Item Generation using GANs

Charlie Brayton (014559415)
Department of Software Engineering
San José State University
San José, California
charles.brayton@sjsu.edu

Mohit Patel (014501461)
Department of Software Engineering
San José State University
San José, California
mohit.patel@sjsu.edu

Andrew Selvia (014547273)
Department of Software Engineering
San José State University
San José, California
andrew.selvia@sjsu.edu

Dylan Zhang (013073437)
Department of Software Engineering
San José State University
San José, California
dylan.zhang@sjsu.edu

***Abstract*—The primary objective of this research is to classify and generate images of clothing items. The work is a compilation of three distinct efforts: (1) to classify fashion-mnist images and, inversely, fool it with adversarial noise; (2) to generate novel images which emulate the true fashion-mnist images; (3) to infuse the grayscale images with life-like colors. Together, these endeavours pushed the team to explore topics at the forefront of modern machine learning, most notably generative adversarial networks (GANs).**

***Index Terms*—machine learning, computer vision, neural networks, generative adversarial networks**

I. Introduction

The diverse branches of this project were made possible by the careful selection of a dataset. The fashion-mnist dataset was chosen specifically for its prevalence as a benchmark in recent research into neural networks. Its size, labels, and compatibility (with the long-established MNIST dataset) make it an ideal choice for evaluating deep neural networks (NNs) and GANs. Rather than attempt to introduce a truly groundbreaking NN architecture given the authors' limited experience, this project aims for breadth.

Each subproject explores a unique research area with industrial applications which may answer the associated questions:

1) Classification: Can we quickly identify an item of clothing to automate sorting and retrieval? Supposing such an automated system existed, how susceptible might it be to failure?
2) Generation: Can the product development lifecycle for clothing be accelerated by avoiding physical prototyping?
3) Colorization: Can grayscale clothing imagery be augmented with colors to enable A/B testing of styles to increase customer satisfaction?

To aid comprehension, each subproject is explored independently.

II. Data

The fashion-mnist dataset is composed of 60,000 training images and 10,000 testing images. In keeping with the legacy of the traditional MNIST, fashion-mnist images are 28x28 grayscale pixels. Each image is labeled as one of the ten classes enumerated below:

0) T-shirt/top
1) Trouser
2) Pullover
3) Dress
4) Coat
5) Sandal
6) Shirt
7) Sneaker
8) Bag
9) Ankle Boot

III. Implementation

A. Classification

Classification was performed using two neural network architectures, one using only dense layers and another using convolutional layers. The first architecture, shown in Figure 1, flattens the 28 by 28 pixel image into a vector 784 pixels long, which is then densely connected to a 128 node layer; finally, the 128 node layer is densely connected to a 10 node output layer. The second architecture, shown in Figure 2, scans the image with a convolutional neural network using a 5 by 5 window and 64 layers; the results of the first convolutional layer are then scanned by another convolutional layer with a 5 by 5 window and 128 layers. The final step is to flatten the layer and densely connect it to the 10 node output layer. The output layer of both architectures indicates the network's confidence that the image belongs to each of the classes, so a large value in 0^{th} node and the 6th node would indicate that the network is uncertain whether an image should be classified as a T-shirt/top or just a Shirt. The neural networks described were constructed using Keras and Tensorflow in python notebooks.

The robustness of the classifier was tested by adding adversarial signals to the input images. The adversarial signal was calculated by determining the gradient of the loss function based on the input image used. This allows the adversarial

Fig. 1

Fig. 2

signal to skew the image to the closest incorrect classification. Once the adversarial signal was determined, it was scaled by an epsilon value (between 0 and 1) and added to the image; for adding this noise the image is considered to be all pixels that have a brightness value greater than 0. Limiting the noise to non-zero values means the classification isn't being skewed due to noise added in the black background portions of the image. Robustness was determined by comparing the accuracy of our classifiers against ever-increasing epsilon values.

B. Generation

https://github.com/AndrewSelviaSJSU/pytorch-generative-model-collections

C. Colorization

The images in the fashion mnist dataset are all grayscale images. We decided to build a machine learning model that would fill colors into these black and white images intelligently. There were several techniques available to explicitly fill colors into the grayscale images. However, in order to smartly colorize the fashion mnist dataset we decided that Generative Adversarial Networks were the best choice for this task.

To build a model capable of smartly colorizing grayscale images we had to first build a dataset for training our GAN model. For this we combined the training and testing sets of the fashion mnist dataset. After this, we explicitly filled colors into these 70,000 grayscale images. Initially, the shape of each image was (28,28,1). After colorizing the images, their shape changed to (28,28,3). This marked the addition of 3 rows for each item in the dataset. This change happened to store the values of the red, blue and green elements of each image. Each label of the fashion mnist dataset was assigned different colors, for instance, ankle boots were colored brown and dresses were colored red. Now, we had 70,000 data instances each representing a colored fashion item from the fashion mnist dataset of the shape (28,28,3).

TODO: Reference Figure 3.

Fig. 3: Training samples for colorization

Fig. 4: ACGAN Learning to Generate Clothing Images

(a) Epoch 1 (b) Epoch 25 (c) Epoch 50

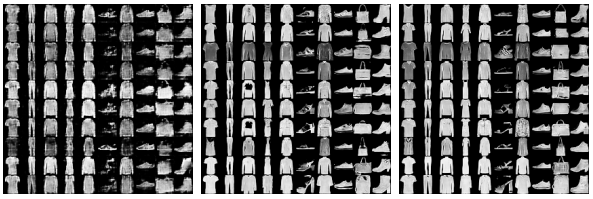
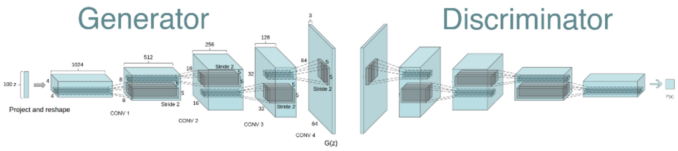


Fig. 5: DCGANs Architecture



Once we had built our training samples, we decided to work on our GAN. We fed the generator in the network the original grayscale images from the fashion mnist dataset. The generator then tried to fill colors into these grayscale images of fashion items using some randomly set weights. These generated images were then passed on to the discriminator. Along with these images, the discriminator also received the explicitly colored images from the training sample. The discriminator tried to identify which image was created from the generator and which was drawn from the training sample. Based on the decision made on the classification task by the discriminator, the discriminator loss as well as the generator loss were computed.

The discriminator in the network learnt from the discriminator loss by backpropagating through the discriminator node and updating the weights associated with the discriminator. Similarly, the generator updated its weights using the results obtained from backpropagating through the generator node and using the generator loss. This process of updating the weights and learning was carried out iteratively. After running this iterative process for certain epochs, we had the final weights for the network that were capable of colorizing the items in the fashion mnist dataset intelligently.

IV. Results

A. Classification

B. Generation

C. Colorization

We used DCGANs to define the neural network model. The DCGANs architecture displays how it worked when we were using it for It introduces the generators and discriminators as well as the training process step by step.

We briefly summarize the generating principles of DCGANs for our dataset by improving and optimizing the Fashion-MNIST dataset, and then step by step introducing the generator and discriminator and the training process. DCGANs

Fig. 6: The result of Discriminator

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 64)        4864
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 32, 32, 64)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 128)       204928
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 16, 16, 128)       0
_________________________________________________________________
dropout (Dropout)            (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 256)         819456
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 8, 8, 256)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 8, 8, 256)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 4, 4, 512)         3277312
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 4, 4, 512)         0
_________________________________________________________________
dropout_2 (Dropout)          (None, 4, 4, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dense (Dense)                (None, 1)                 8193
_________________________________________________________________
activation (Activation)      (None, 1)                 0
=================================================================
Total params: 4,314,753
Trainable params: 4,314,753
Non-trainable params: 0
_________________________________________________________________
```

use a random noise vector as an input, and then the input is scaled up into two-dimensional data similar to CNN's but in the opposite structure. For non-linear layers, we recommend using LeakyReLU for all layers for the discriminator, and our results in the discriminator function can be seen below.

For the generator, we recommend using the BatchNormalize layer because using BatchNormalize allows the CNN version of the generator to learn better, but it is impossible to use BatchNormalize for all layers. The output layer of the generator and the input layer of the discriminator do not require to add BatchNormalize layer. Our result in the generator function can be seen below.

The size of non-trainable parameters in Adam optimizer was larger than RMSprop. This result excludes all parameters of the discriminator. We want to use Adam optimizer in our project.

TODO: Reference Figure 4. TODO: Reference Figure 5. TODO: Reference Figure 6. TODO: Reference Figure 7. TODO: Reference Figure 8. TODO: Reference Figure 9. TODO: Cite [1]. TODO: Cite [2].

With this structure of the generative model and the discriminant model of the CNN structure, DCGANs can achieve considerable results in image generation.

Parameter setting in our project:

- batch size: 64
- Used Adam optimizer instead of Root Mean Square Propagation optimizer
- iteration times: 20000
- learning rate: 0.0001

Fig. 7: The result of Generator

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 16384)             1654784
_________________________________________________________________
batch_normalization (BatchNo (None, 16384)             65536
_________________________________________________________________
activation_1 (Activation)    (None, 16384)             0
_________________________________________________________________
reshape (Reshape)            (None, 8, 8, 256)         0
_________________________________________________________________
dropout_3 (Dropout)          (None, 8, 8, 256)         0
_________________________________________________________________
up_sampling2d (UpSampling2D) (None, 16, 16, 256)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 128)       819328
_________________________________________________________________
batch_normalization_1 (Batch (None, 16, 16, 128)       512
_________________________________________________________________
activation_2 (Activation)    (None, 16, 16, 128)       0
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 32, 32, 128)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 32, 32, 128)       409728
_________________________________________________________________
batch_normalization_2 (Batch (None, 32, 32, 128)       512
_________________________________________________________________
activation_3 (Activation)    (None, 32, 32, 128)       0
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 64, 64, 128)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 64, 64, 64)        204864
_________________________________________________________________
batch_normalization_3 (Batch (None, 64, 64, 64)        256
_________________________________________________________________
activation_4 (Activation)    (None, 64, 64, 64)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 64, 64, 32)        51232
_________________________________________________________________
batch_normalization_4 (Batch (None, 64, 64, 32)        128
_________________________________________________________________
activation_5 (Activation)    (None, 64, 64, 32)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 64, 64, 3)         2403
_________________________________________________________________
activation_6 (Activation)    (None, 64, 64, 3)         0
=================================================================
Total params: 3,209,283
Trainable params: 3,175,811
Non-trainable params: 33,472
_________________________________________________________________
```

Fig. 8: The result of Adam

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_1 (Sequential)    (None, 64, 64, 3)         3209283
_________________________________________________________________
sequential (Sequential)      (None, 1)                 4314753
=================================================================
Total params: 7,524,036
Trainable params: 3,175,811
Non-trainable params: 4,348,225
_________________________________________________________________
```

Fig. 9: The result of RMSprop

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential (Sequential)      (None, 1)                 4314753
=================================================================
Total params: 4,314,753
Trainable params: 4,314,753
Non-trainable params: 0
_________________________________________________________________
```

- BatchNormalization: momentum=0.9

In our project, we used DCGANs to generate colorful clothes, and you can try to change the dataset to generate images that your personal needs or try to modify the neural network parameters to observe different generation effects. It should be noted that the generated images are not as good as the black and white examples, mainly due to the difficulty in setting up the training set. We tried replacing too many background images and then using this new dataset, the generator must generate clothing and have a valid background (retrieved from real images). With more training or some special techniques, you may be able to improve these generated images. However, since this is one way to use GAN to adapt to RGB images, we are delighted with the results.

V. Conclusion

References

[1] C. Brayton, M. Patel, A. Selvia, and D. Zhang, "e-in-style," 2020. https://github.com/AndrewSelviaSJSU/e-in-style.

[2] C. Brayton, M. Patel, A. Selvia, and D. Zhang, "pytorch-generative-model-collections," 2020. https://github.com/AndrewSelviaSJSU/pytorch-generative-model-collections.