Deep Learning Project Milestone 2
Vani Kanoria

Generating Fashion Images Using Generative Adversarial Networks

I use Fashion-MNIST data to train a model that generates fashion images from random noise, uses a discriminator to predict which of the images are fake, and uses the feedback loop during training to improve upon the images generated. I.e. make them more and more realistic and harder to distinguish from the real images.

Fashion-MNIST is a dataset consisting of 28×28 grayscale images of 70,000 fashion products from 10 categories, with 7,000 images per category. The training set has 60,000 images and the test set has 10,000 images. Fashion-MNIST shares the same image size, data format and the structure of training and testing splits with the original MNIST.

I follow the following tutorial to accomplish with the Fashion_MNIST dataset what the author does with the MNIST dataset:
https://towardsdatascience.com/image-generation-in-10-minutes-with-generative-adversarial-networks-c2afc56bfa3b

I use a Convolutional Neural Network (CNN) over an Artificial Neural Network or Recurrent Neural Network (RNN) for the following reasons:

- If I were to use ANN, I would have to convert a 2-dimensional image into a 1-dimensional vector prior to training the mode, which would increase the number of trainable parameters drastically. ANN also loses the spatial features of an image i.e. the arrangement of pixels in the image.
- Both ANN and RNN suffer from the vanishing and exploding gradient problems (although it is true that CNN does, too).
- The building blocks of CNNs are filters a.k.a. kernels. Kernels are used to extract the relevant features from the input using the convolution operation.
- CNN captures the spatial features from an image, which helps it identify an object accurately, along with the location of the object, and the relation with other objects in the image. This is not true in the case of ANN and RNN.
- Both CNN and RNN implement parameter sharing. In CNN, a single filer is applied across different parts of an input to produce a feature map. In RNN, the parameters are shared across

different time steps. RNN is commonly used for sequence data and text data, while CNN is commonly used for spatial data.

(Source: https://www.analyticsvidhya.com/blog/2020/02/cnn-vs-rnn-vs-mlp-analyzing-3-types-of-neural-networks-in-deep-learning/)

As mentioned in the tutorial, one of the best performing generative algorithms for image generation is Generative Adversarial Networks (or GANs).

Networks and Functions created to be used in the algorithm:

- Generate a **generator network** (with Keras Sequential API) that generates 28x28 pixels grayscale fake images from random noise. We use BatchNormalization, LeakyRelu and Conv2DTranspose layers.
- Generate a **discriminator network** by following the inverse version of the generator network. We add Conv2D, LeakyReLU and Dropout layers along with a Flatten layer at the end. The discriminator network takes the 28x28 pixels image data and outputs a single value, representing the possibility of authenticity. Positive and negative values signify a real and fake image respectively.
- Since there are two sub-networks, we define two loss functions and two optimizers. The loss functions calculate the cross_entropy of each output, and the optimizers utilize Adam.
- We set a checkpoints directory to save the progress at every epoch

Algorithm for training: I use the following steps to generate the images of fashion items:

(i) Create random noise from a seed

(ii) Generate images and calculate loss values:

- use GradientTape to record operations for automatic differentiation
- Call the generator to generate images
- Call the discriminator on real (input) images to get real output
- Call the discriminator on generated images to get fake output
- Calculate generator loss from fake output
- Calculate total discriminator loss from real and fake outputs

(iii) Calculate the gradients using loss values and model variables:

- Calculate the generator gradient
- Calculate the discriminator gradient

(iv) Process the gradients and run the optimizers

- Run the optimizer on the gradients and trainable variables of the generator
- Run the optimizer on the gradients and trainable variables of the discriminator

I use tf.function to compile the function encapsulating the above algorithm into a callable Tensorflow graph.

During the training loop, I record time spent at the beginning of each epoch, produce gif images and display them, save the model every five epochs as a checkpoint, print out the completed epoch time, and generate a final image in the end after the training is completed.

At the end, I create a GIF image visualizing the evolution of the samples generated by the GAN.

The evaluation in this case is done by looking at the progression of images created in the gif.

Next steps: I can mix the generated images with the test images, and use the discriminator to test it.