CS 4/510: Computer Vision & Deep Learning, Summer 2025

Programming Assignment #5

A. Rhodes

Note: This assignment is **due by Sunday, 8/17 @ 10pm**; turn in the assignment by email to our TA. Submit assignments to our TA via email using Canvas.

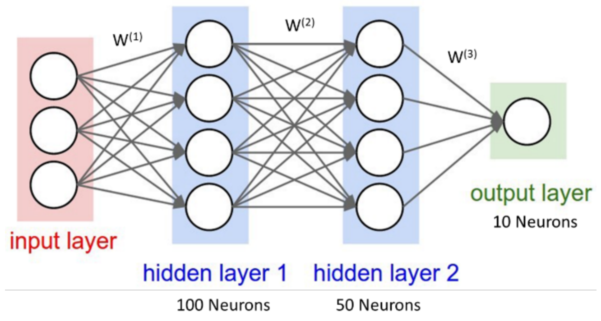
Note that there are (2) exercises listed below. **Complete one of the exercises of your choosing** (please don't do both).

## Exercise #1: Low-Rank Model Compression

In this exercise you will train a reasonably light-weight, dense, feed-forward neural network on the standard MNIST dataset. After training, you will perform various degrees of low-rank matrix approximation (SVD-based) on the weight matrices of this model, then perform refinement training and finally report the test results of the compressed model(s).

### Step 1: Design/Train the Light-Weight Model

Train a light-weight, dense, feed-forward neural network (note: not a CNN) so that the first hidden layers has 100 neurons, the next hidden layer has 50, and the final output layer has 10 neurons. All the layers should be fully-connected; layers should include conventional bias neurons. Include a model summary with trainable parameter counts in your write-up.



Load and pre-process the MNIST dataset; report on the pre-processing procedure that you apply in your assignment write-up.

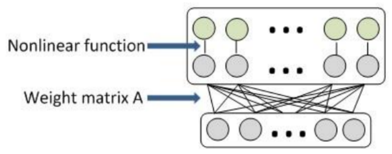
Train your model on MNIST for 100 epochs, report the training/test loss and accuracy for each epoch; include a 10x10 confusion matrix for the test data results on the fully trained model.

## Step 2: Generate the Low-Rank Model

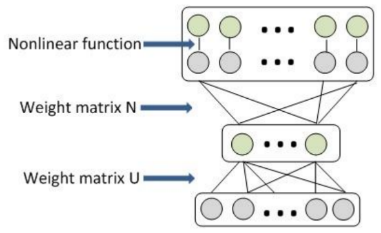
For each weight matrix in your model: $W^{(1)}, W^{(2)}, W^{(3)}$, perform SVD (fine to use a SW library function for this), so that $W^{(i)} \approx U^{(i)}\Sigma^{(i)}\left(V^{(i)}\right)^T$. To perform compression, your SVD decomposition should represent a **k-rank approximation to $W^{(i)}$**; in this way, if, say, $W^{(i)}$ is of dimension $m \times n$, then $U^{(i)}$ will be of dimension $m \times k$ (where $k < n$), $\Sigma^{(i)}$ is of dimension $k \times k$ and $\left(V^{(i)}\right)^T$ is of dimension $k \times n$.

For simplicity, as I have shown in lecture, express this decomposition as the product of two matrices: $W^{(i)} \approx U'^{(i)}\left(V^{(i)}\right)^T$, where $U'^{(i)} = U^{(i)}\Sigma^{(i)}$.

Define a new model – your compressed model – where each dense layer is from your 100-50-10 model is replaced with two dense layers (representing the factors of your low-rank approximation for each $W^{(i)}$). See the schematic below:



(a) One layer in original DNN model



b) Two corresponding layers in new DNN model

I recommend using the Keras function "set_weights" to manually set the weights, after defining your model, e.g.:

```
model.layers[ix].set_weights(A)
```

where above "$ix$" denotes the layer index and $A$ denotes the layer weight matrix. Note that you can randomly initialize the layer biases in your new model or copy them from the previously trained model – either method is fine, but please include details of your design decisions in your assignment write-up. Include a model summary of your compressed model with your assignment write-up.

## Step 3: Apply Refinement Training to the Low-Rank Model

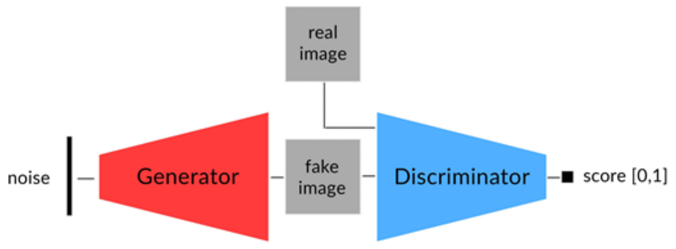
Train your compressed model for 10 epochs; report the training/test loss and accuracy for each epoch; include a 10x10 confusion matrix for the test data results on the fully trained model. Note that you should **not** randomly initialize the weights of your compressed model prior to training – instead, use the weights learned from the low-rank approximation with the set_weights function as described in Step 2, above.

**Step 4: Apply Different Degrees of Compression**

Execute Steps 2-3 above at 2X, 4X and 8X compression, respectively. For instance, with 2X compression, if $W^{(i)}$ is of dimension $m \times n$ (uncompressed), apply SVD-based compression where $k = int\left(\frac{n}{2}\right)$, for 4X compression, set $k = int\left(\frac{n}{4}\right)$ and for 8X compression, $k = int\left(\frac{n}{8}\right)$; for each compression level include a model summary/trainable parameter count.

**Exercise #2: FashionMNIST GAN**

In this exercise you will train a GAN from scratch on the FashionMNIST dataset.



**Step 1: Design the GAN**

Using a high-level DL library (e.g., Keras), design the **generator sub-network** to include 3-4 core dense layers. Begin with a dense layer of input dimension 100 and the final output of your generator should be of dimension 784, which you then reshape to (28x28), i.e., the same size as images in the FashionMNIST dataset. Between the dense layers, I recommend using batch normalization (BN); I also recommend using the "leaky RELU" activation; feel free to incorporate other architecture elements, including dropout, etc. Include a model summary of your generator model, including trainable parameter count, with your assignment write-up.

Next, design your **discriminator sub-network**. As with the generator, I recommend using 3-4 core dense layers (although you can use experiment with more if you are inclined). In general, the number of trainable parameters for your generator and discriminator should be in the same ballpark – note, however, that they don't need to be exactly the same, and the sub-networks likewise do not need to be exactly "symmetric" (as one would typically see with an AE). Once again, I recommend incorporating the leaky RELU activation plus BN and/or dropout. The input to the discriminator should be of dimension (28x28), which should immediately be followed by a "Flatten" command, yielding a vector of dimension 784. The final output of the discriminator should consist of a single neuron; I recommend using a sigmoid activation for this output neuron. Include a model summary of your discriminator model, including trainable parameter count, with your assignment write-up.

Finally, construct the full GAN network by appending the generator and discriminator together into a single sequential network (if using Keras you can simply concatenate the sequential models in one step here); **be sure that the discriminator layer weights in the full GAN model are fixed**, i.e., not trainable (e.g., set discriminator.trainable = False). Use the Adam optimizer with binary cross entropy loss for the GAN; the discriminator network on its own should be trained using the Adam optimizer with binary cross entropy loss.

## Step 2: Train the GAN and Discriminator

Train the GAN and discriminator models for 100 epochs on the FashionMNIST dataset; you should apply standard pre-processing steps on the dataset prior to training. Report your pre-processing steps in the assignment write-up.

Note that some care needs to be taken when training the GAN and discriminator models, as the training process is somewhat more nuanced than standard NN training. I recommend writing your own training for loop to this end. For each batch of training (say with batch size *B*), generate a random vector sampled from a standard MVN distribution – one for each input image in the batch (notice that the MVN should be of dimension 100 here, corresponding with the input dimension of the generator sub-network).

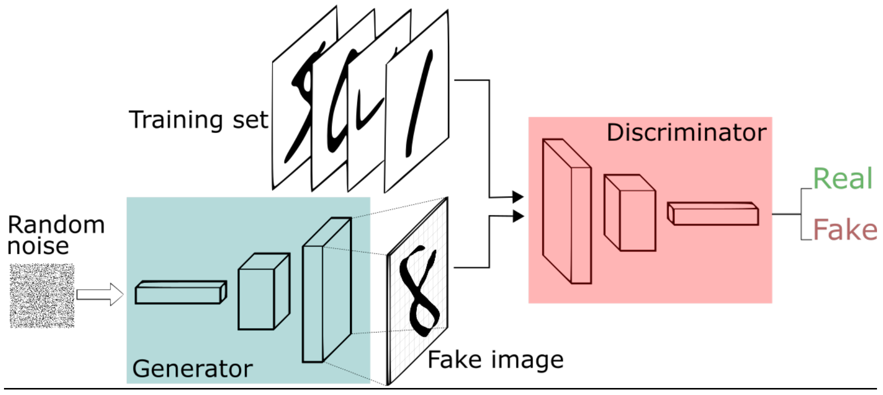
Next, we **train the discriminator.** Pass each of these 100-dimensional vectors through the generator; this gives us *B* "fake images". Now take your original batch of *B* "real" training images and concatenate them together with the fake images, giving you a tensor consisting of a total of 2*B* real/fake images. I recommend permuting the image indices of the real and fake images in this tensor so that the discriminator is trained on a random sequence of real/fake images (be careful that you permute only with respect to the image indices in the tensor batch and not the pixels of the images themselves).

With the parameters of the discriminator unfrozen (e.g., discriminator.trainable = True), train the discriminator on the batch consisting of 2*B* real and fake images. Ensure that the ground-truth labels are consistent during training, e.g., real images have label "1" and synthetic images have label "0", etc.

Finally, we **train the generator**. Freeze the weights of the discriminator (e.g., discriminator.trainable = False) for this step. Generate *B* noise samples from a standard 100-dimensional MVN; pass each of these noise vectors through the GAN model; the predicted score by the GAN should be compared with the ground-truth (which is "1" here because the generator is attempting to trick the discriminator, and so zero loss in incurred when the generator elicits a "real" label issued by the discriminator).

Report the training loss for each epoch for both the GAN and discriminator models. Every 10th epoch of training (including the first epoch), provide a small number of examples of synthetic images produced by the generator for random noise vectors.

**Step 3: CNN-based GAN and Discriminator**



Redo this exercise, but now use a CNN-based GAN and discriminator. In particular, your generator should utilize "Conv2DTranspose" functions for up-sampling – you can decide on additional architectural details including the filter and stride sizes, using of BN, etc. I recommend using a filters of size 5 with stride = 2; notice that the input the generator is now a tensor whose dimensions equal those of the penultimate layer of the discriminator sub-network (e.g., (7x7x128) which would be flattened to dimension 6,272).

For instance, if you are using two conv2D layers in the discriminator, the input to the discriminator should be of dimension (28x28x1), generating with two conv2D layers (filter = 5, stride = 2) a final set of feature maps of dimension 7x7 (suppose we have 128 such filters in this layer), yielding a feature tensor of dimension (7x7x128), which is flattened to a vector of dimension 6,272. Conversely, in the generator, we should include two corresponding Conv2DTranspose operations (filter = 5, stride = 2); the generator input dimension would be 6,272 which we reshape to (7x7x128); the final output of the generator is (28x28x1). Essentially, all other algorithm details for the CNN-based GAN will be equivalent to the instructions given in Steps 1-2 above.

Include a model summary with trainable parameter counts in your write-up. Report the training loss for each epoch for both the CNN-based GAN and discriminator models. Every 10th epoch of training (including the first epoch), provide a small number of examples of synthetic images produced by the generator for random noise vectors.

Note: If you are having trouble successfully training your GAN, here are several suggestions for improving performance: try switching to soft labels (0.1 and 0.9 instead of 0 and 1, respective); visualize the weights of the discriminator/generator – are they vanishing? Plot the losses of the discriminator vs. generator; try use a small learning rate; try adding a small amount of noise to the real and synthetic images when they are presented to the discriminator – this can force the discriminator to be more complex and prevent it from "cheating."

**Report:** Your report should include a short description of your experiments, along with the plots and discussion paragraphs requested above and any other relevant information to help shed light on your approach and results.

**Here is what you need to turn in:**

- Your report.
- Readable code.