Report: Variational Auto-Encoder for MNIST Data

In this project, we will implement a VAE for MNIST dataset. MNIST dataset includes grayscale(that is, images have only one color channel) 28x28 images of handwritten digits. In Figure 1, we can see 100 random images from train data. We expect that we get similar generated images after applying VAE. To make compare the results easy, we plot random images in MNIST data without applying any operation.

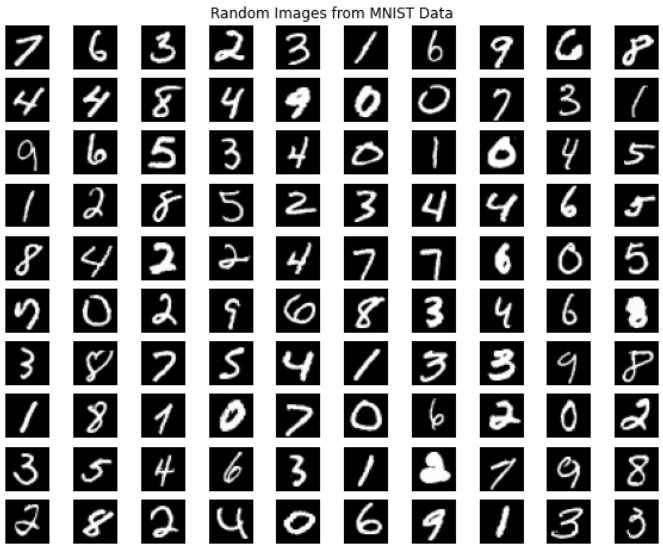

Figure 1: 100 Random Images From MNIST Train Data

Let briefly summarize variational auto-encoders. The difference between auto-encoder and variational auto-encoder is that VAE is a generative model. Auto-encoders compress and decompress data and they do not create new data. VAE enables to get a latent space by using the original data and thus, we obtain a sample. In the encoding part, we produce mean and variance values. We decode the sample according to these mean and variance values and we get new data.

VAE helps us to generate new data by encoding and decoding data. Here, we use one LSTM layer as decoder and transpose convolution layers as decoder. At first, we talk about encoder part. In the encoder part, we use 28 dimensional sequence having length of 28 for the LSTM layer since LSTM is a RNN architecture and LSTM uses for sequence data. Now, a little bit, we can talk about LSTM architecture because it is important to understand its outputs and their shapes. In pytorch, LSTM layer gives us a tensor including hidden state, a tensor including cell or internal state and output features. Also, in decoder part, we use transpose convolution layers because we generate new data from random vectors containing the outputs of encoding part. The convolutional layers helps us to learn features of the image data and the dimensions of the data decrease after the convolution operation. However, the transpose convolution layers enables us to increase dimensions; that is, we can say that the transpose convolution is opposite to convolution.

By using information the blog of Aggarwal (2020), we can say that the formula to calculate dimensions of the output of transpose convolution layer is:

$$\text{Output Size} = (\text{Input Size} - 1) * \text{Strides} + \text{Filter Size} - 2 * \text{Padding} + \text{Output Padding}$$

Now, we can talk about our architecture for VAE with LSTM encoder and CNN decoder. For a single LSTM layer, we use 32 as dimension of hidden state. We get 28x32 dimensional output. After the LSTM layer, we use ReLU activation function and fully connected layer with 32x256 dimensional linear vectors. Also, we try to train model with 32x16 dimensional fully connected layer. We will use to compare results. Then, we calculate mean and variance of the latent space and create a sample by using them. In the CNN decoder part, we use 3 transpose convolutional layers. The transpose convolutional layers contain transpose convolution operation, ReLU activation function, batch normalization, pooling and dropout. Before we talk about kernel size and number of kernels (or filters), we can say that we also try to train model without dropouts. Therefore, we can compare generated samples.

We have 28x256 dimensional outputs from encoder part and it has 2 dimensions but we need 3 dimension with channel, height and width of the image. Then, we reshape this output as 256x28x1. In other words, we can think we have 28x1 images with 256 channels. Here, 256 is number of channels because in pytorch, the input image have the form of channels x height x width. At the end of the decoder part, we should 1x28x28 images. ("1" says they are gray-scale images.) First transpose convolution operation has 256 input channels since it is dimension of encoder output and 128 5x5 filters; that is, 5x5x128 dimensional filter is used with stride 2 for height of the image and stride 1 for width of the image. Therefore, we get 128x86x5 vectors. We use ReLU activation function and batch normalization. These operations does not change dimensions. Also, we have pooling layer with max-pooling. Here, we use 3x2 kernel and we obtain 128x28x2 vectors. Also, we apply dropout with the probability 0.01. It also does not change dimension of the output. Therefore, we can say that we have 128x28x2 outputs after first convolution layer.

Second transpose convolution layer has the same operations with first layer's. In the second layer, we use 128 5x5 filters for the transpose convolution operation with the same strides with first layer and then, we get 64x86x6 vectors. After applying ReLU activation function, batch normalization and pooling with 3x2 kernel, we have 64x28x3 outputs. After we apply dropout with the same probability in the first layer, we can pass third transpose convolution layer. In this layer, we use 1x5x5 kernel. Here, we use 1 as channel because we want to generate gray-scale images at the end of the VAE model. We get 1x86x7 vectors. After the same operations like pooling, ReLU and batch normalization, we get 1x28x3 vectors. Then, we apply a fully connected layer with 3x28 linear function. Therefore, we obtain 1x28x28 reconstruction images after applying these operations. Anymore, we are ready to discuss our results.

Table1: VAE Results for Epochs

Epoch:1/50, Train_Loss:0.2699808180332184, Train_Reg_Term:0.01405326183885336
Epoch:3/50, Train_Loss:0.2328605949878926, Train_Reg_Term:0.01844196766614914
Epoch:5/50, Train_Loss:0.2101295441389084, Train_Reg_Term:0.022014062851667404
Epoch:7/50, Train_Loss:0.19987408816814423, Train_Reg_Term:0.02239052951335907
Epoch:9/50, Train_Loss:0.19428744912147522, Train_Reg_Term:0.022439904510974884
Epoch:11/50, Train_Loss:0.18660761415958405, Train_Reg_Term:0.022746892645955086
Epoch:13/50, Train_Loss:0.17839045822620392, Train_Reg_Term:0.022831201553344727
Epoch:15/50, Train_Loss:0.17468763887882233, Train_Reg_Term:0.022949425503611565
Epoch:17/50, Train_Loss:0.1827358901500702, Train_Reg_Term:0.022359220311045647
Epoch:19/50, Train_Loss:0.17179976403713226, Train_Reg_Term:0.02230830490589142
Epoch:21/50, Train_Loss:0.16798606514930725, Train_Reg_Term:0.022184187546372414
Epoch:23/50, Train_Loss:0.16492648422718048, Train_Reg_Term:0.021665574982762337
Epoch:25/50, Train_Loss:0.16751641035079956, Train_Reg_Term:0.021801074966788292
Epoch:27/50, Train_Loss:0.16398532688617706, Train_Reg_Term:0.021657707169651985
Epoch:29/50, Train_Loss:0.16510458290576935, Train_Reg_Term:0.021368449553847313
Epoch:31/50, Train_Loss:0.16255581378936768, Train_Reg_Term:0.021654251962900162
Epoch:33/50, Train_Loss:0.16123589873313904, Train_Reg_Term:0.021236201748251915
Epoch:35/50, Train_Loss:0.16076241433620453, Train_Reg_Term:0.021205490455031395
Epoch:37/50, Train_Loss:0.1578901708126068, Train_Reg_Term:0.02118915319442749
Epoch:39/50, Train_Loss:0.16217315196990967, Train_Reg_Term:0.02125754952430725
Epoch:41/50, Train_Loss:0.16033892333507538, Train_Reg_Term:0.021140186116099358
Epoch:43/50, Train_Loss:0.15823622047901154, Train_Reg_Term:0.021100422367453575
Epoch:45/50, Train_Loss:0.16266587376594543, Train_Reg_Term:0.02103406935930252
Epoch:47/50, Train_Loss:0.16424018144607544, Train_Reg_Term:0.021063359454274178
Epoch:49/50, Train_Loss:0.1622491180896759, Train_Reg_Term:0.021074645221233368

For training the model, we can use google colaboratory to use GPU. Therefore, the model training spent about only 5 minutes. Our learning rate is 0.001 and Adam optimizer is selected. Also, batch size is 100 and number of epochs is 50. Here, we can use another hyperparameters, of course. Actually, we understand which one is the best by working with different numbers. Through GPU and the training time it provides, we can try different numbers and select them by comparing results. To decide hyperparameters, we should be careful about preventing overfitting.

In Table 1, we can see that the values of loss function and regularization term. Here, we have to say what our loss function is and how to decide regularization term. We use binary cross entropy loss function and KL divergence for regularization term. At the end, our loss value will be addition of binary cross entropy loss and regularization term. Regularization term with KL divergence will enable to prevent overfitting for the training process. We use built-in functions from pytorch. We need to λ value for regularization term and we select it as 0.5. In one of the forums (the last reference), we can see that the regularization term with KL divergence is the mean of $1 + log(variance) - (mean)^2 - variance$. Mean and variance values are calculated in the end of encoder part.

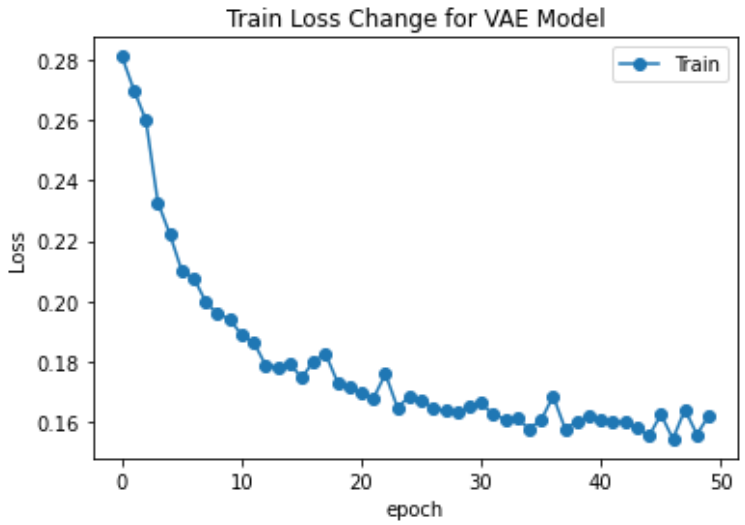

Figure 2: Loss Change During Training

In Figure 2, we can see the loss change during training. Here, we may refer to Quiz 2. In one of the questions, we want to find the difference between stochastic gradient descent, batch gradient descent and mini batch gradient descent in terms of loss changes during epochs. Now, we know that in batch training, the loss can increase and decrease with small changes. Therefore, we can say that fluctuations in Figure 2 are normal here. We also see that our loss tends to decrease during training. Also, we see that in Table 1, the loss after 50 epochs is 0.1622491180896759 and the regularization term is 0.021074645221233368. In the beginning of the model training, our loss is 0.2699808180332184 and regularization term with KL divergence is 0.01405326183885336. Our results are good in terms of loss change and regularization term because they are in balance and with the regularization term, the loss continues to decrease until 50 epochs training.

Also, we can see the change of regularization terms during training for each epoch in Figure 3. Regularization term helps avoiding overfitting. We talk about regularization term property above. In fact, regularization with KL divergence allows us to better model performance. In the study of Yu et al (2013), we can figure out positive effect of KL divergence regularization. In this paper, the researchers compare model performances with regularization and without regularization. They refer that the change of the target distribution is equivalent to addition of this regularization term. So, the regularization with KL divergence gives notice about the distributions because we use mean and variance values to get sample. In Figure 3, the regularization term tends to increase and after 10 epochs it starts to reduce.

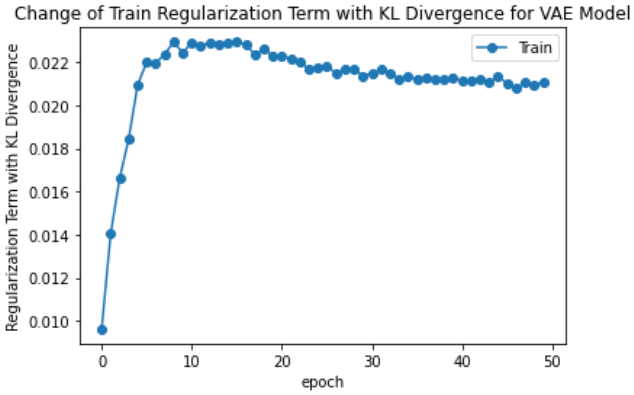

Figure 3: Regularization Term Change During Training

Before looking at generating images from random vectors, we can see the model summary in Table 2. The only difference between them is fully connected layer size in encoder part and dropouts in encoder part. Although this seems like a small change in the models, when we visualize the model results as 100 random images, we will see more clearly how very different the results are.

Table2: Summary of Models and Their Comparison

		Model 1 (Our Best Model)	Model 2
	Encoder Part	$a_1 = LSTM(l, i, h_1, n)$ $a_2 = ReLU(a_1)$ $a_3 = Fully\ Connected(a_2, h_4)$ $a_4 = Sampling(a_3)$ $a_1 \rightarrow 28x32$ $a_2 \rightarrow 28x32$ $a_3 \rightarrow 28x256$ $a_4 \rightarrow 256x28x1$	$a_1 = LSTM(l, i, h_1, n)$ $a_2 = ReLU(a_1)$ $a_3 = Fully\ Connected(a_2, h_5)$ $a_4 = Sampling(a_3)$ $a_1 \rightarrow 28x32$ $a_2 \rightarrow 28x32$ $a_3 \rightarrow 28x16$ $a_4 \rightarrow 16x28x1$
sequence length = l = 28 input size = i = 28 num layers = n = 1 hidden size1 = h_1 = 32 hidden size2 = h_2 = 64 hidden size3 = h_3 = 128 hidden size4 = h_4 = 256 hidden size4 = h_5 = 16 strides = s = (3,1) kernel size = k = 5x5 pooling kernel = p = (3,2) dropout prob. = pr = 0.01 batch size = 100 number of epochs = 50 learning rate = 0.001	Decoder Part	***Layer1*** $a_5 = TranposeConv(a_4, h_4, h_3, k, s)$ $a_6 = ReLU(a_5)$ $a_7 = Batch\ Normalization(a_6)$ $a_8 = Pooling(a_7, p)$ $a_9 = Dropout(a_8, pr)$ ***Layer2*** $a_{10} = TranposeConv(a_9, h_3, h_2, k, s)$ $a_{11} = ReLU(a_{10})$ $a_{12} = Batch\ Normalization(a_{11})$ $a_{13} = Pooling(a_{12}, p)$ $a_{14} = Dropout(a_{13}, pr)$ ***Layer3*** $a_{15} = TranposeConv(a_{14}, h_3, 1, k, s)$ $a_{16} = ReLU(a_{15})$ $a_{17} = Batch\ Normalization(a_{16})$ $a_{18} = Pooling(a_{17}, p)$ $a_{19} = Dropout(a_{18}, pr)$ $a_{20} = Fully\ Connected(a_{19}, 28)$ $a_9 \rightarrow 128x28x2$ $a_{14} \rightarrow 64x28x3$ $a_{19} \rightarrow 1x28x3$ $a_{20} \rightarrow 1x28x28$	***Layer1*** $a_5 = TranposeConv(a_4, h_5, h_3, k, s)$ $a_6 = ReLU(a_5)$ $a_7 = Batch\ Normalization(a_6)$ $a_8 = Pooling(a_7, p)$ ***Layer2*** $a_9 = TranposeConv(a_8, h_3, h_2, k, s)$ $a_{10} = ReLU(a_9)$ $a_{11} = Batch\ Normalization(a_{10})$ $a_{12} = Pooling(a_{11}, p)$ ***Layer3*** $a_{13} = TranposeConv(a_{12}, h_3, 1, k, s)$ $a_{14} = ReLU(a_{13})$ $a_{15} = Batch\ Normalization(a_{14})$ $a_{16} = Pooling(a_{15}, p)$ $a_{17} = Fully\ Connected(a_{16}, 28)$ $a_8 \rightarrow 128x28x2$ $a_{12} \rightarrow 64x28x3$ $a_{16} \rightarrow 1x28x3$ $a_{17} \rightarrow 1x28x28$

In Figure 4, we can see Model 1 generated images and in Figure 5, we also see Model 2 generated images from randomly selected from 100 vectors after decoder part. These results are very different. We cannot say the random images in Figure 5 look like MNIST data. However, in Figure 4, we notice that all handwritten digits are generated. During the training, we save reconstruction images as numpy arrays and we concatenate them at the end of the training. We have 60000 reconstruction images after the VAE model run during 50 epochs. The generated images belong to fiftieth epoch. At the beginning of the training, the generated images are not starting to form fully because the model has not learned enough yet. That's why we save only the results at the end of training. In Figure 4, we can say that these 100 random images look like good for image generation for MNIST. While trying to get the best model, some models generate irrelevant images as in Figure 5. At the end, the model 1 is able to succeed to produce images as in MNIST data.

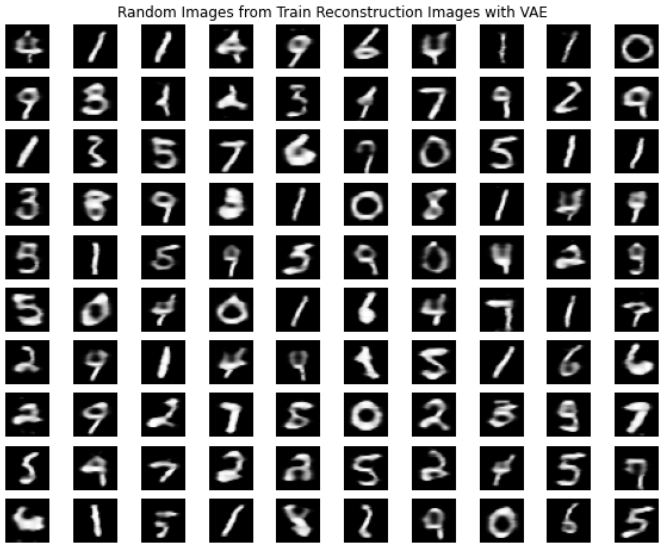

Figure 4: Generated images which are randomly selected from outputs of VAE Model(Our Best Model) with 32x256 size of fully connected layer in encoder part and dropout in decoder part

Figure 5: Generated images which are randomly selected from outputs of VAE Model with 32x16 size of fully connected layer in encoder part and without dropout in decoder part

References

- Aggarwal, K. (2020). Transpose Convolution Explained for Up-Sampling Images. Retrived from https://blog.paperspace.com/transpose-convolution/
- Buwaneswaran, M. (2021). Implementation Differences in LSTM Layers: TensorFlow vs PyTorch. Retrived from https://towardsdatascience.com/implementation-differences-in-lstm-layers-tensorflow-vs-pytorch-77a31d742f74
- Cloudera Fast Forward Labs (2016). Introducing Variational Autoencoders (in Prose and Code). Retrived from https://blog.fastforwardlabs.com/2016/08/12/introducing-variational-autoencoders-in-prose-and-code.html
- Dilokthanakul, N., Mediano, P. A., Garnelo, M., Lee, M. C., Salimbeni, H., Arulkumaran, K., & Shanahan, M. (2016). Deep unsupervised clustering with gaussian mixture variational autoencoders. arXiv preprint arXiv:1611.02648.
- Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). Deep learning (Vol. 1, No. 2). Cambridge: MIT press.
- Kingma, D. P., & Welling, M. (2013). Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114.
- Mohr, F. (2017). Teaching a Variational Autoencoder (VAE) to draw MNIST characters. Retrived from https://towardsdatascience.com/teaching-a-variational-autoencoder-vae-to-draw-mnist-characters-978675c95776
- Nutan (2021). PyTorch Recurrent Neural Networks With MNIST Dataset. Retrived from https://medium.com/@nutanbhogendrasharma/pytorch-recurrent-neural-networks-with-mnist-dataset-2195033b540f
- Ranjan, C. (2019). Step-by-step understanding LSTM Autoencoder layers. Retrived from https://towardsdatascience.com/step-by-step-understanding-lstm-autoencoder-layers-ffab055b6352
- Pu, Y., Gan, Z., Henao, R., Yuan, X., Li, C., Stevens, A., & Carin, L. (2016). Variational autoencoder for deep learning of images, labels and captions. arXiv preprint arXiv:1609.08976.
- Yu, D., Yao, K., Su, H., Li, G., & Seide, F. (2013). KL-divergence regularized deep neural network adaptation for improved large vocabulary speech recognition. In 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (pp. 7893-7897). IEEE.
- https://github.com/dragen1860/pytorch-mnist-vae
- https://github.com/lyeoni/pytorch-mnist-VAE/blob/master/pytorch-mnist-VAE.ipynb
- Rath, S. R. (2020). Convolutional Variational Autoencoder in PyTorch on MNIST Dataset. Retrived from https://debuggercafe.com/convolutional-variational-autoencoder-in-pytorch-on-mnist-dataset/
- https://fairseq.readthedocs.io/en/latest/tutorial_simple_lstm.html
- https://forums.fast.ai/t/intuition-behind-kl-divergence-regularization-in-vaes/1650