1. Autoencoder with Numpy

Autoencoder is an unsupervised technique that learns how to efficiently compress and encode data and then learns how to reconstruct the data back from the reduced encoded representation to a representation that is close to the original input as possible. Autoencoders and neural networks are similar in terms of layers used in the bottleneck part, but the most important difference is that autoencoders set the target values to be equal to the inputs. In other words, they are trying to learn an approximation to the identity function. Additionally, Autoencoders are learned automatically from data example. Also, they can be used in many different areas. For example, data denoising, dimension reduction, image generation, anomaly detection, feature extraction etc.

Autoencoder architecture has 4 main parts which are encoder, bottleneck, decoder, and reconstruction loss. In encoder part, model learns how to reduce the input dimensions and compress the input data into an encoded representation. In bottleneck part, layers occur. Layers contains the compressed representation of the input data. This part is the lowest possible dimensions of the input data. In the decoder part, model learns how to reconstruct the data from the encoded representation to be as close to the original input as possible. In the last part of the architecture, measures how well the decoder is performing and how close the output is to the original input.

There are several Autoencoder types. First type that I want to mention is Vanilla autoencoder. It is the simplest form that has three layers. Input and output are the same and model learns how to reconstruct the input. Second type is Multilayer autoencoder. It has more than one hidden layer. Third type is Convolutional autoencoder. In this type convolutional layers are used instead of fully connected layers. Other type is Regularized autoencoder. Rather than limiting the model capacity by keeping the encoder and decoder shallow and the code small, regularized autoencoders use a loss function that encourages the model to have other properties besides the ability to copy its input to its output. Next is Sparse autoencoder which are typically used to learn features for another task such as classification. An autoencoder that has been regularized to be sparse must respond to unique statistical features of the dataset it has been trained on, rather than simply acting as an identity function. In this way, training to perform the copying task with a sparsity penalty can yield a model that has learned useful features as a byproduct. The last type that I want to mention is denoising autoencoder. Rather than adding a penalty to the loss function, it can be obtained an autoencoder that learns something useful by changing the reconstruction error term of the loss function. This can be done by adding some noise of the input image and make the autoencoder learn to remove it. By this means, the encoder will extract the most important features and learn a robust representation of the data.

In the notebook, I implement a simple autoencoder for handwritten digit generation. First think that I was confused is what is the shape of w2. Then I remember that was an autoencoder network. So I figure it out in a short time. Implementation of relu, sigmoid, backward relu and backward sigmoid was easy. The most confusing part is to implement backpropagation. Taking which value's transpose was hard for me. But with pen&paper experiments I figured it out what are the variables right hand side. In the result

I get incorrect results. In the Figure 1 and Figure 2 you can see that model were not work very well. Indeed, model is wrong. And Figure 3, you can see model's overfit result.

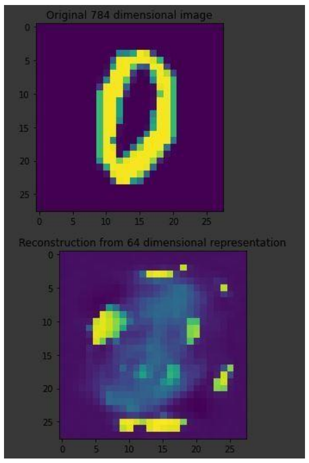


Figure 1: incorrect model result

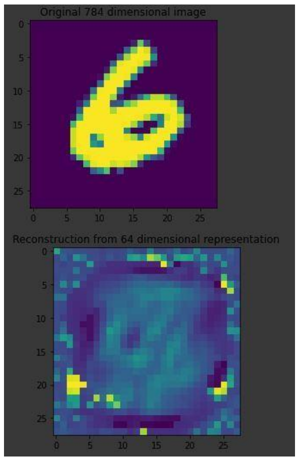


Figure 2: incorrect model result

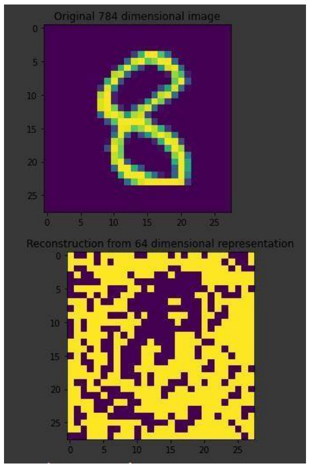


Figure 3: incorrect model result. (overfit)

In my next attempt in backward part, I think there was a mistake in code, for specification first D_o1 variable part is incorrect. I mean, it is not necessary for the model. Then I implement without it. So, results are now seeming good. I think model is right. In Figure4 and Figure 5 you can see my results. I train my model with 20 epochs. My loss is 0.0104.

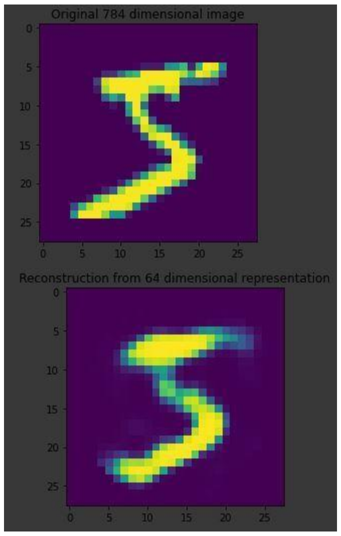


Figure 4: Model result with digit 5.

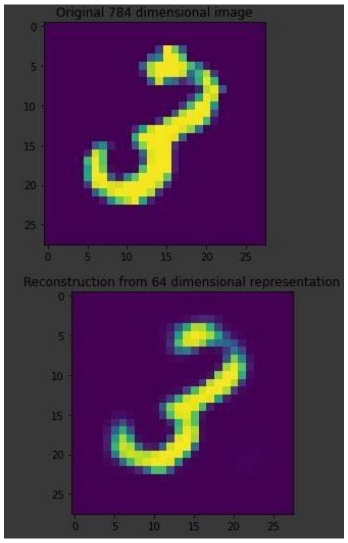


Figure 5: Model result with digit 3.

Moreover, as it can be seen in the Figure 1, Figure 2 and Figure 3, inputs and outputs are not even close. But in Figure 4 and Figure 5 you can see that inputs and outputs have a good match.

2. Generative Adversarial Network for MNIST Digit Generation

A generative adversarial network (GAN) is a class of machine learning frameworks designed by Ian Goodfellow and his colleagues in 2014. Given a training set, this technique learns to generate new data with the same statistics as the training set. For example, a GAN trained on photographs can generate new photographs that look at least superficially authentic to human observers, having many realistic characteristics. Though originally proposed as a form of generative model for unsupervised learning, GANs have also proven useful for semi-supervised learning, fully supervised learning, and reinforcement learning. The generative network generates candidates while the discriminative network evaluates them. GAN mainly focuses generating data from scratch, most of the data contains images but music, text also have been working. Moreover, GANs are using in reinforcement learning in which helping the robot to learn much faster. As I mentioned before, GAN has two main networks, generator, and discriminator. Generator learns to generate plausible data. The generated instances become negative training examples for the discriminator. On the other hand, discriminator learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results. Both generator and discriminator are deep networks that can be applied convolutional, recurrent, or fully connected layer.

After giving brief explanation about Generative Adversarial Network, I want to mention two different GAN types which are DCGAN and CycleGAN. Firstly, DCGAN, Deep Convolutional GAN, is an extension of GAN but except that it explicitly uses convolutional and convolutional-transpose layers in the discriminator and generator, respectively. Generator model consists of Deconvolution layers or more accurately, Transposed Convolution layers that basically perform the reverse of a convolution operation. The discriminator is nothing but a binary classifier that consists of several convolution layers (like any other image classification task). And finally, the flattened activation maps mapped to a probability output to predict if the image is real or fake. Also, DCGAN was created by Alec Radford, Luke Metz and Soumith Chintala in 2016. Secondly, CycleGAN is a very popular GAN architecture primarily being used to learn transformation between images of different styles. The CycleGAN is a technique that involves the automatic training of image-to-image translation models without paired examples. The models are trained in an unsupervised manner using a collection of images from the source and target domain that do not need to be related in any way(unpaired). It was introduced in the now well-known 2017 paper out of Berkeley. Since paired data is harder to find in most domains, and not even possible in some, the unsupervised training capabilities of CycleGAN are quite useful. CycleGAN was created Ian Goodfellow.

As an optimizer I use Adam and Adamax optimizer. I use binary cross entropy loss. Also, I tried learning rates like 0.01, 0.001 and 0.00001 in order to find a better result. Moreover, I train the network with 20, 30 and 50 epochs. I did not have any difficulties in network implementation part. However, I struggled in cpu and gpu problems and visualizer problems but I figured it out easily. I also easly did the loss and optimizer defining parts. On the other hand, training part is really hard for me, I had a complex time. Firstly, I fixed cuda problem. In the training firsly I try to labels in the network and forming the network according to the labels was hard and unnecessary effort me. While striving the problems in the wrong network, I faced with backward implementation problem. In the gradients errors occurred and that was the background implementation of the pytorch. I understand what happened while zero_grad() is working. Also I tried the retain_graph=True in order to solve gradient problems. But finally, I understand that my network implementation completely wrong. I implemented generator and discriminator training part at the beginning. I deleted labels part. Then I added tensor with ones and zeros in order to compute losses correctly. Also, I added generator result in the discriminator training in order to recalculate the gradients for solving zero_grad's problem in the discriminative part. I edited generator loss. So, my network is working.

In the Figure 6, Figure 7 and Figure 8 you can see my model generative outputs with respectively learning rate = 0.1, 0.001, 0.00001 and epoch = 10, 20, 20. Also in the Figure 9 you can see my model which has epoch =50 and learning rate = 0.0001 first epoch result. The purpose for adding Figure 9 is to see how differ the generative model outputs in the epochs.

As you can see in the Figure 6, model has mode collapse problem. All the outputs are the same. I trained the model in more epoch values, but results were not change. In the models which are epoch=20 & learning_rate=0.001 and epoch=50 & learning_rate=0.00001, model outputs seems similar. Moreover, model with learning rate = 0.001 seems more qualified. Model results diversities are again similar except Figure 6's model. Comparison may not seem accurate in terms of epoch values are different but I observe that after 20 epochs change of the model outputs are not different completely. So I decide to put here in different epochs.

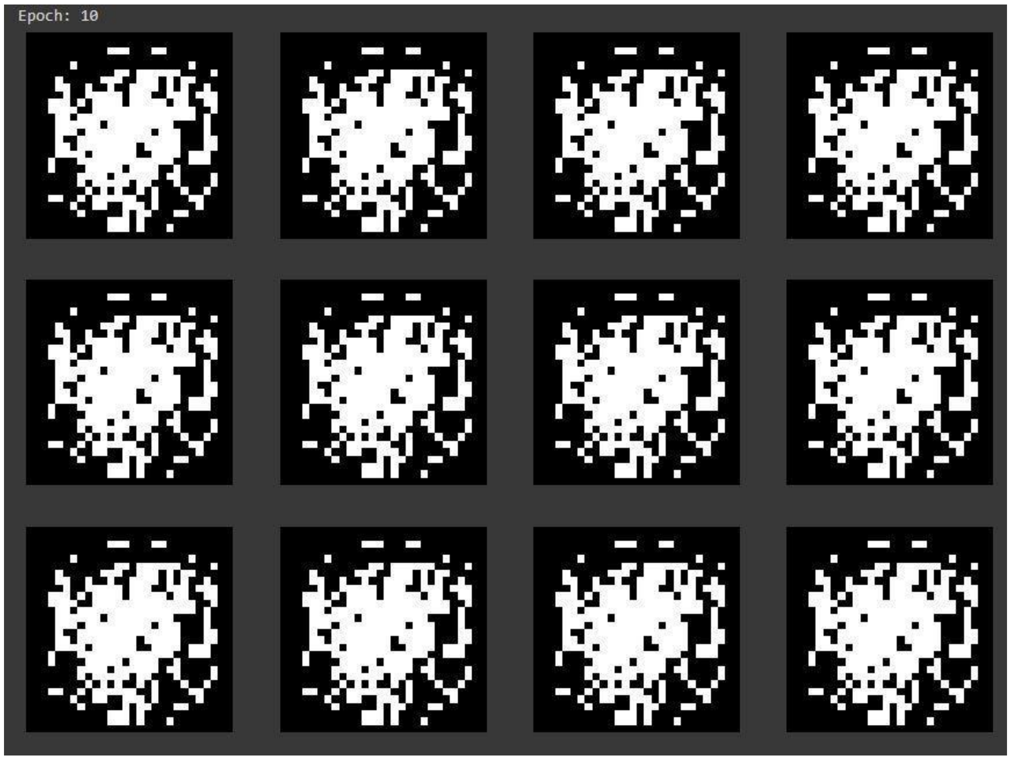


Figure 6: Epoch= 10, learning rate = 0.1 generative model output

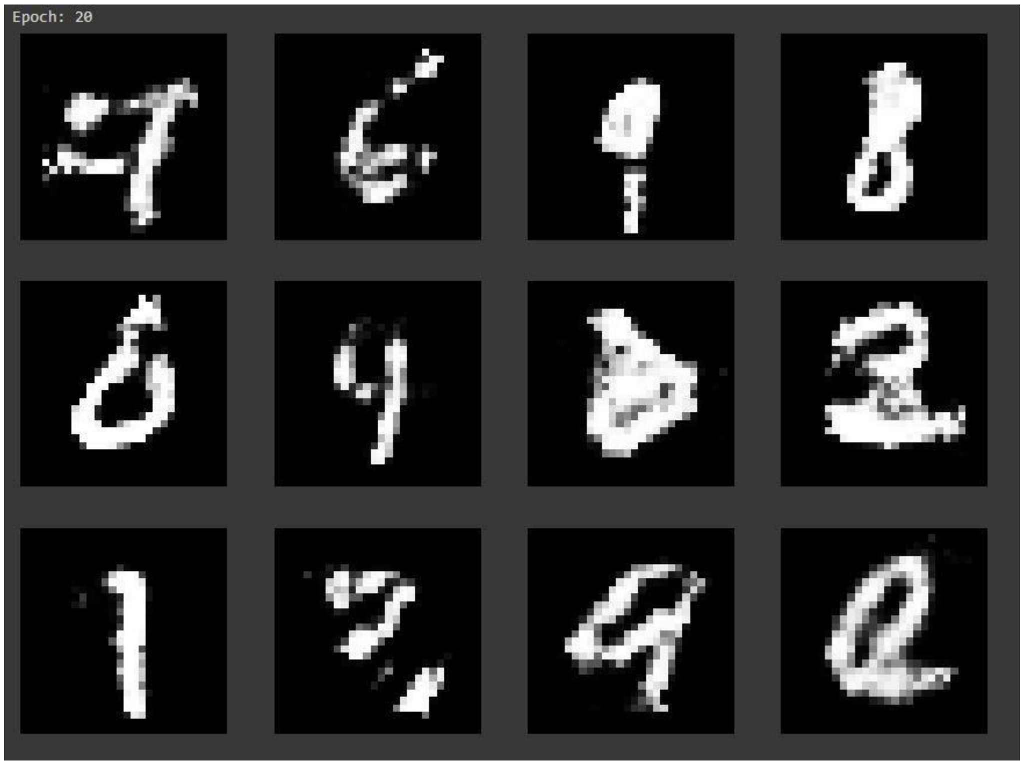


Figure 7: Epoch=20, learning rate = 0.001 generative model output.

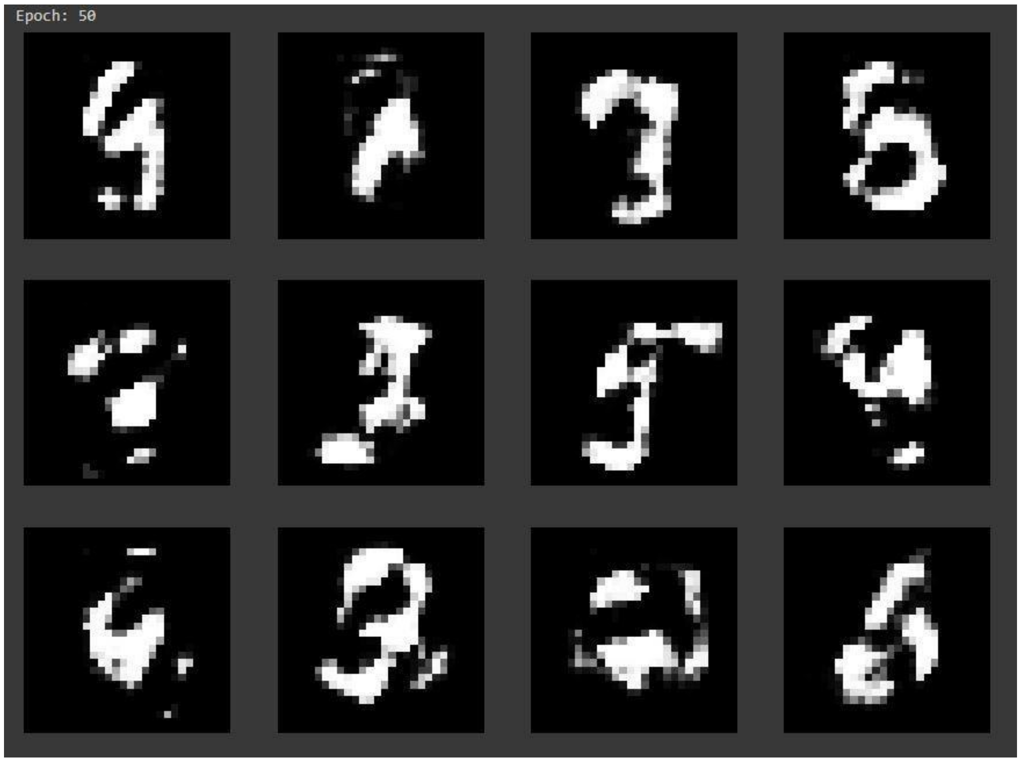


Figure 8: Epoch = 50, learning rate = 0.00001 generative model output.

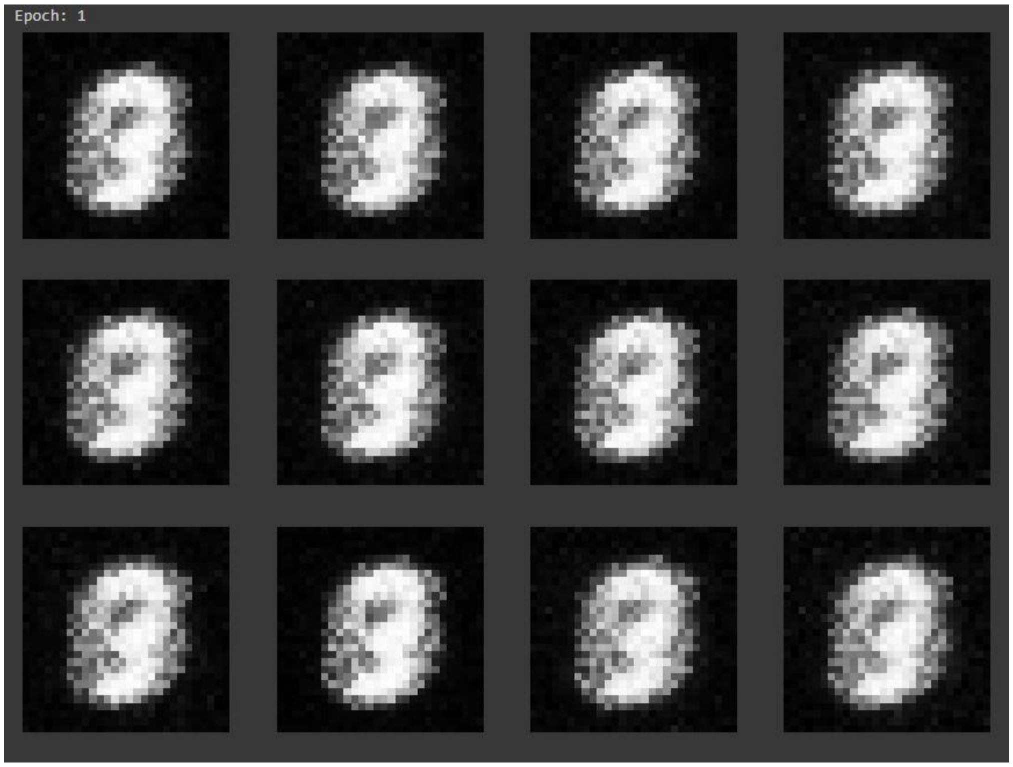


Figure 9: epoch=1, learning rate = 0.00001 generative model result.

3. Text Generation with Recurrent Neural Networks

• Recurrent Neural Network (RNN)

Recurrent Neural Network was developed by John Hopfield in 1982. RNN is a generalization of feedforward neural network that has an internal memory. RNN is recurrent in nature as it performs the same function for every input of data while the output of the current input depends on the past one computation. After producing the output, it is copied and sent back into the recurrent network. For making a decision, it considers the current input and the output that it has learned from the previous input. Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs. This makes them applicable to tasks such as unsegmented, connected handwriting recognition or speech recognition. In other neural networks, all the inputs are independent of each other. But in RNN, all the inputs are related to each other. There are several positive properties that RNN has. One of them is that RNN can model sequence of data so that each sample can be assumed to be dependent on previous ones. Other is Recurrent neural network are even used with convolutional layers to extend the effective pixel neighbourhood. On the other hand, although RNN has positive properties, it has negatives also. Firstly, it has a huge likely to have vanishing gradients problems. Also, It cannot process very long sequences if using tanh or relu as an activation function and training RNN is very hard.

• Gated Recurrent Unit (GRU)

Gated Recurrent Unit was introduced by a team included Yoshua Bengio and it was created to solve a problem that RNN has, which is vanishing gradient problem. GRU can also be considered as a variation on the LSTM because both are designed similarly and, in some cases, produce equally excellent results. For the deeper explanation of the vanishing gradient problem's solution, GRU uses update gate and reset gate. Basically, these are two vectors which decide what information should be passed to the output. The special thing about them is that they can be trained to keep information from long ago, without washing it through time or remove information which is irrelevant to the prediction. Moreover, the update gate helps the model to determine how much of the past information (from previous time steps) needs to be passed along to the future and reset gate is used from the model to decide how much of the past information to forget.

• Long Short-Term Memory (LSTM)

Long Short-Term Memory is a kind of Recurrent Neural Network. It was created in 1995 but current model was developed in 2013 by Kyunghyun Cho. LSTM networks are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series. LSTMs were developed to deal with the vanishing gradient problem that can be encountered when training traditional RNNs. Relative insensitivity to gap length is an advantage of LSTM over RNNs, hidden Markov models and other sequence learning methods in numerous applications. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell.

Let's now explore what the distinction between the design of those RNNs is. In the recurrent layer, all RNNs have feedback loops. This helps them to retain data in 'memory' over time. But it can be hard to train standard RNNs to solve problems requiring long-term temporal dependencies to be trained. This is because, over time, the gradient of the loss function decreases exponentially (called the vanishing gradient problem). LSTM networks are a type of RNN which, in addition to standard units, uses special units. LSTM units have a 'memory cell' that can store data for long periods of time in memory. To monitor when information enters the memory, when it's output, and when it's forgotten, a series of gates is used. This nature helps them to consider longer-term dependencies. GRUs are similar, but use a simpler structure, to LSTMs. To control the flow of information, they also use a series of gates, but they do not use separate memory cells and use fewer gates.

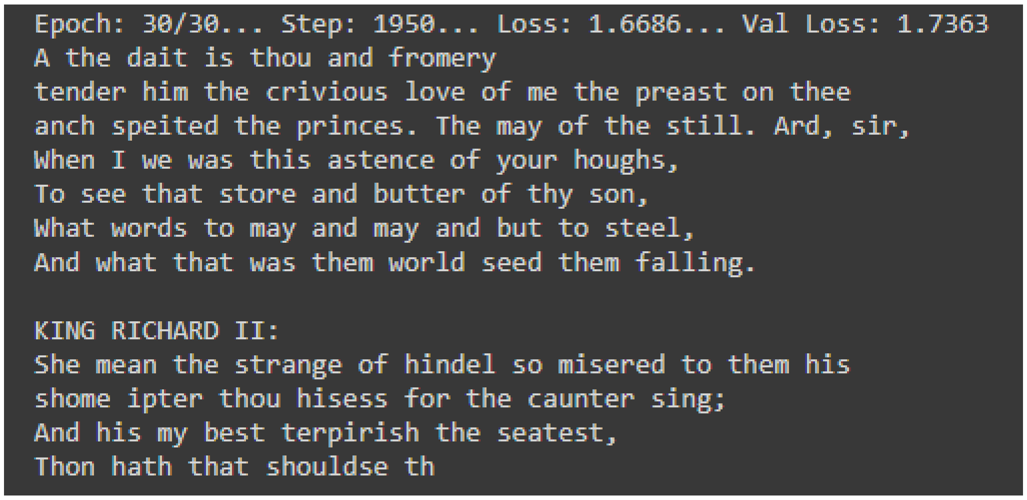
• Autoregressive Generative Models

Autoregressive models are another kind of deep generative model with tractable likelihoods. AR generative models implicitly define a distribution over sequences using the Chain Rule for Conditional Probability, whereby in each step the distribution of the next sequence element is predicted given the previous elements. The main autoregressive architectures are RNNs and causal conv nets. Important examples include PixelRNN, PixelCNN, and WaveNet. Autoregressive models are powerful density estimator but sequential generation can be slow, does not closely reflect the true generating process, tends to emphasize details over global data and not very good for learning representations.

Non-autoregressive models are just the opposite of autoregressive models. They work as a parallel sequence. Most sequence-to-sequence (seq2seq) models are autoregressive; they produce each token by conditioning on already created tokens. In differentiate, non-autoregressive seq2seq models produce all tokens in one pass, which leads to expanded proficiency through parallel handling on equipment such as GPUs. Be that as it may, straightforwardly modeling the joint dissemination of all tokens at the same time is challenging, and indeed with progressively complex demonstrate structures precision slacks essentially behind autoregressive models. In this paper, we propose a basic, proficient, and successful demonstrate for non-autoregressive grouping era utilizing inactive variable models.

BONUS

In this code, firstly, I downloaded one of the Shakespeare plays dataset. And then, I prepared the dataset for the text generation. Also, I divided the dataset into mini-batches since training model with mini-batches is more comfortable since it requires less computational power. Then I trained the model with LSTM with sequence length = 120, batch size = 128, hidden layer = 512, number of layers = 3, number of epochs = 30, learning rate = 0.002, dropout prob = 0.3 hyperparameters. In Figure 10, you can see my model results.



```
Epoch: 30/30... Step: 1950... Loss: 1.6686... Val Loss: 1.7363
A the dait is thou and fromery
tender him the crivious love of me the preast on thee
anch speited the princes. The may of the still. Ard, sir,
When I we was this astence of your houghs,
To see that store and butter of thy son,
What words to may and may and but to steel,
And what that was them world seed them falling.

KING RICHARD II:
She mean the strange of hindel so misered to them his
shome ipter thou hisess for the caunter sing;
And his my best terpirish the seatest,
Thon hath that shouldse th
```

Figure 10: NLP text generation model result.

Note: Code is taken from https://github.com/albertlai431/Machine-Learning/blob/master/Text%20Generation/Shakespeare.py. I only changed the hyperparameters and was trying to understand the code and implementation.

Talya TÜMER