Assignment 01

Deep Learning

- Nitesh Gupta
- R00195231

Question_1_1_1

The below method is implemented in this task. The description of each method is written below-

```
def forward_pass(variables, x):
    # forward propagation step
    layer_1_Z = tf.matmul(x, variables["layer_1_weights"]) +
variables["layer_1_bias"]  # layer 1 Z calculation
    layer_1_activation = tf.keras.activations.relu(layer_1_Z)  # Activation
of layer 1

    layer_2_Z = tf.matmul(layer_1_activation, variables["layer_2_weights"]) +
variables[
        "layer_2_bias"]  # layer 2 z calculation
    predictedY = softmax(layer_2_Z)
    return predictedY
```

Description - Forward pass is the method in which data flow in the neural network. As you know, there are two layers. In the above code, we push our training tensor data through our model. we use a matrix multiple (tf.matmul) to multiply our weights by the training data and add the bias. We subsequently push the results through the SoftMax activation function

```
def cross_entropy(Y, predictedY):
    loss = -tf.reduce_sum(Y * tf.math.log(predictedY), axis=1)
    return loss
```

Description - we push out data through the model and store predicted values. Now we want to calculate our loss (actual – predicted). Since one hot encoder is used in the target column, therefore, CategoryCrossEntropyLoss function should be used due to categorized comparison and we implemented using low-level TensorFlow functions.

Used Formula – sum (Y * log(predY)) for each example, this value comes negative therefore, we use negation sign before this.

```
def calculate_accuracy(y, predicted_y):
    # round the predictions by the logistical unit to either 1 or 0
    predictions = tf.round(predicted_y)

    # tf.equal will return a Boolean array : True if prediction correct,
False otherwise
    # tf.cast converts the resulting Boolean array to a numerical array
    # if True (correct prediction),0 if False( incorrect prediction)
```

```
    predictions_correct = tf.cast(tf.equal(predictions, y), dtype=tf.float32)
    # Finally, we just determine the mean value of predictions_correct
    accuracy = tf.reduce_mean(predictions_correct)
    return accuracy
```

Description - In this code, we calculate the accuracy of our model given the predicted values and true class labels. First, we took round off the predictions by the logistical unit to either 1 or 0. tf.equal will return a Boolean array. True if the prediction correct and otherwise the output would be False. tf.cast converts the resulting Boolean array to a numerical array. Finally, we just determine the mean value of predictions.

```
def gradient_loop(variables, X_train, Y_train, X_val, Y_val, X_test, Y_test,
num_Iterations):
    # Iterate our training loop
    train_accuracy = []
    testAccuracy = []
    val_accuracy = []
    train_loss = []
    val_loss = []
    for i in range(num_Iterations):
        # Create an instance of GradientTape to monitor the forward pass
        # and calculate the gradients for each of the variables m and c
        with tf.GradientTape() as tape:
            predictedY = forward_pass(variables, x=X_train)
            currentLoss = loss_func(Y_train, predictedY)
            gradients = tape.gradient(currentLoss, variables.values())
            accuracy = calculate_accuracy(Y_train, predictedY)
            # print("Iteration", i, ":Loss=", currentLoss.numpy()[-1],
"Acc:", accuracy.numpy() * 100)
            train_accuracy.append(accuracy.numpy() * 100)
            train_loss.append(currentLoss.numpy()[-1])
            adam_optimizer.apply_gradients(
                zip(gradients, variables.values()))
            predictedY_val = forward_pass(variables, x=X_val)
            currentLoss = loss_func(Y_val, predictedY_val)
            Val_accuracy = calculate_accuracy(Y_val, predictedY_val)
            # print("Iteration", i, ":Loss=", currentLoss.numpy()[-1],
"Acc:", accuracy.numpy() * 100)
            val_loss.append(currentLoss.numpy()[-1])
            val_accuracy.append(Val_accuracy.numpy() * 100)
            predictedY = forward_pass(variables, X_test)
            test_accuracy = calculate_accuracy(Y_test, predictedY)
            # print("TestAccuracy:", test_accuracy.numpy() * 100)
            testAccuracy.append(test_accuracy.numpy() * 100)
    return train_accuracy, val_accuracy, testAccuracy, train_loss, val_loss
```

Description - In the above code, iteration is done using our gradient descent rule. Gradient Tape used to track the forward pass and loss calculation. We then get the gradient of the variables and use this to update the variable values using the Adam optimizer.

Evaluation of model

Model 1- (Relu -100 and SoftMax)

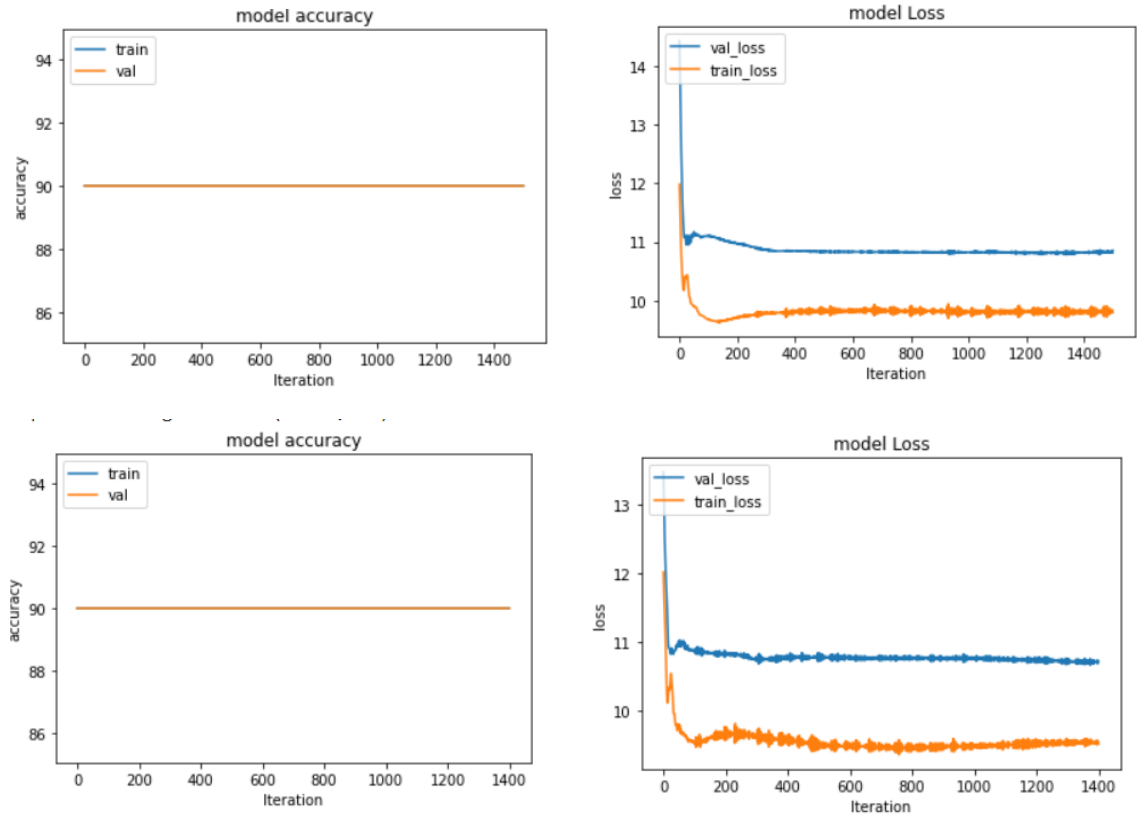
Model accuracy – 90.1%

Model 2-(Relu -300. Relu-100, SoftMax)

Model Accuracy – 90.23%

Since the Loss curve is in L-shape, we can say that it has a high learning rate. High learning rates reduce the loss quickly, so loss gets constant rather than converging properly after few iterations as shown in the blue line.

On the other hand, both model architectures are getting good accuracy (90.1% and 90.2% respectively). Since I have used 1400 Iterations to train the model. if iterations would be increased, it may increase accuracy slightly.



Clearly, we can say that these architectures are sensitive to the number of neurons in layers. Because the loss curve gets sudden constant rather than converging. It means variables of the model might be sensitive towards a few neurons.

Moreover, as we can observe the validation loss curve gets flatten very early while the train loss curve is going down continuously throughout the number of iterations. This evidence shows that both architectures are getting overfitted after approx. 60-70 iterations.

Question_1_2_1

Method - Dropout

```
def dropout(p, h1):
    probthreshold = 1 - p
    u1 = np.random.rand(h1.shape[0], h1.shape[1]) < probthreshold
    h1 *= u1
    h1 = h1 / p
    return tf.convert_to_tensor(h1, dtype=tf.float32)
```

Description – First we generate a 2D array of 1 and 0 using rand function with the same dimensions as H1 (input array for the second layer). Assuming prob Threshold is 0.75 then approx.¼ of the elements of a matrix will be 0. An array is divided by p (prob threshold) for scaling. Now we multiply h1 (input for next layer) with an array of 1 and 0. On multiplying with 0, values would get 0 in the input array. Others will be the same as multiplying with 1. This way, we can drop few neurons from each layer as per requirement.

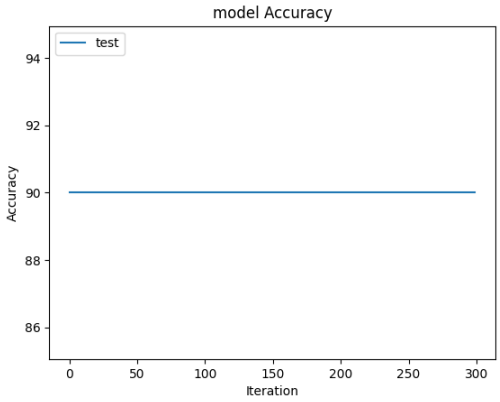
Method – Forward_pass

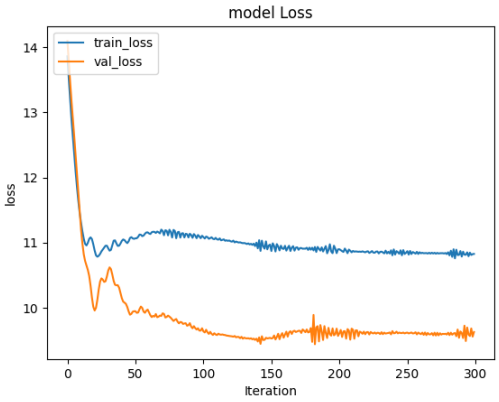
```
def forward_pass(variables, x):
    # forward propagation step
    layer_1_Z = tf.matmul(x, variables["layer_1_weights"]) +
variables["layer_1_bias"]  # layer 1 Z calculation
    layer_1_activation = tf.keras.activations.relu(layer_1_Z)  # Activation
of layer 1

    # dropout few neurons
    dropout(0.25, layer_1_activation.numpy())

    layer_2_Z = tf.matmul(layer_1_activation, variables["layer_2_weights"]) +
variables[
        "layer_2_bias"]  # layer 1 Z calculation
    layer_2_activation = tf.keras.activations.relu(layer_2_Z)

    layer_3_Z = tf.matmul(layer_2_activation, variables["layer_3_weights"]) +
variables[
        "layer_3_bias"]  # layer 2 z calculation
    predictedY = softmax(layer_3_Z)
    return predictedY
```

Description - we used dropout before the second layer, every time the forward pass method will be called, few neurons will be dropped from the layer so that the overfitting problem can be solved.

Evaluation of Dropout Architecture –

Dropout – 30 %



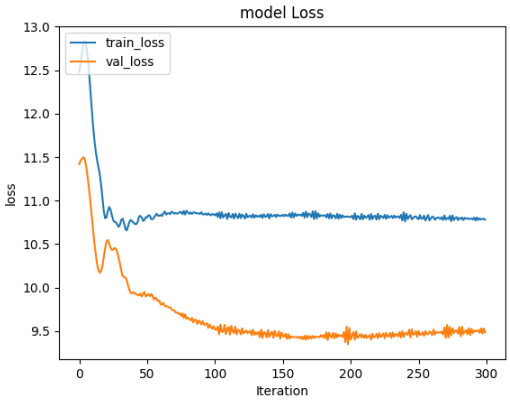


In term of accuracy, we can see that accuracy is approximately the same as the previous model (90.2%). Since layers and iterations are limited in number. Therefore, Accuracy did not change much.

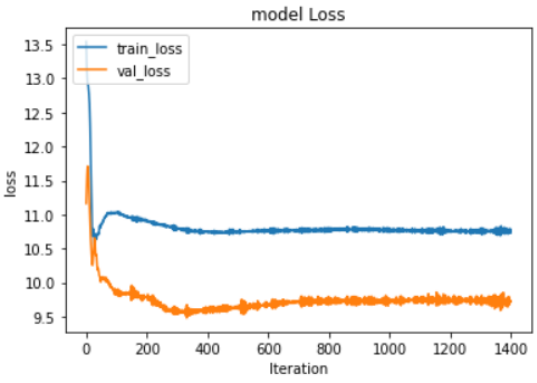
Even though we used dropout in this architecture, we can observe in the above figure that the loss curve for both training data and validation data are following the same pattern. The validation loss curve is falling as same as the training data loss curve. While the network is reasonably small, the minor impact of dropout can be seen on the performance of this model.

Performance of model at different Dropout

Dropout– 45 %



Dropout – 20 %



The above graph is plotted when dropout probability is 45% and 20%. There is not considerable change in the loss. I can say that this architecture is not much sensitive towards dropout probability.

Question_1_3_1

Implementation –

```
def sigmoid(z):
    return tf.math.sigmoid(z)


def forward_pass(variables, x):
    # forward propagation step
    layer_1_Z = tf.matmul(x, variables["layer_1_weights"]) +
variables["layer_1_bias"]  # layer 1 Z calculation
    layer_1_activation = tf.keras.activations.relu(layer_1_Z)  # Activation
of layer 1

    layer_2_Z = tf.matmul(layer_1_activation, variables["layer_2_weights"]) +
variables[
        "layer_2_bias"]  # layer 2 Z calculation
    layer_2_activation = tf.keras.activations.relu(layer_2_Z)

    layer_3_Z = tf.matmul(layer_2_activation, variables["layer_3_weights"]) +
variables[
        "layer_3_bias"]  # layer 3 Z calculation
    layer_3_activation = tf.keras.activations.relu(layer_3_Z)

    layer_4_Z = tf.matmul(layer_3_activation, variables["layer_4_weights"]) +
variables[
        "layer_4_bias"]  # layer 4 Z calculation
    layer_4_activation = tf.keras.activations.relu(layer_4_Z)

    layer_5_Z = tf.matmul(layer_4_activation, variables["layer_5_weights"]) +
variables[
        "layer_5_bias"]  # layer 5 Z calculation
    layer_5_activation = tf.keras.activations.relu(layer_5_Z)

    layer_6_Z = tf.matmul(layer_5_activation, variables["layer_6_weights"]) +
variables[
        "layer_6_bias"]  # layer 6 z calculation
    predictedY = sigmoid(layer_6_Z)
    return predictedY
```

Description – As I earlier implemented, I have used the Relu function in 5 layers and the sigmoid function in the last layer. For multiplying two tensor variables, I have used tf.matmul() function. I have stored weights for the respective layer in the dictionary named “variables” and used them accordingly in each activation layer code. At last predicted values is returned out from the method.

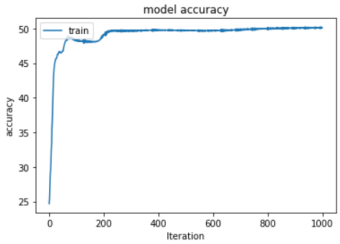
```
def mean_absolute_error(y, predicted_y):
    return tf.reduce_mean(tf.abs(predicted_y - y), axis=-1)
```

Description - This method is used to calculate the difference between predicted image by model and original image. Images are stored in form of an array. We take the difference and rounding off all values. Then we take to mean by each row.

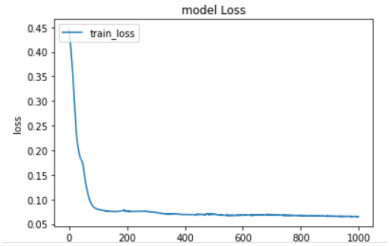
Evaluation of autoencoder –

Autoencoder got trained for 1000 iterations and 50.7% accuracy achieved and loss – 0.07. The model performance for the denoised image prediction is found well-performed according to accuracy and the loss. The predictive capability of all the factors that influenced the denoised image is evaluated to better understand the spatial patterns based on a variable importance analysis. The loss curve gets flatten suddenly it might cause of overfitting of model..

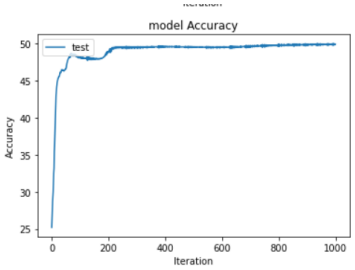
Train Data Accuracy



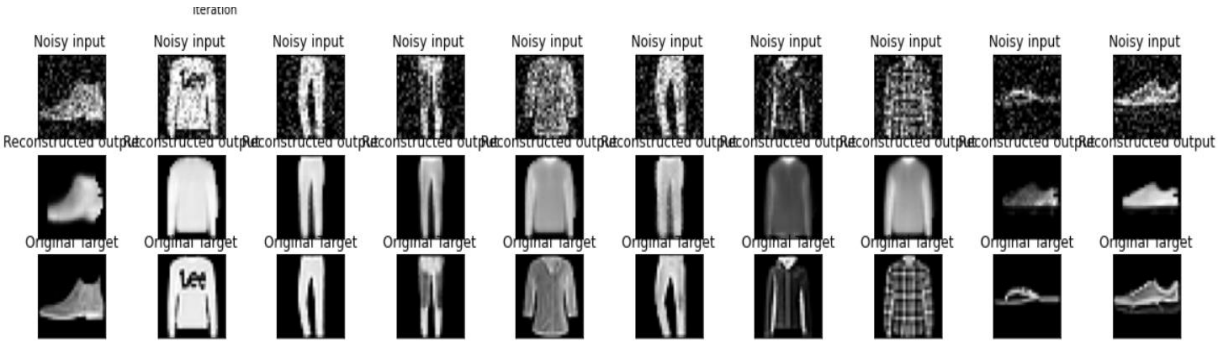
Train Data Loss



Test Data Accuracy



Noisy test input images, Reconstructed output by the model (predicted values) , Excepted output (Original) are displayed respectively below figure.



Autoencoders

Autoencoders is used for taking data from one space to another space from which it can be accurately restored.

Main applications of autoencoders-

- **Dimensionality reduction** – The autoencoders decreased the information and pass in a secret layer whose size is more modest than the input layer. This secret layer where the data has been packed and by extricating this layer from the model. Along these lines, we can conclude that by destroying out the decoder section, it is expected that the element of information is counterfeitly expanded, and its natural measurement is a lot lower. an autoencoder can be utilized for dimensionality decrease with the yield being the secret layer.

- **Image compression** - Even though autoencoders are intended for information pressure yet they are not utilized for this reason in commonsense circumstances. The reasons are:

    - **Lossy compression** - The yield of the autoencoder is not equivalent to the info, it is a nearby however corrupted portrayal. For lossless compression, they are not the best approach
    - **Information explicit**: Autoencoders are simply ready to seriously pack information like what they have been trained on. Since they learn highlights explicit for the given trained data, they are not the same as a standard data compression algo like jpeg. Therefore, we cannot expect an autoencoder prepared on manually written digits to pack scene photographs.

- **Feature extraction** – The encoding portion of Autoencoders assists with learning significant hidden features present in the data, in the process to lessen the reproduction blunder. During encoding, another arrangement of a blend of original features is produced.

- **Image generation** - The thought is that given an input image like an image of a face, the model will create comparative images. The utilization is to produce new characters of liveliness or on the other hand, produce counterfeit human pictures.

- **Sequence to sequence prediction** - The Encoder-Decoder Model that can catch worldly design, for example, LSTMs-based autoencoders, can be utilized to address Machine Translation issues. This can be utilized to foresee the following edge of a video or produce counterfeit recordings.

- **Recommendation system** - Profound Autoencoders can be utilized to comprehend client inclinations to suggest films, books, or different things. Think about the instance of YouTube, the thought is that the information is the bunching of comparative clients dependent on interests

- **Image Denoising** - Autoencoders are truly adept at denoising images. At the point when an image gets undermined or there is a touch of noise in it. We characterize our autoencoder to eliminate (if not all) most of the noise of the image.

The disadvantage of traditional autoencoder –

- **Too lossy** - Autoencoders are lossy, which restricts their utilization in applications when compression corruption influences framework execution hugely. It's impossible to eliminate picture debasement, however, designers can contain misfortune by forcefully pruning the issue space. For this situation, the autoencoder would be more lined up with compacting the data applicable to the issue to be settled.

- **Imperfect decoding** - The decoder process is rarely perfect. In certain conditions, it turns into a business choice to choose how much loss is bearable in the recreated yield. This can be significant in applications like oddity discovery. In these cases, developers need to consistently monitor the performance and update it with new examples

- **Misunderstanding of variables** - The primary limitation of autoencoders is understanding the factors that are pertinent to a model. Building up a decent autoencoder can be a cycle of experimentation, error, and over the long run. Developers can lose the capacity to see which variables are impacting the outcomes.

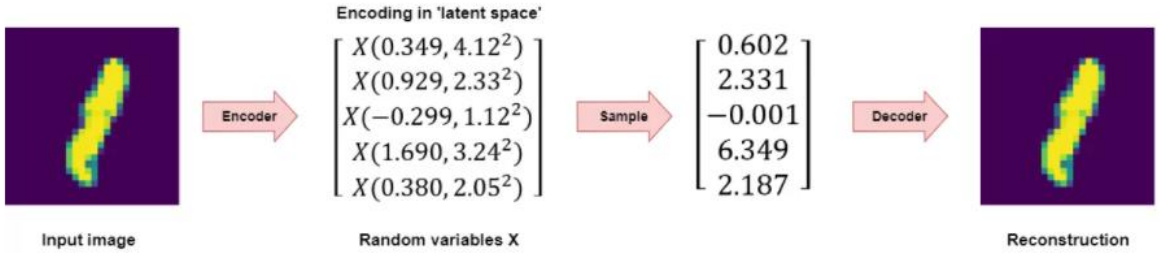
- **Algorithms become too specialized** – Trained autoencoders to learn and duplicate input features is remarkable to the data they are trained on, which produces an explicit algorithm that does not function also for new input. The model can recall the input it was trained on without essentially understanding the reasonable relations between the features

- **Unable in a content generation** – The autoencoder has been trained, we have both an encoder and a decoder yet at the same time no genuine method to create any new substance.

Variational autoencoders - The variational autoencoder is just like an autoencoder that is trained to ensure to keep away from overfitting and guarantee that the latent space has great properties that empower the generative process.

A variational autoencoder consists of an encoder and a decoder and that is trained to limit the reconstructive error between the encoded-decoded data and the initial data. In any case, to present some regularization of the latent space, we continue to a slight change of the encoding-decoding measure i.e. rather than encoding a contribution as a solitary point, we encode it as a distribution over the latent space given their configuration usually Gaussian ones.

The architecture of variational autoencoder-



Source - https://www.machinecurve.com/index.php/2019/12/30/how-to-create-a-variational-autoencoder-with-keras/#recap-what-are-variational-autoencoders

- The data is encoded as a distribution over the latent space
- A point from the latent space is tested from that conveyance
- The examined point is decoded, and the reproduction error can be calculated
- The recreation error is backpropagated through the network

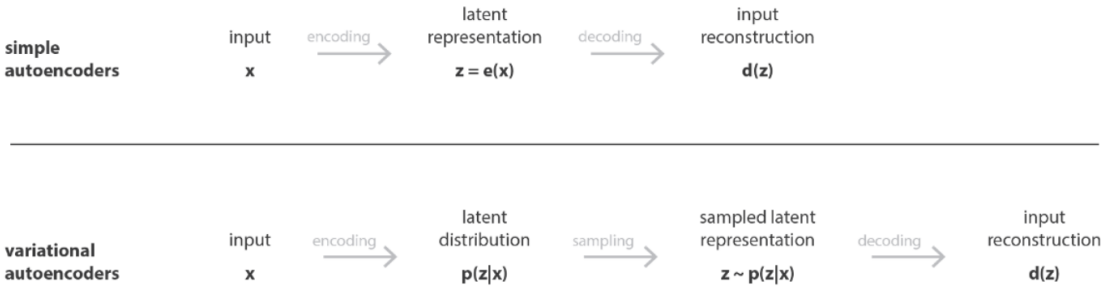
Mathematical details of a variational autoencoder

- Initially, a latent representation z is inspected from the earlier distribution p(z).
- The data x is examined from the restrictive probability distributions p(x|z) that depicts the distributions of the decoded variable given the encoded one.
- The data generation method - encoded data z in the latent space is accepted to follow the earlier distribution p(z) that remind the notable Bayes theorem that makes the connection between the earlier p(z), the probability p(x|z), and the posterior p(z|x)
  Assumption – p(z) – Standard Gaussian distribution
  p(x/z) – Gaussian distribution whose mean defined by a function of variable z.

Differences with traditional autoencoder –

The below two differences permit them to be both continuous and complete, making VAEs possible for the generative process.

- **Encodings are probability distributions** - Basic autoencoders yield one worth for each dimension when planning input data to the latent state. However, a Variational autoencoder yields a Gaussian probability distribution with some mean μ and standard deviation σ for each measurement.



Source - https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73

- **KL divergence and reconstruction error for optimization** - In contrast to traditional autoencoders that limit reconstruction loss only while Variational autoencoders limit a blend of reconstruction error and probability correlation loss called Kullback-Leibler divergence

Question_2

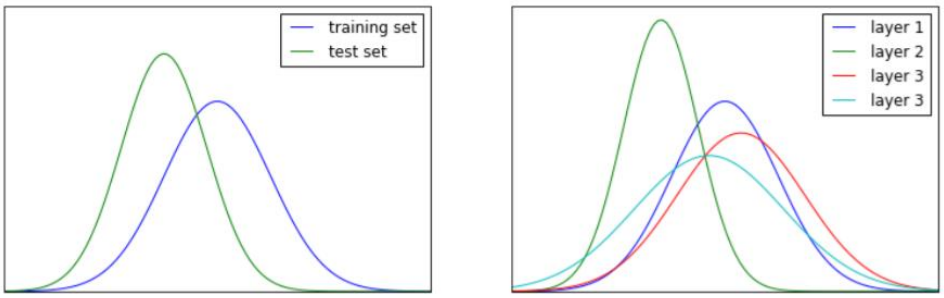
Research – I am selecting Batch normalization as a research topic. A detailed description and the problem that the batch normalization address, how it operates technically, and its advantages are mentioned below.

Batch Normalization –

A deep neural network with many layers is quite sensitive towards random weights and the learning algorithm for that network get trained.

One potential justification of this problem is the appropriation of the distribution to layers somewhere down in the network may change after each mini-batch when the weights are updated. This can make the learning algorithm everlastingly pursue a moving target. This adjustment of the conveyance of contributions to layers in the network is referred to as the specialized name "internal covariate shift."

Batch normalization is a technique for training a deep neural network that normalizes the contributions to a layer for each mini-batch. This balances out the learning interaction and drastically diminishing the number of training epoch needed to train a deep network. we normalize the distribution of each input feature in each layer across each mini-batch to have zero mean and a standard deviation of one as shown in the figure.



(a) Covariate shift (b) Internal covariate shift

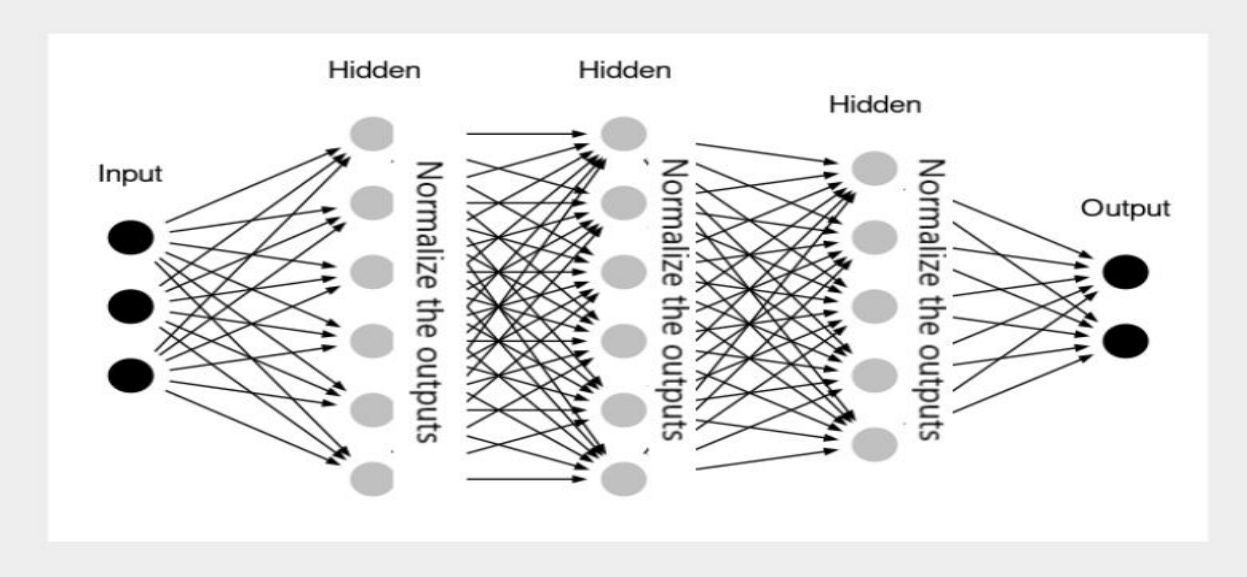
Figure 3.1: Covariate shift vs. internal covariate shift

For example – 1 million examples in 5000 mini-batches with 200 training instances in each mini-batch. This breaking up the training dataset into exclusive partition provides a solution for a serious problem in deep neural network i.e. All 1 million examples push into layers to determine a single update of weights that's why training of model takes a very long time.

How it operates

Dataset breaks into mini-batches and propagates into forward pass. it normalizes the data by subtracting the mean and dividing by the standard deviation. Finally, we scale and shift the data with the learned scale and shift parameters.

Batch normalization is a technique that we add between the layers of the neural network and it ceaselessly takes the yield from the past layer and standardizes it before sending it to the following layer. This makes stable the neural network. Batch normalization is likewise used to keep up the dispersion of the data, to keep up the comparative distribution of the data we use group standardization by normalizing the yields utilizing mean=0, standard dev=1 (μ=0, σ=1). By utilizing this procedure, the model is prepared faster, and it additionally builds the accuracy of the model contrasted with a model that does not utilize the batch normalization



The problems that batch normalization address from a deep neural network –

- Batch normalization is used to normalize the input layer. It can be used with most network types like CNN, RNN etc.
- Its better usage is before the activation function that may output non-gaussian distribution for network type.
- Batch normalization makes the network more stable during training.
- The stability to training brought by batch normalization can make training deep networks less sensitive to the choice of weight initialization method.
- Since mean and standard deviations calculated for each input feature are calculated over the mini-batch, it can use to standardize raw input variable
- Batch normalization does regularization, reducing error itself. Therefore, no longer requiring the use of dropout for regularization

Moreover, normalization between the layers of the network also improves the precision of the model and avoids overfitting too. Because of all these factors, we are noticing consistent use of batch normalization while working with deep learning models.

References –

- https://iq.opengenus.org/applications-of-autoencoders/
- https://www.mygreatlearning.com/blog/autoencoder/
- https://searchenterpriseai.techtarget.com/feature/How-to-troubleshoot-8-common-autoencoder-limitations
- https://www.machinecurve.com/index.php/2019/12/30/how-to-create-a-variational-autoencoder-with-keras/#recap-what-are-variational-autoencoders
- https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73
- https://www.machinecurve.com/index.php/2019/12/24/what-is-a-variational-autoencoder-vae/#how-are-vaes-different-from-traditional-autoencoders
- https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/#:~:text=Batch%20normalization%20is%20a%20technique,required%20to%20train%20deep%20networks.
- https://kth.diva-portal.org/smash/get/diva2:955562/FULLTEXT01.pdf
- https://analyticsindiamag.com/hands-on-guide-to-implement-batch-normalization-in-deep-learning-models/