Endsem Assignment

Sabyasachi Choudhury, 250938

January 1, 2026

1 Q1 - SVD compression

In SVD, a matrix transform A is factorized into three consecutive matrix transforms: $U\Sigma V^T$, where columns of U are the eigenvectors of AA^T and columns of V are the eigenvectors of A^TA, both sorted in decreasing order of their corresponding eigenvalues. The diagonal elements of Σ are the singular values of A, so the square roots of the eigenvalues of AA^T, again in descending order.

The significance of the descending order in this case is that it encodes the contribution of each eigenvector used in the final matrix transform A. Hence, one way in which we can use SVD is by simply truncating U, V and Σ to the highest few columns and values.

So, since SVD can be viewed as a sum of progressively smaller rank 1 matrices of the form $U_i\Sigma_i V_i^T$, where A_i represents the $i'th$ column of a matrix A. To compress it, we can only consider the top few such rank 1 matrices.

1.1 Example

Consider a matrix A,

$$\begin{bmatrix} 3 & 2 \\ 2 & 3 \\ 2 & -2 \end{bmatrix}$$

. I'll skip the tedious computation here, but the decomposition of this matrix is as follows:

$$\begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{6} & -\frac{2}{3} \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{6} & \frac{2}{3} \\ 0 & \frac{2\sqrt{2}}{3} & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 5 & 0 \\ 0 & 3 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{bmatrix}$$

Here, our SVD consists of two rank 1 matrices, if we only include the one with the largest singular value (5), we get the following matrix

$$\begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \\ 0 \end{bmatrix} 5 \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$$

Which equals the first order approximation

$$\begin{bmatrix} \frac{5}{2} & \frac{5}{2} \\ \frac{5}{2} & \frac{5}{2} \\ 0 & 0 \end{bmatrix}$$

We can get a quantitative measure of how good of an approximation this is by using the frobenius energy - $\sqrt{\sum A_{ij}^2}$. The computation here comes out that the frobenius energy of the original matrix is $\sqrt{34}$ and of the approximation is 5, hence the approximation contains around 86% of the original energy of the matrix.

2 Dimensionality Reduction

First center the matrix (subtract mean of each feature) and transpose it to give a 4×2 matrix X which gives

$$\begin{bmatrix} -4 & 2.5 \\ 0 & -4.5 \\ 5 & -3.5 \\ -1 & 5.5 \end{bmatrix}$$

For the given dataset, the covariance matrix

$$Q = X^T X/(n-1)$$

is hence:

$$\begin{bmatrix} 14 & -11 \\ -11 & 23 \end{bmatrix}$$

Now, compute the eigenvectors and sort them in order of their corresponding eigenvalues to for the covariance matrix. The two computed eigenvalues come out to be approximately 30.39 and 6.61, hence we only consider the larger one for our dimension reduction.

The corresponding normal eigenvector V is computed to be

$$\begin{bmatrix} 0.55 \\ 0.82 \end{bmatrix}$$

Hence, the corresponding principal component for the dataset is XV,

$$\begin{bmatrix} -0.15 \\ -3.69 \\ -0.12 \\ 3.96 \end{bmatrix}$$

3 Q3 - Stochastic vs batch gradient descent

SGD - for each update step only calculates the gradients and outputs using one randomly chosen sample from the dataset (hence the stochastic).

Batch GD - Each update is calculated by iterating over the entire dataset and taking the average of calculated gradients.

Minibatch GD - each update is calculated by using small minibatches of say, 32 or 64 samples to calculate the mean gradient.

For more complex datasets generally minibatch GD is used, or we start off with SGD before using batch gd or minibatch with very large batch sizes for fine tuning. This is primarily due to performance considerations, since its often not possible to load the entire dataset into ram for computation.

Batch GD is also very time consuming, and at early stages of training it is redundant to calculate the gradient using the entire dataset, since the gradient calculated using most of the samples will point in approximately similar directions which minimizes the loss.

BGD only becomes more accurate than SGD when near the minima, since here SGD tends to bounce around the minima due to the noise when we take the gradient of only one sample. BGD converges faster here.

SGD is also better for the early stages before fine tuning since it allows the model to escape bad local minima due to its random nature. Mathematically, the gradients computed using SGD have a high variance which, while allowing for better exploration of the loss space, also means it struggles to pinpoint minima.

4 Q4 - training loop

```
#x_train, y_train are train samples and corresponding labels
for epoch in range(len(epochs)):
    indices = permutation(0, len(x_train)-1) #permutation of indices of x_train
```

```
x_train = x_train[indices]
y_train = y_train[indices]
# Shuffled x_train and y_train together
for i, sample in enumerate(x_train):
    pred = model.predict(x_train[i])
    loss = loss_function(pred, y_train[i])
    grad = calculate_gradient_for_each_layer(loss, model, x_train[i], y_train[i])
    model.weights = model.weights - grad*lr
```

The gradients for each layer are calculated using backpropagation. For example, for a model that can be represented as the following operations on an input X:

$$Y_(pred) = f_2(W_2 f_1(W_1 X + B_1) + B_2)$$

$$L = loss(Y_{pred}, Y_{true})$$

where f_i are the activations, W_i are the weight matrices of each layer, and B_i are the bias matrices of each layer.

Hence, define

$$\begin{aligned} Z_1 &= W_1 X + B_1 \\ A_1 &= f_1(Z_1) \\ Z_2 &= W_2 A_1 + B_2 \\ A_2 &= f_2(Z_2) = Y_{pred} \end{aligned}$$

Thus, the following are the gradients for all weights and biases

$$\begin{aligned} \delta_2 &= \frac{\partial L}{\partial Y_{pred}} \odot f_2'(Z_2) \\ \frac{\partial L}{\partial W_2} &= \delta_2 A_1^T \\ \frac{\partial L}{\partial B_2} &= \delta_2 \delta_1 \qquad\qquad = (W_2^T \delta_2) \odot f_1'(Z_1) \\ \frac{\partial L}{\partial W_1} &= \delta_1 X^T \\ \frac{\partial L}{\partial B_1} &= \delta_1 \end{aligned}$$

4.1 Q5 - Adam Optimizer

Based on most current material, is one of the most popular optimizers used in neural networks today. Combines the best properties of previous optimizers like adagrad (per parameter update rule, leveraging sum of squares of all past gradients) and RMSProp (which uses the EMA of past gradients for each parameter), while losing the disadvantages, like the vanishing gradient problem with adagrad.

The eqns used in adam are as follows

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1)\, g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2)\, g_t^2 \end{aligned}$$

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \end{aligned}$$

$$\theta = \theta - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}$$

As visible, the numerator carries on momentum from past gradients. If several update steps have been in the same direction, they boost the next one (kinda like inertia in physics). The denominator causes areas with steep gradients to have a lower learning rate, so they don't go too fast and overshoot. Think of them as applying the brakes while going down a hill on a bike.

4.2 Q6 - Cross Entropy Loss

For classification, cross entropy loss $-\sum t_i * \log(p_k)$ is usually preferred over simple classification accuracy (percentage of predictions which are correct) or MSE for the following reasons.

Easy to compute gradient, unlike classification accuracy.

Puts emphasis on not just having the biggest logit point to the correct class, but also the confidence of the model.

For example, for a target [0, 0, 1], let our two models produce logits [0.3, 0.3, 0.4] and [0.1, 0.1, 0.7]. While both provide the correct answer, the latter is much more confident and hence a better model. CE Loss accounts for this.

While calculating the gradient, MSE has a term of form

```
output*(1-output)
```

This slows down the model from reaching higher output closeness to the ideal values - 1 or 0. CE Loss doesn't have such a term in its calculated gradient.

However, CE loss has disadvantages, such as its sensitivity to outliers. It heavily penalizes confident incorrect predictions, which can lead it to forcefully try and overfit to outliers which cause a massive increase in error.

4.3 Q7 - Activation Layers

Activation layers apply a non-linear function to the output of a neuron (or a whole layer) in a neural network. Without activation functions, a neural network would just be a stack of linear transformations and could only model linear relationships. Activation layers enable networks to learn complex, non-linear patterns. Almost any function can be an activation function, provided it suffices some criteria,

- Function must be non linear.
- Computing the function (and its gradient) should preferably be cheap, otherwise it's better to just add more layers and use a cheaper but simpler activation like ReLU.
- Should be differentiable, or non differentiable at a small, finite number of points, like ReLU - non differentiable only at 0.

4.4 Q8 - Sigmoid Case Study

in the expression

$$\frac{\partial L}{\partial w_i} = \delta \cdot x_i$$

δ represents the error from deeper layers, and x_i is input from previous layer. However, since this input has passed through the sigmoid function, it is strictly positive - thus, updates to all the weights always happen in the same direction - this makes training slow, as different weights in the same layer might actually need to move in different directions.

The above figure explains the observed zigzagging descent when using sigmoid.

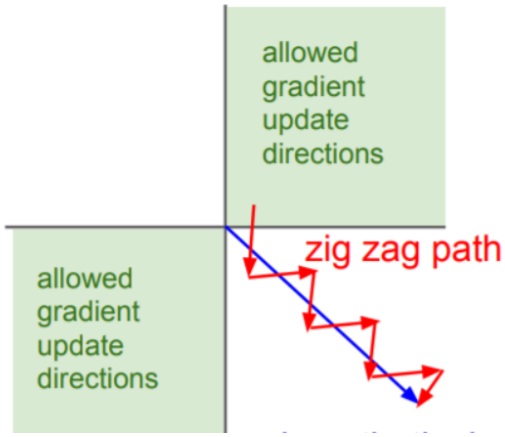

Figure 1: zig zagging in sigmoid activations

4.5 Q9 - Swish

$$f(x) = x\sigma(x)$$

$$f'(x) = \sigma(x) + x\sigma(x)(1 - \sigma(x)) = \sigma(x) + f(x)(1 - \sigma(x))$$

Compare this to the derivative of the sigmoid function

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Swish requires computing the sigmoid, and then its product with x. It also involved two multiplications and then an addition. Sigmoid requires only a multiplication, and no double computation as we simply reuse the value of $\sigma(x)$ calculated earlier

4.6 Q10 - CNN From Scratch

Link to Colab Repo

Implemented CNN in torch using torch.functional.fold and unfold. An attempt has been made to use as much vectorization as possible.

4.7 Q11 - Custom Activation Function

One of the most interesting and mathematically sound activation functions in recent times is the GELU function (Gaussian Error Linear Unit). It's formulated as follows -

$$GELU(x) = \frac{x}{2}(1 + erf(\frac{x}{\sqrt{2}}))$$

Where erf is the gaussian error function, the probability that a random variable chosen from a normal distribution with mean 0 and standard deviation $1/\sqrt{2}$ is in the range $[-x, x]$

Qualitatively, this simulates a normalized dropout function, assuming that the inputs to a layer are approximately normally distributed. We then allow more activation for the values which are rarer.

However, this activation only works for the positive values, and not the negatives. Hence, I propose two new activation functions - a symmetric GELU (SGELU), and an adaptive SGELU (AdaSGELU)

$$SGELU(x) = xerf(x)$$

$$AdaSGELU(x) = \alpha x \times erf(x)$$

where α is a trainable parameter.

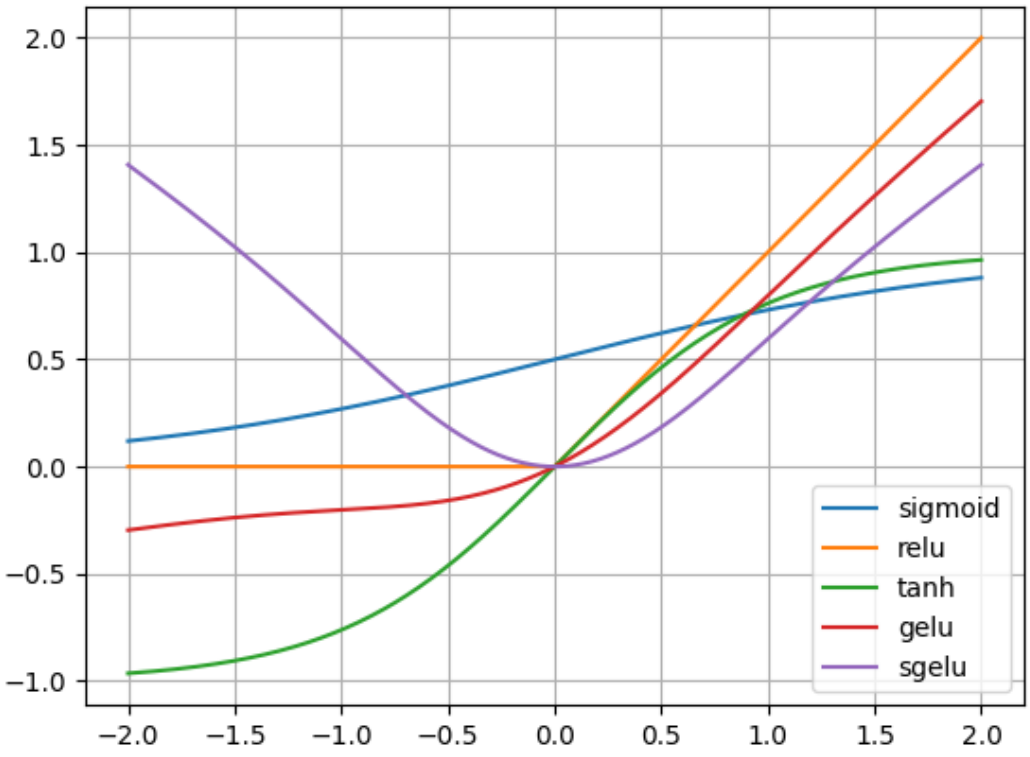

Figure 2: Comparing different activations

4.7.1 Experimentation

We tested this activation function on the mnist dataset, on a fully connected model with shape 784 - 128 - 10. There's a batchnorm layer between the input and hidden layer, as well as a dropout layer with frequency 0.5. Repeated experiments on this model show that our SGELU and AdaSGELU perform on par, if not better than RELU and GELU.

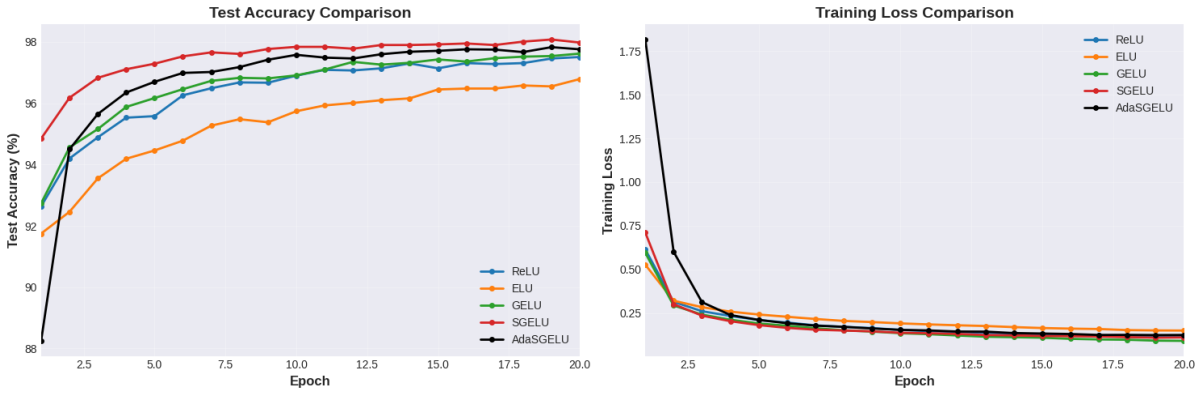

Figure 3: Comparing different activations loss and test accuracy on mnist

We also tried it on two autoencoders - a FC autoencoder with architecture 784-256-128-64, with activation-batchnorm between each pair of dense layers, and a convolutional autoencoder with architecture as follows

```
class MnistAutoencoder(nn.Module):
    def __init__(self, activation):
        super().__init__()
        # Encoder
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, padding=1),
            nn.BatchNorm2d(16),
            activation(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(16, 32, 3, padding=1),
            nn.BatchNorm2d(32),
            activation(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(32, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            activation(),
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.BatchNorm2d(64),
            activation(),
            nn.ConvTranspose2d(64, 32, 3, padding=1),
            nn.Upsample(scale_factor=2, mode='nearest'),
            nn.BatchNorm2d(32),
            activation(),
            nn.ConvTranspose2d(32, 16, 3, padding=1),
            nn.Upsample(scale_factor=2, mode='nearest'),
            nn.BatchNorm2d(16),
            activation(),
            nn.ConvTranspose2d(16, 1, 3, padding=1),
```

```
        nn.Sigmoid()
    )
```

The results on both these are shown below

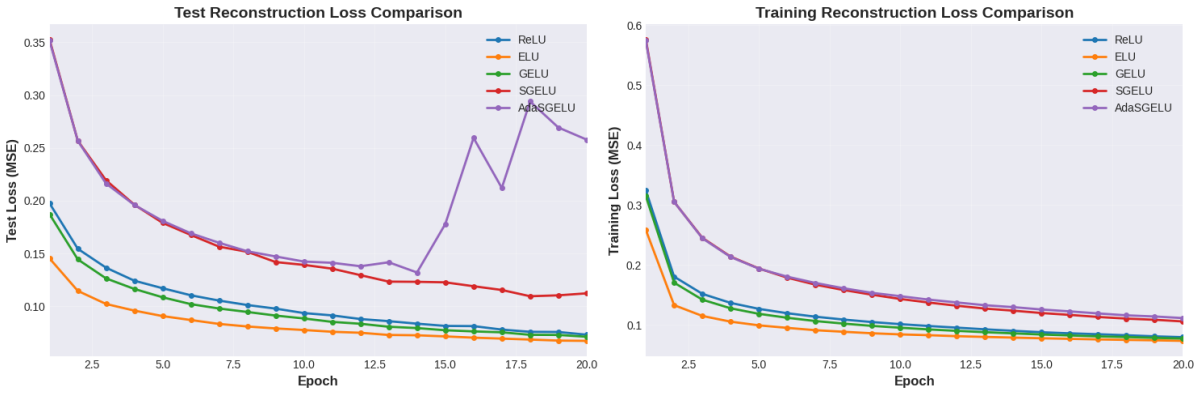

Figure 4: FC Autoencoder

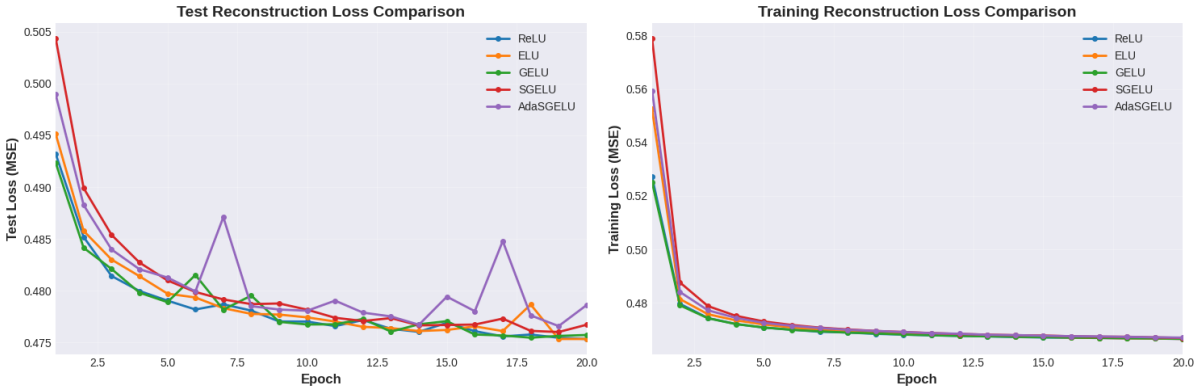

Figure 5: Convolutional Autoencoder

4.7.2 Conclusion

Theoretically, I expected that SGELU and AdaSGELU would be better than GELU due to the equal weightage it provides to both negative and positive inputs, however from experiments It seems that while SGELU and AdaSGELU do better on classification tasks, it struggles with Autoencoding tasks.

Also interesting to note was that without the Batchnorm layer, SGELU struggles even with classification tasks.