CS5787: Exercises 1

https://drive.google.com/drive/folders/1bsQLTDolhzahDo02rZpo3JAspEHwgho4?usp=drive_link

Full Name	**Dahwi Kim**	**Yoo Choi**
Net ID	dk865	yc2822

1 Theory: Question 1 [12.5 pts]

(a) Given that we have m batches of inputs of size 10, we want to simultaneously process all m input samples. Therefore, our input X has the shape (m x 10). It has m rows of inputs with size 10.

(b) Our hidden layer has size 50. Therefore, our hidden layer H will have a shape of (m x 50), having m rows of 50.

To get this hidden layer, we can deduce that the hidden layer's weight vector W_h has the shape (10 x 50), and the bias vector b_h has the shape (1 x 50). To check, we see that (m x 10) * (10 x 50) does yield a vector of (m x 50). For each of the 50 columns, we also add the corresponding bias value specified in the bias vector with (1 x 50).

(c) Likewise, the output weight vector W_o has the shape (50 x 3), and the bias vector b_o has the shape (1 x 3). To check, we see that (m x 50) * (50 x 3) does yield a vector of (m x 3). For each of the 3 columns, we also add the corresponding bias value specified in the bias vector with (1 x 3).

(d) The shape of our output matrix Y is (m x 3). This comes from our batch size m, so we want m rows of outputs of size 3.

(e) We first define the hidden layer:

$$H = Relu(XW_h + b_h)$$

Then, our output matrix Y:

$$Y = Relu(HW_o + b_o)$$

Finally, we can define Y as:

$$Y = Relu(Relu(XW_h + b_h)W_o + b_o)$$

2 Theory: Question 2 [12.5 pts]

First, we want to visualize the structure. Our input is given by (200 x 300) with RGB channels. This is (200 x 300 x 3).

The first layer has 100 feature maps, so we will need 100 kernels to go from input layer to layer 1. Likewise, we need 200 kernels to go from layer 1 to layer 2, and we need 400 kernels to go from layer 2 to layer 3.

Each kernel is (3 x 3). The depth of each kernel depends on the number of feature maps in the previous layer. Given the information above, we can conclude that to go from input layer to lowest layer, we have 100 kernels of (3 x 3) with depth of 3. Then to the middle layer, we have 200 kernels of (3 x3) with depth of 100. Then, we have 400 kernels of (3 x 3) with depth of 200.

The number of parameters can be expressed as,

$((h_{kernel} \cdot w_{kernel} \cdot (num\ of\ input\ channels) + 1)) \cdot (num\ of\ output\ channels)$, 1 being bias term.
Layer1: (3x3x3+1)x100 = 2800
Layer2: (3x3x100+1)x200 = 180200
Layer3: (3x3x200+1)x400 = 720400

Total number of parameters = 903400

3 Theory: Question 3 [25 pts]

a.

$$\frac{\partial f}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial f}{\partial y_i} \cdot \frac{\partial y_i}{\partial \gamma}$$

Since:

$$y_i = \gamma \hat{x}_i + \beta$$
$$\frac{\partial y_i}{\partial \gamma} = \hat{x}_i$$

Thus:

$$\frac{\partial f}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial f}{\partial y_i} \cdot \hat{x}_i$$

b.

$$\frac{\partial f}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial f}{\partial y_i} \cdot \frac{\partial y_i}{\partial \beta}$$

Since:

$$\frac{\partial y_i}{\partial \beta} = 1$$

Thus:

$$\frac{\partial f}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial f}{\partial y_i}$$

c.

$$\frac{\partial f}{\partial \hat{x}_i} = \sum_{j=1}^{m} \frac{\partial f}{\partial y_j} \cdot \frac{\partial y_j}{\partial \hat{x}_i} = \frac{\partial f}{\partial y_i} \cdot \frac{\partial y_i}{\partial \hat{x}_i} \quad \text{since} \quad \frac{\partial y_j}{\partial \hat{x}_i} = 0 \quad \text{for } i \neq j$$

Since:

$$\frac{\partial y_i}{\partial \hat{x}_i} = \gamma$$

Thus:

$$\frac{\partial f}{\partial \hat{x}_i} = \frac{\partial f}{\partial y_i} \cdot \gamma$$

d.

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

Taking the derivative with respect to σ^2:

$$\frac{\partial \hat{x}_i}{\partial \sigma^2} = -\frac{1}{2}(x_i - \mu)(\sigma^2 + \epsilon)^{-\frac{3}{2}}$$

Using the chain rule:

$$\frac{\partial f}{\partial \sigma^2} = \sum_{i=1}^{m} \frac{\partial f}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial \sigma^2}$$

Substituting $\frac{\partial \hat{x}_i}{\partial \sigma^2}$:

$$\frac{\partial f}{\partial \sigma^2} = \sum_{i=1}^{m} \frac{\partial f}{\partial \hat{x}_i} \cdot \left(-\frac{1}{2}(x_i - \mu)(\sigma^2 + \epsilon)^{-\frac{3}{2}} \right) = \sum_{i=1}^{m} \frac{\partial f}{\partial \hat{x}_i} \cdot \frac{-(x_i - \mu)}{2(\sigma^2 + \epsilon)^{3/2}}$$

e.

$$\frac{\partial \hat{x}_i}{\partial \mu} = -\frac{1}{\sqrt{\sigma^2 + \epsilon}}$$

Applying the chain rule:

$$\frac{\partial f}{\partial \mu} = \sum_{i=1}^{m} \frac{\partial f}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial \mu}$$

$$\frac{\partial f}{\partial \mu} = \sum_{i=1}^{m} \frac{\partial f}{\partial \hat{x}_i} \cdot \left(-\frac{1}{\sqrt{\sigma^2 + \epsilon}} \right)$$

f.

$$\frac{\partial f}{\partial x_i} = \frac{\partial f}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial x_i} + \frac{\partial f}{\partial \sigma^2} \cdot \frac{\partial \sigma^2}{\partial x_i} + \frac{\partial f}{\partial \mu} \cdot \frac{\partial \mu}{\partial x_i}$$

We have:

$$\frac{\partial \hat{x}_i}{\partial x_i} = \frac{1}{\sqrt{\sigma^2 + \epsilon}}$$
$$\frac{\partial \sigma^2}{\partial x_i} = \frac{2(x_i - \mu)}{m}$$
$$\frac{\partial \mu}{\partial x_i} = \frac{1}{m}$$

Then we get:

$$\frac{\partial f}{\partial x_i} = \frac{\partial f}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma^2 + \epsilon}} + \frac{\partial f}{\partial \sigma^2} \cdot \frac{2(x_i - \mu)}{m} + \frac{\partial f}{\partial \mu} \cdot \frac{1}{m}$$

Substituting $\frac{\partial f}{\partial \sigma^2}$ and $\frac{\partial f}{\partial \mu}$, we finally get:

$$\frac{\partial f}{\partial x_i} = \frac{\partial f}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma^2 + \epsilon}} - \frac{(x_i - \mu)}{m(\sigma^2 + \epsilon)^{\frac{3}{2}}} \sum_{j=1}^{m} \frac{\partial f}{\partial \hat{x}_j} \cdot (x_j - \mu) - \frac{1}{m\sqrt{\sigma^2 + \epsilon}} \sum_{j=1}^{m} \frac{\partial f}{\partial \hat{x}_j}$$

4 Practical [50 pts]

4.1 Architecture Description

The architecture is based on the original LeNet-5 , adapted for the FashionMNIST dataset, which consists of 28x28 grayscale images. The key components of the implemented models are as follows:

- **Convolutional Layers:**
  - **Conv1:** A 5x5 convolutional layer with 6 output channels, followed by Batch Normalization (if enabled) and ReLU activation. A 2x2 max-pooling layer is applied after activation.
  - **Conv2:** A 5x5 convolutional layer with 16 output channels, followed by Batch Normalization (if enabled) and ReLU activation. A 2x2 max-pooling layer is applied after activation.
- **Fully Connected Layers:**
  - **FC1:** A fully connected layer with 120 output units and ReLU activation.
  - **FC2:** A fully connected layer with 84 output units and ReLU activation.
  - **FC3:** A fully connected layer with 10 output units, corresponding to the 10 classes of FashionMNIST.
- **Dropout:** Dropout is applied after the second fully connected layer (FC2) in the base architecture. In other variations, dropout is applied after both FC1 and FC2, with dropout probabilities of 0.3, 0.5, and 0.7.
- **Batch Normalization:** Two configurations were tested for batch normalization: applying it before the ReLU activation and after ReLU activation in the convolutional layers.
- **Weight Decay:** Different levels of weight decay (L2 regularization) were applied to regularize the model, with values of $1e^{-2}$, $1e^{-3}$, and $1e^{-4}$.

This architecture maintains the core structure of the original LeNet-5 while incorporating modern techniques such as batch normalization and dropout to improve generalization and stability during training.

In contrast with the handwritten-number dataset (32x32) on which Lenet5 was pre-trained, we were given 28x28 images in the Fashion MNIST dataset. We made certain decisions on our architecture: (1) we chose not to add padding to our input images (same with Lenet5) (2) we used the Relu activation function (Lenet used sigmoid) (3) we used max-pooling (Lenet used average-pooling). Using relu is a common practice for modern neural networks for its simplicity and absence of vanishing gradients. We used max-pooling for the sake of feature preservation and non-linearity, as these would be better for the given task than using average-pooling for the sake of smoothing, which is not as important.

We used the standard 80/20 split to divide the given training set into a training set and a validation set. Given that 60,000 images for our training set is considered a medium-size dataset (not >100,000), we wanted to have enough data in the validation set to validate our training, choosing 80/20 over 90/10.

4.2 Training Hyperparameters & Experiments

- Testing different hyperparameters: All the experiments listed below were tested against each other over three different learning rates. To choose the best model over each epoch, we saved the model that yielded the highest validation accuracy. The model with the highest validation accuracy was used at the end to determine the test accuracy. The final result for each regularization technique is saved with the model parameters, learning rate, and other metadata. It is printed at the end of our training loop. The best model for each regularization technique is then used to plot the convergence graphs.
- Batch-size: We chose a batch size of 64 over 32 or 128 as a good balance between stability and training efficiency. Using the T4 GPUs on Colab, we ran all three batch sizes with our models. We found that the batch size of 64 was running fast enough and with reasonable memory usage, so we did not need to downsize to 32. The GPUs on Colab are more than capable of handling larger batch sizes - but with a batch size of 128, we run the risk of faster convergence and more overfitting, especially on small to medium-size datasets.
- Batch normalization: We ran experiments testing whether batch normalization is best used before or after the ReLU activation function. From literature, batch normalization applied **before** the ReLU, directly after the convolutional layer, should yield the highest accuracy. This makes sense because ReLU maps all negative values to 0. As a result, applying batch normalization **after** ReLU might yield a mean value close to 0, with the remaining non-zero values skewed to the right. This reduces the effectiveness of normalization. By applying batch normalization before ReLU, the entire input distribution can be normalized, leading to better overall training performance. However, our experimental results showed that the test accuracies for both variations were quite similar. In the results below, it shows that batch-norm applied after ReLU yielded higher test accuracy.
- Learning rate: It is common practice to use the default learning rate of 0.001 for the Adam optimizer in the beginning. We tested out three learning rates (0.01, 0.001, 0.0001) against all of our models. We found that using a learning rate of 0.001 yielded the highest test accuracies for all four models.
- Optimizer: We used the Adam optimizer as per common practice in modern machine learning due to its faster convergence speed. We tried using stochastic gradient descent (SGD) and Adagrad, but using the Adam optimizer most often produced highest test accuracies.
- Dropout rate: We tested two areas with dropout. First, we experimented with having two dropouts (once after fc1, and once after fc2) against having only one dropout. We found dropping out only once yielded the best results. We implemented dropout in the layer with 84 values, right before we move to the final 10 classes. The other area was the dropout probability. After testing three probabilities: 0.3, 0.5, and 0.7, we found that p=0.3 had the highest test accuracy. This means that at the dropout stage, we randomly keep 70% of our parameters and set 30% of them to 0.
- Weight-decay: We tested three weight decay λ values: 0.01, 0.001, and 0.0001. We found that 0.0001 most often yielded the highest accuracies.
- Epochs: We tried out training our models with 10, 15, and 20 epochs. With 10 epochs, we observed less accuracy(right around 88%) on test dataset, and with 20 epochs, there was clear overfitting with our training set. 15 epochs performed much better than running 10 epochs, and also showed less overfitting than running 20 epochs.

4.3 Model Evaluation

We were able to achieve >88% test accuracy for all four models (no regularization, dropout, weight decay, and batch normalization) within 15 epochs. The four graphs are shown below, plotting train and test accuracies for each epoch.

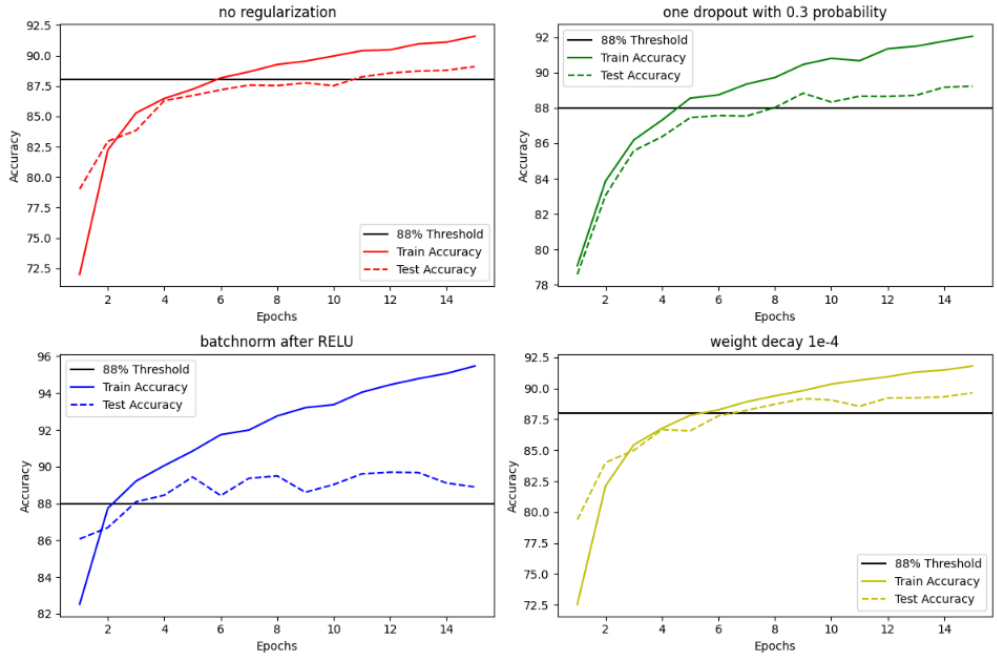

Figure 1: Train and test accuracies against epoch

Final Accuracies for Various Techniques

Model/Technique	Final Train Accuracy (%)	Final Test Accuracy (%)
Baseline (No Regularization)	91.59	89.09
Dropout	92.05	89.23
Batch Normalization	95.47	89.61
Weight Decay (L2)	91.81	89.64

Figure 2: Train and test accuracies table

4.3.1 Overfitting Across Models

All models exhibit a degree of overfitting, as evidenced by the divergence between training and test accuracies after several epochs. However, batch normalization displayed the largest overfitting, suggesting that while it helps the model converge faster, it may cause over-optimization on the training set at the expense of test performance. Weight decay, on the other hand, showed a relatively smaller gap between training and test accuracy, making it the most balanced regularization technique for your model in terms of reducing overfitting.

4.3.2 Impact of Learning Rate

As stated above, a learning rate of 0.001 produced the best results across all models. Higher learning rates (0.01) might have caused the model to miss optimal minima, while lower learning rates (0.0001) may have slowed down the learning process without significant benefits.

4.4 Conclusion

Batch Normalization (BN): Achieved the second highest final test accuracy but showed a significant gap between training and test accuracy. This indicates that while BN helps in stabilizing training, it may also cause overfitting when applied. One reason for this could be that BN adjusts the layer outputs to have unit variance, but in doing so, it might also cause the model to rely too heavily on the learned parameters, reducing the generalization ability on unseen data.

Weight Decay: Achieved the highest performance in terms of final test accuracy. Weight decay helps reduce overfitting by penalizing large weights, thus controlling model complexity. Its consistent performance demonstrates that it can act as a robust regularizer without the risk of inducing too much overfitting as observed in BN.

Dropout: Though dropout achieved decent test accuracy, it generally lagged behind batch normalization and weight decay. This may suggest that the architecture's fully connected layers were not complex enough to benefit fully from dropout. Moreover, higher dropout rates (p=0.5 or p=0.7) could potentially lead to excessive information loss during training, causing poorer performance.

No Regularization: The model without regularization achieved the lowest test accuracy, confirming the importance of regularization techniques in improving generalization, especially on medium-sized datasets like FashionMNIST.

A potential experiment for further optimization could involve combining batch normalization with weight decay. This combination might mitigate some of the overfitting issues while maintaining the training speedup advantages of batch normalization.