CS5787: Exercises 1

`https://github.com/shl225/CS5787_HW1`

Sean Hardesty Lewis
shl225

1 Theory: Question 1 [12.5 pts]

Suppose you have an MLP composed of an input of size 10, followed by one hidden layer with an output of size 50, and finally one output layer with 3 output neurons. All artificial neurons use the ReLU activation function. The batch size used is m.

a. **What is the shape of the input X?**
The shape of the input X is $(m, 10)$ since m is the batch size and each sample in the batch contains 10 features. X is shaped this way since each row corresponds to one sample and each column is one feature of the sample.

b. **What about the shape of the hidden layer's weight vector W_h, and the shape of its bias vector b_h?**
The number of rows in the weight matrix corresponds to the number of neurons in the previous layer as each neuron in the previous layer connects to every neuron in the current layer. The number of columns in the weight matrix is the number of neurons in the current layer. Since the input layer has 10 neurons and the hidden layer has 50 neurons, the shape of the weight matrix W_h connecting the input layer to the hidden layer is $(10, 50)$. Each of the 10 input neurons connects to each of the 50 hidden neurons through a unique weight.
The bias vector has one bias per neuron in the current layer. Since the hidden layer has 50 neurons, the bias vector b_h will have a shape of $(50,)$, showcasing a single bias term for each of the 50 neurons in the hidden layer.

c. **What is the shape of the output layer's weight vector W_o, and its bias vector b_o?**
Since the hidden layer has 50 neurons and the output layer has 3 neurons, the shape of the weight matrix W_o connecting the hidden layer to the output layer is $(50, 3)$. Each of the 50 hidden neurons connects to each of the 3 output neurons through a unique weight.
Since the output layer has 3 neurons, the bias vector b_o will have a shape of $(3,)$, showcasing a single bias term for each of the 3 neurons in the output layer.

d. **What is the shape of the network's output matrix Y?**
The shape of the output Y is $(m, 3)$ since m is the batch size and each sample in the batch contains 3 features. Y is shaped this way because there are only 3 outputs per sample in the output layer.

e. **Write the equation that computes the network's output matrix Y as a function of X, W_h, b_h, W_o, and b_o:**

$$Y = \text{ReLU}\left(\text{ReLU}\left(XW_h + b_h\right) W_o + b_o\right)$$

$XW_h + b_h$ represents the linear combination of inputs X and the weights of the hidden layer W_h, adding the bias b_h.
We apply the ReLU function element-wise to the output of the linear combination, introducing non-linearity. This operation results in activation A_h of the hidden layer.

$$Y = \text{ReLU}\left(A_h W_o + b_o\right)$$

This equation gives the entire forward pass from input to output in the neural network.

2 Theory: Question 2 [12.5 pts]

Consider a CNN composed of three convolutional layers, each with 3×3 kernels, a stride of 2, and SAME padding. The lowest layer outputs 100 feature maps, the middle one outputs 200, and the top one outputs 400. The input images are RGB images of 200×300 pixels. What is the total number of parameters in the CNN? Explain your answer.

Layer	Input Channels	Output Maps	Filter Size	Total Parameters
1	3	100	3×3	$(3 \times 3 \times 3) \times 100 + 100$
2	100	200	3×3	$(3 \times 3 \times 100) \times 200 + 200$
3	200	400	3×3	$(3 \times 3 \times 200) \times 400 + 400$
			Total Parameters	903400

We can approach this problem by using the formula for the paramaters of a convolutional layer:

$$\text{Parameters} = (\text{Filter Width} \times \text{Filter Height} \times \text{Input Channels}) \times \text{Number of Filters} + \text{Number of Filters}$$

For each layer, $XW_a + B_a$ represents the operation, where X is the input size, W_a is the weight matrix, and B_a is the bias vector. Calculating this for each layer:

- For the first layer, X is the kernel size multiplied by the number of input channels (RGB channels), so $X = (3 \times 3 \times 3)$. The weights ($W_a$) are 100 since there are 100 filters. Adding the bias (one for each output feature map), gives us:
$$(3 \times 3 \times 3) \times 100 + 100 = 2800$$
- For the second layer, we use 100 input channels and 200 output maps:
$$(3 \times 3 \times 100) \times 200 + 200 = 180200$$
- For the third layer, we use 200 input channels and 400 output maps:
$$(3 \times 3 \times 200) \times 400 + 400 = 720400$$

Summing these results gives the total number of parameters in the CNN:

$$2800 + 180200 + 720400 = 903400$$

So, the CNN has a total of 903,400 parameters.

3 Theory: Question 3 [25 pts]

Given Equations from Batch Normalization (Sergey Ioffe and Christian Szegedy):

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \quad // \text{ Mean of the mini-batch}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \quad // \text{ Variance of the mini-batch}$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad // \text{ Normalized value}$$

$$y_i = \gamma \hat{x}_i + \beta \quad // \text{ Scale and shift}$$

Calculations:

a. $\frac{\partial f}{\partial \gamma}$:

Using the equation for y_i:

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

b. $\frac{\partial f}{\partial \beta}$:

From the equation $y_i = \gamma \hat{x}_i + \beta$:

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

c. $\frac{\partial f}{\partial \hat{x}_i}$:

From $y_i = \gamma \hat{x}_i + \beta$:

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

d. $\frac{\partial f}{\partial \sigma^2}$:

From the equation of $\hat{x}_i$:

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

e. $\frac{\partial f}{\partial \mu}$:

We consider:

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

f. $\frac{\partial f}{\partial x_i}$:

Considering both $\frac{\partial \hat{x}_i}{\partial x_i}$ and how x_i affects μ and σ^2:

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

4 Practical [50 pts]

This project implements several variants of the LeNet-5 architecture (originally described in LeCun et al., 1998) for classifying images from the FashionMNIST dataset. The variants include:

- Baseline LeNet-5 (No regularization)
- LeNet-5 with Dropout
- LeNet-5 with L2 Regularization
- LeNet-5 with Batch Normalization

- Optimized LeNet-5 with L2 Regularization

The goal is to compare the performance of these regularization techniques and achieve at least 88% accuracy on the test set.

(The reason for the optimized L2 Regularization model with tuned hyperparameters is due to the original L2 Regularization model not scoring over 88% on our training or test.)

4.1 Architecture Modifications

The LeNet-5 architecture was originally designed for the MNIST dataset. Primary changes to adapt this architecture to FashionMNIST are:

- Input dimensions remain the same (28x28 grayscale images).
- Modified activation functions to use Tanh for feature extraction layers.
- Added variants with Dropout, L2 Regularization (weight decay), and Batch Normalization (per Prof. Elor's HW1 reqs).

4.1.1 Specific Modifications

- **Dropout:** Introduced a Dropout layer (with a dropout rate of 0.5) in the fully connected part of the network to mitigate overfitting.
- **Batch Normalization:** Added Batch Normalization after convolutional layers to stabilize learning and accelerate training.
- **L2 Regularization:** Applied weight decay during optimization to penalize large weights, improving generalization.

4.2 Training Settings and Hyperparameters

4.2.1 Hyperparameters

- **Batch Size:** 64 (selected based on memory constraints and performance trade-offs).
- **Learning Rate:** 0.001, a common default that worked well without causing the training to become unstable.
- **Optimizer:** Adam for all models (Prof. Elor mentioned in lecture on 08/10/24 that only using Adam is okay for this).
- **Epochs:** Baseline, Dropout, Batch Norm, and initial L2 models were trained for 12 epochs. The optimized L2 model was trained for 20 epochs to achieve improved convergence.
- **Weight Decay:** Set to 0.01 for all models, reduced to 0.001 for the optimized model to balance underfitting and overfitting.

4.2.2 Train/Validation Split

- **Split Ratio:** 90% training, 10% validation. Referenced this SO post.
- **Method:** Used PyTorch's `random_split` to ensure a fair and random distribution of data into training and validation sets.

4.3 Training Instructions

To train the models with my hyperparameters, use the following commands:

4.3.1 Baseline model

```
train_acc_baseline, val_acc_baseline = train_model(
    model=model_baseline,
    train_loader=train_dataloader,
    val_loader=val_dataloader,
    epochs=12,
    optimizer=Adam,
    accuracy=Accuracy(task='multiclass', num_classes=10)
)
```

4.3.2 Dropout model

```
train_acc_dropout, val_acc_dropout = train_model(
    model=model_dropout,
    train_loader=train_dataloader,
    val_loader=val_dataloader,
    epochs=12,
    optimizer=Adam,
    accuracy=Accuracy(task='multiclass', num_classes=10)
)
```

4.3.3 L2 Regularization model

```
train_acc_l2, val_acc_l2 = train_model(
    model=model_l2,
    train_loader=train_dataloader,
    val_loader=val_dataloader,
    epochs=12,
    optimizer=Adam,
    accuracy=Accuracy(task='multiclass', num_classes=10),
    weight_decay=0.01
)
```

4.3.4 Optimized L2 Regularization model

```
train_acc_l2_optimized, val_acc_l2_optimized = train_model(
    model=model_l2,
    train_loader=train_dataloader,
    val_loader=val_dataloader,
    epochs=20,  # Increased epochs for better convergence
    optimizer=Adam,
    accuracy=Accuracy(task='multiclass', num_classes=10),
    weight_decay=0.001  # Reduced weight decay for better generalization
)
```

4.3.5 Batch Normalization model

```
train_acc_batch_norm, val_acc_batch_norm = train_model(
    model=model_batch_norm,
    train_loader=train_dataloader,
    val_loader=val_dataloader,
    epochs=12,
    optimizer=Adam,
    accuracy=Accuracy(task='multiclass', num_classes=10)
)
```

4.4 Saving the Weights

To save the weights of the trained models, use the following commands:

4.4.1 Baseline Model

```
torch.save(model_baseline.state_dict(), MODEL_SAVE_PATHS['baseline'])
```

4.4.2 Dropout Model

```
torch.save(model_dropout.state_dict(), MODEL_SAVE_PATHS['dropout'])
```

4.4.3 L2 Regularization Model

```
torch.save(model_l2.state_dict(), MODEL_SAVE_PATHS['l2_regularization'])
```

4.4.4 Optimized L2 Regularization Model

```
torch.save(model_l2.state_dict(), MODEL_PATH / "lenet5v1_l2_optimized.pth")
```

4.4.5 Batch Normalization Model

```
torch.save(model_batch_norm.state_dict(), MODEL_SAVE_PATHS['batch_norm'])
```

4.5 Testing with Saved Weights

To test the models with saved weights, use the following commands:

4.5.1 Baseline Model

```
model_baseline_loaded = LeNet5V1()
model_baseline_loaded.load_state_dict(torch.load(MODEL_SAVE_PATHS['baseline']))
```

4.5.2 Dropout Model

```
model_dropout_loaded = LeNet5V1(use_dropout=True)
model_dropout_loaded.load_state_dict(torch.load(MODEL_SAVE_PATHS['dropout']))
```

4.5.3 L2 Regularization Model

```
model_l2_loaded = LeNet5V1()
model_l2_loaded.load_state_dict(torch.load(MODEL_SAVE_PATHS['l2_regularization']))
```

4.5.4 Optimized L2 Regularization Model

```
model_l2_optimized_loaded = LeNet5V1()
model_l2_optimized_loaded.load_state_dict(torch.load(MODEL_PATH / "lenet5v1_l2_optimized.pth"))
```

4.5.5 Batch Normalization Model

```
model_batch_norm_loaded = LeNet5V1(use_batch_norm=True)
model_batch_norm_loaded.load_state_dict(torch.load(MODEL_SAVE_PATHS['batch_norm']))
```

4.6 Summary of Results

The graphs below showcase the training and test accuracies over epochs for each model:

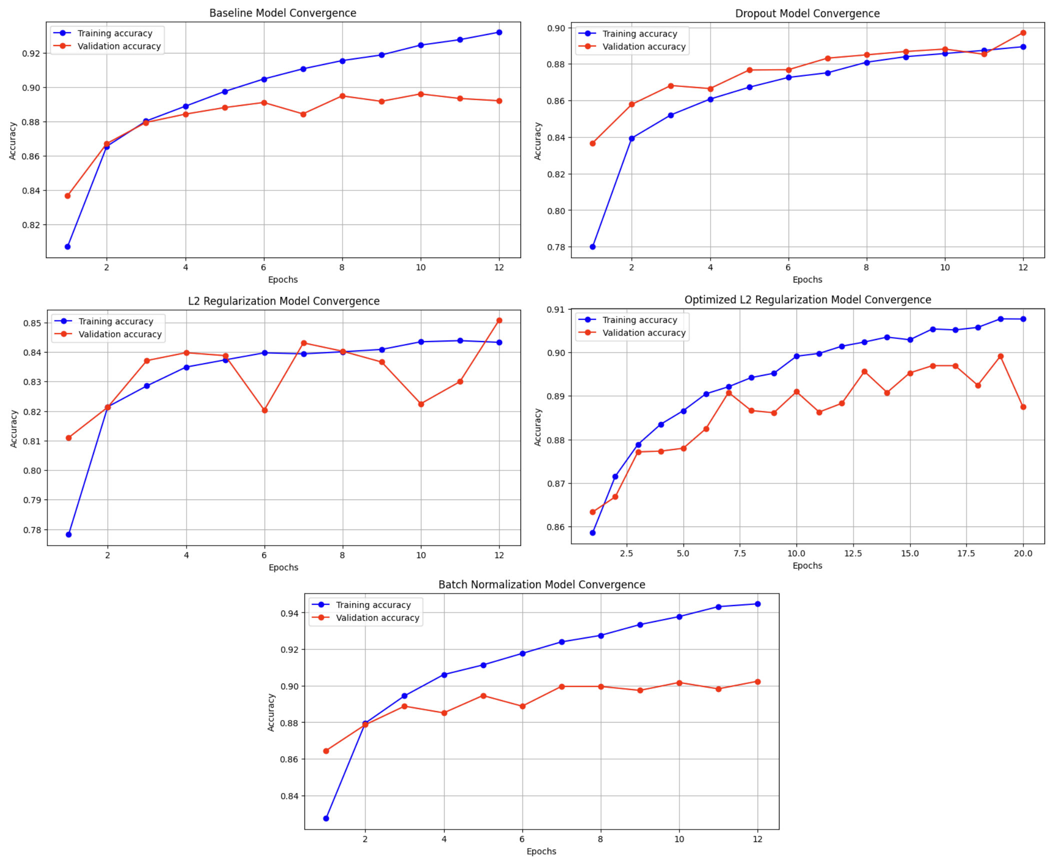

Figure 1: Training and validation accuracies over epochs for each model.

The table below summarizes the final training and validation accuracies for each model:

Model	Train Accuracy	Validation Accuracy
Baseline	0.932076	0.892121
Dropout	0.889422	0.897108
L2 Regularization	0.843269	0.850731
Optimized L2 Regularization	0.907638	0.887467
Batch Normalization	0.944757	0.902427

Table 1: Final training and validation accuracies for each model.

4.7 Analysis and Conclusions

- **Baseline Model:** Achieved a high training accuracy of 93.2%, but the validation accuracy was slightly lower at 89.2%, indicating some overfitting.
- **Dropout Model:** Reduced overfitting compared to the baseline, with a slightly higher validation accuracy (89.7%) than its training accuracy (88.9%), suggesting that dropout effectively improved generalization.
- **L2 Regularization:** Showed lower performance in both training (84.3%) and validation (85.1%) accuracies, indicating that the chosen regularization strength might have been too high, leading to underfitting.
- **Optimized L2 Regularization Model:** Improved over the initial L2 regularization, achieving 90.8% training accuracy and 88.7% validation accuracy. This suggests that fine-tuning the regularization parameters positively impacted model performance, though it still fell short of the baseline in validation.

- **Batch Normalization:** Achieved the highest performance among all models, with a training accuracy of 94.5% and validation accuracy of 90.2%. This indicates that batch normalization was effective in improving model stability and accelerating training.

Overall, our Batch Normalization model was the best-performing, giving us back the highest accuracy on both the training and validation sets. Our Optimized L2 Regularization Model showed improvements over the basic L2 approach, but still did not surpass the baseline model's performance, which highlights the challenges in balancing regularization strength and training duration of these models.

For more details, code, and relevant citations, please see the Github.