Programming Assignment 1

20213073 Donggyu Kim

May 20, 2021

0 Hyperparameters

```
learning_rate = 0.001
num_epochs = 50
l_lambda = 0.01 #1e-5
```

I used the above hyperparameter set in every executions for fair comparison. Since our task is just a binary classification, I think 50 epoch is enough for a model to converge. `l_lambda` is the weight for L1/L2 regularization: 0.01 for L2 and 10^{-5} for L1. The reason that I used the different values is described in Section 2.3.

1 Classification Model

1.0 Setting

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate,\
                                                momentum=0.9)
```

To compare the three classification models under the same condition, I equally used the above SGD with Momentum optimizer, without regularization (though the given VGG and ResNet models use Batch Normalization, which has inherent regularization effect).

1.1 MLP

```
Epoch: 050/050 | Train: 97.517% | Valid: 96.240%
Time elapsed: 29.26 min
Total Training Time: 29.26 min
```

```
with torch.set_grad_enabled(False): # save memory during inference
    print('Test accuracy: %.2f%%' % (compute_accuracy(model, test_loader)))
```

```
Test accuracy: 95.33%
```

Figure 1: Training result of MLP

This is the result of training MLP for 50 epochs. It took 29.26 minutes and gets 97.52%/96.24%/95.33% of train/valid/test accuracy. See Fig 4 for its train loss and train/validation accuracy curves.
Let's call this *vanilla MLP* for further comparisons in the regularization and optimization sections.

1.2 VGG-16

```
Epoch: 050/050 | Train: 100.000% | Valid: 97.594%
Time elapsed: 39.61 min
Total Training Time: 39.61 min
```

```
with torch.set_grad_enabled(False): # save memory during inference
    print('Test accuracy: %.2f%%' % (compute_accuracy(model, test_loader)))
```

```
Test accuracy: 96.91%
```

Figure 2: Training result of VGG-16

VGGNet has several variants according to the number of layers. I chose VGG-16 among them. As Fig 2 shows, VGG-16 took 39.61 minutes for training, and after 50 epochs, it has 100.0%/97.59%/96.91% of train/valid/test accuracy. VGG-16 took more time but get better accuracy compare to MLP. See Fig 4 for its train loss and train/validation accuracy curves.

1.3 ResNet-18

```
Epoch: 050/050 | Train: 100.000% | Valid: 97.493%
Time elapsed: 65.31 min
Total Training Time: 65.31 min
```

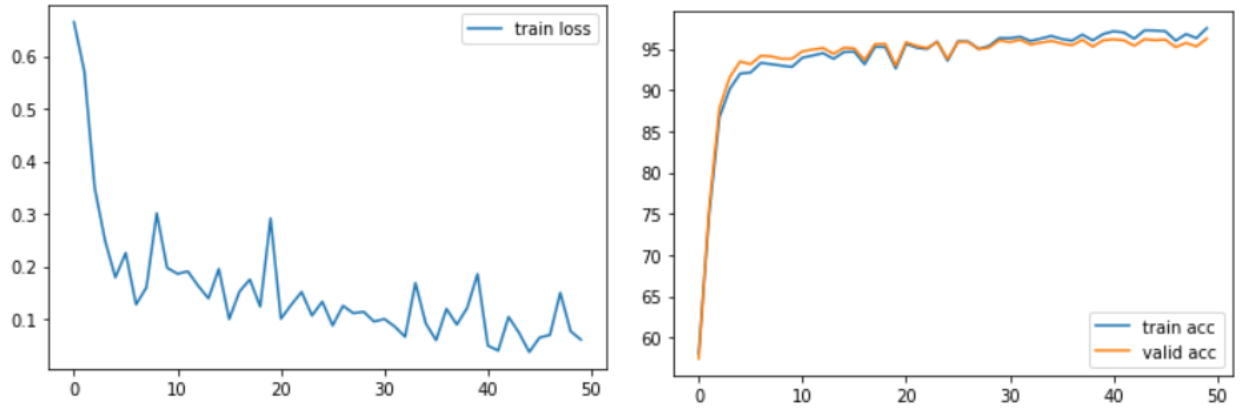
```
with torch.set_grad_enabled(False): # save memory during inference
    print('Test accuracy: %.2f%%' % (compute_accuracy(model, test_loader)))
```

```
Test accuracy: 96.79%
```

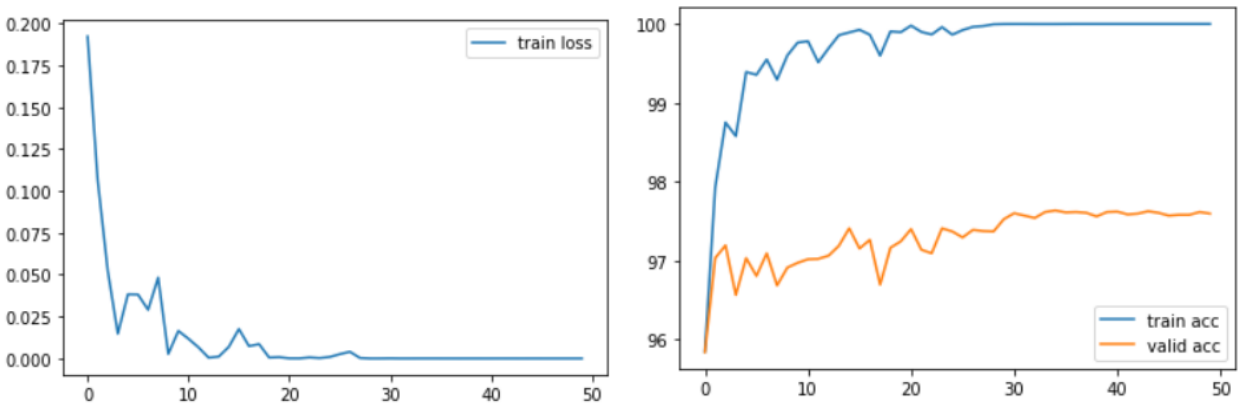
Figure 3: Training result of ResNet-18

ResNet also has several variants, but the instruction specified ResNet-18. As Fig 3 shows, ResNet-18 took 65.31 minutes for training. After 50 epochs, it has 100.0%/97.49%/96.79% of train/valid/test accuracy. It took more time compare to VGG-16 but get a bit lower validation and test accuracies. See Fig 4 for its train loss and train/valid accuracy curves.

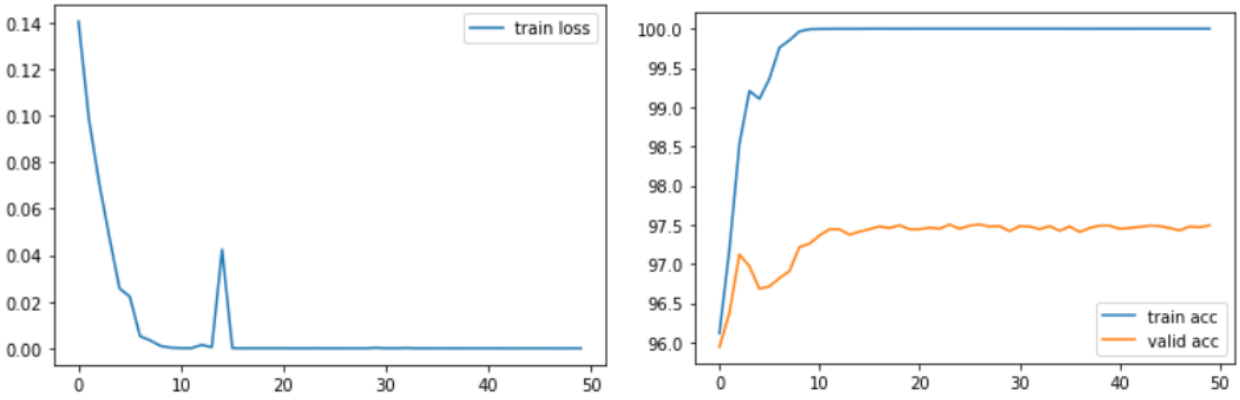
1.4 Comparison



(a) MLP



(b) VGG-16



(c) ResNet-18

Figure 4: Loss and accuracy curves

Both MLP and the CNN models show very high accuracy. Also, there is no sign of overfitting, at least for the 50 epochs. This should be because the task is too easy - binary classification. Though, MLP showed significant differences with the CNN models. It has higher training loss and lower

train/validation accuracy in the first few epochs; MLP had about 50% accuracy while VGG-16 and ResNet-18 reached 90% in their first epoch. Also, MLP has almost same training and validation accuracies, while the CNN models show about 3% gap between them. This can be the evidence that the CNN models learns better from the training data. On the other hand, VGG-16 and ResNet-18 does not differ so much. Although ResNet-18 has lower training loss in the first few epochs, the loss converges to almost 0 for both of them at the end. Interesting point is that ResNet-18 has smoother accuracy curves. I guess this is because the skip connection has stabilizing effect in the training.

2 Regularization

2.0 Setting

In this part, MLP is used as the base model and SGD with Momentum is used as the optimizer. The given VGG and ResNet are using Batch Normalization, which already has regularization effect. Furthermore, Dropout is used on FC layers in general, but the given CNN models are using Average Pooling instead of FC layers. This is why I chose MLP as the base model to compare the regularization methods.

2.1 Dropout

```
class MLP(nn.Module):
    def __init__(self, drop = 0):
        super().__init__()
        layers = []
        in_dim = 3*32*32
        for dim in [1024, 512, 256, 128]:
            out_dim = dim
            layers.append(nn.Linear(in_dim, out_dim))
            if drop > 0:
                layers.append(nn.Dropout(drop))
            layers.append(nn.ReLU())
            in_dim = out_dim
        layers.append(nn.Linear(out_dim, 2))
        self.network = nn.Sequential(*layers)
```

I added the parameter `drop`, a Dropout rate, to construct MLP with Dropout. If `drop` is bigger than 0, each layer of the MLP performs Dropout.

```
model = mlp.MLP(drop=0.5)
```

Dropout rate is set to 0.5 in general, so I also used 0.5 here.

```
Epoch: 050/050 | Train: 95.456% | Valid: 95.747%
Time elapsed: 27.42 min
Total Training Time: 27.42 min
```

```
with torch.set_grad_enabled(False): # save memory during inference
    print('Test accuracy: %.2f%%' % (compute_accuracy(model, test_loader)))
```

```
Test accuracy: 94.89%
```

Figure 5: Training result of MLP with Dropout

This is the result of training MLP with Dropout for 50 epochs. It took 27.42 minutes and gets 95.46%/95.75%/94.89% of train/valid/test accuracy. Since regularization is to avoid overfitting of a model and get better results for unseen (test) data, it is quite disappointing that MLP with Dropout did not beat vanilla MLP's test accuracy. Though, it is comforting that MLP with Dropout has almost same training and validation accuracies. See Fig 11 for its train loss and train/validation accuracy curves.

2.2 L2 Regularization

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate,\
                                momentum=0.9, weight_decay=l_lambda)
```

It is possible to implement L2 regularization explicitly, but is is simply done by assigning `weight_decay` value. As I mentioned at the very first of this document, I set `l_lambda` to 0.01 for L2 regularization.

```
Epoch: 050/050 | Train: 94.930% | Valid: 95.550%
Time elapsed: 27.46 min
Total Training Time: 27.46 min
```

```
with torch.set_grad_enabled(False): # save memory during inference
    print('Test accuracy: %.2f%%' % (compute_accuracy(model, test_loader)))
```

```
Test accuracy: 94.44%
```

Figure 6: Training result of MLP with L2 regularization

This is the result of training MLP with L2 regularization for 50 epochs. It took 27.46 minutes and gets 94.93%/95.55%/94.44% of train/valid/test accuracy. It is quite surprising that the validation accuracy is higher than the training accuracy. Same as Dropout, MLP with L2 regularization did not beat vanilla MLP. See Fig 11 for its train loss and train/validation accuracy curves. To check whether L2 regularization worked well or not, I plotted L2 norm changes for each model (Fig 9).

2.3 L1 Regularization

```
l1_norm = sum(torch.norm(param, p=1) for param in model.parameters())
l2_norm = sum(torch.norm(param) for param in model.parameters())
cost += l_lambda * l1_norm
```

There is no implicit L1 regularization option in the optimizer. Therefore, I added the above code for L1 regularization. The first 2 lines compute L1 and L2 norms of each model, and the results are in Fig. 9 and Fig. 10. The last line must be commented out when the model is not using L1 regularization.

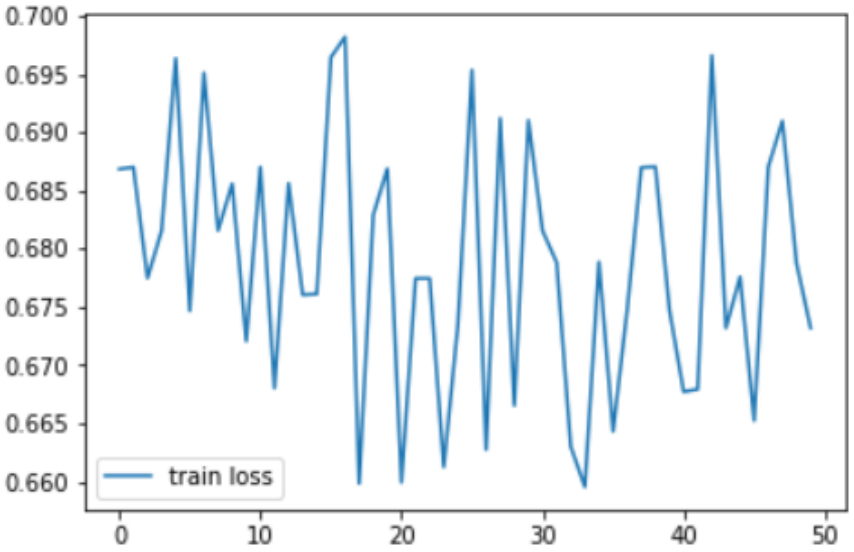


Figure 7: MLP with L1 regularization failed to converge

As in L2 regularization, I first used `l_lambda` value 0.01 for L1 regularization. However, the model did not converged well (Fig. 7) and got only 57% of training accuracy. This is because L1 norm is too large ($> 10^5$) compare to the prediction loss term, so the model only decreased its weights not the training loss. Thus, I changed `l_lambda` value to 10^{-5} and got the appropriate result this time.

```
Epoch: 050/050 | Train: 97.386% | Valid: 96.366%
Time elapsed: 27.52 min
Total Training Time: 27.52 min
```

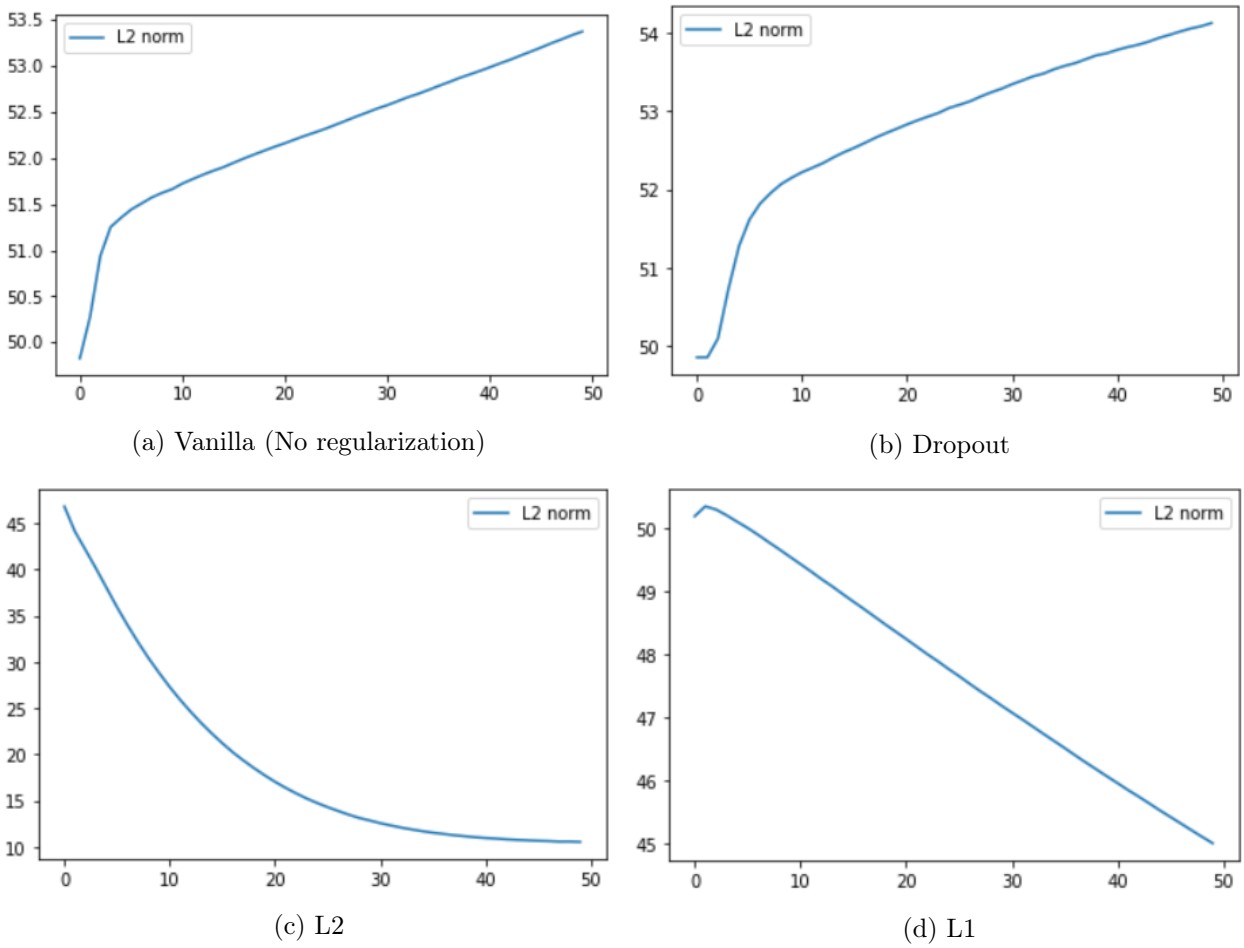
```
with torch.set_grad_enabled(False): # save memory during inference
    print('Test accuracy: %.2f%%' % (compute_accuracy(model, test_loader)))
```

```
Test accuracy: 95.35%
```

Figure 8: Training result of MLP with L1 regularization

This is the result of training MLP with L1 regularization for 50 epochs. It took 27.52 minutes and gets 97.39%/96.37%/95.35% of train/valid/test accuracy. Dropout and L2 regularization did not beat vanilla MLP, but L1 regularization slightly beat vanilla MLP's test accuracy. See Fig 11 for its train loss and train/validation accuracy curves. To check whether L1 regularization worked well or not, I plotted L1 norm changes for each model (Fig 10).

2.4 Comparison



(a) Vanilla (No regularization)

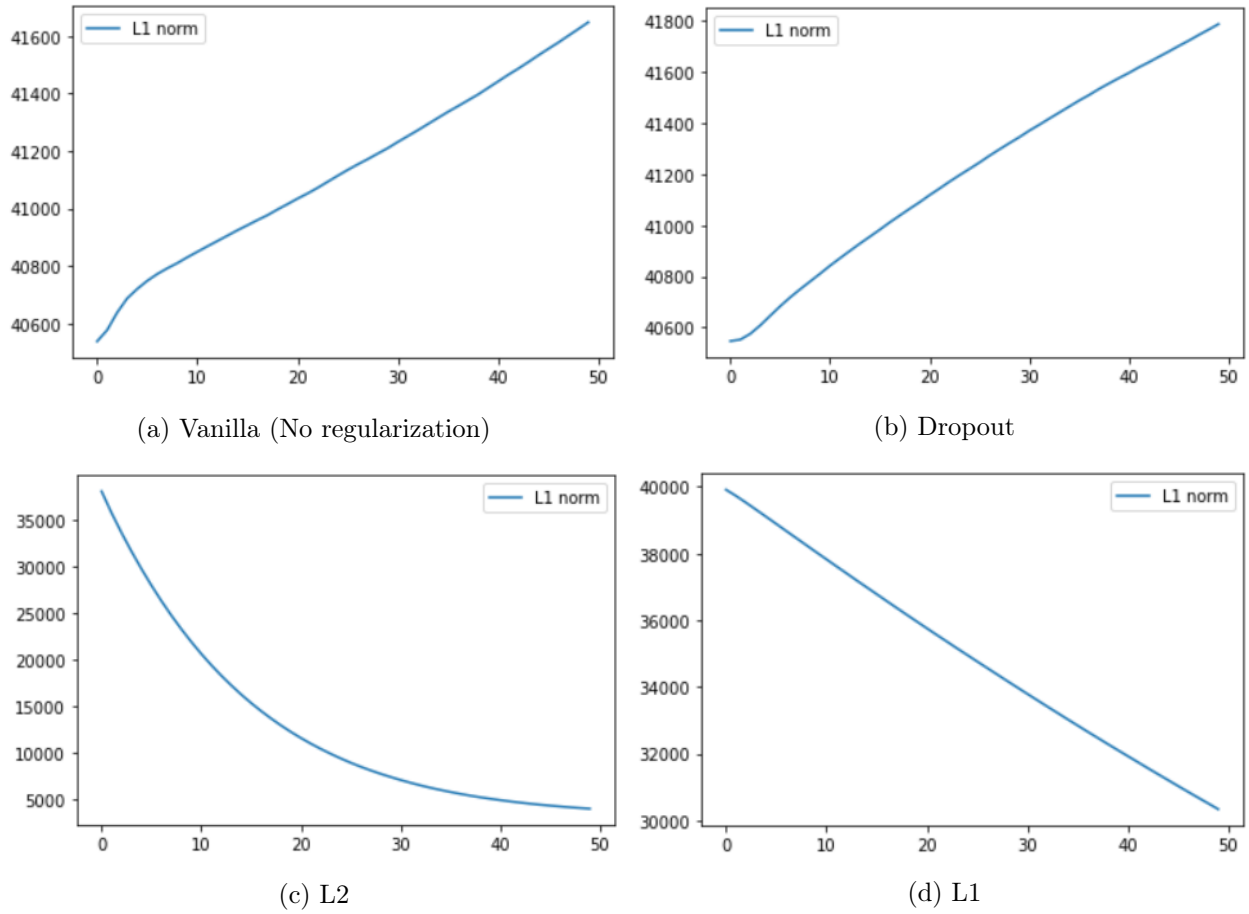
(b) Dropout

(c) L2

(d) L1

Figure 9: L2 norm

While L2 norm increases for vanilla MLP and MLP with Dropout as training goes on, L2 regularization significantly reduced L2 norm. L1 regularization also leads to decrease in L2 norm.



(a) Vanilla (No regularization)

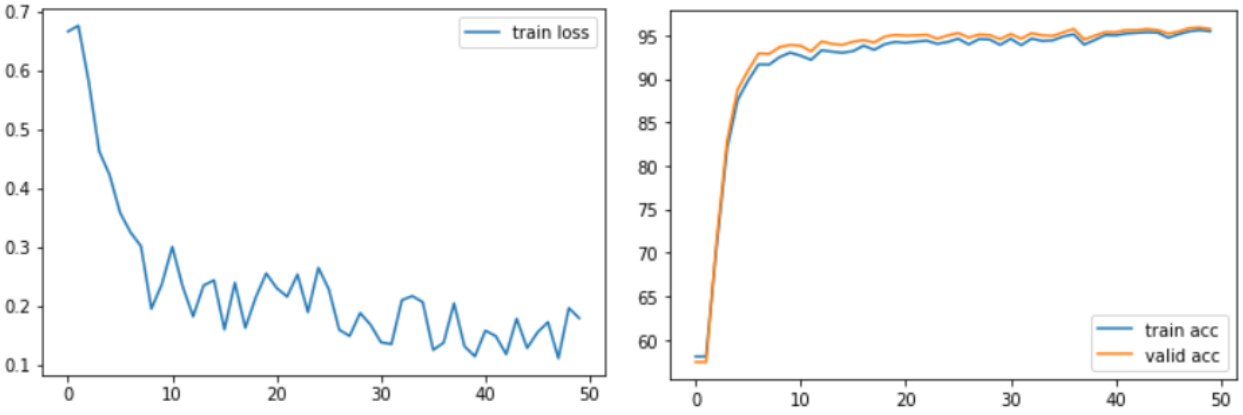
(b) Dropout

(c) L2

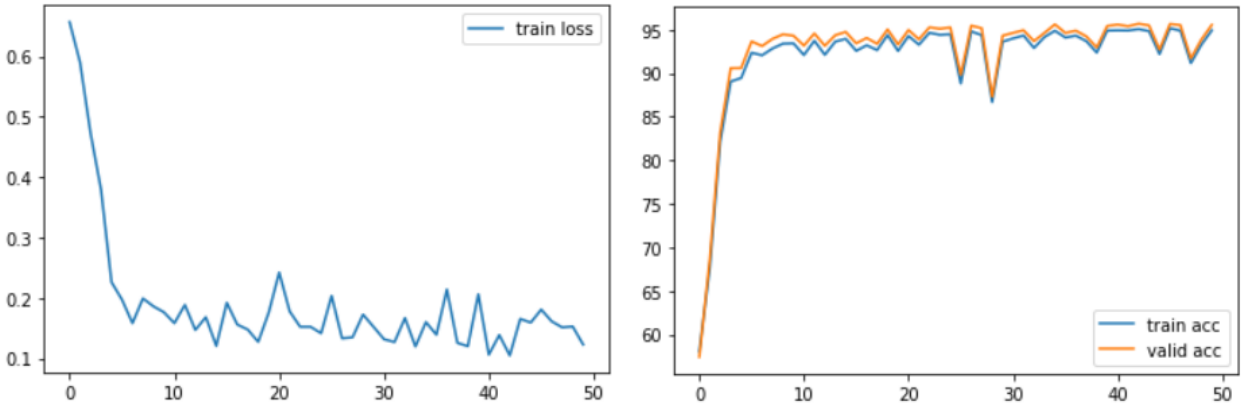
(d) L1

Figure 10: L1 norm

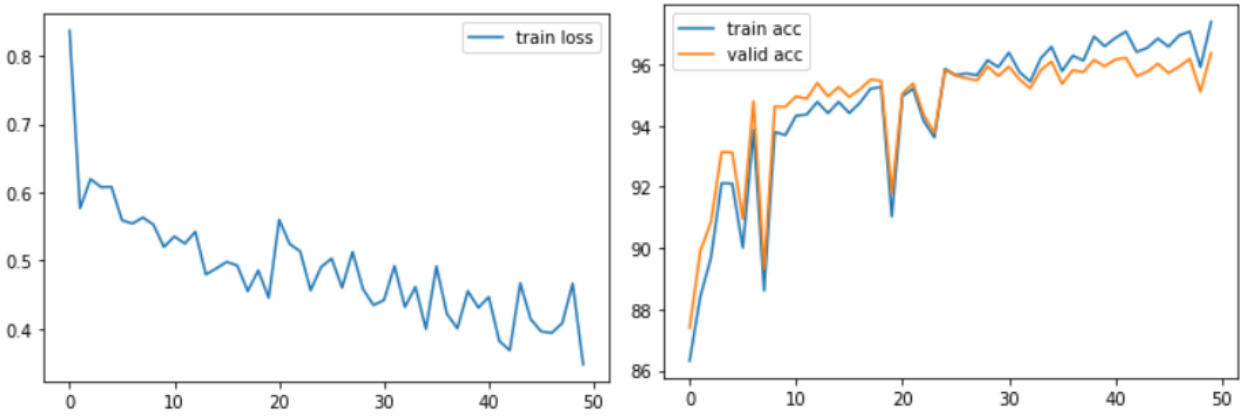
While L1 norm increases for vanilla MLP and MLP with Dropout as training goes on, L1 regularization significantly reduced L1 norm. L2 regularization also leads to decrease in L1 norm. As I previously mentioned, L1 norm is very large so deciding `l_lambda` should take this into account.



(a) Dropout



(b) L2



(c) L1

Figure 11: Loss and accuracy curves

Compare to the result in Fig. 4 (a), Dropout does not show any significant differences while L1 and L2 regularization shows a bit noisy accuracy curves. I think this is because weight decaying disturbs the model for regularization, while Dropout stabilizes the model by the bagging effect.

3 Optimization

3.0 Setting

In this part, MLP is used as the base model, without regularization.

3.1 Momentum

This is vanilla MLP that I've already discussed. See Section 1.1 for the result.

3.2 AdaGrad

```
optimizer = torch.optim.Adagrad(model.parameters(), lr=learning_rate)
```

This time, the MLP is trained with AdaGrad optimizer.

```
Epoch: 050/050 | Train: 96.697% | Valid: 95.596%
Time elapsed: 27.39 min
Total Training Time: 27.39 min
```

```
with torch.set_grad_enabled(False): # save memory during inference
    print('Test accuracy: %.2f%%' % (compute_accuracy(model, test_loader)))
```

```
Test accuracy: 94.68%
```

Figure 12: Training result of MLP with AdaGrad optimizer

This is the result of training MLP for 50 epochs. It took 27.39 minutes and gets 96.70%/95.60%/94.68% of train/valid/test accuracy. See Fig 14 for its train loss and train/validation accuracy curves.

3.3 Adam

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

This time, the MLP is trained with AdaGrad optimizer.

```
Epoch: 050/050 | Train: 97.610% | Valid: 95.555%
Time elapsed: 27.45 min
Total Training Time: 27.45 min
```

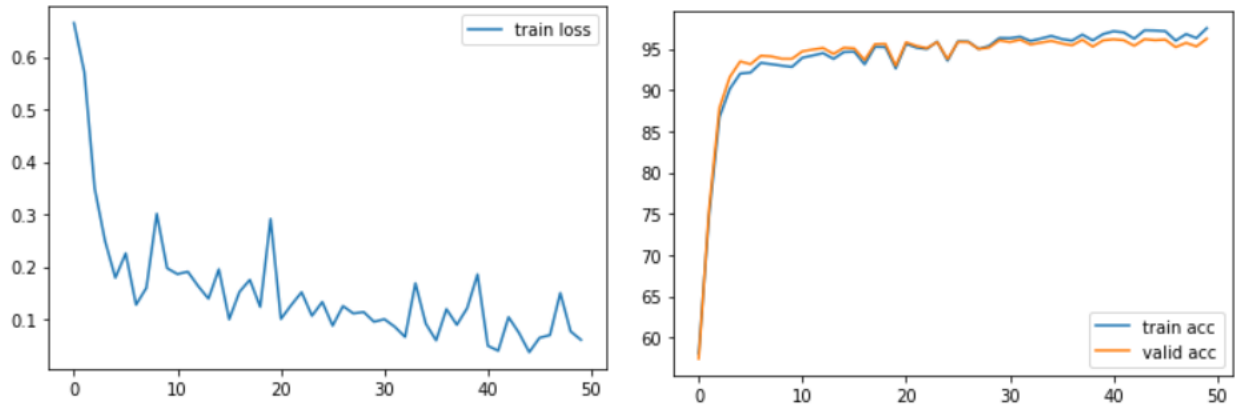
```
with torch.set_grad_enabled(False): # save memory during inference
    print('Test accuracy: %.2f%%' % (compute_accuracy(model, test_loader)))
```

```
Test accuracy: 94.94%
```

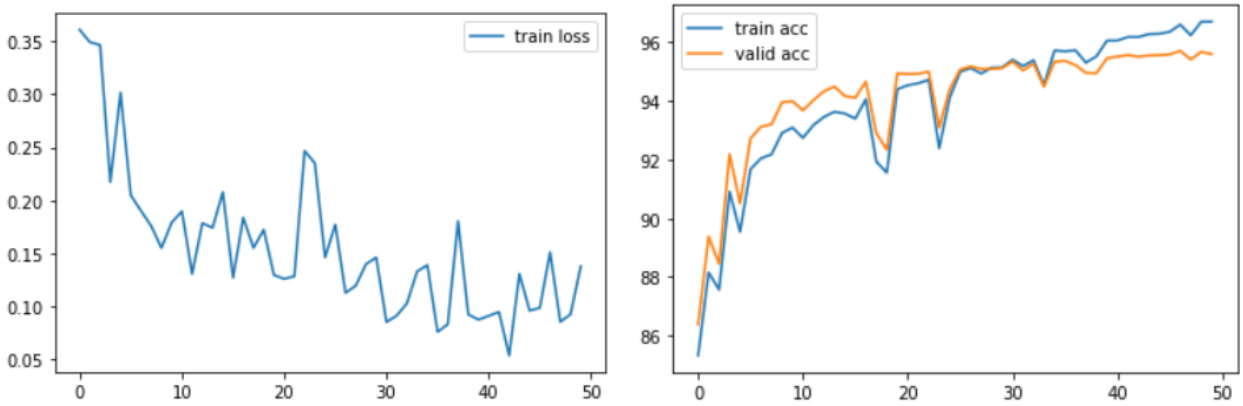
Figure 13: Training result of MLP with Adam optimizer

This is the result of training MLP for 50 epochs. It took 27.45 minutes and gets 97.61%/95.56%/94.94% of train/valid/test accuracy. See Fig 14 for its train loss and train/validation accuracy curves.

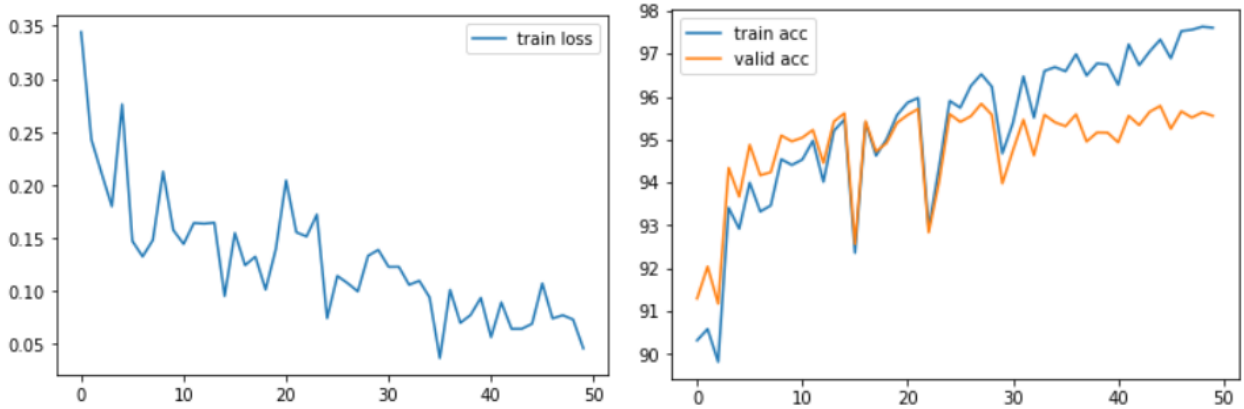
3.4 Comparison



(a) Momentum



(b) AdaGrad



(c) Adam

Figure 14: Loss and accuracy curves

AdaGrad adapts the learning rates of the model parameters by scaling them inversely proportional to the square root of the sum of all their historical. Therefore, the model is trained very fast in the first few epochs and got up to 90% accuracy. However, shrinking the learning rate based on the entire history of the gradient makes the learning rate too small. It appears directly after 5 epochs. Adam is a combination of RMSProp and momentum. The learning rate is bounded, and so the model's training accuracy increased continuously.