CSCI 3485 Lab 3

Rafael Almeida

October 3, 2024

1 Introduction

In this report, four experiments are conducted to better understand convolutional neural networks (CNN). 1) Compares a similarly structured CNN with a Multilayer Perceptron (MLP). 2) Compares similarly structured CNNs with varying kernel sizes in each layer. 3) Compares similarly structured CNNs with varying counts of filters per layer. 4) Compares the CNN in experiment 1 but with the addition of batch normalization layers vs. dropout layers in between each of the original layers.

These four experiments will assist in gauging the pros and cons of CNNs vs. MLPs and the effects of kernel size, filter count, batch normalization, and dropout on CNNs' training time and accuracy. Understanding the relationship between the aforementioned parameters is crucial to decreasing training time and increasing model accuracy. Decreasing training time and increasing model accuracy is vital in understanding how to use less computational power for more performance, especially in a world with an ever-increasing number of ever-larger deep learning models.

2 Methodology

All of the following experiments are on the Fasion MNIST dataset. Additionally, the general architecture of the networks was based on findings from Lab 2. Lab 2 concluded that networks performed the best when the number of layers was at least the number of classes. Therefore, whenever possible, all of the networks have ten layers. Additionally, the CNNs always have three linear layers at the end because Lab 2 showed that, for many classes (6), an MLP needed at least three layers to begin correctly classifying at all.

All of the networks after experiment 1 are based on its CNN. In experiment 1, the final number of filters was arbitrarily chosen to be 128 and to double every layer and the kernel size to be three. Thus, all subsequent CNNs have double the number of filters per convolutional layer until they reach 128 and have a kernel of size three. In experiment 1, the units per layer in the CNN were chosen so that the total number of parameters in the network was approximately one million. The exact number of units per layer in the linear part was used in all subsequent CNNs.

All networks were trained until the change in loss function was less than a delta of 0.01. Although it appears large, this minimum delta was chosen because the average loss value of the networks trained in class decreased at about 0.5 per epoch for five epochs. Therefore, a minimum delta of 0.01 is sufficient to train a network optimally without unnecessary computation. Additionally, a fixed number of epochs was not used because it can give better insights than comparing networks trained in the same number of epochs. When experimenting with testing time and accuracy with a fixed number of epochs, one can infer which network will perform best with the given amount of computational power. However, even with an unfixed number of epochs, you can infer this by dividing the training time by the number of epochs. Additionally, you can better infer which network has a higher accuracy potential before hitting diminishing returns in training. Knowing which network has a higher accuracy ceiling is a better metric for determining which network is better than simply the accuracy after a fixed number of epochs.

Experiment 1 trains the CNN described above and an MLP with ten layers and a fixed number of units per layer. The number of units per layer was chosen such that the total number of parameters in the MLP is similar to that of the CNN. This experiment is relevant because it will give insights into the pros and cons of CNN vs. MLP.

Experiment 2 trains five CNNs with varying kernel sizes (2, 3, 5, 7, 9). The convolutional layers in these CNNs were reduced to three because, with a maximum kernel size of nine, the images with dimensions 28x28 would become too small after repeated convolution operations. Therefore, three layers ensure that the smallest dimension reached is 4x4 $(28 - (9 - 1) * 3)$. A layer count dependent on the kernel size could have been used to maximize the number of layers. However, a variable number of layers would introduce more variables, making it hard to make any inferences from the changing kernel sizes. This experiment is important because it gives insights into the effects of varying kernel sizes in CNNs

Experiment 3 trains five CNNs with varying numbers of filters per layer (5, 10, 15, 20, 25). Changing the number of filters per layer will give insights into its effects on a CNN's performance.

Experiment 4 trains two CNNs completely based on the CNN in experiment 1. The only difference in the CNNs in experiment 4 from experiment 1 is that one has a batch normalization layer between each convolutional and linear layer, and the other has dropout layers. The dropout rate was chosen to be 0.5. The intuition behind the dropout of 0.5 is that dropout allows the training of a network based on a probability distribution. Having a weight be dropped at a rate of 0.5 maximizes the variance in this probability distribution. Therefore, theoretically, it is a suitable value to minimize overfitting. This experiment is important because it will demonstrate the difference in performance between the two strategies of decreasing overfitting, batch normalization, and dropout.

3 Results

3.1 Experiment 1: CNN vs. MLP

	CNN	MLP
Parameter Count	1,017,130	1,026,970
Training Time (s)	105	63
Epochs	12	8
Time per Epoch	8.75	7.88
Accuracy	0.89	0.87

3.2 Experiment 2: Kernel Sizes

Kernel Size	2	3	5	7	9
Parameter Count	2,282,442	1,828,458	1,175,658	863,082	890,730
Training Time (s)	77	74	88	90	82
Epochs	10	10	12	12	11
Time per Epoch	7.7	7.4	7.3	7.5	7.5
Accuracy	0.91	0.91	0.91	0.90	0.90

3.3 Experiment 3: Filter Counts

Filter Count	5	10	15	20	25
Parameter Count	38,170	77,460	119,000	162,790	208,830
Training Time (s)	74	82	82	74	115
Epochs	9	10	10	9	14
Time per Epoch	8.2	8.2	8.2	8.2	8.2
Accuracy	0.88	0.89	0.88	0.89	0.90

3.4 Experiment 4: Batch Normalization and Dropout

	Batch Norm	Dropout
Parameter Count	1,017,746	1,017,130
Training Time (s)	117	102
Epochs	11	11
Time per Epoch	10.62	9.31
Accuracy	0.92	0.79

4 Discussion

4.1 Experiment 1: CNN vs. MPL

Given examples seen in class, the hypothesis for experiment 1 was that the CNN would perform vastly better and train faster. However, the differing results in section 3.1 can be explained by the chosen parameters. Since the networks were architected to have a similar parameter count, their training time should be similar, and as seen, the time per epoch is very similar. The CNN may have longer times per epoch because of the added overhead of the convolution operation. However, as mentioned in section 2, inferences can still be made that the CNN performs better because it is executed for more epochs, showing that it has a higher potential of reaching better accuracies. Therefore, with very similarly structured and sized networks, a CNN network has the potential to outperform an MLP. Additional testing could be done to determine whether the CNN performs better with the same number of epochs as the MLP.

4.2 Experiment 2: Kernel Sizes

The results in section 3.2 are inconclusive. Although increasing the kernel size decreases the parameter count, it does not affect the training time per epoch or the accuracy. However, it would be reasonable to assume that the large values for kernel size lead to worse models that are also harder to train because of the slightly decreasing accuracy and increasing epochs. This correlation between increasing kernel size and worse models could be explained by how, as the kernel sizes increase, more and more information is extracted and condensed into one value. As kernel sizes increase, it is reasonable to conclude that this lossy process is detrimental to the amount of information the model has to make classifications.

4.3 Experiment 3: Filter Counts

The most exciting analysis from experiment 3 is that an increasing number of filters does not increase the training time per epoch. However, it does increase the highest potential accuracy since higher filter counts usually run for more epochs. A hypothesis for why, although there are more weights since there are more filters, the time per epoch is the same is that because there are so many layers with the same number of filters, PyTorch is doing some kind of optimization to compute these weights since they are likely to be very similar. Regardless, the data in experiment 3 does indicate that a greater number of filters correlates with slightly higher accuracies given more compute time.

4.4 Experiment 4: Batch Normalization and Dropout

The data in section 3.4 favor batch normalization strategies to minimize overfitting compared to dropout. This can be seen since the training time is virtually the same while the accuracy is drastically higher for the model with batch normalization. However, this test is not conclusive that dropout is not a good approach. Dropout in this scenario shows a slight benefit in compute time, but it may also be due to the dropout rate being too high and too many layers. The CNN used was the same as in experiment 1, with ten layers. A dropout rate of 0.5 for ten layers may be way beyond the point where the dropout strategy is beneficial. Nonetheless, this data implies that between the choice of a batch normalization layer or a dropout layer, one should choose batch normalization.

5 Conclusion

Overall, the data in section 3 and the analysis made in section 4 indicate that CNNs have superiority over similarly structured MLPs. In the context of CNNs, increasing the kernel size does not bring that many positive benefits. On the other hand, increasing the filter counts brought the most benefits in terms of model accuracy. Finally, it was shown that a CNN can, without intervention, lead to overfitting and hurting accuracy. Therefore, batch normalization is recommended. The data presented shows a strong preference for batch normalization versus dropout layers.

6 Code

GitHub Repository

6.1 main.py

```python
"""
Runs the experiments.
"""

from experiments import batch_norm_dropout, cnn_mlp, filter_count,
    kernel_size

cnn_mlp()
kernel_size()
filter_count()
batch_norm_dropout()
```

6.2 experiments.py

```python
"""
Defines the experiments to be run.
"""

from datetime import datetime
from time import time

from torch import cuda
from torch.backends import mps
from torchsummary import summary

from data import append_to_file, get_data_loaders
from model import build_model, get_parameter_count, test, train

if mps.is_available():
```

```python
    device = "mps"
elif cuda.is_available():
    device = "cuda"
else:
    device = "cpu"

print(f"Using device: {device}")

train_loader, test_loader = get_data_loaders()


def time_it(f):
    """
    Times the execution of a function.
    """

    def wrapper(*args, **kwargs):
        start = time()
        result = f(*args, **kwargs)
        return (time() - start, result)

    return wrapper


def cnn_mlp() -> None:
    """
    Builds and trains the models, and saves the data for the CNN vs MLP
     experiment.
    """

    # building
    cnn = build_model(
        classes=10,
        shape=(1, 28, 28),
        convoluted=[
            (4, 3),
            (8, 3),
            (16, 3),
            (32, 3),
            (64, 3),
            (128, 3),
        ],
        linear=[32768, 28, 28],
    )

    mlp = build_model(
        classes=10,
        shape=(1, 28, 28),
        convoluted=[],
        linear=[784] + [330] * 8,
```

```python
    )

    # training
    cnn_param_count = get_parameter_count(cnn)
    cnn_time, cnn_info = time_it(train)(cnn, train_loader, device)
    cnn_accuracy = test(cnn, test_loader, device)

    mlp_time, mlp_info = time_it(train)(mlp, train_loader, device)
    mlp_param_count = get_parameter_count(mlp)
    mlp_accuracy = test(mlp, test_loader, device)

    # saving data
    file_name = "cnn_mlp.txt"
    append_to_file(file_name, datetime.now())
    append_to_file(file_name, f"cnn param count: {cnn_param_count}")
    append_to_file(file_name, f"cnn time: {cnn_time}")
    append_to_file(file_name, f"cnn info: {cnn_info}")
    append_to_file(file_name, f"cnn accuracy: {cnn_accuracy}")
    append_to_file(file_name, f"mlp param count: {mlp_param_count}")
    append_to_file(file_name, f"mlp time: {mlp_time}")
    append_to_file(file_name, f"mlp info: {mlp_info}")
    append_to_file(file_name, f"mlp accuracy: {mlp_accuracy}")


def kernel_size() -> None:
    """
    Builds and trains the models, and saves the data for the kernel size
     experiment.
    """

    sizes = [2, 3, 5, 7, 9]

    # building
    models = []
    for size in sizes:
        model = build_model(
            classes=10,
            shape=(1, 28, 28),
            convoluted=[
                (32, size),
                (64, size),
                (128, size),
            ],
            linear=[128 * (28 - 3 * (size - 1)) ** 2, 28, 28],
        )
        models.append(model)

    # training
    parameter_counts = []
    for model in models:
```

```
        parameter_count = get_parameter_count(model)
        parameter_counts.append(parameter_count)

    training_times = []
    epochs = []
    for model in models:
        training_time, info = time_it(train)(model, train_loader, device)
        training_times.append(training_time)
        epochs.append(info["epochs"])

    accuracies = []
    for model in models:
        accuracy = test(model, test_loader, device)
        accuracies.append(accuracy)

    # saving data
    file_name = "kernel_size.txt"
    append_to_file(file_name, datetime.now())
    append_to_file(file_name, f"kernel sizes: {sizes}")
    append_to_file(file_name, f"parameter counts: {parameter_counts}")
    append_to_file(file_name, f"training times: {training_times}")
    append_to_file(file_name, f"epochs: {epochs}")
    append_to_file(file_name, f"accuracies: {accuracies}")


def filter_count() -> None:
    """
    Builds and trains the models, and saves the data for the filter count experiment.
    """

    filter_counts = [5, 10, 15, 20, 25]

    # building
    models = []
    for count in filter_counts:
        model = build_model(
            classes=10,
            shape=(1, 28, 28),
            convoluted=[
                (count, 3),
                (count, 3),
                (count, 3),
                (count, 3),
                (count, 3),
                (count, 3),
            ],
            linear=[count * (28 - 6 * (3 - 1)) ** 2, 28, 28],
        )
        models.append(model)
```

```python

    # training
    parameter_counts = []
    for model in models:
        parameter_count = get_parameter_count(model)
        parameter_counts.append(parameter_count)

    training_times = []
    epochs = []
    for model in models:
        training_time, info = time_it(train)(model, train_loader, device)
        training_times.append(training_time)
        epochs.append(info["epochs"])

    accuracies = []
    for model in models:
        accuracy = test(model, test_loader, device)
        accuracies.append(accuracy)

    # saving data
    file_name = "filter_count.txt"
    append_to_file(file_name, datetime.now())
    append_to_file(file_name, f"filter counts: {filter_counts}")
    append_to_file(file_name, f"parameter counts: {parameter_counts}")
    append_to_file(file_name, f"training times: {training_times}")
    append_to_file(file_name, f"epochs: {epochs}")
    append_to_file(file_name, f"accuracies: {accuracies}")


def batch_norm_dropout() -> None:
    """
    Builds and trains the CNN from the CNN vs. MLP experiment but with
     batch normalization and dropout layerrs
    """

    # building
    batch_norm = build_model(
        classes=10,
        shape=(1, 28, 28),
        convoluted=[
            (4, 3),
            (8, 3),
            (16, 3),
            (32, 3),
            (64, 3),
            (128, 3),
        ],
        linear=[32768, 28, 28],
        batch_norm=True,
    )
```

```

    dropout = build_model(
        classes=10,
        shape=(1, 28, 28),
        convoluted=[
            (4, 3),
            (8, 3),
            (16, 3),
            (32, 3),
            (64, 3),
            (128, 3),
        ],
        linear=[32768, 28, 28],
        dropout=0.5,
    )

    # training
    batch_norm_param_count = get_parameter_count(batch_norm)
    batch_norm_time, batch_norm_info = time_it(train)(
        batch_norm, train_loader, device
    )
    batch_norm_accuracy = test(batch_norm, test_loader, device)

    dropout_param_count = get_parameter_count(dropout)
    dropout_time, dropout_info = time_it(train)(dropout, train_loader, device)
    dropout_accuracy = test(dropout, test_loader, device)

    # saving data
    file_name = "batch_norm_dropout.txt"
    append_to_file(file_name, datetime.now())
    append_to_file(
        file_name, f"batch norm param count: {batch_norm_param_count}"
    )
    append_to_file(file_name, f"batch norm time: {batch_norm_time}")
    append_to_file(file_name, f"batch norm info: {batch_norm_info}")
    append_to_file(file_name, f"batch norm accuracy: {batch_norm_accuracy}")

    append_to_file(file_name, f"dropout param count: {dropout_param_count}")
    append_to_file(file_name, f"dropout time: {dropout_time}")
    append_to_file(file_name, f"dropout info: {dropout_info}")
    append_to_file(file_name, f"dropout accuracy: {dropout_accuracy}")
```

6.3 model.py

```
"""
```

```python
Defines helper functions for building, training, and testing models with architectures specific to this lab.
"""

import torch.nn as nn
from torch import no_grad
from torch.optim import Adam
from torch.utils.data import DataLoader


def build_model(
    classes: int,
    shape: tuple[int],
    convoluted: list[tuple[int]],
    linear: list[int],
    batch_norm: bool = False,
    dropout: float = 0.0,
) -> nn.Sequential:
    """
    Builds a model with convolutional first if any, then linear layers if any. Adds a batch normalization or dropout layer in between each if specified
    """

    if batch_norm and dropout:
        raise ValueError(
            "batch_norm and dropout cannot both be True for this lab"
        )

    layers = []

    # first layer
    if convoluted:
        layers.append(nn.Conv2d(shape[0], convoluted[0][0], convoluted[0][1]))
        if batch_norm:
            layers.append(nn.BatchNorm2d(convoluted[0][0]))
        layers.append(nn.ReLU())
        if dropout:
            layers.append(nn.Dropout(dropout))

    # convolutional layers
    for i in range(1, len(convoluted)):
        layers.append(
            nn.Conv2d(convoluted[i - 1][0], convoluted[i][0], convoluted[i][1])
        )
        if batch_norm:
            layers.append(nn.BatchNorm2d(convoluted[i][0]))
        layers.append(nn.ReLU())
```

```python
        if dropout:
            layers.append(nn.Dropout(dropout))

    layers.append(nn.Flatten())

    # linear layers
    for i in range(len(linear) - 1):
        layers.append(nn.Linear(linear[i], linear[i + 1]))
        if batch_norm:
            layers.append(nn.BatchNorm1d(linear[i + 1]))
        layers.append(nn.ReLU())
        if dropout:
            layers.append(nn.Dropout(dropout))

    # last layer
    last_layer = -2
    if batch_norm or dropout:
        last_layer -= 1

    layers.append(nn.Linear(layers[last_layer].out_features, classes))

    return nn.Sequential(*layers)


def get_parameter_count(model: nn.Sequential) -> int:
    """
    Custom function that returns the number of parameters in a model
     like torchvision's summary function
    """
    return sum(p.numel() for p in model.parameters())


def train(
    model: nn.Sequential,
    data_loader: DataLoader,
    device: str,
    max_epochs: int = None,
    lr: float = 1e-3,
    min_delta: float = 1e-2,
) -> dict:
    """
    Trains the model on the data loader following the given parameters.
    """

    model.to(device)
    model.train()

    optimizer = Adam(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()

```

```
    # info to return about the training
    # could include the loss/accuracy per epoch for example
    epochs = 0

    best_loss = float("inf")
    while max_epochs is None or epochs < max_epochs:
        epoch_loss = 0
        # training on each batch
        for x, y in data_loader:
            x, y = x.to(device), y.to(device)

            optimizer.zero_grad()
            outputs = model(x)
            loss = loss_fn(outputs, y)
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()

        epochs += 1
        avg_loss = epoch_loss / len(data_loader)

        # stop if the loss has not improved enough
        if abs(best_loss - avg_loss) < min_delta:
            break

        best_loss = min(best_loss, avg_loss)

    return {"epochs": epochs}


@no_grad()
def test(model: nn.Sequential, data_loader: DataLoader, device: str) -> float:
    """
    Tests the accuracy of the model on the data loader.
    """

    model.to(device)
    model.eval()

    correct = 0
    for x, y in data_loader:
        x, y = x.to(device), y.to(device)
        predicted = model(x)
        correct += (predicted.argmax(dim=1) == y).sum().item()

    return correct / len(data_loader.dataset)
```

6.4 data.py

```python
"""
Manages the data for the experiments.
"""

from torch import Tensor
from torch.utils.data import DataLoader, Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor


class FMNISTDataset(Dataset):
    def __init__(self, root: str, train: bool = True, transform=None) -> None:
        self.dataset = datasets.FashionMNIST(
            root=root, train=train, download=True, transform=transform
        )

    def __getitem__(self, index) -> tuple[Tensor, Tensor]:
        image, label = self.dataset[index]
        return image, label

    def __len__(self) -> int:
        return len(self.dataset)


def get_data_loaders(
    root: str = "./data/FMNIST", batch_size: int = 32
) -> tuple[DataLoader]:
    # Using ToTensor instead of reshaping as we did in class to better leverage the GPU
    test_dataset = FMNISTDataset(root=root, train=False, transform=ToTensor())
    train_dataset = FMNISTDataset(root=root, train=True, transform=ToTensor())

    train_loader = DataLoader(
        train_dataset, batch_size=batch_size, shuffle=True
    )
    test_loader = DataLoader(
        test_dataset, batch_size=batch_size, shuffle=False
    )

    return train_loader, test_loader


def append_to_file(filename: str, data: any) -> None:
    with open(filename, "a") as file:
        file.write(str(data) + "\n")
```