Aditya Rajesh

Topics in Electrical & Computer Engineering: Introduction to Deep Learning

Professor Yuqian Zhang

05/11/2025

Understanding and Implementing AlexNet for Image Classification

Comparison between CIFAR-10 with ImageNet

The original AlexNet was designed for the ImageNet dataset, which contains over 1.2 million high-resolution images across 1,000 complex classes. These images were resized to 256×256 pixels and then cropped to 224×224 for training. In contrast, CIFAR-10 contains only 60,000 low-resolution (32×32) images across 10 simple classes such as airplanes, cats, and ships. Because CIFAR-10 images are much smaller and simpler, directly applying the original AlexNet would cause some issues. For instance, large convolution kernels like 11×11 and aggressive strides would shrink the small input images way too quickly and the network's huge fully connected layers would overfit pretty bad to the small dataset.

In order to adapt AlexNet for CIFAR-10, a few changes are necessary to make. The first convolutional layer's kernel size must be reduced from 11×11 to smaller filters like 5×5 or 3×3, with a stride of 1 instead of 4. Pooling layers should be adjusted for smaller feature maps, and the input size to the fully connected layers must match the smaller output from the convolutional layers (e.g., 256×4×4 instead of thousands of features). While dropout remains useful for regularization, the network overall needs fewer parameters to fit CIFAR-10 appropriately without overfitting. The changes suggested would help preserve the overall architecture of AlexNet while scaling it to fit the size and complexity of CIFAR-10 images.

Results

Results of Testing all Optimizers with All Combinations

Run	Optimizer	LR	Batch Size	Dropout	Batch Norm	Final Accuracy
1	SGD	0.01	32	TRUE	FALSE	85.36%
2	SGD	0.01	32	TRUE	TRUE	87.61%

3	SGD	0.01	32	FALSE	FALSE	85.16%
4	SGD	0.01	32	FALSE	TRUE	87.50%
5	SGD	0.01	64	TRUE	FALSE	85.55%
6	SGD	0.01	64	TRUE	TRUE	87.94%
7	SGD	0.01	64	FALSE	FALSE	87.34%
8	SGD	0.01	64	FALSE	TRUE	87.95%
9	SGD	0.01	128	TRUE	FALSE	84.92%
10	SGD	0.01	128	TRUE	TRUE	88.19%
11	SGD	0.01	128	FALSE	FALSE	87.26%
12	SGD	0.01	128	FALSE	TRUE	88.11%
13	SGD	0.001	32	TRUE	FALSE	86.66%
14	SGD	0.001	32	TRUE	TRUE	87.79%
15	SGD	0.001	32	FALSE	FALSE	86.95%
16	SGD	0.001	32	FALSE	TRUE	88.43%
17	SGD	0.001	64	TRUE	FALSE	85.58%
18	SGD	0.001	64	TRUE	TRUE	87.46%
19	SGD	0.001	64	FALSE	FALSE	83.32%
20	SGD	0.001	64	FALSE	TRUE	87.47%
21	SGD	0.001	128	TRUE	FALSE	79.40%
22	SGD	0.001	128	TRUE	TRUE	85.67%
23	SGD	0.001	128	FALSE	FALSE	79.99%
24	SGD	0.001	128	FALSE	TRUE	86.44%
25	SGD	0.0001	32	TRUE	FALSE	24.54%
26	SGD	0.0001	32	TRUE	TRUE	86.11%
27	SGD	0.0001	32	FALSE	FALSE	24.83%
28	SGD	0.0001	32	FALSE	TRUE	84.80%
29	SGD	0.0001	64	TRUE	FALSE	25.13%
30	SGD	0.0001	64	TRUE	TRUE	84.73%
31	SGD	0.0001	64	FALSE	FALSE	23.44%
32	SGD	0.0001	64	FALSE	TRUE	82.91%
33	SGD	0.0001	128	TRUE	FALSE	10.00%
34	SGD	0.0001	128	TRUE	TRUE	80.48%

35	SGD	0.0001	128	FALSE	FALSE	10.00%
36	SGD	0.0001	128	FALSE	TRUE	83.54%
37	ADAM	0.01	32	TRUE	FALSE	10.00%
38	ADAM	0.01	32	TRUE	TRUE	19.51%
39	ADAM	0.01	32	FALSE	FALSE	10.00%
40	ADAM	0.01	32	FALSE	TRUE	55.85%
41	ADAM	0.01	64	TRUE	FALSE	10.01%
42	ADAM	0.01	64	TRUE	TRUE	32.06%
43	ADAM	0.01	64	FALSE	FALSE	10.00%
44	ADAM	0.01	64	FALSE	TRUE	22.50%
45	ADAM	0.01	128	TRUE	FALSE	10.00%
46	ADAM	0.01	128	TRUE	TRUE	10.03%
47	ADAM	0.01	128	FALSE	FALSE	10.00%
48	ADAM	0.01	128	FALSE	TRUE	23.90%
49	ADAM	0.001	32	TRUE	FALSE	73.37%
50	ADAM	0.001	32	TRUE	TRUE	83.49%
51	ADAM	0.001	32	FALSE	FALSE	76.56%
52	ADAM	0.001	32	FALSE	TRUE	83.47%
53	ADAM	0.001	64	TRUE	FALSE	75.94%
54	ADAM	0.001	64	TRUE	TRUE	83.85%
55	ADAM	0.001	64	FALSE	FALSE	80.38%
56	ADAM	0.001	64	FALSE	TRUE	85.36%
57	ADAM	0.001	128	TRUE	FALSE	80.18%
58	ADAM	0.001	128	TRUE	TRUE	86.26%
59	ADAM	0.001	128	FALSE	FALSE	80.43%
60	ADAM	0.001	128	FALSE	TRUE	86.95%
61	ADAM	0.0001	32	TRUE	FALSE	85.16%
62	ADAM	0.0001	32	TRUE	TRUE	88.14%
63	ADAM	0.0001	32	FALSE	FALSE	85.55%
64	ADAM	0.0001	32	FALSE	TRUE	86.99%
65	ADAM	0.0001	64	TRUE	FALSE	85.95%
66	ADAM	0.0001	64	TRUE	TRUE	87.76%

67	ADAM	0.0001	64	FALSE	FALSE	85.64%
68	ADAM	0.0001	64	FALSE	TRUE	87.32%
69	ADAM	0.0001	128	TRUE	FALSE	85.29%
70	ADAM	0.0001	128	TRUE	TRUE	87.60%
71	ADAM	0.0001	128	FALSE	FALSE	86.06%
72	ADAM	0.0001	128	FALSE	TRUE	87.16%
73	RMSPROP	0.01	32	TRUE	FALSE	13.36%
74	RMSPROP	0.01	32	TRUE	TRUE	11.58%
75	RMSPROP	0.01	32	FALSE	FALSE	13.28%
76	RMSPROP	0.01	32	FALSE	TRUE	13.72%
77	RMSPROP	0.01	64	TRUE	FALSE	11.04%
78	RMSPROP	0.01	64	TRUE	TRUE	10.75%
79	RMSPROP	0.01	64	FALSE	FALSE	13.54%
80	RMSPROP	0.01	64	FALSE	TRUE	15.62%
81	RMSPROP	0.01	128	TRUE	FALSE	10.00%
82	RMSPROP	0.01	128	TRUE	TRUE	15.51%
83	RMSPROP	0.01	128	FALSE	FALSE	10.03%
84	RMSPROP	0.01	128	FALSE	TRUE	17.89%
85	RMSPROP	0.001	32	TRUE	FALSE	75.37%
86	RMSPROP	0.001	32	TRUE	TRUE	84.63%
87	RMSPROP	0.001	32	FALSE	FALSE	76.81%
88	RMSPROP	0.001	32	FALSE	TRUE	84.55%
89	RMSPROP	0.001	64	TRUE	FALSE	75.88%
90	RMSPROP	0.001	64	TRUE	TRUE	84.55%
91	RMSPROP	0.001	64	FALSE	FALSE	74.45%
92	RMSPROP	0.001	64	FALSE	TRUE	84.67%
93	RMSPROP	0.001	128	TRUE	FALSE	80.30%
94	RMSPROP	0.001	128	TRUE	TRUE	83.99%
95	RMSPROP	0.001	128	FALSE	FALSE	81.06%
96	RMSPROP	0.001	128	FALSE	TRUE	86.06%
97	RMSPROP	0.0001	32	TRUE	FALSE	85.16%
98	RMSPROP	0.0001	32	TRUE	TRUE	86.95%

99	RMSPROP	0.0001	32	FALSE	FALSE	85.84%
100	RMSPROP	0.0001	32	FALSE	TRUE	84.42%
101	RMSPROP	0.0001	64	TRUE	FALSE	84.26%
102	RMSPROP	0.0001	64	TRUE	TRUE	86.34%
103	RMSPROP	0.0001	64	FALSE	FALSE	84.64%
104	RMSPROP	0.0001	64	FALSE	TRUE	86.78%
105	RMSPROP	0.0001	128	TRUE	FALSE	82.64%
106	RMSPROP	0.0001	128	TRUE	TRUE	87.50%
107	RMSPROP	0.0001	128	FALSE	FALSE	83.72%
108	RMSPROP	0.0001	128	FALSE	TRUE	85.92%

Discussion of Results

After running 36 configurations for SGD, Adam, and RMSprop, I noticed that SGD consistently produced the highest accuracy overall and proved to be the most stable across all learning rates and hyperparameter combinations. On average, SGD achieved a validation accuracy of 74.68%, with the best configuration reaching 88.43%. These results were not only high in absolute terms but also pretty consistent across runs, demonstrating SGD's robustness and reliability when paired with appropriate tuning.

In contrast to SGD, Adam and RMSprop showed more volatile behavior, especially at higher learning rates. At a learning rate of 0.01, both Adam and RMSprop frequently failed to converge, often plateauing at 10% validation accuracy, which is basically equivalent to random guessing for a 10-class classification task like CIFAR-10. This suggests that these optimizers were highly sensitive to learning rate and could not effectively escape poor local minima or saddle points under certain conditions.

Despite this instability, Adam and RMSprop did also show promising results when used with lower learning rates. Adam peaked at 88.14% validation accuracy and averaged 62.19% across all runs, while RMSprop reached a maximum of 87.50% and averaged 59.80%. These results indicate that both optimizers are capable of matching SGD's performance in the best-case scenario, but require more careful hyperparameter tuning and tend to be less forgiving of suboptimal settings.

Across all three optimizers, I found that the most successful configurations typically involved enabling batch normalization and disabling dropout. Batch normalization helped stabilize training, especially for deeper networks and smaller batch sizes. In contrast, dropout, while useful in some settings, often lowered performance when batch norm was already in use. This suggests that for this architecture and dataset, batch normalization was sufficient to prevent overfitting, and using both may have been redundant.

Overall, I conclude that SGD is the most stable and reliable optimizer for this task. It not only provided the highest mean and maximum accuracies but also showed resilience across a wide range of hyperparameter settings. While adaptive methods like Adam and RMSprop can reach similar peaks, they introduce greater risk of convergence failure if not properly tuned.

Source Code

Usage:

To run the training python script, use the following command:

```
    python3 train.py [--optimizer OPT] [--lr LR] [--batch_size
BATCH] [--dropout true|false] [--batchnorm true|false]
```

Arguments/Flags:

--optimizer: Optimizer to use (sgd, adam, or rmsprop). Default: sgd
--lr: (Optional) Specify learning rate (e.g., --lr 0.001)
--batch_size: (Optional) Specify batch size (e.g., --batch_size 64)
--dropout: (Optional) Whether to use dropout (true or false)
--batchnorm: (Optional) Whether to use batch normalization (true or false)

Examples:

Run all 36 configurations for SGD:

```
python3 train.py --optimizer sgd
```

Run all Adam configurations with learning rate 0.001:

```
python3 train.py --optimizer adam --lr 0.001
```

Run a single test using RMSprop, learning rate 0.0001, batch size 64, with dropout and batchnorm:

```
python3 train.py --optimizer rmsprop --lr 0.0001 --
batch_size 64 --dropout true --batchnorm true
```

Output:

Each run saves results to a CSV file named results_<optimizer>.csv and stores the best model from each run in the checkpoint/ directory.
Invalid arguments (e.g., `--dropout ture`) will trigger a helpful usage message and prevent execution.

model.py:

```
# model.py
import torch.nn as nn

class AlexNetCIFAR10(nn.Module):
    def __init__(self, num_classes=10, use_dropout=True,
use_batchnorm=False):
        super(AlexNetCIFAR10, self).__init__()

        def conv_block(in_channels, out_channels, kernel_size,
stride, padding):
            layers = [nn.Conv2d(in_channels, out_channels,
kernel_size, stride, padding),
                      nn.ReLU(inplace=True)]
            if use_batchnorm:
                layers.append(nn.BatchNorm2d(out_channels))
            return layers

        self.features = nn.Sequential(
```

```
            *conv_block(3, 64, kernel_size=5, stride=1,
padding=2),
            nn.MaxPool2d(kernel_size=2, stride=2),
            *conv_block(64, 192, kernel_size=5, stride=1,
padding=2),
            nn.MaxPool2d(kernel_size=2, stride=2),
            *conv_block(192, 384, kernel_size=3, stride=1,
padding=1),
            *conv_block(384, 256, kernel_size=3, stride=1,
padding=1),
            *conv_block(256, 256, kernel_size=3, stride=1,
padding=1),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.classifier = nn.Sequential(
            nn.Dropout() if use_dropout else nn.Identity(),
            nn.Linear(256 * 4 * 4, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout() if use_dropout else nn.Identity(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 256 * 4 * 4)
        x = self.classifier(x)
        return x
```

train.py:

```
# train.py
import torch
import torch.nn as nn
import torch.optim as optim
```

```
import torchvision
import torchvision.transforms as transforms
import os
import platform
import csv
import argparse
from model import AlexNetCIFAR10

def get_device():
    if torch.backends.mps.is_available() and platform.machine()
== "arm64":
        print("Using MPS backend on ARM64 Mac")
        return torch.device("mps")
    elif torch.cuda.is_available():
        print("Using CUDA backend")
        return torch.device("cuda")
    else:
        print("Using CPU backend")
        return torch.device("cpu")

def train_model(device, learning_rate, batch_size, use_dropout,
use_batchnorm, run_id, optimizer_type):
    num_epochs = 100
    patience = 5

    # Data Preparation
    transform_train = transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(15),
        transforms.ColorJitter(brightness=0.2, contrast=0.2,
saturation=0.2, hue=0.1),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ])
    transform_test = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ])

    trainset = torchvision.datasets.CIFAR10(root='./data',
train=True, download=True, transform=transform_train)
    trainloader = torch.utils.data.DataLoader(trainset,
batch_size=batch_size, shuffle=True, num_workers=0)
```

```
    testset = torchvision.datasets.CIFAR10(root='./data',
train=False, download=True, transform=transform_test)
    testloader = torch.utils.data.DataLoader(testset,
batch_size=batch_size, shuffle=False, num_workers=0)

    # Model
    model = AlexNetCIFAR10(use_dropout=use_dropout,
use_batchnorm=use_batchnorm).to(device)
    model = model.float()

    # Loss
    criterion = nn.CrossEntropyLoss()

    # Optimizer
    if optimizer_type.lower() == 'sgd':
        optimizer = optim.SGD(model.parameters(),
lr=learning_rate, momentum=0.9, weight_decay=1e-4)
    elif optimizer_type.lower() == 'adam':
        optimizer = optim.Adam(model.parameters(),
lr=learning_rate, weight_decay=1e-4)
    elif optimizer_type.lower() == 'rmsprop':
        optimizer = optim.RMSprop(model.parameters(),
lr=learning_rate, weight_decay=1e-4)
    else:
        raise ValueError(f"Unsupported optimizer:
{optimizer_type}")

    # Training Loop
    best_acc = 0.0
    patience_counter = 0

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        total = 0
        correct = 0

        for inputs, labels in trainloader:
            inputs, labels = inputs.to(device),
labels.to(device)
            inputs = inputs.float()

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
```

```
            optimizer.step()

            running_loss += loss.item()
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

        train_acc = 100. * correct / total
        print(f'[Run {run_id}] Epoch [{epoch+1}/{num_epochs}]
Loss: {running_loss/len(trainloader):.4f} | Train Acc:
{train_acc:.2f}%')

        # Evaluation
        model.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for inputs, labels in testloader:
                inputs, labels = inputs.to(device),
labels.to(device)
                inputs = inputs.float()
                outputs = model(inputs)
                _, predicted = outputs.max(1)
                total += labels.size(0)
                correct += predicted.eq(labels).sum().item()

        acc = 100. * correct / total
        print(f'[Run {run_id}] Validation Accuracy: {acc:.2f}%')

        if acc > best_acc:
            best_acc = acc
            patience_counter = 0
            if not os.path.isdir('checkpoint'):
                os.makedirs('checkpoint')
            model_path = f'./checkpoint/{optimizer_type}
_run_{run_id}_best.pth'
            torch.save(model.state_dict(), model_path)
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print(f'[Run {run_id}] Early stopping
triggered.')
                break

    print(f'[Run {run_id}] Best Validation Accuracy:
{best_acc:.2f}%')
```

```
    return best_acc

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--optimizer', type=str, default='sgd',
choices=['sgd', 'adam', 'rmsprop'],
                        help='Optimizer to use: sgd | adam |
rmsprop')
    parser.add_argument('--lr', type=float, help='Specific
learning rate (optional)')
    parser.add_argument('--batch_size', type=int, help='Specific
batch size (optional)')
    parser.add_argument('--dropout', type=str, choices=['true',
'false'], help='Use dropout (true/false)')
    parser.add_argument('--batchnorm', type=str,
choices=['true', 'false'], help='Use batchnorm (true/false)')

    try:
        args = parser.parse_args()
    except SystemExit:
        print("\n[ERROR] Invalid arguments passed. Check for
typos in flags or values.")
        parser.print_help()
        exit(1)

    optimizer_type = args.optimizer.lower()
    device = get_device()

    learning_rates = [args.lr] if args.lr is not None else
[0.01, 0.001, 0.0001]
    batch_sizes = [args.batch_size] if args.batch_size is not
None else [32, 64, 128]
    dropout_options = [args.dropout == 'true'] if args.dropout
is not None else [True, False]
    batchnorm_options = [args.batchnorm == 'true'] if
args.batchnorm is not None else [False, True]

    run_id = 1
    results = []

    # Prepare CSV file
    csv_filename = f'results_{optimizer_type}.csv'
    with open(csv_filename, mode='w', newline='') as f:
        writer = csv.writer(f)
        writer.writerow(['Run', 'Optimizer', 'LR', 'Batch Size',
'Dropout', 'Batch Norm', 'Final Accuracy'])
```

```
    for lr in learning_rates:
        for batch_size in batch_sizes:
            for dropout in dropout_options:
                for batchnorm in batchnorm_options:
                    print(f'\n=== Starting Run {run_id} with
{optimizer_type.upper()} ===')
                    print(f'LR: {lr}, Batch Size: {batch_size},
Dropout: {dropout}, BatchNorm: {batchnorm}\n')

                    best_val_acc = train_model(device, lr,
batch_size, dropout, batchnorm, run_id, optimizer_type)

                    with open(csv_filename, mode='a',
newline='') as f:
                        writer = csv.writer(f)
                        writer.writerow([run_id,
optimizer_type.upper(), lr, batch_size, dropout, batchnorm,
f'{best_val_acc:.2f}%'])

                    results.append({
                        'Run': run_id,
                        'Learning Rate': lr,
                        'Batch Size': batch_size,
                        'Dropout': dropout,
                        'BatchNorm': batchnorm,
                        'Best Validation Accuracy': best_val_acc
                    })

                    run_id += 1

    print("\n=== All Runs Complete ===\n")
    for res in results:
        print(res)



if __name__ == '__main__':
    main()
```

merge_results.py:

```
import pandas as pd

# Load each CSV
sgd = pd.read_csv("results_sgd.csv")
adam = pd.read_csv("results_adam.csv")
rmsprop = pd.read_csv("results_rmsprop.csv")

# Add optimizer column if not present
if "Optimizer" not in sgd.columns:
    sgd["Optimizer"] = "SGD"
if "Optimizer" not in adam.columns:
    adam["Optimizer"] = "Adam"
if "Optimizer" not in rmsprop.columns:
    rmsprop["Optimizer"] = "RMSprop"

# Reorder to keep consistent format
columns_order = ['Run', 'Optimizer', 'LR', 'Batch Size',
'Dropout', 'Batch Norm', 'Final Accuracy']
sgd = sgd[columns_order]
adam = adam[columns_order]
rmsprop = rmsprop[columns_order]

# Adjust run numbers to stay unique
adam["Run"] += sgd["Run"].max()
rmsprop["Run"] += adam["Run"].max()

# Merge all three
merged = pd.concat([sgd, adam, rmsprop], ignore_index=True)

# Save the merged CSV
merged.to_csv("merged_results.csv", index=False)
print("Merged CSV saved as 'merged_results.csv'")
```