Deep Learning 435: Final Report

Tomer Butbul
Rutgers University
May 2025

***Abstract*—This project evaluates the performance of a modified AlexNet architecture on the CIFAR-10 image classification dataset. Through a targeted subset of 48 experiments, we explored the effects of learning rate, batch size, dropout, batch normalization, and optimizer choice on accuracy and training efficiency. The results offer practical insights into deep learning configuration strategies under computational constraints.**

I. Introduction

The objective of this project was to explore how architectural modifications and hyperparameter tuning affect the performance of convolutional neural networks, using AlexNet as the base model. By leveraging PyTorch, we implemented a series of controlled experiments on the CIFAR-10 dataset. This dataset consists of 60,000 32x32 color images categorized into 10 classes, with 50,000 images used for training and 10,000 for testing. Compared to larger datasets like ImageNet, CIFAR-10 poses a unique challenge due to its low resolution and limited per-class examples, necessitating adjustments to the standard AlexNet architecture.

Our implementation features a modular AlexNet variant that allows toggling batch normalization and dropout. A training pipeline was developed to evaluate different optimizers and hyperparameters, and a script was written to run 48 predefined experiments automatically. These experiments incorporate early stopping and log results such as training duration, validation accuracy, and convergence behavior, offering a robust framework for deep learning experimentation.

Before diving into our custom model design, it is important to understand the origin and core contributions of the original AlexNet architecture. Introduced by Krizhevsky et al. in 2012, AlexNet was a breakthrough in deep learning, demonstrating the power of large-scale convolutional neural networks on the ImageNet dataset, which contains over a million high-resolution labeled images across 1,000 categories. The architecture includes five convolutional layers and three fully connected layers, uses ReLU activation for faster convergence, employs dropout to reduce overfitting, and takes advantage of GPU parallelism for efficient training. AlexNet achieved top-1 and top-5 error rates of 37.5% and 17.0%, respectively, significantly outperforming prior approaches. Its innovations in architecture, normalization, and regularization formed the foundation for modern deep learning models [1].

II. Mathematical Foundations

The backbone of convolutional neural networks like AlexNet lies in several fundamental operations. These operations not only enable deep feature extraction but also influence how efficiently and accurately the model converges during training.

A. Convolutional Layer

The convolution operation extracts spatial features from images by applying a learnable filter over the input. Mathematically, for input I and filter K, the convolution output O at position (i, j) is:

$$O(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) \cdot K(m,n)$$

This allows the network to detect edges, textures, and shapes. Deeper layers learn increasingly abstract patterns. In our experiments, convolutional depth played a critical role in generalization capacity.

B. ReLU Activation

After convolution, AlexNet uses the Rectified Linear Unit:

$$f(x) = \max(0, x)$$

ReLU introduces non-linearity while avoiding the vanishing gradient problem common in saturating activations like sigmoid or tanh. Its simplicity accelerates convergence and improved training dynamics, especially evident in the performance gains observed in Figure 1 and Figure 2.

C. Dropout Regularization

Dropout randomly deactivates neurons during training:

$$\tilde{h}_i = h_i \cdot z_i, \quad z_i \sim \text{Bernoulli}(p)$$

where z_i is a binary mask and p is the keep probability. This prevents co-adaptation of neurons and acts as an implicit ensemble method. Our results show that dropout can increase overfitting risk when combined improperly (e.g., with SGD), as seen in overfitting clusters in Figure 3.

D. Batch Normalization

Batch normalization stabilizes training by normalizing activations:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \quad y^{(k)} = \gamma \hat{x}^{(k)} + \beta$$

where μ_B and σ_B^2 are batch mean and variance, and γ, β are learnable scale and shift parameters. BN reduces internal covariate shift, enabling higher learning rates and faster convergence. This directly explains the more stable and higher test accuracy of BN-based configurations.

E. Loss Function and Backpropagation

The model minimizes cross-entropy loss:

$$\mathcal{L} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

where y_i is the true label (one-hot) and $\hat{y}_i$ is the softmax output. Gradients of this loss are propagated backward using chain rule:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Efficient backpropagation enables deep networks like AlexNet to learn from large-scale data. We observed faster and smoother convergence with Adam, attributed to its adaptive learning rate computed as:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{w}_t = \frac{m_t}{\sqrt{v_t} + \epsilon}$$

where g_t is the gradient and m_t, v_t are first and second moment estimates.

III. Model Design

The core model architecture includes five convolutional layers followed by three fully connected layers, using ReLU activations and max pooling. The final layer outputs 10 logits, one for each class in CIFAR-10. Optional components such as batch normalization and dropout (at a rate of 0.5) were selectively integrated to study their individual impact. All trainable weights were initialized using Kaiming Normal initialization, which is well-suited for ReLU-based networks.

Three optimizers were examined: SGD with momentum, RMSprop, and Adam. To make the experiments both meaningful and computationally feasible, we defined a constrained parameter space. For instance, dropout and batch normalization were not used together; instead, configurations tested one or the other. Adam and RMSprop were evaluated with learning rates of 0.001 and 0.0001, while SGD was tested with higher learning rates (0.01 and 0.001) due to its sensitivity to hyperparameters. The batch sizes were fixed at 32 and 64 to maintain training stability and time efficiency. Our hypothesis was that Adam would provide consistent convergence, batch normalization would improve accuracy, and dropout would serve as a useful regularizer under limited training data.

IV. Experiment Setup

The complete hyperparameter space, if fully expanded, would have included 324 combinations. Given the compute time and diminishing returns from certain configurations, we narrowed the scope to 48 well-chosen experiments. The constraints used to trim the space included excluding combinations that applied both dropout and batch normalization simultaneously, limiting learning rate ranges based on optimizer behavior, and retaining only the most informative training/test splits of 70/30 and 90/10.

This streamlined approach maintained the representational diversity necessary to uncover trends while reducing experiment runtime from over 45 hours to a manageable subset. For each configuration, the model was trained from scratch and evaluated independently using consistent scripts and metrics. The collected metrics were stored in a CSV file for post-analysis.

V. Results

Overall, several consistent trends emerged. Batch normalization outperformed dropout across all optimizers in terms of final test accuracy, suggesting it offers better training stability and regularization in the CIFAR-10 context. Adam was the most reliable optimizer, yielding the highest average accuracy and the smoothest convergence curves. In contrast, SGD's performance varied significantly, and one configuration involving SGD with dropout failed catastrophically, resulting in near-random accuracy. Overfitting was evident in multiple dropout experiments, where training accuracy approached 95% but test accuracy stagnated below 70%, indicating poor generalization.

Optimizer sensitivity to learning rate also differed. Adam and RMSprop performed best at a learning rate of 0.001, while SGD required a higher learning rate of 0.01 to reach competitive results. Training time correlated with batch size, with smaller batches (32) consistently taking longer to converge. RMSprop was generally the slowest among the three optimizers. Additionally, early stopping mechanisms triggered later when 90% of the data was used for training, which aligns with expectations that larger training sets require more epochs to overfit or converge.

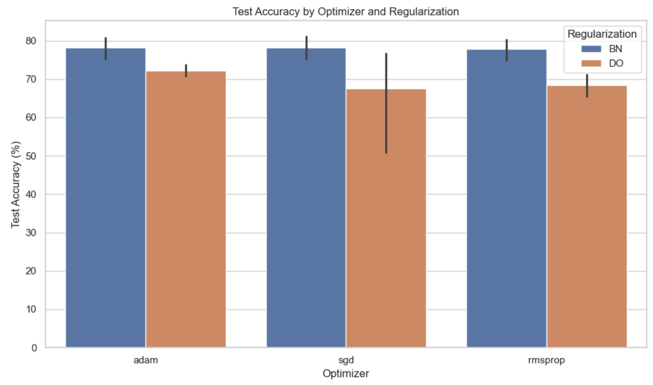


Fig. 1: Test Accuracy by Optimizer and Regularization (BN vs. DO)

Figure 1 illustrates the test accuracy distribution across different optimizers and regularization techniques. Models using batch normalization consistently outperformed those using dropout across all three optimizers. Adam combined with batch normalization achieved the highest accuracy values, reinforcing the stabilizing impact of BN and the adaptability of Adam. Dropout configurations showed higher variance and several low-performing outliers, underscoring its sensitivity and potential for overfitting on smaller datasets.

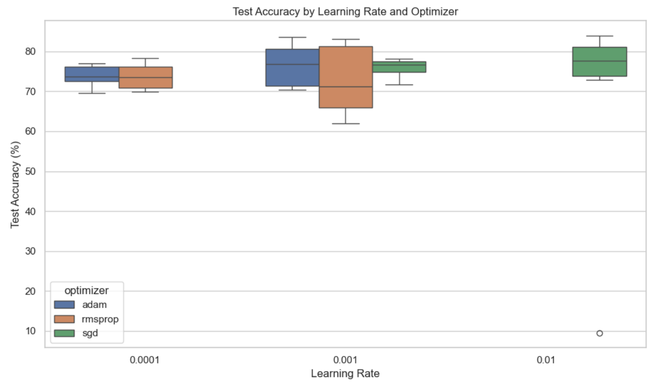


Fig. 2: Test Accuracy Across Learning Rates for Each Optimizer

Figure 2 presents the effect of learning rate on model accuracy, segmented by optimizer. Adam and RMSprop both demonstrated optimal performance at a learning rate of 0.001, while degrading slightly at 0.0001. SGD required a higher learning rate of 0.01 to remain competitive and suffered significant drops at 0.001. This validates the necessity of careful learning rate selection, especially when using more sensitive optimizers like SGD.

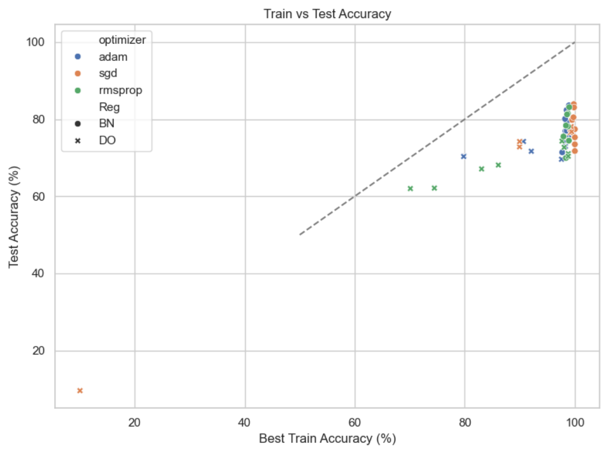


Fig. 3: Scatter Plot of Train vs. Test Accuracy, Showing Overfitting Trends

Figure 3 shows a scatter plot of training versus test accuracy. Points in the upper-left quadrant—high training but low test accuracy—represent overfitted models. This pattern was predominantly observed in dropout-heavy configurations, particularly those using SGD. In contrast, batch normalization configurations were clustered more tightly along the diagonal, indicating better generalization.

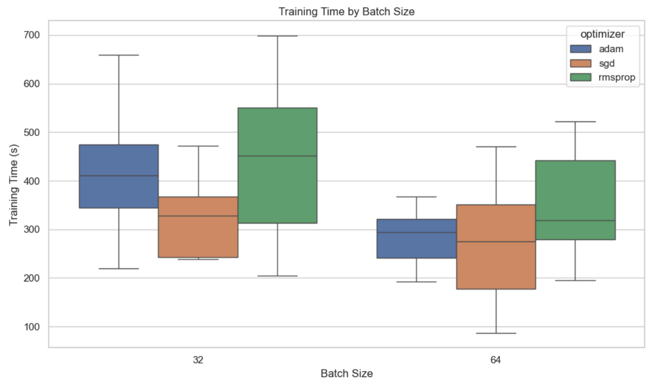


Fig. 4: Training Time Distribution by Batch Size and Optimizer

Figure 4 displays training time as a function of batch size and optimizer. Models trained with batch size 64 were consistently faster, demonstrating the computational benefits of larger batches. Among optimizers, Adam was the fastest on average, followed by SGD and then RMSprop, which took the longest—likely due to its more complex weight updates.

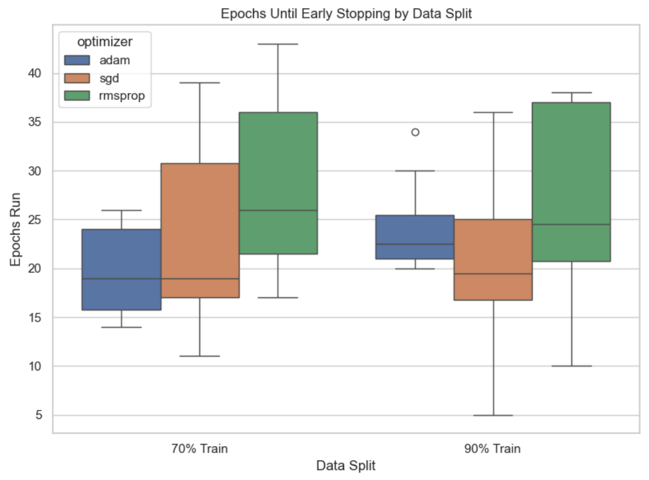


Fig. 5: Number of Epochs Until Early Stopping for Each Train/Test Split

Figure 5 depicts the number of epochs until early stopping for each train/test split. As expected, the 90/10 split required more epochs on average due to the larger training set. However, this also led to marginal improvements in test accuracy for well-regularized models, indicating a trade-off between training time and generalization performance.

VI. Conclusion

This project demonstrates how CNN performance on a constrained dataset like CIFAR-10 is highly sensitive to architectural tweaks, regularization choices, and training parameters. The experiments showed that batch normalization is generally more effective than dropout, providing both accuracy gains and training stability. Adam emerged as the most robust optimizer across all scenarios, and larger batch sizes offered efficiency benefits without compromising model performance.

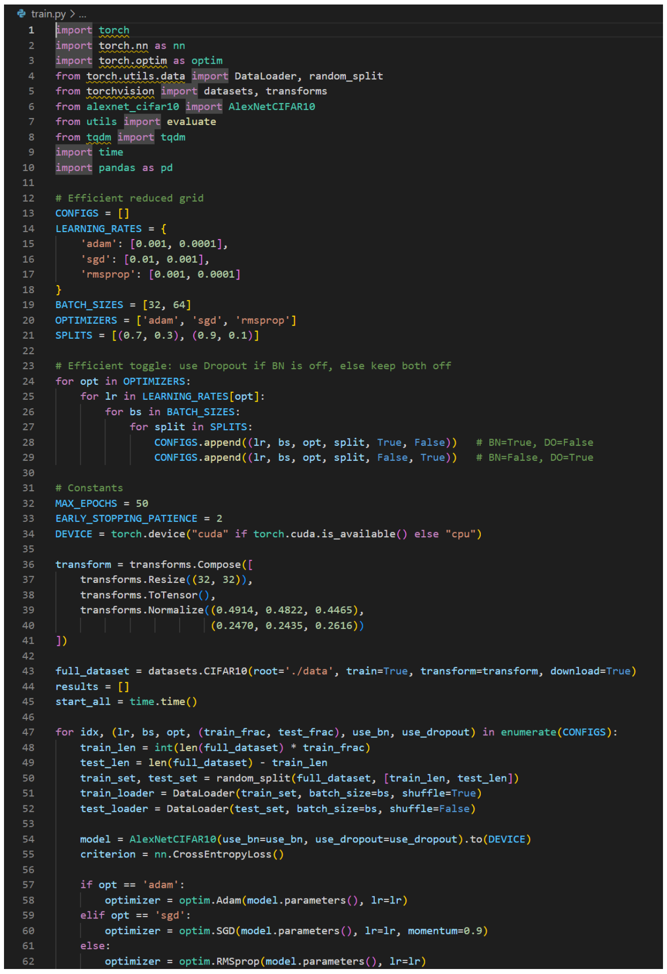
Dropout should be used cautiously, especially when batch normalization is an available option. Learning rate tuning remains a critical component of model performance, particularly for SGD. Early stopping proved effective for managing overfitting and should be integrated with validation-based metrics for improved reliability.

Future work could extend this analysis by integrating cross-validation and seed averaging to improve statistical significance. Testing modern architectures like ResNet or MobileNet would help compare baseline improvements, and advanced data augmentation techniques could further boost generalization. Overall, this project provided a controlled yet insightful exploration into CNN training dynamics using AlexNet on CIFAR-10.

References

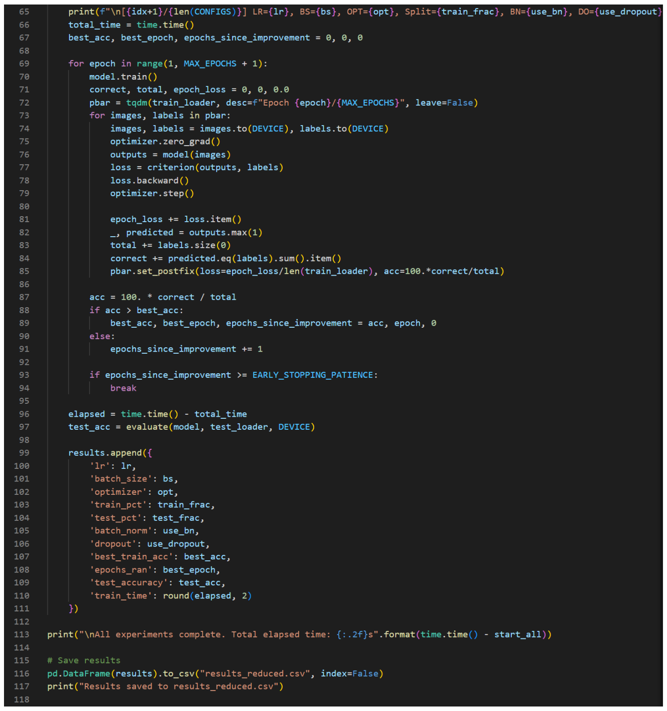
[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, vol. 25, 2012, pp. 1097–1105.

VII. Code

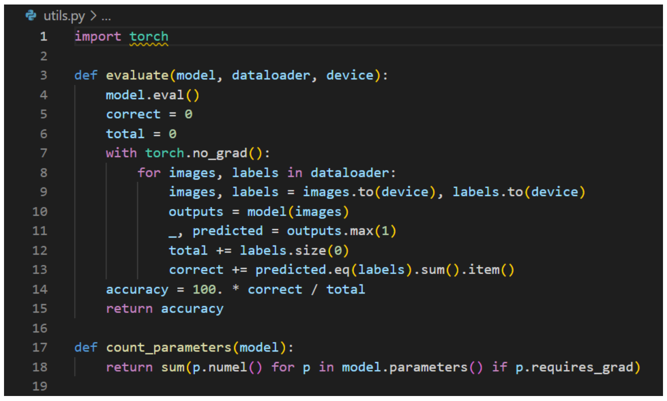


train.py > ...

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, random_split
from torchvision import datasets, transforms
from alexnet_cifar10 import AlexNetCIFAR10
from utils import evaluate
from tqdm import tqdm
import time
import pandas as pd

# Efficient reduced grid
CONFIGS = []
LEARNING_RATES = {
    'adam': [0.001, 0.0001],
    'sgd': [0.01, 0.001],
    'rmsprop': [0.001, 0.0001]
}
BATCH_SIZES = [32, 64]
OPTIMIZERS = ['adam', 'sgd', 'rmsprop']
SPLITS = [(0.7, 0.3), (0.9, 0.1)]

# Efficient toggle: use Dropout if BN is off, else keep both off
for opt in OPTIMIZERS:
    for lr in LEARNING_RATES[opt]:
        for bs in BATCH_SIZES:
            for split in SPLITS:
                CONFIGS.append((lr, bs, opt, split, True, False))   # BN=True, DO=False
                CONFIGS.append((lr, bs, opt, split, False, True))   # BN=False, DO=True

# Constants
MAX_EPOCHS = 50
EARLY_STOPPING_PATIENCE = 2
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),
                         (0.2470, 0.2435, 0.2616))
])

full_dataset = datasets.CIFAR10(root='./data', train=True, transform=transform, download=True)
results = []
start_all = time.time()

for idx, (lr, bs, opt, (train_frac, test_frac), use_bn, use_dropout) in enumerate(CONFIGS):
    train_len = int(len(full_dataset) * train_frac)
    test_len = len(full_dataset) - train_len
    train_set, test_set = random_split(full_dataset, [train_len, test_len])
    train_loader = DataLoader(train_set, batch_size=bs, shuffle=True)
    test_loader = DataLoader(test_set, batch_size=bs, shuffle=False)

    model = AlexNetCIFAR10(use_bn=use_bn, use_dropout=use_dropout).to(DEVICE)
    criterion = nn.CrossEntropyLoss()

    if opt == 'adam':
        optimizer = optim.Adam(model.parameters(), lr=lr)
    elif opt == 'sgd':
        optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9)
    else:
        optimizer = optim.RMSprop(model.parameters(), lr=lr)
```

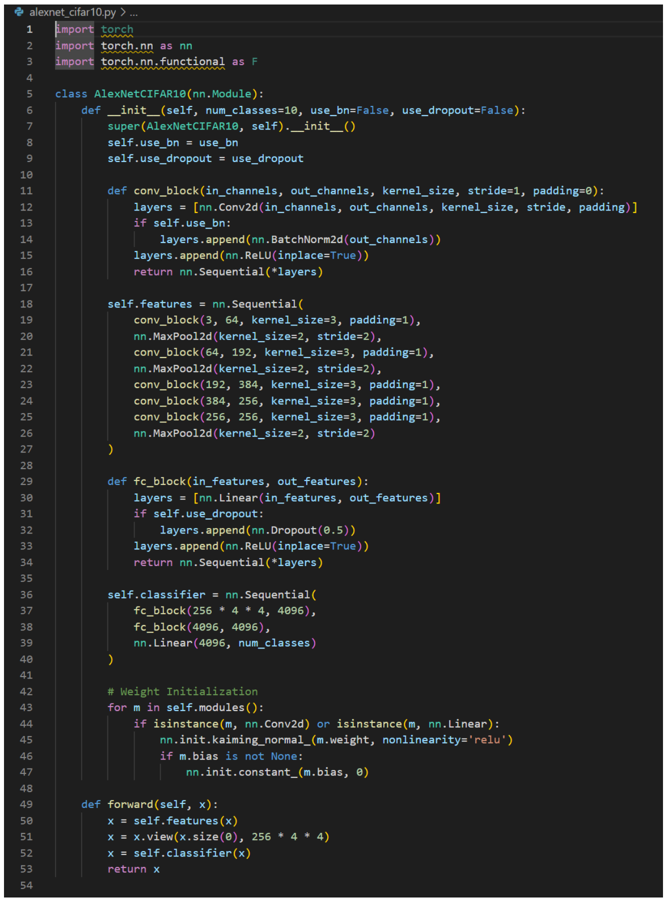


```python
    print(f"\n[{idx+1}/{len(CONFIGS)}] LR={lr}, BS={bs}, OPT={opt}, Split={train_frac}, BN={use_bn}, DO={use_dropout}
    total_time = time.time()
    best_acc, best_epoch, epochs_since_improvement = 0, 0, 0

    for epoch in range(1, MAX_EPOCHS + 1):
        model.train()
        correct, total, epoch_loss = 0, 0, 0.0
        pbar = tqdm(train_loader, desc=f"Epoch {epoch}/{MAX_EPOCHS}", leave=False)
        for images, labels in pbar:
            images, labels = images.to(DEVICE), labels.to(DEVICE)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()
            pbar.set_postfix(loss=epoch_loss/len(train_loader), acc=100.*correct/total)

        acc = 100. * correct / total
        if acc > best_acc:
            best_acc, best_epoch, epochs_since_improvement = acc, epoch, 0
        else:
            epochs_since_improvement += 1

        if epochs_since_improvement >= EARLY_STOPPING_PATIENCE:
            break

    elapsed = time.time() - total_time
    test_acc = evaluate(model, test_loader, DEVICE)

    results.append({
        'lr': lr,
        'batch_size': bs,
        'optimizer': opt,
        'train_pct': train_frac,
        'test_pct': test_frac,
        'batch_norm': use_bn,
        'dropout': use_dropout,
        'best_train_acc': best_acc,
        'epochs_ran': best_epoch,
        'test_accuracy': test_acc,
        'train_time': round(elapsed, 2)
    })

print("\nAll experiments complete. Total elapsed time: {:.2f}s".format(time.time() - start_all))

# Save results
pd.DataFrame(results).to_csv("results_reduced.csv", index=False)
print("Results saved to results_reduced.csv")
```



utils.py > ...

```python
import torch

def evaluate(model, dataloader, device):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()
    accuracy = 100. * correct / total
    return accuracy

def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```



alexnet_cifar10.py > ...

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

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

        def conv_block(in_channels, out_channels, kernel_size, stride=1, padding=0):
            layers = [nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)]
            if self.use_bn:
                layers.append(nn.BatchNorm2d(out_channels))
            layers.append(nn.ReLU(inplace=True))
            return nn.Sequential(*layers)

        self.features = nn.Sequential(
            conv_block(3, 64, kernel_size=3, padding=1),
            nn.MaxPool2d(kernel_size=2, stride=2),
            conv_block(64, 192, kernel_size=3, padding=1),
            nn.MaxPool2d(kernel_size=2, stride=2),
            conv_block(192, 384, kernel_size=3, padding=1),
            conv_block(384, 256, kernel_size=3, padding=1),
            conv_block(256, 256, kernel_size=3, padding=1),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        def fc_block(in_features, out_features):
            layers = [nn.Linear(in_features, out_features)]
            if self.use_dropout:
                layers.append(nn.Dropout(0.5))
            layers.append(nn.ReLU(inplace=True))
            return nn.Sequential(*layers)

        self.classifier = nn.Sequential(
            fc_block(256 * 4 * 4, 4096),
            fc_block(4096, 4096),
            nn.Linear(4096, num_classes)
        )

        # Weight Initialization
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
                nn.init.kaiming_normal_(m.weight, nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)

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

```