NNDL Lab 2 Report

Sharvita Paithankar

6.1 Deliverables

This report shows the impact of regularization techniques on the performance of a neural network trained on the KMNIST dataset(Kuzushiji-MNIST) which is a collection of grayscale images representing handwritten Japanese characters. This dataset has the following:

Total dataset size: 70,000 images (28×28 pixels)
Training set size: 60,000 images
Test set size: 10,000 images
Number of classes: 10 (each class represents a different Japanese character)
Pixel intensity range: 0 to 255 (grayscale)

We compared two models for this, the baseline model and the dropout model. The baseline model is a standard feedforward neural network that has no dropout. The dropout model has a similar architecture but has dropout layers after each fully connected layer. This report shows whether these regularization techniques reduce overfitting and improve test accuracy.

After the data was loaded, preprocessing steps were performed. First normalization was done where each image is converted from a 0-255 range to a 0-1 range using the "torchvision.transforms.ToTensor()" function. This helped to stabilize the training by making sure there are consistent input scales. I did not add any additional transformations so there is a consistent baseline. Then the dataset is split into training (60,000 images) and test (10,000 images) subsets.(This was done by using the "train(bool, optional)") parameter mentioned in the documentation in pytorch. Each image is flattened into a 1D tensor of size 784 (28×28 pixels) before feeding into the network.

The experiment involves two distinct architectures:

1. Baseline Model

The BaselineModel is a fully connected feedforward neural network with three layers:

Input Layer: Accepts 784-dimensional input (flattened 28×28 image).
Hidden Layer 1: 512 neurons, ReLU activation.
Hidden Layer 2: 256 neurons, ReLU activation.
Output Layer: 10 neurons (one per class), Softmax activation for classification.
No dropout layers.
No batch normalization.
Uses L2 weight decay (1e-4) to regularize weights.

2. Dropout Model

The DropoutModel follows a similar structure as the BaselineModel but has dropout layers to avoid overfitting.

Key Differences in DropoutModel:

Dropout (0.5) applied after each hidden layer (50% of neurons randomly deactivated during training).
No L2 weight decay applied.
All other hyperparameters remain unchanged.
Dropout forces the network to learn more robust representations by preventing over-reliance on specific neurons.

Training Setup
Each model is trained using the Stochastic Gradient Descent (SGD) optimizer with the following hyperparameters:

Hyperparameter	Value
Learning Rate	0.1
Momentum	0.9
Batch Size	32
Number of Epochs	20
Optimizer	SGD
Weight Decay (L2 Regularization)	1e-4 (only for BaselineModel)
Weight Decay (L2 Regularization)	1e-4 (only for BaselineModel)
Dropout Rate 0.5	(only for DropoutModel)
Optimizer	SGD

For training the model,first, the KMNIST dataset is loaded from torchvision.datasets.KMNIST. Images are normalized and split into training and test sets. Data is loaded into mini-batches of size 32 for efficient computation.The model is initialized and their architecture is coded (mentioned above). To train the model, I use 20 epochs and for each batch, I use forward pass to compute prediction, I use cross entropy loss to computer loss. We calculate gradient using backpropagation using backward pass and update the weights using SGD optimizer. For each epoch, the train loss, train accuracy, validation loss and validation accuracy are also calculated.

Expected Results

BaselineModel which has no dropout will overfit. It will probably have high training accuracy but lower test accuracy due to overfitting.
DropoutModel will generalize better: By reducing reliance on specific neurons, dropout will reduce overfitting and improve validation accuracy.
L2 weight decay will slightly improve generalization: By penalizing large weights, it will prevent overfitting in the BaselineModel.
Training loss will decrease for both models: But validation loss will increase more significantly for BaselineModel due to overfitting.

6.2 Results

Baseline Model with 25% data

Classification Report:

	precision	recall	f1-score	support
0	0.94	0.93	0.94	1000
1	0.94	0.92	0.93	1000
2	0.89	0.87	0.88	1000
3	0.92	0.96	0.94	1000
4	0.91	0.92	0.91	1000
5	0.94	0.92	0.93	1000
6	0.90	0.96	0.93	1000
7	0.96	0.94	0.95	1000
8	0.93	0.92	0.92	1000
9	0.94	0.92	0.93	1000
accuracy			0.93	10000
macro avg	0.93	0.93	0.93	10000
weighted avg	0.93	0.93	0.93	10000

Baseline Model with 50% data

Classification Report:

	precision	recall	f1-score	support
0	0.94	0.96	0.95	1000

	precision	recall	f1-score	support
1	0.96	0.94	0.95	1000
2	0.93	0.91	0.92	1000
3	0.95	0.97	0.96	1000
4	0.93	0.92	0.93	1000
5	0.98	0.94	0.96	1000
6	0.91	0.97	0.94	1000
7	0.97	0.95	0.96	1000
8	0.95	0.96	0.95	1000
9	0.95	0.95	0.95	1000
accuracy			0.95	10000
macro avg	0.95	0.95	0.95	10000
weighted avg	0.95	0.95	0.95	10000

Baseline Model with 75% data

Classification Report:

	precision	recall	f1-score	support
0	0.96	0.95	0.96	1000
1	0.98	0.94	0.96	1000
2	0.94	0.90	0.92	1000
3	0.95	0.98	0.96	1000
4	0.95	0.94	0.94	1000
5	0.99	0.95	0.97	1000
6	0.94	0.97	0.95	1000
7	0.97	0.97	0.97	1000
8	0.95	0.98	0.97	1000
9	0.95	0.97	0.96	1000
accuracy			0.96	10000
macro avg	0.96	0.96	0.96	10000
weighted avg	0.96	0.96	0.96	10000

Baseline Model with 100% data

Classification Report:

	precision	recall	f1-score	support
0	0.96	0.96	0.96	1000
1	0.98	0.94	0.96	1000
2	0.95	0.94	0.94	1000
3	0.96	0.98	0.97	1000
4	0.96	0.94	0.95	1000
5	0.97	0.96	0.97	1000

	precision	recall	f1-score	support
6	0.94	0.98	0.96	1000
7	0.98	0.98	0.98	1000
8	0.97	0.98	0.97	1000
9	0.97	0.97	0.97	1000
accuracy			0.96	10000
macro avg	0.96	0.96	0.96	10000
weighted avg	0.96	0.96	0.96	10000

Dropout Model 25% data

Classification Report:

	precision	recall	f1-score	support
0	0.97	0.94	0.95	1000
1	0.96	0.92	0.94	1000
2	0.91	0.88	0.90	1000
3	0.93	0.97	0.95	1000
4	0.91	0.93	0.92	1000
5	0.96	0.93	0.94	1000
6	0.92	0.95	0.93	1000
7	0.96	0.96	0.96	1000
8	0.93	0.96	0.95	1000
9	0.93	0.94	0.93	1000
accuracy			0.94	10000
macro avg	0.94	0.94	0.94	10000
weighted avg	0.94	0.94	0.94	10000

Dropout Model 50% data

Classification Report:

	precision	recall	f1-score	support
0	0.98	0.96	0.97	1000
1	0.99	0.92	0.95	1000
2	0.94	0.91	0.92	1000
3	0.95	0.97	0.96	1000
4	0.93	0.95	0.94	1000
5	0.99	0.93	0.96	1000
6	0.91	0.98	0.95	1000
7	0.97	0.92	0.94	1000
8	0.97	0.97	0.97	1000
9	0.87	0.98	0.92	1000
accuracy			0.95	10000

macro avg	0.95	0.95	0.95	10000
weighted avg	0.95	0.95	0.95	10000

Dropout Model 75% data

Classification Report:

	precision	recall	f1-score	support
0	0.97	0.96	0.96	1000
1	0.98	0.94	0.96	1000
2	0.93	0.93	0.93	1000
3	0.97	0.98	0.98	1000
4	0.94	0.95	0.94	1000
5	0.97	0.96	0.97	1000
6	0.95	0.97	0.96	1000
7	0.99	0.96	0.98	1000
8	0.95	0.98	0.97	1000
9	0.97	0.98	0.98	1000
accuracy			0.96	10000
macro avg	0.96	0.96	0.96	10000
weighted avg	0.96	0.96	0.96	10000

Dropout Model 100% data

Classification Report:

	precision	recall	f1-score	support
0	0.96	0.98	0.97	1000
1	0.98	0.95	0.97	1000
2	0.93	0.94	0.94	1000
3	0.97	0.98	0.98	1000
4	0.97	0.94	0.95	1000
5	0.98	0.96	0.97	1000
6	0.97	0.98	0.98	1000
7	0.98	0.98	0.98	1000
8	0.98	0.98	0.98	1000
9	0.97	0.97	0.97	1000
accuracy			0.97	10000
macro avg	0.97	0.97	0.97	10000
weighted avg	0.97	0.97	0.97	10000

L2 Regularization 25% data

Classification Report:

	precision	recall	f1-score	support
0	0.94	0.94	0.94	1000
1	0.95	0.92	0.93	1000
2	0.90	0.87	0.89	1000
3	0.92	0.96	0.94	1000
4	0.90	0.91	0.91	1000
5	0.94	0.92	0.93	1000
6	0.89	0.95	0.92	1000
7	0.98	0.95	0.96	1000
8	0.92	0.93	0.92	1000
9	0.94	0.93	0.93	1000
accuracy			0.93	10000
macro avg	0.93	0.93	0.93	10000
weighted avg	0.93	0.93	0.93	10000

L2 Regularization 50% data

Classification Report:

	precision	recall	f1-score	support
0	0.95	0.96	0.95	1000
1	0.96	0.93	0.95	1000
2	0.93	0.89	0.91	1000
3	0.93	0.97	0.95	1000
4	0.93	0.93	0.93	1000
5	0.97	0.94	0.96	1000
6	0.92	0.96	0.94	1000
7	0.97	0.96	0.96	1000
8	0.95	0.96	0.96	1000
9	0.96	0.96	0.96	1000
accuracy			0.95	10000
macro avg	0.95	0.95	0.95	10000
weighted avg	0.95	0.95	0.95	10000

L2 Regularization 75% data

Classification Report:

	precision	recall	f1-score	support
0	0.96	0.96	0.96	1000
1	0.97	0.95	0.96	1000
2	0.94	0.91	0.92	1000
3	0.94	0.98	0.96	1000
4	0.94	0.94	0.94	1000

	precision	recall	f1-score	support
5	0.98	0.94	0.96	1000
6	0.93	0.98	0.95	1000
7	0.97	0.98	0.98	1000
8	0.95	0.97	0.96	1000
9	0.97	0.96	0.97	1000
accuracy			0.96	10000
macro avg	0.96	0.96	0.96	10000
weighted avg	0.96	0.96	0.96	10000

L2 Regularization 100% data

Classification Report:

	precision	recall	f1-score	support
0	0.96	0.96	0.96	1000
1	0.98	0.95	0.97	1000
2	0.95	0.93	0.94	1000
3	0.96	0.98	0.97	1000
4	0.95	0.95	0.95	1000
5	0.98	0.96	0.97	1000
6	0.94	0.98	0.96	1000
7	0.98	0.98	0.98	1000
8	0.96	0.98	0.97	1000
9	0.97	0.97	0.97	1000
accuracy			0.96	10000
macro avg	0.96	0.96	0.96	10000
weighted avg	0.96	0.96	0.96	10000

Data Augmentation 25% data

Classification Report:

	precision	recall	f1-score	support
0	0.98	0.93	0.95	1000
1	0.99	0.92	0.95	1000
2	0.93	0.93	0.93	1000
3	0.94	0.97	0.96	1000
4	0.92	0.93	0.93	1000
5	0.97	0.95	0.96	1000
6	0.94	0.98	0.96	1000
7	0.98	0.96	0.97	1000
8	0.94	0.97	0.95	1000
9	0.95	0.97	0.96	1000

	precision	recall	f1-score	support
accuracy			0.95	10000
macro avg	0.95	0.95	0.95	10000
weighted avg	0.95	0.95	0.95	10000

Data Augmentation 50% data

Classification Report:

	precision	recall	f1-score	support
0	0.97	0.98	0.98	1000
1	0.99	0.95	0.97	1000
2	0.96	0.95	0.95	1000
3	0.97	0.99	0.98	1000
4	0.98	0.94	0.96	1000
5	0.97	0.98	0.98	1000
6	0.96	0.97	0.96	1000
7	0.99	0.98	0.99	1000
8	0.96	0.99	0.98	1000
9	0.97	0.98	0.98	1000
accuracy			0.97	10000
macro avg	0.97	0.97	0.97	10000
weighted avg	0.97	0.97	0.97	10000

Data Augmentation 75% data

Classification Report:

	precision	recall	f1-score	support
0	0.96	0.98	0.97	1000
1	0.99	0.96	0.97	1000
2	0.98	0.94	0.96	1000
3	0.97	0.99	0.98	1000
4	0.98	0.94	0.96	1000
5	0.98	0.98	0.98	1000
6	0.96	0.99	0.97	1000
7	0.98	0.99	0.99	1000
8	0.96	0.99	0.97	1000
9	0.99	0.98	0.98	1000
accuracy			0.97	10000
macro avg	0.97	0.97	0.97	10000
weighted avg	0.97	0.97	0.97	10000

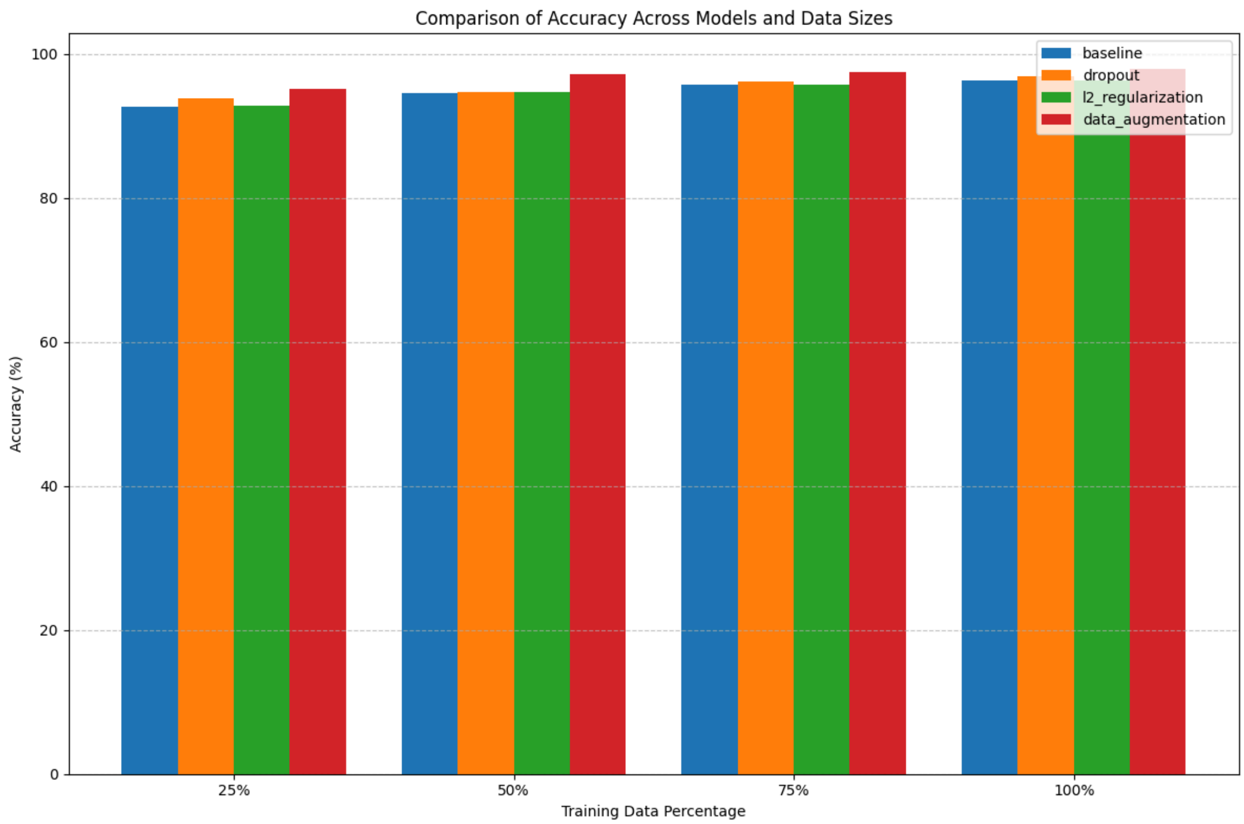
Data Augmentation 100% data

Classification Report:

	precision	recall	f1-score	support
0	0.98	0.98	0.98	1000
1	0.99	0.98	0.98	1000
2	0.97	0.95	0.96	1000
3	0.97	0.99	0.98	1000
4	0.98	0.96	0.97	1000
5	0.98	0.98	0.98	1000
6	0.98	0.98	0.98	1000
7	0.99	0.99	0.99	1000
8	0.98	0.99	0.99	1000
9	0.98	0.99	0.98	1000
accuracy			0.98	10000
macro avg	0.98	0.98	0.98	10000
weighted avg	0.98	0.98	0.98	10000

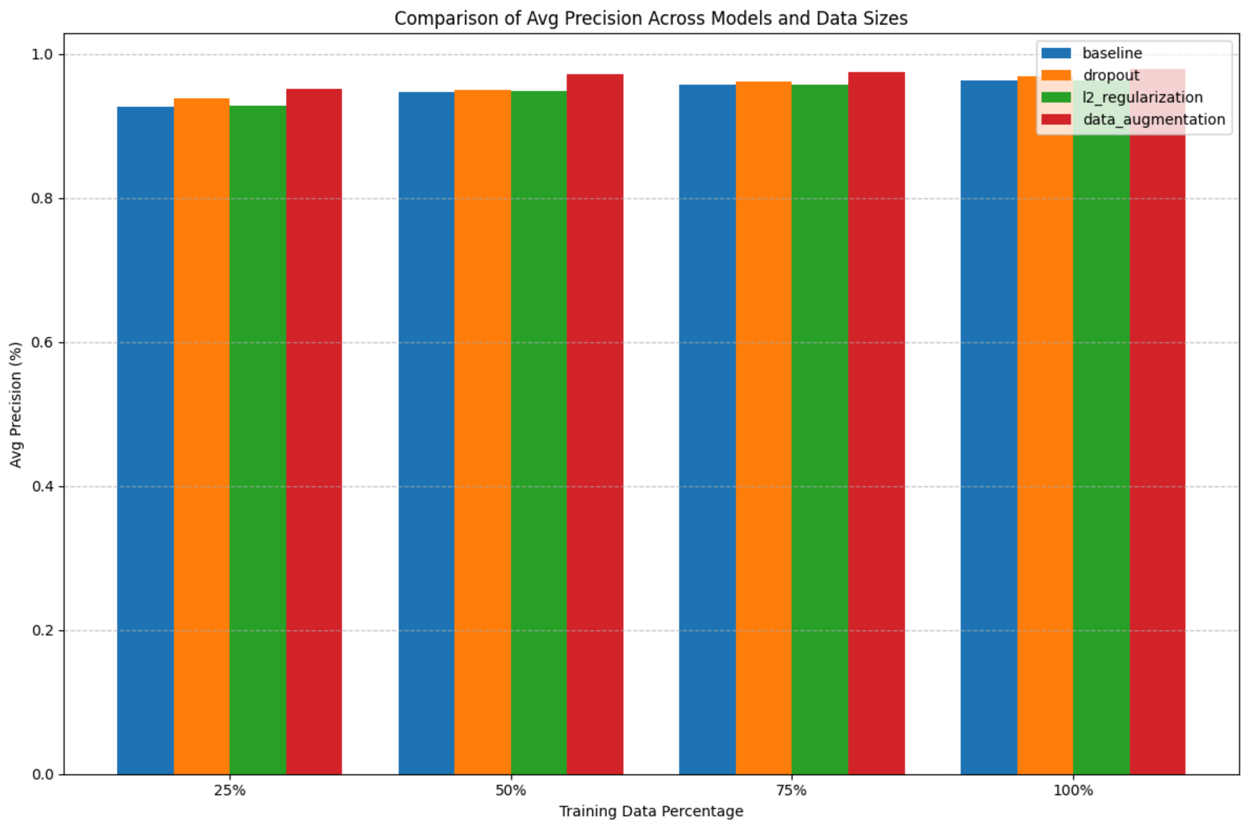
Accuracy Table:

Model	**25%**	**50%**	**75%**	**100%**
baseline	**92.65%**	**94.61%**	**95.67%**	**96.32%**
dropout	**93.76%**	**94.77%**	**96.18%**	**96.92%**
l2_regularization	**92.79%**	**94.76%**	**95.74%**	**96.27%**
data_augmentation	**95.14%**	**97.18%**	**97.45%**	**97.93%**



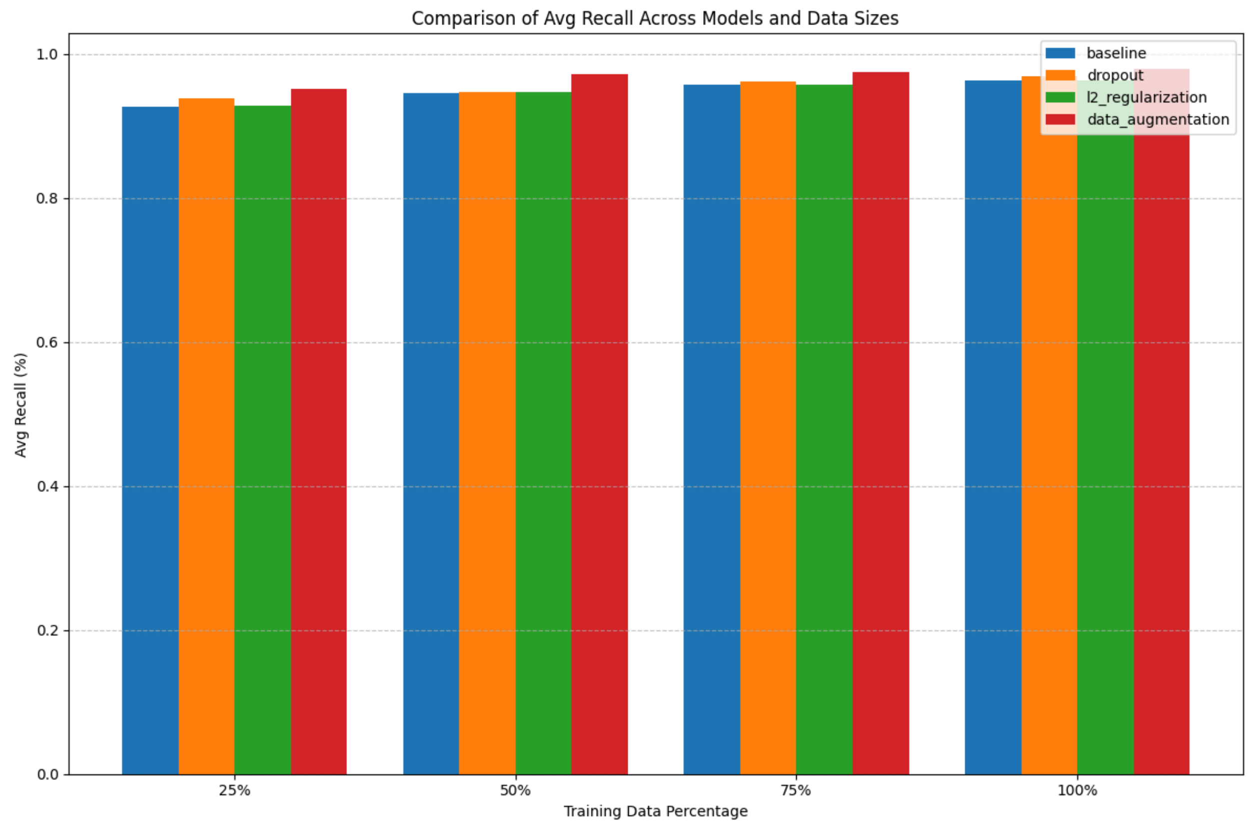
Average Precision Table:

Model	25%	50%	75%	100%
baseline	0.9268	0.9465	0.9569	0.9634
dropout	0.9379	0.9496	0.9620	0.9693
l2_regularization	0.9283	0.9479	0.9577	0.9629
data_augmentation	0.9520	0.9720	0.9748	0.9793



Average Recall Table:

Model	25%	50%	75%	100%
baseline	0.9265	0.9461	0.9567	0.9632
dropout	0.9376	0.9477	0.9618	0.9692
l2_regularization	0.9279	0.9476	0.9574	0.9627
data_augmentation	0.9514	0.9718	0.9745	0.9793



6.3 Analysis

Our experiment tested three techniques—dropout, L2 weight decay, and data augmentation—on different amounts of training data (25%, 50%, 75%, and 100%).
The results show that data augmentation has the highest accuracy efficiency. L2 and Dropout weight decay also helped improve performance, even though dropout had a bigger impact.
It helps to prove that this method helps the model generalize better and avoid overfitting, because data augmentation has the highest accuracy at every dataset size. Data augmentation supports variety to the training data by changing the images. This helped the model to learn patterns rather than just memorizing specific images. Hence, the model becomes better at recognizing new images it has never seen before.
Data augmentation doesn't take anything away unlike dropout which basically removes some neurons during training or L2 weight decay which limits how much neurons can change. Instead, it adds useful variations, which helps the model learn better without any restrictions. This technique is very useful when training data is small because it makes the most out of the available samples. But when a large amount of data is available, the benefits decrease because the model already has a lot of variety to learn from.
Both dropout and L2 weight decay prevent overfitting, but dropout is more effective in most cases because dropout helps neurons work together by randomly turning some of the neurons off while training. This helps the model to distribute learning among all of the neurons, making it

stronger. L2 weight decay focuses on how much neurons contribute, preventing extreme overfitting, however it doesn't improve feature diversity as dropout does.
When the dataset size is large (75-100%), dropout is better because it forces the network to keep learning in a balanced way. When dataset size is small (25-50%), L2 weight decay is helpful because it stops the model from relying too much on specific features. When the dataset size is large (75-100%), dropout is better because it forces the network to keep learning in a balanced way.

There are a couple of trade offs. Training more data always improves accuracy, but the improvement slows down as data size increases. Dropout is better than L2 weight decay when more data is available, since it directly prevents overfitting without limiting weight updates.
Data augmentation is the best method for small datasets because it helps the model learn from fewer examples more efficiently.

Code

```
from torchvision.datasets import KMNIST
import torchvision.transforms as transforms
import torchvision.datasets as datasets
from torch.utils.data import DataLoader, random_split
transform = transforms.Compose([

   transforms.ToTensor(),

])
train_dataset = KMNIST(root="./data", train = True,
transform=transform, download=True)
test_dataset = KMNIST(root="./data", train = False,
transform=transform, download=True)
classes = train_dataset.classes
len(train_dataset), len(test_dataset)
train_loader = DataLoader(train_dataset, batch_size=32,
shuffle=True, num_workers=2)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False,
num_workers=2)
import matplotlib.pyplot as plt

for images, labels in train_loader:
   for i in range(32):
     print(classes[labels[i]])
     plt.imshow(images[i].permute(1, 2, 0), cmap='gray')
     plt.show()
   break
from torch import nn

class BaselineModel(nn.Module):
   def __init__(self, num_classes):
       super(BaselineModel, self).__init__()

       # Layer 1: Convolutional layer + ReLU (non-linear activation
1)
       #image size is 28 x 28
       self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1)
       self.relu1 = nn.ReLU()
```

```
        # Layer 2: Convolutional layer + ReLU (non-linear activation 
2)
          #image size is 26 x 26
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 3: Convolutional layer + ReLU (non-linear activation 
3)
          #image size is 12 x 12
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 4: Fully connected layer + ReLU (non-linear 
activation 4)
          #image size is 5 x 5
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(128 * 5 * 5 , 512)
        self.relu4 = nn.ReLU()

        # Layer 5: Fully connected layer + ReLU (non-linear 
activation 5)
        self.fc2 = nn.Linear(512, 256)
        self.relu5 = nn.ReLU()

        # Output layer
        self.fc3 = nn.Linear(256, num_classes)

    def forward(self, x):

        x = (self.relu1(self.conv1(x)))
        x = self.pool2(self.relu2(self.conv2(x)))
        x = self.pool3(self.relu3(self.conv3(x)))


        x = self.flatten(x)
```

```
            x = self.relu4(self.fc1(x))
            x = self.relu5(self.fc2(x))
            x = self.fc3(x)

            return x
      model3 = BaselineModel(10)
      !pip install torchinfo

      import torchinfo
      torchinfo.summary(model=model3,
             input_size=(32, 1, 28, 28),
             col_names=["input_size", "output_size", "num_params",
"trainable"],
             col_width=20,
             row_settings=["var_names"]
      )
      next(iter(train_loader))[0].shape
      import torch
      import matplotlib.pyplot as plt

      def train_model(model, train_loader, val_loader, optimizer,
criterion, num_epochs, scheduler=None):

         device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
         print(f"Using device: {device}")

         model.to(device)
         train_loss_history = []
         val_loss_history = []
         val_acc_history = []

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

```
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()

            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Backward and optimize
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * inputs.size(0)

            _,predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_train_acc = 100 * correct / total
        epoch_train_loss = running_loss / len(train_loader.dataset)
        train_loss_history.append(epoch_train_loss)

        # Validation
        model.eval()
        running_loss = 0.0
        correct = 0
        total = 0

        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)

                # Forward pass
                outputs = model(inputs)
                loss = criterion(outputs, labels)

                running_loss += loss.item() * inputs.size(0)

                # Calculate accuracy
```

```
                    _, predicted = torch.max(outputs.data, 1)
                    total += labels.size(0)
                    correct += (predicted == labels).sum().item()

            epoch_val_loss = running_loss / len(val_loader.dataset)
            epoch_val_acc = 100 * correct / total
            val_loss_history.append(epoch_val_loss)
            val_acc_history.append(epoch_val_acc)

            print(f'Epoch {epoch}, Train Loss: {epoch_train_loss:.3f},
Test Loss: {epoch_val_loss:.3f} Train Acc: {epoch_train_acc:.3f}, Test
Acc: {epoch_val_acc:.3f}')

        return train_loss_history, val_loss_history, val_acc_history

     def plot_training_results(train_loss_history, val_loss_history,
val_acc_history):

        # Create figure with two subplots
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

        # Plot loss curves
        epochs = range(1, len(train_loss_history) + 1)
        ax1.plot(epochs, train_loss_history, 'b-', label='Training Loss')
        ax1.plot(epochs, val_loss_history, 'r-', label='Validation Loss')
        ax1.set_title('Training and Validation Loss')
        ax1.set_xlabel('Epochs')
        ax1.set_ylabel('Loss')
        ax1.legend()
        ax1.grid(True)

        # Plot accuracy curve
        ax2.plot(epochs, val_acc_history, 'g-', label='Validation
Accuracy')
        ax2.set_title('Validation Accuracy')
        ax2.set_xlabel('Epochs')
        ax2.set_ylabel('Accuracy (%)')
        ax2.legend()
        ax2.grid(True)
```

```
        plt.tight_layout()
        plt.show()

    def test_model(model, test_loader):

        device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
        model.to(device)
        model.eval()
        correct = 0
        total = 0

        with torch.no_grad():
            for inputs, labels in test_loader:
                inputs, labels = inputs.to(device), labels.to(device)

                # Forward pass
                outputs = model(inputs)

                # Calculate accuracy
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
        accuracy = 100 * correct / total
        print(f'Test Accuracy: {accuracy:.2f}%')

        return accuracy

    def save_model(model, filepath='_classifier.pt'):

        torch.save(model.state_dict(), filepath)
        print(f"Model saved to {filepath}")

    def load_model(model_class, filepath='_classifier.pt',
num_classes=37):

        model = model_class(num_classes)
        device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
        return model
```

```
import torch
import torch.nn as nn
import torch.optim as optim

# Initialize your model
num_classes = 10
model = BaselineModel(num_classes=num_classes)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

# Optional: Learning rate scheduler
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5,
gamma=0.1)

# Train model
train_loss, val_loss, val_acc = train_model(
   model=model,
   train_loader=train_loader,
   val_loader=test_loader,  # Using test_loader as validation
   optimizer=optimizer,
   criterion=criterion,
   num_epochs=20,
   # scheduler=scheduler
)

# Plot results
plot_training_results(train_loss, val_loss, val_acc)

# Evaluate on test set
final_accuracy = test_model(model, test_loader)

# Save the model
save_model(model, filepath='_classifier_baseline.pt')

# To load the model later (uncomment when needed)
# loaded_model = load_model(DropoutModel,
filepath='pet_classifier_dropout.pt')
from torch import nn
```

```
class DropoutModel(nn.Module):
    def __init__(self, num_classes, dropout_rate=0.3):
        super(DropoutModel, self).__init__()
        self.flatten = nn.Flatten()

        # Layer 1: Convolutional layer + ReLU (non-linear activation
1)
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1,
padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # self.dropout1 = nn.Dropout(dropout_rate)

        # Layer 2: Convolutional layer + ReLU (non-linear activation
2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1,
padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # self.dropout2 = nn.Dropout(dropout_rate)

        # Layer 3: Convolutional layer + ReLU (non-linear activation
3)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1,
padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        # self.dropout3 = nn.Dropout(dropout_rate)

        # Layer 4: Fully connected layer + ReLU (non-linear
activation 4)
        self.fc1 = nn.Linear(128 * 3 * 3, 512)
        self.relu4 = nn.ReLU()
        self.dropout4 = nn.Dropout(dropout_rate)

        # Layer 5: Fully connected layer + ReLU (non-linear
activation 5)
        self.fc2 = nn.Linear(512, 256)
        self.relu5 = nn.ReLU()
```

```python
        self.dropout5 = nn.Dropout(dropout_rate)

        # Output layer
        self.fc3 = nn.Linear(256, num_classes)

    def forward(self, x):
        # Apply layers
        x = self.pool1(self.relu1(self.conv1(x)))
        x = (self.pool2(self.relu2(self.conv2(x))))
        x = (self.pool3(self.relu3(self.conv3(x))))
        x = self.flatten(x)

        # Fully connected layers
        x = self.dropout4(self.relu4(self.fc1(x)))
        x = self.dropout5(self.relu5(self.fc2(x)))
        x = self.fc3(x)

        return x

import torch
import torch.nn as nn
import torch.optim as optim

# Initialize your model
num_classes = 10
model = DropoutModel(num_classes=num_classes)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001,
weight_decay=1e-4)

# Optional: Learning rate scheduler
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5,
gamma=0.1)

# Train model
train_loss, val_loss, val_acc = train_model(
    model=model,
    train_loader=train_loader,
```

```
        val_loader=test_loader,  # Using test_loader as validation
        optimizer=optimizer,
        criterion=criterion,
        num_epochs=20,
        # scheduler=scheduler
    )

    # Plot results
    plot_training_results(train_loss, val_loss, val_acc)

    # Evaluate on test set
    final_accuracy = test_model(model, test_loader)

    # Save the model
    save_model(model, filepath='_classifier_dropout.pt')

    # To load the model later (uncomment when needed)
    # loaded_model = load_model(DropoutModel,
filepath='pet_classifier_dropout.pt')
    from sklearn.metrics import precision_recall_fscore_support

    def evaluate_model(model, test_loader):
        device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
        model.eval()
        all_preds = []
        all_labels = []

        with torch.no_grad():
            for inputs, labels in test_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                _, predicted = torch.max(outputs.data, 1)

                all_preds.extend(predicted.cpu().numpy())
                all_labels.extend(labels.cpu().numpy())

        # Calculate precision and recall for each class
        precision, recall, f1, support =
precision_recall_fscore_support(all_labels, all_preds, average=None)
```

```
        avg_precision = np.mean(precision)
        avg_recall = np.mean(recall)

        print(f'Average Precision: {avg_precision:.4f}')
        print(f'Average Recall: {avg_recall:.4f}')

        return precision, recall
    import torchvision.transforms as transforms
    from torch.utils.data import DataLoader

    # 1. L2 Weight Regularization (weight decay in optimizer)
    def train_with_l2_regularization(model_class, train_loader,
test_loader, num_epochs=50, weight_decay=1e-4):

        model = model_class(num_classes=10)
        criterion = nn.CrossEntropyLoss()

        # Using SGD with weight decay for L2 regularization
        optimizer = optim.SGD(model.parameters(), lr=0.1,
weight_decay=weight_decay)

        # Train the model
        train_loss, val_loss, val_acc = train_model(
            model=model,
            train_loader=train_loader,
            val_loader=test_loader,
            optimizer=optimizer,
            criterion=criterion,
            num_epochs=num_epochs
        )

        return model, train_loss, val_loss, val_acc

    # 2. Data Augmentation
    def create_augmented_dataset():

        # Define augmentation transforms
        augmentation_transform = transforms.Compose([
            transforms.RandomRotation(10),  # Rotate by up to 10 degrees
```

```
            transforms.RandomAffine(0, translate=(0.1, 0.1)),  # Shift
image by up to 10%
            transforms.ToTensor(),  # Convert to tensor
        ])

        # Create augmented training datasets
        augmented_train_25 = KMNIST(root="./data", train=True,
transform=augmentation_transform, download=False)
        augmented_train_50 = KMNIST(root="./data", train=True,
transform=augmentation_transform, download=False)
        augmented_train_75 = KMNIST(root="./data", train=True,
transform=augmentation_transform, download=False)
        augmented_train_100 = KMNIST(root="./data", train=True,
transform=augmentation_transform, download=False)

        # Create stratified subsets
        aug_train_25 = create_stratified_subset(augmented_train_25, 0.25)
        aug_train_50 = create_stratified_subset(augmented_train_50, 0.50)
        aug_train_75 = create_stratified_subset(augmented_train_75, 0.75)
        aug_train_100 = augmented_train_100

        # Create DataLoaders
        aug_train_loader_25 = DataLoader(aug_train_25,
batch_size=batch_size, shuffle=True, num_workers=2)
        aug_train_loader_50 = DataLoader(aug_train_50,
batch_size=batch_size, shuffle=True, num_workers=2)
        aug_train_loader_75 = DataLoader(aug_train_75,
batch_size=batch_size, shuffle=True, num_workers=2)
        aug_train_loader_100 = DataLoader(aug_train_100,
batch_size=batch_size, shuffle=True, num_workers=2)

        return {
            '25%': aug_train_loader_25,
            '50%': aug_train_loader_50,
            '75%': aug_train_loader_75,
            '100%': aug_train_loader_100
        }

    # Create augmented dataloaders
    augmented_loaders = create_augmented_dataset()
```

```python
    from sklearn.metrics import precision_recall_fscore_support,
confusion_matrix, classification_report

    def evaluate_model_metrics(model, test_loader):

        device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
        model.eval()
        all_preds = []
        all_labels = []

        with torch.no_grad():
            for inputs, labels in test_loader:
                inputs, labels = inputs.to(device), labels.to(device)

                # Forward pass
                outputs = model(inputs)

                # Get predictions
                _, predicted = torch.max(outputs.data, 1)

                # Store predictions and labels for metric calculation
                all_preds.extend(predicted.cpu().numpy())
                all_labels.extend(labels.cpu().numpy())

        # Calculate metrics
        accuracy = 100 * np.mean(np.array(all_preds) ==
np.array(all_labels))

        # Calculate precision and recall for each class
        precision, recall, f1, support = precision_recall_fscore_support(
            all_labels, all_preds, average=None
        )

        avg_precision = np.mean(precision)
        avg_recall = np.mean(recall)

        metrics_dict = {
            'accuracy': accuracy,
            'avg_precision': avg_precision,
```

```
        'avg_recall': avg_recall,
        'per_class_precision': precision,
        'per_class_recall': recall
    }

    print(f'Test Accuracy: {accuracy:.2f}%')
    print(f'Average Precision: {avg_precision:.4f}')
    print(f'Average Recall: {avg_recall:.4f}')

    # Optional: Print detailed classification report
    print("\nClassification Report:")
    print(classification_report(all_labels, all_preds))

    return metrics_dict
```