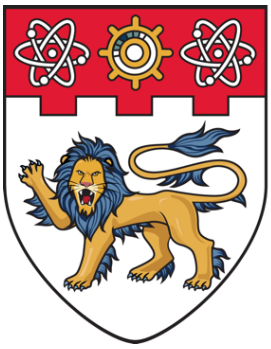


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

AY 2024/25 SEMESTER 2

SC4001 CE/CZ4042
Neural Network and Deep Learning
Assignment 2

Sim Zheng Ze, Abraham [U210855H]

Koh Wee Hock [U2120935J]

Chen Xueer [U2120993D]

CONTENTS

1 INTRODUCTION

Image classification is a fundamental task in computer vision that has seen remarkable advancements with the rise of deep learning techniques. In the fashion domain, accurate classification of clothing items enables a wide range of applications including automated inventory management, recommendation systems, and visual search features for e-commerce platforms. The Fashion MNIST dataset, introduced as a more challenging alternative to the original MNIST dataset, provides a standardized benchmark for evaluating image classification algorithms in the fashion domain.

Fashion MNIST consists of 70,000 grayscale images of clothing items across 10 classes, each image sized at 28×28 pixels. The dataset presents several challenges including intra-class variations, similar-looking classes, and the limited resolution of images. Despite its relatively small image size, the dataset captures the essential complexity of real-world fashion item recognition, making it an ideal testbed for comparing different neural network architectures.

In this project, we explore a progressive approach to improving classification performance on the Fashion MNIST dataset. We begin with standard convolutional neural network (CNN) architectures as a baseline and systematically incorporate more advanced techniques to enhance performance.

This systematic progression allows us to isolate the impact of each enhancement and understand which techniques are most effective for the Fashion MNIST classification task. Our objective is not only to achieve high classification accuracy but also to analyse the trade-offs between model complexity, computational requirements, and performance gains. Through this exploration, we aim to contribute insights into the most effective neural network design choices for fashion item classification tasks.

2 CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks (CNNs) have become the dominant approach for image classification tasks due to their ability to leverage spatially local correlations through convolutional filters. CNNs

employ weight sharing to significantly reduce parameters compared to fully connected networks, making them efficient for image processing tasks.

For the Fashion MNIST dataset, a simple CNN architecture can achieve around higher than 90% accuracy. A deeper CNN model with more convolutional layers should allow for more representational power through additional nonlinearities and a greater number of learnable parameters, potentially enabling the model to better distinguish between the similar clothing categories like shirts and t-shirts.

2.1 Advanced CNN Techniques

2.1.1 Dilated Convolutions

Dilated convolutions introduce gaps in the convolutional kernel, effectively expanding the receptive field without increasing parameters or computational cost. By inserting spaces between kernel elements, dilated convolutions allow the network to capture broader contextual information while maintaining efficiency. For Fashion MNIST, dilated convolutions can help distinguish similar clothing items by incorporating more global context in the feature representations.

2.1.2 Deformable Convolutions

Deformable convolutions extend standard convolutions by adding learnable offsets to the sampling grid. This allows the network to dynamically adjust the receptive field based on the input, enabling the model to focus on the most relevant parts of the image. The standard grid sampling locations are augmented with offsets that are learned during training, making the convolution operation adaptable to the content of the input. For fashion items with varying shapes and orientations, deformable convolutions offer the flexibility to better capture relevant features.

3 Beyond Architecture

While architectural innovations offer significant improvements to model performance, techniques that address data characteristics rather than altering the CNN structure itself can provide complementary benefits. Data augmentation strategies, learning rate scheduling, and specialized loss functions operate orthogonally to architectural changes, often yielding substantial performance gains regardless

of the underlying model. These approaches are particularly valuable when architectural optimizations reach diminishing returns or when computational resources limit the complexity of deployable models.

3.1 Regularization and Normalization Techniques

Batch normalization normalizes layer inputs to mitigate internal covariate shift and accelerate training. For Fashion MNIST, batch normalization can help address the varying contrast levels across different clothing items. Dropout provides regularization by randomly dropping units during training, preventing co-adaptation of neurons and reducing overfitting—particularly important when working with limited training data.

3.2 Data Augmentation with MixUp

MixUp creates synthetic training examples by blending pairs of images and their labels. Unlike conventional augmentations that modify individual images, MixUp generates virtual samples that lie on the linear path between existing data points. This teaches the model to transition gradually between classes, encouraging smoother decision boundaries and improved generalization. For Fashion MNIST, MixUp helps the model distinguish between similar clothing categories by learning from synthesized combinations, making it less susceptible to overfitting on training data peculiarities while developing more robust feature representations.

4 Implementation

In this section, we will go into the implementation code of each architecture. The training and validation of each architecture is done using a generic training function which considers an early stop approach.

4.1 STANDARD CNN

```
# A standard CNN model with 2 Convolutional layer
class StandardCNN(nn.Module):
    def __init__(self):
        super(StandardCNN, self).__init__()

        # Convolutional layer 1: Input channels = 1, Output channels = 32
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)

        # Convolutional layer 2: Input channels = 32, Output channels = 64
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        # Pooling Layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2) # Pooling reduces spatial dimensions by 2

        # Fully connected layers
        self.fc1 = nn.Linear(64 * 7 * 7, 128)  # Flattened size after pooling
        self.fc2 = nn.Linear(128, 10)  # Output layer for 10 classes
```

```
def forward(self, x):
    # Forward pass through the first convolutional layer and pooling
    x = self.pool(nn.functional.relu(self.conv1(x)))    # 28x28 -> 14x14

    # Forward pass through the second convolutional layer and pooling
    x = self.pool(nn.functional.relu(self.conv2(x)))    # 14x14 -> 7x7

    # Flatten the output for the fully connected layers
    x = x.view(-1, 64 * 7 * 7)

    # Forward pass through the first fully connected layer
    x = nn.functional.relu(self.fc1(x))

    # Forward pass through the output layer
    x = self.fc2(x)
```

The standard CNN approach for Fashion MNIST typically employs a 2-layer convolutional architecture as shown in the reference code. This implementation begins with a first convolutional layer that transforms the single-channel input images to 32 feature maps using 3×3 kernels, followed by ReLU activation and max pooling. A second convolutional layer further expands representation to 64 channels, again followed by activation and pooling operations that reduce spatial dimensions. After flattening, the network processes these features through a fully connected layer with 128 neurons before the final classification layer with 10 outputs corresponding to the clothing categories. This straightforward architecture creates a feature hierarchy while progressively reducing spatial dimensions and increasing channel depth, effectively balancing model capacity with computational efficiency for the Fashion MNIST classification task.

4.2 Deep CNN

```
# Third convolutional layer: Input channels = 64, Output channels = 128
self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)

# Fourth convolutional layer: Input channels = 128, Output channels = 256
self.conv4 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
```

The Deep CNN approach extends the standard architecture by incorporating four convolutional layers instead of two, creating a deeper feature hierarchy. As highlighted in the picture above, the network maintains the initial layers while adding two additional convolutional stages that progressively increase feature map depth. The third layer transforms 64 channels into 128 feature maps, while the fourth further expands representation to 256 channels.

4.3 Dilated CNN

```
# Dilation allows for an increased receptive field without increasing the kernel size
self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=dilation_val, dilation=dilation_val)
```

The Dilated CNN approach variant introduces dilated convolutions while maintaining the two-layer architecture. As shown in the implementation, the first layer remains standard while the second applies a dilation factor to expand the receptive field without increasing parameter count. We have tested with a dilation factor of 2 and 10 to observe how low and high dilation factor can affect the model.

4.4 Deformable CNN

```
# Deformable Convolutional Layer 1: Input channels = 1, Output channels = 32
self.offsets1 = nn.Conv2d(1, 18, kernel_size=3, stride=1, padding=1)  # Offsets for deformable conv1
self.deform_conv1 = DeformConv2d(1, 32, kernel_size=3, stride=1, padding=1)

# Deformable Convolutional Layer 2: Input channels = 32, Output channels = 64
self.offsets2 = nn.Conv2d(32, 18, kernel_size=3, stride=1, padding=1)  # Offsets for deformable conv2
self.deform_conv2 = DeformConv2d(32, 64, kernel_size=3, stride=1, padding=1)
```

The Deformable CNN implementation creates deformable convolutions through a two-step process in each layer. As highlighted in the code, each deformable layer first uses a standard convolution to generate 18 channels of learnable spatial offsets. These offset values are then passed to the

specialized DeformConv2d operation which shifts the sampling locations from their regular grid pattern.

```
# Deformable Convolution 1
offsets1 = self.offsets1(x)  # Learnable offsets for deformable convolution
x = self.deform_conv1(x, offsets1)  # Apply deformable convolution
x = nn.functional.relu(x)  # Activation
x = self.pool(x)  # 28x28 -> 14x14

# Deformable Convolution 2
offsets2 = self.offsets2(x)  # Learnable offsets for deformable convolution
x = self.deform_conv2(x, offsets2)  # Apply deformable convolution
x = nn.functional.relu(x)  # Activation
x = self.pool(x)  # 14x14 -> 7x7
```

During the forward pass, offset fields are computed first and then applied in the deformable convolution, allowing the network to dynamically adjust where features are sampled from based on the input content rather than using fixed geometric patterns.

4.5 Normalization and Regularization

```
# An Enhanced Deformable CNN model that integrates regularization and normalization
class EnahncedDeformableCNN(nn.Module):
    def __init__(self, dropout_val=0.3):
        super(EnahncedDeformableCNN, self).__init__()

        # Deformable Convolutional Layer 1: Input channels = 1, Output channels = 32
        self.offsets1 = nn.Conv2d(1, 18, kernel_size=3, stride=1, padding=1)  # Offsets for deformable conv1
        self.deform_conv1 = DeformConv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.batchnorm1 = nn.BatchNorm2d(32)  # BatchNorm for deformable conv1

        # Deformable Convolutional Layer 2: Input channels = 32, Output channels = 64
        self.offsets2 = nn.Conv2d(32, 18, kernel_size=3, stride=1, padding=1)  # Offsets for deformable conv2
        self.deform_conv2 = DeformConv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.batchnorm2 = nn.BatchNorm2d(64)  # BatchNorm for deformable conv2

        # Pooling layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)  # Pooling reduces spatial dimensions by 2

        # Fully connected layers
        self.fc1 = nn.Linear(64 * 7 * 7, 128)  # Flattened size after pooling
        self.dropout1 = nn.Dropout(p=dropout_val)  # Dropout for fc1
        self.fc2 = nn.Linear(128, 10)  # Output layer for 10 classes
        self.dropout2 = nn.Dropout(p=dropout_val)  # Dropout for fc2
```

The EnhancedDeformableCNN model builds upon the previous deformable architecture with key regularization and normalization techniques. As highlighted in the code, batch normalization layers are added after each deformable convolution to stabilize training by normalizing feature activations.

Additionally, dropout mechanisms with a configurable probability are incorporated before the fully connected layers to prevent overfitting by randomly zeroing features during training.

4.6 Data Augmentation

Another approach employs MixUp, a data augmentation technique that creates virtual training examples through weighted linear interpolation of random sample pairs. The core mechanism samples a mixing coefficient λ (lambda) from a distribution, then generates new training instances by combining both inputs and labels. The hyperparameter α (alpha) controls the strength of interpolation between samples.

```
# Applies MixUp to a batch of inputs and their labels.
def mixup_data(x, y, alpha=1.0):
    if alpha > 0:
        lam = torch.distributions.Beta(alpha, alpha).sample().item()
    else:
        lam = 1.0

    batch_size = x.size(0)
    index = torch.randperm(batch_size).to(x.device)  # Shuffle indices

    mixed_x = lam * x + (1 - lam) * x[index, :]
    mixed_y = lam * y + (1 - lam) * y[index]

    return mixed_x, mixed_y, lam
```

The '*mixup_data*' function first determines the interpolation strength by sampling λ from a Beta distribution when α > 0, or defaults to 1.0 (no mixing) when α=0. It then creates random pairings to shuffle batch indices, ensuring diverse combinations during training. The function performs element-wise linear interpolation between original samples and their paired counterparts for both images and labels, returning the mixed data and the sampled λ value.

```
# Convert labels to one-hot encoding for MixUp
labels_onehot = nn.functional.one_hot(labels, num_classes=10).float()

# Apply MixUp augmentation
images, labels_mixed, _ = mixup_data(images, labels_onehot, alpha)

# Forward pass
optimizer.zero_grad()
outputs = model(images)
loss = criterion(outputs, labels_mixed)  # MixUp uses BCEWithLogitsLoss
loss.backward()
optimizer.step()

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

# For MixUp, accuracy here is not entirely valid because labels are mixed.
# However, we can still compare the raw predictions with the primary labels
correct += (predicted == labels).sum().item()
```

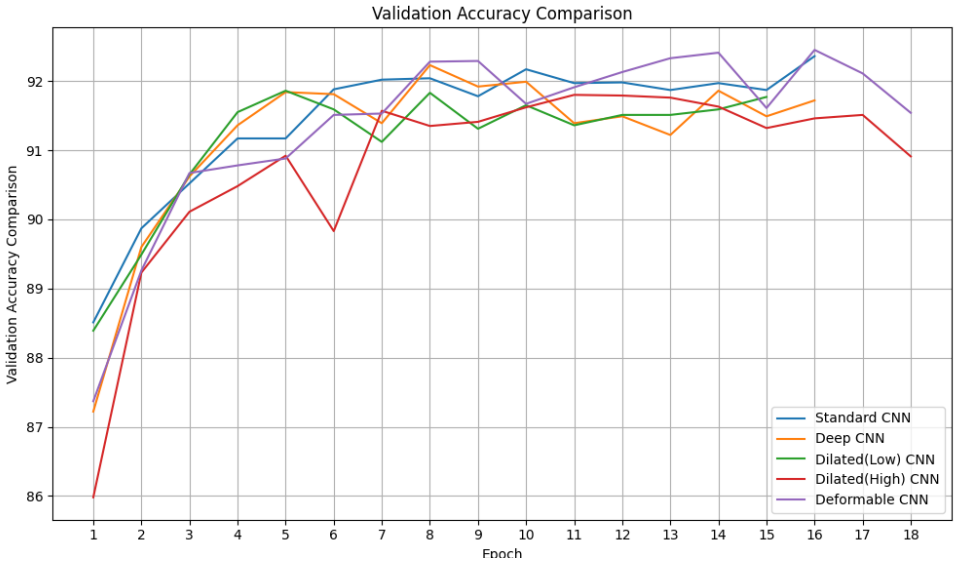
The training loop is also updated to accommodate MixUp by introducing several crucial modifications. Original class labels are converted to one-hot encoded vectors, creating probability distributions suitable for interpolation. These prepared tensors are passed to the *'mixup_data'* function, producing mixed images and soft label targets. During backpropagation, the standard cross-entropy loss is replaced with binary cross-entropy loss (BCEWithLogitsLoss), which properly handles the soft probability distributions resulting from label mixing. The vectors cannot be directly compared so we compare the raw predictions with the primary labels to validate the number of correctly identified labels.

5 RESULTS AND ANALYSIS

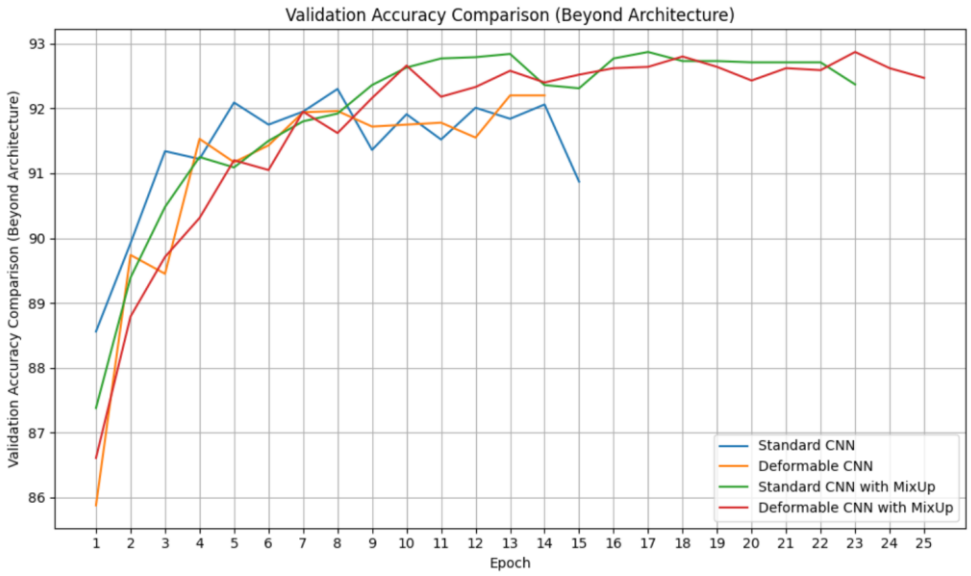
Model	Best Validation Accuracy	Epoch
Standard CNN	92.30%	8
Deep CNN	91.94%	11
Dilated(Low) CNN	91.91%	12
Dilated(High) CNN	91.88%	14
Deformable CNN	92.20%	13
Enhanced Deformable CNN	92.20%	18
Standard CNN with MixUp	92.87%	17
Deformable CNN with MixUp	92.87%	23

The validation accuracy results demonstrate that most models perform comparably within a narrow range. Both the Standard CNN and Deformable CNN show noticeable improvements when combined with MixUp, achieving the highest validation accuracies among all tested models. Conventional CNN architectures remain competitive, while both deeper networks and dilated convolutions show similar performance levels. This suggests that the proper combination of architectural innovations and data augmentation techniques can effectively complement each other to enhance model generalization. This trade-off between improved accuracy and longer training time represents an important consideration for practical applications.

5.1 VALIDATION ACCURACY COMPARISON



The validation accuracy curves reveal distinct learning patterns across CNN architectures. All models share a steep initial improvement phase followed by gradual refinement. The Deformable CNN demonstrates the strongest late-stage improvement trend, suggesting superior learning capacity with extended training. Standard CNN exhibits stability with minimal fluctuation, while Deep CNN shows moderate oscillation despite quick early gains. The Dilated CNN variants present contrasting behaviours for low and high dilation factors. The low one plateaus earlier while the other experiences more pronounced volatility. This means that not necessarily that a high dilation would be better.



MixUp data augmentation consistently enhances model performance across CNN architectures. While the Standard CNN shows quick early convergence but later decline, and the Deformable CNN maintains moderate stability, both MixUp-enhanced variants achieve higher peak accuracies and better long-term performance. Standard CNN with MixUp demonstrates excellent stability, while Deformable CNN with MixUp reaches the highest overall accuracy despite some fluctuations.

The benefits of MixUp become increasingly apparent during extended training, with these models continuing to improve after traditional models stop. This suggests that effective data augmentation strategies can provide greater performance gains than architectural modifications alone. The results highlight the importance of both choosing appropriate regularization techniques and allowing sufficient training time to fully realize their benefits.