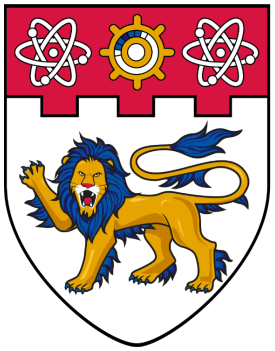

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4042 Group Project

Topic E: Clothing Classification

Members:

Min Kabar Kyaw U2021858K (Group Leader)

Chua Zi Jian U2022354J

Huang QiYuan U2021333H

Content

1. Introduction

Figure 1: FashionMNIST Dataset [1]

Fashion-MNIST (Figure 1) is a dataset created by Zalando [2], featuring images of their clothing articles. It includes a training set with 60,000 examples and a test set with 10,000 examples. Each image is a 28x28 grayscale picture and is labeled into one of 10 classes. Zalando intends Fashion-MNIST to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It maintains the same image size, structure, and label format.

In this project, we approached the problem by **applying a series of data augmentation techniques** to enhance the dataset and employed a **ResNet model** [3]. Various modifications were done to the ResNet model and different experiments were conducted to achieve better model generalization. This includes the implementation of a deformable convolutional layer, the use of DropBlock, and incorporating stochastic depth techniques.

All the techniques are discussed in the following sections.

2. Baseline Model Review

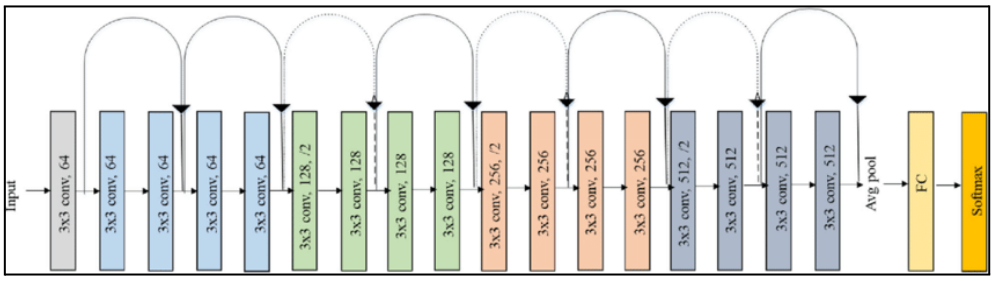

Figure 2: ResNet-18 Architecture [4]

ResNet18 (Figure 2) starts with a 7x7 convolutional layer with 64 filters and includes four residual blocks. Each block contains multiple convolutional layers with skip connections, helping to train deep networks effectively. Global average pooling is used to reduce spatial dimensions, followed by a fully connected layer for class predictions.

ResNet was selected for its remarkable ability to effectively handle deep neural networks. While Fashion-MNIST is a relatively straightforward dataset compared to more complex image datasets, the

capability to construct deep networks is **beneficial for recognizing subtle and intricate features in clothing images.**

Additionally, ResNet's inclusion of **skip connections** plays a pivotal role in mitigating the vanishing gradient problem, ensuring that gradients can efficiently propagate through numerous network layers.
The utilization of ResNet has also proven to be conducive to model generalization, as it facilitates the learning of both high-level and low-level features, a vital characteristic for discerning and classifying various clothing items.

Furthermore, the **availability of pre-trained ResNet models**, which could be easily fine tuned or adapted for Fashion-MNIST, facilitated an **expedited model convergence**, which ultimately led to enhanced classification performance.

The performance of the original ResNet18 Model without fine tuning or modifications would be discussed below. From the accuracy metric, **it is noteworthy that ResNet is an ideal architecture** to employ, achieving an impressive validation accuracy of 87.42% after just one epoch.

(*See Appendix A for ResNet implementation)*

To adapt to the FashionMnist dataset,

1. self.model.conv1 has been reconfigured to accept only 1 channel since our dataset is in grayscale format.
2. layer 4 has been removed to reduce complexity of the ResNet18 model, to effectively prevent overfitting.
3. fully connected layer has been reconfigured to accept input channels of 256, corresponding to the output of the last layer (layer 3).

```
Train Epoch: 5  Average Loss: 0.160318  Accuracy: 94.33%
Val Loss: 0.2001, Val Accuracy: 92.84%
Train Epoch: 6  Average Loss: 0.142305  Accuracy: 94.96%
Val Loss: 0.1910, Val Accuracy: 93.18%
Train Epoch: 7  Average Loss: 0.127227  Accuracy: 95.48%
Val Loss: 0.2035, Val Accuracy: 92.54%
Train Epoch: 8  Average Loss: 0.107077  Accuracy: 96.28%
Val Loss: 0.1800, Val Accuracy: 93.69%
Train Epoch: 9  Average Loss: 0.092213  Accuracy: 96.82%
Val Loss: 0.1912, Val Accuracy: 93.35%
Train Epoch: 10 Average Loss: 0.073183  Accuracy: 97.43%
Val Loss: 0.2244, Val Accuracy: 92.97%
```

Figure 3: Training/Validation Accuracy and Loss for unmodified ResNet18

It took **only 8 epochs** for ResNet18 to **converge** (Figure 3).
In summary, ResNet's adaptability, efficiency, and compatibility with transfer learning made it the ideal choice for our Fashion-MNIST image classification project.

3. Data Augmentation Methods

Data Augmentation technique serves as a form of regularization and **encourages** the **model** to be more **robust** to occlusions and variations in the dataset, ultimately **improving its generalization capabilities**.

3.1 Basic Augmentation Techniques

Basic Augmentation technique is a relatively straightforward data augmentation technique in which some simple transformation is done to the original data. There are various techniques such as Random Cropping, Vertical Flip and Horizontal Flip, Color Jitter, Rotation, Shearing and Translation.

In this project we use Random Cropping, Vertical Flip and Horizontal Flip (Figure 4).

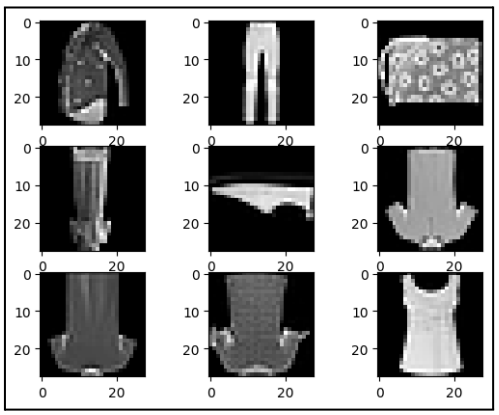

Figure 4: Basic Augmented FashionMNIST data

3.2 MixUp

Mixup [5] is a data augmentation technique that generates a weighted combination of random image pairs from the training data. **The key idea behind MixUp is that linear interpolation of 2 inputs should correspond to the same linear interpolation of their respective labels.** It has been empirically shown to substantially improve test performance and robustness to adversarial noise of state-of-the-art neural network architectures. Visually, depending on the mix ratio we can observe a “faint afterimage” of the image with the lower weight (Figure 5).

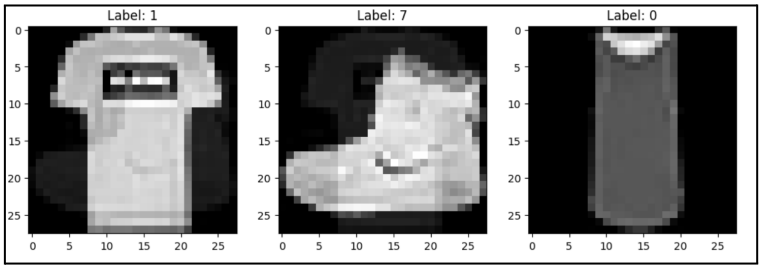

Figure 5: MixUp-ed FashionMNIST data

3.3 CutMix

CutMix [6] is a data augmentation technique that involves cutting a random rectangular portion from one image and pasting it onto another image (Figure 6). CutMix can significantly boost the performance of image classification models as it **introduces diversity** and **challenges the model to handle mixed visual features from different items**.

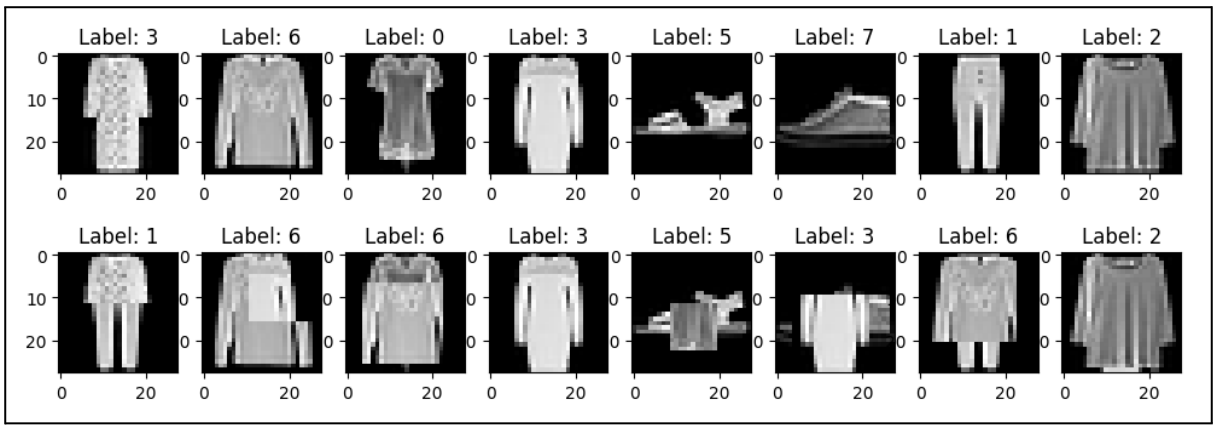

Figure 6: CutMix-ed FashionMNIST data

3.4 Comparison of Different Data Augmentation in isolation on ResNet-18

In this section, we compare different data augmentation techniques, where each technique is applied in isolation to the training dataset. For validation, we use the original data as the model is meant to be trained for images similar to the original dataset.

```
Basic Technique
Train Epoch: 1  Average Loss: 1.091514  Accuracy: 60.90%
Val Loss: 0.9184, Val Accuracy: 67.94%
Train Epoch: 2  Average Loss: 0.839215  Accuracy: 70.08%
Val Loss: 0.9057, Val Accuracy: 68.39%
Train Epoch: 3  Average Loss: 0.747357  Accuracy: 73.36%
Val Loss: 0.8897, Val Accuracy: 69.58%
Train Epoch: 4  Average Loss: 0.686983  Accuracy: 75.40%
Val Loss: 0.9147, Val Accuracy: 68.07%
Train Epoch: 5  Average Loss: 0.646526  Accuracy: 76.74%
Val Loss: 0.7706, Val Accuracy: 73.38%
Train Epoch: 6  Average Loss: 0.616033  Accuracy: 77.73%
Val Loss: 0.9110, Val Accuracy: 70.30%
Train Epoch: 7  Average Loss: 0.586642  Accuracy: 78.87%
Val Loss: 0.6513, Val Accuracy: 76.61%
Train Epoch: 8  Average Loss: 0.562844  Accuracy: 79.68%
Val Loss: 0.6929, Val Accuracy: 76.06%
Train Epoch: 9  Average Loss: 0.540076  Accuracy: 80.32%
Val Loss: 0.6550, Val Accuracy: 76.72%
Train Epoch: 10 Average Loss: 0.514734  Accuracy: 81.13%
Val Loss: 0.6630, Val Accuracy: 76.68%
```

Figure 7: Basic Augmentation Technique Training/Validation Accuracy & Loss

```
MixUp
Train Epoch: 1  Average Loss: 1.509343  Accuracy: 37.85%
Val Loss: 2.2204, Val Accuracy: 33.46%
Train Epoch: 2  Average Loss: 1.331283  Accuracy: 45.17%
Val Loss: 1.0156, Val Accuracy: 69.07%
Train Epoch: 3  Average Loss: 1.245307  Accuracy: 48.94%
Val Loss: 1.1395, Val Accuracy: 58.87%
Train Epoch: 4  Average Loss: 1.182938  Accuracy: 51.70%
Val Loss: 0.9481, Val Accuracy: 69.92%
Train Epoch: 5  Average Loss: 1.139413  Accuracy: 53.41%
Val Loss: 0.7945, Val Accuracy: 77.79%
Train Epoch: 6  Average Loss: 1.101081  Accuracy: 55.48%
Val Loss: 0.6914, Val Accuracy: 79.85%
Train Epoch: 7  Average Loss: 1.069141  Accuracy: 56.55%
Val Loss: 0.7596, Val Accuracy: 76.40%
Train Epoch: 8  Average Loss: 1.043081  Accuracy: 57.82%
Val Loss: 1.0842, Val Accuracy: 59.47%
Train Epoch: 9  Average Loss: 1.021590  Accuracy: 58.64%
Val Loss: 0.6052, Val Accuracy: 82.86%
Train Epoch: 10 Average Loss: 0.998145  Accuracy: 59.47%
Val Loss: 0.6526, Val Accuracy: 78.02%
```

Figure 8: MixUp Technique Training/Validation Accuracy & Loss

```
CutMix
Train Epoch: 1  Average Loss: 0.746700  Accuracy: 73.39%
Val Loss: 0.4558, Val Accuracy: 83.20%
Train Epoch: 2  Average Loss: 0.517566  Accuracy: 81.55%
Val Loss: 0.3096, Val Accuracy: 88.91%
Train Epoch: 3  Average Loss: 0.461144  Accuracy: 83.58%
Val Loss: 0.2889, Val Accuracy: 89.53%
Train Epoch: 4  Average Loss: 0.426749  Accuracy: 84.80%
Val Loss: 0.4197, Val Accuracy: 85.46%
Train Epoch: 5  Average Loss: 0.406363  Accuracy: 85.46%
Val Loss: 0.2408, Val Accuracy: 91.33%
Train Epoch: 6  Average Loss: 0.384689  Accuracy: 86.36%
Val Loss: 0.2802, Val Accuracy: 89.70%
Train Epoch: 7  Average Loss: 0.368613  Accuracy: 86.72%
Val Loss: 0.2370, Val Accuracy: 91.38%
Train Epoch: 8  Average Loss: 0.353785  Accuracy: 87.22%
Val Loss: 0.2320, Val Accuracy: 91.58%
Train Epoch: 9  Average Loss: 0.335885  Accuracy: 87.90%
Val Loss: 0.2195, Val Accuracy: 92.12%
Train Epoch: 10 Average Loss: 0.327953  Accuracy: 88.41%
Val Loss: 0.2632, Val Accuracy: 90.92%
```

Figure 9: CutMix Technique Training/Validation Accuracy & Loss

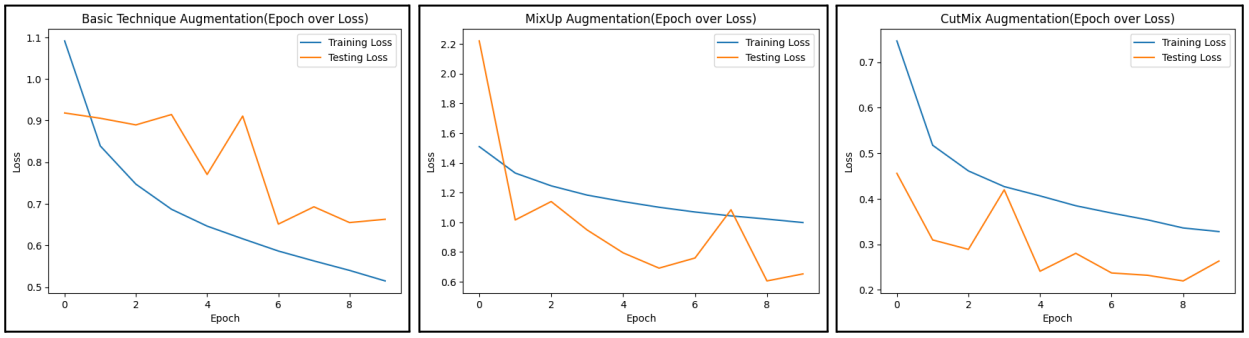

Figure 10: Training/Validation Loss Graph Comparison

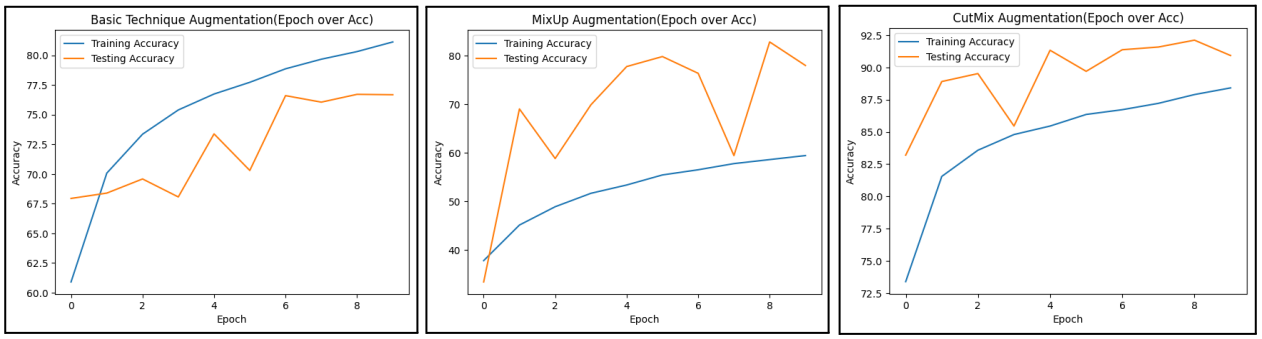

Figure 11: Training/Validation Accuracy Graph Comparison

We perform 10 epochs of training and validation on the basic ResNet18 model and the results are reflected in Figures 7 to 11 above.

3.5 Analysis of Results

1. We observe that the CutMix technique has achieved the both highest accuracy and lowest loss for training and validation compared to others (Figure 9, 10, 11).
2. We observe an interesting phenomena for the results of basic augmentation techniques. The training loss is lower than validation loss and the training accuracy is higher than validation accuracy, which suggests that it might not be a feasible solution for our current problem and dataset.
3. For MixUp and CutMix they have similar graphs where training accuracy is lower than validation accuracy, possibly due to the complexity of augmented data when training.

Thus, **CutMix** yields the best results out of all 3 techniques.

4. Deformable Convolution Layers

Deformable convolution layers [7] introduce spatial adaptability to the convolutional operation. Unlike traditional convolution layers, deformable convolutions allow the **receptive fields** to be **dynamically adjusted** according to the features present in the input. This adaptability is achieved through **learnable offsets** (Figure 12), which enable the network to focus on more relevant regions within the feature maps. The advantages of deformable convolutional layers include:

1. **Improved Spatial Adaptation**: Enables the network to adapt to the specific spatial characteristics of the input data, allows the network to learn the transformation of the convolutional grid based on the content of the image and this leads to better feature extraction and increased sensitivity to object details.
2. **Enhanced Object Localization**: Enables the network to focus on object boundaries and fine-grained details, which can significantly improve object localization accuracy which is crucial for our FashionMnist problem.
3. **Reduced Sensitivity to Grid Sampling**: Reduce the sensitivity to the grid sampling, making the model more robust to small changes in object positions and scales. This is valuable in scenarios where objects have varying sizes and positions within images.

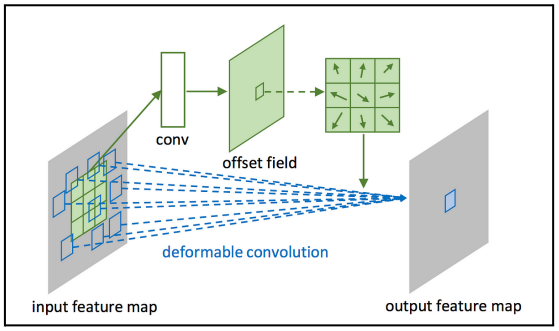

Figure 12: Visual Representation of Deformable Convolution Layers [8]

Instead of a deformable convolution network, we decided to perform a deformable ResNet network. However, we have to **empirically determine** which layers of the ResNet18 to perform the deform convolution to achieve the best accuracy.

BasicBlock Class -> creates a block in a layer the conventional way
(See Appendix B)

DeformConvBlock Class -> creates a deformable block in a layer including:

1. offset convolutional layer that takes the input feature map and produces offset values that are learned during training.These offsets are used to adjust the convolutional kernel's positions for deformable convolution.
2. deform convolutional layer that performs deformable convolution based on the learned offsets.
3. a batch normalization layer and RELU activation function.

(See Appendix C)

CustomResNet implementation
(See Appendix D)

Example : Deformable convolution layer for layer 3
(See Appendix E)

```
possible_layer_configurations = [
    [0],          # Deform none
    [1],          # Deform layer 1
    [2],          # Deform layer 2
    [3],          # Deform layer 3
    [1, 2],       # Deform layers 1 and 2
    [1, 3],       # Deform layers 1 and 3
    [2, 3],       # Deform layers 2 and 3
    [1, 2, 3]     # Deform all layers 1, 2, and 3
]

all_model_results = []
```

Figure 13: All feasible configurations

(See Appendix F)

In order to determine which combination of deformable convolutional layers would result in the highest accuracy, we conducted experiments with **all feasible configurations** (Figure 13). Then, a **graph of validation accuracy against epochs** for different combinations was plotted to visualize the performance and deduce the best combination.

Example: Output for deformable convolution layers {1,3}
To prevent overfitting, it was run for only 10 epochs with patience of 3.
(See Appendix G)

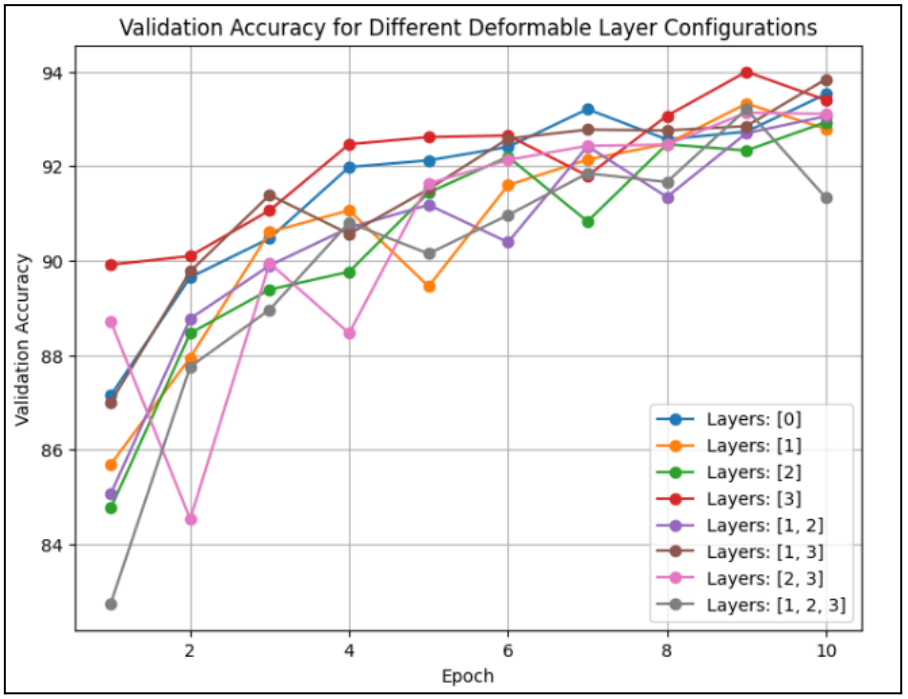

Figure 14: Validation accuracy against epoch graph for different configurations

The **best configuration** is **deforming layers 1 and 3** as it gave the highest accuracy of 93.84% (Figure 14).

The **optimal model** would finally be retrained on **both training and validation datasets** and tested on the **test dataset**.

```
Train Epoch: 1  Average Loss: 0.669438  Accuracy: 76.77%
Train Epoch: 2  Average Loss: 0.459311  Accuracy: 83.70%
Train Epoch: 3  Average Loss: 0.409547  Accuracy: 85.50%
Train Epoch: 4  Average Loss: 0.377811  Accuracy: 86.55%
Train Epoch: 5  Average Loss: 0.351264  Accuracy: 87.50%
Train Epoch: 6  Average Loss: 0.337739  Accuracy: 87.97%
Train Epoch: 7  Average Loss: 0.317807  Accuracy: 88.51%
Train Epoch: 8  Average Loss: 0.307976  Accuracy: 88.89%
Train Epoch: 9  Average Loss: 0.292871  Accuracy: 89.53%
Train Epoch: 10 Average Loss: 0.280956  Accuracy: 89.88%
Val Loss: 0.2058, Val Accuracy: 92.96%
```

Figure 15: Final training/ testing accuracy and loss (Val Loss refers to Test Loss, Val Accuracy refers to Test Accuracy)

The final model has a high accuracy of 92.96% (Figure 15).

Experiments: *(See Appendix H for implementation)*

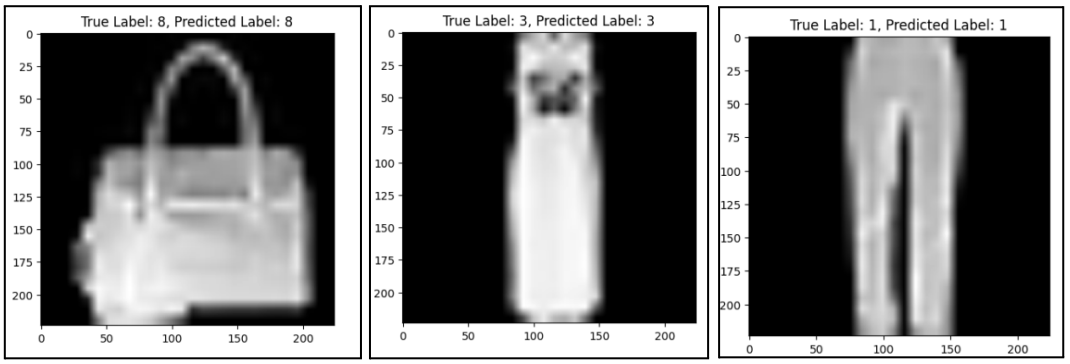

Figure 16: Experiments with test_dataset samples

After hyperparameter tuning, the model delivered largely accurate results (Figure 16).

5. Regularization Methods

We investigate two regularization methods, DropBlock [9] and Stochastic Depth [10] with our previously obtained optimal model. We first perform hyperparameter tuning for the probability of activating the regularization (p) and use it in the subsequent comparison.

DropBlock is a structured form of dropout where a **contiguous region** of the image is dropped in contrast to normal drop out that is random. The intuition behind this approach is that contiguous regions are more likely to be related as part of a feature (e.g. head, arms, leg, etc.).

Stochastic Depth aims to shrink the depth of a network during training, while keeping it unchanged during testing. This is achieved by randomly dropping entire ResBlocks during training and bypassing their transformations through skip connections.

Similar to the above section on Deformable Convolution, we create two custom classes that extend **BasicBlock, DropBlockBasicBlock** and **StoDepthBasicBlock** that implement the respective regularization methods. Our general experiment methodology remains the same in terms of training, validation and testing as above.

5.1 Hyperparameter tuning of p

We perform hyperparameter tuning of p in the interval (0, 1] with increments of 0.1. Block size for DropBlock is fixed at 7 which is stated as the optimal value in the original paper. We augment our dataset with **CutMix**, which our previous experiment on data augmentation has shown to give the best results.

For both DropBlock and Stochastic Depth, p value of 0.1 was found to be optimal (Figure 17) in terms of giving the best validation accuracy.

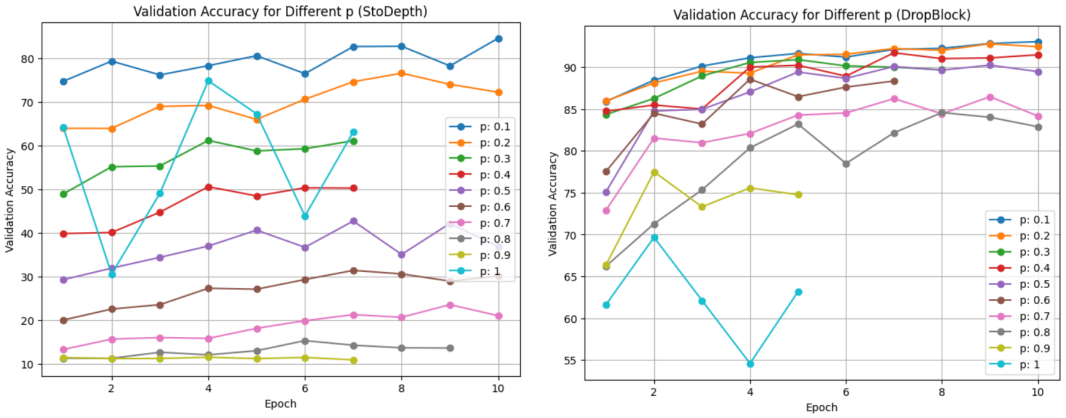

Figure 17: Hyperparameter tuning of DropBlock and Stochastic Depth

5.2 Comparison of DropBlock & Stochastic Depth

We then re-initialize our DropBlock and Stochastic Depth models with our optimal p values, and check its performance on our test data set.

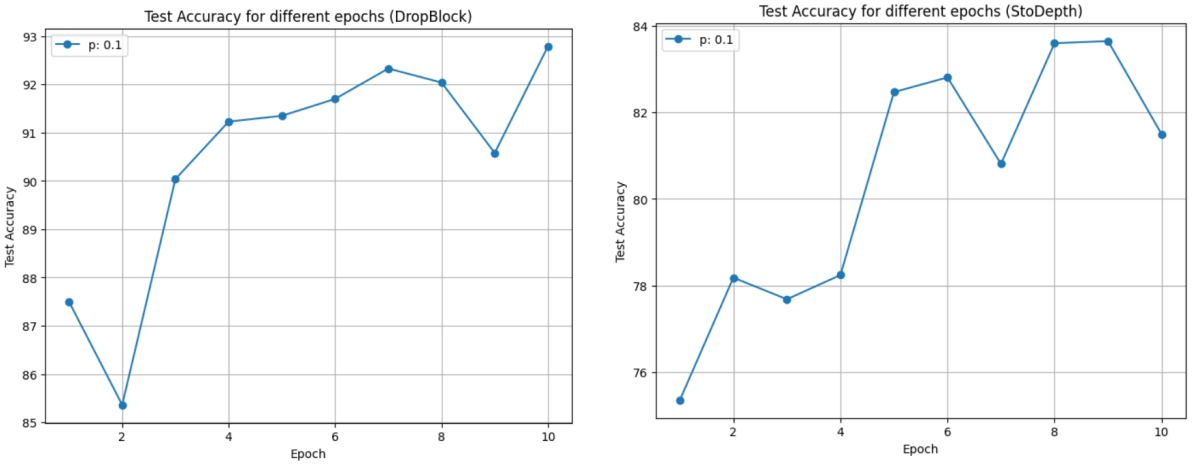

Figure 18: Test accuracies for DropBlock and Stochastic Depth

5.3 Analysis of Results

Based on our results, we see that the model with **DropBlock** performs better than the other with a test accuracy of 92.79% compared to 81.50% (Figure 18). One possible reason for this difference could be that the random removal of layers in **Stochastic Depth** does not have enough structure unlike **DropBlock**. Methods like **Stochastic Depth** tend to work well when there are many input or output branches, but fail otherwise [5]. We conjecture that the residual blocks that are present in ResNet are insufficient for such methods to be effective, hence the lower test accuracy.

6. Conclusion

Our experimentation with different data augmentations and model modifications has shown some insights. In particular, we note that CutMix as a data augmentation method is effective in improving model accuracy. However, we recognize that the effectiveness of different data augmentation methods may also vary based on the nature of the dataset. In the case of FashionMNIST, our dataset consists solely of grayscale images related to clothing articles. Data augmentation methods may fare differently in other datasets.

ResNet with Deformable Convolution resulted in a slight accuracy boost compared to vanilla ResNet, proving the value of incorporating novel techniques into our models. The model with deformable convolution becomes more adept at capturing the necessary features for distinguishing between different fashion items in this particular dataset, making it well-suited to the complexities and nuances presented by FashionMNIST. However, a drawback we foresee for Deformable Convolution would be the time needed to obtain optimal hyperparameters. Testing all possible deformable layer configurations in models with much more layers would prove infeasible as there are 2^n possible configurations.

The investigation on DropBlock and Stochastic Depth reveals that not all regularization methods should be used in varying circumstances. In particular, the performance drop of Stochastic Depth suggests that our model or BasicBlock configuration was not compatible with Stochastic Depth in order to get a performance boost (not enough branches).

Due to the relatively high complexity of ResNet-18 for the FashionMNIST dataset, overfitting can occur quite easily, even with the removal of layer 4. Consequently, it may be necessary to run fewer epochs. As such, improvements in accuracy may appear less pronounced since the base ResNet18 model already attains a high accuracy rate.

References

[1] Irene Pylypenko, “Exploring Neural Networks with fashion MNIST”, Available from: https://medium.com/@ipylypenko/exploring-neural-networks-with-fashion-mnist-b0a8214b7b7b [Accessed November 10, 2023]

[2] Zalando, Fashion-MNIST, Available from: https://github.com/zalandoresearch/fashion-mnist [Accessed October 28, 2023]

[3] He, Kaiming & Zhang, Xiangyu & Ren, Shaoqing & Sun, Jian. (2015). Deep Residual Learning for Image Recognition. 7.

[4] A Deep Learning Approach for Automated Diagnosis and Multi-Class Classification of Alzheimer’s Disease Stages Using Resting-State fMRI and Residual Neural Networks. (n.d.). https://www.researchgate.net/figure/Original-ResNet-18-Architecture_fig1_336642248

[5] Zhang, Hongyi & Cisse, Moustapha & Dauphin, Yann & Lopez-Paz, David. (2017). mixup: Beyond Empirical Risk Minimization.

[6] Yun, Sangdoo & Han, Dongyoon & Chun, Sanghyuk & Oh, Seong Joon & Yoo, Youngjoon & Choe, Junsuk. (2019). CutMix: Regularization Strategy to Train Strong Classifiers With Localizable Features. 6022-6031. 10.1109/ICCV.2019.00612.

[7] Zhu, X., Hu, H., Lin, S., & Dai, J. (2018). Deformable ConvNets v2: More Deformable, Better Results. *ArXiv:1811.11168 [Cs]*. https://arxiv.org/abs/1811.11168

[8] Felix Lau, “*Notes on “Deformable Convolutional Networks.”* (n.d.)”. Available from: https://medium.com/@phelixlau/notes-on-deformable-convolutional-networks-baaabbc11cf3 [Accessed November 10, 2023]

[9] Ghiasi, Golnaz & Lin, Tsung-Yi & Le, Quoc. (2018). DropBlock: A regularization method for convolutional networks.

[10] Huang, Gao & Sun, Yu & Liu, Zhuang & Sedra, Daniel & Weinberger, Kilian. (2016). Deep Networks with Stochastic Depth. 9908. 646-661. 10.1007/978-3-319-46493-0_39.

Appendix

Appendix A: Original Resnet18 Implementation

```
# Define the ResNet18 architecture
class ResNet18(nn.Module):
    def __init__(self, num_classes):
        super(ResNet18, self).__init__()

        self.model = models.resnet18(pretrained=True)
        self.model.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.model.layer4 = nn.Identity()
        self.model.fc = nn.Linear(256, num_classes)

    def make_layer(self, in_channels, out_channels, blocks, stride1, stride=1):
        layers = []
        layers.append(BasicBlock(in_channels, out_channels, stride, stride1))
        for _ in range(1, blocks):
            layers.append(BasicBlock(out_channels, out_channels, stride, stride1=1))
        return nn.Sequential(*layers)



    def forward(self, x):
        return self.model(x)
```

Appendix B: Basic Block class (code)

```
# Define a BasicBlock for the ResNet-like architecture
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, stride1=2):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=2, bias=False),
            nn.BatchNorm2d(out_channels)
            ) if (stride != 1 or in_channels != out_channels) and (out_channels == 128 or out_channels == 256 or out_channels ==512) else nn.Identity()


    def forward(self, x):
        residual = self.downsample(x)
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x += residual
        x = self.relu(x)
        return x
```

Appendix C: DeformConvBlock class (Code)

```
# Define the Deformable Convolution Block
class DeformConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DeformConvBlock, self).__init__()
        self.offset_conv = nn.Conv2d(in_channels, 18, kernel_size=3, padding=1)  # 18 channels for x, y offsets, and masks
        self.deform_conv = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1, bias=False)
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        offset = self.offset_conv(x)

        # Adjust the mask channels to be half of the offset channels
        mask = offset[:, :offset.size(1) // 2, ...]

        x = torchvision.ops.deform_conv2d(input=x,
                                          offset=offset,
                                          weight=self.deform_conv.weight,
                                          bias=None,  # Set bias to None if you don't want bias
                                          padding=self.deform_conv.padding,
                                          mask=mask,  # Use the adjusted mask
                                          stride=self.deform_conv.stride,
                                          dilation=self.deform_conv.dilation)
        x = self.bn(x)
        x = self.relu(x)
        return x
```

Appendix D: Modified Resnet-18 with Deformable Convolution Block class (code)

```python
# Define the CustomResNet architecture that uses DeformConvBlock and Resnet18
class CustomResNet(nn.Module):
    def __init__(self, num_classes, layers_to_deform):
        super(CustomResNet, self).__init__()

        self.model = models.resnet18(pretrained=True)
        self.model.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=False)

        if 1 in layers_to_deform:
            self.model.layer1 = self.make_layer_with_deform(64, 64, blocks=2, stride=1, stride1=1)
        if 2 in layers_to_deform:
            self.model.layer2 = self.make_layer_with_deform(64, 128, blocks=2, stride=1, stride1=2)
        if 3 in layers_to_deform:
            self.model.layer3 = self.make_layer_with_deform(128, 256, blocks=2, stride=1, stride1=2)
        self.model.layer4 = nn.Identity()
        self.model.fc = nn.Linear(256, num_classes)

    def make_layer(self, in_channels, out_channels, blocks, stride1, stride=1):
        layers = []
        layers.append(BasicBlock(in_channels, out_channels, stride, stride1))
        for _ in range(1, blocks):
            layers.append(BasicBlock(out_channels, out_channels, stride, stride1=1))
        return nn.Sequential(*layers)

    def make_layer_with_deform(self, in_channels, out_channels, blocks, stride, stride1):
        layers = []
        layers.append(BasicBlock(in_channels, out_channels, stride, stride1))
        layers.append(DeformConvBlock(out_channels, out_channels))  # Add a deformable convolution block
        for _ in range(1, blocks - 1):
            layers.append(BasicBlock(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)
```

Appendix E: Example of Deformable convolution layer for layer 3

```
(layer3): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): DeformConvBlock(
    (offset_conv): Conv2d(256, 18, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (deform_conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
```

Appendix F: Hyperparameter tuning of deformable convolution implementation

```python
for layers_to_deform in possible_layer_configurations:

    print(f"Training with deformable layers: {layers_to_deform}")
    model = CustomResNet(num_classes, layers_to_deform).to(device)
    optimizer = Adam(model.parameters(), lr=3e-4)
    #print(model)

    validation_accuracies = []
    best_val_acc = float('-inf')
    current_patience = 0

    for epoch in range(1, num_epochs + 1):
        train_cutmix(model, criterion, train_loader, optimizer, epoch, alpha, device)
        val_acc = val(model, criterion, val_loader, device)
        validation_accuracies.append(val_acc[0])
        if val_acc[0] > best_val_acc:
            best_val_acc = val_acc
            current_patience = 0
        else:
            current_patience += 1

        if current_patience >= patience:
            print(f'Early stopping: No improvement for {patience} epochs.')
            break
    # Store the results for this model configuration
    model_results = {
        "layers_to_deform": layers_to_deform,
        "validation_accuracies": validation_accuracies
    }

    # Append the results to the list of all model results
    all_model_results.append(model_results)
```

Appendix G: Output for deforming layers {1,3}

```
Training with deformable layers: [1, 3]
Train Epoch: 1  Average Loss: 0.701901  Accuracy: 75.97%
Val Loss: 0.3545, Val Accuracy: 87.01%
Train Epoch: 2  Average Loss: 0.474969  Accuracy: 83.43%
Val Loss: 0.2772, Val Accuracy: 89.78%
Train Epoch: 3  Average Loss: 0.420491  Accuracy: 85.11%
Val Loss: 0.2406, Val Accuracy: 91.39%
Train Epoch: 4  Average Loss: 0.390451  Accuracy: 86.10%
Val Loss: 0.2614, Val Accuracy: 90.57%
Train Epoch: 5  Average Loss: 0.365492  Accuracy: 86.97%
Val Loss: 0.2371, Val Accuracy: 91.52%
Train Epoch: 6  Average Loss: 0.350836  Accuracy: 87.33%
Val Loss: 0.1992, Val Accuracy: 92.58%
Train Epoch: 7  Average Loss: 0.327823  Accuracy: 88.30%
Val Loss: 0.2012, Val Accuracy: 92.78%
Train Epoch: 8  Average Loss: 0.316179  Accuracy: 88.70%
Val Loss: 0.1993, Val Accuracy: 92.76%
Train Epoch: 9  Average Loss: 0.301465  Accuracy: 89.14%
Val Loss: 0.1953, Val Accuracy: 92.85%
Train Epoch: 10 Average Loss: 0.286816  Accuracy: 89.75%
Val Loss: 0.1729, Val Accuracy: 93.84%
```

Appendix H: Implementation for testing model on some test images

```python
random_index = random.randint(0, len(test_dataset) - 1)
data, label = test_dataset[random_index]
data=data.to(device)

output = model(data.unsqueeze(0))
pred = torch.max(output, 1)[1].item()

# Display the image
data = data.cpu().numpy()
plt.imshow(data[0], cmap='gray')
plt.title(f'True Label: {label}, Predicted Label: {pred}')
plt.show()
```