Deep Learning Practical Work 1-a and 1-b

William Khieu
Akli Mohamed Ait-Oumeziane

October 28, 2024

Contents

Section 1 - Introduction to Convolutional Networks

Questions

1.

- **Considering a single convolution filter of padding p, stride s, and kernel size k, for an input of size $x \times y \times z$, what will be the output size?**

  When applying a convolution filter of kernel size $k \times k$, stride s, and padding p, on an input with dimensions $x \times y \times z$ (where x is the height, y is the width, and z is the number of input channels), the output size can be calculated as follows:

  Output Height (x_{out}):

  $$x_{\text{out}} = \left\lfloor \frac{x + 2p - k}{s} + 1 \right\rfloor$$

  Output Width (y_{out}):

  $$y_{\text{out}} = \left\lfloor \frac{y + 2p - k}{s} + 1 \right\rfloor$$

  Output Depth (z_{out}):

  This is determined by the number of filters f. If you're using f convolutional filters, the depth of the output will be f.

  So, the output size will be $x_{\text{out}} \times y_{\text{out}} \times f$.

- **How much weight is there to learn?**

  For a convolutional layer, the number of weights that need to be learned is determined by the size of the kernel, the number of input channels, and the number of output filters. The formula to calculate the total number of weights is:

  $$\text{Total Weights} = k \times k \times z \times f$$

  This includes the filter weights for each channel and filter. Additionally, you'll have f bias terms (one per filter), so the total parameters including biases will be:

  $$\text{Total Parameters} = (k \times k \times z \times f) + f$$

- **How much weight would it have taken to learn if a fully-connected layer were to produce an output of the same size?**

  If a fully connected layer were used instead of the convolutional layer to produce the same output size, the number of parameters would be much higher, as it connects each input element to each output element.

  The input size is $x \times y \times z$. The output size is $x_{\text{out}} \times y_{\text{out}} \times f$. The number of weights in a fully connected layer would be:

  $$\text{Weights in Fully Connected Layer} = (x \times y \times z) \times (x_{\text{out}} \times y_{\text{out}} \times f)$$

2. What are the advantages of convolution over fully-connected layers? What is its main limitation?

Advantages of Convolutional Layers	Main Limitation
Parameter efficiency: Fewer parameters to learn	Limited to local receptive fields
Spatial hierarchies: Learn local to global features	Difficulty in capturing global context directly
Translation invariance: Robust to shifts in the input	Relies on stacking many layers to increase field size
Scalability: Efficient with large input sizes	
Feature sharing: Weight sharing across the input	

Table 1: Advantages and Limitations of Convolutional Layers

3. Why do we use spatial pooling?

(a) **Dimensionality Reduction:** Pooling reduces the spatial dimensions of feature maps (i.e., height and width) while preserving important information. This leads to:

- Lower computational cost
- Reduced memory usage

(b) **Translation Invariance:** Pooling helps make the model more robust to small translations or shifts in the input.

(c) **Maintaining Important Features:** Pooling allows the model to focus on the most salient features of the input. For example:

- **Max pooling** selects the maximum value from a region, highlighting the strongest or most prominent activation.
- **Average pooling** computes the average value in a region, which can help smooth out features and reduce noise, although it is less commonly used.

(d) **Control Overfitting:** By reducing the size of feature maps, pooling also reduces the number of parameters in the model, which can help prevent overfitting.

(e) **Hierarchical Feature Learning:** Pooling enables CNNs to learn hierarchical representations of data. In the lower layers, the network focuses on small, detailed features (such as edges or textures in an image). As pooling progressively reduces the size of feature maps, the network can learn more abstract, global representations of the input (such as shapes, objects, or patterns) in the higher layers.

4. Suppose we try to compute the output of a classical convolutional network for an input image larger than the initially planned size (e.g., 224×224). Can we (without modifying the image) use all or part of the layers of the network on this image?

If we attempt to process an image larger than the originally planned size (e.g., 224×224) using a classical convolutional network, such as those designed for ImageNet, we can apply

the convolutional layers without modification. However, we cannot use the fully connected layers as they require a fixed-size input, which depends on the specific input dimensions the network was originally designed for. Since the fully connected layers expect a predefined input shape, attempting to use them with a larger image would cause a mismatch in dimensions. Therefore, while the convolutional layers can handle variable input sizes, the fully connected layers cannot.

5. Show that we can analyze fully-connected layers as particular convolutions.

A fully-connected layer can be interpreted as a convolutional layer with the following specific properties:

- **Kernel Size:** The kernel size is the same as the spatial size of the input. For example, if the input is $H \times W \times C$, then the convolutional filter is $H \times W \times C$, covering the entire input.
- **Stride:** The stride is set to 1, meaning the filter moves by 1 unit across the input.
- **No Padding:** There is no zero-padding applied to the input, so the output is a single value (or multiple values, depending on the number of filters).
- **Number of Filters:** The number of filters corresponds to the number of output neurons in the fully-connected layer.

(a) **Conversion Process**

Let's say we have an input image of size $H \times W \times C$ and a fully-connected layer that outputs a vector of size N_{out}.

- **Input as a "Flattened" Image:** A fully-connected layer treats the input as a single 1D vector. However, we can interpret this 1D input as a "flattened" image, where the spatial size of the input is effectively reduced to $1 \times 1 \times (H \times W \times C)$.
- **Set Filter to Cover the Entire Input:** In a convolutional layer, the filter can be set to have a kernel size of $H \times W$ (i.e., it covers the entire spatial size of the input) and depth C (matching the number of input channels).
- **Convolutional Output:** The output of this convolutional operation will be a single value for each filter because the filter covers the entire input. This is exactly what a fully-connected layer does — it computes a single value (the neuron output) for each set of weights (the filter in this case).
- **Multiple Filters for Multiple Outputs:** In a fully-connected layer, we have multiple neurons (outputs). In the equivalent convolutional layer, we would have multiple filters, each producing a single output value (neuron activation). Thus, a fully-connected layer with N_{out} neurons is equivalent to a convolutional layer with N_{out} filters, each filter being the size of the entire input.

(b) **Mathematical Equivalence**

To formalize this equivalence, let's look at how the output of a fully-connected layer and a convolutional layer would be computed for the same input:

- **Fully-Connected Layer:** For each neuron i in the fully-connected layer:

$$y_i = \sum_{j=1}^{N_{\text{in}}} W_{ij} \cdot x_j + b_i$$

Here, W_{ij} is the weight matrix, x_j are the inputs, and b_i is the bias.

- **Convolutional Layer:** For each filter i in the convolutional layer with a filter size that matches the input size (i.e., $H \times W \times C$):

$$y^{(i)} = \sum_{m,n,c} W(m,n,c) \cdot x(m,n,c) + b$$

Here, $W(m,n,c)$ is the convolutional kernel, and $x(m,n,c)$ are the input values at position (m,n) in channel c.

- **Conclusion:** Since the filter covers the entire input (like a fully-connected layer connecting every input to every output), the two operations are mathematically identical:

6. Suppose that we replace fully-connected layers by their equivalent in convolutions. Answer question 4 again. If we can calculate the output, what is its shape and interest?

In this case, we cannot directly use all layers of the network as is. The main issue lies in the softmax layer, which is typically designed to process a specific output shape corresponding to the original input size. Even if we replace the fully connected layers with their equivalent convolutional layers (using a kernel that matches the original fully connected layer's input size), increasing the input size will cause the output to change.

For example, if the input image size doubles, the output would no longer be a flattened vector of size $1\times1\times1000$ (as expected for classification), but instead a tensor of size $2\times2\times1000$. This larger output cannot be directly fed into the softmax layer for classification, as it requires a single vector representing class scores.

When replacing fully connected layers with convolutional layers, the final output will typically be a feature map rather than a single vector. For example, this feature map might have dimensions $H_{out} \times W_{out} \times C_{out}$, where H_{out} and W_{out} depend on the input size, and C_{out} is the number of filters or channels. To handle this, there are two common options:

(a) **Global Pooling:** Apply global average pooling or global max pooling to the final feature map. This reduces the feature map to a single value per channel, regardless of input size. For instance, if the final feature map is 16x16x1000, global pooling will reduce it to a 1x1x1000 tensor, which can then be used for classification. Networks like ResNet and Inception often use global pooling for this purpose.

(b) **Sliding Window Predictions:** Instead of producing a single output, the network can make predictions at multiple spatial locations in the final feature map. This approach is useful for tasks such as semantic segmentation or object detection, where predictions are required for each pixel or region of the image. In this case, the larger output tensor could be interpreted as multiple predictions across the image.

Thus, although the fully connected layers can be replaced by convolutions, handling larger input sizes requires additional techniques since we typically need to produce a fixed-length output. Techniques like global pooling or adjusting the output processing for tasks other than classification.

7. The *receptive field* of a neuron refers to the set of pixels in the image that influences the output of the neuron. What are the sizes of the receptive fields of

the neurons in the first and second convolutional layers? Can you imagine what happens in the deeper layers? How would you interpret this?

(a) **Sizes of the Receptive Fields**

To calculate the sizes of the receptive fields, we use the following formula:

Receptive field size of layer L = Receptive field size of previous layer + $(k-1) \times$ stride

Where:

- k is the kernel size,
- stride is the stride of the convolution.
- If the first layer has a 3×3 kernel and stride 1, the receptive field is 3×3.
- The second layer (also with a 3×3 kernel and stride 1) will have a receptive field of:

$$3 + (3-1) \times 1 = 5 \times 5$$

So, neurons in the second layer are affected by a 5×5 patch of the input image.

(b) **What Happens in Deeper Layers?**

- As you move deeper into the network, the receptive field grows larger with each successive layer.
- Neurons in deeper layers are influenced by progressively larger areas of the input image, eventually covering the entire image.

(c) **Interpretation of the Receptive Field in Deeper Layers:**

The increasing receptive field in deeper layers reflects the hierarchical nature of feature extraction in convolutional networks:

- **Shallow layers:** Focus on small, fine details (like edges and textures) because their receptive field is small, capturing local patterns.
- **Deep layers:** Focus on larger and more abstract features because their neurons aggregate information from larger parts of the image. For example, a neuron in a deeper layer might "recognize" an entire object, such as a face, by integrating information from previous layers that detected eyes, nose, and mouth.

This hierarchical representation is key to how CNNs achieve good performance in tasks like image classification, object detection, and segmentation.

Section 2 - Training from scratch of the model

2.1 Network architecture

8. For convolutions, we want to keep the same spatial dimensions at the output as at the input. What padding and stride values are needed?

To keep the same spatial dimensions at the output as the input :

- Padding = $\frac{k-1}{2}$ where k is the kernel size.
- Stride = 1.

if $k = 5$, then padding = 2 and stride = 1.

9. For max pooling, we want to reduce the spatial dimensions by a factor of 2. What padding and stride values are needed?

To reduce the spatial dimensions by a factor of 2 in max pooling:

- Padding = 0.
- Stride = 2.
- Kernel size = 2.

10. For each layer, indicate the output size and the number of weights to learn. Comment on this distribution.

Layer	**Output Size**	**Number of Parameters**
conv1	$32 \times 32 \times 32$	$32 \times (5 \times 5 \times 3) + 32 = 2,432$
pool1	$16 \times 16 \times 32$	0
conv2	$16 \times 16 \times 64$	$64 \times (5 \times 5 \times 32) + 64 = 51,264$
pool2	$8 \times 8 \times 64$	0
conv3	$8 \times 8 \times 64$	$64 \times (5 \times 5 \times 64) + 64 = 102,464$
pool3	$4 \times 4 \times 64$	0
fc4	$1,000$	$1,000 \times 1,024 + 1,000 = 1,025,000$
fc5	10	$10 \times 1,000 + 10 = 10,010$
Total		**1,191,170**

The vast majority of the parameters are concentrated in the fully-connected layers. Specifically, the *fc4* layer (with 1,025,000 parameters) dominates the parameter count, representing almost 86% of the total parameters. This is typical for convolutional neural networks (CNNs), where the convolutional layers extract features with relatively few parameters, and the fully-connected layers use most of the parameters to combine those features for final predictions.

Convolutional layers, despite having fewer parameters, perform most of the feature extraction and processing. These layers handle local spatial patterns using shared weights, which reduces the parameter count compared to fully-connected layers.

Pooling layers don't introduce any learnable parameters, but they play a crucial role in downsampling the feature maps, reducing the spatial size while preserving important features.

This demonstrates the efficiency of convolutional layers in terms of parameter usage compared to fully-connected layers, while still enabling the network to extract complex features from the input data.

11. What is the total number of weights to learn? Compare that to the number of examples. In summary:

- Total parameters to learn: 1,191,170
- Number of training examples: 50,000
- Ratio of parameters to training examples: 23.8 parameters per example
- The number of parameters is significantly higher than the number of training examples, which could lead to overfitting if not properly regularized.

12. Compare the number of parameters to learn with that of the Bag of Words (BoW) and Support Vector Machine (SVM) approach.

The size of the visual vocabulary (denoted as K) in the Bag of Words (BoW) is a key factor in determining the number of parameters. The BoW model typically involves the following parameters

For multiclass classification with C classes:

$$\text{Parameters} = K \times C + C$$

Where:

- K is the number of visual words (BoW dimension),
- C is the number of output classes (in this case, 10 for the classification task).

Summary

Approach	Total Parameters	Training Examples
CNN	1,191,170	50,000 (training)
BoW + SVM (K=1000)	$\sim$ 10,010	50,000 (training)

The CNN has 1.19 million parameters, while a BoW + SVM model with 1000 visual words has around 10,000 parameters, making the CNN approximately 100 times more parameter-heavy.

Despite having far fewer parameters, BoW + SVM lacks the hierarchical feature extraction capability of CNNs, which makes CNNs more powerful for complex image classification tasks.

2.2 Network learning

13. Read and test the code provided. You can start the training with this command:
`main(batch_size, lr, epochs, cuda = True)`

14. In the provided code, what is the major difference between the way to calculate loss and accuracy in training and in testing (other than the difference in data)?

In training: The optimizer is provided (`optimizer` is not `None`), leading to backpropagation and weight updates.
In testing: The optimizer is `None`, which means no backpropagation or updates, and the model is set to evaluation mode.
This distinction between forward-only evaluation (test) and forward followed by backpropagation (train) is the critical difference in the way loss and accuracy are calculated.

15. Modify the code to use the CIFAR-10 dataset and implement the requested architecture (the class is `datasets.CIFAR10`). Make sure to run enough epochs for the model to finish converging.

We modify the architecture to use the CIFAR-10 dataset and implement the requested architecture. We also increase the number of epochs to ensure the model converges. The code is provided below:

```
class ConvNet(nn.Module):
"""
This class defines the structure of the neural network
"""

def __init__(self):
    super(ConvNet, self).__init__()
    # We first define the convolution and pooling layers as a features extractor
    self.features = nn.Sequential(
        nn.Conv2d(3, 32, (5, 5), stride=1, padding=2),
        nn.ReLU(),
        nn.MaxPool2d((2, 2), stride=2, padding=0),
        nn.Conv2d(32, 64, (5, 5), stride=1, padding=2),
        nn.ReLU(),
        nn.MaxPool2d((2, 2), stride=2, padding=0),
        nn.Conv2d(64, 64, (5, 5), stride=1, padding=2),
        nn.ReLU(),
        nn.MaxPool2d((2, 2), stride=2, padding=0),
    )
    # We then define fully connected layers as a classifier
    self.classifier = nn.Sequential(
        nn.Linear(1024, 1000),
        nn.ReLU(),
        nn.Linear(1000, 10)
        # Reminder: The softmax is included in the loss, do not put it here
    )

# Method called when we apply the network to an input batch
def forward(self, input):
    bsize = input.size(0) # batch size
    output = self.features(input) # output of the conv layers
    output = output.view(bsize, -1) # we flatten the 2D feature maps into one 1D vector for each input
    output = self.classifier(output) # we compute the output of the fc layers
    return output

def get_dataset(batch_size, cuda=False):
    """
    This function loads the dataset and performs transformations on each
    image (listed in `transform = ...`).
    """
    train_dataset = datasets.CIFAR10(PATH, train=True, download=True,
        transform=transforms.Compose([
            transforms.ToTensor()
        ]))
    val_dataset = datasets.CIFAR10(PATH, train=False, download=True,
        transform=transforms.Compose([
            transforms.ToTensor()
        ]))

    train_loader = torch.utils.data.DataLoader(train_dataset,
                         batch_size=batch_size, shuffle=True, pin_memory=cuda,
                             num_workers=2)
    val_loader = torch.utils.data.DataLoader(val_dataset,
                         batch_size=batch_size, shuffle=False, pin_memory=cuda
                            , num_workers=2)

    return train_loader, val_loader
```

16. What are the effects of the learning rate and batch size?

- **Learning Rate:**
  - A high learning rate can lead to faster convergence but risks overshooting optimal values, causing instability.
  - A low learning rate leads to more stable but slower convergence and may get stuck in local minima.
- **Batch Size:**
  - A small batch size provides noisier updates, leading to potentially better generalization but slower training.
  - A large batch size leads to smoother updates, faster training, but might generalize worse and require more memory.

17. What is the error at the start of the first epoch, in both train and test? How can you interpret this?

At the start of the first epoch:

- **Training:**
  - **Loss:** 2.3016
  - **Prec@1 (Top-1 accuracy):** 8.6%
  - **Prec@5 (Top-5 accuracy):** 57.8%
- **Testing (Evaluation):**
  - **Loss:** 1.8551
  - **Prec@1 (Top-1 accuracy):** 35.2%
  - **Prec@5 (Top-5 accuracy):** 86.7%

Interpretation

(a) **Training Phase:**

- The high initial loss (2.3016) and low top-1 accuracy (8.6%) indicate that the model is just starting to learn and has not yet been optimized for the task. This is typical at the beginning of training, especially when the model parameters are initialized randomly.
- The top-5 accuracy (57.8%) is considerably higher than the top-1 accuracy, meaning the correct class is often among the top five predictions, which is promising early on.

(b) **Testing Phase:**

- The evaluation loss is lower (1.8551) than the initial training loss, and the top-1 accuracy (35.2%) is substantially better than in training. This might suggest that the model is already somewhat capable of generalizing, likely because the training loss at this point includes noise from random initialization and a shuffled batch, while the test set reflects the model's capacity to recognize patterns without further weight updates.

- The test top-5 accuracy is quite high (86.7%), indicating that the model is fairly good at ranking the correct class within the top 5, even from the start.

18. Interpret the results. What's wrong? What is this phenomenon?

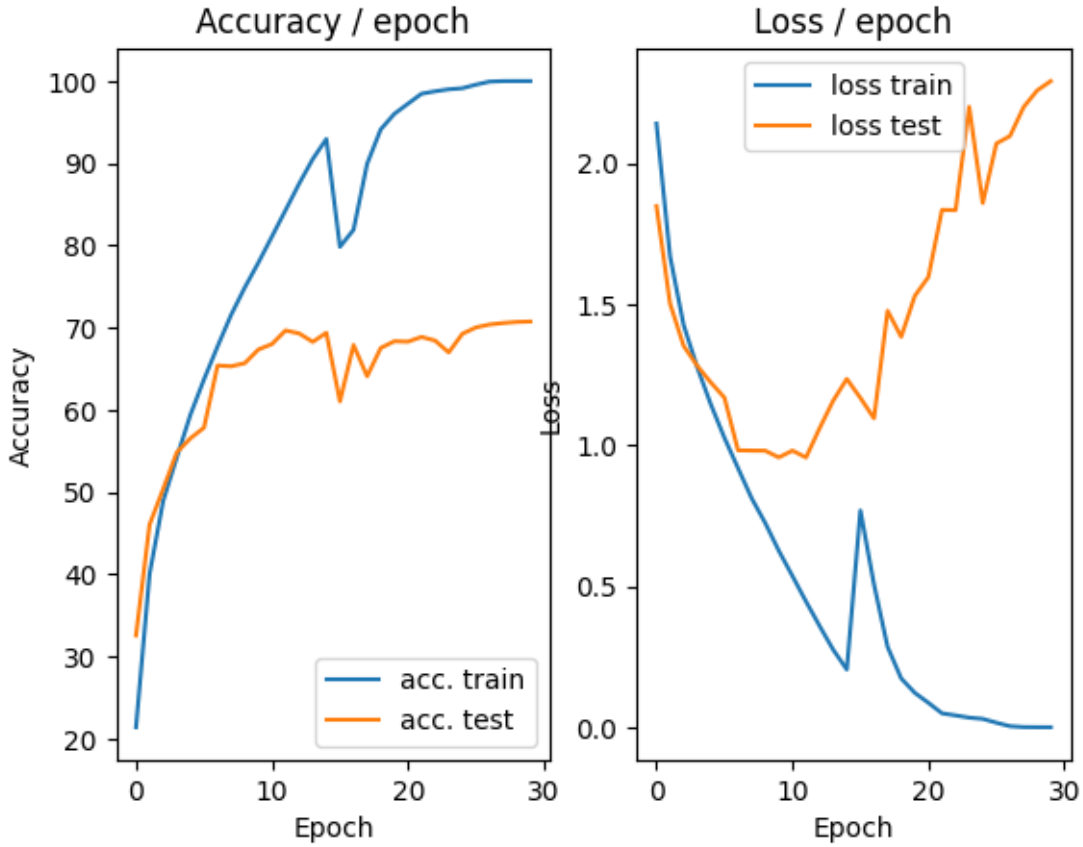

Figure 1: Training and Testing Accuracy and Loss Curves

Accuracy Plot (Left):

(a) **Training Accuracy:** Increases steadily, reaching nearly 100% by the end of the training epochs. This indicates that the model is able to learn and fit the training data very well.

(b) **Testing Accuracy:** Improves initially but plateaus around 70–75%, it's overfitting. The model performs well on the training data but doesn't generalize as effectively to the test data.

Loss Plot (Right):

(a) **Training Loss:** Decreases consistently, approaching zero. This again indicates that the model is fitting the training data effectively.

(b) **Testing Loss:** Decreases initially, then starts to rise, which is a classic sign of overfitting. While the model achieves good training performance, it begins to struggle with the test data, as seen by the increasing loss despite continued training.

Metrics Summary:

(a) **Training Accuracy (Prec@1):** 100% and low training loss (0.0002) confirm the

model's strong fit to the training set.

(b) **Test Accuracy (Prec@1):** 72.82% and higher test loss (2.1596) show the model's limited generalization to new data.

As we can see from the Figure 1, at the start of the training loop the training and testing accuracy increase steadily together with the decrease in loss. However, after a few epochs, the training accuracy continues to increase even reaching 100% while the testing accuracy decreases and plateaus. This indicates that the model is overfitting to the training data, performing well on the training set but failing to generalize to unseen data.

To address this issue, we can use regularization techniques such as dropout, L2 regularization, or batch normalization to prevent overfitting.

Section 3 - Results improvements

For this part, in the end of session report, indicate the methods you have successfully implemented and for each one explain in one sentence its principle and why it improves learning.

3.1 Standardization of examples

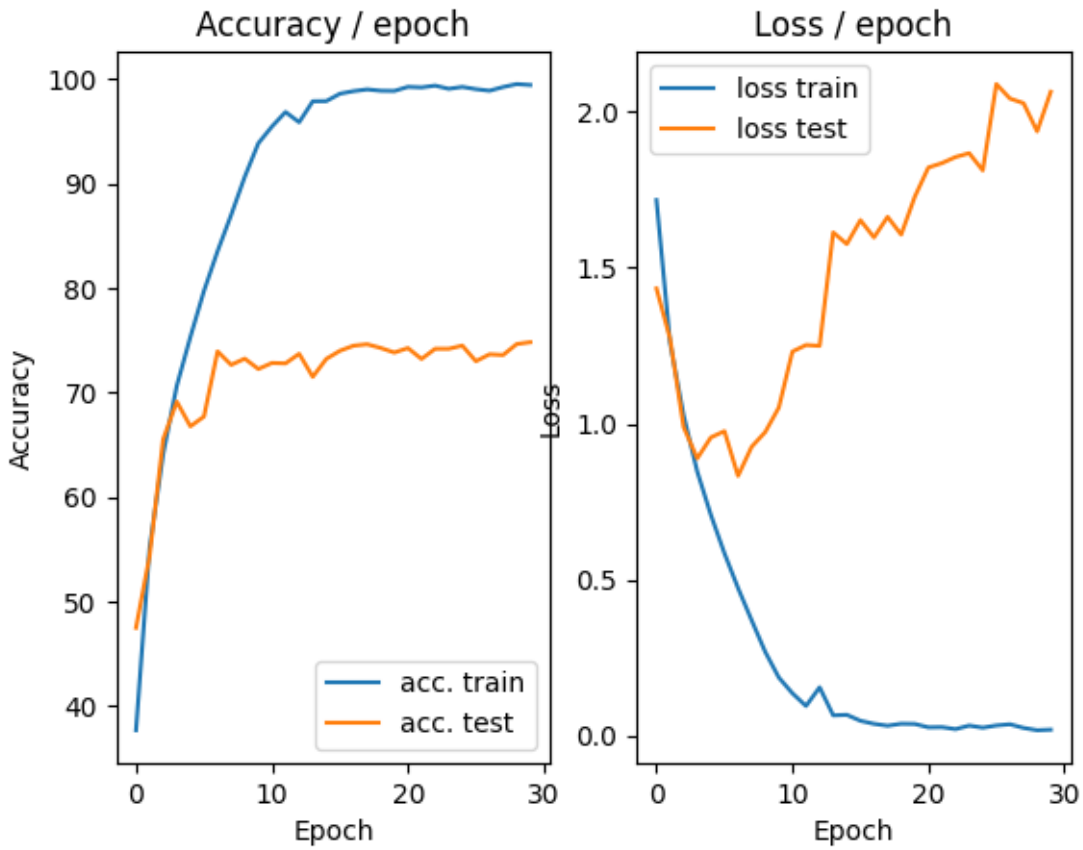

Figure 2: Training and Testing Accuracy and Loss Curves after Normalization

19. Describe your experimental results.

The experimental results are summarized in Figure 2, which shows the evolution of accuracy and loss for both the training and validation sets over 30 epochs.

Accuracy Plot (Left)

(a) **Training Accuracy:** Continues to reach nearly 100%, indicating that the model can still fully learn the training data.

(b) **Test Accuracy:** Improves slightly over the baseline, reaching around 76.43% (compared to the baseline's ~72.82%). This increase in test accuracy suggests that normalization has helped improve generalization, though there is still some gap between training and test performance.

Loss Plot (Right)

(a) **Training Loss:** Decreases and approaches zero, which is consistent with the previous experiment.

(b) **Test Loss:** Appears to be somewhat more stable compared to the baseline, though it does still fluctuate and shows signs of overfitting as it rises over epochs. However, the final average test loss is slightly lower (1.9878 vs. 2.1596), reflecting an improvement.

Metrics Summary

(a) **Test Accuracy (Prec@1):** 76.43% is a notable improvement from the previous baseline accuracy, suggesting normalization helped the model perform better on unseen data.

(b) **Test Loss:** While still relatively high, the decrease compared to the baseline indicates that normalization is beneficial for the model's generalization.

20. Why do we only calculate the average image on the training examples and normalize the validation examples with the same image?

The mean and standard deviation are calculated based on the training data to preserve the statistical integrity of the validation process. This approach is crucial because:

- **Statistical Independence:**
  - Validation set must remain completely independent from training
  - Using validation statistics would leak information
  - Prevents any form of data leakage between sets
- **Distribution Consistency:**
  - Model learns features based on training distribution
  - Validation should follow same distribution
  - Ensures fair evaluation of model generalization
- **Real-world Application:**
  - Mimics deployment scenarios where statistics are pre-computed
  - New data must be normalized using training statistics
  - Maintains consistency in production environments

21. Bonus: There are other normalization schemes that can be more efficient, such as ZCA normalization. Try other methods, explain the differences, and compare them to the one requested.

ZCA normalization (Zero-phase Component Analysis) is a method that reduces correlations between features while preserving the spatial structure of images, thereby improving model generalization. In comparison, classical normalization by the maximum simply scales the data to the same range without addressing feature correlations. The results show that ZCA normalization doesn't improve generalization, as evidenced by lower test accuracy.

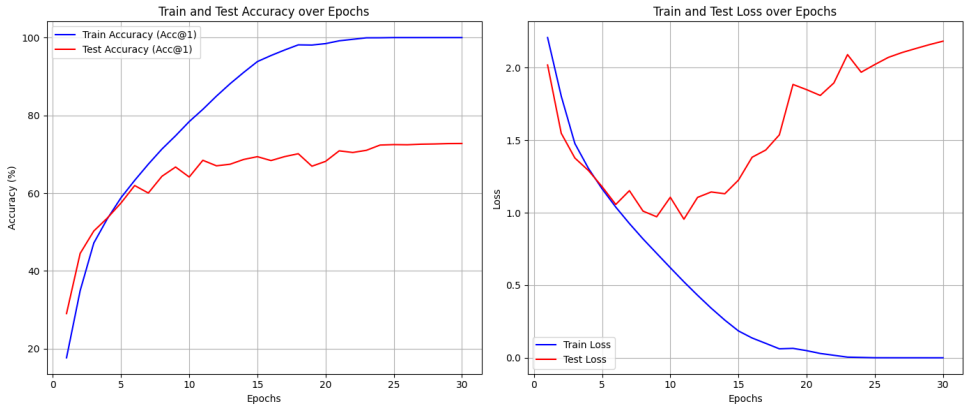

Figure 3: Train and Test Accuracy/Loss for ZCA Normalization

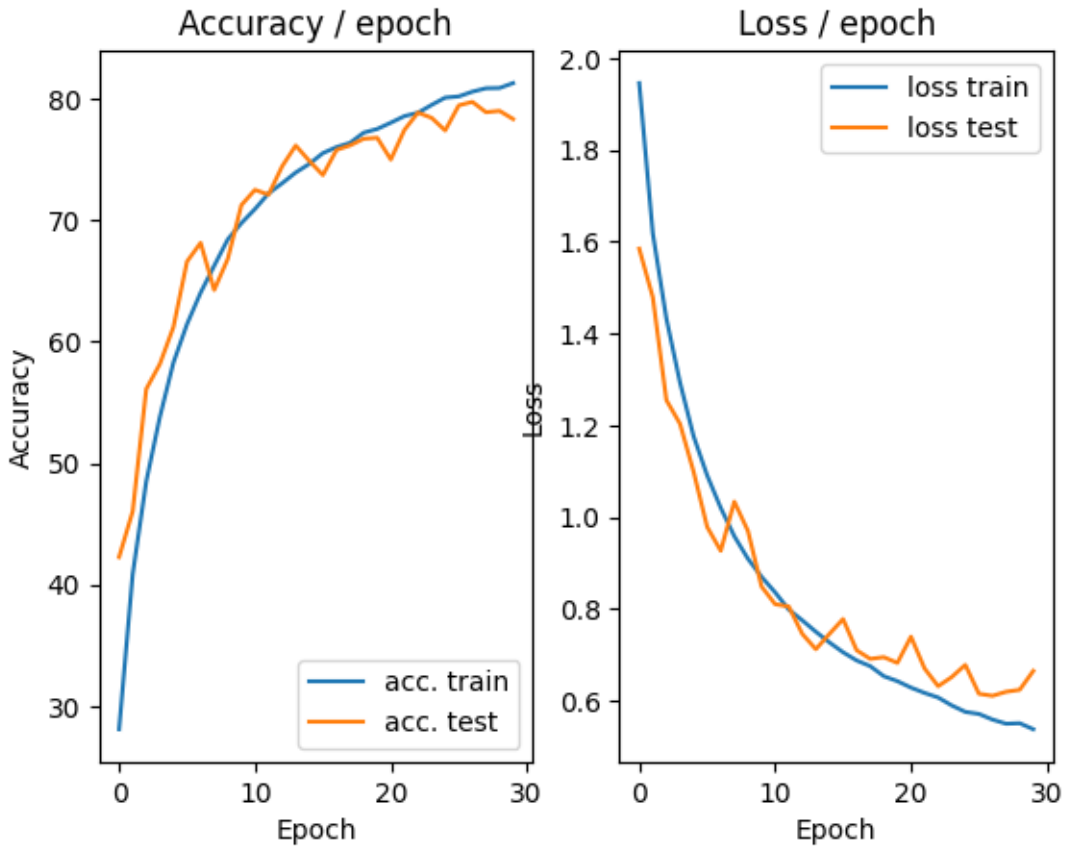

Figure 4: Training and Testing Accuracy and Loss Curves after Normalization + Data Augmentation

3.2 Increase in the number of training examples by data increase

22. Describe your experimental results and compare them to previous results.

The experimental results are summarized in Figure 4, which shows the evolution of accuracy and loss for both the training and validation sets over 30 epochs.

Accuracy Plot (Left)

(a) **Training Accuracy:** Peaks at around 81.51%, which is slightly lower than the baseline but indicates less overfitting since the model isn't perfectly fitting the training data.

(b) **Test Accuracy:** Improves, reaching an average of 77.60% in the final epoch. This is an increase over the baseline (72.82%) and the normalization-only experiment (76.43%), suggesting that data augmentation has indeed helped improve generalization.

Loss Plot (Right)

(a) **Training Loss:** Decreases steadily and ends at a reasonable level (0.5359), not approaching zero as quickly as in previous experiments. This stability in training loss suggests that data augmentation has encouraged the model to generalize rather than memorize.

(b) **Test Loss:** Reduces significantly compared to previous experiments, ending around 0.6845. The lower final test loss compared to the baseline and normalized-only experiments (which had test losses over 1.9) demonstrates that data augmentation has mitigated overfitting.

Metrics Summary

(a) **Test Accuracy (Prec@1):** 77.60% represents an improvement over previous experiments, confirming that data augmentation has enhanced the model's performance on unseen data.

(b) **Test Loss:** The substantial reduction in test loss (0.6845 vs. 2.1596 in baseline) is a strong indicator of better generalization.

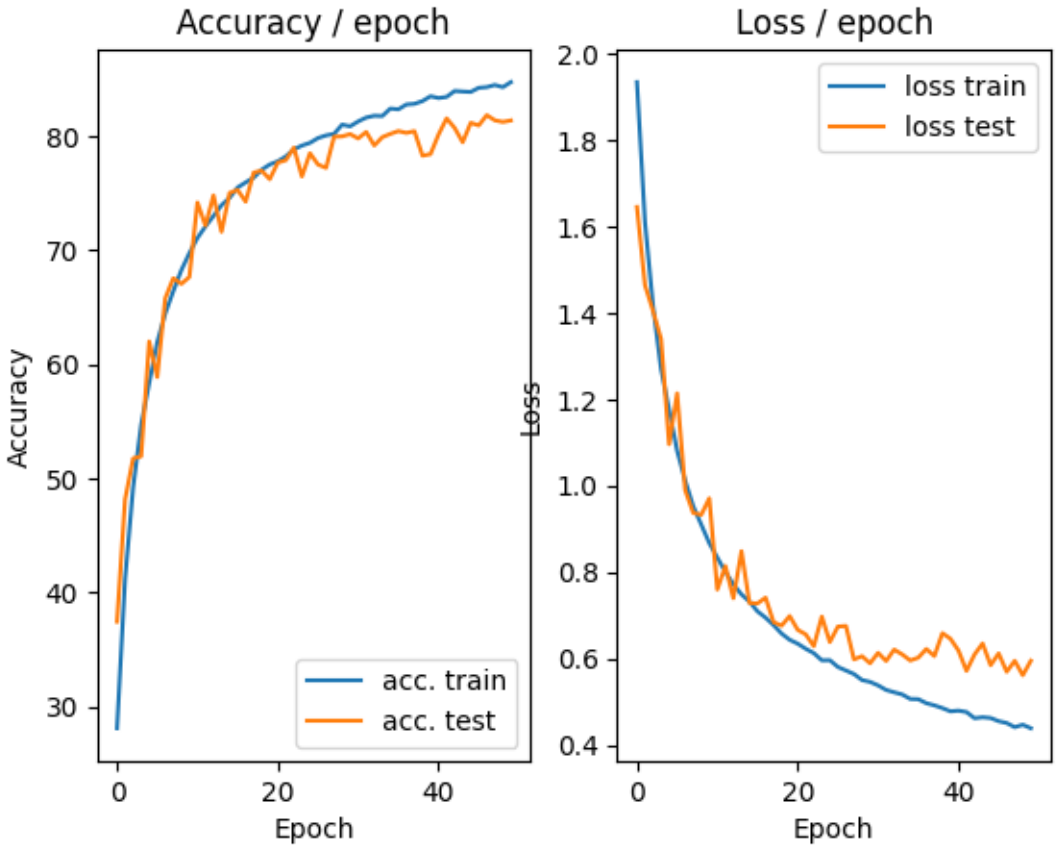

Figure 5: Training and Testing Accuracy and Loss Curves after Normalization + Data Augmentation for 50 epochs

As we can see from Figure 5, training for additional epochs allows the model to achieve higher accuracy on the test set and reduce the test loss further, suggesting that the model still had

capacity to learn and generalize even at 30 epochs. These results suggest that 50 epochs may be closer to the optimal training duration for this model setup on CIFAR-10.

Accuracy Plot (Left)

(a) **Training Accuracy:** Increases to around 84.70%, a slight improvement over the 30-epoch results (81.51%). The model continues to learn effectively with additional training epochs.

(b) **Test Accuracy:** Peaks at approximately 81.34%, which is an improvement from the 77.60% observed with 30 epochs. This indicates that additional epochs have helped the model generalize better.

Loss Plot (Right)

(a) **Training Loss:** Further decreases to around 0.4385, showing that the model continues to fit the training data with more epochs.

(b) **Test Loss:** Reduces to 0.5946, which is a notable improvement compared to the 0.6845 after 30 epochs. The test loss curve is also more stable in the latter epochs, indicating that overfitting remains minimal.

Metrics Summary

(a) **Test Accuracy (Prec@1):** 81.34%, an improvement over previous tests, further demonstrating the benefit of extended training.

(b) **Test Loss:** At 0.5946, the test loss is the lowest observed so far, reflecting improved generalization.

23. Does this horizontal symmetry approach seem usable for all types of images? In what cases might it be usable or not?

The effectiveness of horizontal flipping as an augmentation technique varies significantly depending on the nature of the input images. Our analysis reveals a clear dichotomy between favorable and problematic application cases, requiring careful consideration of image semantics and content.

3.2.1 Effective Applications

Horizontal symmetry proves particularly relevant for:

- Natural objects exhibiting intrinsic bilateral symmetry
- Generic object categories (e.g., vehicles, animals, bottles, boxes)
- Environmental scenes
- Objects lacking critical directional features

3.2.2 Problematic Applications

However, this technique presents significant limitations for:

- Numerical digits (complete loss of semantic meaning)
- Textual content (becomes unreadable after transformation)
- Directional symbols (loss of intended meaning)

- Anatomical images (incorrect orientation)
- Images where semantic integrity is crucial

24. What limits do you see in this type of data augmentation by transformation of the dataset?

Our augmentation approach presents several significant limitations that warrant detailed examination across multiple dimensions:

3.2.3 Resolution and Information Challenges

- Dimensional reduction (32×32 → 28×28) causes non-negligible information loss
- Cropping process may eliminate essential image features
- Empty padding adds no valuable information
- Limited effectiveness of basic padding strategies

3.2.4 Transformation Constraints

- Restriction to basic geometric manipulations
- Inability to introduce new semantic content
- Risk of creating artificial patterns
- Limited variability compared to natural image diversity
- Potential generation of semantically inconsistent results

3.2.5 Implementation Challenges

- Significant increase in required computational resources
- Need for consistent processing during the testing phase
- Increased complexity in data pipeline management
- Search for optimal balance between augmentation and preservation

25. Bonus: Other data augmentation methods are possible. Research which ones exist and test some.

There are a variety of data augmentation techniques that can be used to improve model generalization and robustness. Some common methods include:

(a) **ColorJitter (brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1)**

Description: Randomly adjusts the brightness, contrast, saturation, and hue of the image.

Benefit: This augmentation increases robustness to varying lighting conditions. It helps the model handle differences such as day vs. night or indoor vs. outdoor lighting.

(b) **RandomRotation (15 degrees)**

Description: Rotates the image by a random degree within the range of ±15 degrees.

Benefit: Improves the model's ability to recognize objects at different orientations. This helps with CIFAR-10 images, where objects may not always appear upright.

(c) **RandomAffine (translate=(0.1, 0.1), scale=(0.9, 1.1))**

Description: Applies affine transformations with random translation (up to 10% of the width/height) and scaling (from 0.9 to 1.1).

Benefit: Increases model resilience to object positioning and size variations. It helps the model detect objects even if they are slightly shifted or resized.

(d) **RandomPerspective (distortion_scale=0.2, p=0.5)**

Description: Introduces a random perspective distortion with a distortion scale of 0.2, with a 50% probability of application.

Benefit: Enhances adaptability to perspective changes, simulating different viewpoints. This is useful for recognizing objects from varied camera angles.

With the new, more complex data augmentation strategy, here's how the model's performance compares to the previous experiment :

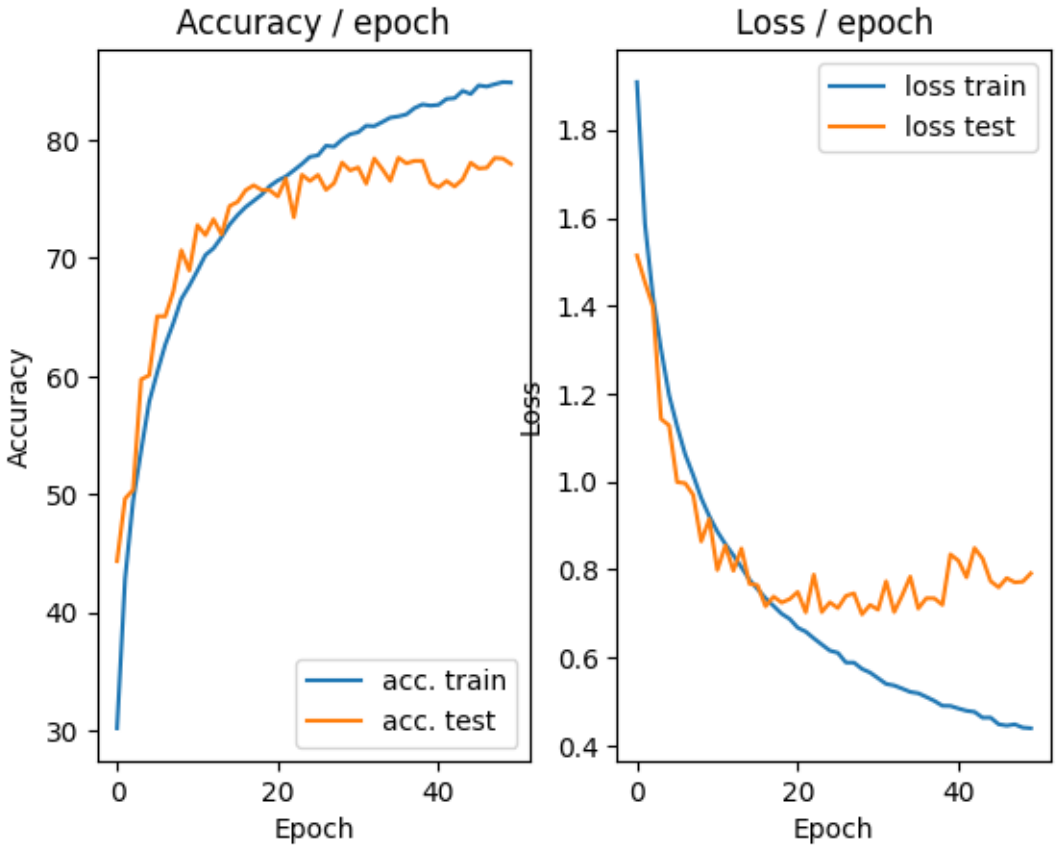

Figure 6: Training and Testing Accuracy and Loss Curves after Normalization + ColorJitter + RandomRotation + RandomAffine + RandomPerspective for 50 epochs

Accuracy Plot (Left)

(a) **Training Accuracy:** Achieves around 84.84%, which is in line with previous experiments and indicates the model has successfully fit the training data despite the added data complexity.

(b) **Test Accuracy:** Peaks at approximately 77.95%, which is slightly lower than previous setups but consistent with the complex data augmentation strategy. This lower test accuracy might suggest that the model needs further adjustments.

Loss Plot (Right)

(a) **Training Loss:** Ends at around 0.4396, a good indication that the model has minimized training loss while still adapting to the transformations.

(b) **Test Loss:** Ends at around 0.7916, which is higher than previous test loss values. This increase in test loss, combined with fluctuating accuracy in later epochs, could indicate that the complex transformations make the model's learning process more challenging, potentially requiring reduced augmentation intensity.

Metrics Summary

(a) **Test Accuracy (Prec@1):** 77.95%, slightly lower than simpler augmentation setups.

(b) **Test Loss:** At 0.7916, higher than setups with simpler augmentations, suggesting that the model may be slightly over-regularized or struggling to generalize with the additional transformations.

3.3 Variants on the optimization algorithm

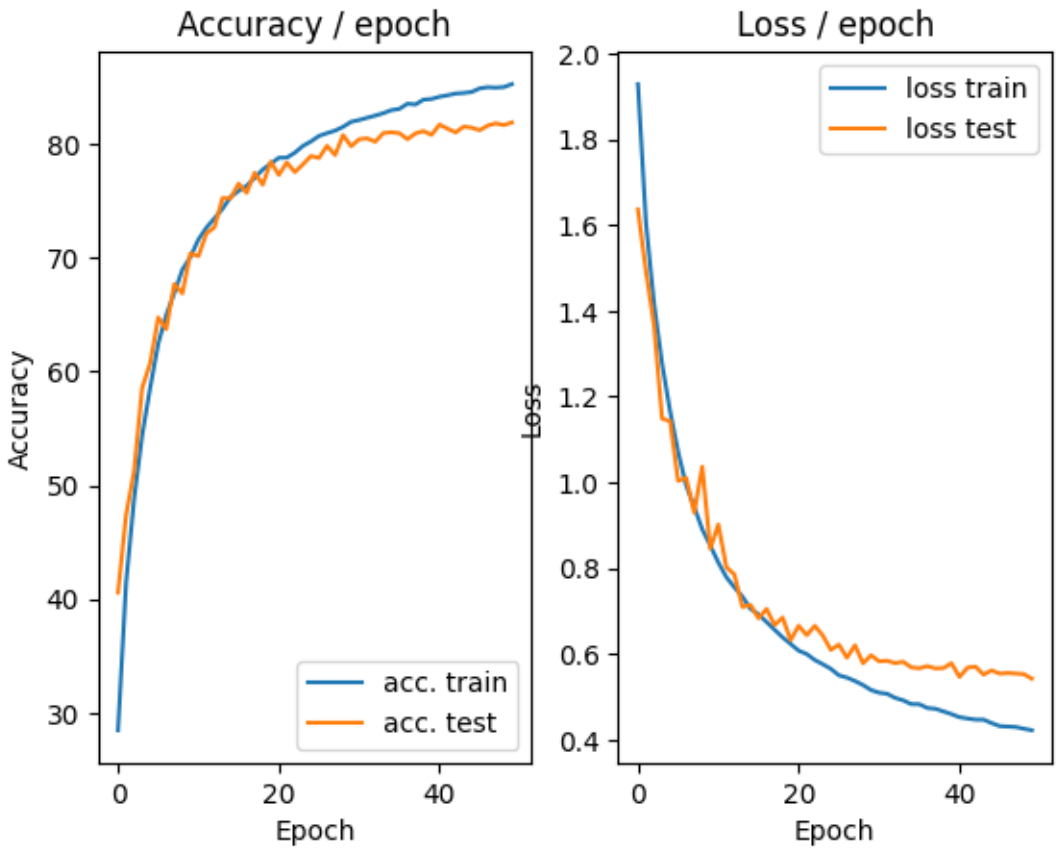

Figure 7: Training and Testing Accuracy and Loss Curves after Normalization + Data Augmentation + LR Scheduler for 50 epochs

26. Describe your experimental results and compare them to previous results, including learning stability.

The experimental results are summarized in Figure 7, which shows the evolution of accuracy and loss for both the training and validation sets over 30 epochs. **Accuracy Plot (Left)**

(a) **Training Accuracy:** Reaches around 85.26%, indicating that the model continues to learn effectively. This is a slight improvement over previous setups, suggesting the scheduler has helped optimize training.

(b) **Test Accuracy:** Improves to approximately 81.88%, which is the highest test accuracy achieved across all experiments. This indicates that the model generalizes better with the learning rate scheduler.

Loss Plot (Right)

(a) **Training Loss:** Reduces steadily to around 0.4226, which is a marginal improvement over the previous training loss without a scheduler (0.4385).

(b) **Test Loss:** Ends at 0.5428, which is a notable reduction compared to previous setups (0.5946 without the scheduler). The test loss curve is also smoother, indicating less fluctuation and better stability due to the decaying learning rate.

Metrics Summary

(a) **Test Accuracy (Prec@1):** 81.88% is the highest observed, confirming that the model benefits from the learning rate adjustments.

(b) **Test Loss:** The reduction to 0.5428 reflects improved convergence and generalization.

27. Why does this method improve learning?

The `ExponentialLR` scheduler enhances learning through multiple mechanisms:

- **Initial Exploration**: A higher learning rate at the start enables rapid exploration of the parameter space, helping the model avoid shallow local minima.
- **Gradual Refinement**: With each epoch, a 5% learning rate reduction (gamma = 0.95) allows finer updates, facilitating precise convergence toward optimal values without premature over-adjustments.
- **Stability in Final Phases**: Lower learning rates in later epochs reduce fluctuations, resulting in a stable convergence process.

These effects combine to improve generalization and support steady convergence, preventing disruptive updates in the final epochs.

28. Bonus: Many other variants of Stochastic Gradient Descent (SGD) exist, along with numerous learning rate scheduling strategies. Which ones? Test some of them.

Overview

Each combination is evaluated for its impact on model performance. The `OptimizerTest` class structures each test, and the function `create_optimizer_tests` initializes a list of configurations, including SGD variants and Adam.

Schedulers Tested

The primary schedulers tested include:

- **ExponentialLR**: Applies exponential decay to the learning rate.
- **CosineAnnealingLR**: Uses cosine annealing for cyclic learning rate adjustment.
- **OneCycleLR**: Increases then decreases the learning rate within a single cycle.
- **ReduceLROnPlateau**: Lowers the rate when the loss plateaus.
- **StepLR**: Reduces the learning rate in defined steps.

Testing Details

Each configuration is tested over 15 epochs, tracking loss, accuracy, and learning rate dynamics.

Performance Observations

The following sections summarize observations from testing various optimizers and schedulers.

Loss Evolution

Loss graphs reveal distinct patterns depending on the optimizer and scheduler:

- **Basic SGD** and **SGD with Momentum** show gradual, steady loss reduction.

- **Adam** converges quickly initially but stabilizes at a relatively higher loss level.
- **ExponentialLR** and **CosineAnnealingLR** combinations demonstrate improved loss reduction, indicative of efficient convergence handling.

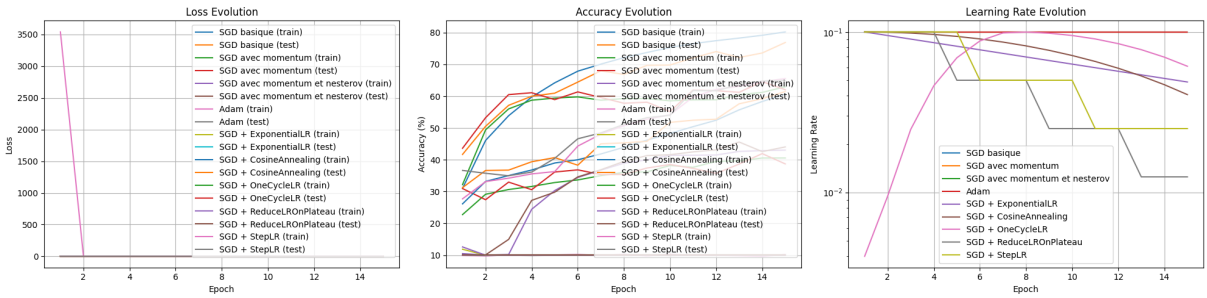

Figure 8: Accuracy (left) and loss (right) over epochs

Accuracy Evolution

Accuracy trends show that combinations like **SGD + ExponentialLR** and **SGD with Momentum** achieve higher accuracy, reflecting stronger generalization capabilities.

Learning Rate Evolution

The learning rate patterns validate each scheduler's design:

- **ExponentialLR**: Applies continuous decay.
- **OneCycleLR**: Increases initially for exploration, then decreases for stabilization.
- **ReduceLROnPlateau**: Only reduces the rate when necessary, optimizing later convergence.

Conclusion: Optimizer and Scheduler Selection

These experiments show the importance of choosing an appropriate optimizer and scheduler combination to balance generalization, convergence speed, and stability. While adaptive optimizers like Adam and RMSprop yield faster initial progress, decaying schedules like ExponentialLR and CosineAnnealingLR provide controlled, gradual convergence that can reduce overfitting risks. **OneCycleLR** combines these strengths, though it requires careful parameter tuning for optimal results.

Ultimately, selecting the right combination depends on training goals, computational limits, and performance trade-offs between speed and accuracy. This analysis supports that **SGD** paired with exponential or cyclic decay schedulers reliably offers a balance between fast convergence and robust generalization for CNN training tasks.

3.4 Regularization of the network by dropout

29. Describe your experimental results and compare them to previous results.

The experimental results are summarized in Figure 9, which shows the evolution of accuracy and loss for both the training and validation sets over 30 epochs.

Accuracy Plot (Left)

(a) **Training Accuracy:** Reaches around 82.34%, slightly lower than the highest value in previous experiments. This is expected with dropout since it restricts the model's ability to memorize the training data.

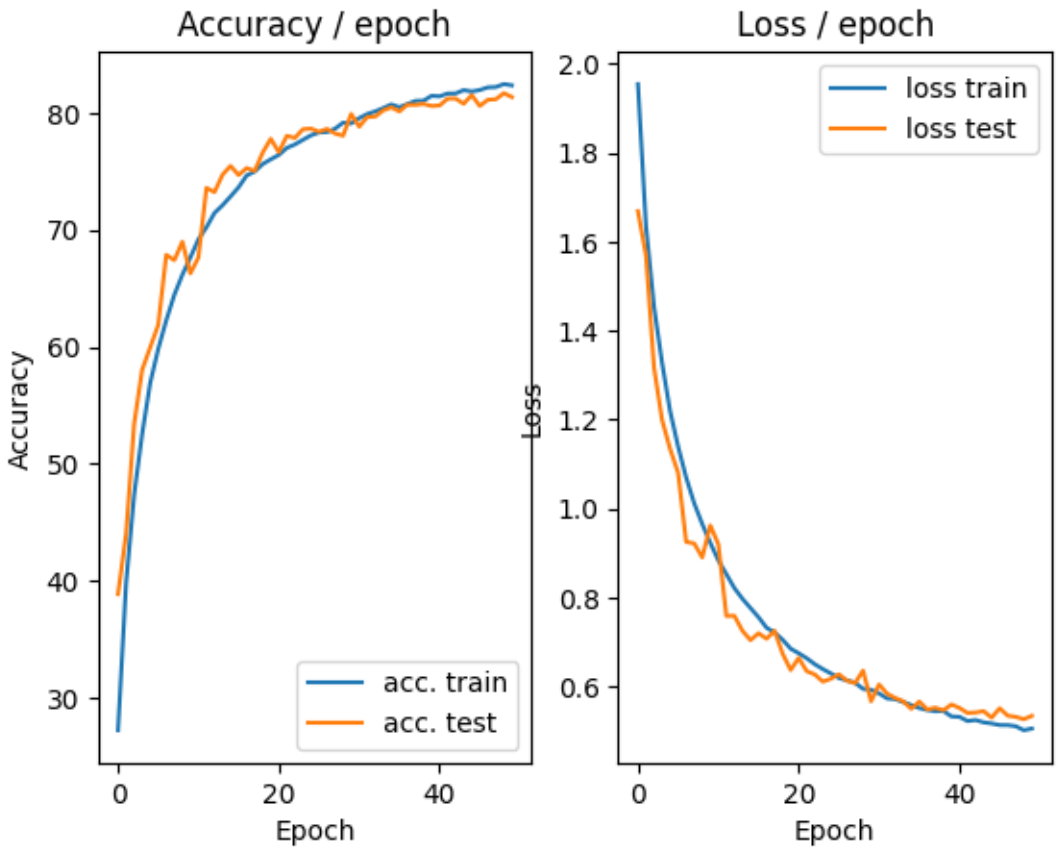

Figure 9: Training and Testing Accuracy and Loss Curves after Normalization + Data Augmentation + LR Scheduler + DropOut for 50 epochs

(b) **Test Accuracy:** Maintains a high level, reaching an average of around 81.34%. Although this is similar to previous runs, the consistency in accuracy suggests better generalization.

Loss Plot (Right)

(a) **Training Loss:** Ends at 0.5058, a bit higher than in prior runs, which is normal due to dropout, as it increases the challenge for the model to fit the training data perfectly.

(b) **Test Loss:** Reduces to around 0.5343, showing a slight improvement and stabilization in test loss. The test loss curve is smooth, and the gap between training and test loss is smaller, indicating reduced overfitting.

Metrics Summary

(a) **Test Accuracy (Prec@1):** Around 81.34%, comparable to previous setups but with reduced overfitting.

(b) **Test Loss:** At 0.5343, a slight improvement from previous tests, reflecting better generalization.

With the increase in epochs to 70, the model demonstrates further improvement in both training and test metrics, stabilizing at a high level of accuracy with minimal overfitting. Here's an analysis of the results:

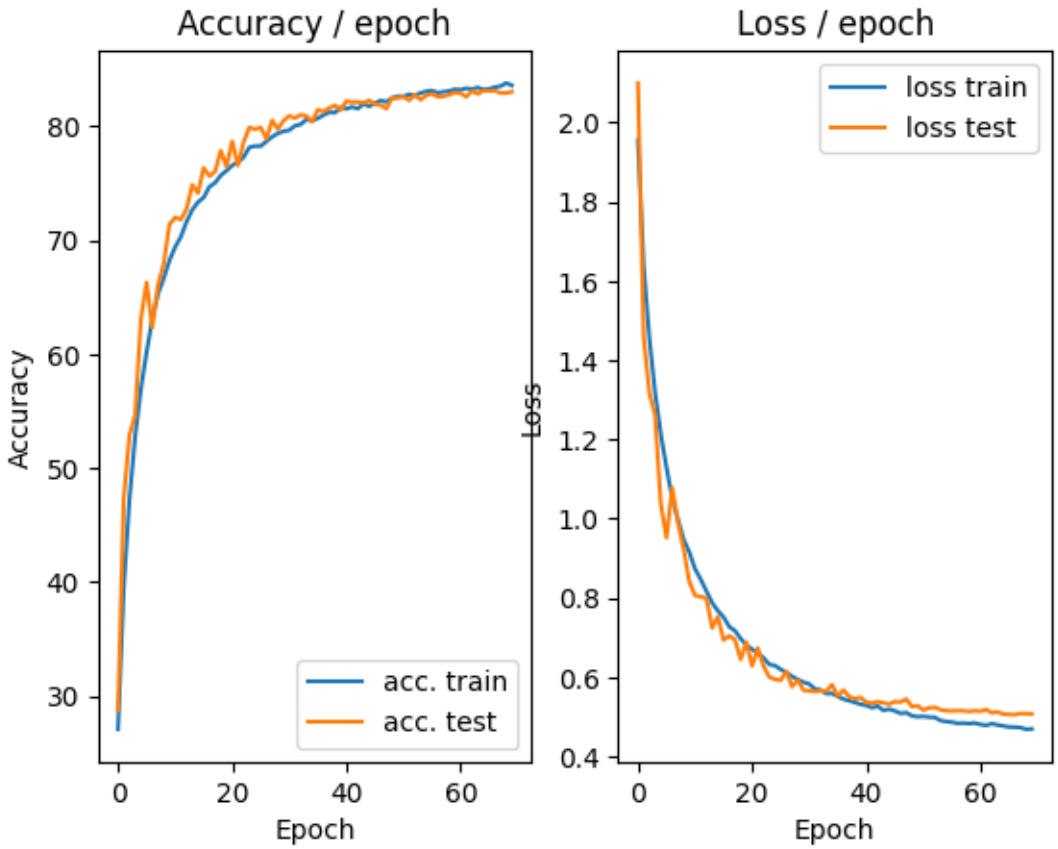

Figure 10: Training and Testing Accuracy and Loss Curves after Normalization + Data Augmentation + LR Scheduler + DropOut for 70 epochs

Accuracy Plot (Left)

(a) **Training Accuracy:** Approaches around 83.60%, suggesting that the model has fully leveraged the additional epochs to fit the training data.

(b) **Test Accuracy:** Reaches an average of 83.03%, marking a slight improvement over previous experiments. The training and test accuracy curves are tightly aligned, indicating no overfitting.

Loss Plot (Right)

(a) **Training Loss:** Settles at around 0.4696, which is low and stable, showing that the model has effectively minimized training loss with additional epochs.

(b) **Test Loss:** Reduces further to 0.5074, the lowest test loss observed in the experiments so far. The test loss curve mirrors the training loss, suggesting that the model generalizes well.

Metrics Summary

(a) **Test Accuracy (Prec@1):** 83.03%, a slight but meaningful improvement, indicating that the model continues to benefit from additional training epochs.

(b) **Test Loss:** At 0.5074, this reduction in test loss reflects enhanced generalization and stability.

30. What is regularization in general?

Regularization is a method aimed at combating overfitting, thereby improving model generalization. It often involves setting certain weights to zero, which effectively removes some features and reduces dimensionality. This is the case with L1 regularization, also known as Lasso. Alternatively, instead of completely nullifying certain weights, other types of regularization aim to minimize them as much as possible, allowing the model to retain these features, albeit minimally. This approach is characteristic of L2 regularization, or Ridge.

31. Research and discuss possible interpretations of the effect of dropout on the behavior of a network using it.

Dropout influences network behavior through several perspectives:

- **Ensemble Learning Perspective:** Dropout creates an ensemble of networks by randomly dropping(e.g with a probability p) units during each forward pass, effectively training multiple sub-networks. During inference, predictions from these sub-networks are averaged, yielding a final output that benefits from ensemble learning.
- **Prevention of Feature Co-adaptation:** By randomly deactivating neurons, dropout forces the network to learn more independent and robust features, reducing reliance on specific activations and discouraging complex co-adaptations among neurons.
- **Noise Robustness:** Dropout introduces noise during training, simulating a regularization effect that makes the model more resilient to input variations. As a result, the learned features are better suited to handle diverse input scenarios.

32. What is the influence of the hyperparameter of the dropout layer?

The choice of dropout rate p has a significant impact on model training:

- A **high dropout rate** (e.g., $p = 0.5$) imposes stronger regularization, slowing convergence but effectively preventing overfitting. However, if set too high, it may lead to underfitting as fewer units remain active.
- Conversely, a **low dropout rate** (e.g., $p = 0.2$) allows faster convergence with less impact on model capacity, though it may not prevent overfitting as effectively.

33. What is the difference in behavior of the dropout layer between training and testing?

During the training phase, dropout randomly deactivates units with a probability p, scaling the remaining activations by $1/(1-p)$. This stochastic behavior creates noise and enforces redundancy. In the testing phase, dropout is disabled, and the full network is used with scaled weights to simulate an ensemble effect, resulting in more deterministic predictions.

3.5 Use of batch normalization

34. Describe your experimental results and compare them to previous results.

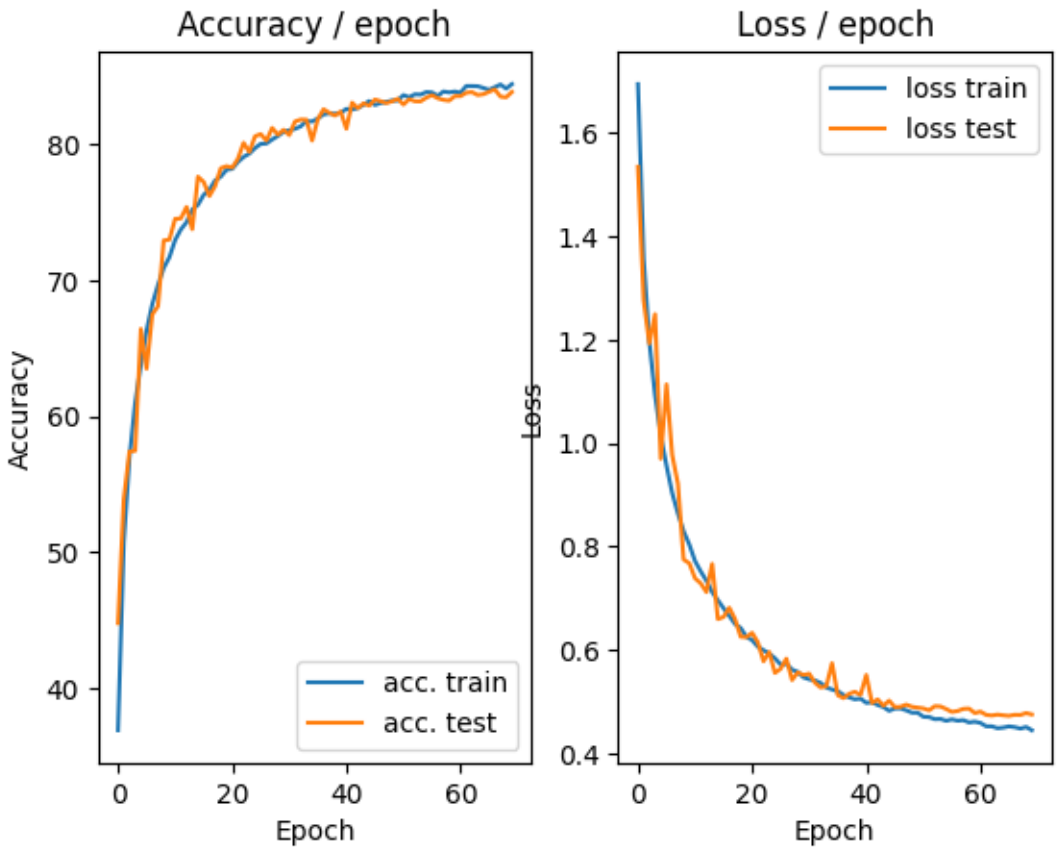

Figure 11: Training and Testing Accuracy and Loss Curves after Normalization + Data Augmentation + LR Scheduler + DropOut + Batch Normalization for 70 epochs

Accuracy Plot (Left)

(a) **Training Accuracy:** Reaches around 84.46%, which is higher than in previous setups without batch normalization. Batch normalization helps stabilize and accelerate training, which appears to improve the model's ability to reach a high training accuracy.

(b) **Test Accuracy:** Peaks at approximately 83.88%, which is slightly higher than in previous experiments. This suggests that batch normalization has improved the model's generalization.

Loss Plot (Right)

(a) **Training Loss:** Ends at around 0.4442, lower than in previous runs, indicating that batch normalization allows the model to fit the training data effectively while maintaining stability.

(b) **Test Loss:** Reduces to 0.4747, the lowest test loss observed across all configurations so far. The test loss curve closely tracks the training loss, indicating that batch normalization has mitigated overfitting and led to consistent model behavior.

Metrics Summary

(a) **Test Accuracy (Prec@1):** 83.88%, the highest observed, confirming that batch normalization has improved test performance.

(b) **Test Loss:** At 0.4747, this is a notable reduction in test loss, showcasing batch normalization's positive effect on generalization.

Section 4 - Conclusion

In this project, we successfully implemented a variety of methods to enhance model training and generalization. Each method was carefully selected and evaluated for its contribution to learning, as detailed below:

1. **Standardization of Examples**: By normalizing the input images based on the training set's mean and standard deviation, this method reduces input variance, helping the model converge faster and improving generalization.
2. **Data Augmentation**: Techniques such as random cropping, horizontal flipping, and additional transformations increased the dataset's diversity, allowing the model to generalize better by learning robust, invariant features.
3. **ZCA Normalization**: Tested as an alternative to standard normalization, ZCA reduces correlations in pixel features while preserving spatial structure, although it did not outperform standard normalization in this context.
4. **Learning Rate Scheduling**: Using an exponential decay learning rate scheduler, we enabled the model to explore the parameter space with a high initial rate and then gradually refine with lower rates, enhancing stability and reducing overfitting.
5. **Optimizer Variants**: Testing SGD with momentum, Adam, and learning rate schedules (e.g., Cosine Annealing and OneCycleLR) demonstrated the importance of learning rate modulation for faster convergence and improved generalization.
6. **Dropout Regularization**: By randomly deactivating units during training, dropout mitigated overfitting and encouraged the model to learn more robust, generalized features without excessive memorization.
7. **Batch Normalization**: Batch normalization stabilized and accelerated the training process by standardizing layer inputs, leading to improved convergence and reduced sensitivity to initialization, which resulted in higher accuracy and lower test loss.

Each of these methods contributes uniquely to model robustness and generalization. By addressing overfitting, convergence speed, and generalization through these techniques, we achieved improvements in both training and testing accuracy, validating the effectiveness of each implemented method.