Homework 1
CSC 277 / 477
End-to-end Deep Learning
Fall 2024

Henry Yin - hyin12@u.rochester.edu

Deadline: 10/04/2024

Instructions

Your homework solution must be typed and prepared in LaTeX. It must be output to PDF format. To use LaTeX, we suggest using http://overleaf.com, which is free.

Your submission must cite any references used (including articles, books, code, websites, and personal communications). All solutions must be written in your own words, and you must program the algorithms yourself. **If you do work with others, you must list the people you worked with.** Submit your solutions as a PDF to Blackboard.

Your programs must be written in Python. If a problem requires code as a deliverable, then the code should be shown as part of the solution. One easy way to do this in LaTeX is to use the verbatim environment, i.e., \begin{verbatim} YOUR CODE \end{verbatim}.

About Homework 1: Homework 1 aims to acquaint you with hyperparameter tuning, network fine-tuning, WandB for Training Monitoring, and model testing. *Keep in mind that network training is time-consuming, so begin early!* Copy and paste this template into an editor, e.g., www.overleaf.com, and then just type the answers in. You can use a math editor to make this easier, e.g., CodeCogs Equation Editor or MathType. You may use the AI (LLM) plugin for Overleaf for help you with LaTeXformatting.

Problem 1 - WandB for Training Monitoring

Training neural networks involves exploring different model architectures, hyperparameters, and optimization strategies. Monitoring these choices is crucial for understanding and improving model performance. Logging experiment results during training helps to:

- Gain insights into model behavior (e.g., loss, accuracy, convergence patterns).
- Optimize hyperparameters by evaluating their impact on stability and accuracy.
- Detect overfitting or underfitting and make necessary adjustments.

In this problem, you'll train ResNet-18 models for image classification on the Oxford-IIIT Pet Dataset while exploring various hyperparameters. You'll use Weights and Biases (W&B) to log your experiments and refine your approach based on the results.

Part 1: Implementing Experiment Logging with W&B (6 points)

NOTE: For the following graph, "Step" in x-axis == training epoch
Prepare the Dataset. Download the dataset and split it into training, validation, and test sets as defined in `oxford_pet_split.csv`. Complete the dataset definition in `train.py`. During preprocessing, resize the images to 224 as required by ResNet-18, and apply image normalization using statistics from the training set or from ImageNet.

Evaluating Model Performance. During model training, the validation set is a crucial tool to prevent overfitting. Complete `evaluate()` function in `train.py` which takes a model and a dataloader as inputs and outputs the model's accuracy score and cross-entropy loss on the dataset.

Integrate W&B Logging. To integrate W&B for experiment logging, follow these steps and add the necessary code to `train.py`:

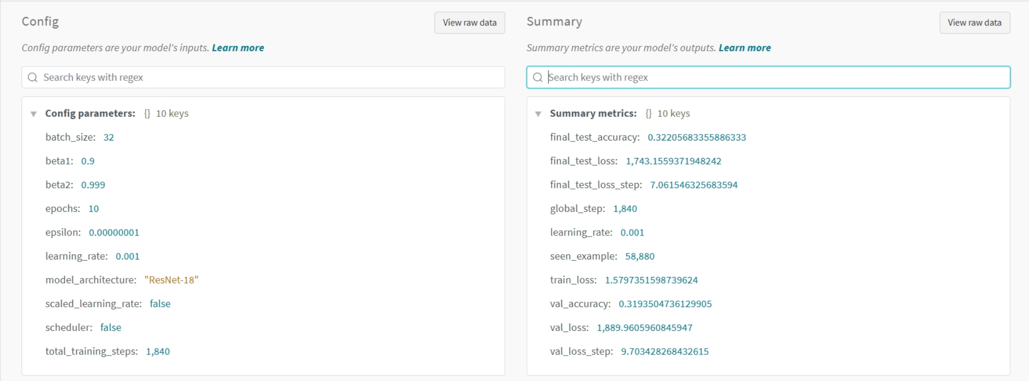
1. Refer to the W&B official tutorial for guidance.
2. Initialize a new run at the start of the experiment following the tutorial's code snippet. Log basic experiment **configurations**, such as total training epochs, learning rate, batch size, and scheduler usage. Ensure the run **name** is interpretable and reflects these key details.
3. During training, log the training loss and learning rate after each mini-batch.
4. After each epoch, log the validation loss and validation accuracy.
5. At the end of the training, log the model's performance on the test set, including loss and accuracy scores.

Experiment and Analysis. Execute the experiment using the **default** setup. Log in to the W&B website to inspect your implementation.

Deliverable:

- Screenshot(s) of the experiment configuration (under the Overview tab)
- Screenshot(s) of all logged charts (under the Charts tab).
- Are the data logged accurately in the W&B interface? Does the experiment configuration align with your expectations?
- Analyze the logged charts to determine whether the training has converged.

Answer:



Config

View raw data

Config parameters are your model's inputs. ***Learn more***

Search keys with regex

Config parameters: {} 10 keys

batch_size: 32

beta1: 0.9

beta2: 0.999

epochs: 10

epsilon: 0.00000001

learning_rate: 0.001

model_architecture: "ResNet-18"

scaled_learning_rate: false

scheduler: false

total_training_steps: 1,840

Summary

View raw data

Summary metrics are your model's outputs. ***Learn more***

Search keys with regex

Summary metrics: {} 10 keys

final_test_accuracy: 0.32205683355886333

final_test_loss: 1,743.1559371948242

final_test_loss_step: 7.061546325683594

global_step: 1,840

learning_rate: 0.001

seen_example: 58,880

train_loss: 1.5797351598739624

val_accuracy: 0.3193504736129905

val_loss: 1,889.9605960845947

val_loss_step: 9.703428268432615

Figure 1: Experiment Config

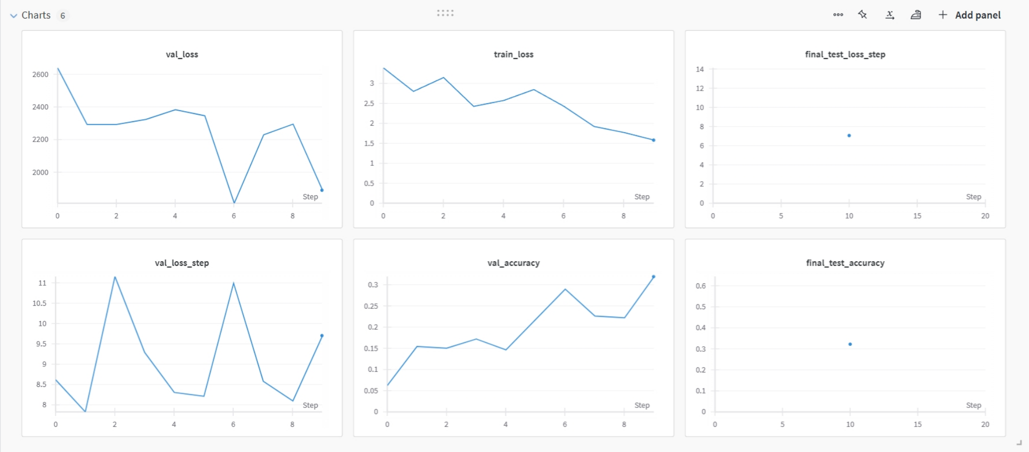


Figure 2: Logged chart

NOTE: val_loss - The total loss on validation set / val_loss_step - The step loss on validation set / train_loss - The step loss on training set.
The logged charts indicate that the training process has not yet fully converged. While the steadily decreasing training loss suggests the model is learning, the fluctuating validation loss points to poor generalization of the model; eventhough the validation accuracy improves from 0.05 to 0.3. To enhance performance, increasing the number of epochs, tuning the learning rate, and incorporating a learning rate scheduler could help stabilize the process and improve generalization.

Part 2: Tuning Hyperparameters

In this section, you'll experiment with key hyperparameters like learning rate and scheduler. For each step, change only one configuration at a time. Try not modify other hyperparameters (except batch size, which can be adjusted based on your computing resources).

2.1. Learning Rate Tuning with Sweep (5 points)

The learning rate is a crucial hyperparameter that significantly affects model convergence and performance. Run the training script using W&B sweep with the following learning rates: $1e-2$, $1e-4$, and $1e-5$. Also, include the default learning rate ($1e-3$) from Part 1 in your analysis.

Deliverable:

- Provide screenshots of logged charts showing learning rate, training loss, validation accuracy, and final test accuracy. Each chart should display results from **multiple runs** (all four learning rates in one chart). Ensure that titles and legends are clear and easy to interpret.
- Analyze how the learning rate impacts the training process and final performance.
- Code of your sweep configuration that defines the search space.

Answer:

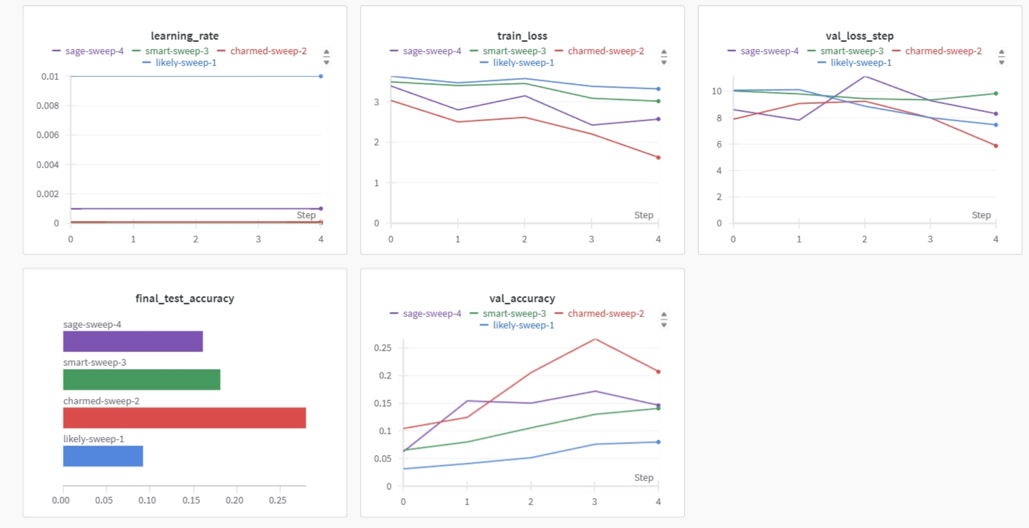


Figure 3: Learning Rate Sweep Charts

```
### Code for Sweep Config for Search Space ###
sweep_config = {
    ’method’: ’grid’,  # Using grid method for fixed learning rates
    ’metric’: {
        ’name’: ’val_accuracy’,
        ’goal’: ’maximize’  # Objective is to maximize validation accuracy
    },
    ’parameters’: {
        ’learning_rate’: {
            ’values’: [1e-2, 1e-4, 1e-5, 1e-3]  # Learning rates to test
        },
        ’batch_size’: {
            ’values’: [32]
        },
        ’epochs’: {
            ’values’: [5]
        },
        ’use_scheduler’: {
            # ’values’: [True, False]
            ’values’: [False]
        }
    }
}
```

Analysis:
The graph shows that In our model, charmed-sweep-2, with its “appropriate” (not too high and not to low) learning rate (LR: 0.0001), leads to faster convergence and the highest test accuracy. The smart-sweep-3 (LR: 0.00001) with relatively lower learning rate might poses risks of instability and potential overfitting, as indicated by fluctuating validation loss. In contrast, smart-sweep-1 (LR: 0.01) and sage-sweep-4 (LR: 0.001), with larger learning rates, exhibit slower descent in training loss and strong fluctuations in validation loss with lower accuracy, potentially indicating risks of instability and potential overfitting as well. Overall, the choice of learning rate should balance quick convergence with the need for stable, reliable generalization, it should be in a appropriate scale avoiding too high or too low values.

2.2. Learning Rate Scheduler (4 points)

Learning rate schedulers dynamically adjust the learning rate during training, improving efficiency, convergence, and overall performance. In this step, you’ll implement the `OneCycleLR` scheduler in the `get_scheduler()` function within `train.py`. Compare the results to the baseline (default setting). If implemented correctly, the learning rate will initially increase and then decrease during training.

Deliverable:

- Provide charts comparing the new setup with the baseline: learning rate, training loss, validation accuracy, and final test accuracy.
- Explain how the `OneCycleLR` scheduler impacts the learning rate, training process, and final performance compared to the baseline.

Answer:

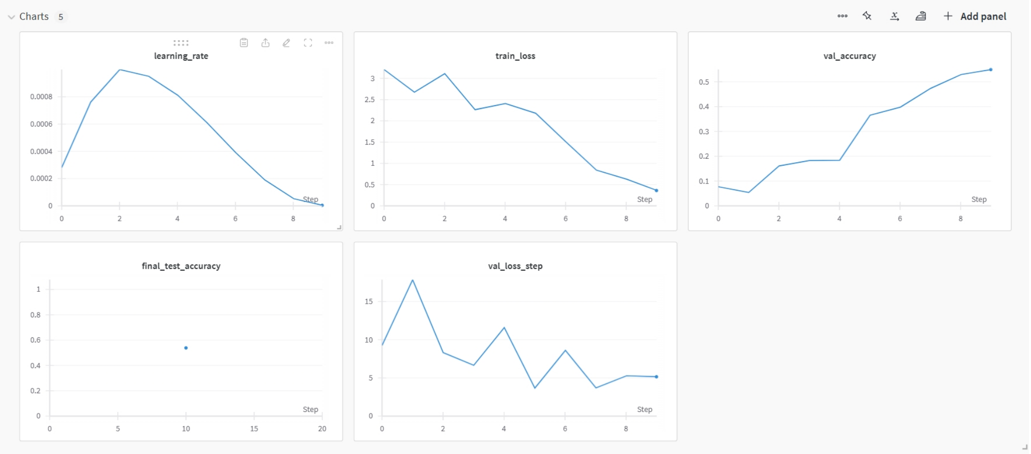


Figure 4: Scheduler Chart

Analysis:
The OneCycleLR scheduler leads to significantly better performance compared to the baseline. In the baseline, where the learning rate remains static, the model shows slower convergence, with training loss fluctuating between 3.0 and 1.5, and final test accuracy stagnating around 0.32. In contrast, with the OneCycleLR scheduler, the learning rate peaks early and then decreases, resulting in faster convergence, with training loss dropping from 3.0 to below 1.0, and final test accuracy improving to approximately 0.6. The validation accuracy also shows steadier improvement, highlighting the scheduler's effectiveness in enhancing both learning stability and final performance. The OneCycleLR scheduler positively impacts the learning process by initially allowing larger learning rate updates to accelerate convergence and avoid potentially local minima, and later reducing the learning rate to refine the model's performance.

Part 3: Scaling Learning Rate with Batch Size (5 points)

As observed in previous parts, the choice of learning rate is crucial for effective training. As batch size increases, the effective step size in the parameter space also increases, requiring adjustments to the learning rate. In this section, you'll investigate how to scale the learning rate appropriately when the batch size changes. Read the first few paragraphs of this blog post to understand scaling rules for Adam (used in default) and SGD optimizers. Then, conduct experiments to verify these rules. First, double (or halve) the batch size without changing the learning rate and run the training script. Next, ONLY adjust the learning rate as suggested in the post. Compare these results with the default setting. Note that since the total training steps vary with batch size, you should also log the number of seen examples to create accurate charts for comparison.

Deliverable:

- Present charts showing: training loss and validation accuracy (with the x-axis being `seen_examples`), and final test accuracy. Ensure the legends are clear. You may apply smoothing for better visualization.
- Analyze the results: do they align with the patterns discussed in the blog post?

Answer:

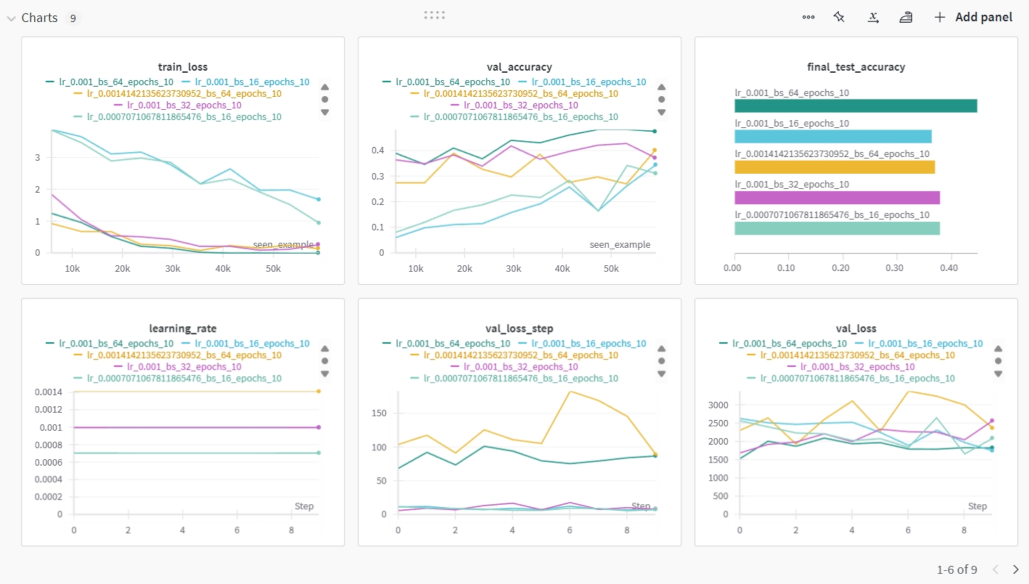


Figure 5: Scheduler Chart

Analysis:
The blog post suggests using the square root scaling rule to adjust the learning rate when changing the batch size, predicting that such adjustments should maintain or improve model performance.

From the supplemental graph, we can observe more detailed evidence supporting the scaling patterns outlined in the blog post. The models with a batch size of 64 with default learning rate of 0.001 and scaled learning rate 0.0014142 (green and yellow lines) show more stable improvements in training and validation metrics, aligning with the theory that learning rates should scale linearly with batch size increases. The lr_0.0014_bs_64_epochs_10 model achieves the lowest training loss loss (around 0.05 when seen_example = 40k) and the highest validation accuracy (above 0.4), reflecting the benefits of higher learning rates with larger batch sizes, as discussed in the blog. Meanwhile, the models with smaller batch sizes (16 and 32) and lower learning rates show more gradual progress, confirming the scaling rules. In conclusion, such observation matches the pattern that larger batch sizes with appropriately adjusted learning rates show improved generalization and stability, consistent with the theory that careful hyper-parameter scaling preserves performance across different batch sizes.

Part 4: Fine-Tuning a Pretrained Model (5 points)

Fine-tuning leverages the knowledge of models trained on large datasets by adapting their weights to a new task. In this section, you will fine-tune a ResNet-18 model pre-trained on ImageNet using `torchvision.models.resnet18(pretrained=True)`. Modify the classification head to match the number of classes in your task, and replace the model definition in the original code. Keep the rest of the setup as default for comparison.

Deliverable:

- Present charts showing: training loss, validation accuracy, and final test accuracy.
- Analyze the impact of pre-training on the model's learning process and performance.

Answer:

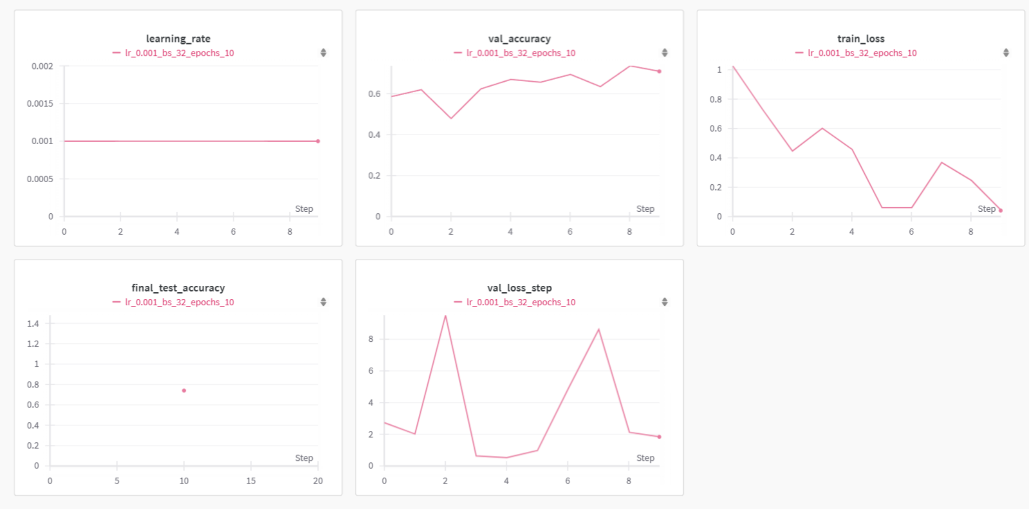


Figure 6: Pretrained-Chart

Analysis:
The pre-trained model shows a clear advantage over the baseline across multiple metrics. In pre-trained model, the validation accuracy reaches 0.6 early and stabilizes, while baseline model struggles around 0.3 with noticeable fluctuations. The train loss in P1 decreases sharply, reaching near-zero faster, whereas baseline model's loss decreases more slowly. Moreover, the validation loss in pre-trained model is significantly lower and more stable than baseline model, indicating better generalization. The final test accuracy for pre-trained model exceeds that of baseline model significantly, showing how pre-training enhances model performance and learning efficiency by improving parameter initialization, accelerating convergence, and improving generalization.

Problem 2 - Model Testing

Unlike model evaluation, which focuses on performance metrics, model testing ensures that a model behaves as expected under specific conditions:

Pre-Train Test: Conducted before training, these tests identify potential issues in the model's architecture, data preprocessing, or other components, preventing wasted resources on flawed training. **Post-Train Test:** Performed after training, these tests evaluate the model's behavior across various scenarios to ensure it generalizes well and performs as expected in real-world situations.

In this problem, you will examine the code and model left by a former employee who displayed a lack of responsibility in his work. The code can be found in the `Problem 2` folder. The necessary predefined functions for this task are available in the `model_testing.py` file. Follow the instructions provided in that file for detailed guidance.

Part 1: Pre-Train Testing

For each question in this part, provide clear deliverables of the following: 1. Observations and analysis of the results; 2. Suggested approaches for addressing the detected issues (if any); 3. Code implementation.

Data Leakage Check (3 points)

Load the training, validation, and test data sets using `get_dataset()` function. Check for potential data leakage between these sets by directly comparing the images, as data augmentation was not applied. Since identical objects usually have different hash values in Python, consider using techniques like image hashing for this comparison.

Answer:
Observations and analysis of the results:
The results show that there were no overlapping images found, which indicates that there is no direct data leakage from the perspective of identical images across these datasets.
Suggested approaches for addressing the detected issues:
Since no data leakage was detected, there are no specific issues to address in this context but I have implemented a *remove_duplicates*() in dataset.py function that might help resovle image overlapping.
Code implementation:

```
# Function to compute image hashes for a dataset
def compute_hashes(dataset):
    hashes = defaultdict(list)
    for idx in range(len(dataset)):
        image, _ = dataset[idx]
        # Convert tensor to PIL image for hashing
        image_pil = transforms.ToPILImage()(image)
        hash_value = imagehash.phash(image_pil)
        hashes[hash_value].append(idx)
    return hashes

# Function to find overlaps between datasets
def find_overlap(hashes1, hashes2):
    overlap = set(hashes1.keys()) & set(hashes2.keys())
    if overlap:
        print(f"Found {len(overlap)} overlapping images.")
        for h in overlap:
            print(f"Hash: {h} - Train indices: {hashes1[h]},
                 Validation/Test indices: {hashes2[h]}")
    else:
        print("No overlapping images found.")

# Function to filter out duplicates from dataset
def filter_dataset(dataset, dataset_hashes, remove_hashes):
    new_indices = []
    for h, indices in dataset_hashes.items():
        if h not in remove_hashes:
            new_indices.extend(indices)
    # Return a new subset dataset with only unique indices
    return torch.utils.data.Subset(dataset, new_indices)

# Potential duplication remove function
def remove_duplicates(train_dataset, val_dataset, test_dataset):
    # Compute hashes for each dataset
    train_hashes = compute_hashes(train_dataset)
    val_hashes = compute_hashes(val_dataset)
    test_hashes = compute_hashes(test_dataset)

    # Find duplicate hashes between train, validation, and test sets
    duplicates = {
        "train_val": set(train_hashes.keys()) & set(val_hashes.keys()),
        "train_test": set(train_hashes.keys()) & set(test_hashes.keys()),
        "val_test": set(val_hashes.keys()) & set(test_hashes.keys())
    }

    # Remove duplicates between train and validation
    train_dataset_cleaned = filter_dataset(train_dataset, train_hashes,
        duplicates[’train_val’] | duplicates[’train_test’])
    val_dataset_cleaned = filter_dataset(val_dataset, val_hashes,
        duplicates[’train_val’] | duplicates[’val_test’])
    test_dataset_cleaned = filter_dataset(test_dataset, test_hashes,
        duplicates[’train_test’] | duplicates[’val_test’])

    print(f"Duplicates removed: Train set:
        {len(train_dataset) - len(train_dataset_cleaned)} images, "
        f"Validation set: {len(val_dataset) - len(val_dataset_cleaned)} images, "
        f"Test set: {len(test_dataset) - len(test_dataset_cleaned)} images.")

    return train_dataset_cleaned, val_dataset_cleaned, test_dataset_cleaned
```

Model Architecture Check (2 points)

Initialize the model using the `get_model()` function. Verify that the model’s output shape matches the label format (hint: consider the number of classes in the dataset).

Answer:

```
    AssertionError: Output shape mismatch: Expected (4, 10), got torch.Size([4, 128])
```

Observations and analysis of the results:
model’s output shape does not match the expected output shape $(batch\_size, num\_classes)$, which is $(4, 10)$, indicating that the final fully connected layer $(fc2)$ in the model is not aligned with the number of classes in the CIFAR-10 dataset.
Suggested approaches for addressing the detected issues:
Removing the fc2 layer or changing it to produce num_classes outputs directly from fc1.

Gradient Descent Validation (2 points)

Verify that ALL the model’s trainable parameters are updated after a single gradient step on a batch of data.

Answer:

```
    Parameter ’conv1.weight’ was successfully updated.
    Parameter ’bn1.weight’ was successfully updated.
    ......
    Parameter ’layer4.2.bn3.bias’ was successfully updated.
    Parameter ’fc1.weight’ was successfully updated.
    Parameter ’fc1.bias’ was successfully updated.
```

Observations and analysis of the results:
According to the code's output, by comparing the parameters after a single training step, all the learnable parameters have been updated successfully.

Learning Rate Check (2 points)

Implement the learning rate range test using pytorch-lr-finder. Determine whether the learning rate is appropriately set by examining the loss-learning rate graph. Necessary components for `torch_lr_finder.LRFinder` are provided in `model_testing.py`.

Answer:

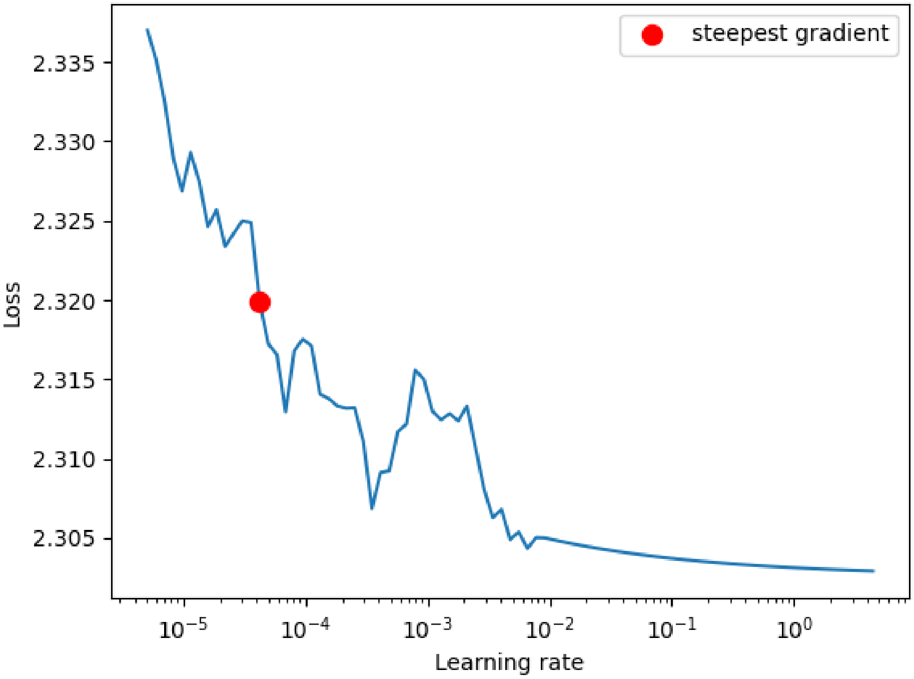


Figure 7: Learning Rate Check

Observations and analysis of the results:
From the learning rate range test using pytorch-lr-finder, we observe a clear pattern where the loss decreases initially as the learning rate increases. This indicates that the model is learning effectively within this range. However, after reaching a certain point (around 10^{-4}), the loss stabilizes and eventually begins to increase, suggesting that the learning rate has surpassed the optimal point, leading to divergence. The steepest descent point

identified at approximately 4.23E-05 suggests that this is the most effective learning rate for training.

All the Code implementations for above questions are avaliable in *model_testing.py* file with clear comments.

Part 2: Post-Train Testing

Dying ReLU Examination (4 points)

In this section, you will examine the trained model for "Dying ReLU." Dying ReLU occurs when a ReLU neuron outputs zero consistently and cannot differentiate between inputs. Load the trained model using `get_trained_model()` function, and the test set using `get_test_set()` function. Review the model's architecture, which is based on ResNet and can be found in `utils/trained_models.py`. Then address the following:

1. Identify the layer(s) where Dying ReLU might occur and explain why.
2. Describe your approach for detecting Dying ReLU neurons.
3. Determine if Dying ReLU neurons are present in the trained model, and provide your code implementation.

Hint: Consider how BatchNorm operation would influence the presence of dying ReLU.

Answer:
Dying ReLU might occur in the deeper convolutional layers (e.g., $layer3$ and $layer4$) where the inputs might have already passed through several ReLU functions. This cumulative effect can cause more neurons to output zero, especially when weights initialized poorly or gradients vanish during backpropagation. Moreover, this issue might also occur in the initial ReLU layers ($bn1$ after $conv1$) if the preceding batch-norm (BatchNorm2d) layer scales inputs to negative values, making the ReLU output zero.
To detect Dying ReLU, we can monitor outputs of all ReLU activation function during the forward pass with batch data. in detail:

- Register forward "hooks" on each ReLU layer in the network to capture output.
- Counting the number of zero activations in the output tensor for each ReLU layer.

Code Implementation:

```
def relu_hook(module, input, output):
    print(f"Hook triggered for module: {module}")
    output = output.detach().cpu()
    total_neurons = output.numel()
    dead_neurons = (output == 0).sum().item()
    activations[module] = dead_neurons / total_neurons
```

```
# Register hooks for all ReLU layers
for name, layer in trained_model.named_modules():
    if isinstance(layer, nn.ReLU):
        print(f"Registering hook for layer: {name}")
        layer.register_forward_hook(relu_hook)

# Pass a batch of data
trained_model.eval()
with torch.no_grad():
    for inputs, labels in test_loader:
        inputs = inputs.to(device)
        if inputs.dim() == 3:  # If 3D, add batch dimension
            inputs = inputs.unsqueeze(0)

        outputs = trained_model(inputs)
        break

# Display
if activations:
    for layer, percentage in activations.items():
        print(f"Layer: {layer}, Dying ReLU Percentage: {percentage:.2%}")
else:
    print("No hooks were triggered or no ReLU layers were found.")
```

Model Robustness Test - Brightness (4 points)

In this section, you will evaluate the model's robustness to changes in image brightness using a defined brightness factor. Define a brightness factor λ, which determines the image brightness by multiplying pixel values by λ. Specifically, $\lambda = 1$ corresponds to the original image's brightness. Load the trained model using `get_trained_model()` function, and the test dataset using `get_test_set()` function. Investigate the model's performance across various brightness levels by adjusting λ from 0.2 to 1.0 in increments of 0.2.

Deliverable:

1. Plot a curve showing how model accuracy varies with brightness levels.
2. Analyze the relationship and discuss any trends observed.

Answer:

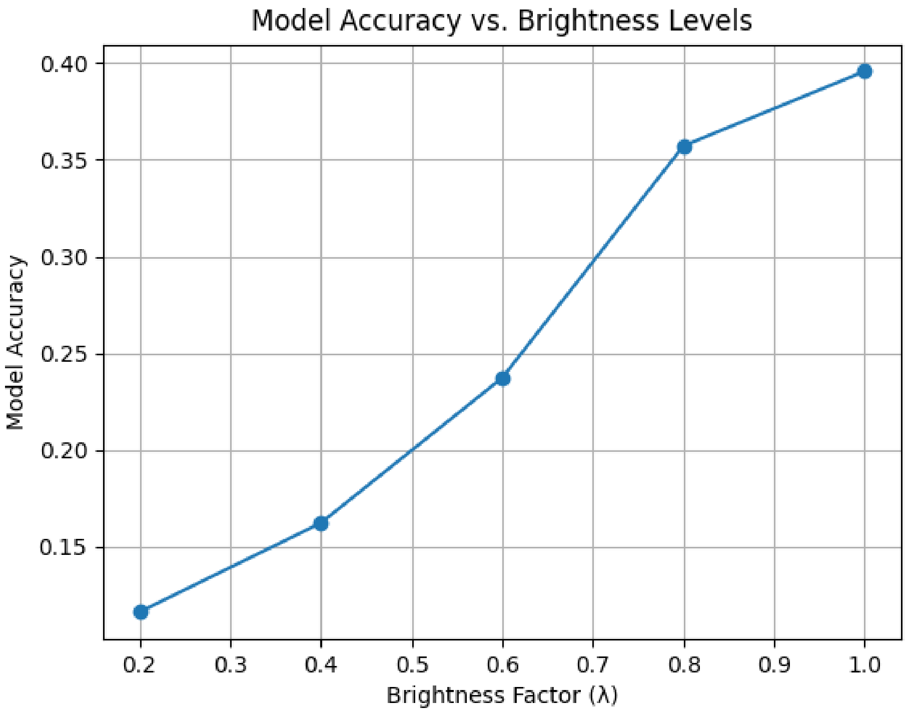


Figure 8: Brightness v.s. Accuracy Chart

Along with the brightness factor increases from 0.2 to 1.0, the model's accuracy steadily improves This trend suggests that the model performs better with images that have brightness closer to original dataset's brightness level. At lower brightness levels, the model's accuracy decreases significantly, indicating that the model is less robust to dimmer images. Moreover, such observed patterns might potentially indicate that the model is overfitting to specific brightness conditions.

Model Robustness Test - Rotation (4 points)

Evaluate the model's robustness to changes in image rotation. Rotate the input image from 0 to 300 degrees in increments of 60 degrees. Similarly, load the trained model using `get_trained_model()` function, and the test set using `get_test_set()` function.

Deliverable:

1. Plot a curve showing the relationship between rotation angles and model accuracy.
2. Analyze the trend and discuss any observed patterns.
3. Suggest potential improvements to enhance model robustness

Answer:

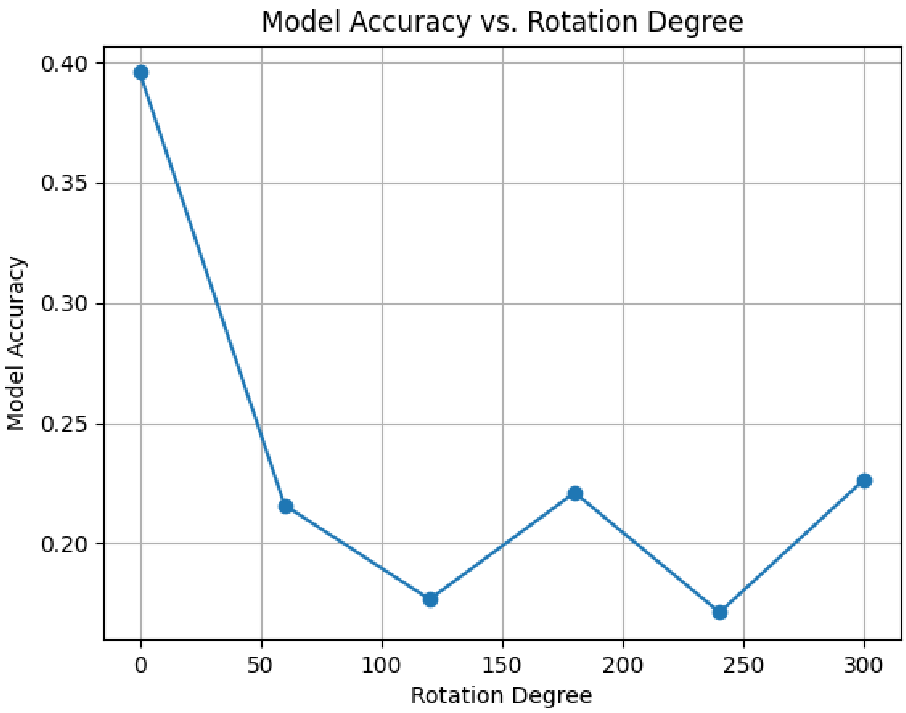


Figure 9: Rotation v.s. Accuracy Chart

The graph shows that model accuracy decreases as rotation angle of input images increases from 0 to 300 degrees. The highest accuracy is observed at 0 degrees (no rotation), and the accuracy drops sharply when the images start to rotate. As the rotation angle increasing, the model's accuracy remains low, with minor fluctuations. This tendency indicates that the model is very sensitive to rotations.
To enhance the robustness of the model, we can do:

- Augment the training dataset with differnet rotations (and brightnesses) to help the model learn rotational invariance.
- Rotation-Invariant Architecture: Use architectures are inherently rotation-invariant, e.g., rotational convolutional layers.
- Fine-tuning: Fine-tune the pretrained model using datasets include a range of rotations (and brightnesses), allowing it to adapt its weights.

Normalization Mismatch (2 points)

Load the test set using the `get_test_set()` function. Assume that the mean and standard deviation (std) used to normalize the testing data are different from those applied to the training data.

Deliverable:

1. Calculate and report the mean and std of the images in the loaded test set (tutorial). Compare these values with the expected mean and std after proper normalization.
2. Discuss one potential impact of this incorrect normalization on the model's performance or predictions.

Answer:

```
Training Mean: [0.49139968 0.48215841 0.44653091]
Training Std: [0.24703223 0.24348513 0.26158784]

Testing Mean: [0.50255482 0.50009291 0.50103502]
Testing Std: [0.28837215 0.28868326 0.28876016]

Differences of mean in %: [2.27007456  3.71962941 12.20612156]
Differences of std in %: [16.7346261  18.56299335 10.38745402]
```

The differences in both mean and standard deviation suggest that there is a mismatch in normalization between the training and testing data. The percentage difference for the standard deviation are quite significant. This discrepancy could negatively impact the model's performance, leading to poorer generalization power.
Moreover, such differences might let model receive inputs that are outside distribution of testing dataset. This can lead low accuracy, unexpected behavior, or higher error rates during testing and other potential issues.