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

John Doe - `jdoe@ur.rochester.edu`

Deadline: See Blackboard

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)

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`:

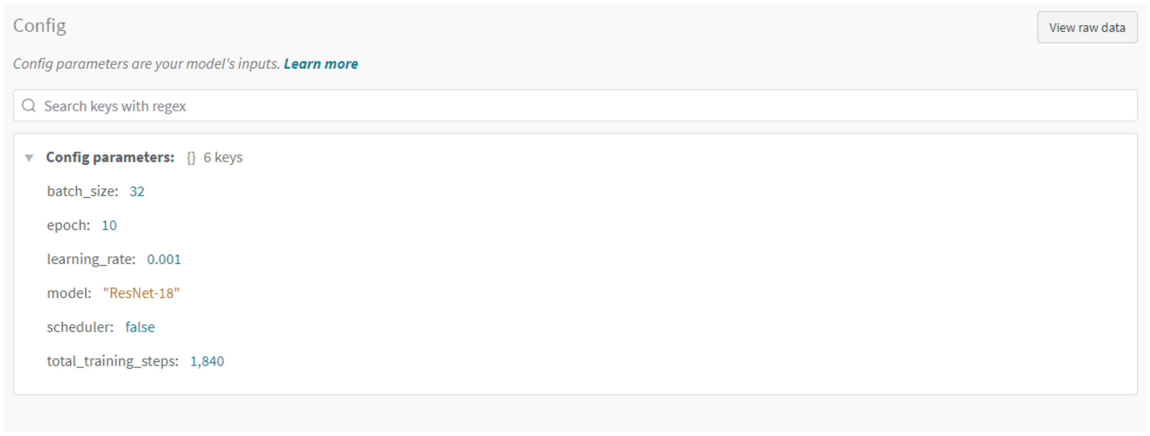
1. Refer to the W&B official tutorial for guidance.
2. Initialize a new run at the start of the experiment following the tutorial's code snippet. Log basic experiment **configurations**, such as total training epochs, learning rate, batch size, and scheduler usage. Ensure the run **name** is interpretable 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:
Configuration:


Config

View raw data

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

Search keys with regex

Config parameters: {} 6 keys

```
batch_size: 32
epoch: 10
learning_rate: 0.001
model: "ResNet-18"
scheduler: false
total_training_steps: 1,840
```

Figure 1: Experiment Configuration

Logged Charts:

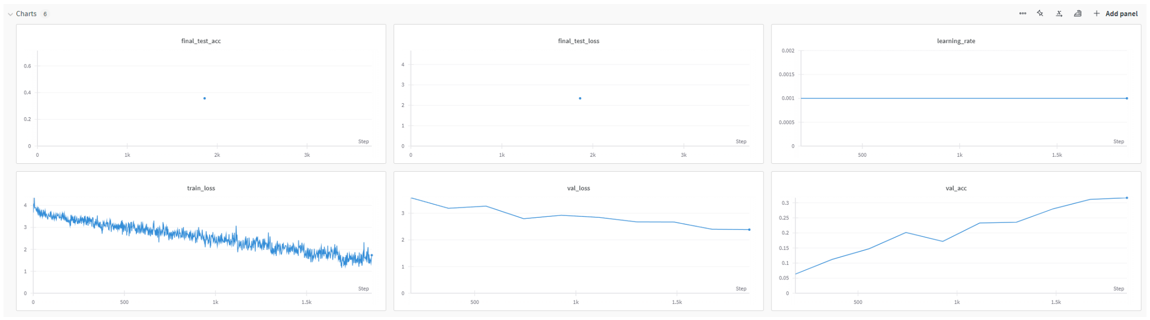


Figure 2: Acc & Loss

Discussion:
The final test accuracy is around 0.356 and the final validation accuracy is around 0.317, which is normally what a small model will get when only given 10 epochs of training. Regarding convergence, I think the training is yet to be converged, since it still has a slight negative slope.

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:

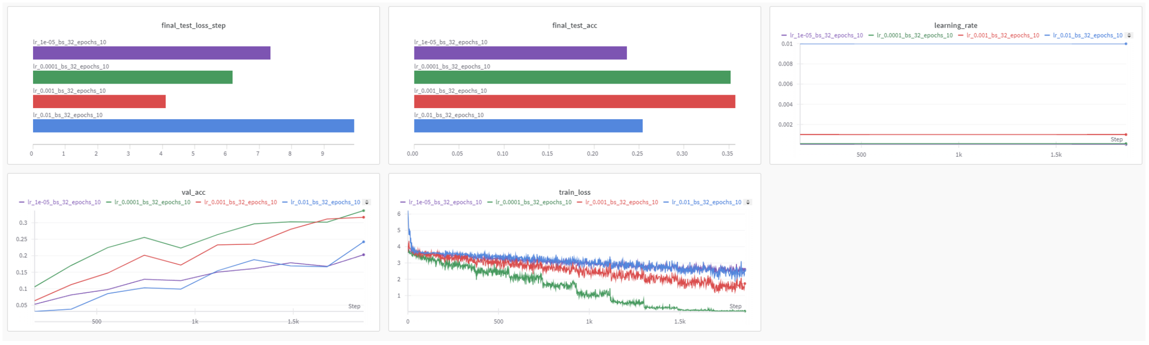


Figure 3: Charts for four learning rates

Discussion:

Learning Rate	**Final Test Accuracy**
1×10^{-2}	0.2544
1×10^{-3}	0.3572
1×10^{-4}	0.3518
1×10^{-5}	0.2368

Table 1: Final Test Accuracies on 4 Learning Rates

I observe the learning rate of 1e-3 (the default value) obtained the highest final test accuracy. Yet, learning rate of 1e-4 obtained the highest validation accuracy. Small learning rate leads to an early convergence and large learning rate makes convergence harder and slower.

Listing 1: Sweep Configuration

```
sweep_config = {
    "name": "Tuning with sweep",
    "method": "grid",  # Ensure each learning rate is picked only once
    "metric": {"goal": "minimize", "name": "validation_loss"},
    "parameters": {
        "learning_rate": {
            'values': [1e-2, 1e-3, 1e-4, 1e-5]
        },
        "batch_size": {
            "values": [32]
        },
        "epochs": {
            "values": [10]
            },
        },
    }
```

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:

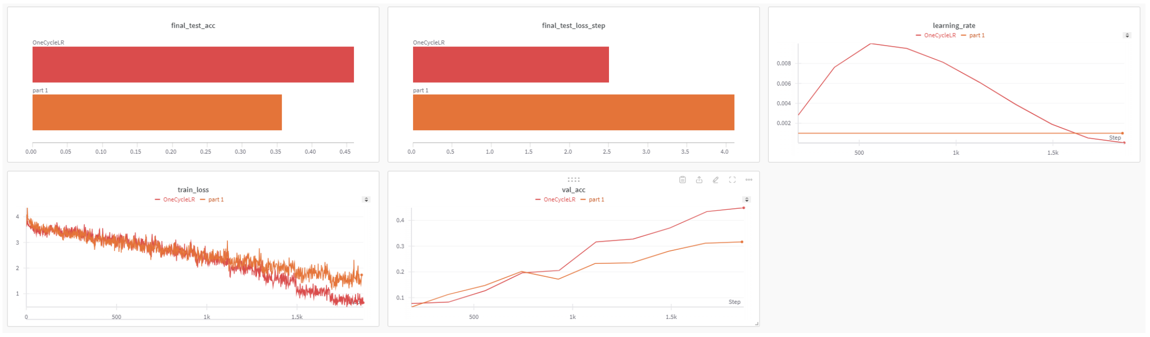


Figure 4: Default vs. OneCycleLR Scheduler

Discussion:
As mentioned in the question prompts, `OneCycleLR` first increase the learning rate to `max_ lr` in the first `pct_ start` fraction of data (I set to 0.3). Once it starts to decrease, a steeper train loss curve occurs, leading to a faster convergence. Simply using `OneCycleLR` scheduler increases final test accuracy from 0.35 to 0.46.

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:

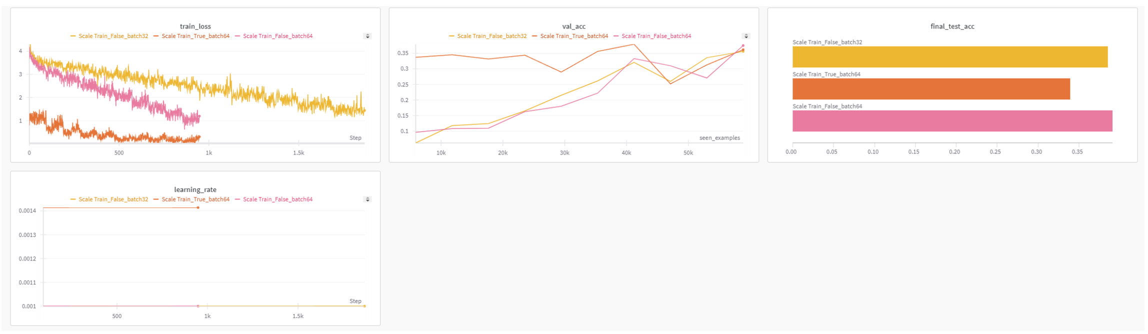


Figure 5: Doubled Batch Size Comparison Chart

Discussion:
I don't find my result align with my observation, that the scaled learning rate didn't boost up performance but rather lead to less final test accuracy. So I wonder what's causing a mismatch between the purpose of scaling law and the result I got.

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:

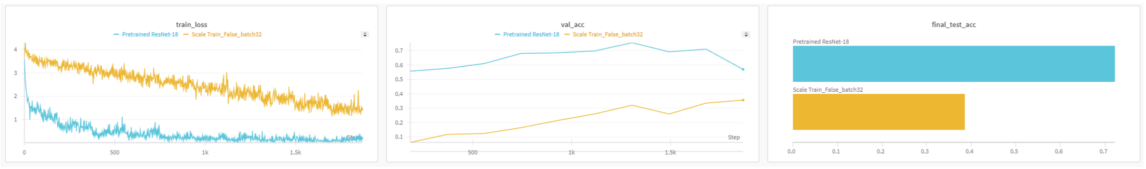


Figure 6: Pretrained vs Default

Discussion:
Usage of pre-trained model provide higher initial performance with low train loss and higher validation accuracy. However, the validation accuracy appears to drop, meaning it lost its generalization capability.

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:

1. Data leakage is found between train set and test set
2. I'm using imagehash library to solve the problem: storing image hashes and compare set directly to detect overlap.
3. Listing 2: datasets.py

```
from PIL import Image
import imagehash
def hash_image(image):
    if isinstance(image, Image.Image):
        return imagehash.phash(image)
    else:
        image = image.detach().numpy()
        image = image.transpose(1, 2, 0)
        image = (image * 255).astype('uint8')
        image = Image.fromarray(image)
        return imagehash.phash(image)


def check_leakage_using_hash(train_dataset, val_dataset,
        test_dataset):
    train_hashes = {hash_image(img): idx for idx, (img, label) in
        enumerate(train_dataset)}
    val_hashes = {hash_image(img): idx for idx, (img, label) in
        enumerate(val_dataset)}
    test_hashes = {hash_image(img): idx for idx, (img, label) in
        enumerate(test_dataset)}
```

```
leakage=False  # initialize leakage flag
if train_hashes.keys() & val_hashes.keys():
    leakage=True
    print('Data leakage found between train set and val set')
if train_hashes.keys() & test_hashes.keys():
    leakage=True
    print('Data leakage found between train set and test set')
if val_hashes.keys() & test_hashes.keys():
    leakage=True
    print('Data leakage found between val set and test set')
if not leakage:
    print('No leakage found')
```

Listing 3: model_testing.py

```
from utils.datasets import get_dataset,
    check_leakage_using_hash
train_dataset, val_dataset, test_dataset = get_dataset()
check_leakage_using_hash(train_dataset, val_dataset,
    test_dataset)
```

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:

1. Model's output shape does not match the label format.
2. I came with a random input tensor and a expected output tensor shape, and I compared the expected output shape to what's the model's output given random input.
3. Listing 4: model_testing.py

```
from utils.models import get_model
model = get_model()
num_classes = 10
input = torch.randn(4, 3, 32, 32)
output = model(input)
expected_output = (4, num_classes)
print(f"Got output shape: {output.shape}, expected output shape
    : {expected_output}")
```

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:

1. Not all trainable parameters are updated. Overall, $\frac{12}{163}$ params are not updated in the network.
2. I looped through all model's *named_parameters* and check if each parameter's gradient is zero, meaning no update.
3. Listing 5: model_testing.py

```
from torch.optim import AdamW
from torch.nn import CrossEntropyLoss
from torch.utils.data import DataLoader
optimizer = AdamW(model.parameters(), lr=1e-6) # the lr is set
    to 1e-6 as specified here
criterion = CrossEntropyLoss()
train_loader = DataLoader(train_dataset, batch_size=32)

inputs, labels = next(iter(train_loader))
outputs = model(inputs)
loss = criterion(outputs, labels)
optimizer.zero_grad()
loss.backward()
all_updated = True
non_count, total=0,0
for name, param in model.named_parameters():
    if param.requires_grad and param.grad is None:
        non_count += 1
        print(f'Parameter \"{name}\" is not updated.')
        all_updated = False
    total+=1
if all_updated: print("All parameters are updated.")
else: print(f"Summary: {non_count}/{total} parameters are not
    updated.")
```

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:

1. Learning rate is initially set to 1^{-6}, which is not appropriately set. The suggested learning rate is $3.35 * 10^{-2}$
2. I used Tweaked version from fastai for the lr examination.

3. Listing 6: model_testing.py

```python
from torch.optim import AdamW
from torch.nn import CrossEntropyLoss
from torch.utils.data import DataLoader
from torch_lr_finder import LRFinder
optimizer = AdamW(model.parameters(), lr=1e-6) # the lr is set
    to 1e-6 as specified here
criterion = CrossEntropyLoss()
train_loader = DataLoader(train_dataset, batch_size=32)
val_loader = DataLoader(val_dataset, batch_size=32)
device=torch.device("cuda:0" if torch.cuda.is_available() else
    "cpu")
lr_finder = LRFinder(model, optimizer, criterion, device=device
    )
lr_finder.range_test(train_loader, val_loader=val_loader,
    end_lr=1, num_iter=500)
lr_finder.plot(log_lr=False)
lr_finder.reset()
```

Part 2: Post-Train Testing

Dying ReLU Examination (4 points)

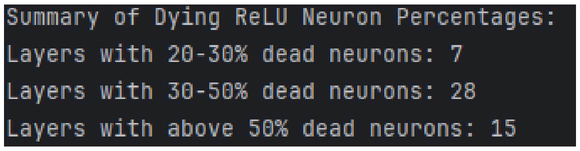
In this section, you will examine the trained model for "Dying ReLU." Dying ReLU occurs when a ReLU neuron outputs zero consistently and cannot differentiate between inputs. Load the trained model using `get_trained_model()` function, and the test set using `get_test_set()` function. Review the model's 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:

1. I think Dying ReLU might occur at two relu layers in the model architecture.
2. My approach to use over 50% dead neurons as an indication as dead neuron. To calculate the percentage of dead neuron in a ReLU layer, the hardest part is to identify intermediate activation; in addition, the model's relu is implemented using 'torch.nn.functional.relu' rather than 'nn.ReLU', so I can not use class detection.



```
Summary of Dying ReLU Neuron Percentages:
Layers with 20-30% dead neurons: 7
Layers with 30-50% dead neurons: 28
Layers with above 50% dead neurons: 15
```

Figure 7: Summary on dyling relu distribution

Therefore, I used ‘contextmanager‘ to replace ‘F.relu‘ to ‘custom_relu‘ to keep track of dead neuron.

3. **Conclusion**:
   Attached is the summary of my ReLU layer dead neuron percentage for ranges 20-30%, 30-50% and 50+%. If considering layer with more than 50% dead neuron to be dead, than I got 15 dead ReLU layers.

Listing 7: trained_models.py

```python
def get_summary(relu_stats):
    summary_stats = {
        "20-30%": 0,
        "30-50%": 0,
        "above 50%": 0
    }
    for i, stat in enumerate(relu_stats):
        if 20 <= stat < 30:
            summary_stats["20-30%"] += 1
        elif 30 <= stat < 50:
            summary_stats["30-50%"] += 1
        elif stat >= 50:
            summary_stats["above 50%"] += 1

    # Print the summary
    print("\nSummary of Dying ReLU Neuron Percentages:")
    print(f"Layers with 20-30% dead neurons: {summary_stats
        ['20-30%']}")
    print(f"Layers with 30-50% dead neurons: {summary_stats
        ['30-50%']}")
    print(f"Layers with above 50% dead neurons: {summary_stats
        ['above 50%']}")
```

Listing 8: trained_models.py

```python
## Check Dying ReLU
from torch.utils.data import DataLoader
from utils.datasets import get_testset
from utils.trained_models import get_trained_model, get_summary
```

```
# Load test data
test_dataset = get_testset()
test_loader = DataLoader(test_dataset, batch_size=32)
relu_stats = []
original_relu = F.relu
def custom_relu(X):
    y = original_relu(X)
    num_zeros = torch.sum(y == 0).item()
    total_neurons = y.numel()
    dead_percentage = num_zeros / total_neurons * 100
    relu_stats.append(dead_percentage)
    return y
# Help to use with statement
@contextmanager
def capture_relu():
    global original_relu
    original_relu = F.relu
    F.relu = custom_relu
    try:
        yield
    finally:
        F.relu = original_relu  # Restore the original F.relu
            after capturing

def check_dying_relu(trained_model, data_loader):
    # Reset relu_stats
    global relu_stats
    relu_stats = []
    trained_model.eval()
    with torch.no_grad():
        with capture_relu():  # Capture F.relu and replace with
             custom relu for inspection
            for inputs, _ in data_loader:
                trained_model(inputs)
                break
    for i, stat in enumerate(relu_stats):
        print(f"ReLU layer {i + 1}: {stat:.2f}% of neurons are
            dead.")
trained_model = get_trained_model()
check_dying_relu(trained_model, test_loader)
# Give summary
get_summary(relu_stats)
```

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:

1. **Graph:**

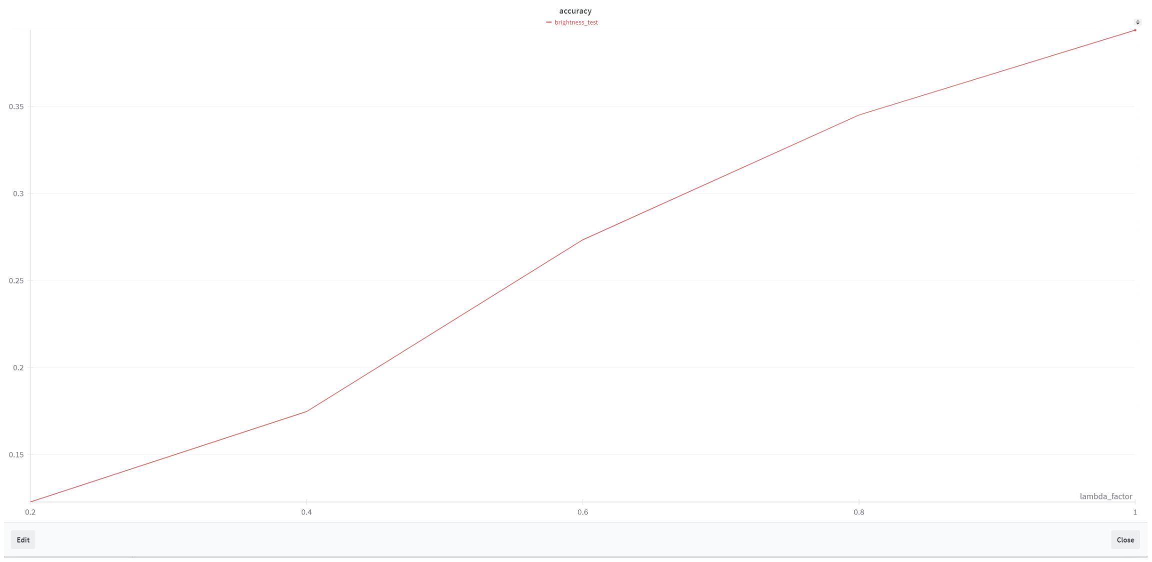


Figure 8: Brightness Curve

2. **Discussion:**
   It can be observed that accuracy is positively related to lambda.

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:

1. **Graph:**

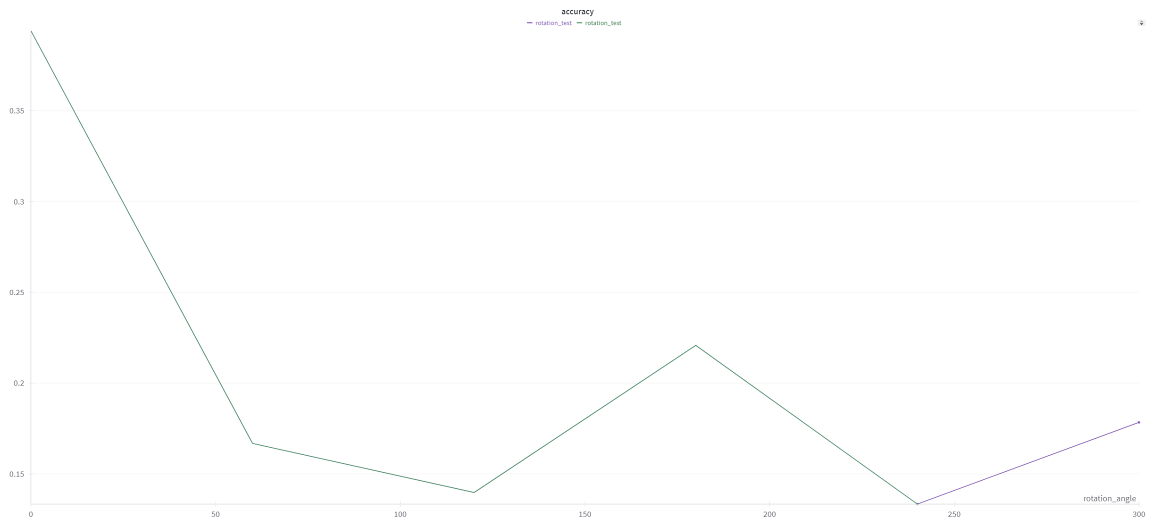


Figure 9: Rotation Curves

2. **Discussion:**
   We can observe the accuracy dropped dramatically soon after images began to rotate; The accuracy recovered somewhat at 180 degree perhaps due to vertical similarity to the image similarity.
3. **Suggestion:**
   Potential ways to improve model robustness is using data augmentation (using random crop, rotate, mixing, etc.)

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:

1. Testing mean: [0.50255482 0.50009291 0.50103502], std: [0.28837215 0.28868326 0.28876016]
   Testing mean: [0.4725, 0.4416, 0.4031], std: [0.2501, 0.2457, 0.2554]
2. **Discussion:**
   Using different statistics to normalize test set may cause shift in data distribution, which will harm model test performance.

1 Reference

1. Exception
2. StepLR
3. Replace model's head
4. imagehash doc
5. Check parameter udpate
6. simple nn dead relu detection
7. forward hook blog
8. python context manager for replacing f.relu to custom_relu
9. adjust_brightness usage
10. F.rotate
11. Calculate image stats