Advanced Deep Learning 2024
Assignment 1

Dengke Chen

May 7, 2024

1 Deep versus wide networks

1.1 Q1

Answer:

The brief description of each part of feedforwardAssignment.ipynb:

1. Imports and Setup: This section loads necessary libraries like PyTorch, NumPy, and Matplotlib. torch: This is the main PyTorch library. It provides support for tensors, which are multi-dimensional arrays optimized for deep learning computations and can be run on GPUs to accelerate processing.

torch.nn: This submodule of PyTorch contains classes and functions that help you create and train neural networks. It includes predefined layers, activation functions, and loss functions, among other necessary tools.

torch.utils.data: This submodule provides helpful utilities for handling datasets. It includes the Dataset and DataLoader classes, which are essential for batch processing and shuffling of data, making it easier to manage data in machine learning workflows.

Tensor: Specifically imported from torch, Tensor is the fundamental data structure of PyTorch. Tensors are similar to NumPy arrays but have additional capabilities that allow them to be used effectively in deep learning models.

numpy: This is a fundamental package for scientific computing with Python. It provides support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

matplotlib.pyplot: This is a plotting library used for creating static, interactive, and animated visualizations in Python. pyplot is a module in matplotlib that makes plotting simpler and similar to MATLAB's plotting system.

```
import torch
...
import matplotlib.pyplot as plt
```

2. Prints the version of PyTorch that is currently being used. This is helpful for debugging and ensuring compatibility with various PyTorch features or third-party libraries.

Determines and sets the computing device for training neural network models. The device can either be a GPU or a CPU. Using a GPU (when available) is preferred because it significantly speeds up the training process of deep learning models due to its parallel processing capabilities. The code checks if a CUDA-capable GPU is available and sets the device to GPU if possible, otherwise it falls back to using the CPU.After determining the device, it prints out which device will be used.

```
print("Using torch", torch.__version__)
device = torch.device("cuda") if torch.cuda.is_available() else
    torch.device("cpu") frame#frame framepreferframe framegpu
print("Device", device)
```

3. Dataset Class Definition: This defines a XORDataset class that inherits from PyTorch's data.Dataset. It is responsible for generating the XOR data along with added Gaussian noise based on a specified standard deviation. The class also contains methods to access the dataset length, individual items, and to convert the data to NumPy format for visualization.

```
class XORDataset(data.Dataset):
    ...
```

4. Visualization Function:The function visualize_binary_samples(dataset) is designed to visually represent a binary-labeled dataset in a scatter plot, which is particularly useful for understanding the distribution and structure of the data points in datasets like the one generated by XORDataset.

```
def visualize_binary_samples(dataset):
    ...
```

5. Model Definition:The class SimpleClassifier which is a PyTorch module intended for creating a neural network designed to handle binary classification tasks, such as the noisy XOR problem. The module also includes functions to train and evaluate such a network. This section defines functions to train the model (train_model) and evaluate its performance (eval_model). These handle the process of running forward passes, backpropagation, updating model parameters, and calculating accuracy.

```
class SimpleClassifier(nn.Module):
    ...

def train_model(model, optimizer, data_loader, loss_module,
   num_epochs=100):
    ...

def eval_model(model, data_loader):
    ...
```

1.2 Q2

The updated code

```
part1:
class SimpleClassifier(nn.Module):
    ...

        layers = [nn.Linear(src, width), nn.Tanh()]
        for _ in range(depth - 1):
            layers.extend([nn.Linear(width, width), nn.Tanh()])
        layers.append(nn.Linear(width, tg))

        layers.append(nn.Identity())
        self.model = nn.Sequential(*layers)

        self.model.to(device)

    def forward(self, x):
        x = self.model(x)
        x = x.squeeze(dim=1)

        return x

def train_model(model, optimizer, data_loader, loss_module,
   num_epochs=100):
    ...

def eval_model(model, data_loader):
    ...



part2:
loss_module = nn.BCEWithLogitsLoss()
train_dataset = XORDataset(size=1000)
train_data_loader = data.DataLoader(train_dataset, batch_size
   =128, shuffle=True)
test_dataset = XORDataset(size=500)

test_data_loader = data.DataLoader(test_dataset, batch_size
   =128, shuffle=False, drop_last=False)
results = []
depths = range(4)
widths = range(1, 4)
repetitions = 3

for depth in depths:
    for width in widths:
        accuracies = []
        for _ in range(repetitions):
            model = SimpleClassifier(src=2, tg=1, depth=depth,
               width=width).to(device)
            optimizer = torch.optim.SGD(model.parameters(), lr
               =0.1)
```

```
                train_model(model, optimizer, train_data_loader,
                    loss_module, num_epochs=200)
                acc = eval_model(model, test_data_loader)
                accuracies.append(acc.cpu().numpy())

            results.append({
                "Depth": depth,
                "Width": width,
                "Mean Accuracy": np.mean(accuracies),
                "Std Accuracy": np.std(accuracies)
            })

results_df = pd.DataFrame(results)
print(results_df)
```

The mean and standard deviation of the resulting loss function

	Depth	Width	Mean Accuracy	Std Accuracy
0	0	1	0.744000	0.002828
1	0	2	0.775333	0.174695
2	0	3	0.906000	0.132936
3	1	1	0.664667	0.115023
4	1	2	0.810667	0.134237
5	1	3	0.901333	0.139536
6	2	1	0.742000	0.002828
7	2	2	1.000000	0.000000
8	2	3	1.000000	0.000000
9	3	1	0.585333	0.117851
10	3	2	0.825333	0.116463
11	3	3	0.915333	0.119737

Figure 1: The mean and standard deviation of the resulting loss function on a new data set for each combination of depth and width

1.3 Q3

Answer:

1.Interpretation of Effects of Changing Depth and Width

The results from my experiment show how different configurations of neural network depths and widths impact classification accuracy on the noisy XOR problem. Here's a brief interpretation:

Depth: Increasing the network depth generally improves the ability to model more complex relationships in data, up to a certain point. In my results, networks

with 2 hidden layers achieved perfect accuracy (1.0) with widths of 2 and 3. However, depth 3 configurations show a decline in performance when the width is 1, indicating potential issues such as overfitting or the network becoming too complex to train effectively with the given data or training regimen.

Width: Increasing the width of the layers increases the capacity of the network. Wider networks can represent more complex functions and have higher representational power. This is evident from the increase in performance as width increases from 1 to 3 in most cases.

Variability: The standard deviation in accuracies tends to decrease with increased depth and width, showing that more complex networks are not only performing better on average but also more consistently across different runs, except for certain configurations like depth 3, width 1.

2.Calculation of Model Parameters

The number of parameters in each network configuration can be calculated based on the number of layers and the number of units in each layer. The formula for calculating the parameters between two connected layers is: $\textbf{parameters} = (input\_features \times output\_features) + output\_features$

Here's the formula applied to each configuration:

For a network with depth d and width w, with input size 2 and output size 1:

First layer: $2 \times w + w$ (input to first hidden layer),

Hidden layers: $(d-1) \times (w \times w + w)$ (subsequent hidden layers if $d > 1$),

Output layer: $w \times 1 + 1$ (last hidden layer to output).

Compute the number of parameters for each configuration using Python code:

```
def calculate_parameters(depth, width):
    parameters = 0
    if depth > 0:
        parameters += 2 * width + width
        for _ in range(depth - 1):
            parameters += width * width + width
        parameters += width * 1 + 1
    else:
        parameters += 2 * 1 + 1
    return parameters

for depth in range(4):
    for width in range(1, 4):
        params = calculate_parameters(depth, width)
        print(f"Depth: {depth}, Width: {width}, Parameters: {
            params}")
```

Output:

Depth: 0, Width: 1, Parameters: 3
Depth: 0, Width: 2, Parameters: 3
Depth: 0, Width: 3, Parameters: 3
Depth: 1, Width: 1, Parameters: 5
Depth: 1, Width: 2, Parameters: 9

Depth: 1, Width: 3, Parameters: 13
Depth: 2, Width: 1, Parameters: 7
Depth: 2, Width: 2, Parameters: 15
Depth: 2, Width: 3, Parameters: 25
Depth: 3, Width: 1, Parameters: 9
Depth: 3, Width: 2, Parameters: 21
Depth: 3, Width: 3, Parameters: 37

3.Influence of Training Parameters

Training parameters such as the learning rate, batch size, and number of epochs significantly influence the training dynamics and final performance of a model:

• Learning Rate: Too high a learning rate can cause the model to converge too quickly to a suboptimal solution, or diverge, while too low a rate can make the training process very slow, potentially getting stuck in local minima.

• Batch Size: Affects the gradient estimation. Smaller batches mean noisier gradient, but faster convergence generally. Larger batches provide a more accurate estimate but require more computation.

• Number of Epochs: More epochs allow more iterations for the model to adjust its weights and find a good fit to the training data.

2 Convolutional neural networks

2.1 Q1

Answer:

The brief description of each part of feedForwardMNIST.ipynb:

1. Dependency Checks and Installations: Checks if running on Google Colab and installs necessary packages like PyTorch and torchvision if it's running there.

```
try:
  import google.colab
  IN_COLAB = True
except:
  IN_COLAB = False
if IN_COLAB:
    !pip3 install torch matplotlib torchmetrics scikit-image
       segmentation-models-pytorch
```

2. Imports: Imports necessary modules from PyTorch for defining a neural network, optimization, and data handling. It also imports torchvision for dataset transformations and data loaders.

```
import torch
...
from torchvision import datasets, transforms
```

3. Set global device: Sets up device configuration to use GPU if available, otherwise falls back to CPU.

```
device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
print('GPU State:', device)
```

4. Training Loop Definition: Defines a function training_loop which takes parameters such as the model, loss function, optimizer, data loader, number of epochs, and device. The loop handles the training process including forward pass, loss calculation, backpropagation, and parameters update. It also prints out loss statistics.

```
def training_loop(model, loss, optimizer, loader, epochs,
    verbose=True, device=device):
     ...
     for epoch in range(epochs):
          ...
                    print('[%d/%d, %d/%d] loss: %.3f' % (epoch
                        +1, epochs, times+1, len(loader),
                        running_loss/2000))
```

5. The evaluate_model function is designed to evaluate the performance of a trained neural network model on the MNIST dataset using PyTorch. It returns detailed results including the total number of correct predictions, the total number of images processed, and per-class accuracy details.

```
def evaluate_model(model, loader, device=device):
     ...
```

6. Defines a transformation pipeline to convert images to tensors and normalize them using mean and standard deviation.

```
transform = transforms.Compose(
     [transforms.ToTensor(),
      transforms.Normalize((0.5,), (0.5,)),]
)
```

7. Data Loading: Downloads and prepares the MNIST dataset for both training and testing. It creates DataLoader objects for both datasets to handle mini-batching, shuffling, and other data iteration mechanisms.

```
trainSet = datasets.MNIST(root='MNIST', download=True, train=
    True, transform=transform)
testSet = datasets.MNIST(root='MNIST', download=True, train=
    False, transform=transform)
trainLoader = dset.DataLoader(trainSet, batch_size=64, shuffle=
    True)
testLoader = dset.DataLoader(testSet, batch_size=64, shuffle=
    False)
```

8. Defines a neural network model Net using PyTorch's nn.Module. This network includes three fully connected layers with ReLU activations and a final layer with log-softmax activation for classification.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(in_features=784, out_features=128),
            nn.ReLU(),
            nn.Linear(in_features=128, out_features=64),
            nn.ReLU(),
            nn.Linear(in_features=64, out_features=10),
            nn.LogSoftmax(dim=1)
        )

    def forward(self, input):
        return self.main(input)
net = Net().to(device)
print(net)
```

9. Training Parameters and Execution: Sets training parameters such as the number of epochs, learning rate, loss function, and optimizer. It then initiates the training process using the earlier defined training_loop. Evaluation: After training, evaluates the model on the test dataset and prints out the overall accuracy as well as class-wise accuracies for the 10 digits.

```
epochs = 4
lr = 0.002
loss = nn.NLLLoss()
optimizer = optim.SGD(net.parameters(), lr=0.002, momentum=0.9)
print('Training on %d images' % trainSet.data.shape[0])
training_loop(net, loss, optimizer, trainLoader, epochs)
print('Training Finished.\n')
correct, total, class_correct, class_total = evaluate_model(net
    , testLoader)
print('Accuracy of the network on the %d test images: %d %%' %
    (testSet.data.shape[0], (100*correct / total)))
for i in range(10):
    print('Accuracy of %d: %3f' % (i, (class_correct[i]/
        class_total[i])))
print(net)
```

Model Definition and Parameters

Defines the neural network model class Net:

Input Layer: Linear layer transforming input from 784 features (flattened $28x28$ image) to 128 features. Parameters: $784 \times 128 + 128$

Hidden Layer 1: Linear layer reducing features from 128 to 64, with ReLU activations following each linear layer. Parameters: $128 \times 64 + 64$

Output Layer: Linear layer transforming 64 features to 10 output classes (digits $0 - 9$) followed by a LogSoftmax layer for classification. Parameters:

$64 \times 10 + 10$

The total number of parameters is the sum of all weights and biases across all layers. The total of 109,386 trainable parameters.

The result of running feedForwardMNIST.ipynb:

```
Training Finished.

Accuracy of the network on the 10000 test images: 94 %
Accuracy of 0: 0.986842
Accuracy of 1: 0.978378
Accuracy of 2: 0.918605
Accuracy of 3: 0.942308
Accuracy of 4: 0.943503
Accuracy of 5: 0.888889
Accuracy of 6: 0.929688
Accuracy of 7: 0.945122
Accuracy of 8: 0.951049
Accuracy of 9: 0.964072
```

Figure 2: Accuracy of the feed-forward network on the 10000 test images

The feed-forward network on the MNIST test dataset is detailed below, both overall and for each digit class:

Overall Accuracy: The model achieves an overall accuracy of 94% on the 10,000 test images. This indicates a strong performance in general, with the model correctly classifying the vast majority of the digits.

Class-wise Accuracy:

Digit 0: 98.68% - Excellent accuracy, indicating near-perfect recognition of this digit.

Digit 1: 97.84% - Similarly high accuracy, showing effective recognition of this digit.

Digit 2: 91.86% - Slightly lower, but still robust accuracy.

Digit 3: 94.23% - Close to the overall average, indicating good recognition.

Digit 4: 94.35% - Again, very close to overall performance.

Digit 5: 88.89% - The lowest among the digits, suggesting some difficulty in correctly identifying this digit.

Digit 6: 92.97% - Decent performance, though there's room for improvement.

Digit 7: 94.51% - Good accuracy, indicating effective recognition.

Digit 8: 95.10% - Slightly above the overall average, showing good recognition capabilities.

Digit 9: 96.41% - Among the higher scores, indicating effective recognition.

From these results, can infer that the network performs very well for most digits, particularly for 0, 1, 7, 8, and 9, which all have accuracies above 94%. The lower performance on digit 5 might indicate specific challenges associated with this digit's variability in handwriting or features that are not as well captured by the network structure.

2.2 Q2

Answer:

The implementation code for this convolutional neural network is in the CNNMNIST.ipynb:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.main = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=(3, 3), stride
                =(1, 1)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(16, 32, kernel_size=(3, 3),
                stride=(1, 1)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.Linear(800, 10),
            nn.LogSoftmax(dim=1)
        )

    def forward(self, input):
        input = input.view(-1, 1, 28, 28)
        return self.main(input)

net = CNN().to(device)
print(net)
```

Why must the Linear layer be preceded by Flatten?

In a CNN, the output from convolutional layers and pooling layers is typically in a multi-dimensional array format (height x width x channels). However, fully connected layers (like the Linear layer) require inputs in a single, flat vector per sample. The Flatten layer reshapes this multi-dimensional output into a one-dimensional array, enabling it to be fed into the fully connected layers.

Why does it have 800 input features?

• The input to the network is typically an MNIST image of size 28×28 pixels with 1 channel (since MNIST images are grayscale).

• After the first convolutional layer $(Conv2d\,(1,\ 16,\ kernel\_size = (3,\ 3),\ stride = (1,\ 1)))$, the dimensionality of the image changes: Each convolution operation reduces the dimension by $kernel\_size - 1$ unless padded. Therefore, the output size after this layer is $(28 - 3 + 1) \times (28 - 3 + 1) = 26 \times 26$. The output depth is 16 due to the 16 filters applied.

• After the first pooling layer $(MaxPool2d(kernel\_size = 2,\ stride = 2))$, the dimension is halved due to the stride of 2, resulting in an output of 13×13 with 16 channels.

• The second convolution layer changes the dimension to $(13-3+1)\times(13-3+1)=11\times 11$, and the output depth increases to 32 due to 32 filters.

• The second pooling layer reduces this to 5×5 with 32 channels.

• Thus, before the flatten operation, the output is $5\times 5\times 32=800$, explaining why the Linear layer needs 800 input features.

The result of running CNNMNIST.ipynb:

```
Training Finished.

Accuracy of the network on the 10000 test images: 97 %
Accuracy of 0: 0.993421
Accuracy of 1: 0.983784
Accuracy of 2: 0.982558
Accuracy of 3: 0.980769
Accuracy of 4: 0.988701
Accuracy of 5: 0.960317
Accuracy of 6: 0.968750
Accuracy of 7: 0.963415
Accuracy of 8: 0.972028
Accuracy of 9: 0.970060
```

Figure 3: Accuracy of the convolutional neural network (CNN) on the 10000 test images

The convolutional neural network (CNN) I implemented for MNIST classification outperforms the previously mentioned feed-forward network, as evidenced by the accuracy results:

Overall Accuracy: The CNN achieves an overall accuracy of 97% on the $10,000$ test images, compared to 94% for the feed-forward network. This 3% improvement indicates a significant enhancement in model performance.

Class-wise Accuracy:

Digit 0: 99.34% (up from 98.68%)

Digit 1: 98.38% (up from 97.84%)

Digit 2: 98.26% (up from 91.86%)

Digit 3: 98.08% (up from 94.23%)

Digit 4: 98.87% (up from 94.35%)

Digit 5: 96.03% (up from 88.89%)

Digit 6: 96.88% (up from 92.97%)

Digit 7: 96.34% (up from 94.51%)

Digit 8: 97.20% (up from 95.10%)

Digit 9: 97.01% (up from 96.41%)

The CNN's improvements are particularly notable in classes where the feed-forward network had lower accuracies, especially such as digits 2, 3, 5, and 6. This enhancement can be attributed to the CNN's ability to capture local dependencies and learn spatial hierarchies,

which is highly advantageous for image data. The spatial feature extraction capabilities of CNNs, thanks to their convolutional and pooling layers, lead to more effective learning of the variabilities and intricacies in handwritten digits. This results in better performance even with potentially fewer parameters.

Calculation of CNN Parameters
First Convolutional Layer Parameters:
Weights: $1 \times 16 \times 3 \times 3 = 144$
Biases: 16
Total: $144 + 16 = 160$
Second Convolutional Layer Parameters:
Weights: $16 \times 32 \times 3 \times 3 = 4608$
Biases: 32
Total: $4608 + 32 = 4640$
Linear Layer Parameters:
Weights: $800 \times 10 = 8000$
Biases: 10
Total: $8000 + 10 = 8010$
Total Parameters in CNN
Adding these up get:
$160 + 4640 + 8010 = 12,810$

The CNN has fewer parameters ($12,810$) compared to the feed-forward network ($109,386$), which is typical since CNNs share weights across spatial locations via the convolution operation, making them much more parameter-efficient.

Despite having fewer parameters, the CNN achieves better performance, showing the efficiency and effectiveness of convolutional layers in handling image data.

3 U-Nets and MLOps

3.1 Q1

Answer:

1.Briefly describe the structure of the notebook x_ray_segmentation_ERDA.ipynb:
Installs – This section probably includes commands to install necessary Python packages.
Imports – Here, the necessary Python libraries are imported.
Set Global Device – Setting the computing device (CPU/GPU) for model training.
Functions:
Visualization – Functions for visualizing data or results.
Evaluating a Model on Data – Functions to evaluate the trained model.
Saving and Loading Model and Optimizer State – Functions to save and load the model state.
Plotting Training/Validation Histories – Functions to plot the training and validation loss or accuracy.
Main Training Loop – The core loop where the model is trained.
Main Program:
Setup Database – Instructions to load and set up the dataset.
Init Model and Optimizer – Initialization of the neural network model and the optimizer.
Continue Training? – Conditional logic to determine whether to continue from a previous training checkpoint.
Run Training – Execution of the training process.
Plot Training and Validation Histories – Visualization of training and validation results.
Evaluate on Single Test-Set Image – Detailed evaluation using a single image from the test set.
Evaluate on Whole Test-Set – Assessment of model performance across the entire test set.

3.2 Q2

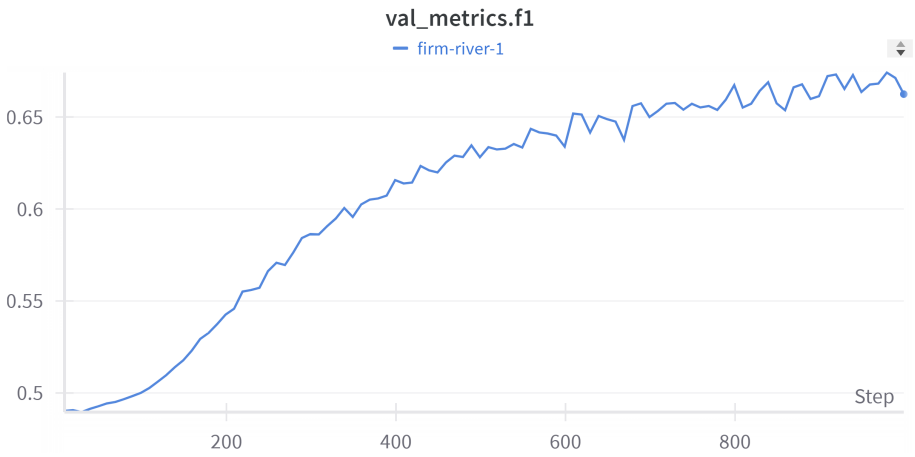
Answer:

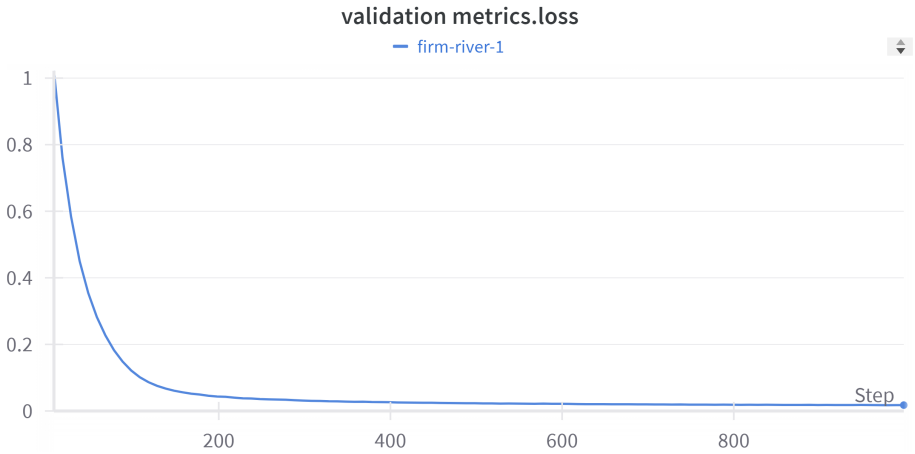
1.This notebook x_ray_segmentation_ERDA.ipynb modification section:
In run_one_epoch: Log the mean training loss at the end of each epoch. Log validation metrics right after they are computed.
In training_loop: Log combined training and validation metrics for an overview after each epoch.

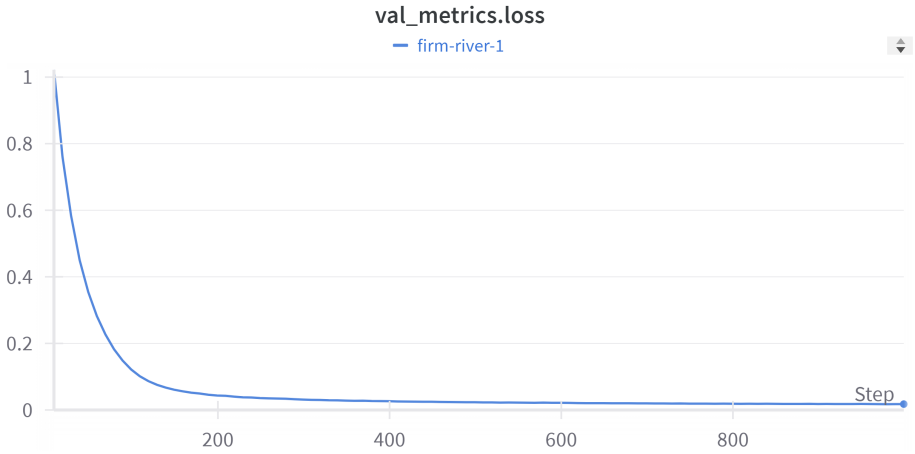
```
!pip3 install wandb
...
```

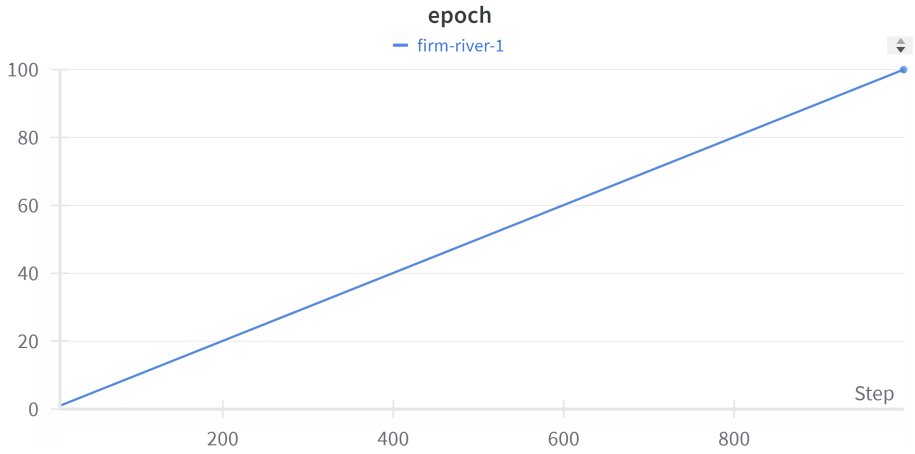
```
import wandb
wandb.login()
...
wandb.init(project='x_ray_segmentation_ERDA', entity='dust1022'
    )
...
def run_one_epoch(model, loss, optimizer, train_loader,
    val_loader, n_epochs, metrics_dict, device=device):
     ...
     for i, (batch_x, batch_y) in enumerate(train_loader):
          ...
          wandb.log({"batch_loss": loss_scalar})

          ...

     mean_train_loss = np.mean(train_losses)
     wandb.log({"train_loss": mean_train_loss})
     ...
     wandb.log({"validation metrics": val_metrics})
     return {"loss": np.mean(train_losses)}, val_metrics

def merge_list_of_dicts(list_of_dicts):
     ...
     return merged

def training_loop(model, loss, optimizer, train_loader,
    val_loader, n_epochs, init_epoch=None, metrics_dict=None,
    save_path=None):
     ...
     try:
          ...
               wandb.log({"epoch": i+1, "train_metrics":
                  train_metrics, "val_metrics": val_metrics})

               ...

               if save_path:
                    ...

     ...
     return merge_list_of_dicts(train_history),
        merge_list_of_dicts(val_history)
```

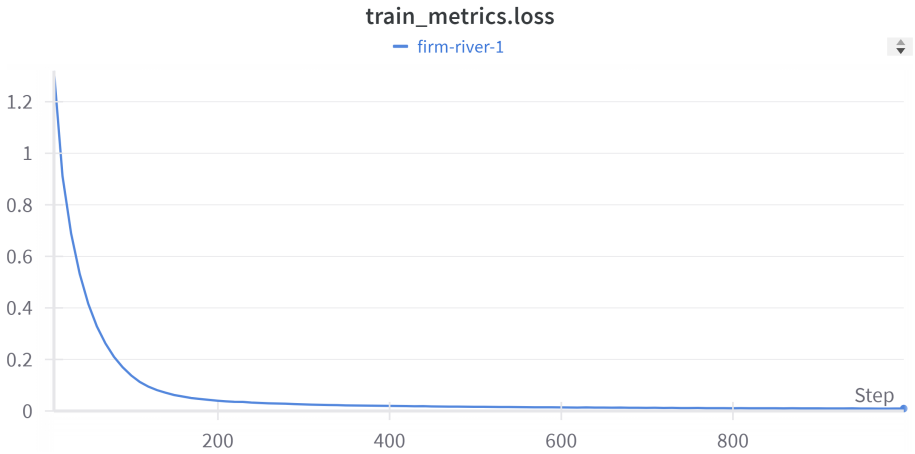
2.The resulting plots:

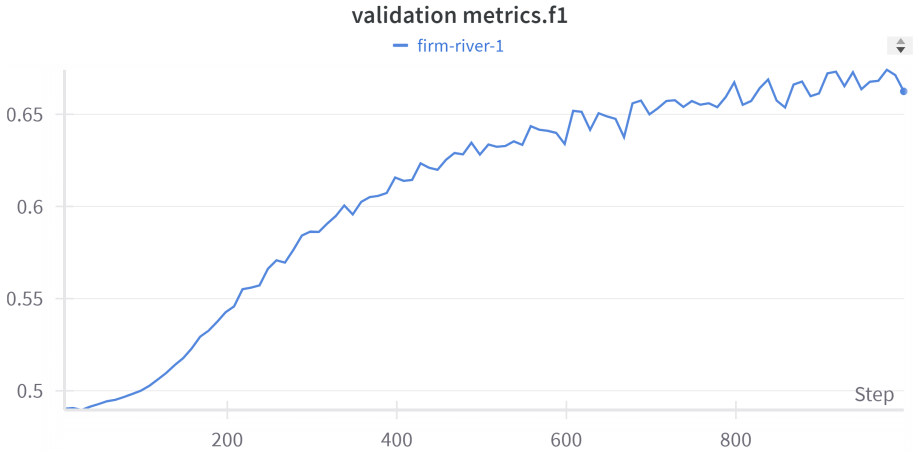

val_metrics.f1
firm-river-1
0.65
0.6
0.55
0.5
Step
200
400
600
800

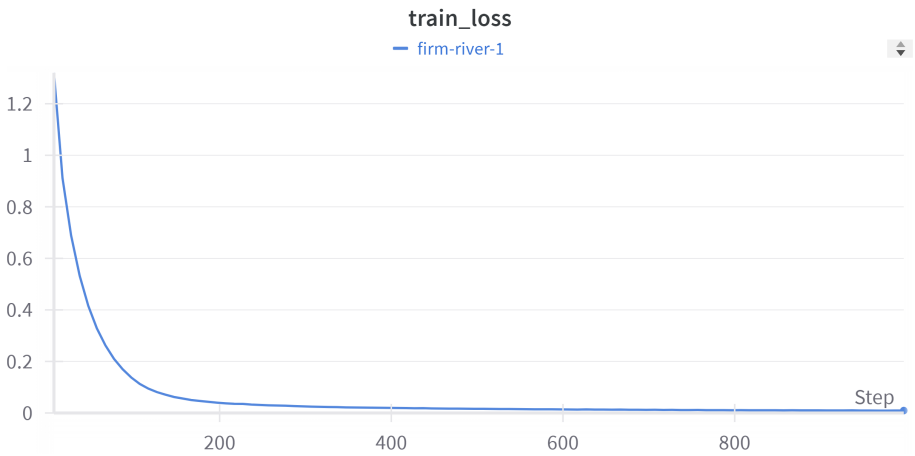

validation metrics.loss
firm-river-1
1
0.8
0.6
0.4
0.2
0
Step
200
400
600
800

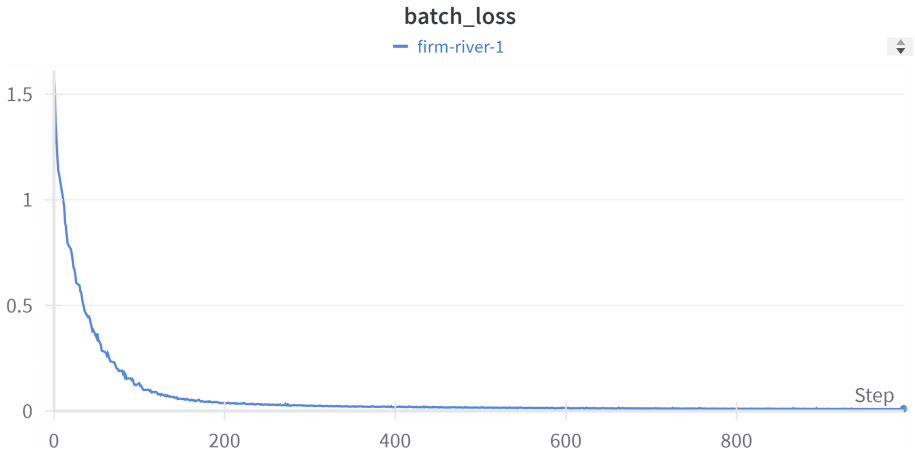

val_metrics.loss
firm-river-1
1
0.8
0.6
0.4
0.2
0
Step
200
400
600
800


epoch
firm-river-1
100
80
60
40
20
0
Step
200
400
600
800


train_metrics.loss
firm-river-1
1.2
1
0.8
0.6
0.4
0.2
0
Step
200
400
600
800


validation metrics.f1
firm-river-1
0.65
0.6
0.55
0.5
Step
200
400
600
800


train_loss
firm-river-1
1.2
1
0.8
0.6
0.4
0.2
0
200
400
600
800
Step

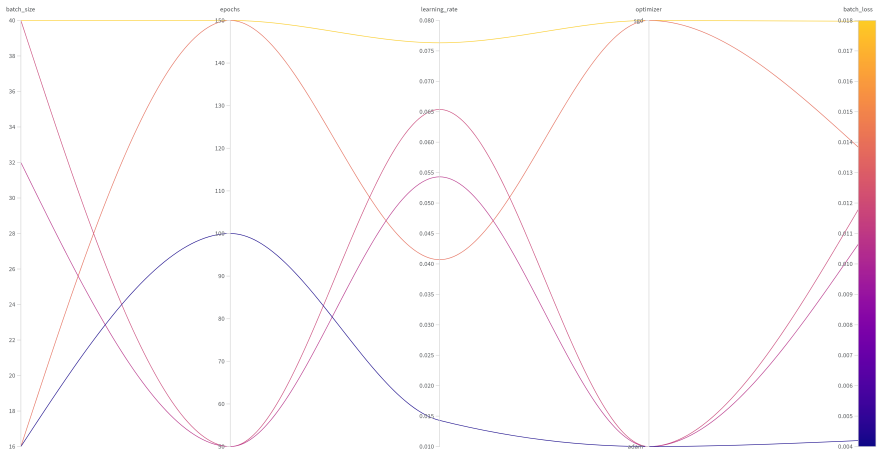

batch_loss
firm-river-1
1.5
1
0.5
0
0
200
400
600
800
Step

3.3 Q3

Answer:

```
import math
sweep_config = {
        'method': 'random',
        'metric': {'goal': 'minimize', 'name': 'loss'},
        'parameters': {
            'batch_size': { 'values': [8, 16, 32,40]
            },
            'epochs': {'values': [50, 100, 150]},
            'learning_rate': {'distribution': 'uniform',
                              'max': 0.1,
                              'min': 0.01},
            'optimizer': {'values': ['adam', 'sgd']}
        }
 }

sweep_id = wandb.sweep(sweep_config, project="test1")
...
wandb.agent(sweep_id, sweep_run, count=5)
```

The parallel coordinate plot in a hyperparameter sweep using an MLOps platform-WandB sweep.



3.4 Q4

Answer:

Experimental Setup

Objective: The main goal of the experiment is to optimize the hyperparameters for a U-Net model used for image segmentation tasks. I aim to minimize the loss, indicating better performance and accuracy of the model.

Model: I utilize a U-Net architecture with an efficientnet-b0 encoder, set for binary classification (segmentation of an object vs. background).

Hyperparameters:

• Batch Size: Determines the number of samples processed before the model updates its internal parameters. I tested batch sizes of 8, 16, 32 and 40 to see how it affects training stability and learning speed.

• Epochs: Total number of complete passes through the training dataset. I explored 50, 100, and 150 epochs to evaluate the trade-off between underfitting and overfitting.

• Learning Rate: The step size at each iteration while moving toward a minimum of a loss function. I used a uniform distribution from 0.01 to 0.1 to observe how the magnitude of updates affects the convergence rate.

• Optimizer: The method used to update weights in the network. I compared 'adam' and 'sgd' to analyze their impact on the convergence speed and final performance.

Methodology: Using WandB, I set up a random sweep with 5 runs, allowing to randomly select combinations of batch size, number of epochs, learning rate, and optimizer type to find the best configuration under varied conditions.

Analysis of Results

The plot provides a visualization of the relationship between different hyperparameters and their corresponding loss values across various configurations during the sweep.

Batch Size and Loss

• Trend: The loss appears to be lowest when the batch size is around 16, with a steep increase as the batch size approaches both the minimum and maximum values tested (8 and 40).

• Implication: Smaller batch sizes might be helping the model generalize better, avoiding overfitting, which could occur with larger batches. However, the smallest batch size tested also leads to higher loss, possibly due to increased noise in the gradient estimates.

Epochs and Loss

• Trend: The number of epochs shows a non-linear relationship with loss. The loss decreases as epochs increase from 50 to around 100, but then increases as epochs approach 150.

• Implication: Training for a moderate number of epochs (around 100) is optimal for this setup. Fewer epochs may not allow sufficient learning, while too many may lead to overfitting.

Learning Rate and Loss

• Trend: The learning rate curve indicates that a learning rate around 0.055 results in the lowest loss. Higher and lower values both lead to increased loss.

• Implication: An intermediate learning rate facilitates better convergence, balancing the speed of learning and the accuracy of updates.

Optimizer and Loss

• Trend: The switch from SGD to Adam optimizer shows a dramatic decrease in loss.

• Implication: Adam optimizer likely provides better performance for this particular model and data due to its adaptive learning rate capabilities, which helps in more efficiently navigating the parameter space.

Color Gradient (Loss Values)

• Trend: The color gradient from purple to yellow represents loss values, with purple indicating lower loss and yellow higher loss. The lower losses are predominantly associated with configurations utilizing the Adam optimizer, intermediate learning rates, moderate epochs, and a batch size around 16.

Preliminary Conclusions

1. Batch Size: A batch size of approximately 16 seems to offer a good balance between efficiency and performance.

2. Epochs: Training the model for around 100 epochs is likely optimal to avoid underfitting and overfitting.

3. Learning Rate: A learning rate close to 0.055 should be considered for further testing as it seems to minimize loss effectively.

4. Optimizer: Adam outperforms SGD significantly in this context, suggesting its use in future experiments for this type of model.

Further Experimentation

• Based on the results, more targeted sweeps focusing around the identified optimal values (batch size near 16, epochs around 100, and learning rate about 0.055) could refine the understanding and possibly uncover even better settings.

• Implement cross-validation to ensure that the conclusions drawn from these runs are robust and generalize well across different subsets of the data.