Gemini as Training Assistant(Documentation of Experiment-001)

Training the Basic Feed Forward Network and Recording all the weights and Biases and learning rates , Accuracies at each epochs

```
Epoch [1/20], Loss: 0.9999, Accuracy: 64.17%, Learning Rate: 0.01
Epoch [2/20], Loss: 0.9481, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [3/20], Loss: 0.8395, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [4/20], Loss: 0.8802, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [5/20], Loss: 0.6411, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [6/20], Loss: 0.8265, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [7/20], Loss: 0.7257, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [8/20], Loss: 0.7822, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [9/20], Loss: 0.7390, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [10/20], Loss: 0.7257, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [11/20], Loss: 0.8146, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [12/20], Loss: 0.6377, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [13/20], Loss: 0.5880, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [14/20], Loss: 0.6925, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [15/20], Loss: 0.8450, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [16/20], Loss: 0.8217, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [17/20], Loss: 0.5894, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [18/20], Loss: 0.9968, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [19/20], Loss: 0.7273, Accuracy: 65.83%, Learning Rate: 0.01
Epoch [20/20], Loss: 0.4853, Accuracy: 65.83%, Learning Rate: 0.01

Training accuracies for each epoch: [64.16666666666667, 65.83333333333333, 65.83333333333333, 65.83333333333333, 65.83333333333333, 65.83333333333333, 65.
83333333333333, 65.83333333333333, 65.83333333333333, 65.83333333333333, 65.83333333333333, 65.83333333333333, 65.83333333333333, 65.83333333333333, 65.83
333333333333, 65.83333333333333, 65.83333333333333, 65.83333333333333, 65.83333333333333, 65.83333333333333]
```

MODEL Summary

[21]:

```python
summary(model, input_size=(1, 4))  # Note: Use batch size of 1 for input size
```

[21…

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
FeedforwardNN                            [1, 3]                    --
├─Linear: 1-1                            [1, 10]                   50
├─Linear: 1-2                            [1, 3]                    33
==========================================================================================
Total params: 83
Trainable params: 83
Non-trainable params: 0
Total mult-adds (M): 0.00
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.00
==========================================================================================
```

Gemini Experimentation

Giving all the weights bias lists and learning rates and other HyperParameters and Model summary to Gemini-Pro-1.5 -002 by a detailed prompt

```
message = (
   f" i trained a feedforward network model, its model summary is like this:{model_summary}"
    f"its learning rates, weights&biases and accuracies are {learning_rates},{weights_biases}, {accuracies} respectively of my model for 20
   "now understand these learning rates, weights and biases, accuracies, and give the optimised list of weights and biases  for my neural ne
)

response = chat_session.send_message(message,safety_settings={
            HarmCategory.HARM_CATEGORY_HATE_SPEECH: HarmBlockThreshold.BLOCK_NONE,
            HarmCategory.HARM_CATEGORY_HARASSMENT: HarmBlockThreshold.BLOCK_NONE,
            HarmCategory.HARM_CATEGORY_DANGEROUS_CONTENT: HarmBlockThreshold.BLOCK_NONE,
            HarmCategory.HARM_CATEGORY_SEXUALLY_EXPLICIT:HarmBlockThreshold.BLOCK_NONE
        })  # Send the first message



print(response.text)
```

Output Generated:

You're asking for optimised weights and biases that will lead to higher accuracy in fewer epochs. Unfortunately, there's no way to magically determine the *optimal* weights and biases directly. Finding these is the entire point of training a neural network!

Here's a breakdown of why and what you *can* do to improve your training process:

Why it's impossible to directly calculate optimal weights:

* **Complex Loss Landscape:** The relationship between weights/biases and the loss function (which measures error) is incredibly complex and non-convex. There are many local minima, and finding the global minimum (the absolute best set of weights) is generally intractable.
* **Data Dependence:** The best weights are highly dependent on your specific training data. A set of weights that works well for one dataset might perform poorly on another.

What you *can* do to improve training:

1. **Learning Rate Tuning:** Your learning rate is constant throughout training. Experiment with different learning rate schedules:
   * **Decaying Learning Rate:** Start with a larger learning rate and gradually reduce it over epochs. This allows for large initial steps and finer adjustments later. Common methods include step decay, exponential decay, and cosine annealing.
   * **Adaptive Learning Rates (Optimizers):** Use optimizers like Adam, RMSprop, or SGD with momentum. These adjust the learning rate for each parameter based on its history, leading to more efficient training.

2. **Hyperparameter Optimization:** The architecture of your network (number of layers, neurons per layer) and other training parameters (batch size, activation functions) can significantly impact performance. Systematically explore different hyperparameter combinations:
   * **Manual Search:** Start with reasonable values and adjust based on your observations.
   * **Grid Search/Random Search:** Automate the process of trying different combinations.
   * **Bayesian Optimization:** More sophisticated methods that try to model the relationship between hyperparameters and performance.

3. **Data Preprocessing:** Ensure your data is properly preprocessed:
   * **Normalisation/Standardization:** Scale your input features to have zero mean and unit variance. This can help with convergence and prevent features with larger magnitudes from dominating.

    * **Data Augmentation:** If you have limited data, artificially increase the size of your training set by applying transformations (e.g., rotations, flips, crops) to your existing data. This can help improve generalization.

4. **Network Architecture:** Consider if your current architecture is appropriate for the task. For more complex relationships, you might need more layers or different activation functions. Too many layers can lead to overfitting, especially with limited data.

5. **Regularization:** Prevent overfitting by:
    * **Dropout:** Randomly ignore neurons during training.
    * **Weight Decay (L2 regularization):** Add a penalty to the loss function based on the magnitude of the weights.

6. **Initialization Strategies:** The initial values of your weights and biases matter. Experiment with different initialization methods (e.g., Xavier/Glorot, He).

7. **Check for Bugs:** Ensure there are no errors in your code or data loading process.

Analyzing Your Current Results:

* **Accuracy Plateau:** Your accuracy seems to plateau quickly. This suggests the learning rate might be too small or the network architecture is not powerful enough.
* **Weights and Biases:** Looking at the raw values of weights and biases doesn't provide much insight. Focus on metrics like loss and accuracy.

Example of incorporating some of these suggestions (PyTorch):

```python
import torch
import torch.nn as nn
import torch.optim as optim

# ... (your data loading and preprocessing code) ...

# Define the model
class FeedforwardNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(FeedforwardNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x)) # Use ReLU activation
        x = self.fc2(x)
        return x

# Create the model
input_size = 4  # Example input size
hidden_size = 10
```

```
output_size = 3
model = FeedforwardNN(input_size, hidden_size, output_size)

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()  # Appropriate for classification
optimizer = optim.Adam(model.parameters(), lr=0.01, weight_decay=0.001) # Adam with weight
decay

# Learning rate scheduler
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1) # Decay LR every 5
epochs


# Training loop
for epoch in range(20):
    # ... (your training loop code) ...
    scheduler.step() # Update learning rate

```

By systematically applying these techniques, you'll have a much better chance of improving your model's performance. Remember, training neural networks is an iterative process. Good luck!

Again Running after Given Suggestions:

Epoch [1/20], Loss: 0.9868, Accuracy: 42.50%, Learning Rate: 0.01
Epoch [2/20], Loss: 0.6509, Accuracy: 76.67%, Learning Rate: 0.01
Epoch [3/20], Loss: 0.6173, Accuracy: 80.83%, Learning Rate: 0.01
Epoch [4/20], Loss: 0.6409, Accuracy: 82.50%, Learning Rate: 0.01
Epoch [5/20], Loss: 0.2067, Accuracy: 85.00%, Learning Rate: 0.01
Epoch [6/20], Loss: 0.5509, Accuracy: 85.83%, Learning Rate: 0.001
Epoch ***[7/20],*** *Loss: 0.5685, Accuracy:* ***86.67%,*** *Learning Rate: 0.001*
Epoch [8/20], Loss: 0.2353, Accuracy: 86.67%, Learning Rate: 0.001
Epoch [9/20], Loss: 0.1943, Accuracy: 86.67%, Learning Rate: 0.001
Epoch [10/20], Loss: 0.3124, Accuracy: 86.67%, Learning Rate: 0.001
Epoch [11/20], Loss: 0.3328, Accuracy: 86.67%, Learning Rate: 0.0001
Epoch [12/20], Loss: 0.5088, Accuracy: 86.67%, Learning Rate: 0.0001
Epoch [13/20], Loss: 0.2376, Accuracy: 86.67%, Learning Rate: 0.0001
Epoch [14/20], Loss: 0.3107, Accuracy: 86.67%, Learning Rate: 0.0001
Epoch [15/20], Loss: 0.3664, Accuracy: 86.67%, Learning Rate: 0.0001
Epoch [16/20], Loss: 0.4555, Accuracy: 86.67%, Learning Rate: 1e-05
Epoch [17/20], Loss: 0.2118, Accuracy: 86.67%, Learning Rate: 1e-05
Epoch [18/20], Loss: 0.2501, Accuracy: 86.67%, Learning Rate: 1e-05
Epoch [19/20], Loss: 0.2795, Accuracy: 86.67%, Learning Rate: 1e-05
Epoch [20/20], Loss: 0.1689, Accuracy: 86.67%, Learning Rate: 1e-05

ACCURACY:86.67%

Given a new prompt to further more optimise it

Prompt: I want to increase my model accuracies to 90 and above modify accordingly the architecture as you need add a gradient descent to get to high accuracy in less time
Epoch [1/20], Loss: 0.8828, Accuracy: 68.33%, Learning Rate: 0.001
Epoch [2/20], Loss: 0.5194, Accuracy: 84.17%, Learning Rate: 0.001
Epoch [3/20], Loss: 0.4107, Accuracy: 87.50%, Learning Rate: 0.000969126572293281
Epoch [4/20], Loss: 0.6791, Accuracy: 86.67%, Learning Rate: 0.0008803227798172156
Epoch [5/20], Loss: 0.1087, Accuracy: 84.17%, Learning Rate: 0.0007445663101277292
Epoch [6/20], Loss: 0.3336, Accuracy: 89.17%, Learning Rate: 0.0005786390152875954
Epoch [7/20], Loss: 0.3189, Accuracy: 92.50%, Learning Rate: 0.00040305238415294404
Epoch [8/20], Loss: 2.0035, Accuracy: 87.50%, Learning Rate: 0.0002395119669243836
Epoch [9/20], Loss: 0.1527, Accuracy: 93.33%, Learning Rate: 0.00010823419302506785
Epoch [10/20], Loss: 0.8135, Accuracy: 89.17%, Learning Rate: 2.5447270110570814e-05
Epoch [11/20], Loss: 0.4600, Accuracy: 87.50%, Learning Rate: 0.0009999037166207915
Epoch [12/20], Loss: 0.0595, Accuracy: 92.50%, Learning Rate: 0.0009904022475614137
Epoch [13/20], Loss: 0.1251, Accuracy: 92.50%, Learning Rate: 0.0009656418599120225
Epoch [14/20], Loss: 1.0876, Accuracy: 87.50%, Learning Rate: 0.0009263937620948692
Epoch [15/20], Loss: 0.2472, Accuracy: 95.00%, Learning Rate: 0.0008738804092678673
Epoch [16/20], Loss: 0.0658, Accuracy: 94.17%, Learning Rate: 0.0008097374276802621
Epoch [17/20], Loss: 0.2513, Accuracy: 92.50%, Learning Rate: 0.0007359626700445858
Epoch [18/20], Loss: 0.6847, Accuracy: 89.17%, Learning Rate: 0.0006548539886902864
Epoch [19/20], Loss: 0.0464, Accuracy: 95.83%, Learning Rate: 0.0005689376646701432
Epoch [20/20], Loss: 0.9455, Accuracy: 85.83%, Learning Rate: 0.00048088972202834545

Observed increase In accuracies..
Experiment-002:Using RNN with LSTM layers for MNIST Classification
Model Summary:

```
==========================================================================================
Layer (type:depth-idx)              Output Shape            Param #
==========================================================================================
LargeRNN                            [32, 3]                 --
├─LSTM: 1-1                         [32, 10, 256]           1,320,960
├─Linear: 1-2                       [32, 128]               32,896
├─BatchNorm1d: 1-3                  [32, 128]               256
├─Dropout: 1-4                      [32, 128]               --
├─Linear: 1-5                       [32, 3]                 387
==========================================================================================
Total params: 1,354,499
Trainable params: 1,354,499
Non-trainable params: 0
Total mult-adds (M): 423.78
==========================================================================================
Input size (MB): 0.01
Forward/backward pass size (MB): 0.72
Params size (MB): 5.42
Estimated Total Size (MB): 6.14
==========================================================================================
```

Training Configuration:

{'optimizer': {'type': 'AdamW', 'learning_rate': 0.001, 'weight_decay': 1e-05, 'betas': (0.9, 0.999)}, 'criterion': {'type': 'CrossEntropyLoss'}, 'scheduler': {'type': 'CosineAnnealingWarmRestarts', 'T_0': 10, 'T_mult': 2, 'eta_min': 1e-06}}
Epoch Details:
Epoch [1/20], Loss: 4.3818, Accuracy: 42.00%, LR: 0.001
Epoch [2/20], Loss: 4.8724, Accuracy: 38.00%, LR: 0.0009619778264893878
Epoch [3/20], Loss: 4.1427, Accuracy: 34.00%, LR: 0.0008536998372026805
Epoch [4/20], Loss: 4.4252, Accuracy: 32.00%, LR: 0.0006916503744663625
Epoch [5/20], Loss: 4.4804, Accuracy: 37.00%, LR: 0.0005005000000000001
Epoch [6/20], Loss: 4.3927, Accuracy: 45.00%, LR: 0.00030934962553363774
Epoch [7/20], Loss: 3.9706, Accuracy: 48.00%, LR: 0.00014730016279731955
Epoch [8/20], Loss: 4.2232, Accuracy: 48.00%, LR: 3.902217351061228e-05
Epoch [9/20], Loss: 4.1204, Accuracy: 44.00%, LR: 0.001
Epoch [10/20], Loss: 4.1267, Accuracy: 44.00%, LR: 0.0009904022475614137
Epoch [11/20], Loss: 4.0107, Accuracy: 46.00%, LR: 0.0009619778264893878
Epoch [12/20], Loss: 3.8474, Accuracy: 54.00%, LR: 0.0009158190713451216
Epoch [13/20], Loss: 4.1737, Accuracy: 51.00%, LR: 0.0008536998372026805
Epoch [14/20], Loss: 3.9390, Accuracy: 51.00%, LR: 0.0007780073313932914
Epoch [15/20], Loss: 3.9510, Accuracy: 45.00%, LR: 0.0006916503744663625
Epoch [16/20], Loss: 4.0083, Accuracy: 55.00%, LR: 0.0005979476158470562
Epoch [17/20], Loss: 3.9629, Accuracy: 52.00%, LR: 0.0005005000000000001
Epoch [18/20], Loss: 3.5524, Accuracy: 60.00%, LR: 0.00040305238415294404
Epoch [19/20], Loss: 4.2349, Accuracy: 56.00%, LR: 0.00030934962553363774
Epoch [20/20], Loss: 3.5362, Accuracy: 57.00%, LR: 0.0002229926686067087

Final Training Configuration Details: {'optimizer': {'type': 'AdamW', 'learning_rate': 0.001, 'weight_decay': 1e-05, 'betas': (0.9, 0.999)}, 'criterion': {'type': 'CrossEntropyLoss'}, 'scheduler': {'type': 'CosineAnnealingWarmRestarts', 'T_0': 10, 'T_mult': 2, 'eta_min': 1e-06}}

Gemini Prompt:

```
message = (
   f" i trained a rnn with  lstm or gru layers with mnist and iris dataset for classification , its model summary is like this:{summary}"
    f"its learning rates, losses and accuracies are {all_lr},{losses}, {accuracies} respectively of my model for 20 epochs"
    f"train_configuration details{training_config}. give me description of trends of lrs and accuracies loses"
   "now understand these learning rates, losses, accuracies, and give the better architecture changes and use nice gradient descents and optimizers and scheduler and do fine tuning for th
)

response = chat_session.send_message(message,safety_settings={
            HarmCategory.HARM_CATEGORY_HATE_SPEECH: HarmBlockThreshold.BLOCK_NONE,
            HarmCategory.HARM_CATEGORY_HARASSMENT: HarmBlockThreshold.BLOCK_NONE,
            HarmCategory.HARM_CATEGORY_DANGEROUS_CONTENT: HarmBlockThreshold.BLOCK_NONE,
            HarmCategory.HARM_CATEGORY_SEXUALLY_EXPLICIT:HarmBlockThreshold.BLOCK_NONE
        })  # Send the first message


print(response.text)
```

The Response:

Trends in Learning Rates, Losses, and Accuracies:

* **Learning Rate (LR):** The learning rate follows a cyclical pattern due to the `CosineAnnealingWarmRestarts` scheduler. It starts at 0.001, gradually decreases, and then resets back to 0.001 every 10 epochs (initially), then 20, 40, and so on. This restarting is evident in the LR values.

* **Loss:** The loss fluctuates significantly and doesn't show a clear downward trend. This instability suggests the model is struggling to converge, possibly due to the frequent LR restarts or the unsuitable architecture (RNN for non-sequential data).

* **Accuracy:** The accuracy remains low (around 33-50%) and doesn't improve consistently. This reinforces the idea that the model isn't learning effectively.

Architectural Changes and Fine-tuning (for MNIST and Iris):

As mentioned in the previous response, using an RNN (LSTM) for MNIST and Iris is fundamentally inappropriate. The correct approach is to use a **CNN for MNIST** and an **MLP for Iris**. The provided RNN architecture and training configuration details are not relevant in this context.

1. MNIST (CNN):

```python
import torch.nn as nn
import torch.optim as optim

class MNIST_CNN(nn.Module):
    # ... (CNN architecture from previous response) ...

model = MNIST_CNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=3, factor=0.1)

# ... (training loop with data loaders for MNIST) ...
```

2. Iris (MLP):

```python
class Iris_MLP(nn.Module):
    # ... (MLP architecture from previous response) ...


model = Iris_MLP()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=3, factor=0.1)

# ... (training loop with data loaders for Iris) ...
```

Key improvements for both MNIST and Iris:

* **Appropriate Architecture:** Using CNN for MNIST and MLP for Iris is the most critical change.
* **ReduceLROnPlateau:** Provides more stable learning rate adjustments.
* **Adam Optimizer:** A generally effective optimizer.

With these changes, you should be able to achieve accuracies well above 95% for MNIST and very high accuracy for Iris (close to 100%). The previous discussions about RNN architecture, fine-tuning with LSTM layers, etc., are not applicable here because the data is not sequential. Focus on using the right model (CNN for MNIST, MLP for Iris) and a good training setup, and you should see significant improvement.

The results from the model after given Gemini suggestions:
Epoch [1/20] - LR: 0.001000, Train Loss: 0.5001, Train Acc: 0.8406, Val Loss: 0.0931, Val Acc: 0.9725
Epoch [2/20] - LR: 0.001000, Train Loss: 0.1492, Train Acc: 0.9531, Val Loss: 0.0440, Val Acc: 0.9864
Epoch [3/20] - LR: 0.001000, Train Loss: 0.1073, Train Acc: 0.9683, Val Loss: 0.0274, Val Acc: 0.9924
Epoch [4/20] - LR: 0.001000, Train Loss: 0.0807, Train Acc: 0.9755, Val Loss: 0.0219, Val Acc: 0.9926
Epoch [5/20] - LR: 0.001000, Train Loss: 0.0743, Train Acc: 0.9750, Val Loss: 0.0167, Val Acc: 0.9947
Epoch [6/20] - LR: 0.001000, Train Loss: 0.0601, Train Acc: 0.9810, Val Loss: 0.0177, Val Acc: 0.9948
Epoch [7/20] - LR: 0.001000, Train Loss: 0.0534, Train Acc: 0.9832, Val Loss: 0.0113, Val Acc: 0.9963
Epoch [8/20] - LR: 0.001000, Train Loss: 0.0503, Train Acc: 0.9829, Val Loss: 0.0123, Val Acc: 0.9962
Epoch [9/20] - LR: 0.001000, Train Loss: 0.0436, Train Acc: 0.9866, Val Loss: 0.0059, Val Acc: 0.9985
Epoch [10/20] - LR: 0.001000, Train Loss: 0.0411, Train Acc: 0.9866, Val Loss: 0.0133, Val Acc: 0.9954
Epoch [11/20] - LR: 0.001000, Train Loss: 0.0336, Train Acc: 0.9884, Val Loss: 0.0042, Val Acc: 0.9988
Epoch [12/20] - LR: 0.001000, Train Loss: 0.0329, Train Acc: 0.9885, Val Loss: 0.0035, Val Acc: 0.9990
Epoch [13/20] - LR: 0.001000, Train Loss: 0.0305, Train Acc: 0.9897, Val Loss: 0.0034, Val Acc: 0.9989
Epoch [14/20] - LR: 0.001000, Train Loss: 0.0315, Train Acc: 0.9891, Val Loss: 0.0049, Val Acc: 0.9984
Epoch [15/20] - LR: 0.001000, Train Loss: 0.0276, Train Acc: 0.9903, Val Loss: 0.0023, Val Acc: 0.9989
Epoch [16/20] - LR: 0.001000, Train Loss: 0.0211, Train Acc: 0.9931, Val Loss: 0.0034, Val Acc: 0.9990
Epoch [17/20] - LR: 0.001000, Train Loss: 0.0249, Train Acc: 0.9911, Val Loss: 0.0012, Val Acc: 0.9997
Epoch [18/20] - LR: 0.001000, Train Loss: 0.0225, Train Acc: 0.9923, Val Loss: 0.0012, Val Acc: 0.9997
Epoch [19/20] - LR: 0.001000, Train Loss: 0.0239, Train Acc: 0.9930, Val Loss: 0.0011, Val Acc: 0.9997
Epoch [20/20] - LR: 0.001000, Train Loss: 0.0239, Train Acc: 0.9915, Val Loss: 0.0014, Val Acc: 0.9995

Observations:
1.Detected the Learning Rates and losses and accuracies trends and analysied the model
2. Made that the model not capable and given better architecture and and optimizer by alnayzing the model Training config and Training report
3. Gave the best changes
4.Meet the given the requirements

This is just experimentation on basic model want to try for Larger models with complex fine tuning