Background and Method Introduction: -

Convolutional Neural Network: -

Morden day Machines now use Convolutional Neural Networks or CNNs to learn from images faster and with better efficiency. These networks help machines to find things, say what is in a picture and divide pictures into parts which helps in analysing the complex patterns and do image classification. CNN architectures typically consist of convolutional layers, which extract spatial features from input images, followed by pooling layers to reduce spatial dimensions and fully connected layers for classification. Scientists use CNN as it has several layers that help them see and think about what the image contains and how the computer can see and process those images. CNN has extensive application in image classification, object detection, and image segmentation.

Application of Convolutional Neural Network in image classification:

In our work, I looked at how well a CNN can identify the pictures in the test set. I chose a type of CNN called ResNet-18 (ResNet stands for Residual Network to use). ResNet is special because it is characterized by its deep structure with skip connections and has 18 layers including convolutional layers, pooling layers, and fully connected layers and it has shortcuts that help it learn better. ResNet-18 has skip connection which help it solve the vanishing gradient problem, allowing for the training of very deep networks.

Dataset and Tasks Description: -

CIFAR-10 Dataset: One often used dataset for image sorting is CIFAR-10. Within it, there are ten groups. Every group has six thousand images in it. There are sixty thousand 32x32 colour images in this dataset. On display are frogs, automobiles, trucks, boats, dogs, cats, deer, and airplanes. There are two parts to the dataset. Training and testing are the two sections. The model is instructed by people during the training phase. 10,000 images are used to evaluate the machine's ability to predict images, and 50,000 images are used for learning.

Resnet-18:

ResNet-18 is a CNN model with 18 layers that uses special blocks to train deep networks without running into the vanishing gradient problem.

Main Parts of ResNet-18:

Convolutional Layers: They get features from images.

Residual Blocks: They have shortcuts to help with training deeper networks.

Pooling Layers: They make the size of feature maps smaller.

Fully Connected Layers: They give probabilities for each category.

Using in Data and Tasks:

People used ResNet-18 for classifying images in the CIFAR-10 dataset. Its deep design and special connections are good for getting features from CIFAR-10's small images, so it classifies Ill.

Classification Tasks: Our job was to teach the CNN how to say what is in each image and classify the images based on the features. This is hard because the pictures are small and different inside the same group and have a variety.

Algorithms Used: -

In-built PY-torch packages and libraries: - I have used the Inbuilt PY-torch packages and used Multiple GPUs to do parallel computing to improve training time and efficiency of the model to predict losses and accuracy. I used PY-torch to make our CNN good because it is flexible and easy to use.

For our model, I used ResNet-18 to put names on images. I have changed its last layer so it would work better with the CIFAR-10 images. I changed the model so it can guess from 10 different groups in the data. I used new methods to make more different types of data. I want the model to learn better. I flip images and cut them in different ways. The pictures keep their meaning. I even tried many settings to make our model work the best. I changed how many parts the model has, how it learns, and if I use a special way to make it learn better. I have used the below algorithms and methods to make the CNN more efficient.

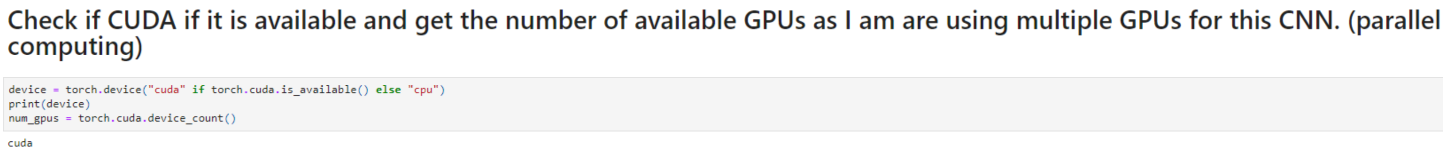
Data Augmentation: I changed the training pictures in different ways by introducing normalization, Regularization, and randomization to transform the data to make it more efficient for learning and prediction. This gave the model more types of pictures to learn from and analyse the image in a better way.

Hyperparameter Tuning: I tried many settings and used grid search so I could identify the best hyper parameters until I found the best ones that could yield the best accuracy. I tweaked on number of hidden nodes in the fully connected layer, the strength of regularization (weight decay), learning rate, and whether to use batch normalization or not.

Parallel Computing: Using more GPU cards made the learning faster. This let us try bigger models and more data easier. When I changed the training time from 10 to 50, the model had more chances to learn the data and it worked better but resulted in a significant overhead but using multiple GPUs to do parallel processing effectively improved the performance significantly.

Let's dive deeper into the code and identify what happened at each step: -

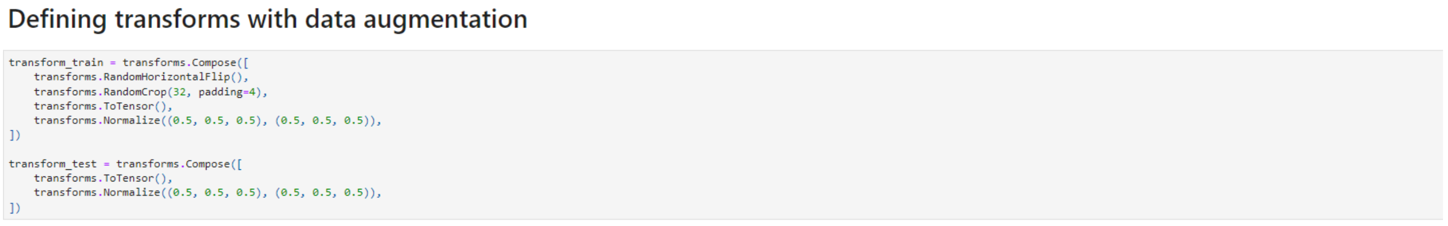
1) **Data Preparation and Augmentation:** After importing the necessary libraries from PY-torch.



Check if CUDA if it is available and get the number of available GPUs as I am are using multiple GPUs for this CNN. (parallel computing)

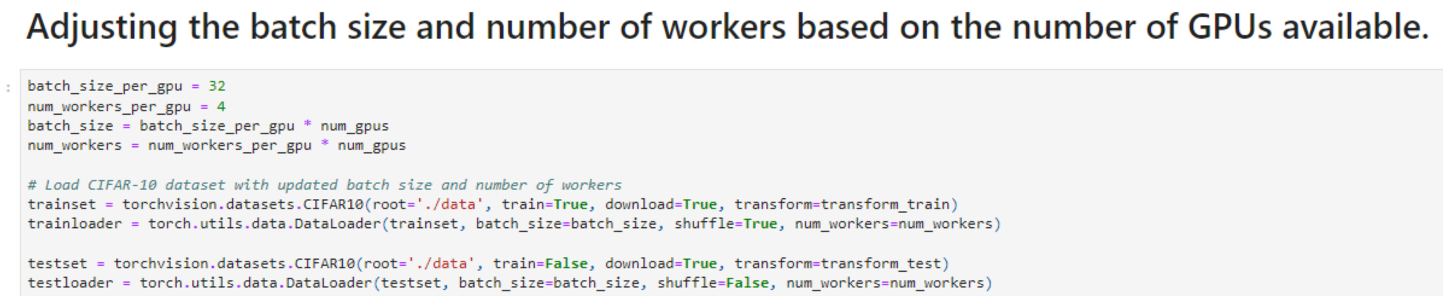
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
num_gpus = torch.cuda.device_count()
```

```
cuda
```



Defining transforms with data augmentation

```
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

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

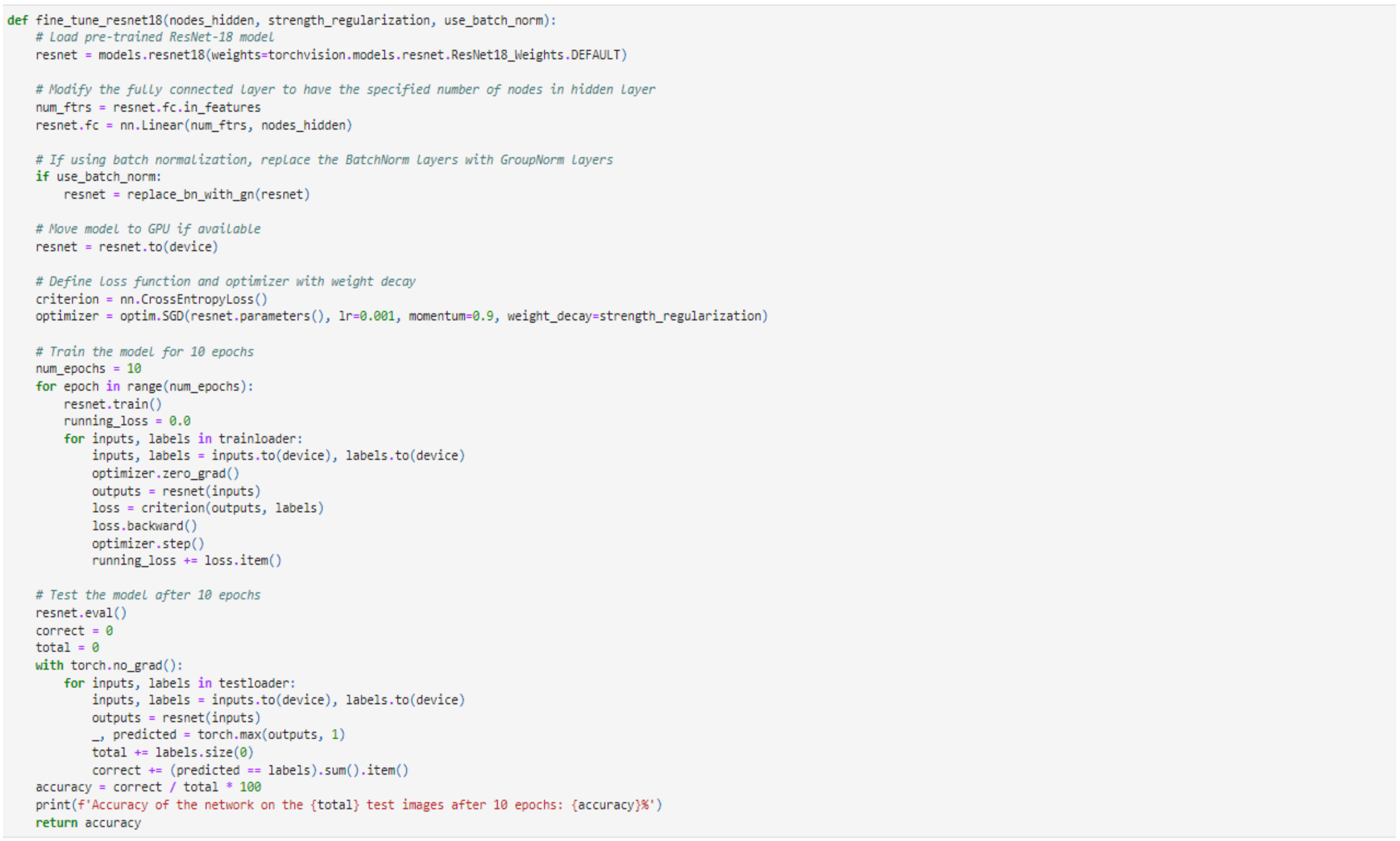


Adjusting the batch size and number of workers based on the number of GPUs available.

```
batch_size_per_gpu = 32
num_workers_per_gpu = 4
batch_size = batch_size_per_gpu * num_gpus
num_workers = num_workers_per_gpu * num_gpus

# Load CIFAR-10 dataset with updated batch size and number of workers
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True, num_workers=num_workers)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False, num_workers=num_workers)
```

2) **Model Definition and Training**: - Here I import Resnet-18 model and define a function to fine tune the resnet-18 model to make it more efficient by doing hyper parameter tuning for (nodes_hidden, strength_regularization, use_batch_norm) and, I replace batch normalization with group normalization.

Defining a function for creating and fine-tuning ResNet-18 model for CIFAR-10 classification.



```python
def fine_tune_resnet18(nodes_hidden, strength_regularization, use_batch_norm):
    # Load pre-trained ResNet-18 model
    resnet = models.resnet18(weights=torchvision.models.resnet.ResNet18_Weights.DEFAULT)

    # Modify the fully connected layer to have the specified number of nodes in hidden layer
    num_ftrs = resnet.fc.in_features
    resnet.fc = nn.Linear(num_ftrs, nodes_hidden)

    # If using batch normalization, replace the BatchNorm layers with GroupNorm layers
    if use_batch_norm:
        resnet = replace_bn_with_gn(resnet)

    # Move model to GPU if available
    resnet = resnet.to(device)

    # Define loss function and optimizer with weight decay
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(resnet.parameters(), lr=0.001, momentum=0.9, weight_decay=strength_regularization)

    # Train the model for 10 epochs
    num_epochs = 10
    for epoch in range(num_epochs):
        resnet.train()
        running_loss = 0.0
        for inputs, labels in trainloader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = resnet(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

    # Test the model after 10 epochs
    resnet.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in testloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = resnet(inputs)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = correct / total * 100
    print(f'Accuracy of the network on the {total} test images after 10 epochs: {accuracy}%')
    return accuracy
```

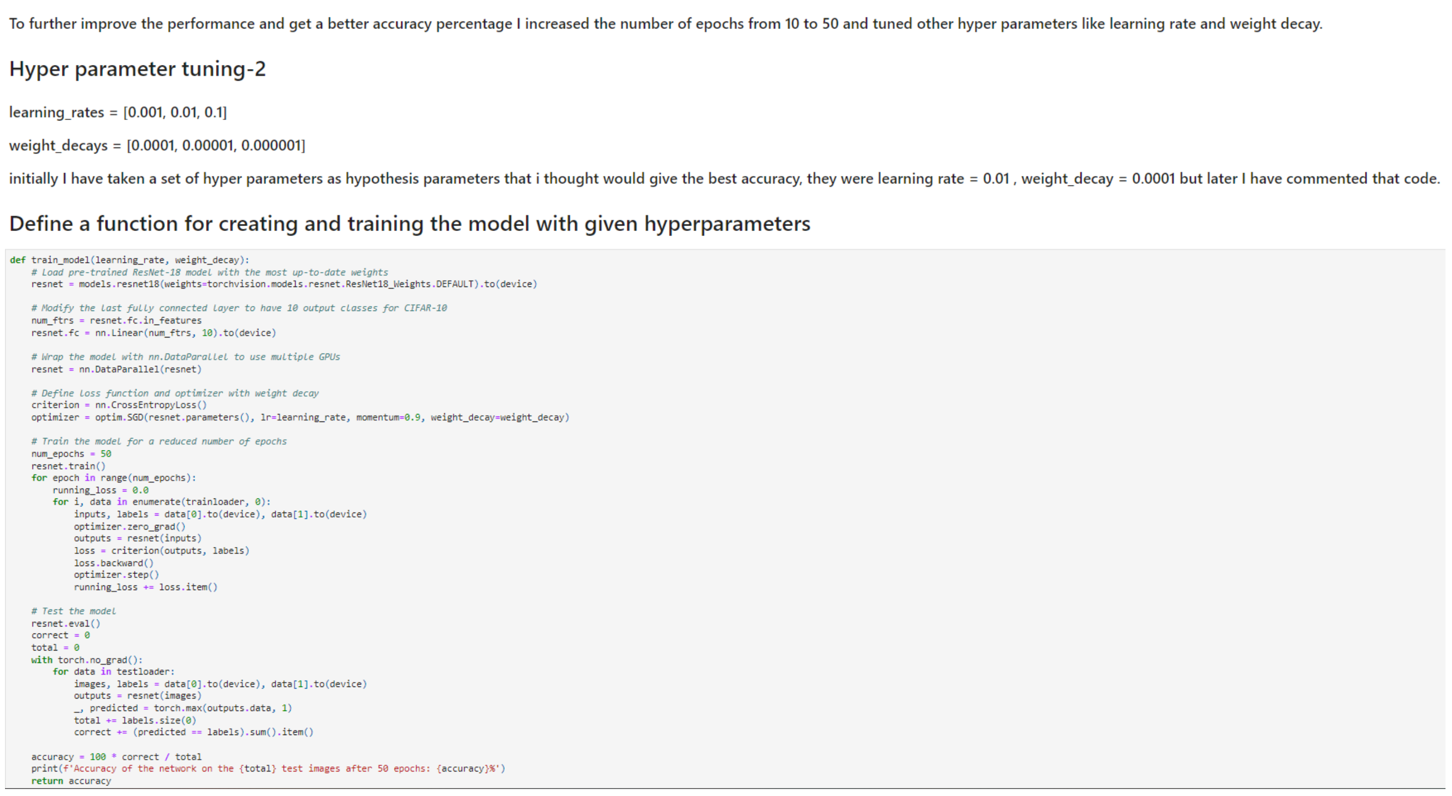
Replace batch normalization with group normalization and perform a grid search to find the best hyper parameter in the list of hyper parameters namely

nodes_hidden_list = [64, 128, 256] this is the number of nodes in the hidden layer

strength_regularization_list = [0.0001, 0.001, 0.01]

use_batch_norm_values = [True, False]

3) **Hyperparameter Tuning: -** After the initial trail of tweaking the hyper parameters, the number of epochs and other hyper parameters were altered.


To further improve the performance and get a better accuracy percentage I increased the number of epochs from 10 to 50 and tuned other hyper parameters like learning rate and weight decay.

Hyper parameter tuning-2

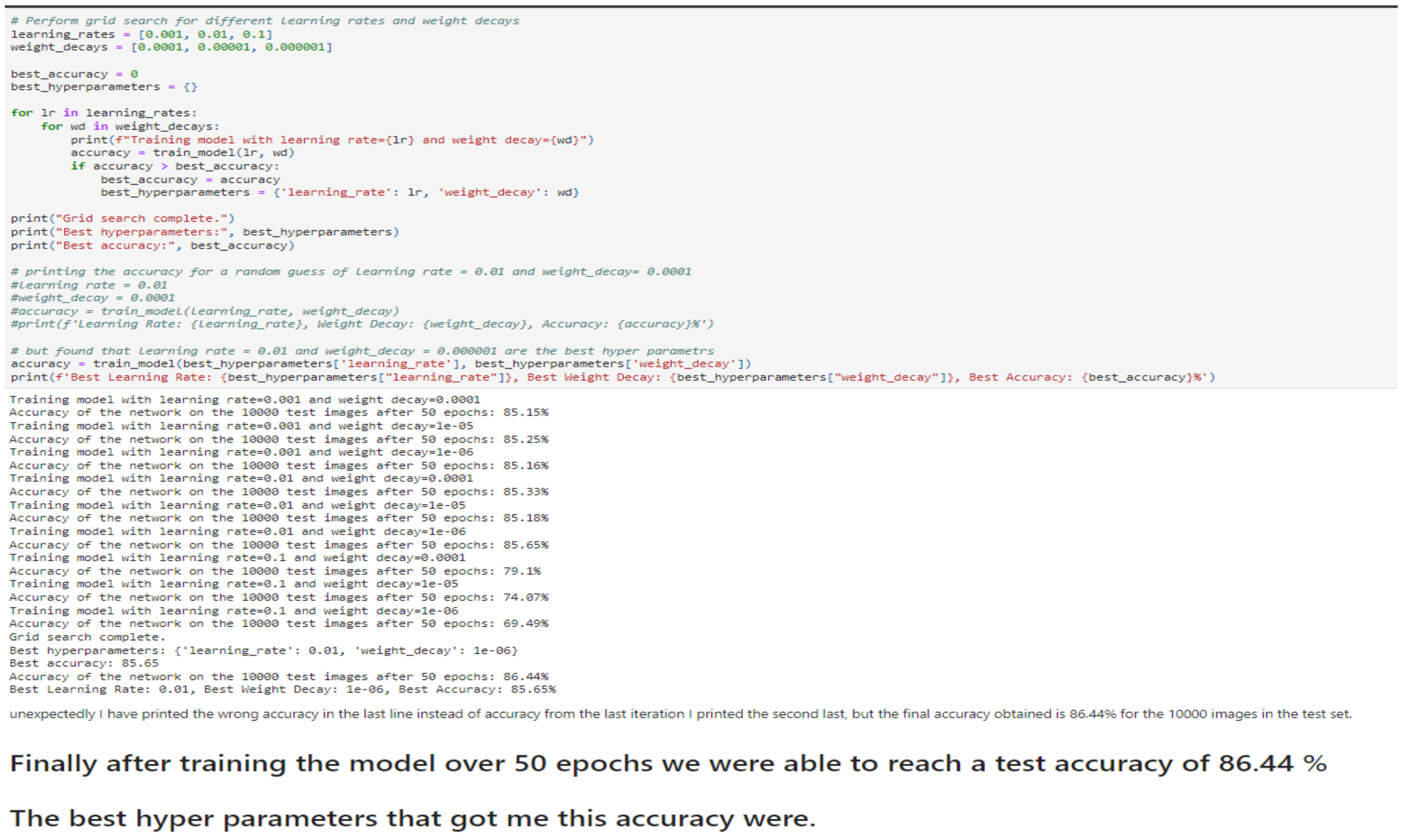
learning_rates = [0.001, 0.01, 0.1]

weight_decays = [0.0001, 0.00001, 0.000001]

initially I have taken a set of hyper parameters as hypothesis parameters that i thought would give the best accuracy, they were learning rate = 0.01 , weight_decay = 0.0001 but later I have commented that code.

Define a function for creating and training the model with given hyperparameters

```
def train_model(learning_rate, weight_decay):
    # Load pre-trained ResNet-18 model with the most up-to-date weights
    resnet = models.resnet18(weights=torchvision.models.resnet.ResNet18_Weights.DEFAULT).to(device)

    # Modify the last fully connected layer to have 10 output classes for CIFAR-10
    num_ftrs = resnet.fc.in_features
    resnet.fc = nn.Linear(num_ftrs, 10).to(device)

    # Wrap the model with nn.DataParallel to use multiple GPUs
    resnet = nn.DataParallel(resnet)

    # Define loss function and optimizer with weight decay
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(resnet.parameters(), lr=learning_rate, momentum=0.9, weight_decay=weight_decay)

    # Train the model for a reduced number of epochs
    num_epochs = 50
    resnet.train()
    for epoch in range(num_epochs):
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            inputs, labels = data[0].to(device), data[1].to(device)
            optimizer.zero_grad()
            outputs = resnet(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

    # Test the model
    resnet.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = resnet(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f'Accuracy of the network on the {total} test images after 50 epochs: {accuracy}%')
    return accuracy
```


```
# Perform grid search for different learning rates and weight decays
learning_rates = [0.001, 0.01, 0.1]
weight_decays = [0.0001, 0.00001, 0.000001]

best_accuracy = 0
best_hyperparameters = {}

for lr in learning_rates:
    for wd in weight_decays:
        print(f"Training model with learning rate={lr} and weight decay={wd}")
        accuracy = train_model(lr, wd)
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_hyperparameters = {'learning_rate': lr, 'weight_decay': wd}

print("Grid search complete.")
print("Best hyperparameters:", best_hyperparameters)
print("Best accuracy:", best_accuracy)

# printing the accuracy for a random guess of learning rate = 0.01 and weight_decay= 0.0001
#learning_rate = 0.01
#weight_decay = 0.0001
#accuracy = train_model(learning_rate, weight_decay)
#print(f'Learning Rate: {learning_rate}, Weight Decay: {weight_decay}, Accuracy: {accuracy}%')

# but found that learning_rate = 0.01 and weight_decay = 0.000001 are the best hyper parametrs
accuracy = train_model(best_hyperparameters['learning_rate'], best_hyperparameters['weight_decay'])
print(f'Best Learning Rate: {best_hyperparameters["learning_rate"]}, Best Weight Decay: {best_hyperparameters["weight_decay"]}, Best Accuracy: {best_accuracy}%')
```

```
Training model with learning rate=0.001 and weight decay=0.0001
Accuracy of the network on the 10000 test images after 50 epochs: 85.15%
Training model with learning rate=0.001 and weight decay=1e-05
Accuracy of the network on the 10000 test images after 50 epochs: 85.25%
Training model with learning rate=0.001 and weight decay=1e-06
Accuracy of the network on the 10000 test images after 50 epochs: 85.16%
Training model with learning rate=0.01 and weight decay=0.0001
Accuracy of the network on the 10000 test images after 50 epochs: 85.33%
Training model with learning rate=0.01 and weight decay=1e-05
Accuracy of the network on the 10000 test images after 50 epochs: 85.18%
Training model with learning rate=0.01 and weight decay=1e-06
Accuracy of the network on the 10000 test images after 50 epochs: 85.65%
Training model with learning rate=0.1 and weight decay=0.0001
Accuracy of the network on the 10000 test images after 50 epochs: 79.1%
Training model with learning rate=0.1 and weight decay=1e-05
Accuracy of the network on the 10000 test images after 50 epochs: 74.07%
Training model with learning rate=0.1 and weight decay=1e-06
Accuracy of the network on the 10000 test images after 50 epochs: 69.49%
Grid search complete.
Best hyperparameters: {'learning_rate': 0.01, 'weight_decay': 1e-06}
Best accuracy: 85.65
Accuracy of the network on the 10000 test images after 50 epochs: 86.44%
Best Learning Rate: 0.01, Best Weight Decay: 1e-06, Best Accuracy: 85.65%
```

unexpectedly I have printed the wrong accuracy in the last line instead of accuracy from the last iteration I printed the second last, but the final accuracy obtained is 86.44% for the 10000 images in the test set.

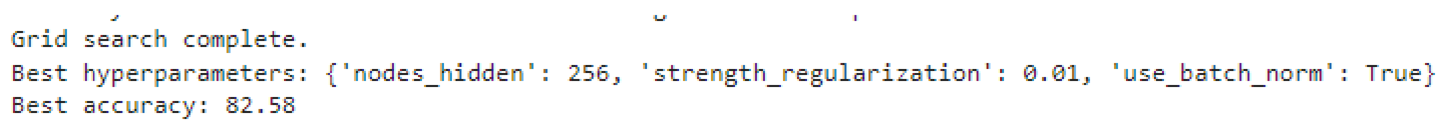
Finally after training the model over 50 epochs we were able to reach a test accuracy of 86.44 %

The best hyper parameters that got me this accuracy were.

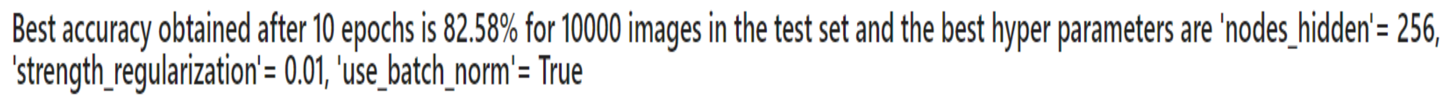
Results: -

Initial Results: -

After doing hyper parameter tuning for number of hidden nodes, regularization strength and use of batch normalization.



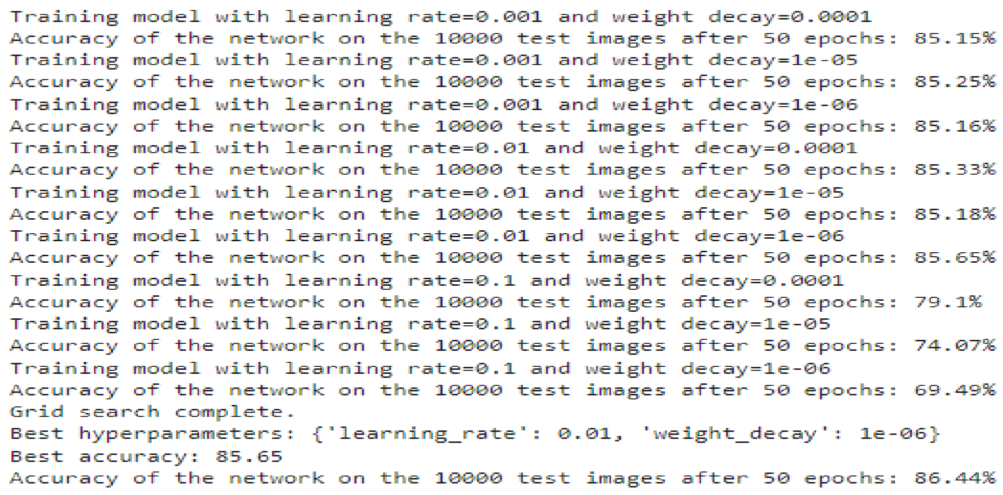
```
Grid search complete.
Best hyperparameters: {'nodes_hidden': 256, 'strength_regularization': 0.01, 'use_batch_norm': True}
Best accuracy: 82.58
```



Best accuracy obtained after 10 epochs is 82.58% for 10000 images in the test set and the best hyper parameters are 'nodes_hidden'= 256, 'strength_regularization'= 0.01, 'use_batch_norm'= True

<u>Final Results: -</u>

I could identify that the accuracy can be further improved as I tweaked the number of epochs from 10 to 50 and changing other hyper parameters like learning rate and weight decay hence, I again could identify a performance improvement, so I did a grid search to identify the best possible hyper parameters to obtain the best accuracy.



```
Training model with learning rate=0.001 and weight decay=0.0001
Accuracy of the network on the 10000 test images after 50 epochs: 85.15%
Training model with learning rate=0.001 and weight decay=1e-05
Accuracy of the network on the 10000 test images after 50 epochs: 85.25%
Training model with learning rate=0.001 and weight decay=1e-06
Accuracy of the network on the 10000 test images after 50 epochs: 85.16%
Training model with learning rate=0.01 and weight decay=0.0001
Accuracy of the network on the 10000 test images after 50 epochs: 85.33%
Training model with learning rate=0.01 and weight decay=1e-05
Accuracy of the network on the 10000 test images after 50 epochs: 85.18%
Training model with learning rate=0.01 and weight decay=1e-06
Accuracy of the network on the 10000 test images after 50 epochs: 85.65%
Training model with learning rate=0.1 and weight decay=0.0001
Accuracy of the network on the 10000 test images after 50 epochs: 79.1%
Training model with learning rate=0.1 and weight decay=1e-05
Accuracy of the network on the 10000 test images after 50 epochs: 74.07%
Training model with learning rate=0.1 and weight decay=1e-06
Accuracy of the network on the 10000 test images after 50 epochs: 69.49%
Grid search complete.
Best hyperparameters: {'learning_rate': 0.01, 'weight_decay': 1e-06}
Best accuracy: 85.65
Accuracy of the network on the 10000 test images after 50 epochs: 86.44%
```

I could identify that the best accuracy achieved in the grid search includes 85.65% accuracy but when I trained the model one last time, I Ire able to get 86.44% accuracy and hence using that iteration for the final result.

<u>Methods of Improvements:</u> -

1) **Parallel computing:** In terms of better coding practices, we have used the GPU instead of CPU and checked to see if we had multiple GPUs to do the training for the model. In my work, I used PyTorch DataParallel to use many GPUs at once. This way we could use more computational power to train the model with data using multiple powerful processors at the same time and make the training faster.

2) **Data Augmentation:** Data augmentation plays a crucial role in training deep learning models, especially when working with limited datasets like CIFAR-10. I have applied the random transformations to the training images, so that I can artificially increase the size and diversity of the training set, thereby improving the model's ability to generalize to unseen data. Some of the common data augmentation techniques are Random Horizontal Flips, Random Crops, Colour Jittering and Data Normalization and adding Regularization.
3) **Hyperparameter Tuning:** Hyperparameters like learning rate, weight decay, and the number of hidden nodes affect how well deep learning models work. We change hyperparameters to

search for the best mix that give good results. I used a grid search to make the model work better.

Learning rate makes the steps bigger or smaller when we make the model better. If the learning rate is high, the model might learn too fast and miss the best answer. If the learning rate is low, the model might learn too slow but does not go past the best answer.

Weight decay helps the model not to learn too much. It does not let the weights in the model get too big. This makes the model work better on new data.

The number of hidden nodes decides how well the model can learn complicated things. By changing this, we make sure the model is not too simple or too complex. By picking and changing these hyperparameters well, we can make the model work better and get better scores on tests.

4) **Increased Training Epochs**: Adding more training epochs lets the model learn better from the data. We must watch how the model does on a different set to stop it from overfitting. If we train the model longer, it learns more complex patterns, and this makes it work better with new data. Using these better ways, we got our CNN model to work much better at sorting images on the CIFAR-10 dataset. Every method makes the model stronger and more right for sorting images in the real world.

Conclusions: -

I have used CNNs to label images from the CIFAR-10 set. We used ResNet-18, a deep CNN model which was very good for labelling images. I changed the data by normalization and transforms. This made our training data more varied and better and then we fine-tuned the ResNet-18 by switching its last layer for the CIFAR-10 data.

Changing settings was key to making this model work better hence I searched for the best Hyper parameters like number of hidden layer nodes, strength of regularization, learning rate and use of batch normalization. I also used many GPUs at once to make training faster. I trained the model longer and increased the epochs from 10 to 50. The longer training and the best settings made the model much more accurate.

The CNN model got 86.44% accuracy after we had done multiple improvements to the model and this good score shows that our chosen model and ways of improving it work well for labelling images.

References: -

https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

https://pytorch.org/hub/pytorch_vision_resnet/

https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

https://arxiv.org/abs/1803.08494

https://medium.com/analytics-vidhya/deep-learning-basics-weight-decay-3c68eb4344e9

https://www.run.ai/guides/multi-gpu

I have also used AI tool for reference where relevant according to policy.