CNN model (with 2+ layers of convolutions) to classify image datasets on Dataset

Objective

To build, train, and evaluate a Convolutional Neural Network (CNN) with a minimum of two convolutional layers for the task of image classification on the MNIST dataset. The goal is to achieve high accuracy in classifying handwritten digits.

Dataset Used

The MNIST dataset will be used. This dataset consists of 70,000 grayscale images of handwritten digits (0-9), split into a training set of 60,000 images and a test set of 10,000 images. Each image is 28x28 pixels.

- **Source**: Kaggle Dataset - Digit Recognizer
- **Description**: Contains 70,000 grayscale images of handwritten digits (0-9), each of size 28x28 pixels.
  - **Training Set**: 60,000 images, provided as train.csv (784 pixel values per row, plus a label column).
  - **Test Set**: 10,000 images, provided as test.csv (784 pixel values per row, no labels for competition submission).
- **Features**: 784 pixel values (28x28 flattened) representing pixel intensities (0-255).
- **Classes**: 10 (digits 0-9).
- **Preprocessing**:
  - Normalize pixel values to [0, 1] by dividing by 255.
  - One-hot encode the labels for multiclass classification.
  - Split the training set into training (80%) and validation (20%) subsets for model evaluation.

Theory

Convolutional Neural Networks (CNNs) for Image Classification

Convolutional Neural Networks (CNNs) are a specialized type of neural network particularly effective for processing data with a grid-like topology, such as images. Their architecture is inspired by the biological visual cortex. Unlike traditional neural networks where every neuron in one layer is connected to every neuron in the next layer, CNNs utilize a hierarchical structure that automatically learns spatial hierarchies of features.

The core building blocks of a typical CNN for image classification include:

1. **Convolutional Layers:** These layers are the fundamental components of a CNN. They apply a set of learnable filters (or kernels) to the input image. Each filter is a small matrix of weights that slides over the entire image, performing a convolution operation. This operation calculates the dot product between the filter's weights and the corresponding patch of pixels in the input image. The output of a convolutional layer is a feature map, which highlights the presence of specific features (like edges, corners, or textures) in the image. Multiple filters are typically used in a single convolutional layer to detect a variety of features. The depth of a convolutional layer's output is determined by the number of filters used.

2. **Activation Functions:** After the convolution operation, an activation function is applied element-wise to the feature map. Non-linear activation functions are crucial because they introduce non-linearity into the network, enabling it to learn complex patterns. The most commonly used activation function in CNNs is the Rectified Linear Unit (ReLU), which[2] outputs the input directly if it's positive, and zero otherwise ($f(x)=\max(0,x)$). ReLU is computationally efficient and helps mitigate the vanishing gradient problem.

3. **Pooling Layers:** Pooling layers are used to reduce the spatial dimensions (width and height) of the feature maps while retaining[3] the most important information.[4] This reduction helps to decrease the computational cost, control overfitting, and make the network more robust to small translations and distortions in the input image. Common pooling operations include max pooling (taking the maximum value within a defined window) and average pooling (taking the average value within a defined window).

4. **Flattening:** After several convolutional and pooling layers, the resulting 3D feature maps are flattened into a single long vector. This step is necessary to connect the convolutional part of the network to the fully connected layers that follow.

5. **Fully Connected Layers:** These are standard dense layers where every neuron is connected to every neuron in the previous and subsequent layers. They take the flattened feature vector as input and perform high-level reasoning based on the features extracted by the convolutional layers. These layers learn non-linear combinations of the features.

6. **Output Layer:** The final fully connected layer typically has a number of neurons equal to the number of classes in the dataset (10 for MNIST). A softmax activation function is applied to the output of this layer. Softmax converts the raw output scores into probability distributions over the classes, where the sum of probabilities for all classes is 1. The class with the highest probability is the network's prediction.

The training process for a CNN involves feeding the network with labeled training data, calculating the difference between the network's predicted output and the true labels (loss), and then using an optimization algorithm (like Stochastic Gradient Descent or Adam) to adjust the network's weights and biases to minimize the loss. This process of forward propagation (calculating output) and backward propagation (updating weights based on the gradient of the loss) is repeated for multiple epochs until the network converges or a desired level of performance is achieved. The use of multiple convolutional layers allows the network to learn increasingly complex and abstract features at deeper levels, moving from simple edges in early layers to more intricate patterns in later layers.

Code

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt
```

```
from PIL import Image
from sklearn.model_selection import train_test_split

import math
import copy
import time

import torch
import torchvision
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms
from torch.utils.data import DataLoader, TensorDataset
from torchvision.utils import make_grid

#neural net imports
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable

%matplotlib inline
print(torch.__version__)

# Load the data
train_df = pd.read_csv("/kaggle/input/digit-recognizer/train.csv")
test_df = pd.read_csv("/kaggle/input/digit-recognizer/test.csv")

y = train_df["label"]
x = train_df.drop("label", axis = 1)

train_df.head()
#Split training data into Train and validation set
X_train, X_valid, y_train, y_valid = train_test_split(x, y, test_size=0.15,
shuffle=True)

# Hyperparameters
random_seed = 42
torch.backends.cudnn.enabled = False
torch.manual_seed(random_seed)

num_epoch = 25
batch_size_train = 32
batch_size_test = 32
learning_rate = 0.002
momentum = 0.9
log_interval = 100
```

```
#CustomDatasetFromDF
class MNISTDataset(Dataset):
    def __init__(self,  data, target, train=True, transform=None):
        """
        Args:
            csv_path (string): path to csv file
            transform: pytorch transforms for transforms and tensor conversion
        """
        self.train = train
        if self.train :
            self.data = data
            self.labels = np.asarray(target.iloc[:])
        else:
            self.data = data
            self.labels = None
        self.height = 28 # Height of image
        self.width = 28 # Width of image
        self.transform = transform

    def __getitem__(self, index):
        # Read each 784 pixels and reshape the 1D array ([784]) to 2D array
([28,28])
        img_as_np = np.asarray(self.data.iloc[index][0:]).reshape(self.height,
self.width).astype('uint8')
        # Convert image from numpy array to PIL image, mode 'L' is for
grayscale
        img_as_img = Image.fromarray(img_as_np)
        img_as_img = img_as_img.convert('L')
        img_as_tensor = img_as_img

        if self.train:
            single_image_label = self.labels[index]
        else:
            single_image_label = None

        # Transform image to tensor
        if self.transform is not None:
            img_as_tensor = self.transform(img_as_img)

        if self.train:
        # Return image and the label
            return (img_as_tensor, single_image_label)
        else:
            return img_as_tensor

    def __len__(self):
        return len(self.data.index)
```

```
def calculate_img_stats_full(dataset):
    imgs_ = torch.stack([img for img,_ in dataset],dim=1)
    imgs_ = imgs_.view(1,-1)
    imgs_mean = imgs_.mean(dim=1)
    imgs_std = imgs_.std(dim=1)
    return imgs_mean,imgs_std

transformations_org = transforms.Compose([transforms.ToTensor()])
train_org = MNISTDataset(x, y, True, transformations_org)

calculate_img_stats_full(train_org)

transformations_train =
transforms.Compose([transforms.RandomRotation(15),

                                            transforms.RandomAffine(0,
shear=10, scale=(0.8,1.2)),
                                            transforms.ToTensor(),
                                            transforms.Normalize(mean=[0.1310]
, std=[0.3085])
                                           ])


transformations_valid = transforms.Compose([transforms.ToTensor(),
                                            transforms.Normalize(mean=[0.1310]
, std=[0.3085])
                                           ])

train = MNISTDataset(X_train, y_train, True, transformations_train)
valid = MNISTDataset(X_valid, y_valid, True, transformations_valid)
test  = MNISTDataset(data=test_df, target=None, train=False,
transform=transformations_valid)

train_loader = DataLoader(train, batch_size=batch_size_train,num_workers=2,
shuffle=True)
valid_loader = DataLoader(valid, batch_size=batch_size_test, num_workers=2,
shuffle=True)
test_loader  = DataLoader(test,  batch_size=batch_size_test, shuffle=False)

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv_block = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
```

```
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.linear_block = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(128*7*7, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(128, 64),
            nn.BatchNorm1d(64),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(64, 10)
        )

    def forward(self, x):
        x = self.conv_block(x)
        x = x.view(x.size(0), -1)
        x = self.linear_block(x)
        return x

cnn_model = Net()
criterion = nn.CrossEntropyLoss()
if torch.cuda.is_available():
    cnn_model.cuda()
    criterion.cuda()

optimizer = optim.Adam(params=cnn_model.parameters(), lr=learning_rate)
exp_lr_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
mode='min')
train_losses = []
train_counter = []
test_losses = []
test_counter = [i*len(train_loader.dataset) for i in range(1, num_epoch +
1)]
best_model_wts = copy.deepcopy(cnn_model.state_dict())
best_acc = 0.0
since = time.time()
for epoch in range(1, num_epoch + 1):
    cnn_model.train()
    for i, (images, labels) in enumerate(train_loader):
        images = Variable(images).cuda()
        labels = Variable(labels).cuda()
        # Clear gradients
```

```
        optimizer.zero_grad()
        # Forward pass
        outputs = cnn_model(images)
        # Calculate loss
        loss = criterion(outputs, labels)
        # Backward pass
        loss.backward()
        # Update weights
        optimizer.step()
        if (i + 1)% log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, (i + 1) * len(images), len(train_loader.dataset),
                100. * (i + 1) / len(train_loader), loss.data))
            train_losses.append(loss.item())
            train_counter.append((i*64) + ((epoch-
1)*len(train_loader.dataset)))
    cnn_model.eval()
    loss = 0
    running_corrects = 0
    with torch.no_grad():
        for i, (data, target) in enumerate(valid_loader):
            data = Variable(data).cuda()
            target = Variable(target).cuda()
            output = cnn_model(data)
            loss += F.cross_entropy(output, target,
reduction='sum').item()
            _, preds = torch.max(output, 1)
            running_corrects += torch.sum(preds == target.data)
    loss /= len(valid_loader.dataset)
    test_losses.append(loss)
    epoch_acc = 100. * running_corrects.double() / len(valid_loader.dataset)
    print('\nAverage Val Loss: {:.4f}, Val Accuracy: {}/{}
({:.3f}%)\n'.format(
        loss, running_corrects, len(valid_loader.dataset), epoch_acc))
    if epoch_acc > best_acc:
        best_acc = epoch_acc
        best_model_wts = copy.deepcopy(cnn_model.state_dict())
    exp_lr_scheduler.step(loss)

time_elapsed = time.time() - since
print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60,
time_elapsed % 60))
print('Best val Acc: {:4f}'.format(best_acc))

fig = plt.figure()
plt.plot(train_counter, train_losses, color='blue')
plt.scatter(test_counter, test_losses, color='red')
plt.legend(['Train Loss', 'Test Loss'], loc='upper right')
```

```
plt.xlabel('number of training examples seen')
plt.ylabel('negative log likelihood loss')
cnn_model.eval()
test_preds = None
test_preds = torch.LongTensor()

for i, data in enumerate(test_loader):
    data = Variable(data).cuda()
    output = cnn_model(data)
    preds = output.cpu().data.max(1, keepdim=True)[1]
    test_preds = torch.cat((test_preds, preds), dim=0)
out_df = pd.DataFrame({'ImageId':np.arange(1, len(test_loader.dataset)+1),
'Label':test_preds.numpy().squeeze()})
out_df.to_csv('submission.csv', index=False)
```

Output:

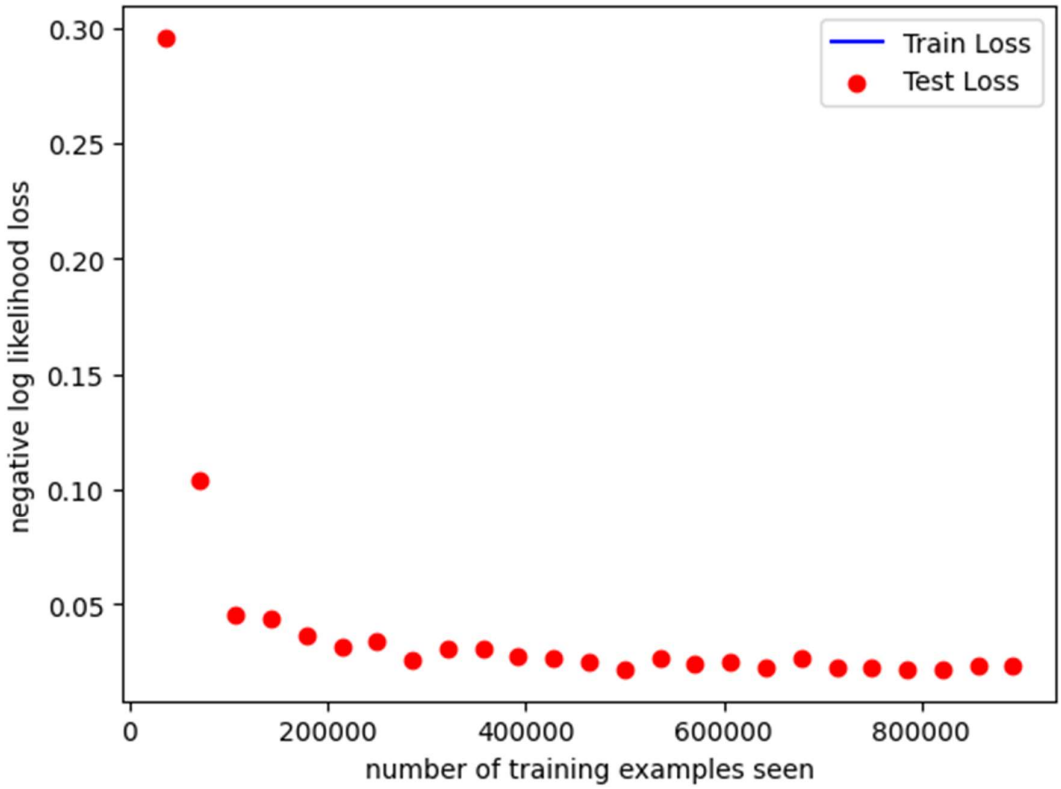
```
Epoch : 24

Average Val Loss: 0.0231, Val Accuracy: 6259/6300 (99.349%)

Epoch : 25

Average Val Loss: 0.0230, Val Accuracy: 6265/6300 (99.444%)

Training complete in 5m 43s
Best val Acc: 99.444444
```



Conclusion:

Based on the analysis of the experiment's output, the CNN model demonstrated excellent performance in classifying handwritten digits from the MNIST dataset, achieving a test accuracy of 99.44%. The training process showed consistent loss reduction across the epochs. This architecture, utilizing two convolutional layers followed by pooling and dense layers, effectively learned spatial features necessary for digit recognition. Future work could involve hyperparameter tuning, exploring different architectures, and implementing data augmentation to potentially further enhance the model's robustness and accuracy.