Import Modules

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import Dataset
from torch.utils.data import DataLoader

import matplotlib.pyplot as plt
import numpy as np
```

Dataset Class and Loader

In [2]:
```
'''
Dataset class given by the teaching team that represents the notMNIST datase
Contains length and getter functions to retrieve number of examples and exam
label pairs.
'''
class notMNIST(Dataset):
    def __init__(self, annotations, images, transform=None, target_transform
        self.img_labels = annotations
        self.imgs = images
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        image = self.imgs[idx]
        label = self.img_labels[idx]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

In [3]:
```
'''
Dataset-loading function given by teaching team that loads all data from
notMNIST and splits into training, validation and testing splits.
'''
def loadData(datafile = "notMNIST.npz"):
    with np.load(datafile) as data:
        Data, Target = data["images"].astype(np.float32), data["labels"]
        np.random.seed(521)
        randIndx = np.arange(len(Data))
        np.random.shuffle(randIndx)
        Data = Data[randIndx] / 255.0
        Target = Target[randIndx]
        trainData, trainTarget = Data[:10000], Target[:10000]
        validData, validTarget = Data[10000:16000], Target[10000:16000]
        testData, testTarget = Data[16000:], Target[16000:]
    return trainData, validData, testData, trainTarget, validTarget, testTar
```

Part 1: Fully Connected Network

In [4]:
```
'''
Class defintion for the Fully Connected Network used in Experiment 1
'''
class FNN(nn.Module):
    def __init__(self, drop_out_p=0.0):
        super(FNN, self).__init__()
        self.input_layer = nn.Flatten()
        self.fc1 = nn.Linear(784, 10)
        self.fc2 = nn.Linear(10, 10)
        self.dropout = nn.Dropout(p=drop_out_p)
        self.fc3 = nn.Linear(10, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.input_layer(x)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return x
```

Part 2: Convolutional Neural Networks

In [5]:
```
'''
Class definition for Convolutional Neural Network used in all three experime
'''
class CNN(nn.Module):
    def __init__(self, drop_out_p=0.0):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=4
        self.bn1 = nn.BatchNorm2d(num_features=32)
        self.pool1 = nn.MaxPool2d(kernel_size=2)

        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=
        self.bn2 = nn.BatchNorm2d(num_features=64)
        self.pool2 = nn.MaxPool2d(kernel_size=2)

        self.fc1 = nn.Flatten()
        self.dropout = nn.Dropout(p=drop_out_p)
        self.linear = nn.Linear(1024, 784)

        self.fc2 = nn.Linear(784, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.bn1(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = self.relu(x)
        x = self.bn2(x)
        x = self.pool2(x)

        x = self.fc1(x)
        x = self.dropout(x)
        x = self.linear(x)
        x = self.relu(x)

        x = self.fc2(x)
        # Note that we do not perform softmax here.
        # This is because cross-entropy loss does that by itself.
        return x
```

Part 3: Model Training and Experiments

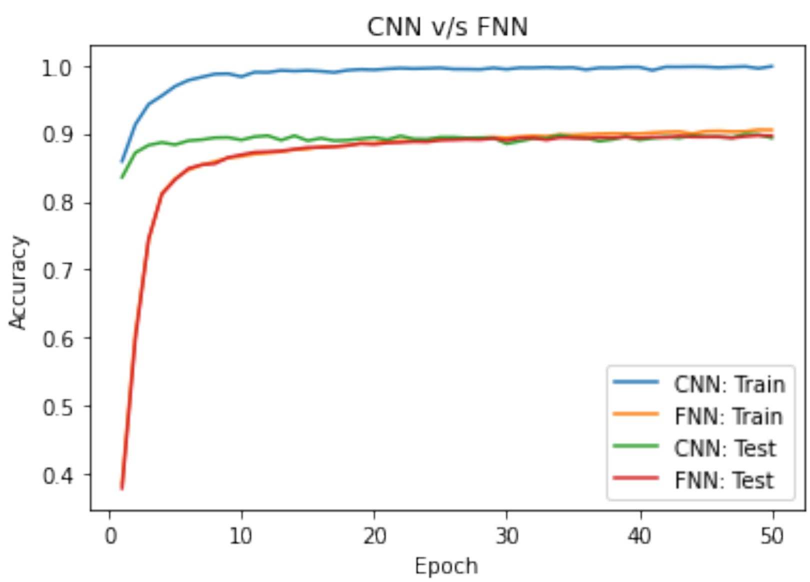
In [6]:
```
'''
Function to get predicition accuracy of given model on any dataset split
'''
def get_accuracy(model, dataloader):
    # De-activate batch-normaliszation layers and set to evaluation mode.
    model.eval()
    device = next(model.parameters()).device

    # Initialize integers for correct predicitons and total samples.
    correct = 0.0
    total = 0.0

    # Turn-off gradient calculation and back-propagation not needed.
    with torch.no_grad():
        for data in dataloader:
            # Get examples and true labels, and convert to device
            images, labels = data
            images = images.to(device)
            labels = labels.to(device)

            # Get predictions over batch
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)

            # Sum list of correct/incorrect predictions
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

    # Return ratio of correct-to-total as accuracy
    return correct/total
```

In [7]:
```
'''
Model training code for given model, device, lr, epochs and weight_decay
'''
def train(model, device, learning_rate, weight_decay, train_loader, val_load
    # Define loss function as Cross-Entropy Loss (Does softmax within)
    criterion = nn.CrossEntropyLoss()

    # Define optimizer as Adam Optimizer
    optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate, weig

    # Declare dictionary to maintain all accuracies for all epochs
    acc_hist = {'train':[], 'val':[], 'test': []}

    # Perform each epoch
    for epoch in range(num_epochs):
        # Set the model to training mode, as it was changed to eval earlier
        model = model.train()

        # Train over one batch
        for i, (images, labels) in enumerate(train_loader):
            # Convert examples and labels to device
            images = images.to(device)
            labels = labels.to(device)

            # Obtain predictions and calculate loss
            predictions = model(images)
            loss = criterion(predictions, labels)

            # Calculate gradients, propagate backwards and update weights
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        # Set model to eval mode to get its accuracies
        model.eval()
        acc_hist['train'].append(get_accuracy(model, train_loader))
        acc_hist['val'].append(get_accuracy(model, val_loader))
        acc_hist['test'].append(get_accuracy(model, test_loader))

        # Print accuracies if verbosity is on
        if verbose:
            print('Epoch: %d | Train Accuracy: %.2f | Validation Accuracy: %
                  %(epoch, acc_hist['train'][-1], acc_hist['val'][-1], acc_h

    return model, acc_hist
```

In [18]:
```
'''
Function to perform experiment on given type of model and specification i.e.
dropout_rate and weight_decay
'''
def experiment(model_type='CNN', learning_rate=0.0001, dropout_rate=0.5, wei
    # Get currently available device
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    BATCH_SIZE = 32

    # Transform incoming data to tensors and load dataset split
    transform = transforms.Compose([transforms.ToTensor()])
    trainData, validData, testData, trainTarget, validTarget, testTarget = l

    # Apply above transform to each dataset split and create object of the c
    train_data = notMNIST(trainTarget, trainData, transform=transform)
    val_data = notMNIST(validTarget, validData, transform=transform)
    test_data = notMNIST(testTarget, testData, transform=transform)

    # Create data loaders of given batch-size
    train_loader = torch.utils.data.DataLoader(train_data, batch_size=BATCH_
    val_loader = torch.utils.data.DataLoader(val_data, batch_size=BATCH_SIZE
    test_loader = torch.utils.data.DataLoader(test_data, batch_size=BATCH_SI

    # Create a new model instance based on type of model being experimented
    if model_type == 'CNN':
      model = CNN(dropout_rate)
    elif model_type == 'FNN':
      model = FNN(dropout_rate)

    # Convert the model to given device type and train it
    model = model.to(device)
    criterion = nn.CrossEntropyLoss()
    model, acc_hist = train(model, device, learning_rate, weight_decay, trai
    model.cpu()

    return model, acc_hist
```

Experiment 1

In [19]:
```
'''
Function to facilitate Experiment 1 to compare performace of CNN v/s FNN
'''
def compare_arch():
    # Get accuracies of CNN and FNN keeping all parameters same
    _, cnn = experiment(model_type="CNN", learning_rate=0.0001, dropout_rate
    _, fnn = experiment(model_type="FNN", learning_rate=0.0001, dropout_rate

    # Plot the training and testing accuracies for both models
    plt.title("Accuracies")
    plt.title("CNN v/s FNN")
    num_epochs = len(cnn["train"])
    plt.plot(range(1,num_epochs+1), cnn["train"], label="CNN: Train")
    plt.plot(range(1,num_epochs+1), fnn["train"], label="FNN: Train")
    plt.plot(range(1,num_epochs+1), cnn["test"], label="CNN: Test")
    plt.plot(range(1,num_epochs+1), fnn["test"], label="FNN: Test")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()

compare_arch()
```



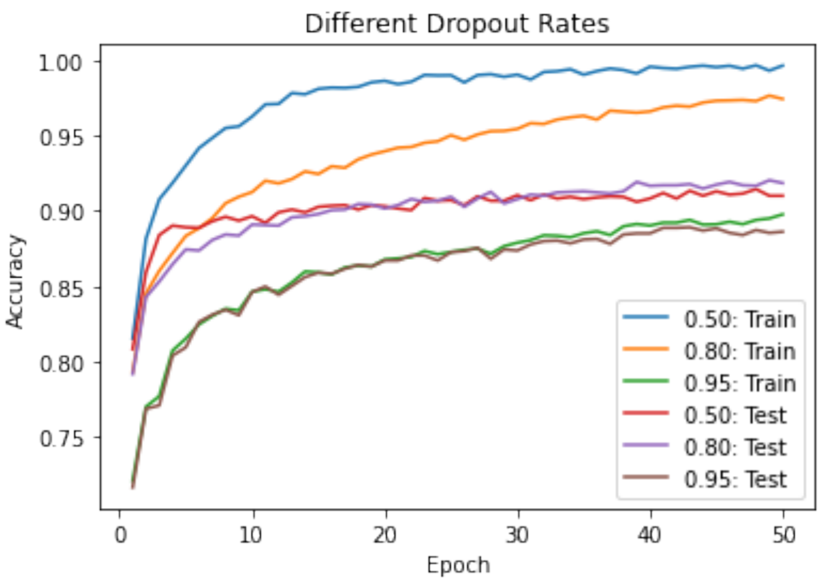
The FNN's training and testing accuracies go hand-in-hand meaning the model is not over-fitting and learning the data well, however, it can only reach a peak of around 87 percent. Looking at the CNN model, we see that it quickly learns the available data i.e. at very early number of epochs, and provides steadily good testing accuracy.

These point to the fact that CNNs perform image-related tasks much better than fully connected layers.

Experiment 2

In [16]:

```
'''
Function to facilitate Experiment 2 to compare performace of different dropc
rates in a CNN model
'''
def compare_dropout():
    # Get accuracies of CNN and FNN keeping all parameters same
    _, dropout1 = experiment(model_type="CNN", learning_rate=0.0001, dropout
    _, dropout2 = experiment(model_type="CNN", learning_rate=0.0001, dropout
    _, dropout3 = experiment(model_type="CNN", learning_rate=0.0001, dropout

    # Plot the training and testing accuracies for all three values of dropc
    plt.title("Training Accuracies")
    plt.title("Different Dropout Rates")
    num_epochs = len(dropout1["train"])
    plt.plot(range(1,num_epochs+1), dropout1["train"], label="0.50: Train")
    plt.plot(range(1,num_epochs+1), dropout2["train"], label="0.80: Train")
    plt.plot(range(1,num_epochs+1), dropout3["train"], label="0.95: Train")
    plt.plot(range(1,num_epochs+1), dropout1["test"], label="0.50: Test")
    plt.plot(range(1,num_epochs+1), dropout2["test"], label="0.80: Test")
    plt.plot(range(1,num_epochs+1), dropout3["test"], label="0.95: Test")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()

compare_dropout()
```



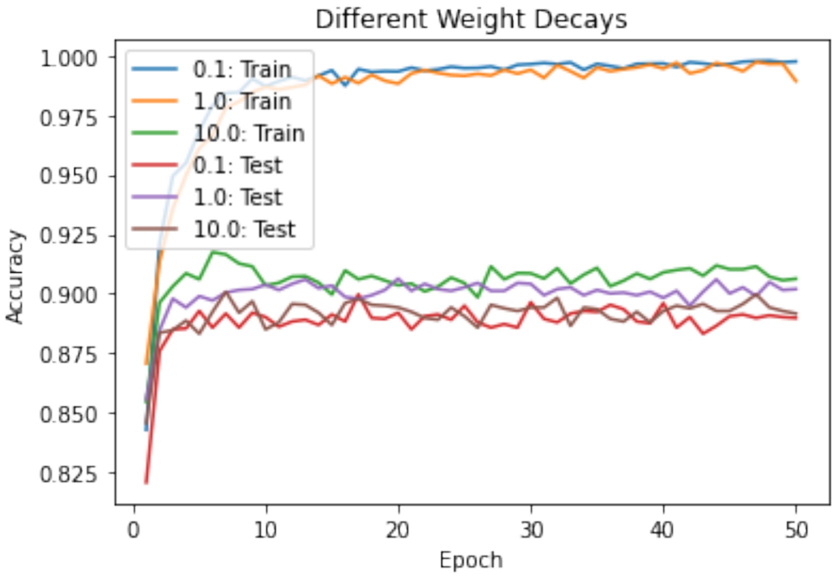
The general purpose of dropout is to prevent over-fitting on the training data by ignoring randomly selected neurons with a certain probability, during training of the model.

Keeping that in mind, the higher dropout rates i.e. 0.95 result in the model saturating at a low training accuracy indicating in-sufficient training. Coming to dropout rate of 0.50, we see better accuracy and finer curve that saturated very early on.

The most interesting results come from droput rate of 0.80 in which we see that the model has not saturated i.e. there is room for training while the testing accuracy are already similar to those obtained in the previous model. The testing accuracy might also increase very slightly if trained for more number of epochs.

Experiment 3

In [17]:
```
'''
Function to facilitate Experiment 3 to compare performace of different weigh
decay in a CNN model
'''
def compare_l2():
    # Get accuracies of CNN and FNN keeping all parameters same
    _, weightdecay1 = experiment(model_type="CNN", learning_rate=0.0001, drc
    _, weightdecay2 = experiment(model_type="CNN", learning_rate=0.0001, drc
    _, weightdecay3 = experiment(model_type="CNN", learning_rate=0.0001, drc

    # Plot the training and testing accuracies for all three values of dropc
    plt.title("Training Accuracies")
    plt.title("Different Weight Decays")
    num_epochs = len(weightdecay1["train"])
    plt.plot(range(1,num_epochs+1), weightdecay1["train"], label="0.1: Train
    plt.plot(range(1,num_epochs+1), weightdecay2["train"], label="1.0: Train
    plt.plot(range(1,num_epochs+1), weightdecay3["train"], label="10.0: Trai
    plt.plot(range(1,num_epochs+1), weightdecay1["test"], label="0.1: Test")
    plt.plot(range(1,num_epochs+1), weightdecay2["test"], label="1.0: Test")
    plt.plot(range(1,num_epochs+1), weightdecay3["test"], label="10.0: Test"
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()

compare_l2()
```



Weight decay is another regularization technique that prevents model weights from blowing up, and forcing the model to learn from smaller weight values. From the plot, we notice that using a weight decay parameter of 10 is too much as the model gets stuck at a very low training accuracy. For the other two values, training accuracies saturate very early on.

Looking at the testing accuracies, results for all three values fluctuate a lot. Accuracies for values 0.1 and 10 are generally lower as compared to those for weight decay of 1.0.