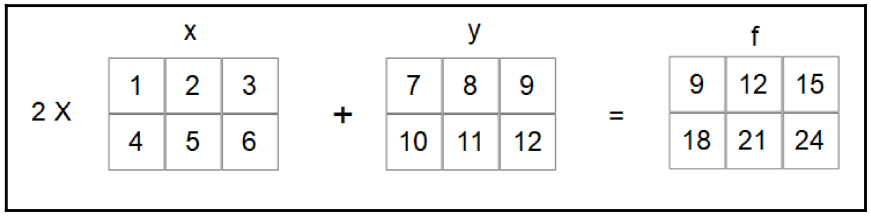
Chapter 1: Introduction to PyTorch

In [5]:

```
1 import torch
2
3 x= torch.tensor([[1,2,3],[4,5,6]])
4 y= torch.tensor([[7,8,9], [10,11,12]])
5
6 f= 2*x + y
7 print(f)
```

```
tensor([[  9,  12,  15],
        [ 18,  21,  24]])
```




```
1 shape=[2,3]
2 xzeros =torch.zeros(shape)
3 xones = torch.ones(shape)
4 xrnd = torch.rand(shape)
5 print(xzeros)
6 print(xones)
7 print(xrnd)
```

```
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])
tensor([[ 0.7104,  0.9464,  0.7890],
        [ 0.2814,  0.7886,  0.5895]])
```

```
1 torch.manual_seed(42)
2 print(torch.rand([2,3]))
```

```
tensor([[ 0.8823,  0.9150,  0.3829],
        [ 0.9593,  0.3904,  0.6009]])
```

```
1 import numpy as np
2
3 xnp= np.array([[1,2,3],[4,5,6]])
4 f2= xnp + y
5 print(f2)
6 f2.type()
```

```
tensor([[  8,  10,  12],
        [ 14,  16,  18]])

'torch.LongTensor'
```

```
1 xtensor = torch.from_numpy(xnp)
2 print(xtensor)
3 print(xtensor.type())
4
```

```
tensor([[ 1,  2,  3],
        [ 4,  5,  6]])
torch.LongTensor
```

```
1 print(f.type()) #call the tensors type method
2 fnp = f.numpy() #create aan array from the tensor
3 type(fnp) #uses the python inbuilt type()
```

```
torch.LongTensor

numpy.ndarray
```

In [80]:

```
a = np.ones(3)
t = torch.from_numpy(a)      #create a tensor from an array
b = t.numpy()                #Create an array from the tensor
b[1] = 0                     #change a value in the array
print(a[1] == b[1])          #this value changes in the original array
print(t)                     # and also in the tensor - they share the same memory
```

```
True
tensor([ 1.,  0.,  1.], dtype=torch.float64)
```

```
int8np=np.ones((2,3),dtype=np.int8)
bad= torch.from_numpy(int8np)
```

```
good = torch.from_numpy(int8np.astype(np.int32))
good.type()
```

```
'torch.IntTensor'
```

```
xint=torch.ones((2,3), dtype=torch.int)
xint.type()
```

```
'torch.IntTensor'
```

:

```
print(x[0])
print(x[1][0:2])
```

```
[1 2 3]
[4 5]
```

```
print(x.view(-1))
print(x.view(3,2))
print(x.view(6,1))
```

```
tensor([ 1,  2,  3,  4,  5,  6])
tensor([[ 1,  2],
        [ 3,  4],
        [ 5,  6]])
tensor([[ 1],
        [ 2],
        [ 3],
        [ 4],
        [ 5],
        [ 6]])
```

```
print(x.view(3,-1))
```

```
tensor([[ 1,  2],
        [ 3,  4],
        [ 5,  6]])
```

```
print(x.transpose(0,1))
```

```
tensor([[ 1,  4],
        [ 2,  5],
        [ 3,  6]])
```

```
a = torch.ones(1,2,3,4)
print(a.transpose(0,3).transpose(1,2).size())    #swaps axis in two steps
print(a.permute(3,2,1,0).size())                 #swaps all axis at once
```

```
torch.Size([4, 3, 2, 1])
torch.Size([4, 3, 2, 1])
```

```
print(x)
x.transpose_(1,0)
print(x)
```

```
tensor([[ 1,  2,  3],
        [ 4,  5,  6]])
tensor([[ 1,  4],
        [ 2,  5],
        [ 3,  6]])
```

```
print(y)
y.add_(x*2)
print(y)
```

```
tensor([[  7,   8,   9],
        [ 10,  11,  12]])
tensor([[  9,  12,  15],
        [ 18,  21,  24]])
```

```
import torch
import torchvision
import torchvision.transforms as transforms

trainset = torchvision.datasets.CIFAR10(root='./data',   #data root directory
                                        #The training set
                                       download=True,   #checks if data has downloaded else does so
                                       transform = transforms.ToTensor())  #transforms to tensor
 trainset
```

```
Files already downloaded and verified

Dataset CIFAR10
    Number of datapoints: 50000
    Split: train
    Root Location: ./data
    Transforms (if any): ToTensor()
    Target Transforms (if any): None
```

```
for i in range(len(trainset)):
    print('size of image {} label {}'.format(trainset[i][0].size() , trainset[i][1]))
    if i>2: break
```

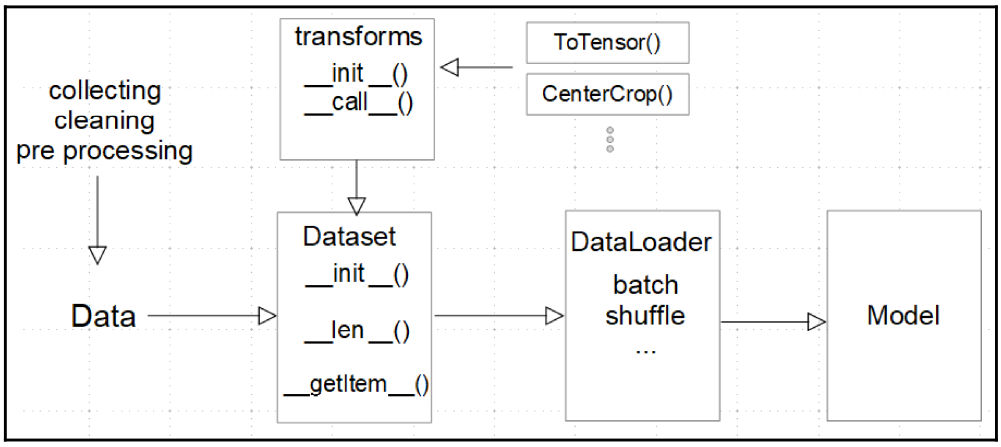
```
size of image torch.Size([3, 32, 32]) label 6
size of image torch.Size([3, 32, 32]) label 9
size of image torch.Size([3, 32, 32]) label 9
size of image torch.Size([3, 32, 32]) label 4
```

```
import matplotlib.pyplot as plt
%matplotlib inline

torchimage=trainset[0][0]    #Indexes the first element of the first tuple ie the 1st image
npimage=torchimage.permute(1, 2, 0) #changes the axis C H W to H W C
plt.imshow(npimage)    #plots the image - no need to convert to numpy

```

```
<matplotlib.image.AxesImage at 0x7f1fa7c594a8>
```

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True)
dataiter = iter(trainloader)          #create an iterator from a dataloader object
images, labels = dataiter.next()      #builds tensors for the images and labels in the batch
print(labels[0:])                      #prints the label of the images in the batch
print(images.size())                  #prints the size of the batch
```

```
tensor([ 6,  2,  0,  6])
torch.Size([4, 3, 32, 32])
```



```python
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
import torch
import csv
import os

class toyDataset(Dataset):

    def __init__(self, dataPath, labelsFile,  transform=None):

        self.dataPath=dataPath     #the path to the data directory
        self.transform = transform  #a transform object

        #builds a list of (name,label) tuples
        with open(os.path.join(self.dataPath,labelsFile)) as f:
            self.labels=[tuple(line) for line in csv.reader(f)]
        # checks that all images files exist
        for i in range(len(self.labels)):
             assert os.path.isfile(dataPath + '/' + self.labels[i][0])

    # so we can use dataset.len()
    def __len__(self):
        return len(self.labels)

    #so we can use indexing
    def __getitem__(self, idx):
        imageName,imageLabel=self.labels[idx][0:]
        imagePath =  os.path.join(self.dataPath,imageName)
        image = Image.open(open(imagePath, 'rb'))

        #transforms the image if required
        if self.transform:
            image = self.transform(image)
        return((image,imageLabel))
```

```python
toydata = toyDataset('data/GiuseppeToys','labels.csv', transform=transforms.ToTensor())
print(toydata[0][0].size()) #the size of the first image in the dataset
print(toydata[0][1]) #the label
```

```
torch.Size([3, 551, 816])
 toy
```

```python
tforms= transforms.Compose([transforms.Grayscale(3), transforms.CenterCrop(300), transforms.ToTensor()])
toyData=toyDataset('data/GiuseppeToys','labels.csv', transform =tforms)
```

```
toyloader = DataLoader(toyData, batch_size=4, shuffle=True)
toyiter= iter(toyloader)
images, labels = toyiter.next()
print(labels[0:])
print(images.size())
```

```
(' notoy ', ' toy ', ' toy', ' toy ')
torch.Size([4, 3, 500, 500])
```

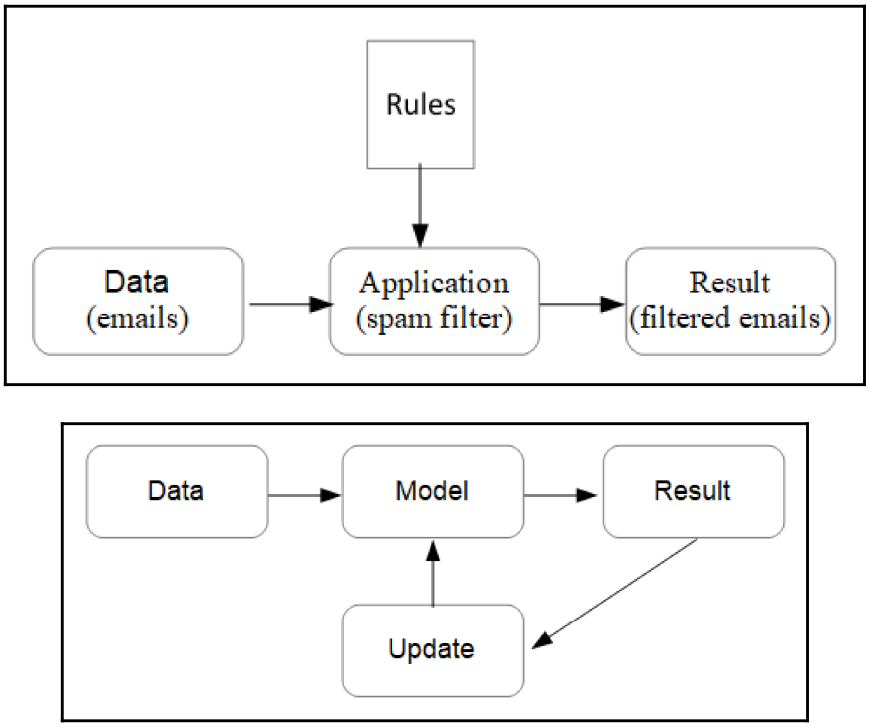
```
from torchvision import datasets
dataFromFolders = datasets.ImageFolder(root='data/GiuseppeToys/images', transform=tforms)
folderloader = DataLoader(dataFromFolders, batch_size=4, shuffle=True)
images, labels = iter(folderloader).next()
print(labels)
```

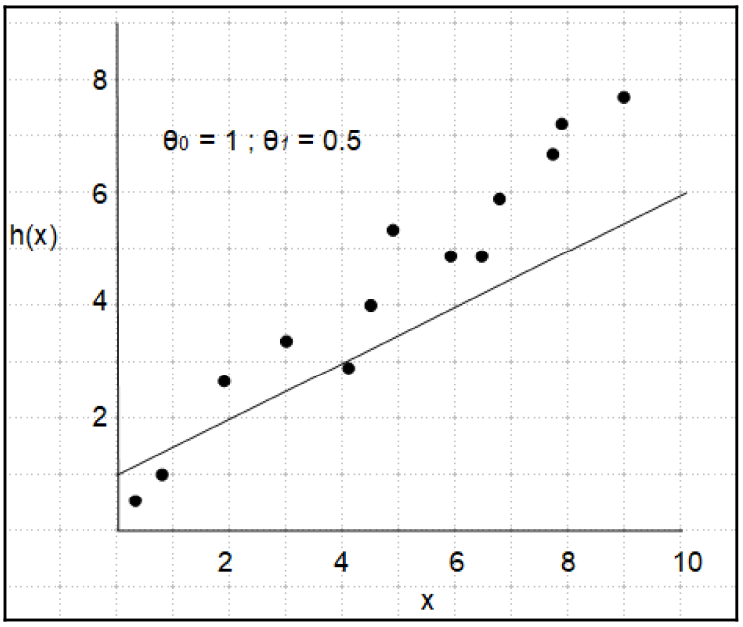
```
tensor([ 2,  2,  1,  3])
```

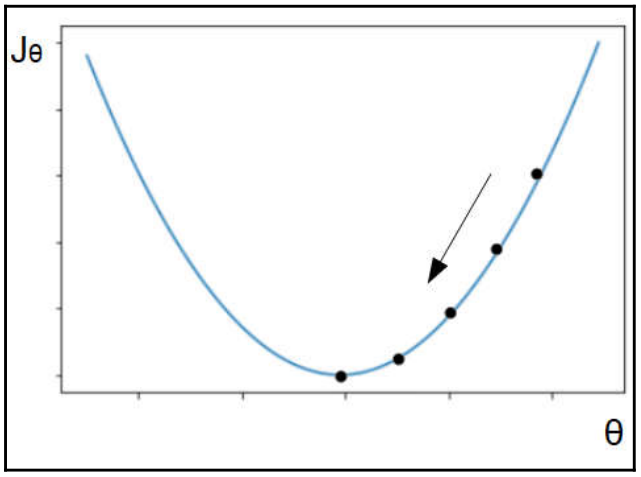
```
cc2,cc3=RemoveChannel('b'), RemoveChannel('g')
tforms2=transforms.Compose([transforms.CenterCrop(500), transforms.ToTensor(), cc2])
tforms3=transforms.Compose([transforms.CenterCrop(500), transforms.ToTensor(), cc2])
toydata2=toyDataset('data/GiuseppeToys','labels.csv', transform =tforms2, train=True)
toydata3=toyDataset('data/GiuseppeToys','labels.csv', transform =tforms3, train=True)
concatDataset= torch.utils.data.ConcatDataset([toydata,toydata2, toydata3])
len(concatDataset)
```

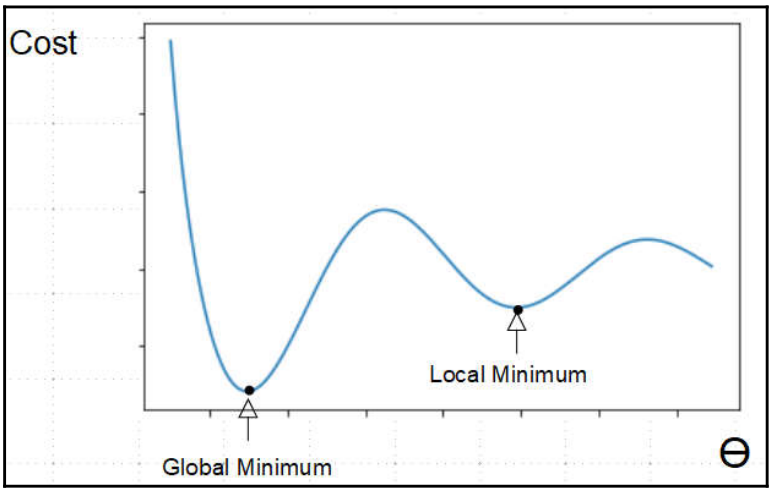
```
324
```

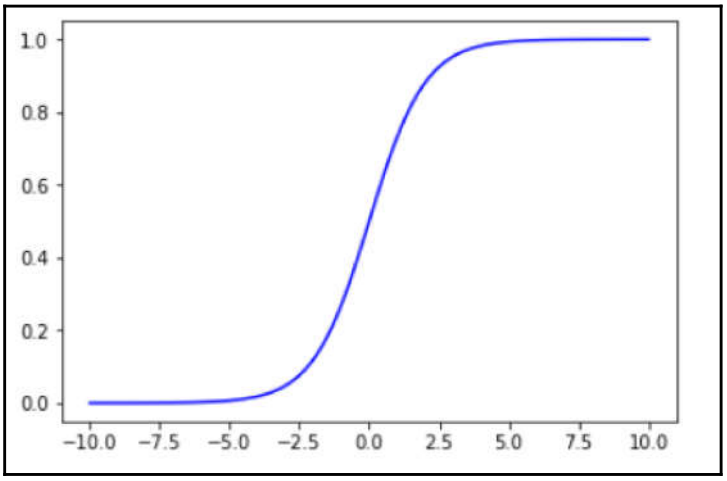
Chapter 2: Deep Learning Fundamentals

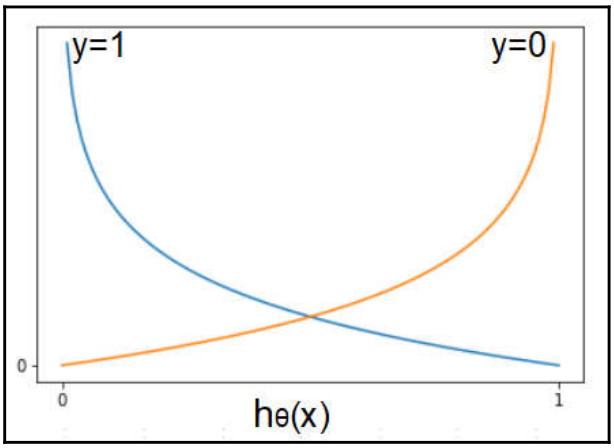



8
6
4
2
h(x)
θ0 = 1 ; θ1 = 0.5
2
4
6
8
10
x


Jθ
θ


Cost
Local Minimum
Global Minimum
Θ


1.0
0.8
0.6
0.4
0.2
0.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


y=1
y=0
0
0
1
hθ(x)

Chapter 3:
Computational Graphs and Linear Models

```
import torch
a= torch.tensor([[1,2,3],[4,5,6]], requires_grad=True, dtype=torch.float)
b= a +2
c=2*b*b
out=c.mean()
out.backward()
print(a.grad)
```

```
tensor([[ 2.0000,  2.6667,  3.3333],
        [ 4.0000,  4.6667,  5.3333]])
```

```
print(a.requires_grad)
with torch.no_grad():
    print((a**2).requires_grad)
```

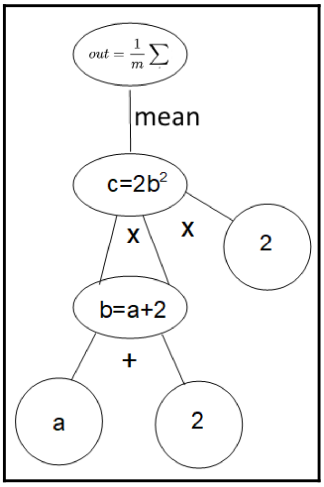
```
True
False
```

```
print(a.detach().requires_grad)
```

```
False
```

```python
a= torch.ones((2,3), requires_grad=True)
b= a +2
c=2*b*b
out=c.mean()
out.backward(retain_graph=True)
print(a.grad)
out.backward()
print(a.grad)
```

```
tensor([[ 2.,  2.,  2.],
        [ 2.,  2.,  2.]])
tensor([[ 4.,  4.,  4.],
        [ 4.,  4.,  4.]])
```




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

#Standard model class
class LinearModel(nn.Module):
    def __init__(self, in_dim, out_dim):
        super(LinearModel, self).__init__()
        self.linear = nn.Linear(in_dim, out_dim)

    def forward(self,x):
        out = self.linear(x)
        return out

model = LinearModel(1, 1)
```

```
learnRate = 0.01
optimiser =  torch.optim.SGD(model.parameters(), lr =learnRate)
criterion = nn.MSELoss()
```

```
x_train = torch.tensor([1,2,3,4,5,6,7,8,9,10], dtype=torch.float).reshape(-1,1)
y_train = torch.tensor([3*x+5 for x in x_train]).reshape(-1,1)
```

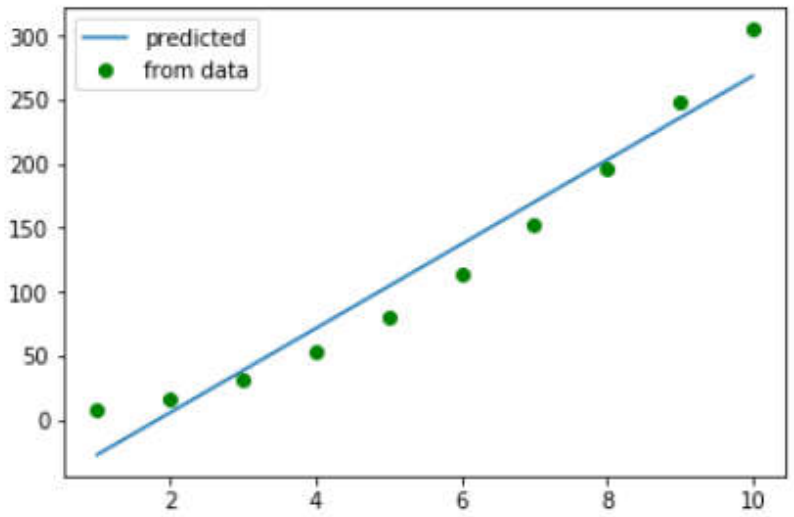
```
epochs = 1000
for epoch in range(epochs):
    epoch +=1
    inputs = x_train
    labels = y_train
    out = model(inputs)
    optimiser.zero_grad()
    loss = criterion(out, labels)
    loss.backward()
    optimiser.step()
    predicted = model.forward(x_train)
    print('epoch{}, loss {}'.format(epoch, loss.item()))

```

```
epoch1, loss 564.4509887695312
epoch2, loss 30.568267822265625
epoch3, loss 6.033814907073975
epoch4, loss 4.869321346282959
epoch5, loss 4.777401924133301
```

```
print(model.state_dict())
```

```
OrderedDict([('linear.weight', tensor([[ 3.0113]])), ('linear.bias', tensor([ 4.9210]))])
```

```python
import matplotlib.pyplot as plt
x = x_train.detach().numpy()
plt.plot(x, predicted.detach().numpy(), label = 'predicted')
plt.plot(x, y_train.detach().numpy(),'go', label = 'from data')
plt.legend()
plt.show()
```




```python
torch.save(model.state_dict(), 'testmodel.pkl')
```

```python
model = LinearModel(1,1)
model.load_state_dict(torch.load('testmodel.pkl'))
```

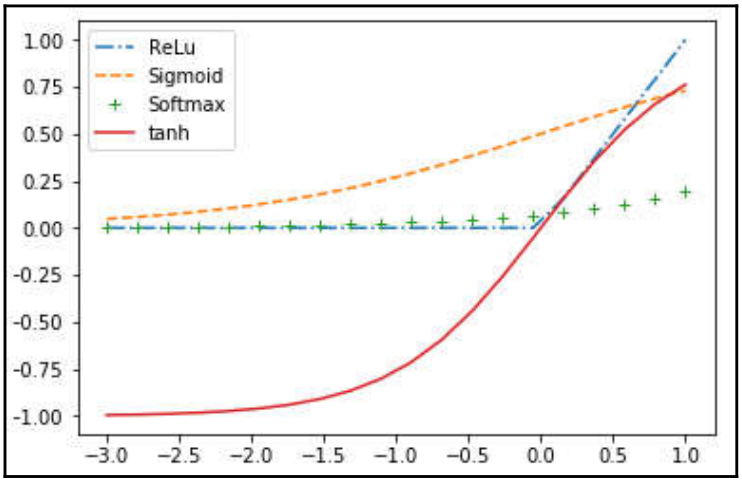
```python
import torch
import torch.nn as nn
import torch.nn.functional as func

class LogisticModel(nn.Module):
    def __init__(self, in_dim, out_dim):
        super(LogisticModel, self).__init__()
        self.linear = nn.Linear(in_dim, out_dim)

    def forward(self,x):
        out = func.sigmoid(self.linear(x))
        return out

model = LogisticModel(1, 1)
```

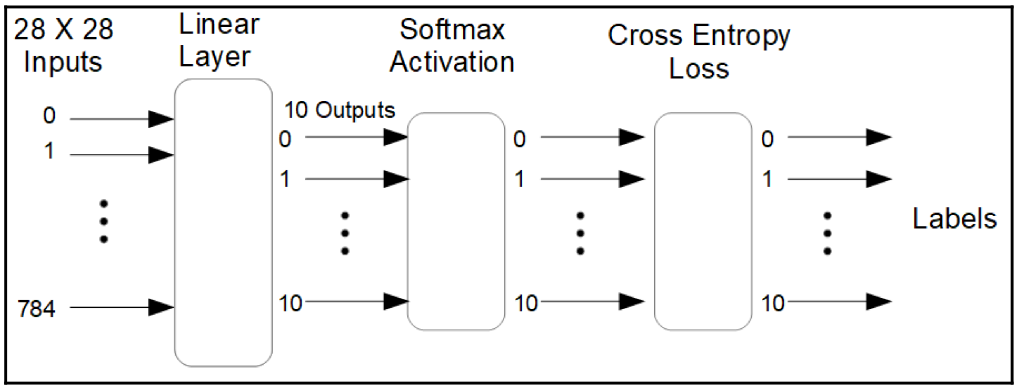
```
criterion = torch.nn.BCELoss(size_average=True)
optimiser =  torch.optim.SGD(model.parameters(), lr =0.01)
```

```
x_train = torch.tensor([[1.6],[2.1],[1.3],[4.8],[3.5]], dtype=torch.float).reshape(-1,1)
y_train = torch.tensor([[0],[0],[0],[1],[1]], dtype=torch.float).reshape(-1,1)
```

```
test=torch.tensor([[0.1],[1.5],[2.3],[3.0],[6.4]])
results = model(test)
for result in results:
    if result <= 0.5:
        print(result,'false')
    else: print(result, 'true')
```

```
tensor([ 0.3011]) false
tensor([ 0.4324]) false
tensor([ 0.5133]) true
tensor([ 0.5837]) true
tensor([ 0.8483]) true
```






```python
import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as trans

trainSet = dsets.MNIST(root='./data', train = True, transform=trans.ToTensor(), download=True)
```

```python
print('Number of images {}'.format(len(trainSet)))
print('Type {}'.format(type(trainSet[0][0])))
print('Size of each image {}'.format(trainSet[0][0].size()))
```

```
Number of images 60000
Type <class 'torch.Tensor'>
Size of each image torch.Size([1, 28, 28])
```

```python
class MultiLogisticModel(nn.Module):
    def __init__(self, in_dim, out_dim):
        super(MultiLogisticModel, self).__init__()
        self.linear = nn.Linear(in_dim, out_dim)

    def forward(self,x):
        out =self.linear(x)
        return out
```

```python
in_dim = 28*28 #input dimensions
out_dim = 10 #output dimensions
model = MultiLogisticModel(in_dim, out_dim) #instantiate model
criterion = nn.CrossEntropyLoss()  #instantiate the loss class
optimiser = torch.optim.SGD(model.parameters(), lr=.001) #instantiate the optimiser class
```

```
batchSize = 100
epochs = 5
trainloader = torch.utils.data.DataLoader(dataset=trainSet, batch_size=batchSize, shuffle = True)
```

```
for epoch in range(epochs):

    runningLoss = 0.0
    for i, (images,labels) in enumerate(trainloader):
        images = images.view(-1, 28 * 28)
        optimiser.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimiser.step()
        runningLoss += loss.item()

    print(runningLoss)
```

```
1213.2532706260681
968.1708756685257
809.9442014694214
704.4292406439781
630.6951693296432
```

```
predicted = model.forward(images)
predicted.size()
```

```
torch.Size([100, 10])
```

```
print('predictions {}'.format(predicted[0]))
print('labels {}'.format(labels[0]))
```

```
predictions tensor([-1.2102,  1.3957, -0.8418, -0.2278, -0.7412, -0.0017, -0.6341,
         0.9142,  0.1112,  0.7837])
labels 1
```

```
import numpy as np
def successRate(predicted, labels):
    predict = [np.argmax(p.detach().numpy()) for p in predicted]
    actual =  [labels[i].item() for i in range(len(predicted))]
    correct = [i for i, j in zip(predict, actual) if i == j]
    return(len(correct)/(len(predict)))
```

```
successRate(predicted, labels)
```

```
0.83
```

```
testSet = dsets.MNIST(root='./data', train = False, transform=trans.ToTensor(), download=True)
testloader = torch.utils.data.DataLoader(dataset=trainSet, batch_size = 10000, shuffle = True)

testData = iter(testloader)
images, labels = testData.next()
output = model(images.view(-1, 28 * 28))
successRate(output,labels)
```

```
0.8255
```

Chapter 4: Convolutional Networks

```python
import torch
import torch.nn as nn
class Model4_1(nn.Module):
    def __init__(self):
        super(Model4_1, self).__init__()
        self.lin1 = nn.Linear(784, 100)
        self.relu = nn.ReLU()
        self.lin2 = nn.Linear(100, 10)

    def forward(self, x):
        out = self.lin1(x)
        out = self.relu(out)
        out = self.lin2(out)
        return out

model4_1 = Model4_1()
```

```python
class Model4_2(nn.Module):
    def __init__(self):
        super(Model4_2, self).__init__()
        self.lin1 = nn.Linear(784, 100)
        self.tanh = nn.Tanh()
        self.lin2 = nn.Linear(100, 10)

    def forward(self, x):
        out = self.lin1(x)
        out = self.tanh(out)
        out = self.lin2(out)
        return out

model4_2 = Model4_2()
```

```python
import torch.optim as optim
import time

def benchmark(trainLoader, model, epochs=1, lr=0.01):
    model.__init__()
    start=time.time()
    optimiser = optim.SGD(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        for i, (images, labels) in enumerate(trainLoader):
            optimiser.zero_grad()
            outputs = model(images.view(-1, 28*28))
            loss = criterion(outputs, labels)
            loss.backward()
            optimiser.step()
    print('Accuracy: {0:.4f}'.format(accuracy(testLoader,model)))
    print('Training time: {0:.2f}'.format(time.time() - start))
```

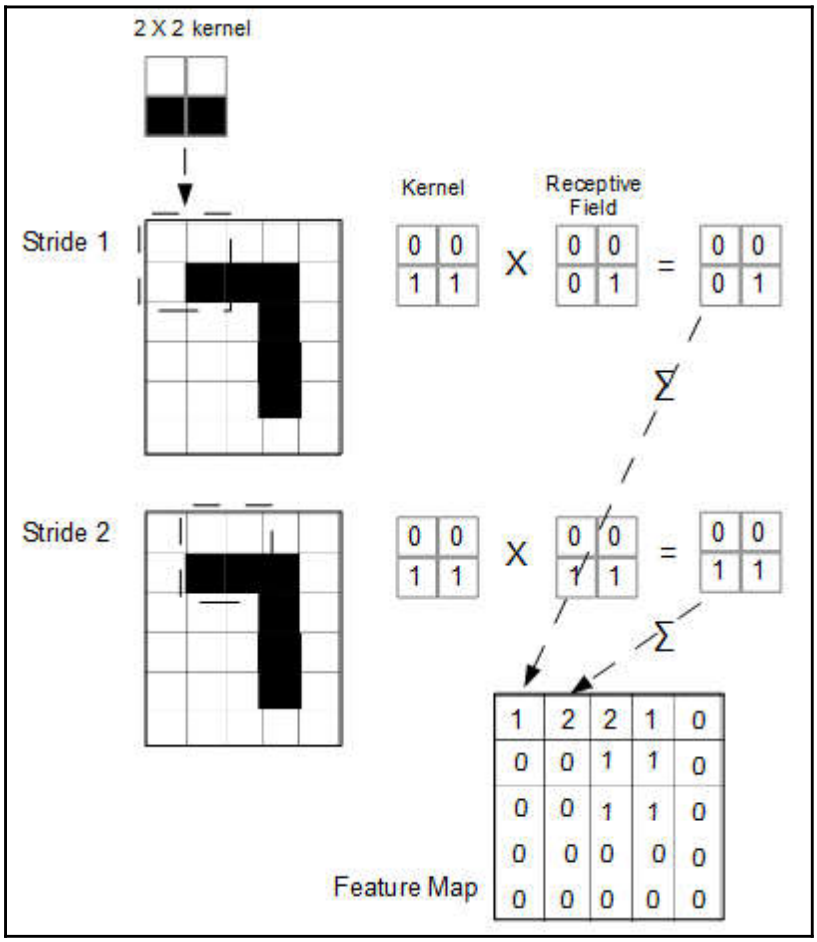
```python
def accuracy(testLoader,model):
    correct, total = 0, 0
    with torch.no_grad():
        for data in testLoader:
            images, labels = data
            outputs = model(images.view(-1, 28*28))
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return(correct / total)
```

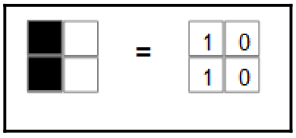
```python
print('ReLU actiavtion:')
benchmark(trainLoader, model4_1, epochs=5, lr = 0.1)
print('Tanh activation')
benchmark(trainLoader, model4_2, epochs=5, lr = 0.1)
print('sigmoid activation')
benchmark(trainLoader, model4_3, epochs=5, lr = 0.1)
```

```
ReLU actiavtion:
Accuracy: 0.9575
Training time: 36.56
Tanh activation
Accuracy: 0.9516
Training time: 39.04
sigmoid activation
Accuracy: 0.9199
Training time: 38.46
```

```python
class Model4_4(nn.Module):
    def __init__(self):
        super(Model4_4b, self).__init__()
        self.layer1=nn.Sequential(nn.Linear(784, 100),
            nn.ReLU())
        self.layer2=nn.Sequential(nn.Linear(100, 50),
            nn.ReLU(),
            nn.Linear(50, 10))

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        return out

model4_4b = Model4_4b()
```

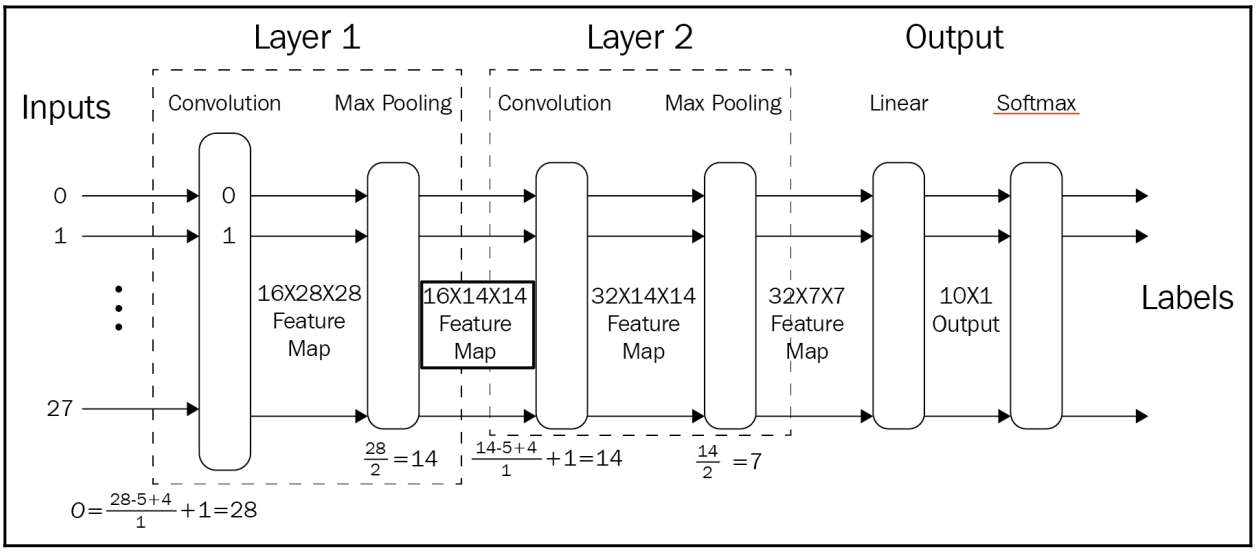
```
3 Linear layers 100 - 50 - 10:
Accuracy: 0.9667
Training time: 38.98
3 Linear layers 1000 -100 - 10:
Accuracy: 0.9724
Training time: 71.02
4 Linear layers 1000 - 500 -50 - 10 :
Accuracy: 0.9773
Training time: 90.86
```


2 X 2 kernel
Stride 1
Stride 2
Kernel
Receptive Field
0 0
1 1
X
0 0
0 1
=
0 0
0 1
Σ
0 0
1 1
X
0 0
1 1
=
0 0
1 1
Σ
1 2 2 1 0
0 0 1 1 0
0 0 1 1 0
0 0 0 0 0
0 0 0 0 0
Feature Map


=
1 0
1 0

```python
class CNNModel1(nn.Module):
    def __init__(self):
        super(CNNModel1, self).__init__()
        self.cnn1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=5, stride=1, padding=2)
        self.relu1 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)
        self.fc1 = nn.Linear(32* 14 * 14, 10)

    def forward(self, x):
        out = self.cnn1(x)
        out = self.relu1(out)
        out = self.maxpool1(out)
        out = out.view(out.size(0), -1)
        out = self.fc1(out)
        return out

model1=CNNModel1()
```

```python
class CNNModel2(nn.Module):
    def __init__(self):
        super(CNNModel2, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))

        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.lin1=nn.Linear(32* 7* 7, 10)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.view(out.size(0), -1)
        out = self.lin1(out)
        return out
model2=CNNModel2()
```



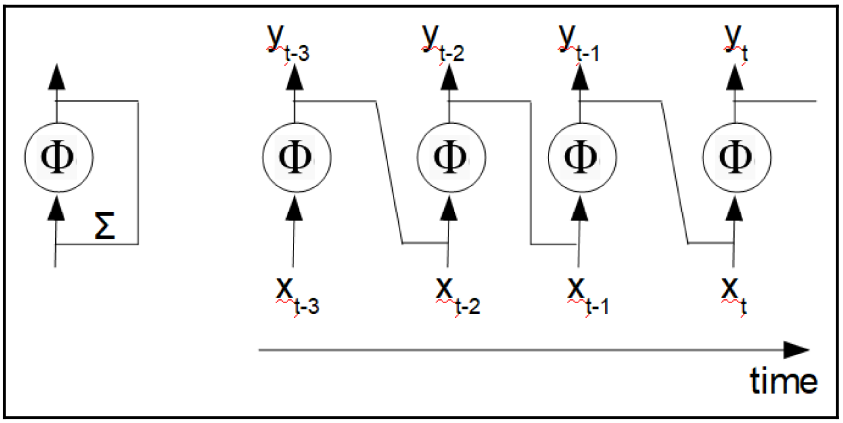

```
1 parameters=list((model.parameters()))
2 for parameter in parameters:
3     print(parameter.size())
```

```
torch.Size([16, 1, 5, 5])
torch.Size([16])
torch.Size([32, 16, 5, 5])
torch.Size([32])
torch.Size([10, 1568])
torch.Size([10])
```

```python
class CNNModel3(nn.Module):
    def __init__(self):
        super(CNNModel3, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))

        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.lin1=nn.Linear(32* 7* 7, 10)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.view(out.size(0), -1)
        out = self.lin1(out)
        return out
model3=CNNModel3()
```

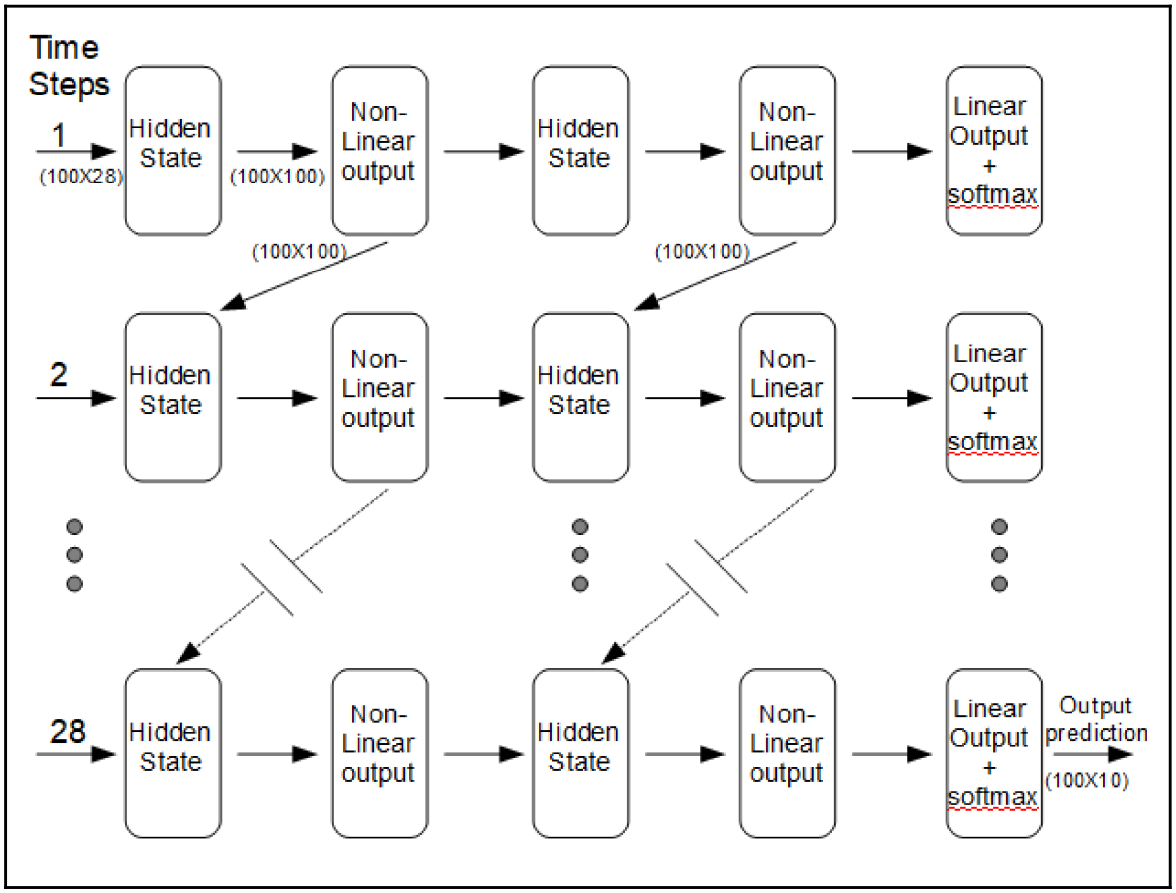
```
Single convolutional layer accuracy: 0.9343
Training time: 234.96
Two convolutional layers accuracy: 0.9676
Training time  model2: 386.19
Two convolutional layers with batchnorm accuracy: 0.9844
Training time  model3 :471.86
```

Chapter 5: Other NN Architectures




```python
class Model5_1(nn.Module):
    def __init__(self):
        inSize=28
        hiddenSize=100
        numLayers=2
        outSize = 10
        super(Model5_1, self).__init__()
        self.rnn = nn.RNN(inSize, hiddenSize, numLayers, batch_first=True)
        self.fc = nn.Linear(hiddenSize, outSize)

    def forward(self, x):
        h0 = torch.zeros(numLayers, x.size(0), hiddenSize)
        out, hn = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])
        return out

model5_1 = Model5_1()
```

```
2 layers, hidden size 100
Accuracy: 0.9473
Training time: 235.75

 2 layers, hidden size 200
Accuracy: 0.9675
Training time: 497.41

 3 layers, hidden size 200
Accuracy: 0.9717
Training time: 804.50
```



```
for i in range(len(list(model5_1.parameters()))):
                print(list(model5_1.parameters())[i].size())
```

```
torch.Size([100, 28])
torch.Size([100, 100])
torch.Size([100])
torch.Size([100])
torch.Size([100, 100])
torch.Size([100, 100])
torch.Size([100])
torch.Size([100])
torch.Size([10, 100])
torch.Size([10])
```

```
dataiter = iter(trainLoader)
images, labels = dataiter.next()
rnn = nn.RNN(28,100, 2, batch_first=True)
h0 = torch.zeros(2, images.size(0), 100)
output, hn = rnn(images.view(-1, 28,28), h0)
```

```
output.size()
```

```
torch.Size([100, 28, 100])
```

```
hn.size()
```

```
torch.Size([2, 100, 100])
```

```
images.size()
```

```
torch.Size([100, 1, 28, 28])
```

```
images.view(-1, 28,28).size()
```

```
torch.Size([100, 28, 28])
```

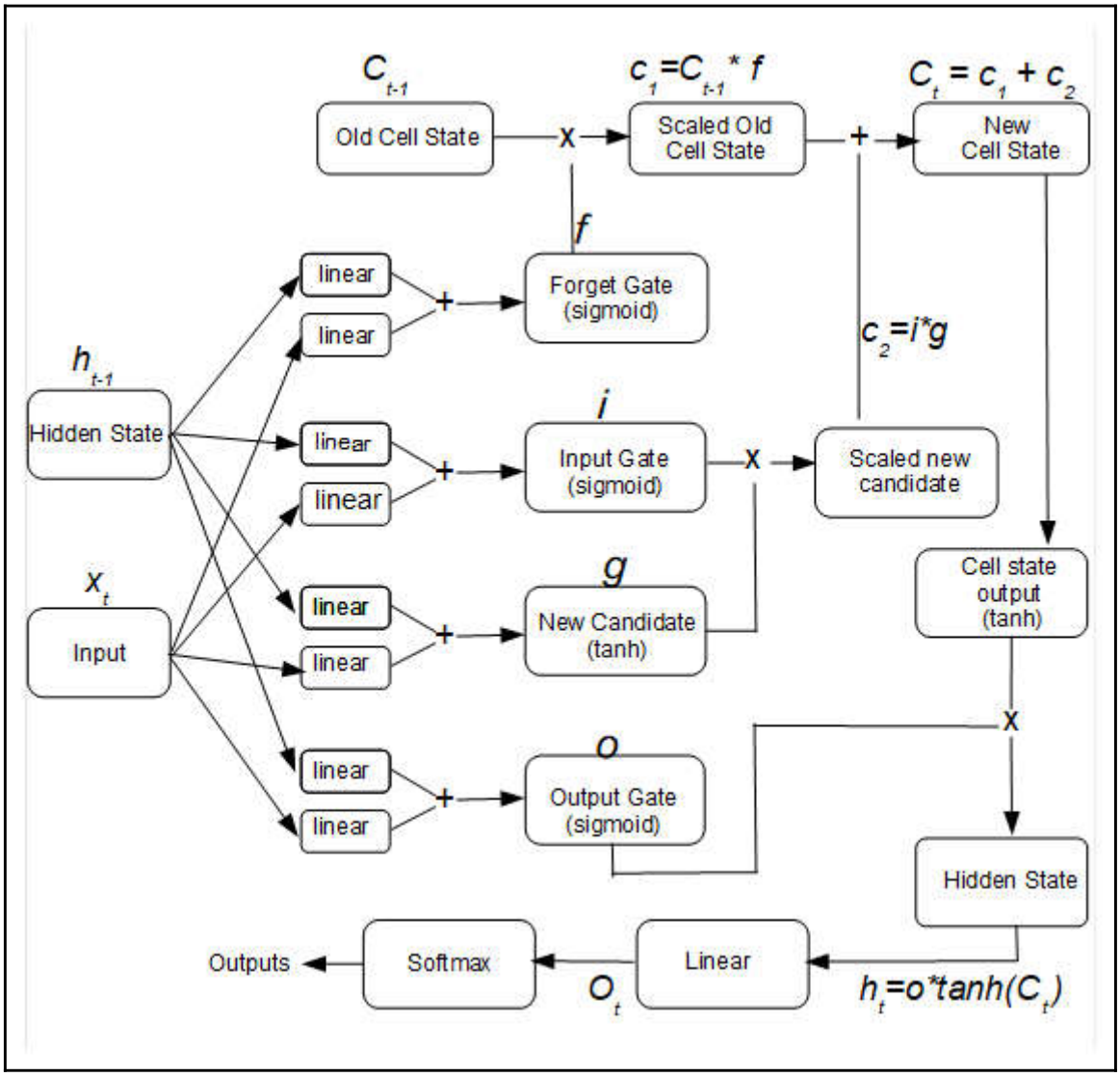
```
fc=nn.Linear(100,10)
output2=fc(output[:, -1, :])
```

```
output2.size()
```

```
torch.Size([100, 10])
```

```
1 output[:, -1, :].size()
```

```
torch.Size([100, 100])
```



```python
class Model5_3(nn.Module):
    def __init__(self):
        super(Model5_3, self).__init__()
        self.inSize=28
        self.hiddenSize=100
        self.numLayers=2
        self.outSize = 10
        self.lstm = nn.LSTM(self.inSize, self.hiddenSize, self.numLayers, batch_first=True)
        self.fc = nn.Linear(self.hiddenSize, self.outSize)

    def forward(self, x):
        h0 = torch.zeros(self.numLayers, x.size(0), self.hiddenSize)
        c0 = torch.zeros(self.numLayers, x.size(0), self.hiddenSize)
        out, (hn, cn) = self.lstm(x, (h0,c0))
        out = self.fc(out[:, -1, :])
        return out

model5_3 = Model5_3()
```

```python
for i in range(len(list(model5_2.parameters()))):
                print(list(model5_2.parameters())[i].size())
```

```
torch.Size([400, 28])
torch.Size([400, 100])
torch.Size([400])
torch.Size([400])
torch.Size([10, 100])
torch.Size([10])
```

```python
class Model15_2(nn.Module):
    def __init__(self, numCharacters):
        super(Model15_2, self).__init__()
        self.input_size = numCharacters
        self.hidden_size = 100
        self.output_size = numCharacters
        self.n_layers = 2

        self.encoder = nn.Embedding(numCharacters, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers)
        self.decoder = nn.Linear(hidden_size, numCharacters)

    def forward(self, input, hidden):
        input = self.encoder(input.view(1, -1))
        output, hidden = self.gru(input.view(1, 1, -1), hidden)
        output = self.decoder(output.view(1, -1))
        return output, hidden

    def init_hidden(self):
        return torch.zeros(self.n_layers, 1, self.hidden_size)
```

```python
allCharacters = string.printable
numCharacters = len(allCharacters)
file = unidecode.unidecode(open('data/warandpeace.txt').read())
chunk_len = 200

def char_tensor(string):
    tensor = torch.zeros(len(string)).long().unsqueeze(1)
    for c in range(len(string)):
        tensor[c] = allCharacters.index(string[c])
    return tensor

def random_training_set():
    start_index = random.randint(0, len(file) - chunk_len)
    end_index = start_index + chunk_len + 1
    chunk = file[start_index:end_index]
    inp = char_tensor(chunk[:-1])
    target = char_tensor(chunk[1:])
    return inp, target
```

```python
def evaluate(prime_str='A', predict_len=500, temperature=0.8):
    hidden = decoder.init_hidden()
    prime_input = char_tensor(prime_str)
    predicted = prime_str

    for p in range(len(prime_str) - 1):
        _, hidden = decoder(prime_input[p], hidden)
    inp = prime_input[-1]

    for p in range(predict_len):
        output, hidden = decoder(inp, hidden)
        output_dist = output.data.view(-1).div(temperature).exp()
        top_i = torch.multinomial(output_dist, 1)[0]
        predicted_char = allCharacters[top_i]
        predicted += predicted_char
        inp = char_tensor(predicted_char)
    return predicted
```

```python
def train(inp, target):
    hidden = decoder.init_hidden()
    decoder.zero_grad()
    loss = 0

    for c in range(chunk_len):
        output, hidden = decoder(inp[c], hidden)
        loss += criterion(output, target[c])

    loss.backward()
    decoder_optimizer.step()
    return loss.item() / chunk_len
```

```
n_epochs = 1000
print_every = 50
hidden_size = 100
n_layers = 2
lr = 0.01

decoder = Model5_2(numCharacters)
decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)
criterion = nn.CrossEntropyLoss()
loss_avg = 0

for epoch in range(1, n_epochs + 1):
    loss = train(*random_training_set())
    loss_avg += loss
    if epoch % print_every == 0:
        print('Epoch: {0:1d} Loss: {1:.4f}'.format(epoch, loss))
        print(evaluate('but', 100), '\n')
```

Chapter 6: Getting the Most out of PyTorch

```
import torch
w=torch.rand(3,3).to('cuda')
print(w)
```

```
tensor([[0.2629, 0.2429, 0.8316],
        [0.1465, 0.3592, 0.9654],
        [0.5141, 0.1318, 0.9772]], device='cuda:0')
```

```
for epoch in range(epochs):
    for i, (images, labels) in enumerate(trainLoader):
        images = images.requires_grad_().to(device)
        labels = labels.to(device)
        optimiser.zero_grad()
        outputs = model(images.view(-1, 28*28)).to(device)
```

```
lamb1 = lambda epoch: 0.9 ** epoch
lamb2 = lambda epoch: 0.8 ** epoch
schedular = optim.lr_scheduler.LambdaLR(optimizer,lr_lambda=[lamb1,lamb2], last_epoch =-1)
```

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

w1 = torch.randn(3, 3)
w1.requires_grad = True
optimizer = optim.SGD([w1], lr=0.1)
print(optimizer.param_groups)
```

```
[{'params': [tensor([[-1.2673,  1.5080, -0.2775],
        [-0.5443,  0.7693,  0.2868],
        [ 1.3169, -0.4790,  1.6926]])], 'lr': 0.1, 'momentum': 0, 'dampening': 0, 'weight_decay': 0, 'nesterov': False}]
```

```
w2 = torch.randn(3, 3)
w2.requires_grad = True
optimizer.add_param_group({'params': w2})
print(optimizer.param_groups)
```

```
[{'params': [tensor([[-1.2673,  1.5080, -0.2775],
        [-0.5443,  0.7693,  0.2868],
        [ 1.3169, -0.4790,  1.6926]])], 'lr': 0.1, 'momentum': 0, 'dampening': 0, 'weight_decay': 0, 'nesterov': False}, {'para
ms': [tensor([[ 0.1312,  0.0589, -0.1158],
        [ 1.0284,  0.7809,  0.0052],
        [-0.4003,  0.1439, -1.2612]])], 'lr': 0.1, 'momentum': 0, 'dampening': 0, 'weight_decay': 0, 'nesterov': False}]
```

```
optimizer.param_groups[1]['momentum'] = 0.9
print(optimizer.param_groups[1])
```

```
{'params': [tensor([[ 0.0743,  0.8700,  0.9258],
        [ 0.3472,  0.3480,  0.9883],
        [ 0.3408,  0.8535,  0.1347]])], 'lr': 0.1, 'momentum': 0.9, 'dampening': 0, 'weight_decay': 0, 'nesterov': False}
```

```
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

data_dir = 'toydata'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'val']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=4,
                                              shuffle=True)
               for x in ['train', 'val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
class_names = image_datasets['train'].classes

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
image_datasets['train'].classes
```

```
['NoToy', 'Scenes', 'SingleToy']
```

```
inputs, classes = next(iter(dataloaders['train']))
inputs.size()
```

```
torch.Size([4, 3, 224, 224])
```

```
[class_names[x] for x in classes]
```

```
['Scenes', 'SingleToy', 'SingleToy', 'SingleToy']
```

```
def train_model(model, criterion, optimizer, scheduler, num_epochs=1):
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        for phase in ['train', 'val']:
            if phase == 'train':
                scheduler.step()
                model.train()
            else:
                model.eval()
            running_loss = 0.0
            running_corrects = 0
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)
                optimizer.zero_grad()
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]
            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())
        print()
    print('Best val Acc: {:4f}'.format(best_acc))
    model.load_state_dict(best_model_wts)
    return model
```

```
model = models.resnet50(pretrained=True)
num_ftrs = model_ft.fc.in_features
model.fc = nn.Linear(num_ftrs, 3)
model = model_ft.to(device)
criterion = nn.CrossEntropyLoss()
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)
now=time.time()
model = train_model(model_ft, criterion, optimizer_ft, exp_lr_scheduler,
                       num_epochs=22)
```

```
num_ftrs = model_vgg.classifier[6].in_features
features = list(model_vgg.classifier.children())[:-1]
features.extend([nn.Linear(num_ftrs, 3)])
model_vgg.classifier = nn.Sequential(*features)
```