

Introduction

- Trains Convolutional Neural Network to classify images from the CIFAR-10 database.
- Each of the images (32x32x3) fall into one of ten classes

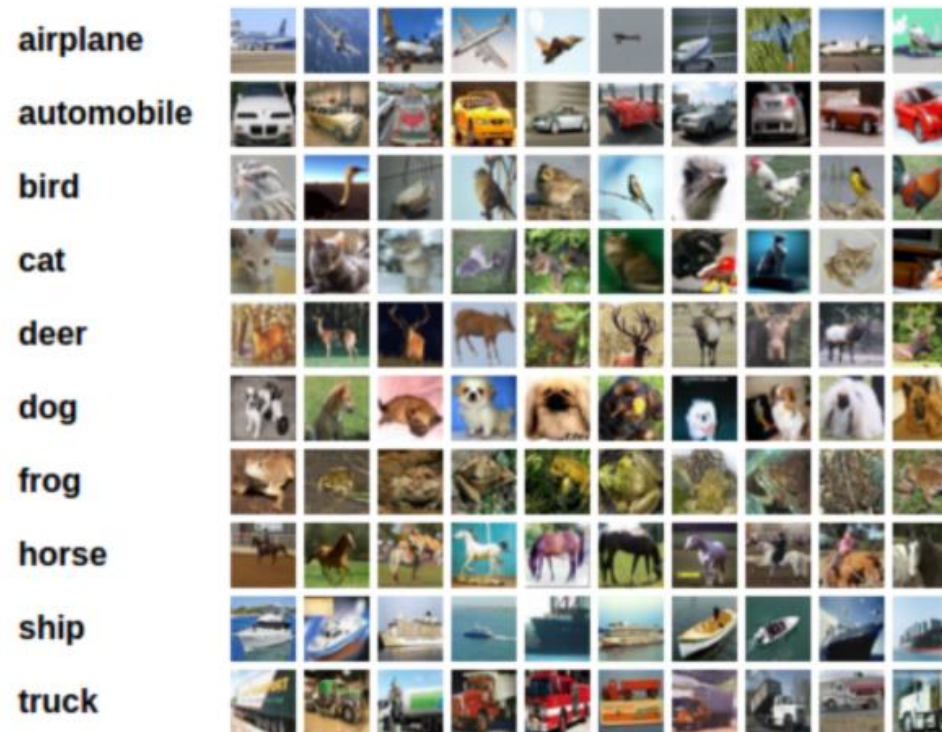


Fig. 1 : Images in this database

Overview

- Data preprocessing (Load and Augment)
- Visualization of a batch of training data
- Definition the network architecture
- Specification of loss and optimizer
- Training of the network
- Testing of the trained network
- Visualization of the sample test results

Data Preprocessing

Data Preprocessing : Augmentation

- Performs data augmentation
- Converts each image pixels into float tensor
- Normalizes the image pixels using mean and Standard deviation

```
transform = transforms.Compose([  
    transforms.RandomHorizontalFlip(),  
    transforms.RandomRotation(10),  
  
    transforms.ToTensor(),  
  
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  
])
```

 Data Augmentation

 Conversion

 Normalization (mean, SD)

Data Preprocessing: Downloading, Splitting

- Downloads data for training and testing

```
train_data = datasets.CIFAR10('data', train=True, download=True, transform=transform)
test_data = datasets.CIFAR10('data', train=False, download=True, transform=transform)
```

- Splits training data into train(80%) and validation set (20%)

```
num_train = len(train_data)
indices = list(range(num_train))
np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train)) # valid_size = 0.2
train_idx, valid_idx = indices[split:], indices[:split]
```

Data Preprocessing: Data Loader, Prediction

- Prepares DataLoader with batch size 20
- Takes random samples
- Defines classes

```
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
    sampler=train_sampler, num_workers=num_workers) # batch_size = 20, num_workers = 0
valid_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
    sampler=valid_sampler, num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
    num_workers=num_workers)
```

Visualization of a batch of training data

Visualization: Plotting after conversion

- Un-normalizes and converts from tensor images
- Displays one batch (20) input images from train dataset

```
# helper function to un-normalize and display an image
def imshow(img):
    img = img / 2 + 0.5 # unnormalize
    plt.imshow(np.transpose(img, (1, 2, 0))) # convert from Tensor image

# obtain one batch of training images
dataiter = iter(train_loader)
images, labels = dataiter.next()
images = images.numpy() # convert images to numpy for display

# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))

# display 20 images
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    imshow(images[idx])
    ax.set_title(classes[labels[idx]])
```


Visualization :displaying images



Defining Network Architecture

Defining Network Architecture: 1

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        #convolutional layer (sees 32x32x3 image tensor)
        #uses a kernel of 3X3 to slide over the images with padding 1


        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        # convolutional layer (sees 16x16x16 tensor)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        # convolutional layer (sees 8x8x32 tensor)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)

        self.pool = nn.MaxPool2d(2, 2)

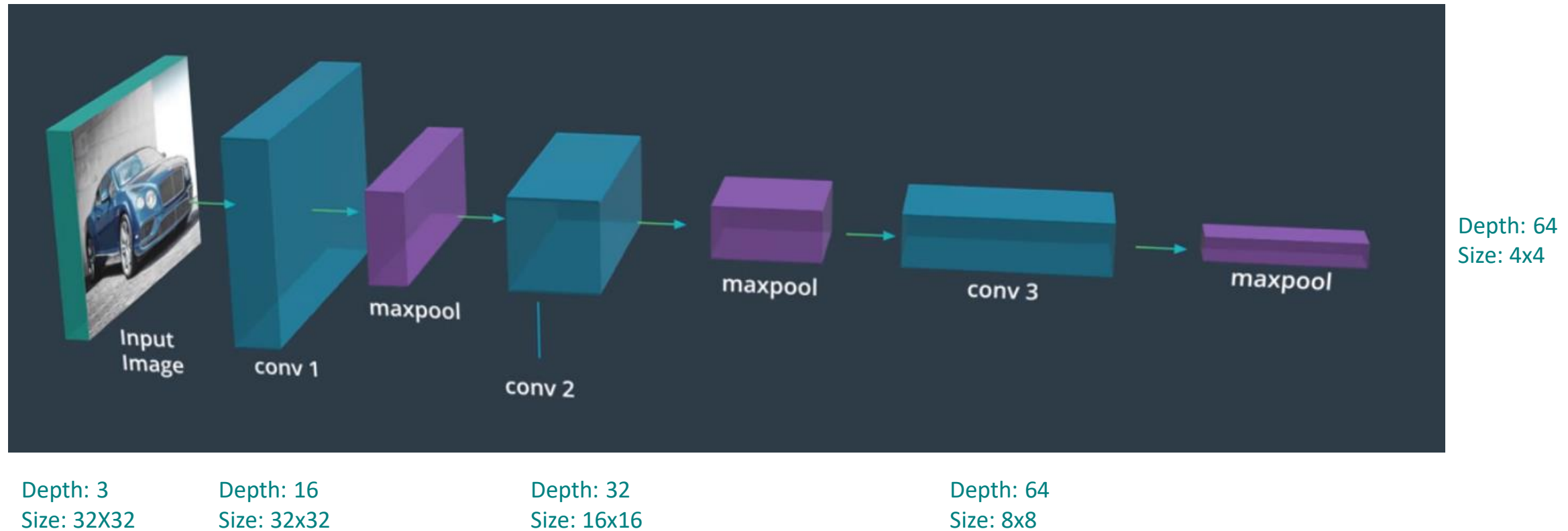
        self.fc1 = nn.Linear(64 * 4 * 4, 500)
        self.fc2 = nn.Linear(500, 10)
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        # add sequence of convolutional and max pooling layers
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
```

 Defining Layers

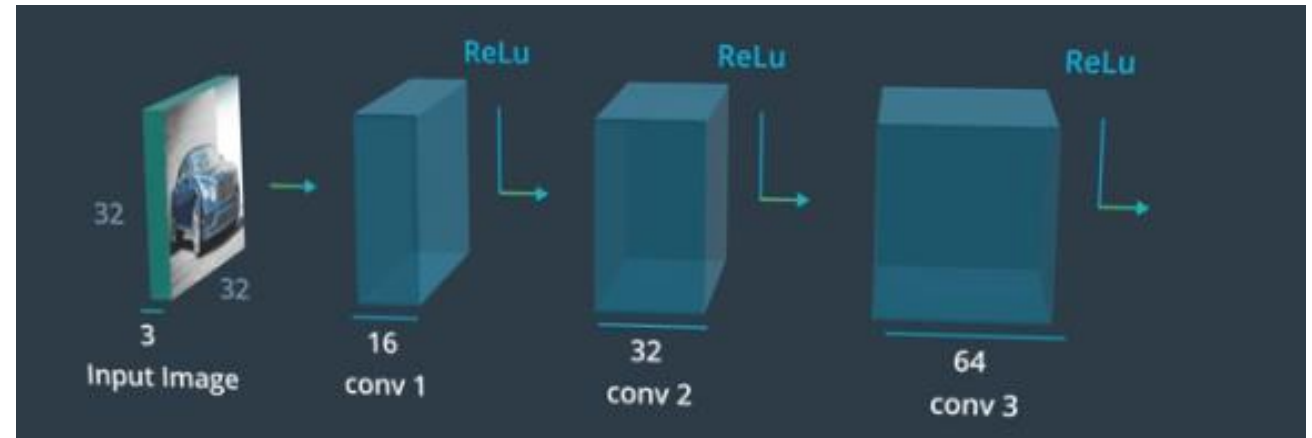
 Pooling layer applied
after each
convolutional layer

Defining Network Architecture: 2



Defining Network Architecture: 3

```
def forward(self, x):  
    x = self.pool(F.relu(self.conv1(x)))  
    x = self.pool(F.relu(self.conv2(x)))  
    x = self.pool(F.relu(self.conv3(x)))  
  
    # flatten image input  
    x = x.view(-1, 64 * 4 * 4)  
  
    x = self.dropout(x)  
    x = F.relu(self.fc1(x))  
  
    x = self.dropout(x)  
    x = self.fc2(x)  
    return x
```



```
model = Net()  
print(model)  
  
# move tensors to GPU if CUDA is available  
if train_on_gpu:  
    model.cuda()
```



- Model is instantiated .
- Model is moved to GPU

Defining Loss and Optimizer

Loss and Optimizer

```
criterion = nn.CrossEntropyLoss()  
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

Training The Network

Training the Network : on train dataset

```
for epoch in range(1, n_epochs+1):  
    train_loss = 0.0  
    valid_loss = 0.0  
  
    model.train()  
    for batch_idx, (data, target) in enumerate(train_loader):  
        if train_on_gpu:  
            data, target = data.cuda(), target.cuda()  
            optimizer.zero_grad()  
  
            output = model(data)  
  
            loss = criterion(output, target)  
  
            loss.backward()  
  
            optimizer.step()  
  
            train_loss += loss.item()*data.size(0)
```

- **Optimizer.zero_grad()** : sets the gradient to zero
- **Model(data)** : calculates predicted output
- **Criterion(,)** : computes loss
- **Backward()**: computes gradient of the loss
- **Optimizer.step()**: updates weights in the network

Training the Network : on validation dataset

```
model.eval()
for batch_idx, (data, target) in enumerate(valid_loader):

    if train_on_gpu:
        data, target = data.cuda(), target.cuda()

    output = model(data)

    loss = criterion(output, target)

    valid_loss += loss.item()*data.size(0)
```

No Backpropagation

Training the Network : calculating loss

```
# calculate average losses
train_loss = train_loss/len(train_loader.sampler)
valid_loss = valid_loss/len(valid_loader.sampler)

# print training/validation statistics
print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
    epoch, train_loss, valid_loss))

# save model if validation loss has decreased
if valid_loss <= valid_loss_min:
    print('Validation loss decreased ({:.6f} --> {:.6f}). Saving model ...'.format(
        valid_loss_min,
        valid_loss))
    torch.save(model.state_dict(), 'model_augmented.pt')
    valid_loss_min = valid_loss
```

Training the Network : Saving model

```
Epoch: 13      Training Loss: 0.964897      Validation Loss: 0.930926
Validation loss decreased (0.962342 --> 0.930926). Saving model ...
Epoch: 14      Training Loss: 0.937324      Validation Loss: 0.904205
Validation loss decreased (0.930926 --> 0.904205). Saving model ...
Epoch: 15      Training Loss: 0.910009      Validation Loss: 0.883212
Validation loss decreased (0.904205 --> 0.883212). Saving model ...
Epoch: 16      Training Loss: 0.888246      Validation Loss: 0.862911
Validation loss decreased (0.883212 --> 0.862911). Saving model ...
Epoch: 17      Training Loss: 0.865455      Validation Loss: 0.846402
Validation loss decreased (0.862911 --> 0.846402). Saving model ...
Epoch: 18      Training Loss: 0.846341      Validation Loss: 0.806580
Validation loss decreased (0.846402 --> 0.806580). Saving model ...
Epoch: 19      Training Loss: 0.825042      Validation Loss: 0.826254
Epoch: 20      Training Loss: 0.809559      Validation Loss: 0.809797
Epoch: 21      Training Loss: 0.789205      Validation Loss: 0.779899
Validation loss decreased (0.806580 --> 0.779899). Saving model ...
```

Testing the Trained Network

Testing Trained Network

```
model.eval()
# iterate over test data
for batch_idx, (data, target) in enumerate(test_loader):
    # move tensors to GPU if CUDA is available
    if train_on_gpu:
        data, target = data.cuda(), target.cuda()

    output = model(data)

    loss = criterion(output, target)

    test_loss += loss.item()*data.size(0)

    _, pred = torch.max(output, 1)

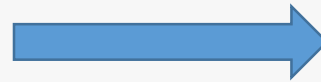
    correct_tensor = pred.eq(target.data.view_as(pred))
    correct = np.squeeze(correct_tensor.numpy()) if not train_on_gpu else np.squeeze(correct_tensor.cpu().numpy())

    for i in range(batch_size):
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

# average test loss
test_loss = test_loss/len(test_loader.dataset)
print('Test Loss: {:.6f}\n'.format(test_loss))

for i in range(10):
    if class_total[i] > 0:
        print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (
            classes[i], 100 * class_correct[i] / class_total[i],
            np.sum(class_correct[i]), np.sum(class_total[i])))
    else:
        print('Test Accuracy of %5s: N/A (no training examples)' % (classes[i]))

print('\nTest Accuracy (Overall): %2d%% (%2d/%2d)' % (
    100. * np.sum(class_correct) / np.sum(class_total),
    np.sum(class_correct), np.sum(class_total)))
```



Test Loss: 0.692109

Test Accuracy of airplane: 79% (795/1000)
Test Accuracy of automobile: 86% (867/1000)
Test Accuracy of bird: 56% (567/1000)
Test Accuracy of cat: 58% (585/1000)
Test Accuracy of deer: 76% (765/1000)
Test Accuracy of dog: 68% (682/1000)
Test Accuracy of frog: 81% (810/1000)
Test Accuracy of horse: 84% (848/1000)
Test Accuracy of ship: 86% (865/1000)
Test Accuracy of truck: 83% (837/1000)

Test Accuracy (Overall): 76% (7621/10000)

Output