Classification with Deep Neural Networks

Contents


```
import numpy as np
np.set_printoptions(precision=3)
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set(rc={"figure.dpi":100, "savefig.dpi":300})
sns.set_context("notebook")
sns.set_style("ticks")

import scipy
import scipy.stats as st
import urllib.request
import os

def download(
    url : str,
    local_filename : str = None
):
    """Download a file from a url.

    Arguments
    url            -- The url we want to download.
    local_filename -- The filemame to write on. If not
                      specified
    """
    if local_filename is None:
        local_filename = os.path.basename(url)
    urllib.request.urlretrieve(url, local_filename)
```

Objectives

- Implement image classification network in `PyTorch`.
- Add L2 regularization.
- Add convolutional layers.
- Add hyperparameter tuning.

References

- Deep Learning with PyTorch: A 60 minute blitz and in particular:
  - Training a Classifier - with which we use the same dataset in this hands-on activity.

The CIFAR10 dataset

We are going to use the CIFAR10 dateset to demonstrate multiclass classification. The dataset consists of 60000 32x32 color images in 10 classes (plane, car, bird, cat, deer, dog, frog, horse, ship, and truck), with 6000 images per class. The dataset can be download direclty from `PyTorch` using the module `torchvision`.

You can think of the original images as 32x32x3 arrays. The first two dimensions correspond to the pixels. The third dimension corresponds to the color (red, green, blue). Of course, we will have to turn them into `PyTorch` tensors. Also, it is more convenient to scale them to be between $[-1, 1]$. We will achieve this using a transformation. Don't worry about this now. We will explain it as we go.

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

# This is the transformation that we will apply to each image
transform = transforms.Compose(
    [transforms.ToTensor(),   # This turns the picture to a Tensor
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]) # This scales it to [-1, 1]

# Here is how you can download the training dataset
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)

# And here is how to download the test dataset:
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)

# These are the class labels
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-
python.tar.gz
```

```
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

Now, all these data went in the folder "./data." Here is what this folder contains:

```
!ls ./data/
```

```
cifar-10-batches-py    cifar-10-python.tar.gz
```

The file `cifar-10-python.tar.gz` is a compressed file containing everything. The contents were automatically extracted and put in the folder `cifar-10-batches-py`. Let's look insider this folder:

```
!ls -lht data/cifar-10-batches-py
```

```
total 363752
-rw-r--r--@ 1 ibilion  staff    88B Jun  4  2009 readme.html
-rw-r--r--@ 1 ibilion  staff   158B Mar 31  2009 batches.meta
-rw-r--r--@ 1 ibilion  staff    30M Mar 31  2009 data_batch_4
-rw-r--r--@ 1 ibilion  staff    30M Mar 31  2009 data_batch_1
-rw-r--r--@ 1 ibilion  staff    30M Mar 31  2009 data_batch_5
-rw-r--r--@ 1 ibilion  staff    30M Mar 31  2009 data_batch_2
-rw-r--r--@ 1 ibilion  staff    30M Mar 31  2009 data_batch_3
-rw-r--r--@ 1 ibilion  staff    30M Mar 31  2009 test_batch
```

You see several files. The important ones are `data_batch_1` to `data_batch_5` and `test_batch`. Each of these contains 10000 images in a binary format. The format is explained here. We can read them as follows:

```
def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict

data = unpickle('data/cifar-10-batches-py/data_batch_1')
# data is a dictionary
# Here are the keys
print(data.keys())
```

```
dict_keys([b'batch_label', b'labels', b'data', b'filenames'])
```

```
# One key has to do with the pictures
# It gives you a numpy array:
print(data[b'data'].shape)
```

```
(10000, 3072)
```

```
# The first dimension correspond to differnt picture
# The second dimension is
32 * 32 * 3
```

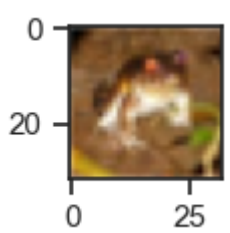
```
3072
```

```
# So this is the first picture:
img = data[b'data'][0, :].reshape((32, 32, 3), order='F')
# Here is the Red channel:
print(img[:, :, 0])
```

```
[[ 59  16  25 ... 208 180 177]
 [ 43   0  16 ... 201 173 168]
 [ 50  18  49 ... 198 186 179]
 ...
 [158 123 118 ... 160 184 216]
 [152 119 120 ...  56  97 151]
 [148 122 109 ...  53  83 123]]
```

The numbers go from 0 (no red) to 255 (full red). Here is how to visualize it:

```
fig, ax = plt.subplots(figsize=(1, 1))
ax.imshow(np.transpose(img, (1, 0, 2)));
```



This is clearly a frog. Let's verify this:

```
classes[data[b'labels'][0]]
```

```
'frog'
```

This is nice. And we could proceed manually like this. However, `PyTorch` offers some useful functionality. Let's investigate the `trainset` that was returned by `CIFAR10`:

```
trainset
```

```
Dataset CIFAR10
    Number of datapoints: 50000
    Root location: ./data
    Split: Train
    StandardTransform
Transform: Compose(
               ToTensor()
               Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
           )
```

```
# Here are the classes:
trainset.classes
```

```
['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']
```

```
# Here is the correspondence between classes and discrete labels
trainset.class_to_idx
```

```
{'airplane': 0,
 'automobile': 1,
 'bird': 2,
 'cat': 3,
 'deer': 4,
 'dog': 5,
 'frog': 6,
 'horse': 7,
 'ship': 8,
 'truck': 9}
```

```
# Here are the images from all training batches
print(trainset.data.shape)
```

```
(50000, 32, 32, 3)
```

```
# Here are the labels
print(trainset.targets[:10])
```

```
[6, 9, 9, 4, 1, 1, 2, 7, 8, 3]
```

Alright. Now, let's use `PyTorch` functionality for looping over the training and the test datasets. We need a DataLoader:

```
# One for the training data:
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=0)

# One for the test data:
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=0)
```

These objects work as follows:

```
# They help you loop over all the data in a random way (because we had shuffle=True)
for i, data in enumerate(trainloader, 0):
    inputs, labels = data
    # Here inputs are of size batch_size x (3 x 32 x 32)
    # Since we had specified, the batch_size to be 4
    # this essentially loads four images per iteration
    if i % 1000 == 0:
        print('Data point:', i, 'input size:', str(inputs.shape))
```

```
Data point: 0 input size: torch.Size([4, 3, 32, 32])
Data point: 1000 input size: torch.Size([4, 3, 32, 32])
Data point: 2000 input size: torch.Size([4, 3, 32, 32])
Data point: 3000 input size: torch.Size([4, 3, 32, 32])
Data point: 4000 input size: torch.Size([4, 3, 32, 32])
Data point: 5000 input size: torch.Size([4, 3, 32, 32])
Data point: 6000 input size: torch.Size([4, 3, 32, 32])
Data point: 7000 input size: torch.Size([4, 3, 32, 32])
Data point: 8000 input size: torch.Size([4, 3, 32, 32])
Data point: 9000 input size: torch.Size([4, 3, 32, 32])
Data point: 10000 input size: torch.Size([4, 3, 32, 32])
Data point: 11000 input size: torch.Size([4, 3, 32, 32])
Data point: 12000 input size: torch.Size([4, 3, 32, 32])
```

When you reach the end of the loop you have visited all the images once. Notice that `PyTorch` has reshaped the images to 3 x 32 x 32 3D arrays. This is more convenient for the convolutional layers we are going to use later. Also, `PyTorch` is using the transformations we gave it to scale the data to array elements to $[-1, 1]$. Let me show you an example:

```
for i, data in enumerate(trainloader, 0):
    inputs, labels = data
    print(inputs[0])
    break
```

```
tensor([[[-0.8980, -0.8667, -0.8510,  ..., -0.8039, -0.8118, -0.8196],
         [-0.8196, -0.7882, -0.8118,  ..., -0.8039, -0.8196, -0.8196],
         [-0.7255, -0.7255, -0.8431,  ..., -0.7804, -0.7961, -0.8196],
         ...,
         [ 0.4353,  0.4902,  0.4824,  ...,  0.6392,  0.7020,  0.7255],
         [ 0.2157,  0.2235,  0.0745,  ...,  0.7961,  0.7569,  0.7333],
         [-0.0431, -0.0824, -0.1137,  ...,  0.7255,  0.6549,  0.6706]],

        [[-0.6471, -0.6392, -0.6471,  ..., -0.5843, -0.5922, -0.6000],
         [-0.6235, -0.6000, -0.6549,  ..., -0.6000, -0.6157, -0.6157],
         [-0.5608, -0.5843, -0.7490,  ..., -0.6000, -0.6157, -0.6314],
         ...,
         [ 0.3412,  0.2941,  0.2000,  ...,  0.5765,  0.6392,  0.6627],
         [ 0.0431, -0.0196, -0.2314,  ...,  0.7333,  0.6941,  0.6706],
         [-0.2392, -0.3176, -0.3725,  ...,  0.6627,  0.5922,  0.6078]],

        [[-0.3333, -0.3333, -0.3804,  ..., -0.2549, -0.2549, -0.2471],
         [-0.2235, -0.2549, -0.3961,  ..., -0.3098, -0.3176, -0.3098],
         [-0.1373, -0.2392, -0.5059,  ..., -0.2706, -0.2941, -0.3255],
         ...,
         [ 0.2078,  0.1059, -0.0667,  ...,  0.5529,  0.6157,  0.6392],
         [-0.1137, -0.2078, -0.4353,  ...,  0.7098,  0.6784,  0.6471],
         [-0.3725, -0.4510, -0.5059,  ...,  0.6314,  0.5608,  0.5765]]])
```

Training a classifier using a dense DNNs

Let's just train a classifer using a dense neural network. It's not going to work very well, but it is very easy to put together. We are going to start the network with 3 x 32 x 32 = 3072, followed up with a few dense layers that end at 10 outputs passed through softmax. However, for reasons of numerical stability, we are not going to end with the softmax layer during training.

```
import torch.nn as nn

# The classifer - The dimensions of the layers have
# been picked to match those of the convolutional neural network
# that we are going to build later
# For now, just notice that we gradually take the 3072-dimensional input
# down to 10 dimensions (the number of classes we have)
# Also, notice that I do not add the softmax layer at this point
model_dense = nn.Sequential(nn.Linear(3072, 1176), nn.ReLU(),
                            nn.Linear(1176, 400), nn.ReLU(),
                            nn.Linear(400, 120), nn.ReLU(),
                            nn.Linear(120, 84), nn.ReLU(),
                            nn.Linear(84, 10))

# This is our loss function.
# Read this: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html
criterion = nn.CrossEntropyLoss()
# The reason we did not add the Softmax layer at the end is because
# the loss function above is doing it internally.
# It expects that you provide "contain raw, unnormalized scores for each class"
```

```
# Here is the optimizer
import torch.optim as optim
optimizer = optim.SGD(model_dense.parameters(), lr=0.001, momentum=0.9)
```

Let's train the network. This is going to take a while...

```
# How many times do you want to go over the entire dataset?
# Don't pick a very big number because you will overfit
num_epochs = 2

# Here is the main training algorithm
for epoch in range(num_epochs):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model_dense(inputs.reshape(4, 3 * 32 * 32))
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 1000 == 999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 1000))
            running_loss = 0.0

print('Finished Training')
```

```
[1,  1000] loss: 2.274
[1,  2000] loss: 2.061
[1,  3000] loss: 1.908
[1,  4000] loss: 1.845
[1,  5000] loss: 1.766
[1,  6000] loss: 1.737
[1,  7000] loss: 1.663
[1,  8000] loss: 1.665
[1,  9000] loss: 1.620
[1, 10000] loss: 1.598
[1, 11000] loss: 1.616
[1, 12000] loss: 1.604
[2,  1000] loss: 1.507
[2,  2000] loss: 1.529
[2,  3000] loss: 1.462
[2,  4000] loss: 1.467
[2,  5000] loss: 1.460
[2,  6000] loss: 1.480
[2,  7000] loss: 1.495
[2,  8000] loss: 1.431
[2,  9000] loss: 1.457
[2, 10000] loss: 1.459
[2, 11000] loss: 1.412
[2, 12000] loss: 1.428
Finished Training
```

Since training networks takes a while, it's a good idea to save it:

```
torch.save(model_dense.state_dict(), 'hands-on-25-model-dense.pth')
```

Here it is as a file:

```
!ls -lht hands-on-25-model-dense.pth
```

```
-rw-r--r--  1 ibilion  staff    16M May  9 08:34 hands-on-25-model-dense.pth
```

Now let's make some predictions:

```
# Get the first four images and their labels
dataiter = iter(testloader)
images, labels = dataiter.next()
```

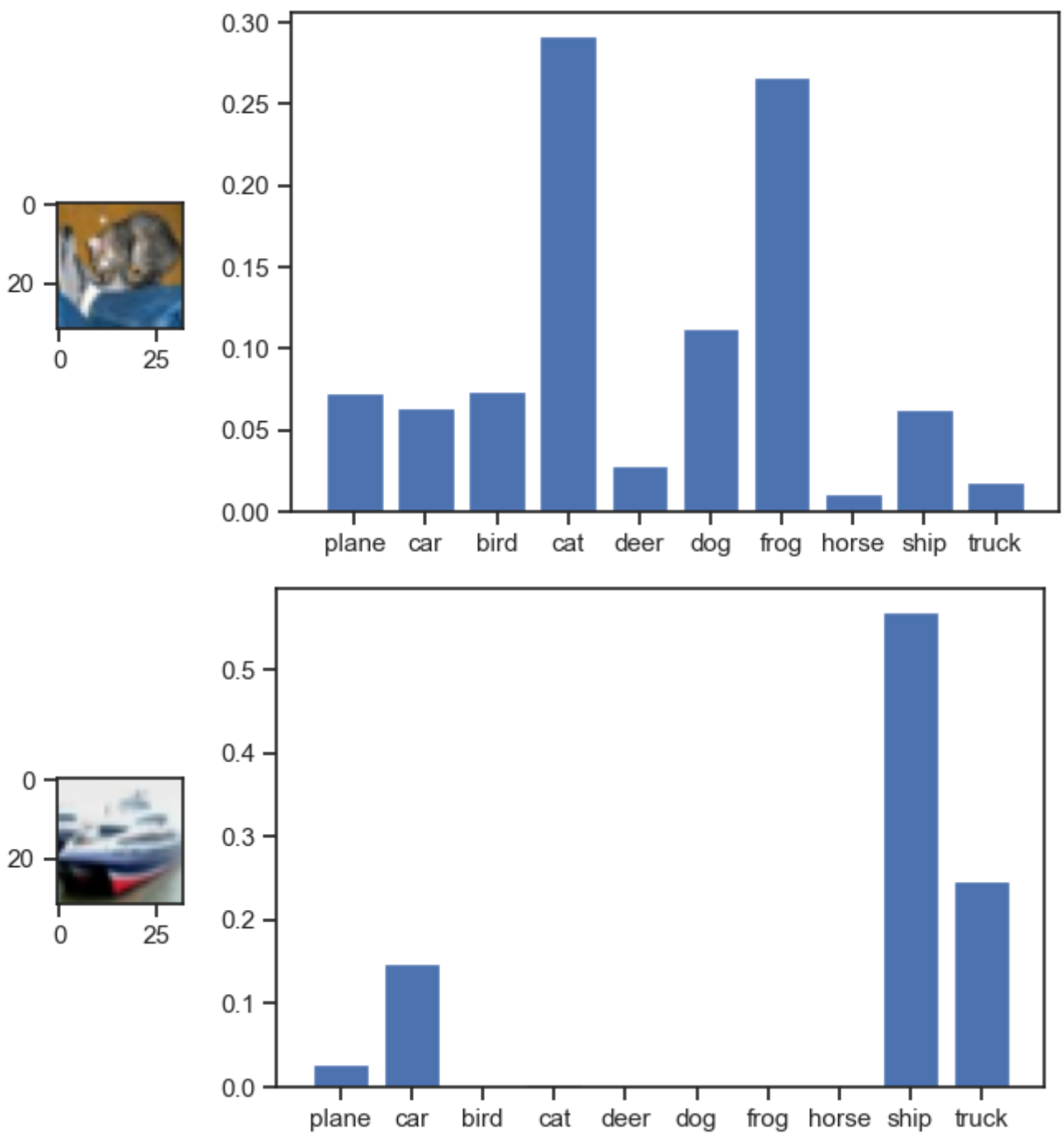
```
print(labels)
```

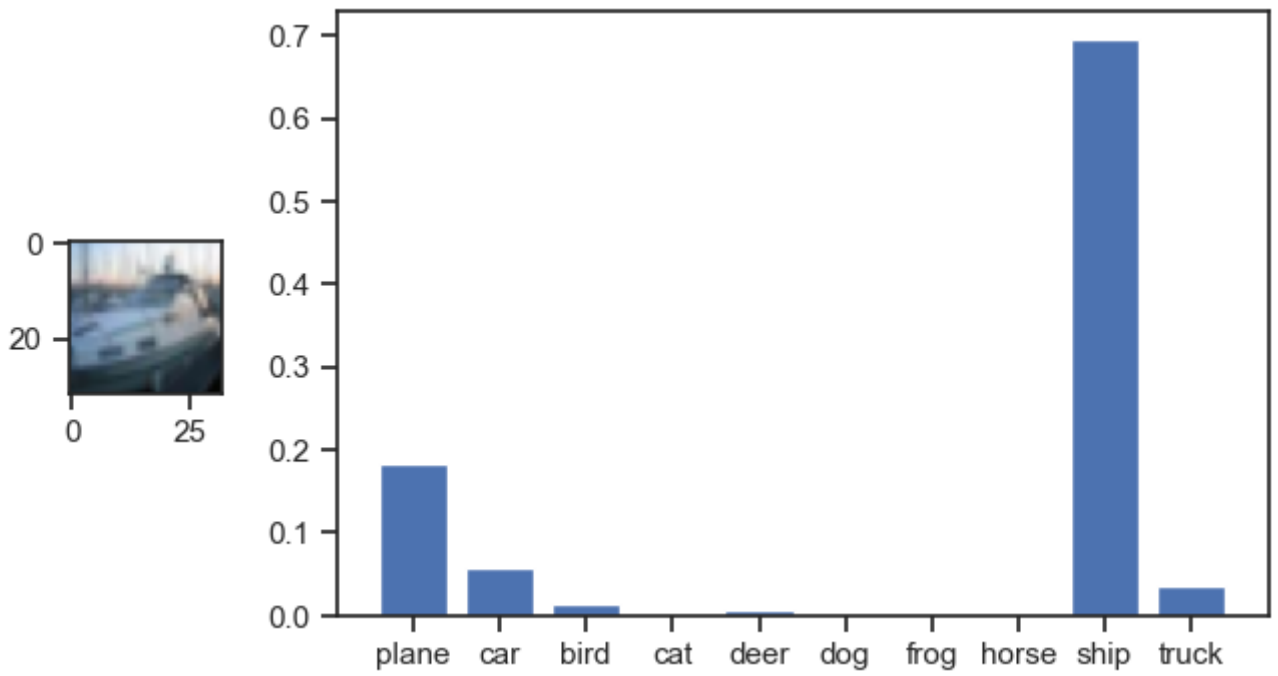
```
tensor([3, 8, 8, 0])
```

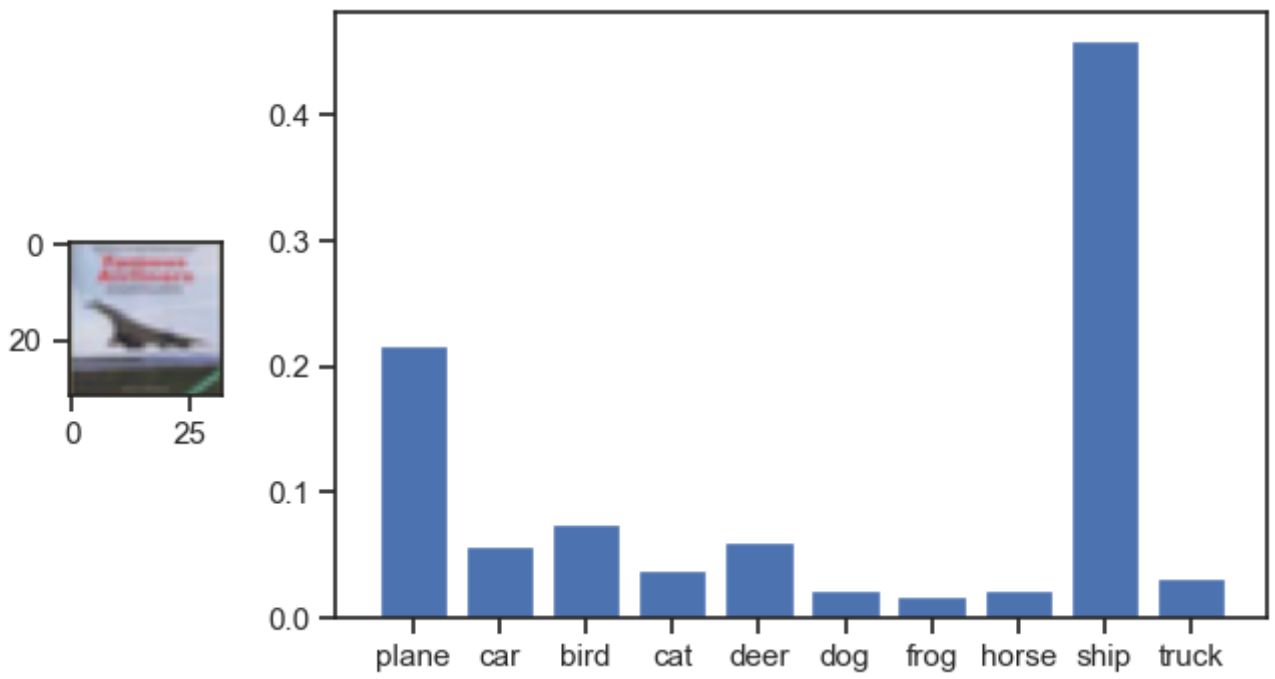
```
tensor([3, 8, 8, 6])
```

```
# Make predictions with the net and pass them through
# softmax to turn them into probabilities
st = nn.Softmax(dim=1)
predictions = st(model_dense(images.reshape(4, 3072)))
```

```
def imshow(img, ax):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    ax.imshow(np.transpose(npimg, (1, 2, 0)))

# Plot the pictures and the predictions
for i in range(4):
    fig, ax = plt.subplots(figsize=(1,1))
    imshow(images[i], ax)
    fig2, ax2 = plt.subplots()
    ax2.bar(np.arange(10), predictions[i].detach().numpy())
    ax2.set_xticks(np.arange(10))
    ax2.set_xticklabels(classes)
```







Now, let's do the same thing with a convolutional neural network. We are not going to use `nn.Sequential` this time. Instead, we are going to use `nn.Module` to manually create the network. The documentation is here. You basically need to inherit `nn.Module`, and implement `__init__()` and `forward()`.

```
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # A convolutional layer:
        # 3 = input channels (colors),
        # 6 = output channels (features),
        # 5 = kernel size
        self.conv1 = nn.Conv2d(3, 6, 5)
        # A 2 x 2 max pooling layer - we are going to use it two times
        self.pool = nn.MaxPool2d(2, 2)
        # Another convolutional layer
        self.conv2 = nn.Conv2d(6, 16, 5)
        # Some linear layers
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # This function implements your network output
        # Convolutional layer, followed by relu, followed by max pooling
        x = self.pool(F.relu(self.conv1(x)))
        # Same thing
        x = self.pool(F.relu(self.conv2(x)))
        # Flatting the output of the convolutional layers
        x = x.view(-1, 16 * 5 * 5)
        # Go throught the first dense linear layer followed by relu
        x = F.relu(self.fc1(x))
        # Through the second dense layer
        x = F.relu(self.fc2(x))
        # Finish up with a linear transformation
        x = self.fc3(x)
        return x


model_cnn = Net()
```

Here is a new optimizer:

```
model_cnn
```

```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

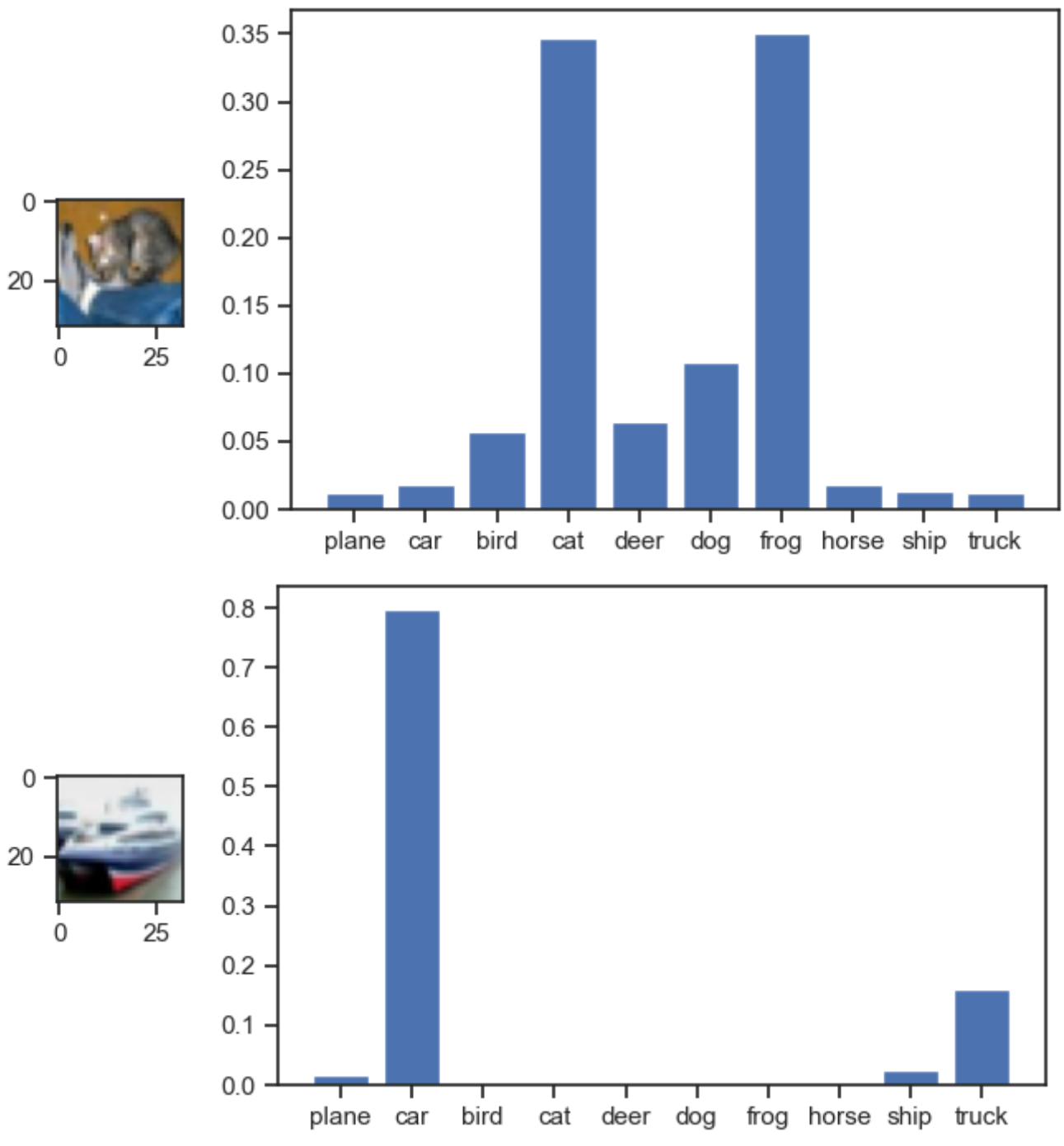
```
optimizer = optim.SGD(model_cnn.parameters(), lr=0.001, momentum=0.9)
```

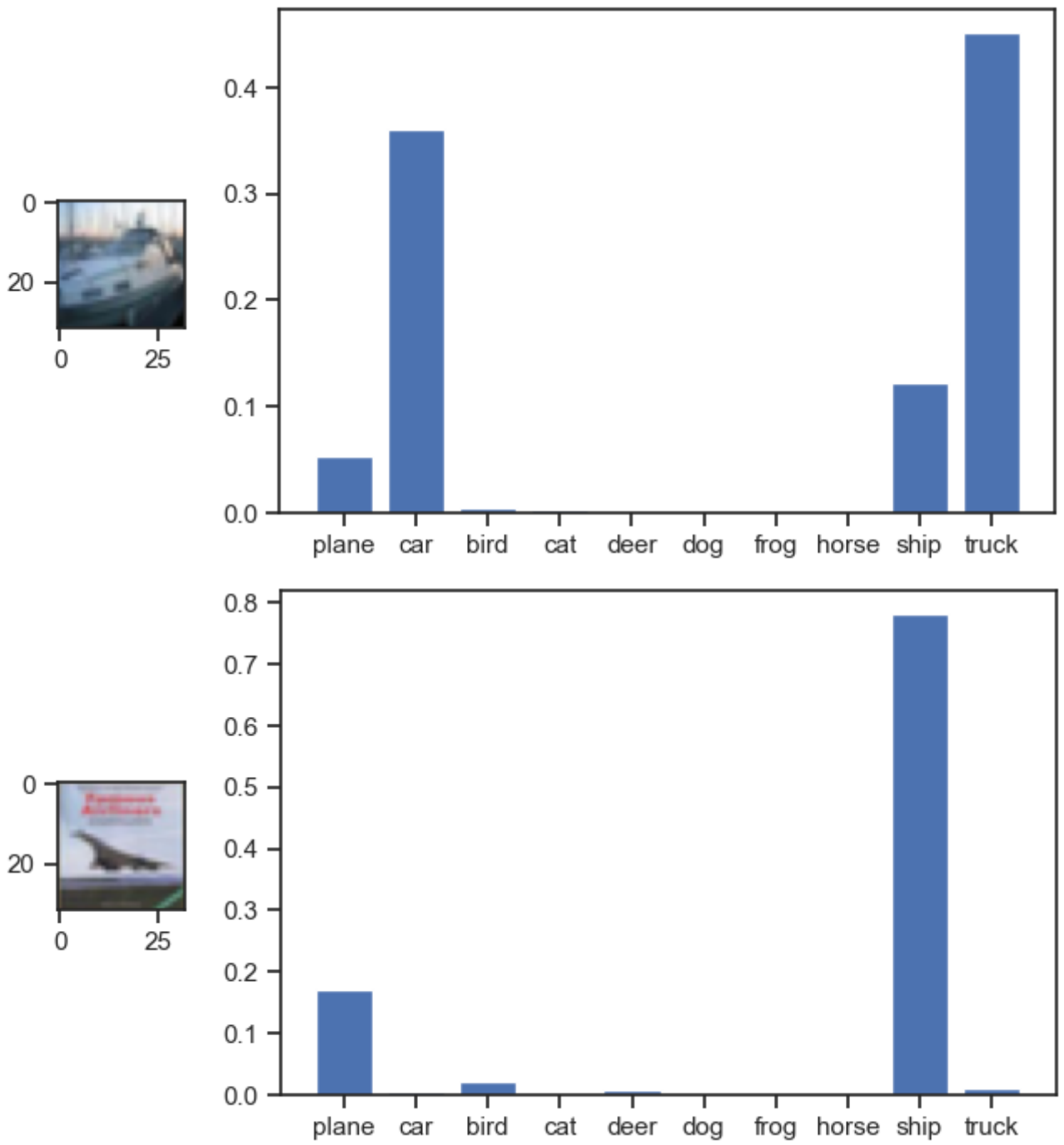
```
# How many times do you want to go over the entire dataset?
# Don't pick a very big number because you will overfit
num_epochs = 2

# Here is the main training algorithm
for epoch in range(num_epochs):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model_cnn(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 1000 == 999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 1000))
            running_loss = 0.0

print('Finished Training')
```

```
[1,  1000] loss: 2.270
[1,  2000] loss: 2.052
[1,  3000] loss: 1.881
[1,  4000] loss: 1.756
[1,  5000] loss: 1.695
[1,  6000] loss: 1.607
[1,  7000] loss: 1.581
[1,  8000] loss: 1.538
[1,  9000] loss: 1.534
[1, 10000] loss: 1.493
[1, 11000] loss: 1.456
[1, 12000] loss: 1.450
[2,  1000] loss: 1.444
[2,  2000] loss: 1.377
[2,  3000] loss: 1.373
[2,  4000] loss: 1.380
[2,  5000] loss: 1.356
[2,  6000] loss: 1.362
[2,  7000] loss: 1.333
[2,  8000] loss: 1.317
[2,  9000] loss: 1.318
[2, 10000] loss: 1.270
[2, 11000] loss: 1.294
[2, 12000] loss: 1.310
Finished Training
```

Make some predictions:

```
# Make predictions with the net and pass them through
# softmax to turn them into probabilities
st = nn.Softmax(dim=1)
predictions = st(model_cnn(images))
for i in range(4):
    fig, ax = plt.subplots(figsize=(1,1))
    imshow(images[i], ax)
    fig2, ax2 = plt.subplots()
    ax2.bar(np.arange(10), predictions[i].detach().numpy())
    ax2.set_xticks(np.arange(10))
    ax2.set_xticklabels(classes)
```





It doesn't work equally well for all classes. Here is some code from the `PyTorch` tutorial to get the accuracy for each class:

```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = model_cnn(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

```
Accuracy of plane : 56 %
Accuracy of   car : 81 %
Accuracy of  bird : 55 %
Accuracy of   cat : 17 %
Accuracy of  deer : 53 %
Accuracy of   dog : 36 %
Accuracy of  frog : 76 %
Accuracy of horse : 53 %
Accuracy of  ship : 64 %
Accuracy of truck : 48 %
```

This is not very good. There are several things that we can do. First, we would run this for more epochs. At least 50 epochs are probably needed to train it properly. Second, we could add data augmentation. This can be done through transformation, see this. Third, we have to make the netork a little bit bigger. Here is a list of large networks trained on CIFAR10. It is possible to reach an accuracy of 95%.

Questions

- Set the number of epochs for the CNN-based model to 40. How much better accuracy do you get? Make sure you do this right before you go to bed and look at it in the morning. Alternatively, you can go for a run...

By Ilias Bilionis (ibilion[at]purdue.edu)