Programming differentiable variables

Most learning tasks consists of adjusting variable parameters in an algebraic expression to fit the given data. For instance, consider the expression, $ax+b=y$. Here, a and b are the variable parameters that affect how x is fitted to y. In more complicated situations, a and b would be vectors—or matrices… or *tensors*, that is, higher-dimensional vectors in deep learning terminology—but still, just parameters to adjust.

A neural network is *just* a network of tensors. The training of a neural network is a task of adjusting its parameters to fit the data. The predominant method to perform these adjustments is through gradient descent. That is, by adjusting the parameters in the direction which will best minimize (to our best knowledge) the difference between the current output and the true output. This direction is discovered through differentiation.

A deep learning programming framework such as TensorFlow, Keras, or PyTorch would provide many features to make the task easy. For example, in the case of PyTorch, the following features are provided:

- Automatic computation of the direction for adjustment (`torch.autograd`)
- Built-in commonly-used deep learning layers (e.g. `torch.nn.Linear`, `torch.nn.Conv2d`, `torch.nn.RNN`)
- Built-in commonly-used activation functions (e.g. `torch.nn.functional.relu`, `torch.nn.functional.linear, torch.nn.functional.sigmoid`)
- Built-in commonly-used functions to compute how well the data fits (e.g. `torch.nn.MSELoss`, `torch.nn.BCELoss`, `torch.nn.CrossEntropyLoss`)
- Built-in commonly-used strategies for making adjustments to the parameters (`torch.optim`)
- Standardized code structure for computing losses and making adjustments
- Last but not least, GPU acceleration (only CUDA is supported at present)

In this practical, we will use PyTorch to solve several tasks that can be solved by parameter adjustments through differentiation. While we use PyTorch herein, the codes can be easily modified to run on other deep learning frameworks such as Keras or TensorFlow.

Task 1: Install PyTorch

The Lubuntu VM provided has PyTorch installed. Alternatively, you can install PyTorch from the instructions in `http://pytorch.org/`. If you have a GPU, you definitely want to do the latter. However, the codes provided herein are not GPU-aware, so you will have to add a few lines of codes (that convert variables to use CUDA) if you want that.

Alternatively, you can run this practical session on Colab, Google's new cloud-based GPU-enabled Python platform, `https://colab.research.google.com/`, which is (mind-bogglingly) free. See the Appendix on how to use colab for this practical session.

Task 2: Linear regression

Recall that linear regression allows a closed form solution. Just for the sake of illustration, in this task we instead solve linear regression with a gradient descent approach (which is, of course, neither efficient nor effective). Here, the gradient is the same as that in the perceptron training rule (see lecture notes).

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

# Find correlation between the first two columns of the iris setosa data (first 50 rows)

# The iris dataset can be retrieved from the sklearn package
#from sklearn.datasets import load_iris
#(X, y) = load_iris(return_X_y=True)

# Alternatively, we can load them from our TSV file
df = pandas.read_csv('iris.tsv', usecols=[0, 1], sep='\t')
x = df.as_matrix()[0:50,0]
y = df.as_matrix()[0:50,1]

# The following implements the perceptron training rule
w = 0
b = 0
lr = 0.0001

for n in range(1000):
    w_grad = 0
    b_grad = 0
    for i in range(0, len(x)):
        y_pred = (w * x[i]) + b
        loss = y[i] - y_pred
        w_grad = w_grad - (x[i] * loss)
        b_grad = b_grad - loss
    b = b - (lr * b_grad)
    w = w - (lr * w_grad)

# Plot the data points
plt.scatter(x, y)

# Draw a linear regression line
x_axis = np.linspace(3.5, 6.5, 50)
y_axis = w * x_axis + b
plt.plot(x_axis, y_axis, 'r--', label='regression line')

plt.legend()
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.show()
```

Using PyTorch, the same codes would become

```
import pandas
import numpy as np
import torch
from torch.autograd import Variable
import matplotlib.pyplot as plt

df = pandas.read_csv('iris.tsv', usecols=[0, 1], sep='\t')
x = df.as_matrix()[0:50,0]
y = df.as_matrix()[0:50,1]
x = np.reshape(x, (len(x), 1))
y = np.reshape(y, (len(y), 1))
x_model = Variable(torch.from_numpy(x).float())
y_model = Variable(torch.from_numpy(y).float())

model = torch.nn.Linear(1, 1)

criterion = torch.nn.MSELoss() # mean squared error
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

for epoch in range(1000):
    optimizer.zero_grad()
    y_pred = model(x_model)
    loss = criterion(y_pred, y_model)
    loss.backward()
    optimizer.step()

# Plot the data points
plt.scatter(x, y)

# Draw a linear regression line
param = list(model.parameters())
w = param[0].data.numpy()[0][0]
b = param[1].data.numpy()[0]
x_axis = np.linspace(3.5, 6.5, 50)
y_axis = w * x_axis + b
plt.plot(x_axis, y_axis, 'r--', label='regression line')
# Alternatively, plot the linear regression line using the model output
# This can be convenient when there is no easy way to compute y from the parameters
#model.eval()
#predict = model(x_model).data.numpy()
#plt.plot(x, predict, 'r--', label='regression line')

plt.legend()
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.show()
```

Notice how each of the components in the earlier program is replaced by a convenient feature in PyTorch. The loss is now computed using a `criterion` object, while the parameter updating is handled by an `optimizer` object. Take note that the input and output data must now be converted into `torch.autograd.Variable` objects in order for them to work within the PyTorch framework. A `torch.autograd.Variable`, `v` say, can be converted into a NumPy array, by calling `v.data.numpy()`. Notice that are several such conversions in the codes above.

© Ng Yen Kaow, 2013-2018

Task 3: Logistic regression

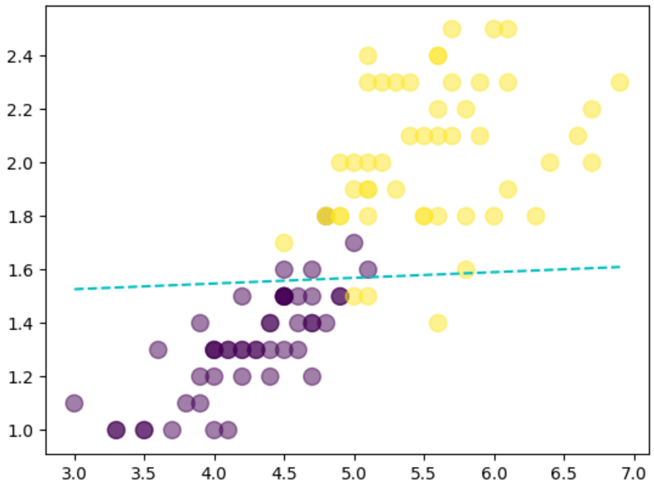
In the previous task, we used a linear "layer" that is similar to a perceptron, by using the PyTorch module `torch.nn.Linear`. It suffices that we apply the sigmoid function on that output to create a logistic regression model. In the following, we create a custom module called `LogisticRegression` to perform just that.

```
import pandas
import numpy as np
import torch
from torch.autograd import Variable
import matplotlib.pyplot as plt

# Construct a classifier for versicolor (row 50 to 100) and virginica (row 100 to 150)
# based on their petal width and petal length (3rd and 4th column)

df = pandas.read_csv('iris.tsv', sep='\t')
df.loc[df.Species!='Iris-virginica', 'Species']=0 # range of sigmoid function: 0 to 1
df.loc[df.Species=='Iris-virginica', 'Species']=1 # range of sigmoid function: 0 to 1
X = df.as_matrix()[50:150,2:4]
y = df.as_matrix()[50:150,4]
y = np.reshape(y, (len(y), 1))

X_model = Variable(torch.Tensor(X))
y_model = Variable(torch.Tensor(y))

class LogisticRegression(torch.nn.Module):
    def __init__(self):
        super(LogisticRegression, self).__init__()
        self.linear = torch.nn.Linear(2, 1)
    def forward(self, x):
        y_pred = torch.nn.functional.sigmoid(self.linear(x))
        return y_pred

model = LogisticRegression()

criterion = torch.nn.BCELoss(size_average=True) # binary cross entropy
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

for epoch in range(5000):
    optimizer.zero_grad()
    y_pred = model(X_model)
    loss = criterion(y_pred, y_model)
    loss.backward()
    optimizer.step()

# Plot the data points
plt.scatter(X[:,0], X[:,1], c=y.reshape(len(y)), s=100, alpha=0.5)

# Draw the logistic regression boundary
param = list(model.parameters())
w = param[0].data.numpy()[0]
b = param[1].data.numpy()[0]
x_axis = np.linspace(np.min(X[:,0]), np.max(X[:,0]), 100)
y_axis = -(b + x_axis*w[0]) / w[1]
plt.plot(x_axis, y_axis, 'c--', label='cutoff boundary')
plt.show()
```

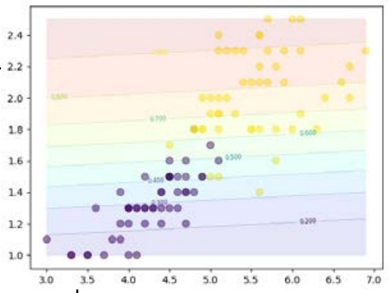
The module accepts two parameters as input, performs a linear combination of the input values, and passes the value to a sigmoid function. (We use only two input values here because we want to be able to produce a 2D graph as shown below. There are no other reasons for not using the full 4 input values in the iris dataset.)

Notice that with the exception of the new module, the codes are exactly the same as that in the linear regression task.

The output of the program is as follows. The two classes of data are shown in differently colored points. Points which represent more sample data are shown with darker shades of its color. A cutoff line shows where the sigmoid function would output 0.5 on the input values.



When the cutoff boundary cannot be easily obtained from the model parameters, one way to show the boundary is to compute the output values of the model for the entire plot area. You can then use matplotlib to construct contour lines out of those output values.



```
xticks = np.linspace(min(X[:,0]), max(X[:,0]), 100)
yticks = np.linspace(min(X[:,1]), max(X[:,1]), 100)
levels = np.linspace(0.1, 0.9, 9)
xx, yy = np.meshgrid(xticks, yticks)
grid = np.c_[xx.ravel(), yy.ravel()]
gridvals = Variable(torch.Tensor(grid))
model.eval()
preds = model(gridvals).data.numpy().reshape(xx.shape)
cp = plt.contour(xx, yy, preds, linewidths=0.1, levels=levels)
plt.clabel(cp, inline=True, fontsize=6)
plt.contourf(xx, yy, preds, 9, cmap=plt.cm.jet, alpha=0.1)
plt.show()
```

Task 4: Exercise

1. Write a Python program that finds the linear regression line using the closed form solution (as shown) to verify the parameters found with PyTorch in Task 2.

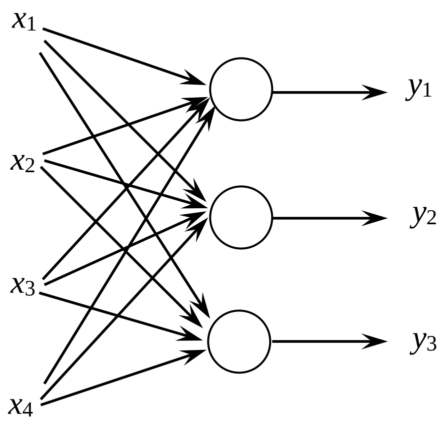
$$w = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2},\ b = \bar{y} - w\bar{x}$$

2. Modify the classifier in Task 3 to use different sets of features as input. (If you use more than 2 features, comment out the codes for visualization.)

3. Use the following components to write a linear classifier that classifies all 3 iris classes using all the 4 features:

Model:	`torch.nn.Linear`
Criterion:	`torch.nn.CrossEntropyLoss`
Optimizer:	`torch.optim.SGD`

   HINT: Use `torch.nn.Linear(4,3)`, which will create a model that consists of the following three linear nodes.



Task 5: Using a pre-trained model

Training a neural network model often consumes considerable time. Furthermore, the success in training a well-performing model is sometimes contingent on the hyperparameters used in the training. These make successfully-trained models valuable for inclusion in a deep learning framework for reuse purposes. PyTorch provides many such pre-trained models, e.g. ResNet, DenseNet, Inception, NASNet. These models are pre-trained with some large image data sets such as ImageNet or CIFAR.

In this task we shall load a pre-trained ResNet model to classify iris images. The residual net (ResNet) architecture is described in the paper "Deep Residual Learning for Image Recognition" (`https://arxiv.org/abs/1512.03385`). The architecture allows an arbitrary setup of number of layers. For instance, the smallest model reported in the paper, resnet18, consists of 18 layers.

For our classification task, 20 images have been prepared for each of the iris species in the directory called `iris_pics`. The images for iris setosa, versicolor, and virginica are located respectively in the subdirectories called `se`, `ve`, and `vi`.

We will perform a training as well as a validation phase on the images, using 15 images from each species for training and the remaining 5 for validation. Our training data for the three classes of iris will be placed in the directories `resnet_wd/train/{se,vi,se}` respectively, whereas the validation data will be placed in the directories `resnet_wd/val/{se,vi,se}`. The following UNIX script creates these directories, then copies a random selection of 15 images for each species to the training directories and copies the remaining 5 images to the validation directories.

```
SRCDIR=iris_pics
TARDIR=resnet_wd
TOTFILES=15
rm -r $TARDIR/
for X in se vi ve; do
    mkdir -p $TARDIR/train/$X
    NUMFILES=`ls -1 $TARDIR/train/$X| wc -l`
    while [ $NUMFILES -lt $TOTFILES ]; do
        FILE=`echo $((1 + RANDOM % 20)) |awk '{$1 = sprintf("%02d", $1); print}'`
        if [ -f $TARDIR/train/$X/$X$FILE.jpg ]; then continue; fi
        cp $SRCDIR/$X/$X$FILE.jpg $TARDIR/train/$X/
        NUMFILES=`ls -1 $TARDIR/train/$X |wc -l`
    done
    mkdir -p $TARDIR/val/$X
    for FILE in $SRCDIR/$X/*; do
        if [ -f $TARDIR/train/$X/`basename $FILE` ]; then continue; fi
        cp $FILE $TARDIR/val/$X/
    done
done
```

The following PyTorch codes uses the torchvision library to load the images from the `resnet_wd` directory. It then resizes the images and normalizes them for training. The number of classes is automatically deduced from the number of subdirectories in `resnet_wd/train/`. The codes also initialize a `DataLoader` object which defines how the images will be batch processed in the training phase. The batch size will have an effect on the training since it decides how frequently the backpropagation is performed. Batch processing is not performed in the validation phase.

```
import torch
from torch.autograd import Variable
from torchvision import transforms, datasets, models
import os

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

data_dir = 'resnet_wd'
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=3,
                  shuffle=True, num_workers=3) for x in ['train', 'val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
class_names = image_datasets['train'].classes
```

The following PyTorch codes then performs training as well as validation on the iris images. Note these three important lines in the codes:

```
model = models.resnet18(pretrained=True)
num_features = model.fc.in_features
model.fc = torch.nn.Linear(num_features, len(class_names))
```

The first line loads the pre-trained resnet18 model, which the program will look for in a directory called "`.torch/model/`" within your home directory. If it does not find it there, it will download it from the internet. Hence, create the directory `.torch/models` in your home directory and place the file "`resnet18-5c106cde.pth`" in there.

The 2nd and 3rd line replace the layer called `fc` in resnet with a new `torch.nn.Linear` layer. This `fc` layer is the final layer in the resnet architecture, where the output will be used to determine the class of the image. Since we want to produce a model which classifies three iris classes, we replace `fc` with a new one which accepts the same number of features, but outputs only 3 classes. Since we are reusing large parts of the knowledge in the pre-trained model, such a training process is known as *transfer learning*.

```
model = models.resnet18(pretrained=True)
num_features = model.fc.in_features
model.fc = torch.nn.Linear(num_features, len(class_names))

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

best_acc = 0.0
num_epochs = 5

# Train the model (also perform evaluation with the same codes)
for epoch in range(num_epochs):
    print('\nEpoch {}/{}'.format(epoch, num_epochs-1))
    for phase in ['train', 'val']: # Each epoch has train and val phase
        if phase == 'train':
            model.train(True)  # Set model to training mode
        else:
            model.train(False) # Set model to evaluate mode
        running_loss = 0.0
        running_corrects = 0
        for data in dataloaders[phase]: # Iterate over data
            inputs, labels = data
            inputs, labels = Variable(inputs), Variable(labels)
            optimizer.zero_grad()
            outputs = model(inputs)
            _, preds = torch.max(outputs.data, 1)
            loss = criterion(outputs, labels)
            if phase == 'train':
                loss.backward()
                optimizer.step()
            running_loss += loss.data[0] * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)
        epoch_loss = running_loss / dataset_sizes[phase]
        epoch_acc = running_corrects / dataset_sizes[phase]
        print('{} Loss: {:.4f} Accuracy: {:.4f}'.format(
            phase, epoch_loss, epoch_acc))
        if phase == 'val' and epoch_acc > best_acc:
            best_acc = epoch_acc

print('\nBest val accuracy: {:4f}'.format(best_acc))
```

At around the end of the codes, you will find a line which says

```
running_corrects += torch.sum(preds == labels.data)
```

The function torch.sum is a convenient utility function provided by PyTorch for working with PyTorch data structures (in this case, a Tensor object). There are many other such functions in PyTorch, e.g. `torch.max` (also in the codes above), `torch.min`, `torch.sort`.

Task 6: Exercise

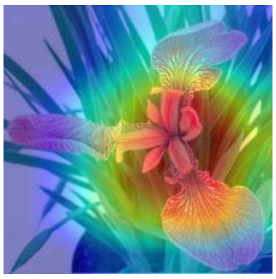
1. Modify the codes in Task 5 to output the names of the misclassified images.

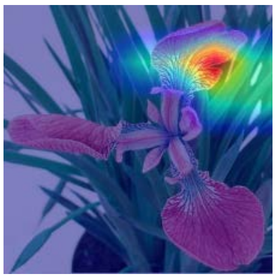
2. Modify the codes in Task 5 to save the best model into a file.

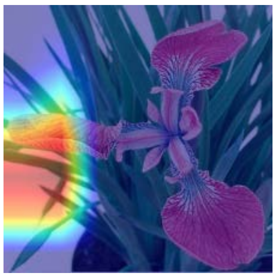
   – To copy the states of a model to a variable, call
   ```
   import copy
   best_model = copy.deepcopy(model.state_dict())
   ```
   – To save the states of a model to a file called `iris.pth`, use
   ```
   torch.save(best_model, 'iris.pth')
   ```

3. (Advanced) Use Grad-CAM to examine the activation maps that `iris.pth` based on in classifying all the iris images. For example, the activation maps for the two images `se01.jpg` and `ve03.jpg` are as follows.

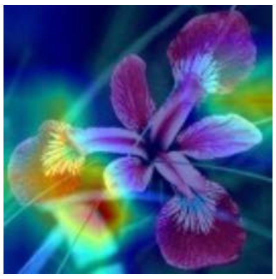
   Setosa image: `se01.jpg` (correctly classified)

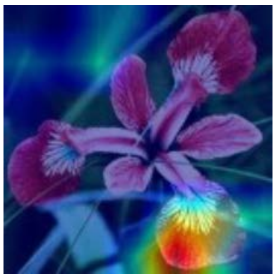

Setosa activation map

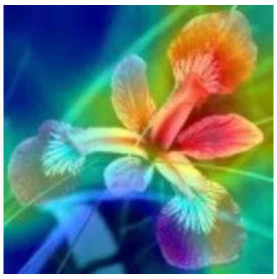

Versicolor activation map


Virginica activation map

   Versicolor image: `ve03.jpg` (misclassified)


Setosa activation map


Versicolor activation map


Virginica activation map

   There are several PyTorch implementations for Grad-CAM, for instance
   `https://github.com/jacobgil/pytorch-grad-cam`, or
   `https://github.com/kazuto1011/grad-cam-pytorch`.

   – To reload `iris.pth` for Grad-CAM, use
   ```
   model.load_state_dict(torch.load('iris.pth'))
   ```

Appendix: Using colab

Go to https://colab.research.google.com and open a Jupyter notebook for Python3. Set the notebook to use the GPU: go to Edit→"Notebook settings", and set "Hardware accelerator" to GPU.

You can paste either Python codes or UNIX commands into the notebook and click on the run button (or press shift-enter) to run them—however, UNIX commands must begin with the character '!' in Jupyter. Note that colab puts you in a UNIX system with present working directory set to '/content/'.

First, you will need to install PyTorch. The easiest way to do so is to run "import torch". This will result in an error, as well as some suggestion codes on how to install PyTorch. Insert those codes into the notebook and run them—after that, PyTorch will be readily installed and imported.

To access your data, upload them into Google Drive. Then, install the Google FUSE library with the following codes, in Jupyter.

```
!apt-get install -y -qq software-properties-common python-software-properties module-init-tools
!add-apt-repository -y ppa:alessandro-strada/ppa 2>&1 > /dev/null
!apt-get update -qq 2>&1 > /dev/null
!apt-get -y install -qq google-drive-ocamlfuse fuse
```

Perform authentication and create credentials to allow colab to use your Google Drive.

```
from google.colab import auth
auth.authenticate_user()
# Generate creds for the Drive FUSE library.
from oauth2client.client import GoogleCredentials
creds = GoogleCredentials.get_application_default()
import getpass
!google-drive-ocamlfuse -headless -id={creds.client_id} -secret={creds.client_secret} < /dev/null 2>&1 | grep URL
vcode = getpass.getpass()
!echo {vcode} | google-drive-ocamlfuse -headless -id={creds.client_id} -secret={creds.client_secret}
```

Create a mount point in colab for your Google Drive, and mount the drive there.

```
# Create a directory and mount Google Drive using that directory
!mkdir -p drive
!google-drive-ocamlfuse drive
```

You should now be able to see your drive content in the directory 'drive'.

```
!ls
```

And you are all set to do Practical 9. However, there appears to be some bugs in the PIL library in colab, which you should remedy with

```
!pip install Pillow==4.0.0
!pip install PIL
!pip install image
```

To tell PyTorch to use the GPU, you will need to add the line "model=model.cuda()" before the training loop. You will need to change Tensors to cuda as well each time before you wrap them with Variable(). For example, instead of "inputs = Variable(inputs)", you should write "inputs = Variable(inputs.cuda())".