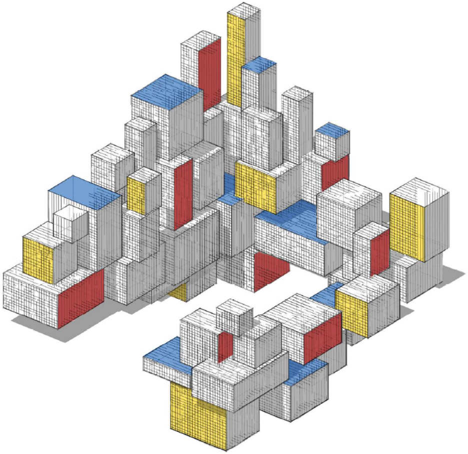
Neural Networks in Python



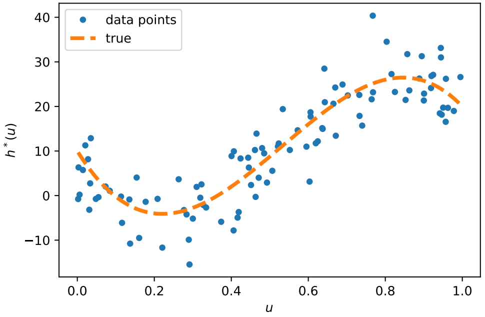
Purpose

In this lecture we will look at some implementations of neural networks in Python:

- We train a neural network with the stochastic gradient descent method using the polynomial regression example, without using any specialized Python packages.
- We consider a realistic application of a neural network to image recognition and classification. Here we use the specialized open-source Python package **`Pytorch`**.

Simple Polynomial Regression

Consider again the polynomial regression data set.



We use a network with architecture

$$[p_0, p_1, p_2, p_3] = [1, 20, 20, 1].$$

Thus, we have two hidden layers with 20 neurons, resulting in a learner with a total of $\dim(\boldsymbol{\theta}) = 481$ parameters.

To implement such a neural network, we first import the **numpy** and the **matplotlib** packages, then read the regression problem data and define the feed-forward neural network layers.

NeuralNetPurePython.py

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

# import data
data = np.genfromtxt('polyreg.csv',delimiter=',')
X = data[:,0].reshape(-1,1)
y = data[:,1].reshape(-1,1)

# Network setup
p = [X.shape[1],20,20,1] # size of layers
L = len(p)-1             # number of layers
```

Next, the `initialize` method generates random initial weight matrices and bias vectors $\{\mathbf{W}_l, \boldsymbol{b}_l\}_{l=1}^{L}$. Specifically, all parameters are initialized with values distributed according to the standard normal distribution.

```python
def initialize(p, w_sig = 1):
    W, b = [[]]*len(p), [[]]*len(p)

    for l in range(1,len(p)):
       W[l]= w_sig * np.random.randn(p[l], p[l-1])
       b[l]= w_sig * np.random.randn(p[l], 1)
    return W,b

W,b = initialize(p) # initialize weight matrices and bias vectors
```

The following code implements the ReLU activation function and the squared error loss. Note that these functions return both the function values and the corresponding gradients.

```
def RELU(z,l):  # RELU activation function: value and derivative
    if l == L: return z, np.ones_like(z)
    else:
        val = np.maximum(0,z) # RELU function (element-wise)
        J = np.array(z>0, dtype = float) # derivative of RELU
        return val, J

def loss_fn(y,g):
    return (g - y)**2, 2 * (g - y)

S = RELU
```

Next, we implement the feed-forward and back-propagation algorithms.

Note the averaging inside the backward-propagation loop.

```
def feedforward(x,W,b):
    a, z, gr_S = [0]*(L+1), [0]*(L+1), [0]*(L+1)

    a[0] = x.reshape(-1,1)
    for l in range(1,L+1):
        z[l] = W[l] @ a[l-1] + b[l] # affine transformation
        a[l], gr_S[l] = S(z[l],l) # activation function
    return a, z, gr_S
```

```python
def backward(W,b,X,y):
    n =len(y)
    delta = [0]*(L+1)
    dC_db, dC_dW = [0]*(L+1), [0]*(L+1)
    loss=0

    for i in range(n): # loop over training examples
        a, z, gr_S = feedforward(X[i,:].T, W, b)
        cost, gr_C = loss_fn(y[i], a[L]) # cost i and gradient wrt g
        loss += cost/n

        delta[L] = gr_S[L] @ gr_C

        for l in range(L,0,-1): # l = L,...,1
            dCi_dbl = delta[l]
            dCi_dWl = delta[l] @  a[l-1].T

            # ---- sum up over samples ----
            dC_db[l] = dC_db[l] + dCi_dbl/n
            dC_dW[l] = dC_dW[l] + dCi_dWl/n
            # -----------------------------

            delta[l-1] =  gr_S[l-1] * W[l].T @ delta[l]

    return dC_dW, dC_db, loss
```

Map the weight matrices and the bias vectors into a single parameter vector, and vice versa.

```python
def list2vec(W,b):
    # converts list of weight matr. and bias vects. into column vect.
    b_stack = np.vstack([b[i] for i in range(1,len(b))] )
    W_stack = np.vstack(W[i].flatten().reshape(-1,1) for i in range(1,
         len(W)))
    vec = np.vstack([b_stack, W_stack])
    return vec

def vec2list(vec, p): # converts vector to weight matr. and bias vect.
    W, b = [[]]*len(p),[[]]*len(p)
    p_count = 0

    for l in range(1,len(p)): # construct bias vectors
       b[l] = vec[p_count:(p_count+p[l])].reshape(-1,1)
       p_count = p_count + p[l]

    for l in range(1,len(p)): # construct weight matrices
      W[l] = vec[p_count:(p_count + p[l]*p[l-1])].reshape(p[l], p[l-1])
      p_count = p_count + (p[l]*p[l-1])

    return W, b
```

Finally, we run the stochastic gradient descent for 10^4 iterations using a minibatch of size 20 and a constant learning rate of $\alpha_t = 0.005$.

```python
batch_size = 20
lr = 0.005
beta = list2vec(W,b)
loss_arr = []
n = len(X)
num_epochs = 10000
print("epoch | batch loss")
print("----------------------------")
for epoch in range(1,num_epochs+1):
    batch_idx = np.random.choice(n,batch_size)
    batch_X = X[batch_idx].reshape(-1,1)
    batch_y=y[batch_idx].reshape(-1,1)
    dC_dW, dC_db, loss = backward(W,b,batch_X,batch_y)
    d_beta = list2vec(dC_dW,dC_db)
    loss_arr.append(loss.flatten()[0])
    if(epoch==1 or np.mod(epoch,1000)==0):
        print(epoch,": ",loss.flatten()[0])
    beta = beta - lr*d_beta
    W,b = vec2list(beta,p)
```

```python
# calculate the loss of the entire training set
dC_dW, dC_db, loss = backward(W,b,X,y)
print("entire training set loss = ",loss.flatten()[0])
xx = np.arange(0,1,0.01)
y_preds = np.zeros_like(xx)

for i in range(len(xx)):
    a, _, _ = feedforward(xx[i],W,b)
    y_preds[i],  = a[L]

plt.plot(X,y, 'r.', markersize = 4,label = 'y')
plt.plot(np.array(xx), y_preds, 'b',label = 'fit')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.show()
plt.plot(np.array(loss_arr), 'b')
plt.xlabel('iteration')
plt.ylabel('Training Loss')
plt.show()
```

```
epoch | batch loss
----------------------------
1 :  158.6779278688539
1000 :  54.52430507401445
2000 :  38.346572088604965
3000 :  31.02036319180713
4000 :  22.91114276931535
5000 :  27.75810262906341
6000 :  22.296907007032928
7000 :  17.337367420038046
8000 :  19.233689945334195
9000 :  39.54261478969857
10000 :  14.754724387604416
entire training set loss =  28.904957963612727
```

The left panel of Figure 1 shows a trained neural network with a training loss of approximately 28.9. As seen from the right panel of Figure 1, the algorithm initially makes rapid progress until it settles down into a stationary regime after 400 iterations.

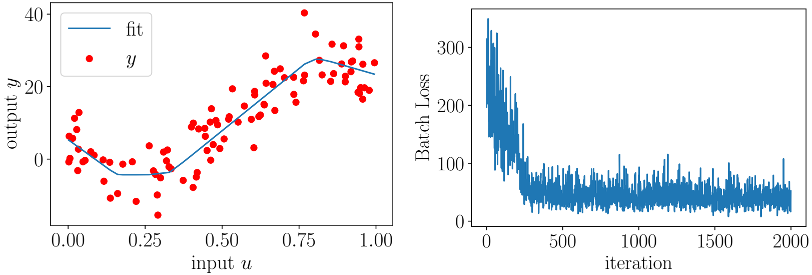


Figure: Left panel: The fitted neural network with training loss of $\ell_\tau(g_\tau) \approx 28.9$. Right panel: The evolution of the estimated loss, $\widehat{\ell_\tau}(g_\tau(\cdot \mid \boldsymbol{\theta}))$, over the steepest-descent iterations.

Image Classification

Next, we use the `Pytorch` package on the Fashion-MNIST data set from
`https://www.kaggle.com/zalando-research/fashionmnist`.
The Fashion-MNIST data set contains 28×28 gray-scale images of clothing.

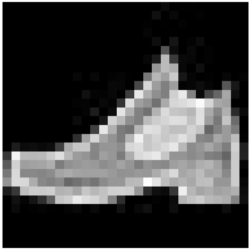
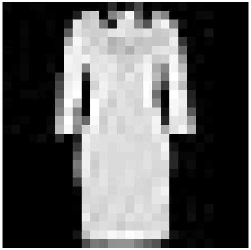



Figure: Left: an ankle boot. Right: a dress.

Classify each image according to its label: T-Shirt, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, and Ankle Boot.

Import the required libraries and load the Fashion-MNIST data set.

ImageClassificationPytorch.py

```
import torch
import torch.nn as nn
from torch.autograd import Variable
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from PIL import Image
import torch.nn.functional as F

class LoadData(Dataset):
    def __init__(self, fName, transform=None):
        data = pd.read_csv(fName)
        self.X = np.array(data.iloc[:, 1:], dtype=np.uint8).reshape(-1,
              1, 28, 28)
        self.y = np.array(data.iloc[:, 0])
    def __len__(self):
        return len(self.X)
    def __getitem__(self, idx):
        img = self.X[idx]
        lbl = self.y[idx]
        return (img, lbl)
```

```
# load the image data
train_ds = LoadData('fashionmnist/fashion-mnist_train.csv')
test_ds  = LoadData('fashionmnist/fashion-mnist_test.csv')

# set labels dictionary
labels = {0 : 'T-Shirt', 1 : 'Trouser', 2 : 'Pullover',
          3 : 'Dress', 4 : 'Coat', 5 : 'Sandal', 6 : 'Shirt',
          7 : 'Sneaker', 8 : 'Bag', 9 : 'Ankle Boot'}
```

Since image input data is generally memory intensive, it is important to partition the data set into (mini-)batches. The code below defines a batch size of 100 images and initializes the **Pytorch** data loader objects.

These objects will be used for efficient iteration over the data set.

```
# load the data in batches
batch_size = 100

train_loader = torch.utils.data.DataLoader(dataset=train_ds,
                                           batch_size=batch_size,
                                           shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_ds,
                                          batch_size=batch_size,
                                          shuffle=True)
```

Next, to define the network architecture in `Pytorch` all we need to do is define an instance of the `torch.nn.Module` class.

Choosing a network architecture with good generalization properties can be a difficult task. Here, we use a network with

- two convolution layers (defined in the `cnn_layer` block),
- a 3×3 kernel, and
- three hidden layers (defined in the `flat_layer` block).

Since there are ten possible output labels, the output layer has ten nodes.

More specifically, the first and the second convolution layers have 16 and 32 output channels.

Combining this with the definition of the 3×3 kernel, we conclude that the size of the first flat hidden layer should be:

$$\Bigg(\overbrace{\underbrace{(28 - 3 + 1)}_{\text{first convolution layer}} - 3 + 1}^{\text{second convolution layer}} \Bigg)^2 \times 32 = 18432,$$

where the multiplication by 32 follows from the fact that the second convolution layer has 32 output channels.

The `flat_fts` variable determines the number of output layers of the convolution block. This number is used to define the size of the first hidden layer of the `flat_layer` block.

The rest of the hidden layers have 100 neurons and we use the ReLU activation function for all layers.

Finally, note that the `forward` method in the `CNN` class implements the forward pass.

```
# define the network
class CNN(nn.Module):
   def __init__(self):
       super(CNN, self).__init__()
       self.cnn_layer = nn.Sequential(
               nn.Conv2d(1, 16, kernel_size=3, stride=(1,1)),
               nn.ReLU(),
               nn.Conv2d(16, 32, kernel_size=3, stride=(1,1)),
               nn.ReLU(),)
       self.flat_fts = (((28-3+1)-3+1)**2)*32

       self.flat_layer = nn.Sequential(
               nn.Linear(self.flat_fts, 100), nn.ReLU(),
               nn.Linear(100, 100), nn.ReLU(),
               nn.Linear(100, 100), nn.ReLU(),
               nn.Linear(100, 10))

   def forward(self, x):
       out = self.cnn_layer(x)
       out = out.view(-1, self.flat_fts)
       out = self.flat_layer(out)
       return out
```

Next, we specify how the network will be trained.

We choose the device type, namely, the central processing unit (CPU) or the GPU (if available), the number of training iterations (epochs), and the learning rate.

Then, we create an instance of the proposed convolution network and send it to the predefined device (CPU or GPU). Note how easily one can switch between the CPU or the GPU without major changes to the code.

In addition to the specifications above, we need to choose an appropriate loss function and training algorithm. Here, we use the cross-entropy loss and the Adam adaptive gradient algorithm.

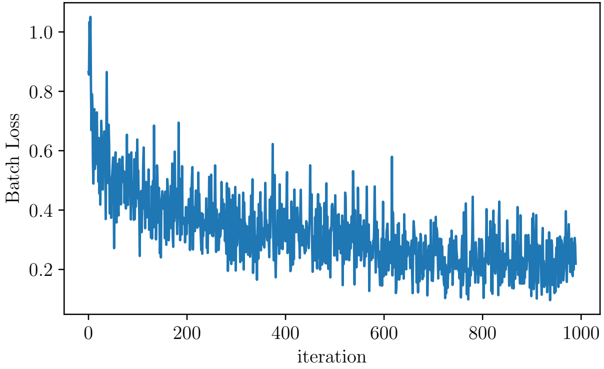
Once these parameters are set, the learning proceeds to evaluate the gradient of the loss function via the back-propagation algorithm.

```
# learning parameters
num_epochs = 50
learning_rate = 0.001

#device = torch.device ('cpu') # use this to run on CPU
device = torch.device ('cuda') # use this to run on GPU

#instance of the Conv Net
cnn = CNN()
cnn.to(device=device)

#loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn.parameters(), lr=learning_rate)
```

```python
# the learning loop
losses = []
for epoch in range(1,num_epochs+1):
    for i, (images, labels) in enumerate(train_loader):
        images = Variable(images.float()).to(device=device)
        labels = Variable(labels).to(device=device)

        optimizer.zero_grad()
        outputs = cnn(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        losses.append(loss.item())
    if(epoch==1 or epoch % 10 == 0):
        print ("Epoch : ", epoch, ", Training Loss: ",  loss.item())
```

```python
# evaluate on the test set
cnn.eval()
correct = 0
total = 0
for images, labels in test_loader:
    images = Variable(images.float()).to(device=device)
    outputs = cnn(images)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted.cpu() == labels).sum()
print("Test Accuracy of the model on the 10,000 training test images: "
     , (100 * correct.item() / total),"%")

plt.rc('text', usetex=True)
plt.rc('font', family='serif',size=20)
plt.tight_layout()
plt.plot(np.array(losses)[10:len(losses)])
plt.xlabel(r'{iteration}',fontsize=20)
plt.ylabel(r'{Batch Loss}',fontsize=20)
plt.subplots_adjust(top=0.8)
plt.show()
```

```
Epoch :  1 , Training Loss:  0.412550151348114
Epoch :  10 , Training Loss:  0.05452106520533562
Epoch :  20 , Training Loss:  0.07233225554227829
Epoch :  30 , Training Loss:  0.01696968264877796
Epoch :  40 , Training Loss:  0.0008199119474738836
Epoch :  50 , Training Loss:  0.006860652007162571
Test Accuracy of the model on the 10,000 training test images: 91.02 %
```


Batch Loss
iteration