Exercise 2 -- Implement and train neural networks

- Implement a feed-forward neural network to perform image classification
- You will train this network using backpropagation.
- Derive and implement the algorithm.
- Train the network using
  - Stochastic gradient descent.
- Implement the same model using PyTorch
Neural networks are function approximators

- **Universal function approximators**
  - Networks with at least one hidden layer can approximate any function*

- Previously - Feature extract + Classifier

- Now → Let the neural network learn this from scratch.
Network architecture
Function fitting- convex optimization

- Need a loss function to measure the task.
- Smooth convex loss-functions are great!
- We will use stochastic gradient descent to optimize our function approximation.
- Compute gradients w.r.t to the loss and change the parameters in the direction of steepest descent.

Gif from https://hackernoon.com/life-is-gradient-descent-880c60ac1be8
Loss function - Cross Entropy loss

- Cross entropy loss

\[ J(u) = \sum_{k=1}^{K} (\neg yk \log u_k - (1 - yk) \log (1 - u_k)) \]

- Measures the conditional entropy between predicted label and the true label.

- Lower loss implies predicted and true labels are close to each other.
Side-note → Differentiability

If $f$ is smooth and $g$ is smooth, then $g \circ f$ is also smooth.

“Smooth”:
- differentiable, twice differentiable, ..., infinitely differentiable ($C^\infty$).
- continuously differentiable ($C^1$), twice continuously differentiable ($C^2$), ..., infinitely differentiable ($C^\infty$).

Our neural network is $C^\infty$. 

Slide credit - Seong Joon Oh
Backpropagation

- How do you change the weights to optimize the loss?
  - Since we use gradient descent, we compute the gradient of the loss function w.r.t each weight.

- Simply apply chain rule to compute the gradients.

\[
f : \mathbb{R}^M \rightarrow \mathbb{R}^N \\
g : \mathbb{R}^L \rightarrow \mathbb{R}^M
\]

\[
\frac{\partial (f \circ g)_i(x)}{\partial x_k} \bigg|_{x=u} = \sum_{j=1}^{M} \frac{\partial f_i(y)}{\partial y_j} \bigg|_{y=g(u)} \frac{\partial g_j(x)}{\partial x_k} \bigg|_{x=u}
\]
Example

\[ f(y) = \sum_{p=1}^{3} y_p^2 \]

\[ g_p(x) = \sum_{q=1}^{2} w_{pq} x_q^2 \]
Example, continued

\[
\frac{\partial f(y)}{\partial y_j} = \frac{\partial}{\partial y_j} \sum_{p=1}^{3} y_p^2
\]

\[
= \sum_{p=1}^{3} \frac{\partial}{\partial y_j} y_p^2
\]

\[
= \sum_{p=1}^{3} 2y_p \delta_{jp}
\]

\[
= 2y_j
\]
Example, continued

\[ \frac{\partial g_j(x)}{\partial x_k} = \frac{\partial}{\partial x_k} \sum_{q=1}^{2} w_{jq} x_q^2 \]

\[ = \sum_{q=1}^{2} \frac{\partial}{\partial x_k} (w_{jq} x_q^2) \]

\[ = \sum_{q=1}^{2} (2w_{jq} x_q \delta_{kq}) \]

\[ = 2w_{jk} x_k \]
Example, continued

\[
\frac{\partial (f \circ g) (x)}{\partial x_k} \bigg|_{x=u} = \sum_{j=1}^{3} \left. \frac{\partial f(y)}{\partial y_j} \right|_{y=g(u)} \left. \frac{\partial g_j(x)}{\partial x_k} \right|_{x=u}
\]

\[
= \sum_{j=1}^{3} 2g_j(u) (2w_{jk}u_k)
\]

\[
= 4u_k \sum_{j=1}^{3} w_{jk}g_j(u)
\]

\[
= 4u_k \sum_{j=1}^{3} w_{jk} \sum_{q=1}^{2} w_{jq}u_q^2
\]
Numerical Gradients

- Wiggle the parameters and compute gradients numerically

\[
\frac{\partial \tilde{J}}{\partial \theta_p}(\theta) \approx \frac{\tilde{J}(\theta + \epsilon e_p) - \tilde{J}(\theta - \epsilon e_p)}{2\epsilon}
\]

- Can do this for all parameters in the network.

- Too slow for practical use in training but great for verifying backpropagation equations.
Batch gradient descent

- Once you have the gradients you can update the parameters.
  \[ v \leftarrow -\alpha \nabla_{\theta} \tilde{J}(\theta^{(t-1)}) ; \]
  \[ \theta^{(t)} \leftarrow \theta^{(t-1)} + v ; \]

  -ve sign to decrease the loss  Averaged over all training samples

- Guaranteed to converge to local minima.
- Very slow since parameters are updated once for each pass on the data.
- Large memory consumption on large datasets.
Stochastic Gradient descent

- Compute the gradients for every sample and update instantly.
- Fast and low memory consumption.
- Can be noisy.
- But noise is good! Can again avoid getting stuck in local minima.
- Better generalization properties*

Visualization from https://wikidocs.net/3413
Understanding and debugging training dynamics

Stochastic descent

On the toy dataset

On the CIFAR-10 dataset
Hyper-parameter tuning

- **Underfitting** → Increase model capacity, decrease regularization
- **Overfitting** → Decrease model capacity, increase regularization
- **Slow learning** → increase learning rate, check initialization for saturation
- **Unstable learning** → decrease learning rate
PyTorch - Quick Introduction

Rakshith Shetty - 29/04/2019

Some slides borrowed from:
What is it?

Tensors and Dynamic neural networks in Python with strong GPU acceleration.

PyTorch is a deep learning framework that puts Python first.

We are in an early-release Beta. Expect some adventures.
What is it?

- A library that allows tensor based computation (like matlab/numpy)
  - Easily run on GPU or CPU.
  - **Do automatic differentiation! Very useful for backpropagation**
  - One of the fastest (maybe caffe is a bit faster)
  - Several library functions which allows you to quickly

- What’s different to other platforms?
  - **Dynamic computational graphs**
  - Very useful when dealing with recurrent networks or other wacky architectures

```
from torch.autograd import Variable
x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```
Basics

```python
import numpy as np
import torch

# Task: compute matrix multiplication C = AB

d = 3000

# using numpy
A = np.random.rand(d, d).astype(np.float32)
B = np.random.rand(d, d).astype(np.float32)
C = A.dot(B)

# using torch with gpu
A = torch.rand(d, d).cuda()
B = torch.rand(d, d).cuda()
C = torch.mm(A, B)
```

350 ms

0.1 ms
Auto-differentiate

```python
import torch
from torch.autograd import Variable

# Task: compute d(||x||^2)/dx
x = Variable(torch.arange(1., 6.), requires_grad=True)
print(x.data)  # x.data = [1, 2, 3, 4, 5]

f = x.dot(x)
pf = f.data  # f.data = 55

f.backward()
print(x.grad)  # x.grad = [2, 4, 6, 8, 10]
```
Auto-differentiate

```python
import torch
from torch.autograd import Variable

# Task: compute d(||x||^2)/dx
x = Variable(torch.range(1, 5), requires_grad=True)
print(x.data) # x.data = [1, 2, 3, 4, 5]

f = x.dot(x)
print(f.data) # f.data = 55

f.backward()
print(x.grad) # x.grad = [2, 4, 6, 8, 10]
```
# Components

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>torch</td>
<td>a Tensor library like NumPy, with strong GPU support</td>
</tr>
<tr>
<td>torch.autograd</td>
<td>a tape based automatic differentiation library that supports all differentiable Tensor operations in torch</td>
</tr>
<tr>
<td>torch.nn</td>
<td>a neural networks library deeply integrated with autograd designed for maximum flexibility</td>
</tr>
<tr>
<td>torch.optim</td>
<td>an optimization package to be used with torch.nn with standard optimization methods such as SGD, RMSProp, LBFGS, Adam etc.</td>
</tr>
<tr>
<td>torch.multiprocessing</td>
<td>python multiprocessing, but with magical memory sharing of torch Tensors across processes. Useful for data loading and hogwild training.</td>
</tr>
<tr>
<td>torch.utils</td>
<td>DataLoader, Trainer and other utility functions for convenience</td>
</tr>
</tbody>
</table>
Sample Code

Create Layers

Initialize weights

Do forward computations
Useful resources

- Official documentation
  - http://pytorch.org/docs/

- Tutorials
  - http://pytorch.org/tutorials/
  - https://github.com/pytorch/tutorials
  - http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html (Useful)

- Example projects
  - https://github.com/pytorch/examples
Submission

- Next week, Friday midnight (10/05/2018 23:59)
- Send to rshetty@mpi-inf.mpg.de
- One zip file per team
- Do not send the dataset
- Solutions next tutorial

Questions?