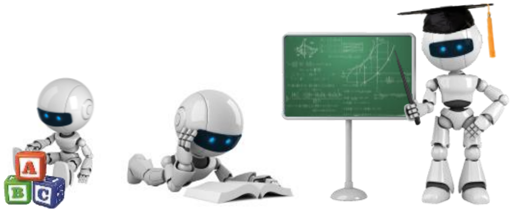
CSCI 544
Applied Natural Language Processing

Mohammad Rostami
USC Computer Science Department

Based on slides prepared by I-Hung Hsu and Usman Roshan



Logistical Notes

- Project Topics
  - You propose the topic: an ongoing project, improving existing algorithms, applying existing tools on new applications.
  - Paper improvement and exploration
- Computing Resources: be realistic
- Coordinate and meet with your advisor

Logistical Notes

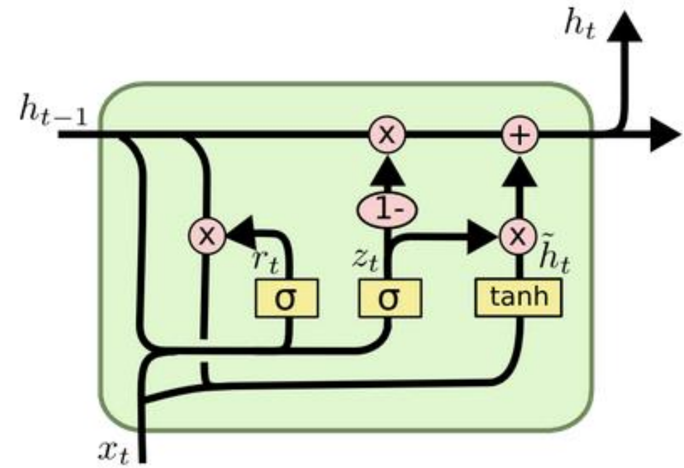
- Proposals: 10/10
  - 2 Pages in ACL format
- Sections:
  - Project Domain and Goals
  - Related Work
  - Datasets
  - Technical Challenge

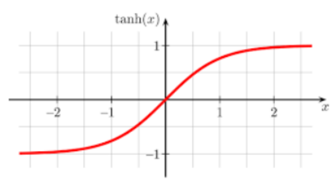
Logistical Notes

- Midterm Exam: 10/17
  - Similar to quizzes but longer
  - Will be taken remotely
  - More details will be provided on Piazza

Gated RNN

- Gated recurrent unit: can learn long-range dependencies
- Control mechanism on information flow
- Gates control information flow
- Resent and Update gates are often close to either 0 or 1 due to using sigmoid
- New gate is used as a preliminary candidate to update the state variable





Reset Gate $$r_t = \sigma(W_{ir}x_t + W_{hr}h_{(t-1)})$$

Update Gate $$z_t = \sigma(W_{iz}x_t + W_{hz}h_{(t-1)})$$

New Gate $$\tilde{h}_t = \tanh(W_{in}x_t + r_t \odot (W_{hn}h_{(t-1)}))$$

$$h_t = (1 - z_t) \odot h_{(t-1)} + z_t \odot \tilde{h}_t$$

Hadamard Product

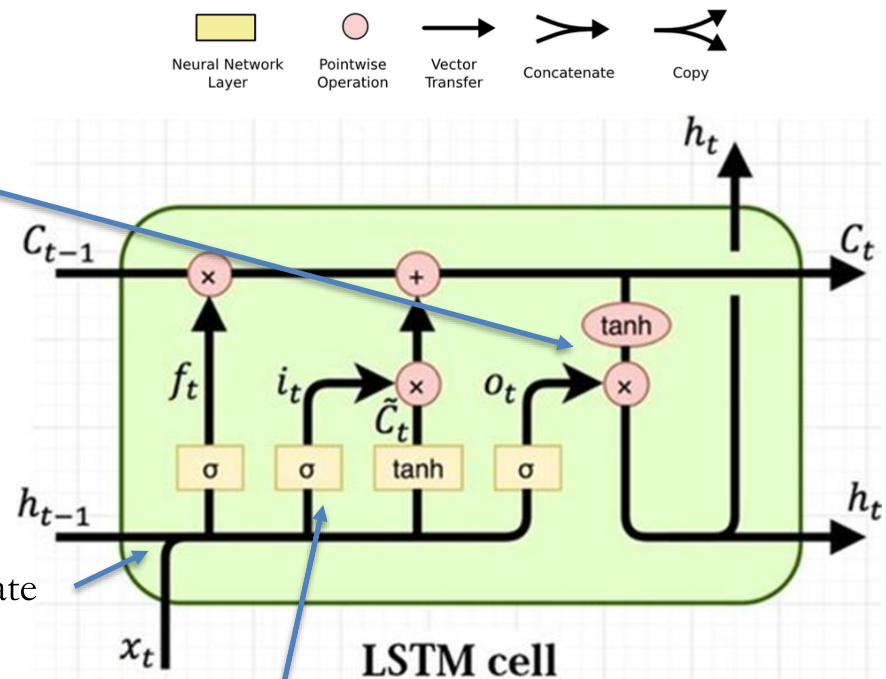
Long Short Term Memory Network

LSTM is designed to resolve the long dependency problem

- Hochreiter and Schmidhuber, Long short-term memory. Neural computation, 1997.

Output Gate
$$o_t = \sigma\left(x_t U^o + h_{t-1} W^o\right)$$
$$h_t = \tanh(C_t) * o_t$$

Cell State: Models memory
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$



$$f_t = \sigma\left(x_t U^f + h_{t-1} W^f\right)$$ Forget Gate

Input Gate
$$i_t = \sigma\left(x_t U^i + h_{t-1} W^i\right)$$
$$\tilde{C}_t = \tanh\left(x_t U^g + h_{t-1} W^g\right)$$

- It’s a python-based scientific computing package
- A library that helps using the power of GPUs
- Why popular?
- PyTorch vs TensorFlow

Installation

- Follow instructions in https://pytorch.org/get-started/locally/

PyTorch Build	Stable (1.9.0)	Preview (Nightly)	LTS (1.8.2)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python	C++ / Java		
Compute Platform	CUDA 10.2	CUDA 11.1	ROCm 4.2 (beta)	CPU
Run this Command:	`conda install pytorch torchvision torchaudio cudatoolkit=10.2 -c pytorch`			

PyTorch

PyTorch data structure format

- Tensor: similar to Numpy array but runs on GPU or other hardware accelerators.
- (Trainable) Tensor: stores data and gradient and can be considered similar to a variable

$$y = \beta^T x$$

- Module: A neural network layer; may store state or learnable weights

Tensors vs Numpy Arrays

PyTorch Tensors are just like numpy arrays, but they can run on GPU.

Tensor operations: x.mm(), torch.add(x,y), etc

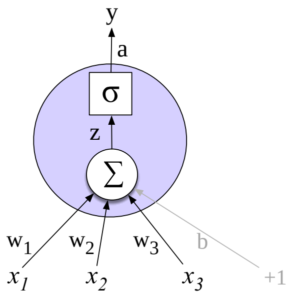
Numpy Conversion: x.numpy()

Training Feedforward Networks

- For every training data point (x, y)
  - Run *forward* computation to find model estimate $\hat{y}$
  - Run *backward* computation to update weights:
    - For every output node
  - Compute loss L between true y and the estimated $\hat{y}$
  - For every weight w from hidden layer to the output layer

Update the weight using gradient descent $\frac{d}{dw} L(f(x; w), y)$

- For all other nodes
  - Assess how much blame it deserves for the current answer



$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i}$$

Example

Here a two-layer net is trained using PyTorch Tensors.

- Forward Pass

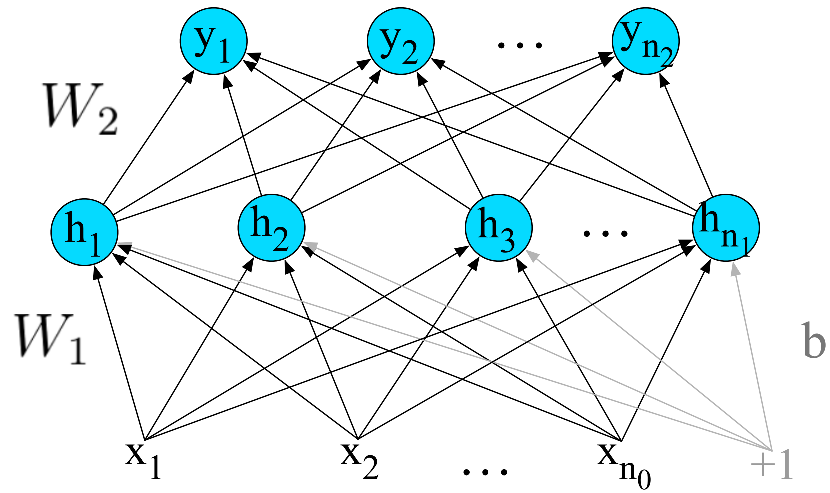
$$\mathcal{L} = \|y - \hat{y}\|_2^2$$

$$y = W_2^T ReLu(W_1^T x)$$

- Backword Pass

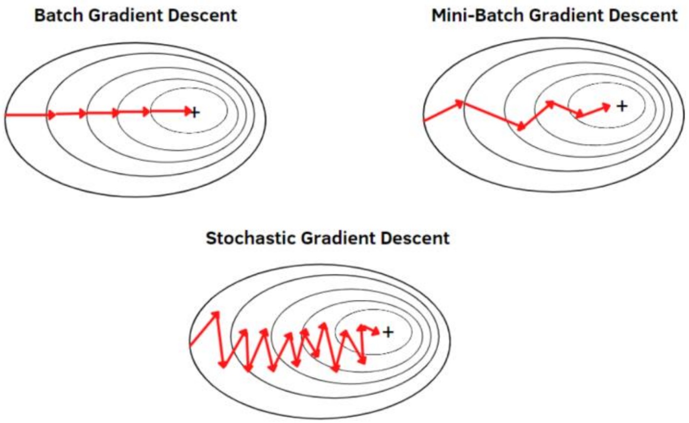
$$W^{i+1} = W^i - \eta \frac{d\mathcal{L}}{dW^i}$$

$$\frac{d\mathcal{L}}{dW_2} = 2(y - \hat{y})\frac{d\hat{y}}{dW_2}$$



Variants of gradient descent

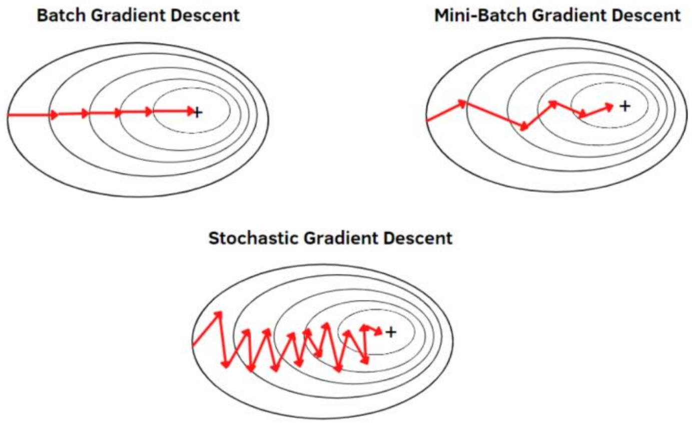
- Stochastic gradient descent




```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

Variants of gradient descent

- Mini batch SGD



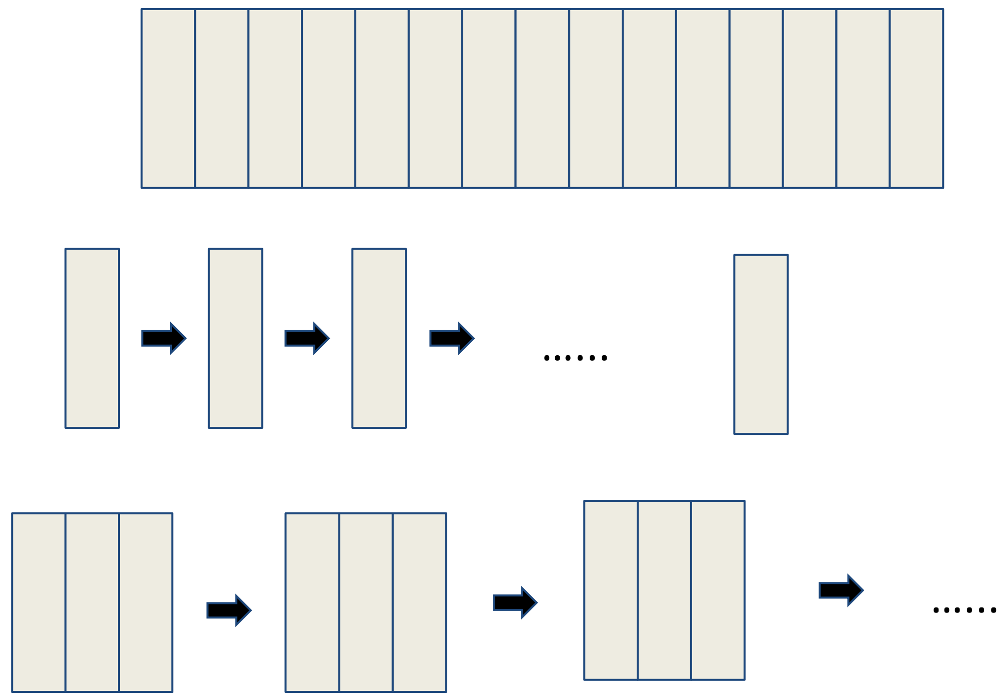

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

(Mini)batching

All your training data:

Standard for loop:

(Mini) batching:



Optimization with Tensors

Create random tensor for data and weight

Tensor can also be loaded by:

1. Load from data (list)

```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

2. From Numpy array

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

lr = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred-y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred-y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= lr*grad_w1
    w2 -= lr*grad_w2
```

Optimization with Tensors

Forward pass: compute predictions and loss

```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

lr = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred-y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred-y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= lr*grad_w1
    w2 -= lr*grad_w2
```

Optimization with Tensors

Backward pass: manually computed gradients ***if you don't have autograd***.

```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

lr = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred-y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred-y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= lr*grad_w1
    w2 -= lr*grad_w2
```

Optimization with Tensors

Optimization: Gradient descent step on weights

```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

lr = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred-y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred-y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= lr*grad_w1
    w2 -= lr*grad_w2
```

Optimization with Tensors

To run on GPU, just cast tensors to a cuda datatype

```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

lr = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred-y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred-y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= lr*grad_w1
    w2 -= lr*grad_w2
```

Autograd

The previous process:

- Slow
- Gradient is hard to compute when model becomes more complex

=> **This is a primary reason for popularity of PyTorch**

Trainable Tensors:

Set “required_grad” to True to enable torch.autograd

Optimization with Autograd

We set the weights' "requires_grad" to be True

```
dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

w3 = w1.detach().clone()
w3.requires_grad=True
# To initialize, you can use:
# w3 = torch.randn(D_in, H, requires_grad=True).type(dtype)

w4 = w2.detach().clone()
w4.requires_grad=True
```

Optimization with Autograd

Forward pass looks exactly the same as the Tensor/Numpy version.

```
lr = 1e-6
for t in range(500):
    y_pred2 = x.mm(w3).clamp(min=0).mm(w4)
    loss = (y_pred2-y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w3 -= lr * w3.grad
        w4 -= lr * w4.grad

    w3.grad.zero_()
    w4.grad.zero_()
```

Optimization with Autograd

But the gradient of loss with respect to w3 and w4 can be done by a simple one-line code.

```
lr = 1e-6
for t in range(500):
    y_pred2 = x.mm(w3).clamp(min=0).mm(w4)
    loss = (y_pred2-y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w3 -= lr * w3.grad
        w4 -= lr * w4.grad

    w3.grad.zero_()
    w4.grad.zero_()
```

```
grad_y_pred = 2.0 * (y_pred-y)
grad_w2 = h_relu.t().mm(grad_y_pred)
grad_h_relu = grad_y_pred.mm(w2.t())
grad_h = grad_h_relu.clone()
grad_h[h < 0] = 0
grad_w1 = x.t().mm(grad_h)
```

Optimization with Autograd

Make gradient step on weights.

What's torch.no_grad()?

- We need to use NO_GRAD to keep the update out of the gradient computation

```
lr = 1e-6
for t in range(500):
    y_pred2 = x.mm(w3).clamp(min=0).mm(w4)
    loss = (y_pred2-y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w3 -= lr * w3.grad
        w4 -= lr * w4.grad

    w3.grad.zero_()
    w4.grad.zero_()
```

Optimization with Autograd

grad.zero()?

- After each gradient descent step, we should restart from zero
- Why is that? Gradients are accumulated

```
lr = 1e-6
for t in range(500):
    y_pred2 = x.mm(w3).clamp(min=0).mm(w4)
    loss = (y_pred2-y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w3 -= lr * w3.grad
        w4 -= lr * w4.grad

    w3.grad.zero_()
    w4.grad.zero_()
```

New Autograd Functions

We can define our own autograd functions by writing forward and backward for Tensors

```
class ReLU(torch.autograd.Function):

    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```

New Autograd Functions

```python
class ReLU(torch.autograd.Function):

    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```

```python
lr = 1e-6
for t in range(500):
    y_pred3 = ReLU.apply(x.mm(w5)).mm(w6)
    loss = (y_pred3-y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w5 -= lr * w5.grad
        w6 -= lr * w6.grad

    w5.grad.zero_()
    w6.grad.zero_()
```

We then apply our new function in the forward pass.

Networks: torch.nn

- High-level wrapper for creating neural networkss
- This is another reason behind popularity of PyTorch and TensorFlow
- Various classes of neural networks can be built using buil-in modules, e.g., CNN, RNN, LSTM, transformers, etc
- A diverse set of hyperparamters are buil-in implemnted, e.g., various activation function, loss functions, etc.

torch.nn

Define our model as a sequence of layers

NN also can be used for common loss functions

```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)
)
model.cuda()

loss_fn = torch.nn.MSELoss(reduction='sum')

def weights_init(m):
    if isinstance(m, torch.nn.Linear):
        torch.nn.init.zeros_(m.weight)
        torch.nn.init.ones_(m.bias)

model.apply(weights_init)
print(model)

lr = 1e-6

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param.data -= lr * param.grad.data
```

torch.nn

Forward pass:

- Feed data to model
- Use prediction to and ground truth to get loss function

```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)
)
model.cuda()

loss_fn = torch.nn.MSELoss(reduction='sum')

def weights_init(m):
    if isinstance(m, torch.nn.Linear):
        torch.nn.init.zeros_(m.weight)
        torch.nn.init.ones_(m.bias)

model.apply(weights_init)
print(model)

lr = 1e-6

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param.data -= lr * param.grad.data
```

torch.nn

Pytorch generates autograd easily

We should use model.zero_grad() to clear all gradients for the parameters in the model

```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)
)
model.cuda()

loss_fn = torch.nn.MSELoss(reduction='sum')

def weights_init(m):
    if isinstance(m, torch.nn.Linear):
        torch.nn.init.zeros_(m.weight)
        torch.nn.init.ones_(m.bias)

model.apply(weights_init)
print(model)

lr = 1e-6

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param.data -= lr * param.grad.data
```

Optimizer

Make gradient step on each model parameter.

Question:

- How can we apply more advanced rules for updating

```python
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)
)
model.cuda()

loss_fn = torch.nn.MSELoss(reduction='sum')

def weights_init(m):
    if isinstance(m, torch.nn.Linear):
        torch.nn.init.zeros_(m.weight)
        torch.nn.init.ones_(m.bias)

model.apply(weights_init)
print(model)

lr = 1e-6

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param.data -= lr * param.grad.data
```

Optimizer

Call nn.optim package, which contains various advanced optimizer other than SGD.

Now, all the parameters can be updated via one-line code.

```
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    #model.zero_grad()
    optimizer.zero_grad()
    loss.backward()

    optimizer.step()
#     with torch.no_grad():
#         for param in model.parameters():
#             param.data -= lr * param.grad.data
```

Define new modules

Pytorch **Module** is *a neural network layer*, it can contain weights or other modules.

```
class TwoLayerMLP(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super().__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)
        self.weight_init()

    def weight_init(self):
        torch.nn.init.zeros_(self.linear1.weight)
        torch.nn.init.zeros_(self.linear2.weight)
        torch.nn.init.ones_(self.linear1.bias)
        torch.nn.init.ones_(self.linear2.bias)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

model = TwoLayerMLP(D_in, H, D_out)
model.cuda()
print(model)
loss_fn = torch.nn.MSELoss(reduction='sum')
lr = 1e-6

optimizer = torch.optim.Adam(model.parameters(), lr=lr)
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

DataLoaders

A **DataLoader** wraps a **Dataset** and provides minibatching, multithreading, etc.

When you need to load custom data, just **write your own Dataset class**

```
import torch
from torch.utils import data

class Dataset(data.Dataset):
  'Characterizes a dataset for PyTorch'
  def __init__(self, list_IDs, labels):
        'Initialization'
        self.labels = labels
        self.list_IDs = list_IDs

  def __len__(self):
        'Denotes the total number of samples'
        return len(self.list_IDs)

  def __getitem__(self, index):
        'Generates one sample of data'
        # Select sample
        ID = self.list_IDs[index]

        # Load data and get label
        X = torch.load('data/' + ID + '.pt')
        y = self.labels[ID]

        return X, y
```

Adapt Dataset to DataLoaders

```
# Parameters
params = {'batch_size': 64,
          'shuffle': True,
          'num_workers': 6}
max_epochs = 100

# Datasets
partition = # IDs
labels = # Labels

# Generators
training_set = Dataset(partition['train'], labels)
training_generator = data.DataLoader(training_set, **params)

validation_set = Dataset(partition['validation'], labels)
validation_generator = data.DataLoader(validation_set, **params)

# Loop over epochs
for epoch in range(max_epochs):
    # Training
    for local_batch, local_labels in training_generator:
        # Transfer to GPU
        local_batch, local_labels = local_batch.to(device), local_labels.to(device)

        # Model computations
        [...]
```

DataLoader perform batching automatically

Summary

1. Prepare you data
   a. Write your own Dataset (inherit torch.nn.util.dataset)
2. Create your model
   a. A sequential module if your model is super easy and will not be reused.
   b. A nn.Module module
3. Write the loop (how many epoch/steps) to train your model:
   a. Create a DataLoader that wraps the Dataset you provide
   b. Set an optimizer
   c. Set a loss for optimization
   d. For loop….
      i. Forward pass
      ii. Zero-grad
      iii. Backward pass => Get gradient
      iv. Optimizer step to update your models' weight.

Using Different Version of Data for Efficieny

- Tiny-size: for debugging syntactic bug
- Small-size: check the behavior of the model
- Mid-size: for understanding model behavior and fast development
- Full-size: conduct final experiments

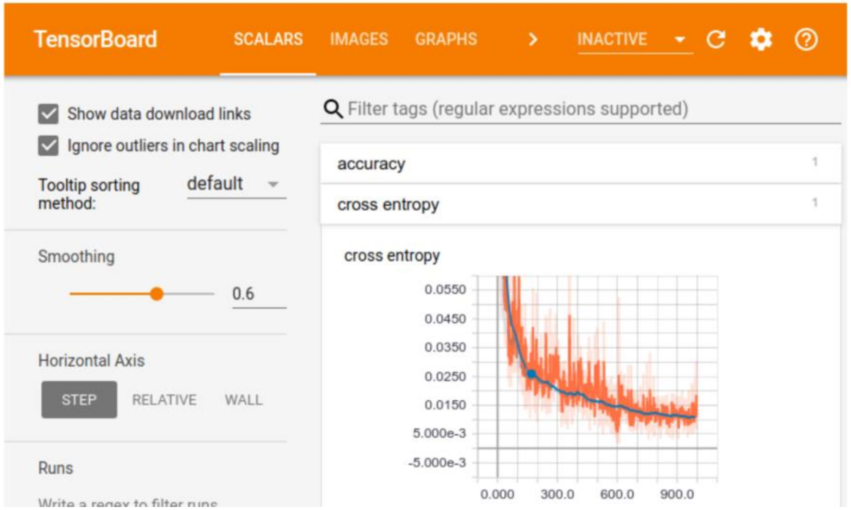
Tensorboard

- Installation: https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html
- Create a Summary Writer

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter()
```

- During Training/Dev/Test

```
writer.add_scalar("Loss/train", loss, epoch)
```



End to end Examples

- https://www.analyticsvidhya.com/blog/2020/01/first-text-classification-in-pytorch/
- https://towardsdatascience.com/lstm-text-classification-using-pytorch-2c6c657f8fc0