

## Example

- Feed-forward language model
- Full example

### Code

```
https://github.com/neubig/nl4nlp-code/blob/master/02-lm-pytorch/nl-lm-batch.py
```



# Frameworks

Computational Linguistics: Jordan Boyd-Graber  
University of Maryland  
NEURAL NETWORKS IN PYTORCH

Slides adapted from Soumith Chintala

## Data and Model (should be familiar)

### Data

$x_1$	$x_2$	$y$
1.00	1.00	0.00
1.00	0.00	1.00
0.00	0.00	0.00
0.00	1.00	1.00

### First Layer

$$w^{(1)} = \begin{bmatrix} 1.00 & 1.00 \\ 1.00 & 1.00 \end{bmatrix} \quad (1)$$

$$b^{(1)} = [0.00 \quad 1.00] \quad (2)$$

### Second Layer

$$w^{(2)} = [0.00 \quad 0.00] \quad (3)$$

$$b^{(2)} = 0.00 \quad (4)$$

Using ReLU as non-linearity

## ReLu Model

## ReLu Model

```
class ReLuModel(nn.Module):  
    def __init__(self, input_size=2, hidden_dim=2, num_classes=10):  
        super(ReLuModel, self).__init__()  
        self.first_layer = nn.Linear(input_size, hidden_dim)  
        self.second_layer = nn.Linear(hidden_dim, num_classes)  
  
    def forward(self, x):  
        first_layer = self.first_layer(x)  
        first_activation = first_layer.clamp(min=0)  
        second_layer = self.second_layer(first_activation)  
        out = second_layer.clamp(min=0)  
        return out
```

# Learning

## Learning

```
loss_func = nn.MSELoss()
optimizer = torch.optim.SGD(relu.parameters(), lr=0.1)
for ii in range(kITER):
    for x, y in zip(data_x, data_y):
        optimizer.zero_grad()
        loss = loss_func(relu.forward(x), y)
        loss.backward()
        optimizer.step()
```

## Inspecting Model

```
def model_string(self):  
    first_layer = "%0.2f %0.2f\n%0.2f %0.2f\n" % \  
        (self.first_layer.weight.data[0][0],  
         self.first_layer.weight.data[0][1],  
         self.first_layer.weight.data[1][0],  
         self.first_layer.weight.data[0][1])  
  
    first_bias = "%0.2f %0.2f\n" % \  
        (self.first_layer.bias.data[0],  
         self.first_layer.bias.data[1])  
  
    second_layer = "%0.2f %0.2f\n" % \  
        (self.second_layer.weight.data[0][0],  
         self.second_layer.weight.data[0][1])  
  
    second_bias = "%0.2f" % self.second_layer.bias.data[0]  
  
    return "FL: " + first_layer + "FB: " + first_bias + "
```



## Inspecting Model

```
FL: 1.00 1.00  
1.00 1.00  
FB: 0.00 1.00  
SL: 0.00 0.00  
SB: 0.00
```

## Running Computation Forward

```
>>> x = torch.Tensor(1, 5)
>>> x
tensor([[ 0.0000, -0.0000,  0.0000, -0.0000,  0.0000]])
>>> x = x*0 + 1
>>> x
tensor([[1., 1., 1., 1., 1.]])
>>> model.forward(x)
tensor([[ -0.2263,  0.5485]], grad_fn=<ThAddmmBackward>)
```

## Models and Parameters

- **Parameters** are the things that we optimize over (vectors, matrices).
- **Model** is a collection of parameters.
- **Parameters** out-live the computation graph.

## Modules allow computation graph

- Each module must implement forward function
- If forward function just uses built-in modules, autograd works
- If not, you'll need to implement backward function (i.e., backprop)

## Modules allow computation graph

- Each module must implement forward function
- If forward function just uses built-in modules, autograd works
- If not, you'll need to implement backward function (i.e., backprop)
  - input: as many Tensors as outputs of module (gradient w.r.t. that output)
  - output: as many Tensors as inputs of module (gradient w.r.t. its corresponding input)
  - If inputs do not need gradient (static) you can return None

## Trainers and Backprop

- Initialize a Optimizer with a given model's parameter
- Get output for an example / minibatch
- Compute loss and backpropagate
- Take step of Optimizer
- Repeat ...

## Trainers and Backprop

```
model = dy.Model()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# Training the Model
for epoch in range(num_epochs):
    for i, (Variable(doc), Variable(label)) in enumerate(training_data_loader):
        optimizer.zero_grad()
        outputs = model(doc)
        loss = nn.CrossEntropyLoss(outputs, label)
        loss.backward()
        optimizer.step()
```

## Options for Optimizers

Adadelata

Adagrad

Adam

LBFGS

SGD

Closure (LBFGS), learning rate, etc.



## Multilayer Perceptron for XOR

- Model

$$\hat{y} = \sigma(\hat{v} \cdot \tanh(U \vec{x} + b)) \quad (5)$$

- Loss

$$\ell = \begin{cases} -\log \hat{y} & \text{if } y = 0 \\ -\log(1 - \hat{y}) & \text{if } y = 1 \end{cases} \quad (6)$$

## Imports and Data

```
import dynet as dy
import random

data = [ ([0,1],0),
          ([1,0],0),
          ([0,0],1),
          ([1,1],1) ]
```

## Create Model

```
model = dy.Model()  
pU = model.add_parameters((4, 2))  
pb = model.add_parameters(4)  
pv = model.add_parameters(4)  
  
trainer = dy.SimpleSGDTrainer(model)  
closs = 0.0
```

```

for x,y in data:
    # create graph for computing loss
    dy.renew_cg()
    U = dy.parameter(pU)
    b = dy.parameter(pb)
    v = dy.parameter(pv)
    x = dy.inputVector(x)
    # predict
    yhat = dy.logistic(dy.dot_product(v,dy.tanh(U*x+b)))
    # loss
    if y == 0:
        loss = -dy.log(1 - yhat)
    elif y == 1:
        loss = -dy.log(yhat)

    closs += loss.scalar_value() # forward
    loss.backward()
    trainer.update()

```

```

for x,y in data:
    # create graph for computing loss
    dy.renew_cg()
    U = dy.parameter(pU)
    b = dy.parameter(pb)
    v = dy.parameter(pv)
    x = dy.inputVector(x)
    # predict
    yhat = dy.logistic(dy.dot_product(v,dy.tanh(U*x+b)))
    # loss
    if y == 0:
        loss = -dy.log(1 - yhat)
    elif y == 1:
        loss = -dy.log(yhat)

    closs += loss.scalar_value() # forward
    loss.backward()
    trainer.update()

```

**Important:** loss expression defines objective you're optimizing

## Key Points

- Create computation graph for each example.
- Graph is built by composing expressions.
- Functions that take expressions and return expressions define graph components.

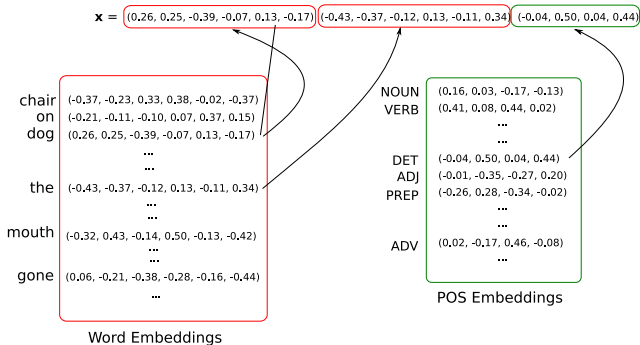
## Word Embeddings and Lookup Parameters

- In NLP, it is very common to use feature embeddings
- Each feature is represented as a  $d$ -dim vector
- These are then summed or concatenated to form an input vector
- The embeddings can be pre-trained
- But they are usually trained (fine-tuned) with the model

# "feature embeddings"

$w=\text{dog}$        $pw=\text{the}$        $pt=\text{NOUN}$        $pt=\text{DET}$        $w=\text{dog}\&pt=\text{DET}$        $w=\text{dog}\&pw=\text{the}$        $w=\text{chair}\&pt=\text{DET}$

$\mathbf{x} = (0, \dots, 0, 1, 0, \dots, 0, 1, 0, \dots, 0, 1, 0, \dots, 0, 1, 0, 0, 1, 0, \dots, 0, 0, 0, \dots, 0)$





```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)
word_to_ix = {"hello": 0, "world": 1}
embeds = nn.Embedding(2, 5)  # 2 words in vocab, 5 dimensions
lookup_tensor = torch.tensor([word_to_ix["hello"]], dtype=torch.long)
hello_embed = embeds(lookup_tensor)
```

# Deep Unordered Composition Rivals Syntactic Methods for Text Classification

**Mohit Iyyer,<sup>1</sup> Varun Manjunatha,<sup>1</sup> Jordan Boyd-Graber,<sup>2</sup> Hal Daumé III<sup>1</sup>**

<sup>1</sup>University of Maryland, Department of Computer Science and UMIACS

<sup>2</sup>University of Colorado, Department of Computer Science

{miyyer, varunm, hal}@umiacs.umd.edu, Jordan.Boyd.Graber@colorado.edu

Implementing a non-trivial example ...

## Deep Averaging Network

$w_1, \dots, w_N$



$z_0 = \text{CBOW}(w_1, \dots, w_N)$

$z_1 = g(W_1 z_0 + b_1)$

$z_2 = g(W_2 z_1 + b_2)$

$\hat{y} = \text{softmax}(z_2)$

- Works about as well as more complicated models
- Strong baseline
- Key idea: Continuous Bag of Words

$$\text{CBOW}(w_1, \dots, w_N) = \sum_i E[w_i] \quad (7)$$

- Actual non-linearity doesn't matter, we'll use tanh
- Let's implement in DyNet

## Deep Averaging Network

$$\begin{aligned} &w_1, \dots, w_N \\ &\downarrow \\ &z_0 = \text{CBOW}(w_1, \dots, w_N) \\ &z_1 = g(z_0) \\ &z_2 = g(z_1) \\ &\hat{y} = \text{softmax}(z_2) \end{aligned}$$

### Encode the document

```
def encode_doc(doc):  
    doc = [w2i[w] for w in doc]  
    embs = [E[idx] for idx in doc]  
    return dy.esum(embs)
```

### First Layer

```
def layer1(x):  
    W = dy.parameter(pW1)  
    b = dy.parameter(pb1)  
    return dy.tanh(W*x+b)
```

### Second Layer

```
def layer2(x):  
    W = dy.parameter(pW2)  
    b = dy.parameter(pb2)  
    return dy.tanh(W*x+b)
```

## Deep Averaging Network

Encode the document

```
def encode_doc(doc):  
    doc = [w2i[w] for w in doc]  
    embs = [E[idx] for idx in doc]  
    return dy.esum(embs)
```

First Layer

```
def layer1(x):  
    W = dy.parameter(pW1)  
    b = dy.parameter(pb1)  
    return dy.tanh(W*x+b)
```

Second Layer

```
def layer2(x):  
    W = dy.parameter(pW2)  
    b = dy.parameter(pb2)  
    return dy.tanh(W*x+b)
```

$w_1, \dots, w_N$

↓

$z_0 = \text{CBOW}(w_1, \dots, w_N)$

$z_1 = g(z_0)$

$z_2 = g(z_1)$

$\hat{y} = \text{softmax}(z_2)$

## Deep Averaging Network

Encode the document

```
def encode_doc(doc):  
    doc = [w2i[w] for w in doc]  
    embs = [E[idx] for idx in doc]  
    return dy.esum(embs)
```

First Layer

```
def layer1(x):  
    W = dy.parameter(pW1)  
    b = dy.parameter(pb1)  
    return dy.tanh(W*x+b)
```

Second Layer

```
def layer2(x):  
    W = dy.parameter(pW2)  
    b = dy.parameter(pb2)  
    return dy.tanh(W*x+b)
```

$w_1, \dots, w_N$

↓

$z_0 = \text{CBOW}(w_1, \dots, w_N)$

$z_1 = g(z_0)$

$z_2 = g(z_1)$

$\hat{y} = \text{softmax}(z_2)$

## Deep Averaging Network

$$\begin{aligned} &w_1, \dots, w_N \\ &\quad \downarrow \\ &z_0 = \text{CBOW}(w_1, \dots, w_N) \\ &z_1 = g(z_0) \\ &z_2 = g(z_1) \\ &\hat{y} = \text{softmax}(z_2) \end{aligned}$$

### Loss

```
def do_loss(probs, label):  
    label = label_indicator[label]  
    return -dy.log(dy.pick(probs, label)) # select that index
```

### Putting it all together

```
def predict_labels(doc):  
    x = encode_doc(doc)  
    h = layer1(x)  
    y = layer2(h)  
    return dy.softmax(y)
```

### Training

```
for (doc, label) in data:  
    dy.renew_cg()  
    probs = predict_labels(doc)  
  
    loss = do_loss(probs, label)  
    loss.forward()  
    loss.backward()  
    trainer.update()
```

## Deep Averaging Network

$$\begin{aligned} &w_1, \dots, w_N \\ &\quad \downarrow \\ &z_0 = \text{CBOW}(w_1, \dots, w_N) \\ &z_1 = g(z_0) \\ &z_2 = g(z_1) \\ &\hat{y} = \text{softmax}(z_2) \end{aligned}$$

### Loss

```
def do_loss(probs, label):  
    label = label_indicator[label]  
    return -dy.log(dy.pick(probs, label)) # select that index
```

### Putting it all together

```
def predict_labels(doc):  
    x = encode_doc(doc)  
    h = layer1(x)  
    y = layer2(h)  
    return dy.softmax(y)
```

### Training

```
for (doc, label) in data:  
    dy.renew_cg()  
    probs = predict_labels(doc)  
  
    loss = do_loss(probs, label)  
    loss.forward()  
    loss.backward()  
    trainer.update()
```



## Deep Averaging Network

$$\begin{aligned} &w_1, \dots, w_N \\ &\quad \downarrow \\ &z_0 = \text{CBOW}(w_1, \dots, w_N) \\ &z_1 = g(z_0) \\ &z_2 = g(z_1) \\ &\hat{y} = \text{softmax}(z_2) \end{aligned}$$

### Loss

```
def do_loss(probs, label):  
    label = label_indicator[label]  
    return -dy.log(dy.pick(probs, label)) # select that index
```

### Putting it all together

```
def predict_labels(doc):  
    x = encode_doc(doc)  
    h = layer1(x)  
    y = layer2(h)  
    return dy.softmax(y)
```

### Training

```
for (doc, label) in data:  
    dy.renew_cg()  
    probs = predict_labels(doc)  
  
    loss = do_loss(probs, label)  
    loss.forward()  
    loss.backward()  
    trainer.update()
```

## Summary

- Computation Graph
- Expressions ( $\approx$  nodes in the graph)
- Parameters, LookupParameters
- Model (a collection of parameters)
- Trainers
- Create a graph for each example, compute loss, backdrop, update