Department of Computer Science
UNIVERSITY OF COLORADO **BOULDER**

Frameworks

Advanced Machine Learning for NLP

Jordan Boyd-Graber

NEURAL NETWORKS IN DYNET

Slides adapted from Chris Dyer, Yoav Goldberg, Graham Neubig

Major Players

- Computation Graph
- Expressions (nodes in the graph)
- Parameters
- Model (a collection of parameters)
- Trainer

Computation Graph

```python
import dynet as dy

dy.renew_cg() # create a new computation graph

v1 = dy.inputVector([1,2,3,4])
v2 = dy.inputVector([5,6,7,8])
# v1 and v2 are expressions

v3 = v1 + v2
v4 = v3 * 2
v5 = v1 + 1
v6 = dy.concatenate([v1,v3,v5])
```

Computation Graph

```
import dynet as dy

dy.renew_cg() # create a new computation graph

v1 = dy.inputVector([1,2,3,4])
v2 = dy.inputVector([5,6,7,8])
# v1 and v2 are expressions

v3 = v1 + v2
v4 = v3 * 2
v5 = v1 + 1
v6 = dy.concatenate([v1,v3,v5])

>>> print(v6)
expression 5/1
>>> print(v6.npvalue())
[  1.   2.   3.   4.   6.   8.  10.  12.   2.   3.   4.   5
```

Computation Graph and Expressions

- Create basic expressions.
- Combine them using operations.
- Expressions represent symbolic computations.
- Actual computation:

```
.value()
.npvalue()                    #numpy value
.scalar_value()
.vec_value()                  # flatten to vector
.forward()                    # compute expression
```

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.

Models and Parameters

```
model = dy.Model()

pW = model.add_parameters((2,4))
pb = model.add_parameters(2)

dy.renew_cg()
x = dy.inputVector([1,2,3,4])
W = dy.parameter(pW) # convert params to expression
b = dy.parameter(pb) # and add to the graph

y = W * x + b
```

Inspecting

Let's inspect x, W, b, and y.

Inspecting

Let's inspect x, W, b, and y.

```
>>> x.value()
[1.0, 2.0, 3.0, 4.0]
```

Inspecting

Let's inspect x, W, b, and y.

```
>>> x.value()
[1.0, 2.0, 3.0, 4.0]
```

```
>>> W.value()
array([[ 0.64952731, -0.06049263,  0.90871298, -0.11073416]
       [ 0.75935686,  0.25788534, -0.98922664,  0.20040739]
```

Inspecting

Let's inspect x, W, b, and y.

```
>>> x.value()
[1.0, 2.0, 3.0, 4.0]
```

```
>>> W.value()
array([[ 0.64952731, -0.06049263,  0.90871298, -0.11073416]
       [ 0.75935686,  0.25788534, -0.98922664,  0.20040739]
```

```
>>> b.value()
[-1.5444282293319702, -0.6606682338714 6]
```

Inspecting

Let's inspect x, W, b, and y.

```
>>> x.value()
[1.0, 2.0, 3.0, 4.0]
```

```
>>> W.value()
array([[ 0.64952731, -0.06049263,  0.90871298, -0.11073416]
       [ 0.75935686,  0.25788534, -0.98922664,  0.20040739]
```

```
>>> b.value()
[-1.5444282293319702, -0.6606668233387146]
```

```
>>> y.value()
[1.267316222190857, -1.5515896081924438]
```

Initialization

```
model = dy.Model()

pW = model.add_parameters((4,4))

pW2 = model.add_parameters((4,4),
                      init=dy.GlorotInitializer())

pW3 = model.add_parameters((4,4),
                      init=dy.NormalInitializer(0,1))
```

Glorot Initialization

$$\mathcal{N}(w_i \,|\, w_i, 0) \tag{1}$$

Trainers and Backprop

- Initialize a Trainer with a given model.
- Compute gradients by calling expr.backward() from a scalar node.
- Call trainer.update() to update the model parameters using the gradients.

Trainers and Backprop

```
model = dy.Model()

trainer = dy.SimpleSGDTrainer(model)

p_v = model.add_parameters(10)

for i in xrange(10):
    dy.renew_cg()

    v = dy.parameter(p_v)
    v2 = dy.dot_product(v,v)
    v2.forward()

    v2.backward()  # compute gradients
    trainer.update()
```

Options for Trainers

```
dy.SimpleSGDTrainer(model,...)

dy.MomentumSGDTrainer(model,...)

dy.AdagradTrainer(model,...)

dy.AdadeltaTrainer(model,...)

dy.AdamTrainer(model,...)
```

Training with DyNet

- Create model, add parameters, create trainer.
- For each training example:
  - create computation graph for the loss
  - run forward (compute the loss)
  - run backward (compute the gradients)
  - update parameters

Multilayer Perceptron for XOR

- Model

$$\hat{y} = \sigma(\hat{v} \cdot \text{tanh}(U\vec{x} + b)) \tag{2}$$

- Loss

$$\ell = \begin{cases} -\log \hat{y} & \text{if } y = 0 \\ -\log(1 - \hat{y}) & \text{if } y = 0 \end{cases} \tag{3}$$

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()
```

```python
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

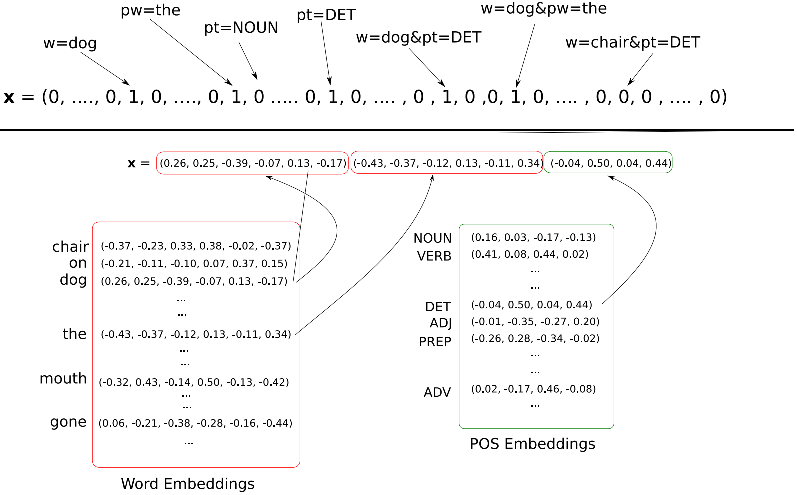
Key Points

- 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.
- They are usually trained with the model.

"feature embeddings"



```python
vocab_size = 10000
emb_dim = 200

E = model.add_lookup_parameters((vocab_size, emb_dim))

dy.renew_cg()
x = dy.lookup(E, 5)
# or
x = E[5]
# x is an expression
```

Deep Unordered Composition Rivals Syntactic Methods for Text Classification

Mohit Iyyer,[1] **Varun Manjunatha,**[1] **Jordan Boyd-Graber,**[2] **Hal Daumé III**[1]
[1]University of Maryland, Department of Computer Science and UMIACS
[2]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

$$w_1, \ldots, w_N$$
$$\downarrow$$
$$z_0 = \text{CBOW}(w_1, \ldots, w_N)$$
$$z_1 = g(z_1)$$
$$z_2 = g(z_2)$$
$$\hat{y} = \text{softmax}(z_3)$$

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

$$\text{CBOW}(w_1, \ldots, w_N) = \sum_i E[w_i] \tag{4}$$

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

Deep Averaging Network

$$w_1, \ldots, w_N$$

$$\downarrow$$

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

$$z_1 = g(z_1)$$

$$z_2 = g(z_2)$$

$$\hat{y} = \text{softmax}(z_3)$$

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)
```

$$w_1, \ldots, w_N$$
$$\downarrow$$
$$z_0 = \text{CBOW}(w_1, \ldots, w_N)$$
$$z_1 = g(z_1)$$
$$z_2 = g(z_2)$$
$$\hat{y} = \text{softmax}(z_3)$$

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)
```

$$w_1, \dots, w_N$$
$$\downarrow$$
$$z_0 = \text{CBOW}(w_1, \dots, w_N)$$
$$z_1 = g(z_1)$$
$$z_2 = g(z_2)$$
$$\hat{y} = \text{softmax}(z_3)$$

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)
```

$$w_1, \ldots, w_N$$

$$\downarrow$$

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

$$z_1 = g(z_1)$$

$$z_2 = g(z_2)$$

$$\hat{y} = \text{softmax}(z_3)$$

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()
```

$$w_1, \ldots, w_N$$
$$\downarrow$$
$$z_0 = \text{CBOW}(w_1, \ldots, w_N)$$
$$z_1 = g(z_1)$$
$$z_2 = g(z_2)$$
$$\hat{y} = \text{softmax}(z_3)$$

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()
```

$$w_1, \ldots, w_N$$

$$\downarrow$$

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

$$z_1 = g(z_1)$$

$$z_2 = g(z_2)$$

$$\hat{y} = \text{softmax}(z_3)$$

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, thenâĂÍcompute loss, backdrop, update.