Frameworks

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

Simple Model

```python
import torch
import torch.nn as nn

class LogisticRegression(nn.Module):
    def __init__(self, input_size, num_classes):
        super(LogisticRegression, self).__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, x):
        out = self.linear(x)
        return out
```

Simple Model

```
>>> model = LogisticRegression(5, 2)
>>> model.parameters
<bound method Module.parameters of LogisticRegression(
  (linear): Linear(in_features=5, out_features=2, bias=True
)>
>>> model.linear.weight
Parameter containing:
tensor([[ 0.0650,  0.0221,  0.1673, -0.1365, -0.1233],
        [-0.1289,  0.2455,  0.3255,  0.0409, -0.1908]], req
>>> model.linear.bias
Parameter containing:
tensor([-0.2208,  0.2562], requires_grad=True)
```

Where did these numbers come from?

```
class Bilinear(Module):
    r"""Applies a bilinear transformation to the incoming d
    :math:`y = x_1 A x_2 + b`
    """

    def reset_parameters(self):
        stdv = 1. / math.sqrt(self.weight.size(1))
        self.weight.data.uniform_(-stdv, stdv)
        if self.bias is not None:
            self.bias.data.uniform_(-stdv, stdv)
```

Where did these numbers come from?

```
class Bilinear(Module):
    r"""Applies a bilinear transformation to the incoming c
    :math:`y = x_1 A x_2 + b`
    """

    def reset_parameters(self):
        stdv = 1. / math.sqrt(self.weight.size(1))
        self.weight.data.uniform_(-stdv, stdv)
        if self.bias is not None:
            self.bias.data.uniform_(-stdv, stdv)
```

Beauty and peril of working with something like PyTorch!

Computation Graph and Expressions

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

```
.value()
.npvalue()                  #numpy value
.scalar_value()
.cuda()                      # move to GPU
.forward()                  # compute expression
```

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>)
```

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

```
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(train_loader):
        optimizer.zero_grad()
        prediction = model(doc)
        loss = nn.CrossEntropyLoss(prediction, label)
        loss.backward()
        optimizer.step()
```

Options for Optimizers

`Adadelta`
`Adagrad`
`Adam`
`LBFGS`
`SGD`

Closure (LBFGS), learning rate, etc.

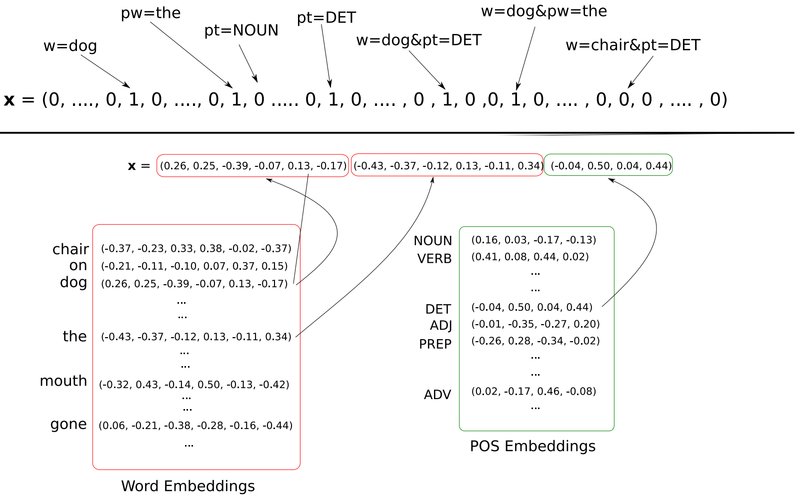
Key Points

- 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-tunded) with the model

"feature embeddings"



```python
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 dim embe
lookup_tensor = torch.tensor([word_to_ix["hello"]],
                             dtype=torch.long)
hello_embed = embeds(lookup_tensor)
```