CS378: Natural Language Processing
Lecture 6: NN Implementation

Greg Durrett

Announcements

‣ A1 due today at 5pm
‣ A2 out late tonight
‣ Goldberg reading link fixed

Recall: Feedforward NNs

\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]

Recall: Training Feedforward NNs

\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]

‣ Maximize log likelihood of training data. For one point:
\[ \mathcal{L}(x, i^*) = \log P(y = i^*|x) = \log (\text{softmax}(Wz) \cdot e_{i^*}) \]

‣ How to compute the gradient with respect to \( W \) and \( V \)?
Recall: Backpropagation

\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]

This Lecture

- Neural net implementation / PyTorch 101
- Neural net training
- Word representations

Implementing Neural Networks: PyTorch 101

Computation Graphs

- Computing gradients is hard!
- Automatic differentiation: instrument code to keep track of derivatives
  \[ y = x \times x \quad \Rightarrow \quad (y, dy) = (x \times x, 2 \times x \times dx) \]
- Computation is now something we need to reason about symbolically
- Use a library like Pytorch or Tensorflow. This class: Pytorch
- Ensuing code examples are on the course website: ffnn_example.py under “Readings”
PyTorch

- Framework for defining computations that provides easy access to derivatives
- Module: defines a neural network (can use wrap other modules which implement predefined layers)
- If forward() uses crazy stuff, you have to write backward yourself

```python
torch.nn.Module
    # Takes an example x and computes result
    forward(x):
        ...
    # Computes gradient after forward() is called
    backward():
        # produced automatically
        ...
```

Computation Graphs in Pytorch

- Define forward pass for \( P(y|x) = \text{softmax}(Wg(Vf(x))) \)

```python
class FFNN(nn.Module):
    def __init__(self, input_size, hidden_size, out_size):
        super(FFNN, self).__init__()
        self.V = nn.Linear(input_size, hidden_size)
        self.g = nn.Tanh() # or nn.ReLU(), sigmoid()...
        self.W = nn.Linear(hidden_size, out_size)
        self.softmax = nn.Softmax(dim=0)

    def forward(self, x):
        return self.softmax(self.W(self.g(self.V(x))))
```

apply is syntactic sugar for forward

Input to Network

- Whatever you define with torch.nn needs its input as some sort of tensor, whether it’s integer word indices or real-valued vectors

```python
def form_input(x) -> torch.Tensor:
    # Index words/embed words/etc.
    return torch.from_numpy(x).float()
```

More on this later

Training and Optimization

\( P(y|x) = \text{softmax}(Wg(Vf(x))) \) one-hot vector of the label (e.g., \([0, 1, 0]\])

```python
ffnn = FFNN(inp, hid, out)
optimizer = optim.Adam(ffnn.parameters(), lr=lr)
for epoch in range(0, num_epochs):
    for (input, gold_label) in training_data:
        ffnn.zero_grad() # clear gradient variables
        probs = ffnn.forward(input)
        loss = torch.neg(torch.log(probs)).dot(gold_label)
        loss.backward()
        optimizer.step()
```
Optimizer in Pytorch

```python
optimizer = optim.SGD(network.parameters(), lr=0.01)
optimizer = optim.Adam(network.parameters(), lr=0.001)

- Learning rates for deep learning are often tiny! (0.01 or lower)
- Adam: adaptive method, incorporates momentum (gradient is smoothed with running average of past gradients). We will discuss a bit more but it's outside the scope of this class.
```

Initialization in Pytorch

```python
class FFNN(nn.Module):
    def __init__(self, inp, hid, out):
        super(FFNN, self).__init__()
        self.V = nn.Linear(inp, hid)
        self.g = nn.Tanh()
        self.W = nn.Linear(hid, out)
        self.softmax = nn.Softmax(dim=0)
        nn.init.uniform(self.V.weight)

- Initializing to a nonzero value is critical, more in a bit
```

Training a Model

```
Define a computation graph
Initialize weights and optimizer
For each epoch:
    For each batch of data:
        Zero out gradient
        Compute loss on batch
        Autograd to compute gradients and take step
Decode test set
```

Batching in Neural Networks
**Batching**

- Batching data gives speedups due to more efficient matrix operations
- Need to make the computation graph process a batch at the same time

```python
# input is [batch_size, num_feats]
# gold_label is [batch_size, num_classes]
def make_update(input, gold_label):
    ...  
    probs = ffnn.forward(input)  # [batch_size, num_classes]
    loss = torch.sum(torch.neg(torch.log(probs)).dot(gold_label))
    ... 
```

- Batch sizes from 1-100 often work well

**Nonconvex Optimization**

- For logistic regression, there is a global optimum: sum of log probabilities is a convex function in the weights
- Neural networks are much harder to optimize!

**Optimization Redux**

**How does initialization affect learning?**

\[ P(y|x) = \text{softmax}(W g(V f(x))) \]

- How do we initialize V and W? What consequences does this have?
- *Nonconvex* problem, so initialization matters!
How does initialization affect learning?

- Nonlinear model...how does this affect things?
  - If cell activations are too large in absolute value, gradients are small
  - ReLU: larger dynamic range (all positive numbers), but can produce big values, can break down if everything is too negative

Initialization

1) Can’t use zeroes for parameters to produce hidden layers: all values in that hidden layer are always 0 and have gradients of 0, never change

2) Initialize too large and cells are saturated

- Can do random uniform / normal initialization with appropriate scale
- Fancier initializers (Xavier Glorot initializer, Kaiming He) to match variances across layers

Optimizer

- Adam (Kingma and Ba, ICLR 2015) is very widely used
- Adaptive step size, incorporates momentum

- Wilson et al. NIPS 2017: adaptive methods can actually perform badly at test time (Adam is in pink, SGD in black)
- Check dev set periodically, decrease learning rate if not making progress
Dropout

- Probabilistically zero out parts of the network during training to prevent overfitting, use whole network at test time
- Form of stochastic regularization
- Similar to benefits of ensembling: network needs to be robust to missing signals, so it has redundancy
- Dropout layers exist in PyTorch

Srivastava et al. (2014)

Nonconvex Optimization

- For logistic regression, there is a global optimum: sum of log probabilities is a convex function in the weights
- Neural networks are hard to optimize
  
  Big Points
  
  - Basic recipe (take gradients + apply update) is still the same
  - Neural networks need to be initialized to nonzero values
  - Optimizer choice is very important; use Adam unless you know what you’re doing