Bells and whistles in neural net training

Tricks in training neural networks

There are various tricks that people use when traini
networks:

- Regularizat
- Dropout: Adj
- Optimization methods: Adjusting the l
- Initialization: Using particular forms o i

Regularization

Neural networks can be regularized in a similar way as linear models. Neural networks can also with **Frobenius norm**, which is a trivial extension to L2 norm for matrices. In fact, in ma[illegible] it is just referred to as L2 regularization.

$$\mathcal{L} = \sum_{i=1}^{N} \ell^{(i)} + \lambda_{z\to y} \|\boldsymbol{\Theta}^{(z\to y)} [illegible] {}^{x\to z)}\|_F^2$$

where $\|\boldsymbol{\Theta}\|_F^2 = \sum_{i,j} \theta_{i,j}^2$ is the squred **Frobenius norm**, which generalizes the L_2 norm to matrices. The bias parameters b are not regularized, as they do not contribute to the classifier to the inputs.

L2 regularization

- Compute the gra

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{i=1}^{N} \frac{\partial \ell^{(i)}}{\partial \theta}$$

- Update the we

$$\theta = \theta - \eta \left(\sum_{i=1}^{N} \right.$$

- "Weigh decay factor": λ is a tunable hyper parameter that pulls a weight back when it has become too big
- Question: Does it matter which layer θ is from when computing the regularization term?

L1 regularization

- L1 regularizatio[illegible]

$$\mathcal{L} = \sum_{i=1}^{N} \ell^{(i)} + \lambda_{z \to y} \|\mathbf{\Theta}^{(z \to y)} \text{[illegible]} {}^{(x \to z)}\|_1$$

- Compute the [illegible]

$$\frac{\text{[illegible]}}{\partial \theta} = \sum_{i=1} \frac{\text{[illegible]}}{\partial \theta} \text{[illegible]}$$

- update the weights

$$\theta = \theta - \eta \left(\sum_{i=1}^{N} \frac{\partial \ell^{(i)}}{\partial \theta} + \lambda \, sign(\theta) \right)$$

Comparison of L1 and L2

- In L1 regularization, the weights shrink by a constant amount toward 0. In L2 regularization, the weights shrink by an amount which is proportional to w.

- When a particular weight has a large absolute $|\theta|$, L1 regularizati
regularizati mall, L1
regularizati
regularization.

- The net result is that L1 regularization ten
the weight of the network in a relatively small number of high-importance connections, while the other weights are driven toward zero. So L1 regularization effectively does *feature selection*.

Dropout

- Randomly drops a certain percentage of the n over-relianc **feature co-adapta** en working toge oal is to avoid overfitting.

Dropout

- Dropout can be ac

$$\boldsymbol{z}^{(1)} = g^1(\boldsymbol{\Theta}^{(1)}\boldsymbol{x} + \boldsymbol{b}^1)$$

$$\boldsymbol{m}^1 \sim Bernoulli($$

$$\tilde{\boldsymbol{z}}^{(1)} = \boldsymbol{m}^1 \quad \boldsymbol{z}^{(1}$$

$$\boldsymbol{m}^2 \sim Bernou \quad 2$$

$$\tilde{\boldsymbol{z}}^{(2)} = \boldsymbol{m}^2 \odot$$

$$\boldsymbol{y} = \boldsymbol{\Theta}^{(3)}\tilde{\boldsymbol{z}}^{(2)}$$

where $\boldsymbol{m}^1$ and $\boldsymbol{m}^2$ are mask vectors. The values of the elements in these vectors are either 1 or 0, drawn from a Bernoulli distribution with parameter r (usually $r = 0.5$)

Optimization methods

- SGD with Momentum
- AdaGrad
- Root Mean Sq
- Adam

SGD with Momentum

- At each timestep t, compute $\nabla_\theta \mathcal{L}$, and then compute the momentum as follows:

$$\theta_j = \theta_j - \eta V_t$$

- The momentum term increases for dimensions whose gradient point in the same directions and reduces updates for dimensions whose gradient change directions.

AdaGrad

- Keep a running sum of the squared gradient V_{∇_θ}. When updating the weight of this *theta*, divide the gradient by the square root of this term

$$V_0 = 0$$

e.g., $\epsilon = 10^{-8}$

- The net effect is to slow down the update for weights with large gradient and accelerate the update for weights with small gradient

Root Mean Square Prop (RMSProp)

- A minor adjustment of AdaGrad. Instead of letting the sum of squared gradient continuously grow, we let t

$$V = 0$$

^{2}

$$\theta_j = \theta_j - \eta \frac{\quad}{\sqrt{V}}$$

e.g. $\beta \approx 0.9, \eta = 0.001, \epsilon = 10^{-8}$

Adaptive Moment Estimation (Adam)

- Weight update at t

$$V_0 = 0, \; S_0 = 0,$$

$$V_t = \beta_1 V_{t-1} + (1 - \beta_1) \qquad \text{m}$$

$$S \qquad \text{SProp}$$

$$V_t \qquad = \frac{}{\beta_1}$$

$$S_t^{corrected} = \frac{S_t}{\beta_2^t}$$

$$\theta_j = \theta_j - \eta \frac{V_t^{corrected}}{\sqrt{S_t^{corrected}} + \epsilon}$$

- Adam combines Momentum and RMSProp

Initialization

Xavier Initialization:

$$\Theta \qquad \cdots \qquad (l+1) \Big]$$

where $n^{(l)}$ is the number of input units to Θ (fan $n^{(l+1)}$ is the number of output units from Θ

Neural net in PyTorch

```python
from torch imp
class Net(nn.Mod
    """subclass from nn.Module is
    important to inspecting the parameters"""

    def __init__(self, in_dim=25                ch_size=1):
        super(Net, self).__init__
        self.in_dim = in_dim
        self.o
        self.                                im, self.out_dim)
        self.softmax = nn.Soft              )

    def forward(self, input_ma
        logit = self.linear(input_matrix)
        #return raw score, not normalized score
        return logit


    def xtropy_loss(self, input_matrix, target_label_vec):
        loss = nn.CrossEntropyLoss()
        logits = self.forward(input_matrix)
        return loss(logits, target_label_vec)
```

Use optimizers in Pytorch

```
import torch.opt
net = Net(input_dim, output_dim)
optimizer = optim.Adam(net.parameters(), lr=lrate)
for epoch in range(epochs):
    total_nll = 0
    for batch in batchize(train_d                    e)
        optimizer.zero_grad() #zer
        vecto                          _            ch,\
                              _             _  dex)
        feat_                                      ectorized)
        label_vec = map(item                        ctorized)
        feat_list = list(feat_
        label_list = list(lab  _
        x = torch.Tensor(feat_list)
        y = torch.LongTensor(label_list)
        loss = net.xtropy_loss(x,y)
        total_nll += loss
        loss.backward()
        optimizer.step()
    torch.save(net.state_dict(), net_path)
```