Bells and whistles in neural net training

Tricks in training neural networks

There are various tricks that people use when training neural networks:

- Regularization: Adjusting the gradient
- Dropout: Adjusting the hidden units
- Optimization methods: Adjusting the learning rate
- Initialization: Using particular forms of initialization

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 many cases it is just referred to as L2 regularization.

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

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 gradient of a loss with L2 regularization

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

- Update the weights

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

- "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 regularization loss

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

- Compute the gradient

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

- 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 value, $|\theta|$, L1 regularization shrinks the weight much less than L2 regularization does. By contrast, when $|\theta|$ is small, L1 regularization shrinks the weight much more than L2 regularization.
- The net result is that L1 regularization tends to concentrate 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 nodes to prevent over-reliance on a few features or hidden units, or **feature co-adaptation**, where some features are only useful when working together with a few other features. The ultimate goal is to avoid overfitting.

Dropout

- Dropout can be achieved using a mask:

$$
\begin{aligned}
\boldsymbol{z}^{(\mathbf{1})} &= g^1(\boldsymbol{\Theta}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{\mathbf{1}}) \\
\boldsymbol{m}^{\mathbf{1}} &\sim \mathit{Bernoulli}(r^1) \\
\tilde{\boldsymbol{z}}^{(\mathbf{1})} &= \boldsymbol{m}^{\mathbf{1}} \odot \boldsymbol{z}^{(\mathbf{1})} \\
\boldsymbol{z}^{(\mathbf{2})} &= g^2(\boldsymbol{\Theta}^{(2)}\tilde{\boldsymbol{z}}^{(\mathbf{1})} + \boldsymbol{b}^{\mathbf{2}}) \\
\boldsymbol{m}^{\mathbf{2}} &\sim \mathit{Bernoulli}(r^2) \\
\tilde{\boldsymbol{z}}^{(\mathbf{2})} &= \boldsymbol{m}^{\mathbf{2}} \odot \boldsymbol{z}^{(\mathbf{2})} \\
\boldsymbol{y} &= \boldsymbol{\Theta}^{(3)}\tilde{\boldsymbol{z}}^{(2)}
\end{aligned}
$$

where $\boldsymbol{m}^{\mathbf{1}}$ and $\boldsymbol{m}^{\mathbf{2}}$ 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 Square Prop (RMSProp)
- Adam

SGD with Momentum

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

$$V_0 = 0,\ \beta \approx 0.9$$

$$V_t = \beta V_{t-1} + (1-\beta)\nabla_\theta \mathcal{L}$$

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

- 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

$$V_0 = 0$$

$$V_t = V_{t-1} + \nabla_\theta \mathcal{L}^2$$

$$\theta_j = \theta_j - \eta \frac{\nabla_\theta \mathcal{L}}{\sqrt{V_t} + \epsilon}$$

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 the sum decay:

$$V_0 = 0$$

$$V_t = \beta V_{t-1} + (1 - \beta)\nabla_\theta \mathcal{L}^2$$

$$\theta_j = \theta_j - \eta \frac{\nabla_\theta \mathcal{L}}{\sqrt{V_t} + \epsilon}$$

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

Adaptive Moment Estimation (Adam)

- Weight update at time step t for Adam:

$$
\begin{aligned}
&V_0 = 0,\ S_0 = 0, \\
&V_t = \beta_1 V_{t-1} + (1-\beta_1)\nabla_\theta \mathcal{L} \quad \text{Momentum} \\
&S_t = \beta_2 S_{t-1} + (1-\beta_2)\nabla_\theta \mathcal{L}^2 \quad \text{RMSProp} \\
&V_t^{corrected} = \frac{V_t}{\beta_1} \\
&S_t^{corrected} = \frac{S_t}{\beta_2^t} \\
&\theta_j = \theta_j - \eta \frac{V_t^{corrected}}{\sqrt{S_t^{corrected}} + \epsilon}
\end{aligned}
$$

- Adam combines Momentum and RMSProp

Initialization

Xavier Initialization:

$$\Theta \sim \mathcal{U}\left[-\sqrt{\frac{6}{n^{(l)} + n^{(l+1)}}}, \sqrt{\frac{6}{n^{(l)} + n^{(l+1)}}}\right]$$

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

Neural net in PyTorch

```python
from torch import nn
class Net(nn.Module):
    """subclass from nn.Module is
    important to inspecting the parameters"""

    def __init__(self, in_dim=25, out_dim=3, batch_size=1):
        super(Net, self).__init__()
        self.in_dim = in_dim
        self.out_dim = out_dim
        self.linear = nn.Linear(self.in_dim, self.out_dim)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, input_matrix):
        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.optim as optim
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_data, batch_size):
        optimizer.zero_grad() #zero out the gradient.
        vectorized = vectorize_batch(batch,\
                feat_index, label_index)
        feat_vec = map(itemgetter(0), vectorized)
        label_vec = map(itemgetter(1), vectorized)
        feat_list = list(feat_vec)
        label_list = list(label_vec)
        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)
```