More on Training Deep Neural Networks

Oliver Schulte
School of Computing Science
Simon Fraser University
Introduction to Deep Learning
Overview

- Rectified Linear Units
- Regularization
  - Dropout
  - Norm Constraint
- Adapting Step Sizes
- Batch Normalization
- Babysitting the Training Process
Rectified Linear Units

A New Activation Function
Vanishing Gradients

- Activation functions that approximate step functions have small gradients outside their center.
- This is exacerbated by backpropagation across many layers: according to the chain rule, gradients are multiplied.

➢ Problem for deep learning, recurrent neural networks
Problems with Sigmoid Activation Function

- Dense: typically all units are active for any given input.
- Vanishing gradient: as number of layers increase, the error derivative for each goes to 0.

➢ Do not use sigmoid for hidden nodes
- Hyperbolic tangent is better than sigmoid
- Usually Rectified Linear activation is best
- See tensor playground demo
Rectified Linear Unit

- $f(x) = \max(0,x)$
- Gradient is trivial.

Co-Adaptation and Regularization
Local Minima and Local Gradients

- How does a neural network get stuck in a local minimum?
- Many reasons, but key phenomenon is that gradients are directions for single weights, not sets of weights.
  - Definition of gradient for \( w \): fix all other weights, consider depending of error function \( E \) on \( w \) in isolation.
- Example: XOR
  - Moving weight for single feature does not help, need to move weight for both.
Local Minima and Co-Adaptation

- Because weights are changed one at a time, a bad value for $w_1$ can lead to bad values for $w_2$.
  - See UBC tool demo neural.jar
- This is called co-adaptation.
- Related to overfitting.
- Toy Example:

<table>
<thead>
<tr>
<th></th>
<th>$w_1$</th>
<th>$w_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal Value</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Local Minimum</td>
<td>8</td>
<td>-6</td>
</tr>
</tbody>
</table>

A common symptom of co-adaptation and overfitting are excessively large weight magnitudes. How can we address this?
Regularization

- Add a term to loss function that penalizes large weights
  minimize $E(w) + \lambda ||w||$
- Need to set the trade-off parameter $\lambda$
- Very common in machine learning
- The norm constraint: Fix the length $||w||$ of each weight vector to be less than a constant.
Dropout

- Model Averaging comes to Neural Nets
- Averaging Models is a good idea
  - See boosting
Dropout in Neural Nets

1. Stochastic gradient descent: Cycle through each case.
2. For each case, randomly drop out some neurons.
   1. Independently with probability $p$, e.g. 0.5.
   2. Train only the weights for the remaining neurons.
3. After training, multiply the weights by $p$. Intuition:
   1. A kind of average over the neural net for each case.
   2. If $p = 0.5$, twice as many hidden units are present as during training.

Typically also use the norm constraint.
Dropout Picture

Step Size Adaptation

See normalization survey on canvas
Gradients of Gradients

• Remember the driving example: Want to slow down getting close to the goal.
  • Especially if gradients flip sign (left-right-left-right).
    ➢ Fast change in gradients $\rightarrow$ smaller steps.

• Formal Idea: Make step size the inverse of 2\textsuperscript{nd}-order derivative.

• Newton Raphson Update Rule:
  $$x := x - \eta f'(x)/f''(x)$$
Analyzing Gradients

- Newton—Raphson is good
- But for many (millions) of parameters, cannot feasibly get all the 2\textsuperscript{nd}-order gradients (the Hessian).
  - Nor can we invert the Hessian matrix.
- Instead use the trends in gradient sequence as estimates of curvature.
- Many developments of this basic idea.
  - Typically user specifies initial learning rate and method adapts as training proceeds
  - We’ll look at the ADAM method (Adaptive Moment Estimation).
**ADAM Intuitions**

- **Input:** sequence of observed gradients $g_1, \ldots, g_t$

1. **Divide learning rate by observed variance/standard deviation.**
   - Variance replaces curvature in Newton-Raphson

2. **Update with average of gradients seen so far (not current gradient)**
   - Intuitively, like the momentum of a moving object
Estimate Gradient Moments

- Input: sequence of observed gradients \( g_1, \ldots, g_t \)
- Output: estimate \textit{exponentially discounted} gradient average, (uncentered) standard deviation
- Decay factor \( \beta_1 \) for average. E.g. for \( t = 3 \), \( \beta_1 = 0.9 \)
  - \( m_3 = 0.9 \times 0.9 \times (1-0.9) \, g_1 + 0.9 \times (1-0.9) \, g_2 + (1-0.9) \, g_3 \)
  - Common idea for time series
- Incremental Running Average Update:
  \[ m_t = \beta_1 \, m_{t-1} + (1- \beta_1) \, g_t \]
- Similar for variance with another decay factor
  \[ v_t = \beta_2 \, v_{t-1} + (1- \beta_2) \, (g_t)^2 \]
Update Formula

- Bias correction:
  \[ m_t := m_t / (1 - \beta_1^t) \]
  \[ v_t := v_t / (1 - \beta_2^t) \]
- \[ w_t := w_{t-1} - \eta \frac{m_t}{(v_t^{1/2} + \epsilon)} \]

update by (estimated) average gradient  Divide update by standard deviation
Batch Normalization

Normalization Survey on Canvas
High-level Intuition

- From the point of view of hidden layers inside a deep network:
  output activation of previous layer = input “data”
- whatever properties we want in input data ➔ properties we want in output activations
- one nice input data property was normalization
  - means and variances on the same scale
  - e.g. all data dimensions on the same scale (see preprocessing section)
Benefits of Normalized Activations

- avoid saturation
- less dependence on initial weight values
- some regularization: large values scaled back
Normalization Algorithm

For minibatch $\mathbf{x}_1, \ldots, \mathbf{x}_m$ of data points

For each node $x^i$

1. Find the activation values $x^i_j, \ldots, x^i_m$ for each data point

2. Normalize the activation values:

   \[
   \text{mean}_i := \frac{1}{m} \sum_{j=0}^{m} x^i_j \\
   \text{var}_i := \frac{1}{m} \sum_{j=0}^{m} (x^i_j - \text{mean}_i)^2 \\
   \hat{x}_i := \frac{x_i - \text{mean}_i}{\sqrt{\text{var}_i + \varepsilon}}
   \]

3. Scale and shift:

   \[
   y_i := \gamma \hat{x}_i + \beta
   \]

   where $\gamma, \beta$ are learned during backpropagation
Conclusion

- Many tips and tricks to try, little theory or guarantees
- For output nodes:
  - Use sigmoid + cross-entropy for classification
  - Use linear + least-squares for regression
- For hidden nodes:
  - Don’t use sigmoid
  - Relu is good default
  - Can try hyperbolic tangent
- Adapting the step size is a good idea
- Drop out and batch normalization sometimes help
- Regularization is a good idea (more next time)