Neural Network Optimization 1
CS 519: Deep Learning, Winter 2017

Fuxin Li

With materials from Zsolt Kira
Backpropagation learning of a network

• The algorithm
  • 1. Compute a forward pass on the compute graph (DAG) from the input to all the outputs
  • 2. Backpropagate all the outputs back all the way to the input and collect all gradients $G$
  • 3. $W = W - \alpha G$ for all the weights in all layers
Modules (Layers)

• Each layer can be seen as a module

• Given input, return
  • Output $f_a(x)$
  • Network gradient $\frac{\partial f_a}{\partial x}$
  • Gradient of module parameters $\frac{\partial f_a}{\partial w_a}$

• During backprop, propagate/update
  • Backpropagated gradient $\frac{\partial E}{\partial f_a}$

$$
\frac{\partial E}{\partial w_k} \frac{\partial E}{\partial f_k} g(f_{k-1}(x)) = \frac{\partial E}{\partial f_{k+1}} \frac{\partial f_{k+1}}{\partial f_k} g(f_{k-1}(x))
$$

where $f_k(x) = w_k^T g(f_{k-1}(x))$, $f_0(x) = x$
The abundance of online layers

<table>
<thead>
<tr>
<th>Core Layers</th>
<th>Convolutional Layers</th>
<th>Advanced Activations Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer</td>
<td>Convolution1D</td>
<td>LeakyReLU</td>
</tr>
<tr>
<td>Methods</td>
<td>Convolution2D</td>
<td>PReLU</td>
</tr>
<tr>
<td>Masking</td>
<td>MaxPooling1D</td>
<td>ELU</td>
</tr>
<tr>
<td>TimeDistributedMerge</td>
<td>AveragePooling1D</td>
<td>ParametricSoftplus</td>
</tr>
<tr>
<td>Merge</td>
<td>MaxPooling2D</td>
<td>ThresholdedLinear</td>
</tr>
<tr>
<td>Dropout</td>
<td>AveragePooling2D</td>
<td>ThresholdedReLU</td>
</tr>
<tr>
<td>Activation</td>
<td>UpSampling1D</td>
<td></td>
</tr>
<tr>
<td>Reshape</td>
<td>UpSampling2D</td>
<td></td>
</tr>
<tr>
<td>Permute</td>
<td>ZeroPadding1D</td>
<td></td>
</tr>
<tr>
<td>Flatten</td>
<td>ZeroPadding2D</td>
<td></td>
</tr>
<tr>
<td>RepeatVector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dense</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ActivityRegularization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TimeDistributedDense</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoEncoder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaxoutDense</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lambda</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LambdaMerge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Siamese</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Highway</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Recurrent Layers</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Recurrent</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SimpleRNN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GRU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LSTM</td>
<td></td>
</tr>
</tbody>
</table>

**Normalization Layers**
- BatchNormalization

**Embedding Layers**
- Embedding

**Noise Layers**
- GaussianNoise
- GaussianDropout
Learning Rates

• Gradient descent is only guaranteed to converge with *small enough* learning rates
  • So that’s a sign you should decrease your learning rate if it explodes

• Example:
  • $C(w) = \frac{1}{2}w^2$
  • Learning rate of $\alpha = 0.5$
  • $\alpha = 1$
  • $\alpha = 2$
  • $\alpha = 3$
Weight decay regularization

• Instead of using a normal step, add a

\[ G = G + \lambda W \]

• This corresponds to:

\[
\min_W \frac{1}{N} \sum_{i=1}^{N} l(f(x_i; W), y_i) + \frac{1}{2} \lambda \|W\|^2
\]

• Early stopping as well!
  • Help generalization
Momentum

• Basic updating equation (with momentum):

\[
\begin{align*}
D_0 &= 0 \\
D_{t+1} &= \mu D_t - \alpha G_t \\
W_{t+1} &= W_t + D_{t+1}
\end{align*}
\]

• \( \mu = 0.6\sim0.9 \), a lot of “inertia” in optimization

• Check the previous example with a momentum of 0.5
Normalization

• Each component to 0 mean, 1 standard deviation

• For ease of L2 regularization + optimization convergence rates

0.1, 10
0.1, -10
101, 101
101, 99
1, 1
1, -1
Computing the energy function and gradient

• Usual ERM energy function

\[
\min_W E(f) = \sum_{i=1}^n L(f(x_i; W), y_i)
\]

\[
\nabla_W E = \sum_{i=1}^n \frac{\partial L(f(x_i; W), y_i)}{\partial W}
\]

• One problem:
  • Very slow to compute when \( n \) is large
  • One gradient step takes long time!
  • Approximate?
Stochastic Mini-batch Approximation

\[
\min_{W} E(f) = \sum_{i=1}^{n} L(f(x_i; W), y_i)
\]

\[
\nabla_{W} E = \sum_{i=1}^{n} \frac{\partial L(f(x_i; W), y_i)}{\partial W}
\]

\[
\nabla_{W} \tilde{E} \approx \sum_{i \in N_m} \frac{\partial L(f(x_i; \tilde{W}), y_i)}{\partial \tilde{W}} \quad N_m \subset \{1, \ldots, n\}
\]

- Ensure the expectation is the same
  \[
  \mathbb{E}(\nabla_{W} \tilde{E}) = \nabla_{W} E
  \]

- Uniformly sample every time
  - Sample how many? 1 (SGD) – 256 (Mini-batch SGD)
  - Common mini-batch size is 32-256
  - In practice: dependent on GPU memory size
In Practice

- Randomly re-arrange the input examples
- Use a fixed order on the input examples
- Define an *iteration* to be every time the gradient is computed
- An *epoch* to be every time that all the input examples is looped through once
A practical run of training a neural network

• Check:
  • Energy
  • Training error
  • Validation error
Data Augmentation

• Create artificial data to increase the size of the dataset
• Example: Elastic deformations on MNIST

Figure 2. Top left: Original image. Right and bottom: Pairs of displacement fields with various smoothing, and resulting images when displacement fields are applied to the original image.
Data Augmentation

Training Image

Enlarge the dataset!

Horizontal Flip

Training Images

256x256

224x224

224x224

224x224
Data Augmentation

• One of the easiest ways to prevent overfitting is to augment the dataset.
CIFAR-10 dataset

- 60,000 images in 10 classes
  - 50,000 training
  - 10,000 test
- Designed to mimic MNIST
- 32x32

Assignment (will post on Canvas with more explicity):
- Write your own backpropagation NN to test on CIFAR-10