EG+- Optimizer for Tensorflow

VS. SGD & Momentum

VS. Adversarial Samples
SGD based algorithms: SGD

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta)$$

$\theta$ - the parameters of the network (weights and biases)

$\eta$ (eta) - the learning rate, a.k.a. step size

$J$ - the cost function

$\nabla_{\theta}$ - gradient w.r.t. the cost $J$

Equations from: http://ruder.io/optimizing-gradient-descent/
**SGD based algorithms: Momentum**

\[ v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta) \]

\[ \theta = \theta - v_t \]

\( v_t \) - velocity at time step \( t \)

\( \gamma \) - gamma hyperparameter controlling the momentum (\( \gamma < 1 \), typically 0.9)

---

**Figure 1.1:** Momentum updates in yellow, SGD in red. SGD can oscillate back and forth in a valley, but momentum is drawn out of the oscillation pattern.

Equations from: http://ruder.io/optimizing-gradient-descent/
SGD based algorithms: Nesterov Accelerated GD

\[ v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \]
\[ \theta = \theta - v_t \]

**Figure 1.2:** Momentum update in red, Nesterov update in yellow. Nesterov updates take momentum from the previous step into account prior to deciding where to compute the gradient, then computes the gradient and takes a "corrected" gradient step, smoothing out the progression and still avoiding SGD's oscillation.
SGD based algorithms: Adagrad & Adadelta

\[
\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}
\]

\[
\Delta \theta_t = -\frac{RMS[\Delta \theta]_{t-1}}{RMS[g]_t} g_t
\]

\[
\theta_{t+1} = \theta_t + \Delta \theta_t
\]

Adagrad

Adadelta

Maintain a history of square gradients per parameter and adjust the learning rate per parameter.

Can be better for sparse data, example: word embeddings where less common words require larger steps.

FIGURE 1.4: Adagrad in red compared to Adadelta in yellow. Adadelta is initialized with the gradient sums = 1.0 at start to avoid pathological initialization issues.
SGD based algorithms: RMSProp

\[ E[g^2]_t = 0.9 E[g^2]_{t-1} + 0.1 g^2_t \]

\[ \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \]

Like Adadelta, uses an exponential decaying average of the square gradient to set per-parameter learning rates

*Figure 1.7: In red, Adadelta with a gradient sum initialization of 1’s (e.g. a “good” initialization), and RMS Prop in yellow. Both perform very similarly on this 2 dimensional example, and long term convergence is quite similar. Both converge on the x dimension in 6 steps, though after 100 iterations RMS Prop had moved down to -100 in the y direction, vs. <500 for Adagrad, so Adagrad took larger steps down the chasm.*
SGD based algorithms: Adam

\[ m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \]
\[ v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \]

\( m_t \) - estimate of the first moment (mean)
\( v_t \) - estimate of the second moment (uncentered variance)
\( m, v \) are initialized to 0, thus bias towards 0
\( \beta_1, \beta_2 \) - decay rate hyperparameters typically 0.9/0.999

Bias corrected first and second moments

\[ \hat{m}_t = \frac{m_t}{1 - \beta_1^t} \]
\[ \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \]

\[ \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t \]

Update step

**Figure 1.8:** RMS Prop shown in red with Adam updates shown in yellow.
Exponentiated Gradient Algorithm

Normalization Methods:

- **Per weight pair** - sum of positive and negative weights
- **Per neuron** - sum across all inputs to a neuron
- **Per layer** - sum across all weights in a layer (by typical convention)
- **Per graph** - sum across all weights in the full model

\[
\begin{align*}
    w_{t+1,i}^+ &= U \cdot \frac{w_{t,i}^+ r_{t,i}^+}{\sum_{j=1}^N w_{t,j}^+ r_{t,j}^+ + w_{t,j}^- r_{t,j}^-} \\
    w_{t+1,i}^- &= U \cdot \frac{w_{t,i}^- r_{t,i}^-}{\sum_{j=1}^N w_{t,j}^+ r_{t,j}^+ + w_{t,j}^- r_{t,j}^-} \\
    r_{t,i}^+ &= \exp \left( -\eta \nabla_{w_{t,i}^\pm} L_{y_t}(\hat{y}_t) \right) \\
    r_{t,i}^- &= \exp \left( \eta \nabla_{w_{t,i}^\pm} L_{y_t}(\hat{y}_t) \right) = \frac{1}{r_{t,i}^+}
\end{align*}
\]
import tensorflow as tf
from tensorflow.python.ops import control_flow_ops, math_ops, state_ops, import_ops, tensor_shape, optimizer

# Optimizer class
class EGPlusMinusOptimizer(optimizer.Optimizer):
    def __init__():
        # Basic initialization

    def _prepare(self):
        # Run at session creation

    def _create_slots(self, var_list):
        # A place to create needed variables pos / neg values.
        for v in var_list:
            self._get_or_make_slot_with_initializer(...)
            self._get_or_make_slot_with_initializer(...)

    def _apply_dense(self, grad, var)
        # Run optimizer - main function to implement

    def _apply_sparse(self, grad, var):
        # Run optimizer for sparse vectors
Experiment setup

Model: 32 Layer Residual Network

Dataset: CIFAR10
10 Classes, [32 x 32] images

Hyperparameter search across optimizer hyperparameters

Final models trained for 30+ hours

Training set: 50,000 images
Test set: 10,000 images

https://www.cs.toronto.edu/~kriz/cifar.html
**EG +/- vs SGD vs Momentum**

<table>
<thead>
<tr>
<th>Method</th>
<th>Best accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best accuracy 0.9 smoothing</td>
<td></td>
</tr>
<tr>
<td>EG per weight pairs</td>
<td>0.91</td>
</tr>
<tr>
<td>EG per layer (per variable)</td>
<td>0.92</td>
</tr>
<tr>
<td>EG per neuron weights +/-</td>
<td>0.75</td>
</tr>
<tr>
<td>SGD</td>
<td>0.89</td>
</tr>
<tr>
<td>Momentum</td>
<td>0.89</td>
</tr>
</tbody>
</table>
## Adversarial Examples

<table>
<thead>
<tr>
<th></th>
<th>Accuracy on 1000 original image</th>
<th>Accuracy on 1000 adversarial images</th>
</tr>
</thead>
<tbody>
<tr>
<td>EG per weight pairs</td>
<td>0.917</td>
<td>0.351</td>
</tr>
<tr>
<td>EG per layer (per variable)</td>
<td>0.931</td>
<td>0.392</td>
</tr>
<tr>
<td>EG per neuron weights +/-</td>
<td>0.718</td>
<td>0.710</td>
</tr>
<tr>
<td>SGD</td>
<td>0.859</td>
<td>0.399</td>
</tr>
<tr>
<td>Momentum</td>
<td>0.873</td>
<td>0.441</td>
</tr>
</tbody>
</table>