Topic: Optimization for Neural Networks

- Background:
  1. Neural Networks (NN) are a broad class of predictive models that learns the relationship between the features and the outcomes.
  2. Training a NN involves updating the parameters describing the connections in order to minimize some loss function over the training data.
  3. NN have been applied to broad range of fields and reached the state-of-the-art performance in many tasks.
  4. NN require large datasets and sufficient computing power.
  5. Optimizing NN means reducing overfitting, having a lower tendency to get stuck in local saddle points, or converging in a smaller number of epochs.
- Goal:
  1. Code Deep Neural Network from scratch. Apply regularization and momentum techniques. Main Resource: *A Computational Approach to Statistical Learning*
  2. Implement two variants of SGD as the optimizers for the NN and experiment with simulated data. Compare their performances.

Review of the Basics

Architecture

- Layers ($l \in 0 : L$)
  - Input Layer x
  - Output Layer $\hat{y}$
  - Hidden Layer(s)
- Nodes (Neurons) a^l
  - Weights w^l
  - Biases b^l
- Activation Function σ
  - ReLU
  - Softmax (before output layer)
- Relationship (Forward propagation)
  - $a^0 = x$
  - $z^l = w^l a^{l-1} + b^l$
  - $a^l = \sigma(z^l)$

Backward Propopation

- Goal: Adjusting the weights to reach the minimized loss function
- Gradient descent updates:
  $w^{l+1} = w^l - \eta \nabla_w f(w)$
  $b^{l+1} = b^l - \eta \nabla_b f(b)$
- NNs generally need many thousands of iterations to converge to a reasonable minimizer of the loss function. With large datasets and models, while still feasible, gradient descent can become quite slow. To avoid working with the entire dataset at each step, we use SGD, which incrementally updates the weights in the model.

Backward Propagation

- Notation: For each $l \in (0, L)$, $w^l_{j,k}$ denotes the weights on the kth neuron in the $(l-1)$-st layer within the jth neuron in the lst layer; b^l_j denotes the bias term for the jth neuron in the lst layer
- SGD updates:
  $w^{l+1} = w^l - (\eta/n)\nabla_{w^l} f(w^l)$
  $b^{l+1} = b^l - (\eta/n)\nabla_{b^l} f(b^l)$
  - Loss: $f = (a^L - y)^2$ (regression)
    $f = -\sum_k y_k \cdot log(a^L_k)$ (classification)
  - $\partial_{b^l_j} f = \frac{\partial f}{\partial z^l_j}\frac{\partial z^l_j}{\partial b^l_j} = \frac{\partial f}{\partial z^l_j}$
  - $\partial_{w^l_{j,k}} f = \frac{\partial f}{\partial z^l_j}\frac{\partial z^l_j}{\partial w^l_{j,k}} = \frac{\partial f}{\partial z^l_j} a^{l-1}_k$

- $z^{l+1}_k = \sum_m w^{l+1}_{k,m}\sigma(z^l_m) + b^{l+1}_k$
- $\frac{\partial z^{l+1}_k}{\partial z^l_j} = w^{l+1}_{k,j}\sigma'(z^l_j)$
- $\frac{\partial f}{\partial z^l_j} = \sum_k \frac{\partial f}{\partial z^{l+1}_k}\frac{\partial z^{l+1}_k}{\partial z^l_j} = \sum_k \frac{\partial f}{\partial z^{l+1}_k} w^{l+1}_{k,j}\sigma'(z^l_j)$
- $\Longrightarrow \frac{\partial f}{\partial z^l} = \left[(w^{l+1})^T\left(\frac{\partial f}{\partial z^{l+1}_j}\right)\right] \odot \sigma'(z^l)$
- (Regression) $\frac{\partial f}{\partial z^L_j} = \sum_k \frac{\partial f}{\partial a^L_k}\frac{\partial a^L_k}{\partial z^L_j} = \frac{\partial f}{\partial a^L_j}\frac{\partial a^L_j}{\partial z^L_j} = \frac{\partial f}{\partial a^L_j}\sigma'(z^L_j)$
- (Classification) $\frac{\partial f}{\partial z^L_j} = a^L_j - y_i$

Regularization

- Problem: Overfitting
- Solution: Includes a penalty term directly into the loss function. For example, adding an l_2 penality yields

$$f_\lambda(w, b) = f(w, b) + \frac{\lambda}{2}||w||_2^2$$
$$\nabla_w f_\lambda = \nabla_w f(w, b) + \lambda \cdot w$$

- The updating rule becomes

$$\begin{aligned} w^{l+1} &= w^l - \eta \nabla_w f(w^{l-1}) \\ &= w^l - \eta[\nabla_w f(w^{l-1}) + \lambda w^{l-1}] \\ &= [1 - \eta\lambda] w^{l-1} - \eta \nabla_w f(w^{l-1}) \end{aligned}$$

SGD with Momentum

- Problem: Relying only on the gradient leads to models getting stuck at saddle points and being very sensitive to the learning rate.
- Solution: Incorporating the momentum vector v into the algorithm
- Advantages:
  1. if the gradient remains relatively unchanged over several steps it will 'pick-up speed' (momentum) and effectively use a larger learning rate;
  2. if the gradient is changing rapidly, the step-sizes will shrink accordingly;
  3. when passing through a saddle point, the built up momentum from prior steps helps propel the algorithm past the point.
- Updating Rule:

$$v^l = v^{l-1} \cdot \mu - \eta \cdot \nabla_w f(w^{l-1})$$
$$w^l = (1 - \eta\lambda) w^{l-1} + v^l$$

where $\mu \in [0, 1]$

Stochastic Average Gradient (SAG)

- Roux et al. 2012
- SAG incorporates a memory of previous gradient values in order to achieve a linear convergence rate.
- Updating Rule

$$w^{l+1} = w^l - \frac{\eta}{n} \sum_{i=1}^{n} y_i^l$$

where at each iteration a random training example i_l is selected and we set

$$y_i^l = \begin{cases} f_i'(w^l) & \text{if } i = i_l \\ y_i^{l-1} & \text{otherwise} \end{cases}$$

Stochastic Variance Reduced Gradient (SVRG)

- Johnson et al. 2013
- SVRG was developed to reduce the variance of the gradients
- Start with $\tilde{w}$ which is close to the optimal w, and calculate the average gradient by one pass over the data using using $\tilde{w}$

$$\tilde{\mu} = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\tilde{w})$$

- Updating Rule:
  Draw i_t randomly from $\{1, ..., n\}$

$$w^{(l+1)} = w^{(l)} - \eta[\nabla f_i(w^{(l)}) - \nabla f_{i_t}(\tilde{w}) + \tilde{\mu}]$$

  Update $\tilde{w}$ and $\tilde{\mu}$ after m SGD iterations.

SAGA

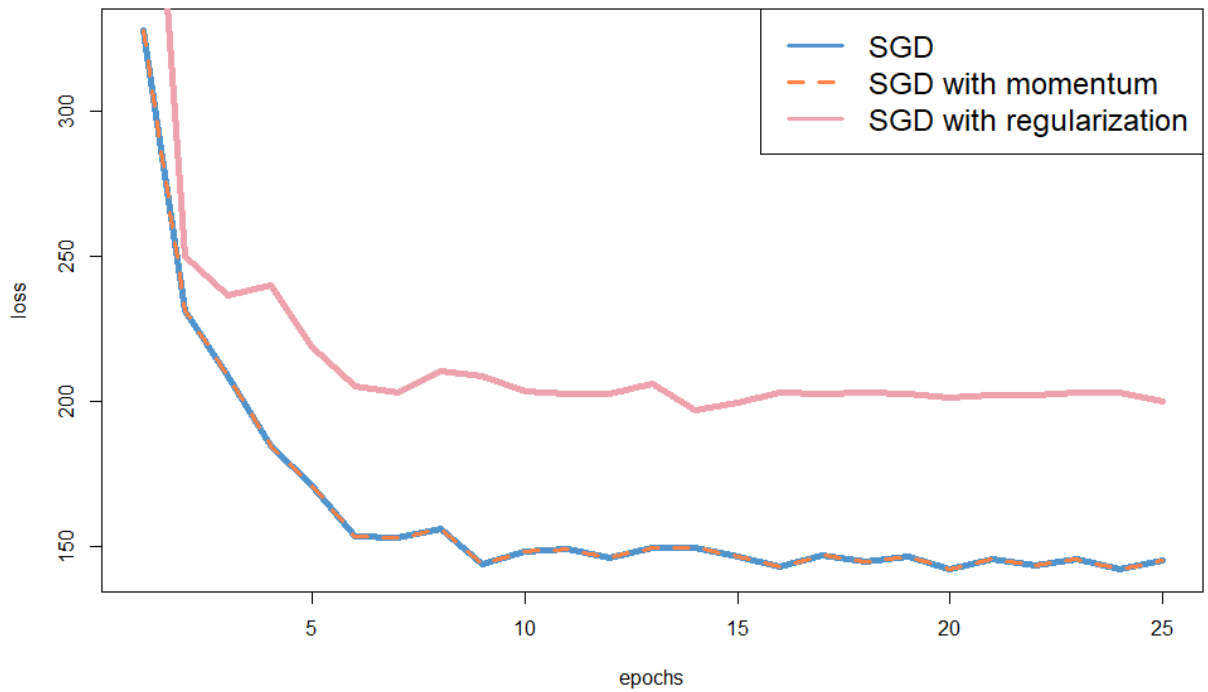
- Defazio et al. 2014
- SAGA is inspired by SVRG. Instead of approximating the parameters w, SAGA stores the gradient vectors for each sample.

Choose an initial iterate $w_1 \in \mathbb{R}^d$ and stepsize $\alpha > 0$.
for $i = 1, \ldots, n$ **do**
  Compute $\nabla f_i(w_1)$.
  Store $\nabla f_i(w_{[i]}) \leftarrow \nabla f_i(w_1)$.
for $k = 1, 2, \ldots$ **do**
  Choose j uniformly in $\{1, \ldots, n\}$.
  Compute $\nabla f_j(w_k)$.
  Set $g_k \leftarrow \nabla f_j(w_k) - \nabla f_j(w_{[j]}) + \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(w_{[i]})$.
  Store $\nabla f_j(w_{[j]}) \leftarrow \nabla f_j(w_k)$.
  Set $w_{k+1} \leftarrow w_k - \alpha g_k$.

Simulation Studies

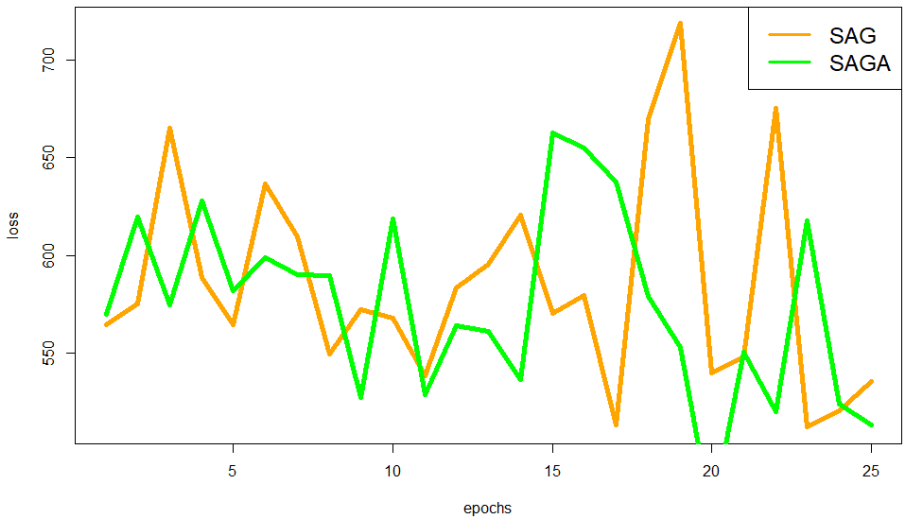
- Simple case:
  - Data: $X \sim 1000 \times 1$, where $x_i \sim \text{Unif}\ (-1, 1)$
    $y_i = x_i^2 + \epsilon$ where $\epsilon \sim N(0, 0.1)$
    $y_i = I_{(y_i > 0.5)}$
  - Total epochs: 25
    Learning rate: 0.01

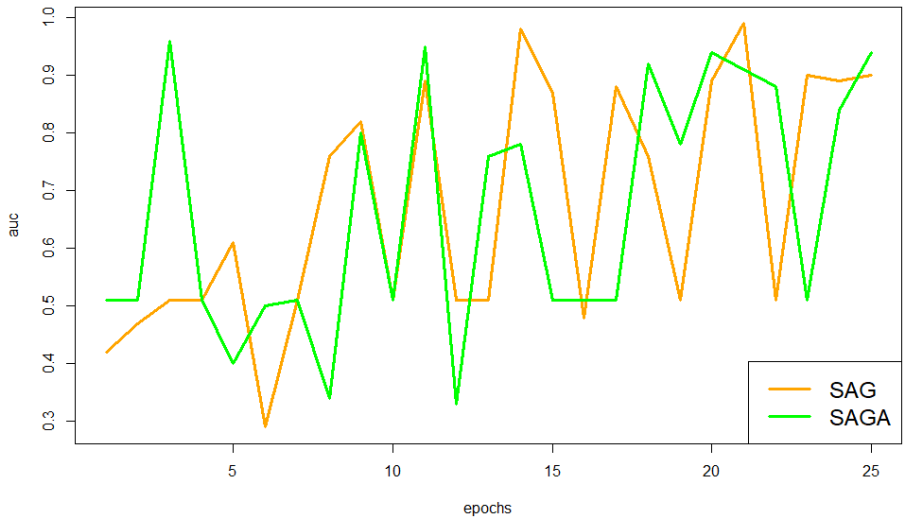
SGD, SGD with momentum (0.09), and SGD with regularization (0.01)



AUC: 0.98 $\sim$ 0.99 with 0.5 as the discriminating threshold

SAG vs SAGA


SAG
SAGA
loss
epochs
550
600
650
700
5
10
15
20
25


SAG
SAGA
auc
epochs
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
5
10
15
20
25

2021 Phenology Data (Blooming)

- Real data from *rnpn* package
- training data 500×5

	latitude	longitude	elevation_in_meters	day_of_year	phenophase_status
1	36.44084	-84.32359	384	1	0
2	36.44084	-84.32359	384	1	0
3	36.44084	-84.32359	384	1	0
4	36.44084	-84.32359	384	1	0
5	36.44084	-84.32359	384	1	0
6	36.44084	-84.32359	384	1	0
7	32.39724	-111.00216	801	1	0
8	32.39724	-111.00216	801	1	0
9	32.39724	-111.00216	801	1	0
10	32.39724	-111.00216	801	1	0
11	32.37423	-110.96838	800	1	0
12	32.37423	-110.96838	800	1	0
13	32.37423	-110.96838	800	1	0
14	32.37423	-110.96838	800	1	0

Training NN only using 1 feature *(day of year)*

	epoch	Hidden Layers (25 nodes)	loss (median)
SGD	50	10	538.87
SAGA	50	10	453.21
SAG	50	10	893.72

Training NN using all 4 features

	epoch	Hidden Layers (25 nodes)	loss (median)
SGD	50	10	1336.4
SGD *	50	10	940.38
SAGA	50	10	782.92
SAGA*	50	10	658.24
SAG	50	10	1557
SAG*	50	10	1763.28

Conclusions

- Summary
  - For the simple case where X is uniform and Y is X^2 plus a normal noise, all types of NNs are working great, showing promising AUC values. While for the real data, where the relationship between features and responses is unclear, SAGA displays better performances. SAG is performing the worst. Also by updating the gradients selectively, SAG ans SAGA require less cpu time than SGD.
  - If we exclude the cases of human error, we may reach the conclusion that SAG and SAGA have the potential of complex machine learning tasks where tens of thousand feature may exist.
- Limits
  - learning rate