New Optimization Algorithms for Neural Network Training Using Operator Splitting Techniques

Ciristian Daniel Alecsa, Imre Boroș, Titus Pința

Zilele Academice Clujene 2019

Tiberiu Popoviciu Institute of Numerical Analysis
Romanian Academy
ictp.acad.ro

TIBERIU POPOVICIU
INSTITUTE OF NUMERICAL ANALYSIS
ROMANIAN ACADEMY

Contents

Neural Networks

Inspiration for neural networks came from the way the human brain works.

Neural Networks

Inspiration for neural networks came from the way the human brain works.

A neuron has as it's input the activation values of the previous neurons, each having a weight in the current neuron's activation

$$a_i^l = f\left(\sum_{k=0}^{n} w_{ik}^{l-1} a_k^{l-1} + b_i^l\right)$$

Neural Networks

Inspiration for neural networks came from the way the human brain works.

A neuron has as it's input the activation values of the previous neurons, each having a weight in the current neuron's activation

$$a_i^l = f\left(\sum_{k=0}^{n} w_{ik}^{l-1} a_k^{l-1} + b_i^l\right)$$

In matrix form the activation of a layer is

$$a^l = f(w^{l-1} a^{l-1} + b^l)$$

Neural Networks

Inspiration for neural networks came from the way the human brain works.

A neuron has as it's input the activation values of the previous neurons, each having a weight in the current neuron's activation

$$a_i^l = f\left(\sum_{k=0}^{n} w_{ik}^{l-1} a_k^{l-1} + b_i^l\right)$$

In matrix form the activation of a layer is

$$a^l = f(w^{l-1} a^{l-1} + b^l)$$

Combining multiple layers we can construct a function from $\mathbb{R}^n$ to $\mathbb{R}^m$

It is proven in[3] that a single layer neural network with suitable weights and biases can approximate any continuous function with an error less than ϵ.

Neural Networks

Similar with a phenomenon in electrical engineering, lowering the error bound ϵ requires exponential increase in the number of neurons.

Neural Networks

Similar with a phenomenon in electrical engineering, lowering the error bound ϵ requires exponential increase in the number of neurons.

Fortunately if we increase the depth of the network we wold require only a logarithmic growth in the number of neurons.

Neural Networks

Similar with a phenomenon in electrical engineering, lowering the error bound ϵ requires exponential increase in the number of neurons.

Fortunately if we increase the depth of the network we wold require only a logarithmic growth in the number of neurons.

In order to compute the weights and biases that would yield an interpolation to a function defined on a dataset x^i, y^iwe could minimize a cost function that depends on the weights and biases of our network

$$c(x, y) = \|f(x) - y\|$$

Backpropagation

We can use gradient based methods of optimization in order to minimize the cost function

Backpropagation

We can use gradient based methods of optimization in order to minimize the cost function

The gradient for the weights and biases of the last layer is easily computed with the chain rule

$$\frac{\partial c}{\partial w_{ij}^n} = \frac{\partial c}{\partial f}\frac{\partial f}{\partial x}a_j^n$$

Backpropagation

We can use gradient based methods of optimization in order to minimize the cost function

The gradient for the weights and biases of the last layer is easily computed with the chain rule

$$\frac{\partial c}{\partial w_{ij}^n} = \frac{\partial c}{\partial f}\frac{\partial f}{\partial x}a_j^n$$

Applying the chain rule again we can arrive to the formula for the derivative of the cost function with respect to the weights at each layer

$$\frac{\partial c}{\partial w_{ij}^l} = \frac{\partial a^l}{\partial f}\frac{\partial f}{\partial x}a_j^l$$

Backpropagation

We can use gradient based methods of optimization in order to minimize the cost function

The gradient for the weights and biases of the last layer is easily computed with the chain rule

$$\frac{\partial c}{\partial w_{ij}^n} = \frac{\partial c}{\partial f}\frac{\partial f}{\partial x}a_j^n$$

Applying the chain rule again we can arrive to the formula for the derivative of the cost function with respect to the weights at each layer

$$\frac{\partial c}{\partial w_{ij}^l} = \frac{\partial a^l}{\partial f}\frac{\partial f}{\partial x}a_j^l$$

A similar technique gives the derivative with respect to the biases

Gradient Descent

Consider a convex C^2 function E with the global minimum in x_0

Gradient Descent

Consider a convex C^2 function E with the global minimum in x_0

A simple application of Lyapunov's theorem proves that dynamic system

$$\dot{x} = -\nabla E(x)$$

has a unique global attractor in x_0.

Gradient Descent

Consider a convex C^2 function E with the global minimum in x_0

A simple application of Lyapunov's theorem proves that dynamic system

$$\dot{x} = -\nabla E(x)$$

has a unique global attractor in x_0.

Applying a forward Euler ODE solver we obtain the update rule

$$x_n = x_{n-1} - h\nabla E(x_{n-1})$$

Gradient Descent

Consider a convex C^2 function E with the global minimum in x_0

A simple application of Lyapunov's theorem proves that dynamic system

$$\dot{x} = -\nabla E(x)$$

has a unique global attractor in x_0.

Applying a forward Euler ODE solver we obtain the update rule

$$x_n = x_{n-1} - h\nabla E(x_{n-1})$$

The maximum step size that guarantees convergence for the discrete system is L^{-1}, where L is the Lipschitz constant of the gradient of E.

Gradient Descent

Consider a convex C^2 function E with the global minimum in x_0

A simple application of Lyapunov's theorem proves that dynamic system

$$\dot{x} = -\nabla E(x)$$

has a unique global attractor in x_0.

Applying a forward Euler ODE solver we obtain the update rule

$$x_n = x_{n-1} - h\nabla E(x_{n-1})$$

The maximum step size that guarantees convergence for the discrete system is L^{-1}, where L is the Lipschitz constant of the gradient of E.

The convergence rate of the algorithm is $\mathcal{O}(n^{-1})$

Stochastic Gradient Descent

Consider now a function E defined as

$$E = E_0 + E_1 + E_2 + \ldots + E_k$$

Stochastic Gradient Descent

Consider now a function E defined as

$$E = E_0 + E_1 + E_2 + \ldots + E_k$$

In this situation we can approximate the gradient of E with the gradient of E_i, yielding the following update rule

$$x_n = x_{n-1} - h\nabla E_i(x_{n-1})$$

Stochastic Gradient Descent

Consider now a function E defined as

$$E = E_0 + E_1 + E_2 + \ldots + E_k$$

In this situation we can approximate the gradient of E with the gradient of E_i, yielding the following update rule

$$x_n = x_{n-1} - h\nabla E_i(x_{n-1})$$

Computing the update for all components E_i we iterate one epoch in the Stochastic Gradient Descent Algorithm.

Stochastic Gradient Descent

Consider now a function E defined as

$$E = E_0 + E_1 + E_2 + \ldots + E_k$$

In this situation we can approximate the gradient of E with the gradient of E_i, yielding the following update rule

$$x_n = x_{n-1} - h\nabla E_i(x_{n-1})$$

Computing the update for all components E_i we iterate one epoch in the Stochastic Gradient Descent Algorithm.

We do this because computing the entire gradient for the dataset can be expensive.

Stochastic Gradient Descent

Consider now a function E defined as

$$E = E_0 + E_1 + E_2 + \ldots + E_k$$

In this situation we can approximate the gradient of E with the gradient of E_i, yielding the following update rule

$$x_n = x_{n-1} - h\nabla E_i(x_{n-1})$$

Computing the update for all components E_i we iterate one epoch in the Stochastic Gradient Descent Algorithm.

We do this because computing the entire gradient for the dataset can be expensive.

If the expected value of the gradient components is equal to the true gradient we get a convergence rate of $\mathcal{O}(n^{-0.5})$

Stochastic Gradient Descent

Consider now a function E defined as

$$E = E_0 + E_1 + E_2 + \ldots + E_k$$

In this situation we can approximate the gradient of E with the gradient of E_i, yielding the following update rule

$$x_n = x_{n-1} - h\nabla E_i(x_{n-1})$$

Computing the update for all components E_i we iterate one epoch in the Stochastic Gradient Descent Algorithm.

We do this because computing the entire gradient for the dataset can be expensive.

If the expected value of the gradient components is equal to the true gradient we get a convergence rate of $\mathcal{O}(n^{-0.5})$ A thorough exposition of different Stochastic Gradient Descent algorithms is available in[4]

The Heavy Ball with Friction Model

If we consider a dynamic system inspired by a ball with mass that rolls with friction on a surface under the force of gravity we arrive at the ODE

$$\ddot{x} + \gamma \dot{x} = -\nabla E(x)$$

The Heavy Ball with Friction Model

If we consider a dynamic system inspired by a ball with mass that rolls with friction on a surface under the force of gravity we arrive at the ODE

$$\ddot{x} + \gamma \dot{x} = -\nabla E(x)$$

With a constant γ a finite difference scheme gives Polyak's algorithm

$$y_n = y_{n-1} + h\nabla E(x_{n-1})$$
$$x_n = x_n - \eta y_n$$

For $\gamma = 3t^{-1}$ the discrete dynamic system associated to the ODE yields Nesterov's Accelerated Gradient[6]

$$y_n = y_{n-1} + h\nabla E(y_{n-1})$$
$$x_n = x_n - \eta y_n$$

Adaptive Learning Rate Algorithms

The problem with momentum based algorithms comes when x_n is close to a minimum of the function.

Adaptive Learning Rate Algorithms

The problem with momentum based algorithms comes when x_n is close to a minimum of the function.

Because of the inertia of the ball it will overshoot the desired point, oscillating around until the kinetic energy is dissipated via friction.

Adaptive Learning Rate Algorithms

The problem with momentum based algorithms comes when x_n is close to a minimum of the function.

Because of the inertia of the ball it will overshoot the desired point, oscillating around until the kinetic energy is dissipated via friction.

In order to avoid this problem we need to diminish the learning rate when we get closer to the minimum.

Adaptive Learning Rate Algorithms

The problem with momentum based algorithms comes when x_n is close to a minimum of the function.

Because of the inertia of the ball it will overshoot the desired point, oscillating around until the kinetic energy is dissipated via friction.

In order to avoid this problem we need to diminish the learning rate when we get closer to the minimum.

The RMSprop algorithm divides the learning rate with the square root of the cumulated sum of the squares of the gradients.

$$\mathbb{E}(g)_n = \rho \mathbb{E}(g)_{n-1} + (1 - \rho) g_n$$

Adaptive Learning Rate Algorithms

The problem with momentum based algorithms comes when x_n is close to a minimum of the function.

Because of the inertia of the ball it will overshoot the desired point, oscillating around until the kinetic energy is dissipated via friction.

In order to avoid this problem we need to diminish the learning rate when we get closer to the minimum.

The RMSprop algorithm divides the learning rate with the square root of the cumulated sum of the squares of the gradients.

$$\mathbb{E}(g)_n = \rho\mathbb{E}(g)_{n-1} + (1-\rho)g_n$$

The Adadelta algorithm also multiplies the learning rate with the square root of the cumulated sum of the squares of the coordinate updates Δx_n.

Adaptive Learning Rate Algorithms

The problem with momentum based algorithms comes when x_n is close to a minimum of the function.

Because of the inertia of the ball it will overshoot the desired point, oscillating around until the kinetic energy is dissipated via friction.

In order to avoid this problem we need to diminish the learning rate when we get closer to the minimum.

The RMSprop algorithm divides the learning rate with the square root of the cumulated sum of the squares of the gradients.

$$\mathbb{E}(g)_n = \rho\mathbb{E}(g)_{n-1} + (1-\rho)g_n$$

The Adadelta algorithm also multiplies the learning rate with the square root of the cumulated sum of the squares of the coordinate updates Δx_n. An analysis of the continuous dynamic systems for adaptive algorithms is presented in[1]

Operator Splitting

Operator Splitting is a technique that approximates the solution of a linear ODE

$$\dot{x} = (A + B)x$$

Normally

$$e^{(At+Bt)} = e^{At}e^{Bt}$$

only if the matrices A and B commute.

Operator Splitting

Operator Splitting is a technique that approximates the solution of a linear ODE

$$\dot{x} = (A + B)x$$

Normally

$$e^{(At+Bt)} = e^{At}e^{Bt}$$

only if the matrices A and B commute.

The Baker Campbell Hausdorff formula gives the following bound

$$\|e^{(At+Bt)} - e^{At}e^{Bt}\| \in \mathcal{O}(t)$$

Operator Splitting

Operator Splitting is a technique that approximates the solution of a linear ODE

$$\dot{x} = (A + B)x$$

Normally

$$e^{(At+Bt)} = e^{At}e^{Bt}$$

only if the matrices A and B commute.

The Baker Campbell Hausdorff formula gives the following bound

$$\|e^{(At+Bt)} - e^{At}e^{Bt}\| \in \mathcal{O}(t)$$

A similar bound was proven in[5] for

$$\|e^{(At+Bt)} - e^{Bt/2}e^{At}e^{Bt/2}\| \in \mathcal{O}(t^2)$$

Sequential Splitting Algorithms

We start from the dynamic system with constant dampening

$$\ddot{x} + \dot{x} = -\nabla E(x)$$

Sequential Splitting Algorithms

We start from the dynamic system with constant dampening

$$\ddot{x} + \dot{x} = -\nabla E(x)$$

We split the problem in a linear part

$$\begin{cases} \dot{u}(t) = 0 \\ \dot{v}(t) = -v(t) \end{cases}$$

Sequential Splitting Algorithms

We start from the dynamic system with constant dampening

$$\ddot{x} + \dot{x} = -\nabla E(x)$$

We split the problem in a linear part

$$\begin{cases} \dot{u}(t) = 0 \\ \dot{v}(t) = -v(t) \end{cases}$$

and a nonlinear part

$$\begin{cases} \dot{u}(t) = v(t) \\ \dot{v}(t) = -\nabla f(u(t)) \end{cases}$$

Sequential Splitting Algorithms

We start from the dynamic system with constant dampening

$$\ddot{x} + \dot{x} = -\nabla E(x)$$

We split the problem in a linear part

$$\begin{cases} \dot{u}(t) = 0 \\ \dot{v}(t) = -v(t) \end{cases}$$

and a nonlinear part

$$\begin{cases} \dot{u}(t) = v(t) \\ \dot{v}(t) = -\nabla f(u(t)) \end{cases}$$

We use a forward Euler method in order to obtain a discrete version of each problem.

Sequential Splitting Algorithms

Combining the 2 sub-problems, denoting $n/(n+3)$ with β_n and introducing a hyper-parameter k in order to better control the velocity we arrived at the final algorithm SSA1

$$\begin{cases} y_n = u_n + h\beta_n v_n \\ v_{n+1} = \beta_n^k \cdot ((1 - h\beta_n)v_n - h\nabla f(y_n)) \\ u_{n+1} = u_n + \beta_n(1 - h\beta_n)(y_n - u_n) - h^2\nabla f(y_n) \end{cases}$$

Sequential Splitting Algorithms

Combining the 2 sub-problems, denoting $n/(n+3)$ with β_n and introducing a hyper-parameter k in order to better control the velocity we arrived at the final algorithm SSA1

$$\begin{cases} y_n = u_n + h\beta_n v_n \\ v_{n+1} = \beta_n^k \cdot ((1 - h\beta_n)v_n - h\nabla f(y_n)) \\ u_{n+1} = u_n + \beta_n(1 - h\beta_n)(y_n - u_n) - h^2\nabla f(y_n) \end{cases}$$

Using a backward Euler method on the nonlinear problem and introducing 2 hyper-parameters k and q for a similar reason we get the second algorithm SSA2

$$\begin{cases} y_n = u_n + h\beta_n v_n \\ v_{n+1} = \beta_n^k \cdot ((1 - h\beta_n)^q v_n - h\nabla f(y_n)) \\ u_{n+1} = u_n + \dfrac{1 - h\beta_n}{\beta_n + \tau}(y_n - u_n) \end{cases}$$

Sequential Splitting Algorithms

Combining the 2 sub-problems, denoting $n/(n+3)$ with β_n and introducing a hyper-parameter k in order to better control the velocity we arrived at the final algorithm SSA1

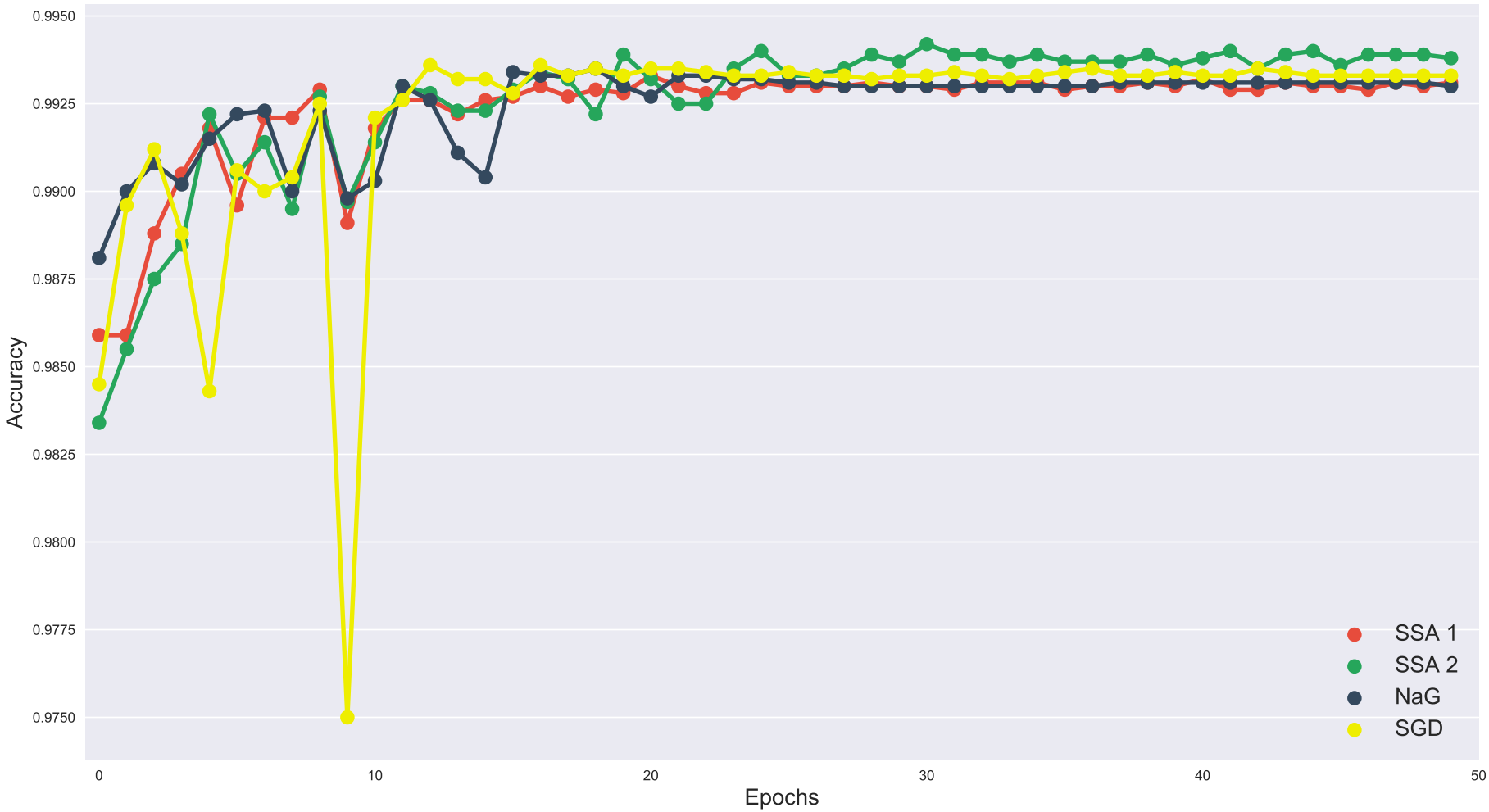
$$\begin{cases} y_n = u_n + h\beta_n v_n \\ v_{n+1} = \beta_n^k \cdot ((1 - h\beta_n)v_n - h\nabla f(y_n)) \\ u_{n+1} = u_n + \beta_n(1 - h\beta_n)(y_n - u_n) - h^2\nabla f(y_n) \end{cases}$$

Using a backward Euler method on the nonlinear problem and introducing 2 hyper-parameters k and q for a similar reason we get the second algorithm SSA2

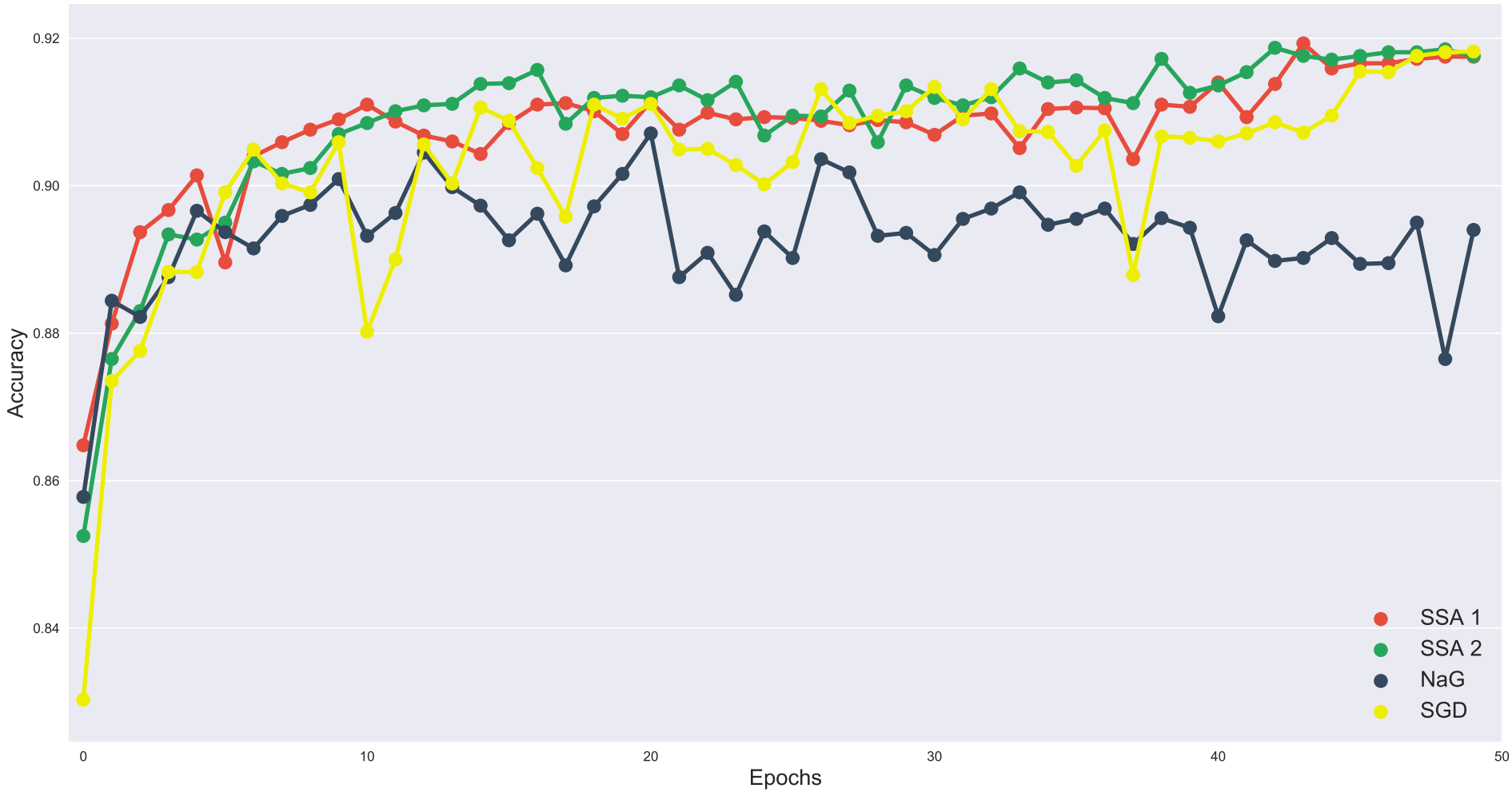
$$\begin{cases} y_n = u_n + h\beta_n v_n \\ v_{n+1} = \beta_n^k \cdot ((1 - h\beta_n)^q v_n - h\nabla f(y_n)) \\ u_{n+1} = u_n + \dfrac{1 - h\beta_n}{\beta_n + \tau}(y_n - u_n) \end{cases}$$

A more detailed construction is available in[2]

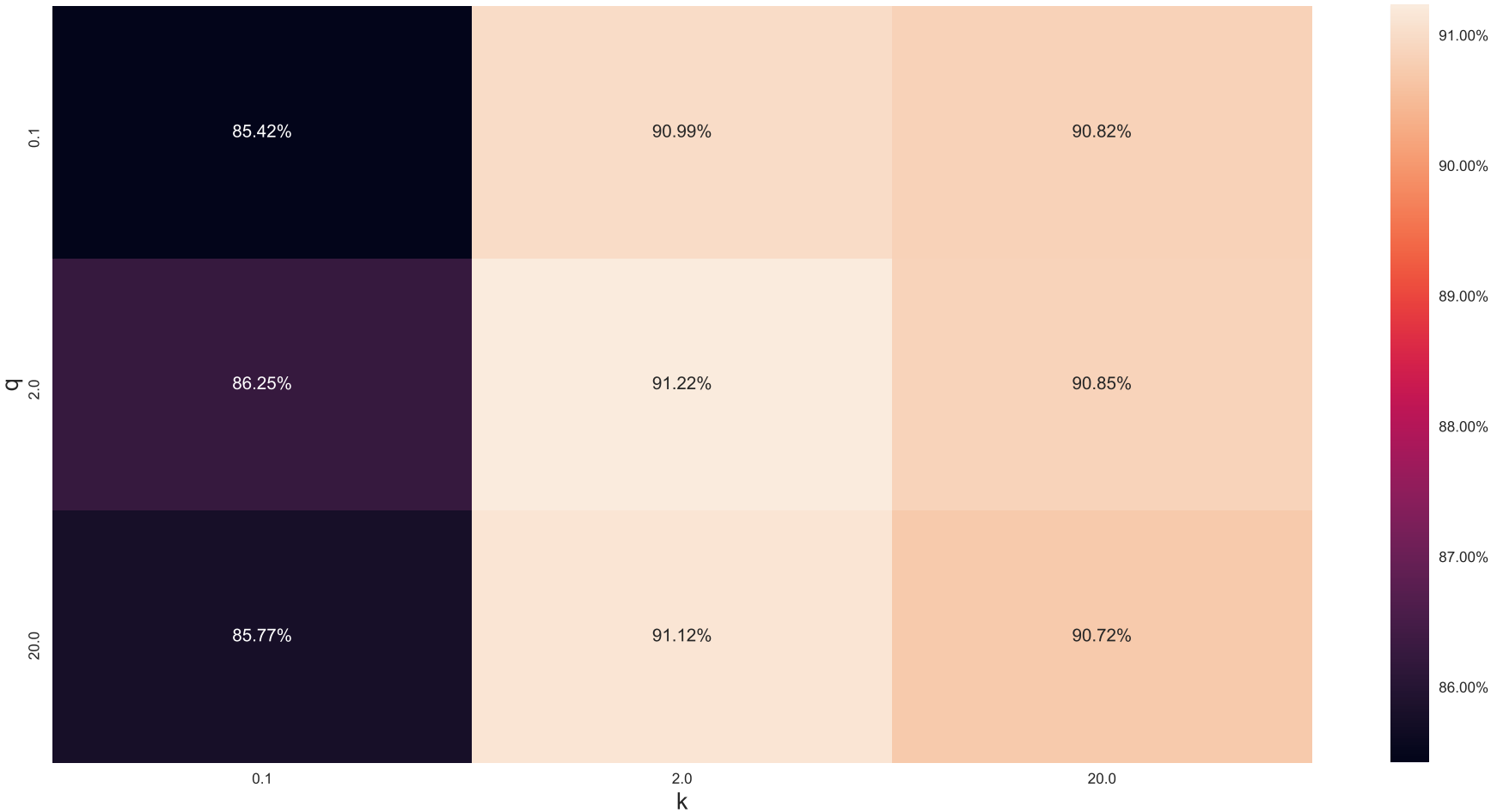
Results


0.9950
0.9925
0.9900
0.9875
0.9850
0.9825
0.9800
0.9775
0.9750
Accuracy
0
10
20
30
40
50
Epochs
SSA 1
SSA 2
NaG
SGD

Results


0.92
0.90
0.88
0.86
0.84
Accuracy
0
10
20
30
40
50
Epochs
SSA 1
SSA 2
NaG
SGD

Results


85.42%
90.99%
90.82%
86.25%
91.22%
90.85%
85.77%
91.12%
90.72%
q
0.1
2.0
20.0
k
0.1
2.0
20.0
91.00%
90.00%
89.00%
88.00%
87.00%
86.00%

Bibliography


M. Gazeau A. B. Da Silva. "A General System of Differential Equations to Model First Order Adaptive Algorithms". In: *aeXiv* (2018). preprint arXiv:1810.13108.


T. Pinta C. D. Alecsa I. Boros. "New Optimization Algorithms for Neural Networks Trainig Using Operator Splitting Techniques". In: (2019). in progress.


Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.


S. Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv* (2016). preprint arXiv:1609.04747.


G. Strang. "On the construction and comparison of difference schemes". In: *SIAM Journal on Numerical Analysis* 5.3 (1968), pp. 506–517.


E. J. Candes W. Su S. Boyde. "A Differential Equation for Modeling Nesterov's Accelerated Gradient Method: Theory and Insights". In: *Jurnal of Machine Learning Research* 17 (2016), pp. 1–43.