Optimization Algorithms

Duc Do

2024-12-09

Some Optimizers Algorithms

There are many algorithms for optimizing a function. One of the most well known is `Gradient Descent`, but there are also a lot of upgraded algorithm from it too. These algorithms are fundamental in every learning model to reduce the loss. This notebook will briefly discuss and implement `Gradient Descent`, `Newton method` and `Stochastic Gradient Descent with momentum`,`AdaGrad`, `RMSProp` and `Adam`.

Function

For example, my goal is to optimize (find local minimum) this function:

$$f(x) = e^{-x^2} + x\cos(x)$$

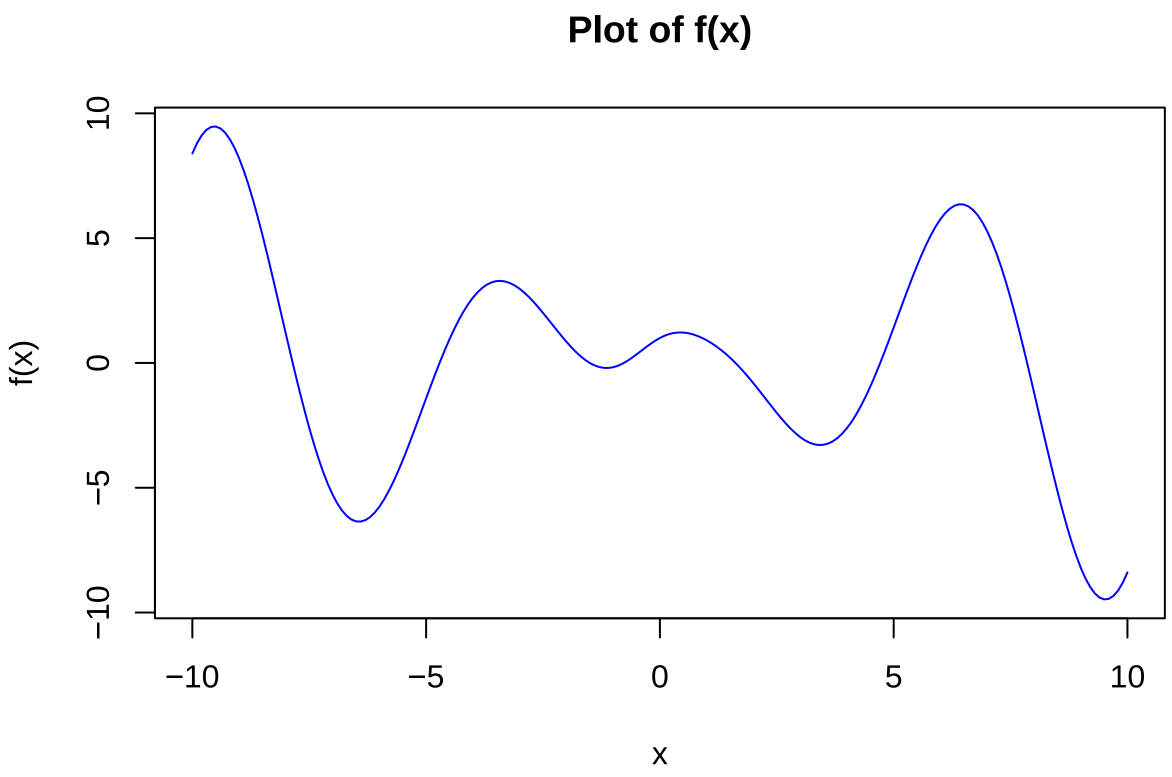
Compute its first derivative and second derivative:

$$f'(x) = -2xe^{-x^2} + \cos(x) - x\sin(x)$$
$$f''(x) = (4x^2 - 2)e^{-x^2} - 2\sin(x) - x\cos(x)$$

It is pretty tedious to find the root of $f'(x) = 0$ as we normally do by hand. That is why an optimization algorithms might be helpful for these kinds of function.

Define and plot the function

```r
# Define the function
f = function(x) {
  exp(-x^2) + x * cos(x)
}

plot_fx = function(x = NULL) {
  x_values = seq(-10, 10, by = 0.1)

  y_values = sapply(x_values, f)

  # Plot the function
  plot(x_values, y_values, type = "l", col = "blue",
       xlab = "x", ylab = "f(x)",
       main = "Plot of f(x)")

  # Plot a point
  if (!is.null(x)) {
    points(x, f(x), col = "red", pch = 19)
  }
}

plot_fx()
```



Its 1st and 2nd derivative:

```
# first derivative
df = function(x) {
  -2 * x * exp(-x^2) + cos(x) - x * sin(x)
}

# second derivative
Hf = function(x) {
  (4 * x^2 - 2) * exp(-x^2) - 2 * sin(x) - x * cos(x)
}
```

- **Common parameters: `n` is the maximum number of iterations, `tol` is the tolerance to check convergence, `x0` is the initial value of `x`.**

Gradient Descent

$$x_{k+1} = x_k - lr \times f'(x_k)$$

Where `lr` is the learning rate. Recall that `f'(x)` represents the slope of `f(x)` at x. So when $f'(x) > 0$, x will go down the slope to the left (x decreases). When $f'(x) < 0$, x will go down the slope to the right (x increases). Proper selection of the initial point `x0` ensures that the algorithm starts in an area of the function where these updates are beneficial for locating a local minimum.

```
GD = function(lr = 1e-2, x0, n = 500, tol = 1e-3) {
  x = x0
  iters = n

  for (i in 1:n) {
    df.x = df(x)

    # if converges
    if (abs(df.x) < tol) {
      iters = i
      break
    }

    x = x - lr * df.x
  }

  return (list(x = x, f.x = f(x), iters = iters))
}
```

Newton method

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

The well-known Newton's method is commonly used to find the root of $f(x) = 0$. When applied to $f'(x) = 0$ for optimization problems, it may converge to a local maximum instead of a local minimum, which is often undesirable in optimization tasks.

```
newton = function(x0, n = 500, tol = 1e-3) {
  x = x0
  iters = n

  for (i in 1:n) {
    df.x = df(x)
    Hf.x = Hf(x)

    # if converges
    if (abs(df.x) < tol) {
      iters = i
      break
    }

    # if f''(x) == 0
    if (Hf.x == 0) {
      iters = i
      stop("Hessian matrix is not invertible at x =", x)
    }
```

```r
    x = x - df.x %*% solve(Hf.x)
  }

  return (list(x = x, f.x = f(x), iters = iters))
}
```

Stochastic Gradient Descent with momentum

This is an improvement of the traditional Gradient Descent to escape local minima.

$$\delta_{x_{k+1}} = b \times \delta_{x_k} + (1-b) \times f'(x_k)$$

$$x_{k+1} = x_k - lr \times \delta_{x_{k+1}}$$

where `b` is the momentum parameter (between 0 and 1) which allows the update of x to be influenced not only by the current gradient but also by the previous update. The idea is from momentum in physics, the further x goes down the slope, the bigger the momentum is. This can help `x` escape local minima, potentially leading to convergence at a global minima.

```r
SGDmomentum = function(lr = 1e-2, x0, n = 500, tol = 1e-3, b = 0.8) {
  x = x0
  delta_x = 0
  iters = n

  for (i in 1:n) {

    df.x = df(x)

    # if converges
    if (abs(df.x) < tol) {
      iters = i
      break
    }

    delta_x = b*delta_x + (1-b)*df.x
    x = x - lr*delta_x
  }

  return (list(x = x, f.x = f(x), iters = iters))
}
```

AdaGrad

`AdaGrad` is also an improvement of Gradient Descent to adjust the `learning rate` dynamically based on the slope and the time into the optimization process to enhance stability.

$$s_{k+1} = s_k + f'(x_k)^2$$
$$x_{k+1} = x_k - \frac{lr}{\sqrt{s_{k+1} + \epsilon}} \times f'(x_k)$$

where `lr` is the initial learning rate, ϵ is a small constant to prevent division by zero. As the optimization progresses, the accumulated squared gradient `s` grows larger resulting in a smaller learning rate, which slows down updates over time especially for parameters experiencing consistently large gradients. Conversely, at the start of optimization, the step size is larger, allowing for faster progress.

This adaptive mechanism addresses the issue of an improperly chosen initial learning rate (too big), which may lead to excessive oscillations or divergence around the local minima. Think of it as friction in physics, slowing progress as `x` nears convergence.

```
AdaGrad = function(lr = 1e-1, x0, n = 500, tol = 1e-3, epsilon = 1e-3) {
  x = x0
  s = 0
  iters = n

  for (i in 1:n) {
    df.x = df(x)

    # if converges
    if (abs(df.x) < tol) {
      iters = i
      break
    }

    s = s + df.x^2
    x = x - lr/sqrt(s + epsilon) * df.x
  }

  return (list(x = x, f.x = f(x), iters = iters))
}
```

RMSProp

Similar to `AdaGrad` but with an important modification: it introduces a decay factor (or weight) to control how the squared gradients are accumulated. This helps address the diminishing learning rate problem in `AdaGrad`, where the learning rate can become excessively small over time.

$$s_{k+1} = a \times s_k + (1-a) \times f'(x_k)^2$$
$$x_{k+1} = x_k - \frac{lr}{\sqrt{s_{k+1} + \epsilon}} \times f'(x_k)$$

where `a` is the decay rate. The current gradient has $1-a$ influence on `s` while older gradients' contributions decrease exponentially with time (similar to δ from SGD with momentum).

```
RMSProp = function(lr = 1e-1, x0, n = 500, tol = 1e-3, epsilon = 1e-3, a = 0.8) {
  x = x0
  s = 0
  iters = n

  for (i in 1:n) {
    df.x = df(x)

    # if converges
    if (abs(df.x) < tol) {
      iters = i
      break
    }

    s = a*s + (1-a) * df.x^2
    x = x - lr/sqrt(s + epsilon) * df.x
  }

  return (list(x = x, f.x = f(x), iters = iters))
}
```

Adam

`Adam` is a combination of `SGD with momentum` and `RMSProp`, making it one of the most widely used optimization algorithms for deep learning.

$$\delta_{x_{k+1}} = b \times \delta_{x_k} + (1-b) \times f'(x_k)$$
$$\hat{\delta}_{x_{k+1}} = \frac{\delta_{x_{k+1}}}{1-b^{k+1}}$$
$$s_{k+1} = a \times s_k + (1-a) \times f'(x_k)^2$$
$$\hat{s}_{k+1} = \frac{s_{k+1}}{1-a^{k+1}}$$
$$x_{k+1} = x_k - \frac{lr}{\sqrt{\hat{s}_{k+1} + \epsilon}} \times \hat{\delta}_{x_{k+1}}$$

Since both δ_x and s are initialized to 0, Adam applies bias corrections to prevent these values from being underestimated, especially in the early stages (as `k` increases, $(1-a^k) \to 1$ which means no more correction).

```
Adam = function(lr = 1e-2, x0, n = 500, tol = 1e-3, epsilon = 1e-3, a = 0.8, b = 0.8) {
  x = x0
  delta_x = 0
  s = 0
  iters = n

  for (i in 1:n) {

    df.x = df(x)

    # if converges
    if (abs(df.x) < tol) {
      iters = i
      break
    }

    delta_x = b*delta_x + (1-b)*df.x
    delta_x = delta_x / (1 - b^i)

    s = a*s + (1-a) * df.x^2
    s = s / (1 - a^i)
    x = x - lr/sqrt(s + epsilon) * delta_x
  }

  return (list(x = x, f.x = f(x), iters = iters))
}
```

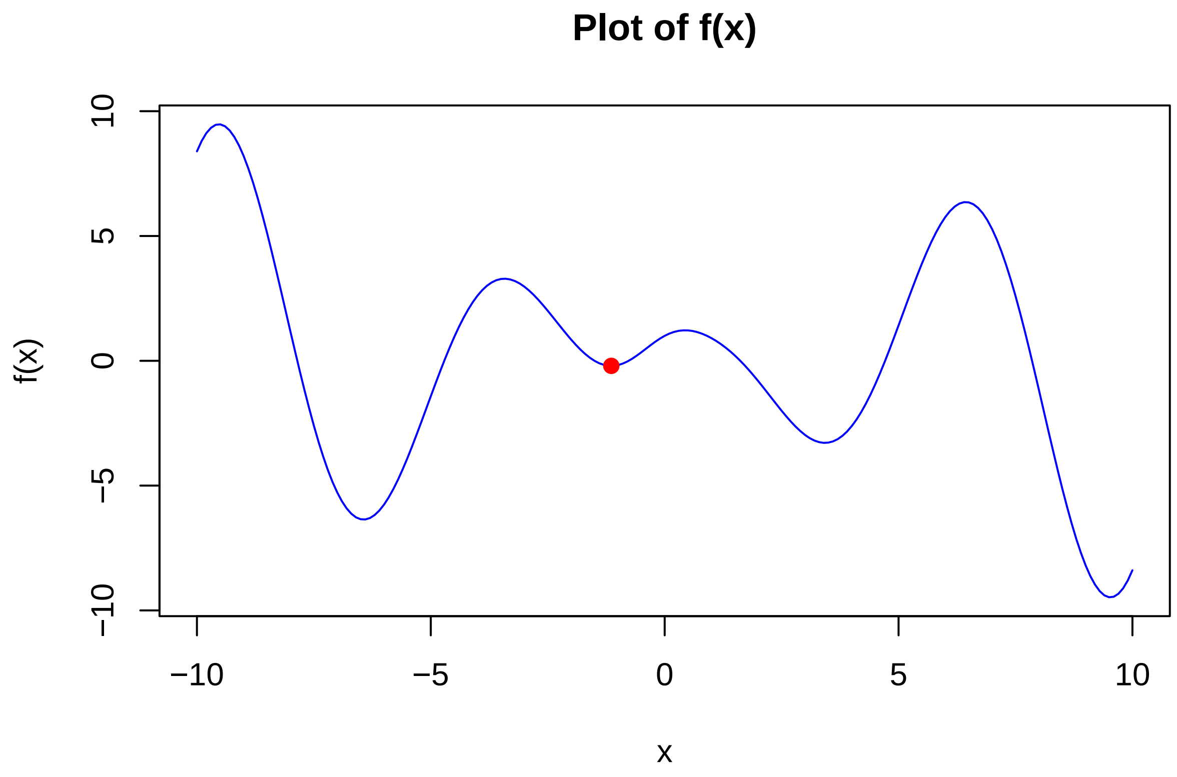
Testing

Gradient Descent

```
result.GD = GD(lr = 1e-2, x0 = 0, n = 1000, tol = 1e-3)
result.GD
```

```
## $x
## [1] -1.140755
##
## $f.x
## [1] -0.2034186
##
## $iters
## [1] 284
```

```
plot_fx(result.GD$x)
```



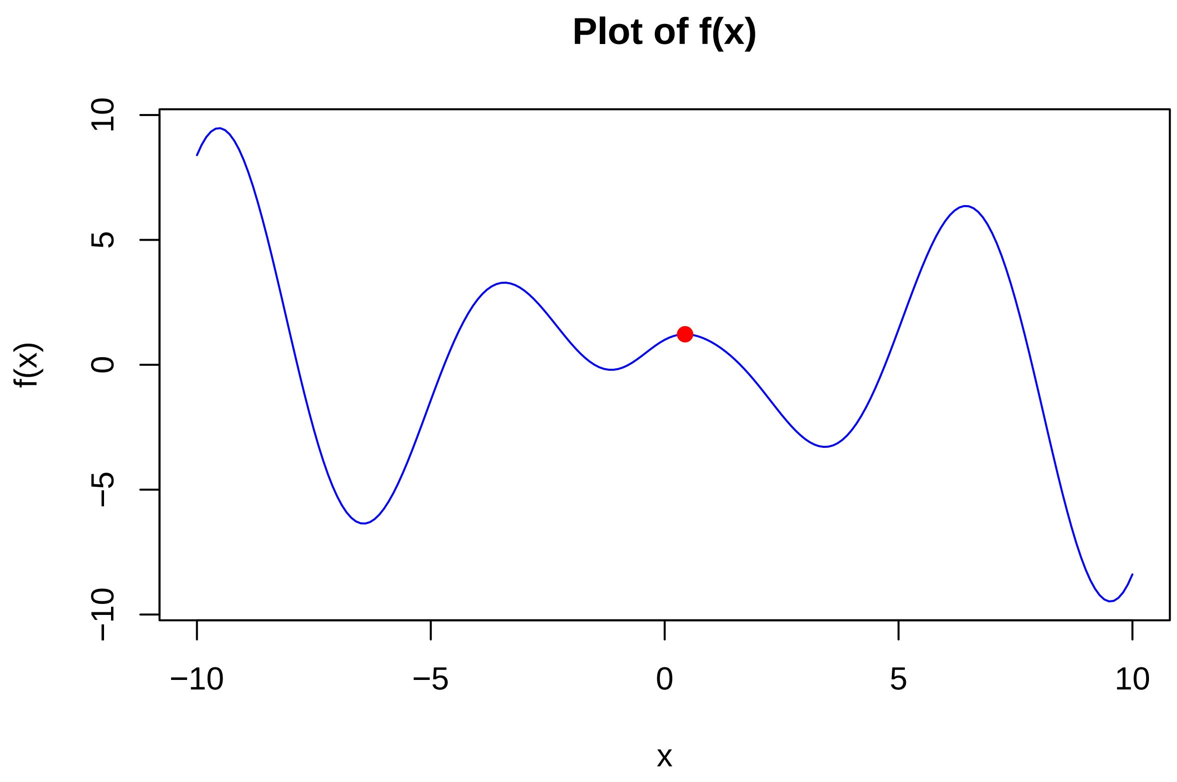
`Gradient Descent` converges after 284 iterations at a local minima x = -1.140755.

Newton method

```r
result.newton = newton(x0 = 0, n = 1000, tol = 1e-3)
result.newton
```

```
## $x
##           [,1]
## [1,] 0.4365642
##
## $f.x
##          [,1]
## [1,] 1.222092
##
## $iters
## [1] 4
```

```r
plot_fx(result.newton$x)
```



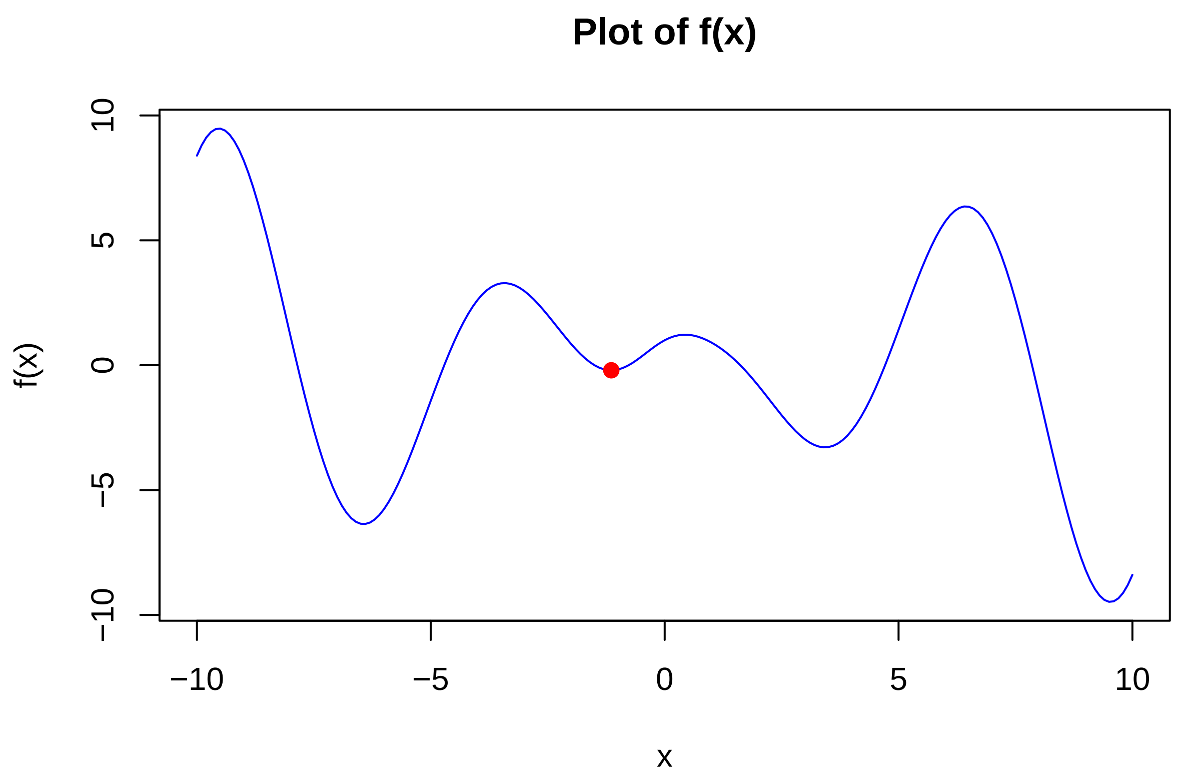
`Newton method` converges after 4 iterations but at a local maxima x = 0.4365642.

SGD with momentum

```r
result.SGDmomentum = SGDmomentum(lr = 1e-2, x0 = -3, n = 1000, tol = 1e-3, b = 0.9)
result.SGDmomentum
```

```
## $x
## [1] -1.14133
##
## $f.x
## [1] -0.2034186
##
## $iters
## [1] 169
```

```r
plot_fx(result.SGDmomentum$x)
```



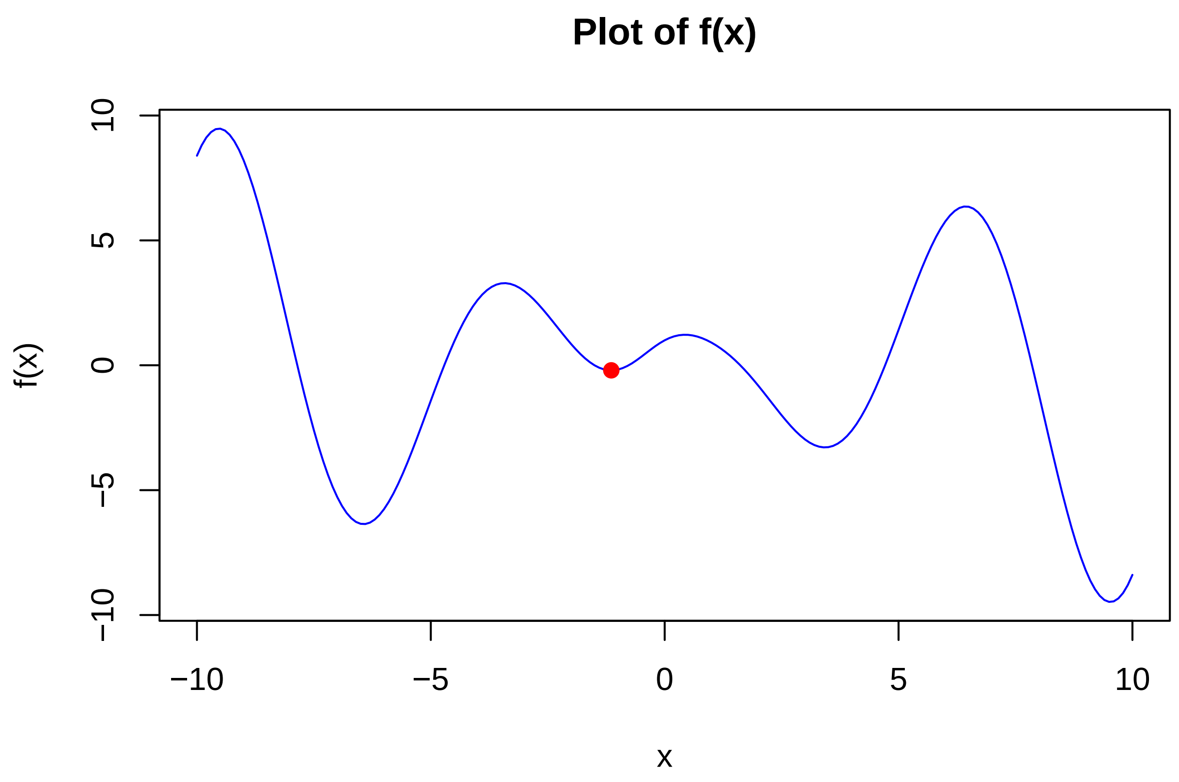
`SGD with momentum` converges at local mimina x = -1.14133. Unfortunately, it still cannot escape this local minima.

AdaGrad

```r
result.AdaGrad = AdaGrad(lr = 1e-1, x0 = -3, n = 1000, tol = 1e-3, epsilon = 1e-3)
result.AdaGrad
```

```
## $x
## [1] -1.141376
##
## $f.x
## [1] -0.2034186
##
## $iters
## [1] 445
```

```r
plot_fx(result.AdaGrad$x)
```



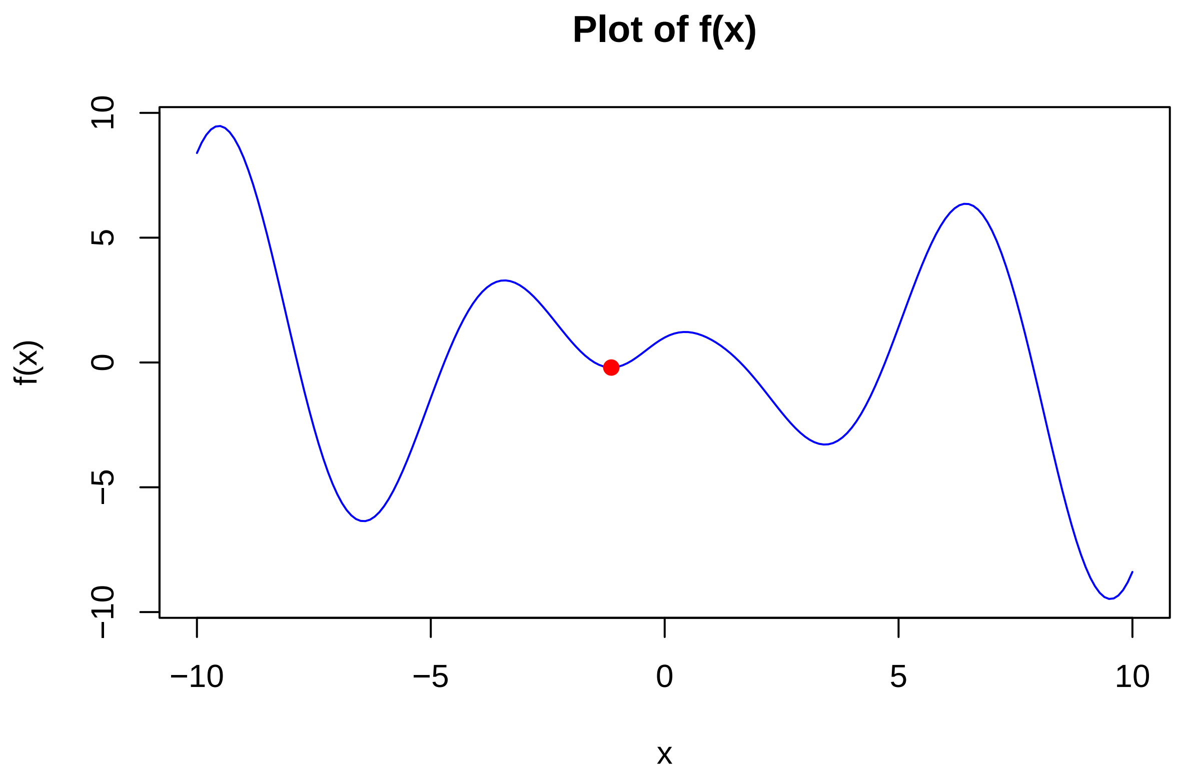
`AdaGrad` converges after 445 iterations at local mimina x = -1.141376.

RMSProp

```r
result.RMSProp = RMSProp(lr = 1e-1, x0 = -3, n = 1000, tol = 1e-3, epsilon = 1e-3, a = 0.8)
result.RMSProp
```

```
## $x
## [1] -1.141162
##
## $f.x
## [1] -0.2034187
##
## $iters
## [1] 27
```

```
plot_fx(result.RMSProp$x)
```



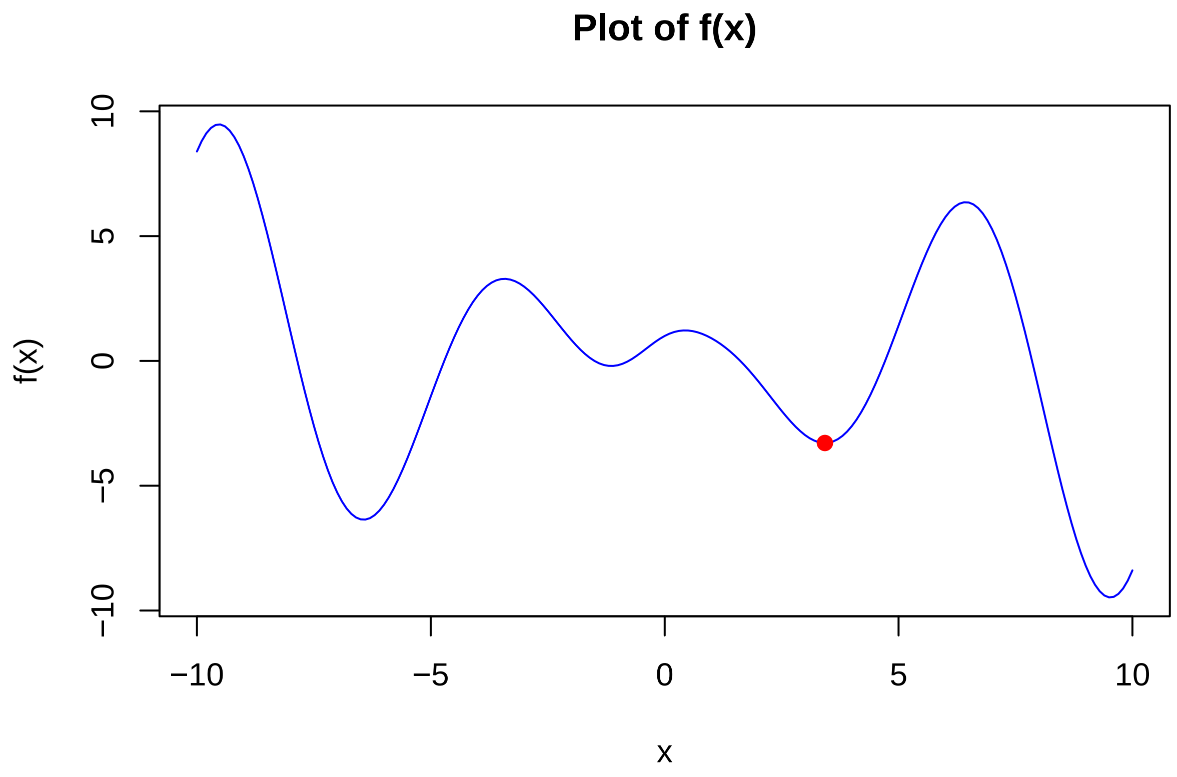
RMSProp converges after 27 iterations at local mimina x = -1.141162.

Adam

```
result.Adam = Adam(lr = 1e-1, x0 = -3, n = 1000, tol = 1e-3, epsilon = 1e-3, a = 0.8)
result.Adam
```

```
## $x
## [1] 3.425415
##
## $f.x
## [1] -3.288363
##
## $iters
## [1] 87
```

```r
plot_fx(result.Adam$x)
```



`Adam` converges after 87 iterations at local mimina x = 3.425415, which is a different local minima from others.