深度学习基础

2022 年 9 月 30 日

Homework 1 and 2

Question 1. 实现正向传播和反向传播的推导

在进行证明前，需要先定义一些符号：

表 1. 符号说明

符号含义	含义
$a^{(l)}$	第 l 层经过激活函数激活后的输出
$a^{(L)}$	最后一层输出，这时 $y = a^{(L)}$
x	输入数据, 并且 $x = a^{(1)}$
$w_{ij}^{(l)}$	权重，l 代表第 l 层，i 代表第 l 层的第 i 个神经元，j 代表第 l-1 层的第 j 个神经元
$b^{(L)}$	第 l 层的偏置
$z^{(l)}$	对第 l-1 层的输出进行仿射变化得到结果，也就是 $z^{(l)} = w^{(l)}a^{(l-1)} + b^{(l)}$
$\sigma_{(l)}$	第 l 层的激活函数，并且有 $a^{(l)} = \sigma_{(l)}(z^{(l)})$
$\delta_j^{(l)}$	损失函数对于第 l 层第 j 个加权输出的偏导，也就是 $\delta_j^{(l)} = \frac{\partial E}{\partial z_j^{(l)}}$
t	输出的监督值
$E(y,t)$	损失函数, 这里才用 L2 范数，也就是 $E(y,t) = \frac{\sum_k (y_k - t_k)^2}{2}$
θ	权重, 就是 w 的总称
L	最后一层的神经网络的层数

正向传播和方向传播都是为了求解得到损失函数关于权重以及偏置的梯度，采用的都是链式法则。

(a) 反向传播公式推导：

由链式法则，可以得到:

$$\begin{cases} \frac{\partial E}{\partial w_{ij}^{(l)}} = \frac{\partial E}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}} & \text{(1a)} \\ \frac{\partial E}{\partial b_j^{(l)}} = \frac{\partial E}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial b_j^{(l)}} & \text{(1b)} \end{cases}$$

对于 (1a) 和 (1b) 式子，可以化为:

$$\begin{cases} \frac{\partial E}{\partial w_{ij}^{(l)}} = \delta^{(l)} \frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}} & \text{(2a)} \\ \frac{\partial E}{\partial b_j^{(l)}} = \delta^{(l)} \frac{\partial z^{(l)}}{\partial b_j^{(l)}} & \text{(2b)} \end{cases}$$

所以，也就是求解 $\delta^{(l)}, \frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}}, \frac{\partial z^{(l)}}{\partial b_j^{(l)}}$ 这三个变量。

对于该神经网络，假设我们使用的激活函数都为同一个激活函数，也就是 $\sigma = \sigma_{(l)}$, 为了方便起见，后文中将激活函数写为：σ。

首先先求解 $\delta^{(l)}$:

- 求解 $\delta^{(l)}$:

  对于最后一层神经网络，我们从定义中知道：

$$a^{(L)} = \sigma(z^{(L)})$$

  那么对于最后一层:

$$\delta^{(L)} = \frac{\partial E}{\partial z^{(L)}} = \frac{\partial \frac{\sum_k (y_k - a_k^{(L)})^2}{2}}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}}$$

  也就是

$$\delta^{(L)} = \sum_k (a^{(L)} - y_k)\sigma'(z^{(L)})$$

  当 $1 \leq l \leq L-1$ 时, 我们这里采用链式法则来计算，由于是反向传播，所以我们选择用 l+1 层的偏导来表示，也就是:

$$\delta^{(l)} = \frac{\partial E}{\partial z^{(l)}} = \frac{\partial E}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial z^{(l)}} = \delta^{(l+1)} \frac{\partial z^{(l+1)}}{\partial z^{(l)}}$$

  这是一个递推关系式，为了进一步求解，我们需要求解 $\frac{\partial z^{(l+1)}}{\partial z^{(l)}}$ 从定义中可以知道:

$$z^{(l+1)} = w^{(l+1)} a^{(l)} + b^{(l+1)} = w^{(l+1)} \sigma(z^{(l)}) + b^{(l+1)}$$

  所以有:

$$\frac{\partial z^{(l+1)}}{\partial z^{(l)}} = w^{(l+1)} \sigma'(z^{(l)})$$

  那么这样我们便可以得到 $\delta^{(l)}$ 的表达式为:

$$\delta^{(l)} = \sigma'(z^{(l)})(w^{(l+1)})^T \delta^{(l+1)}$$

  至此我们完成了第一步，也就是求解 $\delta^{(l)}$ 然后我们求解 $\frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}}, \frac{\partial z^{(l)}}{\partial b_j^{(l)}}$ 这两个即可，便可以完成反向传播的推导。

- 求解 $\frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}}$

  从定义中可以知道:

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

  所以有:

$$\frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}} = a^{(l-1)}$$

- 求解 $\frac{\partial z^{(l)}}{\partial b_j^{(l)}}$

  从定义中可以知道:

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

  所以有:

$$\frac{\partial z^{(l)}}{\partial b_j^{(l)}} = 1$$

至此，我们便完成了反向传播的推导，也就是:

$$
\begin{cases}
\frac{\partial E}{\partial w_{ij}^{(l)}} = \delta^{(l)} a^{(l-1)} & \text{(3a)} \\
\frac{\partial E}{\partial b_j^{(l)}} = \delta^{(l)} & \text{(3b)} \\
\text{其中}: \delta^{(l)} = \begin{cases} \sigma'(z^{(l)})(w^{(l+1)})^T \delta^{(l+1)}, & l = 1...L-1 \\ \sum_k (a^{(L)} - y_k)\sigma'(z^{(L)}), & l = L \end{cases} & \text{(3c)}
\end{cases}
$$

(b) 正向传播公式推导：

所谓正向传播，与上面不同的地方仅仅存在于关于 $\delta^{(l)}$ 求解过程中的链式法则使用上，这里将使用 l-1 层的偏导来表示，所以被称为正向传播。所以我们只需要改写其中关于 $\delta^{(l)}$ 的求解部分即可，其他部分与反向传播相同。

- 对于第一层神经网络，我们从定义中知道：

$$a^{(1)} = x$$

那么对于第一层:

$$\delta^{(L)} = \frac{\partial E}{\partial z^{(L)}} = \frac{\partial E}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} = \frac{\partial E}{\partial x} \frac{\partial x}{\partial z^{(L)}}$$

由于 x 是常数，所以 $\frac{\partial x}{\partial z^{(L)}} = 0$ 也就是:

$$\delta^{(L)} = 0$$

- 当 $2 \leq l \leq L$ 时, 我们这里采用链式法则来计算，由于是正向传播，所以我们选择用 l-1 层的偏导来表示，也就是:

$$\delta^{(l)} = \frac{\partial E}{\partial z^{(l)}} = \frac{\partial E}{\partial z^{(l-1)}} \frac{\partial z^{(l-1)}}{\partial z^{(l)}} = \delta^{(l-1)} \frac{\partial z^{(l-1)}}{\partial z^{(l)}}$$

这是一个递推关系式，为了进一步求解，我们需要求解 $\frac{\partial z^{(l+1)}}{\partial z^{(l)}}$ 从定义中可以知道:

$$z^{(l+1)} = w^{(l+1)} a^{(l)} + b^{(l+1)} = w^{(l+1)} \sigma(z^{(l)}) + b^{(l+1)}$$

所以有:

$$z^{(l-1)} = \sigma^{-1}(\frac{z^{(l)} - b^{(l)}}{w^{(l)}})$$

那么可以得到:

$$\frac{\partial z^{(l-1)}}{\partial z^{(l)}} = (\sigma^{-1}(\frac{z^{(l)} - b^{(l)}}{w^{(l)}}))'$$

那么这样我们便可以得到 $\delta^{(l)}$ 的表达式为:

$$\delta^{(l)} = \delta^{(l-1)}(\sigma^{-1}(\frac{z^{(l)} - b^{(l)}}{w^{(l)}}))'$$

至此，我们便完成了正向传播的推导，也就是:

$$
\begin{cases}
\dfrac{\partial E}{\partial w_{ij}^{(l)}} = \delta^{(l)} a^{(l-1)} & \text{(4a)} \\
\dfrac{\partial E}{\partial b_j^{(l)}} = \delta^{(l)} & \text{(4b)} \\
\text{其中}: \delta^{(l)} = \begin{cases} \delta^{(l-1)}(\sigma^{-1}(\frac{z^{(l)}-b^{(l)}}{w^{(l)}}))', & l = 2...L \\ 0, & l = 1 \end{cases} & \text{(4c)}
\end{cases}
$$

Question 2. 对于一个神经网络 $y = DNN(x, \theta)$, 试求解 $\frac{\partial y}{\partial x}$

从上面，推导过程，不难看出，对于一个 n 层的神经网络，如果其输出为 y，那么有:

$$y = DNN(x, \theta) = \sigma(w^{(n)}\sigma(w^{(n-1)}\sigma(....) + b^{(n-1)}) + b^{(n)})$$

所以一个神经网络的输出对于其输入的偏导可以表示为:

$$\frac{\partial y}{\partial x} = \frac{\partial DNN(x,\theta)}{\partial x} = \sigma'(z^{(n)} + b) \times w^{(n)} \times \sigma'(z^{(n-1)} + b) \times w^{(n-1)} \times \cdots \times \sigma'(w^{(1)}x + b^{(1)}) \times w^{(1)}$$

或者采用数值求导的方法, 令 h 为一个较小的值，那么：

$$\frac{\partial y}{\partial x} = \frac{\partial DNN(x,\theta)}{\partial x} = \frac{DNN(x+h,\theta) - DNN(x-h,\theta)}{2h}$$

Question 3. 对于以下实现以下几种算法，并画出其对应的收敛情况。

1) 1:gradient descend
2) 2:Ada grad
3) 3:RMS prop
4) 4:Momentum
5) 5:Nesterov
6) 6:Adam

对于各个算法的实现如下:

```python
"""
This project will include the normal optimization algorithms for the DeepLearning :
1:gradient descend
2:Ada grad
3:RMS prop
4:Momentum
5:Nesterov
6:Adam
"""
import numpy as np
import matplotlib.pyplot as plt


class OptimizationAlgorithm:
    def __init__(self, obj_func, num_var, lower_bound, upper_bound, epoch, analytical_grad=None, iter_hist=False):
        """
        This is the initialization of this problem , and it is usually defined as a minimizing problem .
        Parameters
        ----------
```

```
        obj_func: Objective function
        num_var: number of variables for this function
        lower_bound:  the lower bound for the variables . Obviously , this is a list .
        upper_bound:  the upper bound for the variables . Obviously , this is a list .
        epoch: the number of the iteration.
        analytical_grad: the analytical grad for the function if given .
        iter_hist: whether store the value for each step .
        """

        self.obj_func = obj_func
        self.num_var = num_var
        self.lower_bound = lower_bound
        self.upper_bound = upper_bound
        self.epoch = epoch
        if analytical_grad:
            self.analytical_grad = analytical_grad
            self.has_analytical_grad = True
        else:
            self.has_analytical_grad = False
        if iter_hist:
            self.has_iter_hist = True
            self.iter_hist = np.array(np.zeros(shape=(self.num_var,)))
        else :
            self.has_iter_hist = False

    @staticmethod
    def numerical_gradient(f, x, h=1e-8):
        """

        Parameters
        ----------
        f:objective function
        x:at where the grad is obtained
        h:the step

        Returns
        -------
        return the corresponding gradient
        """
        grad = np.zeros_like(x)

        it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
        while not it.finished:
            idx = it.multi_index
            tmp_val = x[idx]
            x[idx] = float(tmp_val) + h
            fxh1 = f(x)  # f(x+h)
            x[idx] = tmp_val - h
            fxh2 = f(x)  # f(x-h)
            grad[idx] = (fxh1 - fxh2) / (2 * h)
            x[idx] = tmp_val
            it.iternext()

        return grad

    def gradient_descend(self, grad_method, lr=1e-3, display_process=False):
        """
        Use the general gradient descend method to obtain the optimal solution.
        Parameters
        ----------
        grad_method: The method to obtain gradient , by analytical or numerical
```

```
        lr: The learning rate .
        display_process: whether display the iteration process.

        Returns
        -------
        Return the x , where the objective function gets the minimum.
        """
        x = np.random.uniform(low=self.lower_bound, high=self.upper_bound)
        for i in range(self.epoch):
            if grad_method == "analytical" and self.has_analytical_grad:
                # use the analytical method to obtain the gradient
                grad = self.analytical_grad(x)
            if grad_method == "numerical":
                # use the numerical method to obtain the gradient
                grad = self.numerical_gradient(self.obj_func, x)
            if self.has_iter_hist:
                self.iter_hist = np.vstack((self.iter_hist, x))
            x -= lr*grad
            if display_process:
                if i % 50 == 0:
                    print(f"epoch: {i}/{self.epoch},  x={x},  f={self.obj_func(x)}")

        return x

    def ada_grad(self, grad_method, lr=1e-1, display_process=False):
        """
        Use the adaptive gradient descend method to obtain the optimal solution.
        For the sumation of the grad will accumulate , so the step for each iteration
        will approach the 0 . In the end , the iteration will terminate .
        Parameters
        ----------
        grad_method: The method to obtain gradient , by analytical or numerical
        lr: The learning rate .
        display_process: whether display the iteration process.

        Returns
        -------
        Return the x , where the objective function gets the minimum.
        """
        # for the ada grad algorithm , a list to record the history should be maintained
        gt = 0
        x = np.random.uniform(low=self.lower_bound, high=self.upper_bound)
        for i in range(self.epoch):
            if grad_method == "analytical" and self.has_analytical_grad:
                # use the analytical method to obtain the gradient
                grad = self.analytical_grad(x)
            if grad_method == "numerical":
                # use the numerical method to obtain the gradient
                grad = self.numerical_gradient(self.obj_func, x)
            if self.has_iter_hist:
                self.iter_hist = np.vstack((self.iter_hist, x))
            # get the accumulative grad
            gt += np.sum(np.square(grad))
            # The Ada grad algorithm is implemented!
            x -= lr*grad/np.sqrt(gt+1e-9)
            if display_process:
                if i % 50 == 0:
                    print(f"epoch: {i}/{self.epoch},  x={x},  f={self.obj_func(x)}")

        return x
```

```
    def rms_prop(self, grad_method, lr=1e-2, display_process=False):
        """
        Use the rms prop method to obtain the optimal solution.
        For the sumation of the grad will accumulate , so the step for each iteration
        will approach the 0 . In the end , the iteration will terminate .
        Parameters
        ----------
        grad_method: The method to obtain gradient , by analytical or numerical
        lr: The learning rate .
        display_process: whether display the iteration process.

        Returns
        -------
        Return the x , where the objective function gets the minimum.
        """
        # for the rms prop algorithm , a sumation of the grad should be maintained .
        gt = 0
        beta = 0.9

        x = np.random.uniform(low=self.lower_bound, high=self.upper_bound)
        # self.w = np.array((x[0], x[1]))
        for i in range(self.epoch):
            if grad_method == "analytical" and self.has_analytical_grad:
                # use the analytical method to obtain the gradient
                grad = self.analytical_grad(x)
            if grad_method == "numerical":
                # use the numerical method to obtain the gradient
                grad = self.numerical_gradient(self.obj_func, x)
            if self.has_iter_hist:
                self.iter_hist = np.vstack((self.iter_hist, x))
            # get the accumulative grad
            gt = beta*gt+(1-beta)*np.sum(np.square(grad))
            # The Ada grad algorithm is implemented!
            x -= lr * grad / (np.sqrt(gt) + 1e-9)
            if display_process:
                if i % 50 == 0:
                    print(f"epoch: {i}/{self.epoch},  x={x},  f={self.obj_func(x)}")

        return x

    def momentum(self, grad_method, lr=1e-3, display_process=False):
        """
        Use the momentum modified general gradient descend method to obtain the optimal solution.
        The momentum method will use the momentum to avoid the local optimal
        Parameters
        ----------
        grad_method: The method to obtain gradient , by analytical or numerical
        lr: The learning rate .
        display_process: whether display the iteration process.

        Returns
        -------
        Return the x , where the objective function gets the minimum.
        """
        x = np.random.uniform(low=self.lower_bound, high=self.upper_bound)
        # the initial velocity
        velocity = np.zeros_like(x)
        momentum_factor = 0.1
        for i in range(self.epoch):
            if grad_method == "analytical" and self.has_analytical_grad:
                # use the analytical method to obtain the gradient
```

```
                grad = self.analytical_grad(x)
            if grad_method == "numerical":
                # use the numerical method to obtain the gradient
                grad = self.numerical_gradient(self.obj_func, x)
            if self.has_iter_hist:
                self.iter_hist = np.vstack((self.iter_hist, x))
            # update the velocity
            velocity = velocity*momentum_factor-lr*grad
            x = x+velocity
            if display_process:
                if i % 50 == 0:
                    print(f"epoch: {i}/{self.epoch},  x={x},  f={self.obj_func(x)}")

        return x

    def nesterov(self, grad_method, lr=1e-3, display_process=False):
        """
        Use the momentum modified general gradient descend method to obtain the optimal solution.
        The momentum method will use the momentum to avoid the local optimal.
        Nesterov method is a kind of acceleration method that step ahead a bit before
        the gradient descend .
        Parameters
        ----------
        grad_method: The method to obtain gradient , by analytical or numerical
        lr: The learning rate .
        display_process: whether display the iteration process.

        Returns
        -------
        Return the x , where the objective function gets the minimum.
        """
        x = np.random.uniform(low=self.lower_bound, high=self.upper_bound)
        # the initial velocity
        velocity = np.zeros_like(x)
        momentum_factor = 0.1

        # then all of the grads should be acquired in the accelerated_x rather than x
        for i in range(self.epoch):
            # accelerate the x first before obtaining the grad
            accelerated_x = x + velocity * momentum_factor
            if grad_method == "analytical" and self.has_analytical_grad:
                # use the analytical method to obtain the gradient
                grad = self.analytical_grad(accelerated_x)
            if grad_method == "numerical":
                # use the numerical method to obtain the gradient
                grad = self.numerical_gradient(self.obj_func, accelerated_x)
            if self.has_iter_hist:
                self.iter_hist = np.vstack((self.iter_hist, x))
            # update the velocity , this is the same as the momentum method
            velocity = velocity * momentum_factor - lr * grad
            x = x + velocity
            if display_process:
                if i % 50 == 0:
                    print(f"epoch: {i}/{self.epoch},  x={x},  f={self.obj_func(x)}")

        return x

    def adam(self, grad_method, lr=1e-1, display_process=False):
        """
        This is the implementation for the adam algorithm , which
        is the combination of the ada grad and momentum algorithm.
```

```
        Parameters
        ----------
        grad_method: The method to obtain gradient , by analytical or numerical
        lr: The learning rate .
        display_process: whether display the iteration process.

        Returns
        -------
        Return the x , where the objective function gets the minimum.

        """
        # the declining rate for the moment estimation
        rho1, rho2 = 0.9, 0.999
        delta = 1e-8
        gt = 0
        x = np.random.uniform(low=self.lower_bound, high=self.upper_bound)
        # initialize the first moment and the second moment
        s = np.zeros_like(x)
        r = np.zeros_like(x)
        for i in range(self.epoch):
            if grad_method == "analytical" and self.has_analytical_grad:
                # use the analytical method to obtain the gradient
                grad = self.analytical_grad(x)
            if grad_method == "numerical":
                # use the numerical method to obtain the gradient
                grad = self.numerical_gradient(self.obj_func, x)
            if self.has_iter_hist:
                self.iter_hist = np.vstack((self.iter_hist, x))
            # update the first and the second moment
            s = rho1*s+(1-rho1)*grad
            r = rho2*r+(1-rho2)*np.square(grad)
            # get the partial first and second moment
            s_hat = s/(1-rho1**(i+1))
            r_hat = r/(1-rho1**(i+1))
            # use the partial first and second moment to modify the GD method
            x -= lr * s_hat / (np.sqrt(r_hat)+delta)
            if display_process:
                if i % 50 == 0:
                    print(f"epoch: {i}/{self.epoch},  x={x},  f={self.obj_func(x)}")

        return x

    def draw_process(self):
            x = np.arange(self.lower_bound[0], self.upper_bound[0], 0.1)
            y = np.arange(self.lower_bound[1], self.upper_bound[1], 0.1)
            [x, y] = np.meshgrid(x, y)
            plt.contour(x, y,  x**2/20+y**2, 20)
            w = self.iter_hist[1:]
            plt.plot(w[:, 0], w[:, 1], 'g*', w[:, 0], w[:, 1])
            plt.show()

    def clear_hist(self):
        self.iter_hist = np.array(np.zeros(shape=(self.num_var,)))


if __name__ == '__main__':
    obj_func = lambda x: x[0]**2/20+x[1]**2
    # solver = OptimizationAlgorithm(obj_func=obj_func, num_var=2, lower_bound=[-100, -100], upper_bound=[100, 100], epoch=1000000)
    # optimal_x = solver.gradient_descend(grad_method="numerical", display_process=True)
    # save the data for each algorithm
```

```
    solver = OptimizationAlgorithm(obj_func=obj_func, num_var=2,
                                   lower_bound=[-100, -100], upper_bound=[100, 100],
                                   epoch=20000, analytical_grad=lambda x: np.array([x[0]/10, 2*x[1]]), iter_hist=
True)
    optimal_x = solver.adam(grad_method="numerical", display_process=True)
    solver.draw_process()
    print(f"optimal_x:{optimal_x}")
    print(f"minimum function value {obj_func(optimal_x)}")
    # draw_process(solver.w)
```

Listing 1. 基于 python3 对于 6 中优化算法的实现

得到的迭代曲线如下:

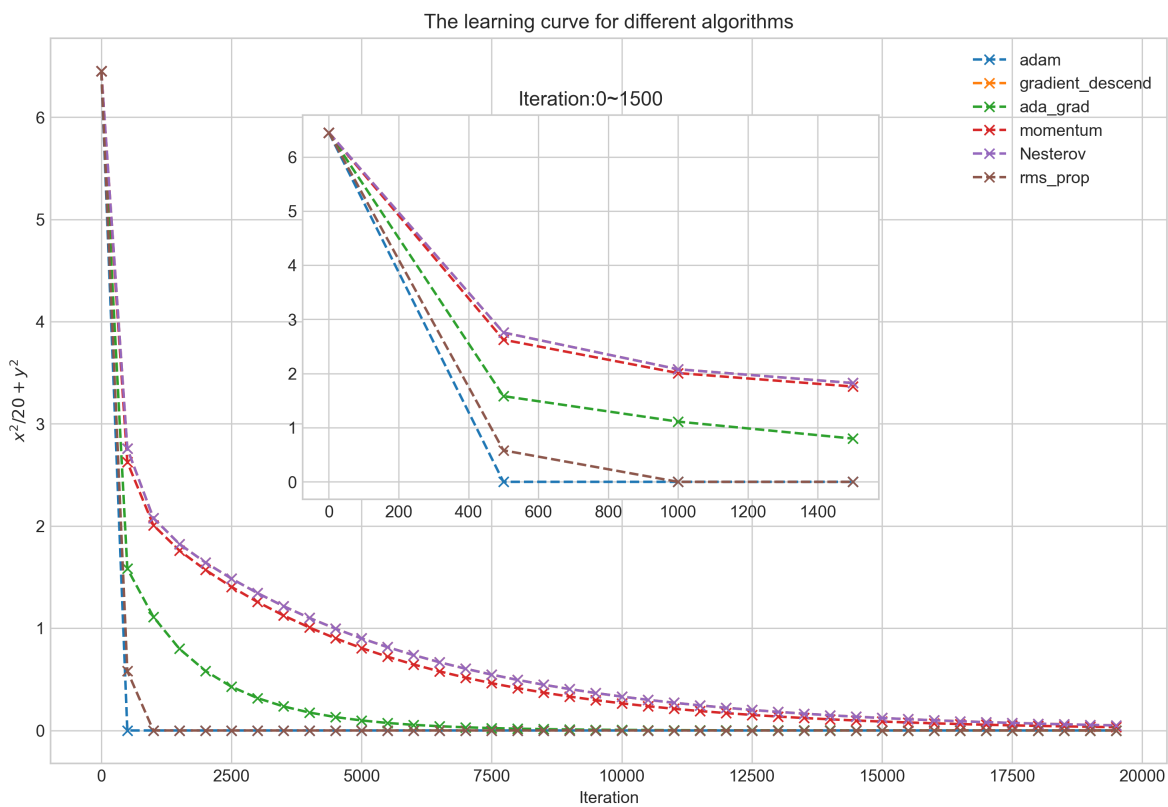

图 1. The iteration curve for the six kinds of optimization algorithms

从迭代曲线不难看出，adam 算法的收敛速度最快，其次的时 RMS prop 算法，然后剩下四种算法速度都比较慢，其中 Ada grad 最快，剩下的动量法，梯度下降以及 Nesterov 速度基本上一样，都比较慢。

速度较快的算法都是基于 Ada grad 来实现的，所以速度较快，所以 Adam 和 RMS prop 算法速度较快，这两个都是基于 Ada grad 的变种。而 Nesterov 算法实现时是对动量法增加了一个提前一步的迭代，都是基于最基本的梯度下降的优化，所以最后的收敛速度都是比较慢的。

Question 4. 对于一个隐藏层，输入层，输出层均为 100 的五层神经网络，给予一个满足高斯分布的输入 $x \in R^{1000\times 100}$ 分别讨论以下情况的每层输出的分布, 并画出直方图：

- $\sigma = \begin{cases} sigmoid \\ relu \end{cases}$
- $\theta = \begin{cases} 0 \\ \sim N(0,\delta), \qquad \delta = 1 \quad or \quad 0.01 \end{cases}$

对神经网络进行初始化，然后进行正向传播，得到不同参数的神经网络的每一个输出层的直方图如下:

由于输出的图片多达 40 个，所以这里给出图片的链接:

点击这个超链接查看所有输出图片。

从输出的图片的直方图可以看出：

1) 当 $\theta = 0$ 时，由于线性层的权值和偏置都是 0，所以不管输入的值是多少进行加权求和以及激活后得到的值都是 0。所以此时第 1 到第 5 层的输出层都是 0。
2) 当 $\theta \sim N(0,1)$ 时, 也就是其满足标准正态分布时：

   当激活函数为 Sigmoid 函数时：第一层 a_1 输出是基本满足单峰正态分布的，a_2 的输出的分布就发生了部分的偏离。而 $a_3 a_4$ 基本满足多峰正态分布，而对于 a_5 层就趋向于均匀分布了。

   当激活函数为 Relu 函数时：对于所有的输出层，直方图中输出分布都是 0 值较多，其他值的数量都较少。

   而 δ 影响的就是分布的方差，也就是分布的范围，当 δ 较大时 $(\delta = 1)$ 分布更广。
3) 对于上述情况，应该是 Relu 函数对于小于 0 的输入输出为 0 所以导致了大部分的输出都是 0，而 Sigmoid 对于正负的输入都可以激活，这里面的由于是标准正态分布，所以还没达到 Sigmoid 函数的饱和区。