CS361 course project

Zayd Khan

November 26, 2024

1 THEORY

Part 1: Stochastic Optimization Theory

Exercise 1

- **Solution for (a):**

$$E[g(\theta)] = E[Q(\theta, i)] = \sum_{i=1}^{k} P(I = i) Q(\theta, i)$$

Since i follows a discrete uniform distribution over $[1, k]$, we have:

$$P(I = i) = \frac{1}{k}, \text{ for all } i \in [1, k].$$

Substituting this into the summation:

$$E[g(\theta)] = \sum_{i=1}^{k} \frac{1}{k} Q(\theta, i).$$

The constant $\frac{1}{k}$ can be factored out of the summation:

$$E[g(\theta)] = \frac{1}{k} \sum_{i=1}^{k} Q(\theta, i).$$

Renaming the index i to j, this matches the formula for $f(\theta)$:

$$\boxed{E[g(\theta)] = f(\theta)}$$

- **Solution for (b):**

The gradient of $g(\theta)$ is the gradient of $Q(\theta, i)$. Therefore:

$$E[\nabla g(\theta)] = E[\nabla Q(\theta, i)] = \sum_{i=1}^{k} \frac{1}{k} \nabla Q(\theta, i).$$

Renaming the index i to j, we get:

$$E[\nabla g(\theta)] = \frac{1}{k} \sum_{j=1}^{k} \nabla Q(\theta, j).$$

Because $f(\theta)$ is a summation, its gradient is the sum of the gradients of each individual part. Therefore:

$$\nabla f(\theta) = \frac{1}{k} \sum_{j=1}^{k} \nabla Q(\theta, j).$$

The two expressions match, therefore:

$$\boxed{E[\nabla g(\theta)] = \nabla f(\theta).}$$

- **Solution for (c):**

The noise is defined as:

$$z = Q(\theta, i) - f(\theta).$$

Since $Q(\theta, i)$ and $g(\theta)$ are equivalent, we can rewrite this as:

$$z = g(\theta) - f(\theta).$$

Using the linearity of expectation:

$$E[z] = E[g(\theta)] - E[f(\theta)].$$

Because $f(\theta)$ is a constant:

$$E[f(\theta)] = f(\theta).$$

$E[g(\theta)] = f(\theta)$ as established in part (a). Substituting these results:

$$\boxed{E[z] = f(\theta) - f(\theta) = 0.}$$

Part 2: Comparing SGD and GD running times

Exercise 2

	SGD	**GD**
Computational Cost per Update	$O(1)$	$O(k)$
Number of Updates to Reach ϵ	$O\left(\frac{1}{\epsilon}\right)$	$O\left(\ln\left(\frac{1}{\epsilon}\right)\right)$
Total Cost	$O\left(\frac{1}{\epsilon}\right)$	$O\left(k\ln\left(\frac{1}{\epsilon}\right)\right)$

Table 1: Comparing SGD vs GD in terms of training precision.

Explanations:

1. **Computational Cost per Update for GD:** calculating $\nabla f(\theta)$ requires summing over k data points, which takes kc time, and the subtraction step requires constant time. Therefore, the total runtime is linear in k, giving $O(k)$.
2. **Number of Updates to Reach ϵ for SGD:** Since f is strongly convex and twice continuously differentiable, we can apply the result from Theorem 1.1.
3. **Number of Updates to Reach ϵ for GD:** For gradient descent, we use the result from Theorem 1.3.
4. **Total Cost for SGD:** The total cost is the computational cost per update multiplied by the number of updates. $O(1) \times O\left(\frac{1}{\epsilon}\right)$ simplifies to $O\left(\frac{1}{\epsilon}\right)$.
5. **Total Cost for GD:** Similarly, the total cost for gradient descent is $O(k) \times O\left(\ln\left(\frac{1}{\epsilon}\right)\right)$, which simplifies to $O\left(k\ln\left(\frac{1}{\epsilon}\right)\right)$.

Part 3: Comparing SGD and GD w.r.t test loss

Exercise 3

	SGD	**GD**
Computational Cost per Update	$O(1)$	$O(\rho^{-1/\gamma})$
Number of Updates to Reach ρ	$O\left(\frac{1}{\rho}\right)$	$O\left(\ln\left(\frac{1}{\rho}\right)\right)$
Total Cost	$O\left(\frac{1}{\rho}\right)$	$O\left(\rho^{-1/\gamma}\ln\left(\frac{1}{\rho}\right)\right)$

Table 2: Comparing SGD vs GD in terms of testing precision.

Explanations for table entries:

1. **Computational Cost per Update for GD:** In the first table, the cost per update is $O(k)$. The relationship $\rho = O(k^{-\gamma})$, can be rewritten as $k^{-\gamma} = O(\rho)$. By raising both sides to the power of $-1/\gamma$, we get $k = O(\rho^{-1/\gamma})$. Therefore, the computational cost per update for GD is $O(\rho^{-1/\gamma})$.

2. **Number of Updates to Reach ρ for SGD:** Substituting $\epsilon = c\rho$ into the result for SGD, the number of updates becomes $O\left(\frac{1}{c\rho}\right)$, which simplifies to $O\left(\frac{1}{\rho}\right)$, since c is a constant.

3. **Number of Updates to Reach ρ for GD:** Substituting $\epsilon = c\rho$ into the result for GD, the number of updates becomes $O(\ln(1/c\rho)) = O(\ln(1/\rho) - \ln(c))$. Since $\ln(c)$ is constant, this reduces to $O(\ln(1/\rho))$.

4. **Total Cost for SGD:** The total cost is the computational cost per update multiplied by the number of updates. For SGD, this gives $O(1) \times O\left(\frac{1}{\rho}\right) = O\left(\frac{1}{\rho}\right)$.

5. **Total Cost for GD:** Following same logic as above, the total cost for GD is $O(\rho^{-1/\gamma}) \times O(\ln(1/\rho))$, which simplifies to $O(\rho^{-1/\gamma}\ln(1/\rho))$.

2:

If $\gamma = 0.2$, then the computational cost of one update for GD is $O(\rho^{-5})$. This means that decreasing the test loss is very computationally expensive. For example, if we set $\hat{\rho} = 0.8\rho$, then:

$$\hat{\rho}^{-5} = (0.8)^{-5} \cdot \rho^{-5} = 3.05 \cdot \rho^{-5}.$$

In other words, reducing the test loss by 20% would make each update take over 3 times longer, and we would require 3 times more data.

On the other hand, the update cost and number of iterations for SGD are independent of γ, and the total cost is simply $O\left(\frac{1}{\rho}\right)$, so no crazy negative powers of ρ. If we set $\hat{\rho} = 0.8\rho$, then:

$$\frac{1}{\hat{\rho}} = \frac{1}{0.8\rho} = 1.25 \cdot \frac{1}{\rho}.$$

This means that reducing the test loss by 20% would only require training for 1.25 times longer. The difference between SGD and GD would only widen the more we want to reduce the test loss.

IMPLEMENTATION

ADAM Gradient Descent Algorithm

Exercise 4

1) The closed-form solution to least squares regression is:

$$\boxed{\boldsymbol{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}}$$

Python code for solving using 'numpy':

```
'''
Using numpy's lstsq() to get the closed-form least squares solution
'''
theta_pred = la.lstsq(A, y_data)[0]

print('Empirical theta: ', theta_pred.reshape(-1))

>>> Empirical theta:  [1.01117113 0.99879624 0.99707909 1.01475117 1.00365197 0.98530922
 1.00109998 0.99715709 0.99246624 0.99988042]
```

This is very close to the target value of all 1's.

2) Starting with the original loss function:

$$Q(\theta, j) = \left|\mathbf{x}_j^T\boldsymbol{\theta} - y_j\right|^\gamma$$

Because of chain rule, the gradient of the loss function is:

$$\nabla Q(\theta, j) = \gamma \cdot \left|\mathbf{x}_j^T\boldsymbol{\theta} - y_j\right|^{\gamma-1} \cdot \frac{d\left(\mathbf{x}_j^T\boldsymbol{\theta} - y_j\right)}{d\boldsymbol{\theta}}$$

The derivative term evaluates to:

$$\frac{d\left|\mathbf{x}_j^T\boldsymbol{\theta} - y_j\right|}{d\boldsymbol{\theta}} = \text{sgn}(\mathbf{x}_j^T\boldsymbol{\theta} - y_j) \cdot \mathbf{x}_j$$

Therefore, the final gradient is:

$$\boxed{\nabla Q(\theta, j) = \gamma \cdot \left|\mathbf{x}_j^T\boldsymbol{\theta} - y_j\right|^{\gamma-1} \cdot \text{sgn}(\mathbf{x}_j^T\boldsymbol{\theta} - y_j) \cdot \mathbf{x}_j}$$

python implementation:

```
def noisy_val_grad(theta_hat, data_, label_, deg_=2.):
gradient = np.zeros_like(theta_hat)
loss = 0

for i in range(data_.shape[0]):
    x_ = data_[i, :].reshape(-1,1)
    y_ = label_[i, 0]
    err = np.sum(x_ * theta_hat) - y_

    grad = deg_ * np.abs(err) ** (deg_ - 1) * np.sign(err) * x_
    l = np.abs(err) ** deg_

    loss += l / data_.shape[0]
    gradient += grad / data_.shape[0]

return loss, gradient
```

3) Initializing ADAM hyperparameters:

```
if method == 'adam':
    beta_1 = 0.9
    beta_2 = 0.999
    m = np.zeros((dim_theta, 1)) # TODO: Initialize parameters
    v = np.zeros((dim_theta, 1))
    epsilon = 1e-8
```

ADAM implementation:

```
m = beta_1 * m + (1-beta_1) * gradient
v = beta_2 * v + (1-beta_2) * (gradient * gradient)
m_hat = m / (1 - beta_1 ** (t+1))
v_hat = v / (1 - beta_2 ** (t+1))
theta_hat = theta_hat - lr * (m_hat/(np.sqrt(v_hat)+epsilon))
```

Final parameters after 1000 updates:

```
print(params["adam"].reshape(-1))

>>> [0.35388391 0.27269951 0.31367533 0.36023661 0.28976016 0.36528544
0.33467022 0.36753296 0.29897261 0.30564255]
```

4) This is a plot of the mean test losses after running ADAM and SGD for 1000 iterations, 15 replicates.

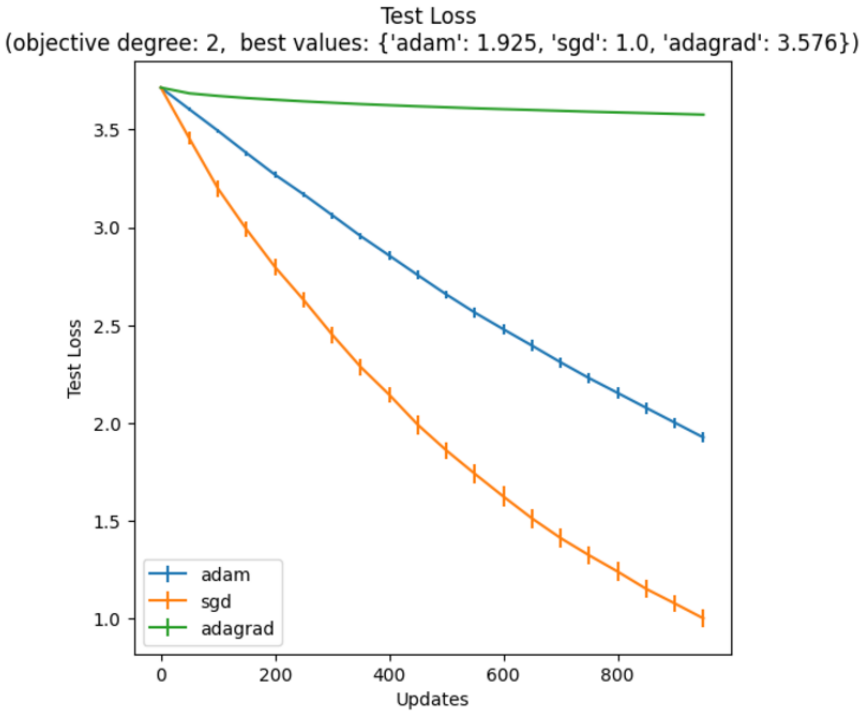

Figure 1: ADAM vs SGD test loss, γ=2

5) Plots for $\gamma = 0.4, 0.7, 1, 2, 3, 5$:

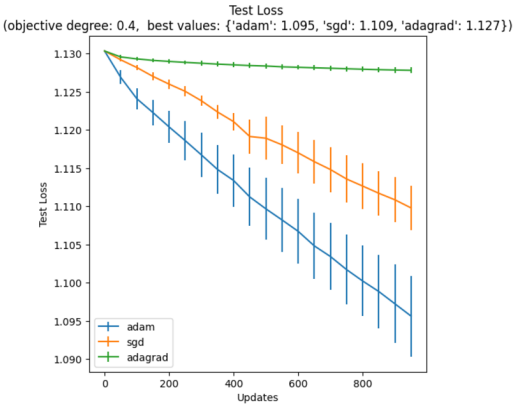

Figure 2: $\gamma = 0.4$

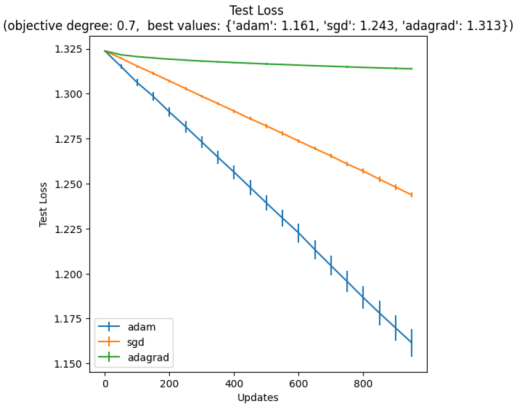

Figure 5: $\gamma = 0.7$

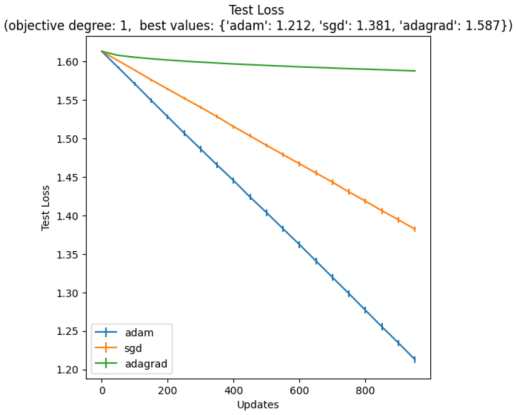

Figure 3: $\gamma = 1$

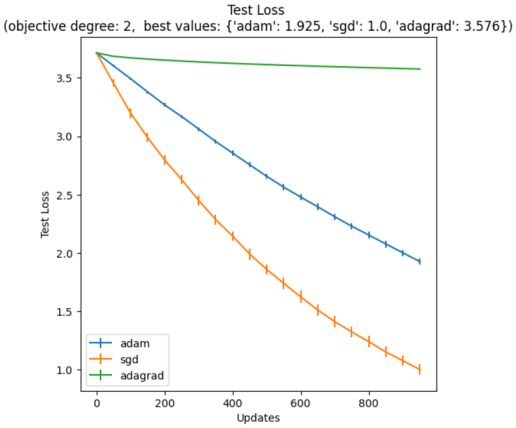

Figure 6: $\gamma = 2$

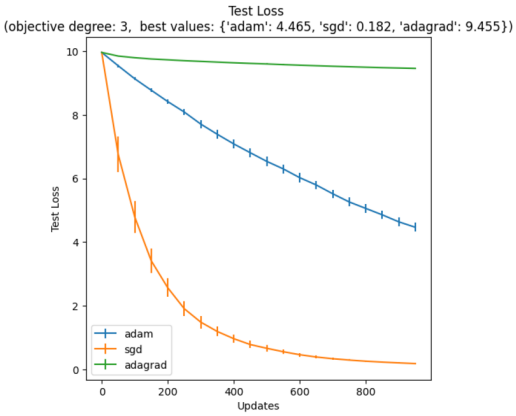

Figure 4: $\gamma = 3$

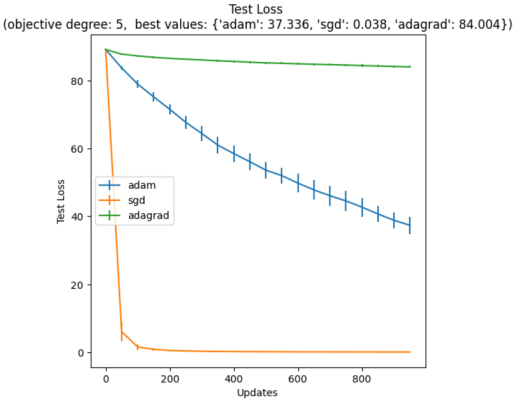

Figure 7: $\gamma = 5$

ADAM updates the parameters θ by dividing $\hat{m}$ by $(\sqrt{\hat{v}} + \epsilon)$. $\hat{m}$ is an exponentially weighted average of $\nabla(Q(\theta, j))$, while $\hat{v}$ is an exponentially weighted average of the squared magnitude of $\nabla(Q(\theta, j))$. Therefore, by dividing by $(\sqrt{\hat{v}} + \epsilon)$, we are basically dividing by the magnitude of $\nabla(Q(\theta, j))$ for each parameter. This means that ADAM can make sizable steps even when $\nabla(Q(\theta, j))$ is small. **When γ takes on small values such as 0.4, 0.7, and 1, this scaling behavior allows it to progress faster than SGD.**

On the other hand, when γ takes on larger values like 2, 3, and 5, $\nabla(Q(\theta, j))$ has a much greater magnitude. **SGD takes advantage of this large $\nabla(Q(\theta, j))$ to take large steps, but the ADAM algorithm prevents these large steps by dividing by the magnitude of $\nabla(Q(\theta, j))$.** This is why SGD outperforms ADAM at larger values of γ.

Classifying Handwritten Digits with Neural Networks

Exercise 5

1. (a) It is the same as GD when the batch size is 60000 because the minibatch would just be the entire dataset. So we would be making an update after computing the gradient on the entire dataset, which is what GD does.

   (b) The loss decreases rapidly at first but then slows down after 5 epochs. Our accuracy at the end is 92.8%, which is good but nowhere near human-level performance. The loss after the final epoch is approximately 0.261.

   (c) To improve the accuracy, we could a) train for more epochs and b) reduce the batch size so we can perform more updates with less computational cost.

2. (a) The final training loss is significantly less than when we did gradient descent (0.261 to 0.180). The accuracy is 94.7% now.

   (b) The steps slowed down towards the end because the gradient's magnitude was small. Using an optimizer like ADAM would allow us to take significant steps even when the gradient is small, allowing for even better performance.

3. (a) The final training loss is even less than SGD (0.180 to 0.0693). The accuracy is 97.0%.

   (b) ADAM converges faster than SGD for the reasons discussed in 2b; by dividing by the magnitude of the gradient for each parameter, we allow the learning rate to "increase" as the gradient gets smaller, so the optimization doesn't slow down as much when performance improves.