STA4001 Project Report

江敬鑫 117020119

Answer

1. For the sampling part, please check 1.1. Sample State Space and for the plotting part, please check 1.4. Plot Value Function
2. For the computing state space part, please check 1.5. Limitation
   For the modling part, please check 2. Neural Network, and 2.1.1 Fit the Model, 2.2.1 Fit the Model, 2.3.1 Fit the Model
   a) Please check 2.1.4 Predict, 2.2.4 Predict, 2.3.4 Predict
   b) Please check 2.1.3. Test the Model, 2.2.3. Test the Model, 2.3.3. Test the Model
   c) Please check 2.1.2. Mean Squared Error, 2.2.2. Mean Squared Error, 2.3.2. Mean Squared Error

---

In [398]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from math import sqrt, log

# neural network package
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
```

0. Notations

0.1. DTMC

$X_t = (X_{t,1}, X_{t,2}, \dots, X_{t,L-1}, X_{t,L})$ denotes the ending state, where $X_{t,\ell}$ is the amount of inventory with a remaining lifetime less than or equal to ℓ

$X_{t+1} = (X_{t+1,1}, X_{t+1,2}, \dots, X_{t+1,L-1}, X_{t+1,L}) = ((X_{t,2} - D)^+, (X_{t,3} - D)^+, \dots, (X_{t,L} - D)^+, (Q - X_{t,1} \vee D)) = \mathbf{f}(X_t)$ where $D \sim Poisson(\lambda)$

0.2. Parameters

In [126]:
```
Q = 90 # inventory level = 90
L = 6 # lifetime = 7 periods ==> L = len(X) = 6
beta = 0.8 # discount factor
rate = 17 # demand follows Poisson(rate)
Cp = 1 # selling price
Cv = 0.4# variable cost
Cd = 0.1 # disposal cost
h = 0.1 # holding cost
```

1. Monte Carlo Simulation

1.1. Sample State Space

`sample(K)` returns a set S consists of K distinct initial states: $S = \{X_0^1, X_0^2, \dots, X_0^K\}$

In [127]:
```
def sample(K): # K is the sample size
    X = np.zeros((1001, L)) # X represents every 1000 simulations
    S = set() # use set S to store distinct state spaces
    while (len(S) < K): # keep sampling while the sample size is not achieved
        for i in range(1, 10**3+1):
            D = np.random.poisson(rate) # D is demand following Poisson Dist.
            # update state
            X[i, -1] = Q-max(X[i-1, 0], D)
            for j in range(1,L):
                X[i, j-1] = max(X[i-1, j]-max(X[i-1, 0], D), 0)
        S.add(tuple(X[-1, :]))
        X[0,:] = X[-1, :] # set the 1000th state as initial state for next interation
    return S
```

To give an illustration of the sampling process (the first 5 steps):

In [366]:
```
X = np.zeros((5, L))
print('X0:\t', X[0, :], end='') # print initial state
for i in range(1, 5):
    D = np.random.poisson(rate)
    print('\tD=', D) # print the demand
    X[i, -1] = Q - max(X[i-1, 0], D)
    for j in range(1, L):
        X[i,j-1] = max(X[i-1, j] - max(X[i-1, 0], D), 0)
    print('X', i, ':\t', X[i, :], sep='', end='') # print the ending state
```

```
X0:      [0. 0. 0. 0. 0. 0.]    D= 14
X1:     [ 0.  0.  0.  0.  0. 76.]       D= 16
X2:     [ 0.  0.  0.  0. 60. 74.]       D= 14
X3:     [ 0.  0.  0. 46. 60. 76.]       D= 21
X4:     [ 0.  0. 25. 39. 55. 69.]
```

1.2. Monte Carlo

1.2.1. Tolerence (choice of T)

We use $\frac{1}{N}\sum_{n=1}^{N}\sum_{t=0}^{T}\beta^n g_x(X_{nt})$ to estimate $v(x) = \mathbb{E}\left[\sum_{t=0}^{\infty}\beta^t g(X_t)|X_0 = x\right]$
Since T is a finite number, the profit in periods $t > T$: $\sum_{t=T}^{\infty}\beta^t g(X_t)$ is neglected

we choose T such that the discounted total profit after T is sufficiently small
i.e. $\sum_{t=T}^{\infty}\beta^t g(X_t) \le \sum_{t=T}^{\infty}\beta^t Qc_p = \frac{\beta^T Qc_p}{1-\beta} < 10^{-d} \Rightarrow T > \log_\beta\frac{10^{-d}(1-\beta)}{Qc_p}$, for simplicity, set $T = \left[\left(\log_\beta\frac{10^{-d}(1-\beta)}{Qc_p}\right)//10 \times 10\right]$

In this case, we want the accuracy be 10^{-2}, set $T = \left[\left(\log_{0.8}\frac{10^{-2}(1-0.8)}{90}\right)//10 \times 10\right] = 50$

In [128]:
```
digit = 2 # significant digit
TOL = 10**(-digit) # tolerence = 0.01
T = int((log(TOL*(1-beta)/(Q*Cp), beta)//10+1)*10) # T = 50
T
```

Out[128]: `50`

1.2.2. Profit

The accounting method is on cash flow basis:
The profit at period ℓ is $profit_\ell = \beta^\ell(-c_v \times order + c_p \times sold - c_d \times disposed - h \times leftover)$
Profit is computed from: 1. variable cost of new arrivals 2. sales revenue of this period 3. disposed cost of expired units 4.holding cost of leftovers

To give an illustration, suppose the ending inventory last period is $X_{t-1} = (10, 20, 30, 40, 50, 80)$

1. At the beginning of period t, 10 brand-new product with remaining lifetime 7 arrives and incurs the variable cost $10 \times \$0.4 = \$4$;
2. Suppose the demand is 5, the sales revenue is $5 \times \$1 = \$5$
3. The remaining 5 units of product with lifetime 1 is disposed and incurs disposed cost $5 \times \$0.1 = \$0.5$
4. $X_t = (10, 20, 30, 40, 70, 80)$, at the end of the period, 80 units of leftover incurs holding cost $80 \times \$0.1 = \$8$

The profit of this period is $-\$4 + \$5 - \$0.5 - \$8 = -\$7.5$

1.2.3. Confidence Interval

Note that $\bar{X} = \frac{\sum_{i=0}^{N} X_i}{N}$, $\hat{\sigma}_X = \sqrt{\frac{\sum_{i=0}^{N}(X_i - \bar{X})^2}{N-1}} = \sqrt{\frac{\sum_{i=0}^{N} X_i^2 - N\bar{X}^2}{N-1}}$, thus we can generate the mean and standard deviation in real time.

`MonteCarlo(N,X0)` returns an array: $[\ \hat{v}(X_0) \quad \hat{v}(X_0) - \epsilon \quad \hat{v}(X_0) + \epsilon\ ]$ where $\epsilon = \frac{1.96\hat{\sigma}_{v(X_0)}}{\sqrt{N}}$

In [359]:
```
def MonteCarlo(N, X0):
    mean = 0
    std = 0
    for i in range(N): # N episodes
        profit = 0 # initialize profit = 0 for each episode
        X = np.zeros((T, L))
        X[0, :] = X0 # initialized the initial state
        for t in range(1, T):
            D = np.random.poisson(rate) # D is demand following Poisson Dist.
            disposed = max(X[t-1, 0]-D, 0) # unsold items with remaining life time = 0 is disposed
            order = Q - X[t-1, -1] # order up to the preset inventory level
            sold = min(Q, D) # at most Q units can be sold
            leftover = Q-D-disposed # leftover = Q - demand - disposed
            profit += beta**(t-1)*(-Cd*disposed-Cv*order+Cp*sold-h*leftover) # calculate the profit
            # update state
            X[t, -1] = leftover
            for j in range(1, L):
                X[t, j-1] = max(X[t-1, j]-D-disposed, 0)
        mean += profit
        std += profit**2
    mean = mean / N # compute the mean
    std = sqrt((std-N*mean**2)/(N-1)) # compute the standard error
    epsilon = 1.96*std/sqrt(N) # compute half length of confidence interval
    return np.array([mean, mean-epsilon, mean+epsilon]) # return the mean and 95% confidence interval
```

To give an illustration:

In [360]:
```
MonteCarlo(10**3, np.array([10, 20, 30, 40, 50, 80]))
```

Out[360]: `array([17.11556738, 16.78356754, 17.44756721])`

1.3. Create Dataset

`createDataset(K,N)` returns a matrix, with each row being the initial state and the corresponding mean and confidence interval

$$\begin{bmatrix} X_{0,1}^1 & X_{0,2}^1 & \cdots & X_{0,L}^1 & \hat{v}(X_0^1) & \hat{v}(X_0^1) - \epsilon_1 & \hat{v}(X_0^1) + \epsilon_1 \\ X_{0,1}^2 & X_{0,2}^2 & \cdots & X_{0,L}^2 & \hat{v}(X_0^2) & \hat{v}(X_0^2) - \epsilon_2 & \hat{v}(X_0^2) + \epsilon_2 \\ \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots \\ X_{0,1}^K & X_{0,2}^K & \cdots & X_{0,L}^K & \hat{v}(X_0^K) & \hat{v}(X_0^K) - \epsilon_K & \hat{v}(X_0^K) + \epsilon_K \end{bmatrix}$$

In [368]:
```
def createDataset(K, N):
    S = sample(K) # generates a sample of K states
    dataset = np.zeros((K, L+3)) # store each state and the corresponding simulation results in dataset
    index = 0
    for i in S:
        dataset[index, :] = np.hstack((np.array(i), MonteCarlo(N, np.array(i))))
        index += 1
    return dataset
```

In [20]:
```
train_20_2 = createDataset(20,10*2) # generate dataset for K=20, N=100
train_200_2 = createDataset(200,10*2) # generate dataset for K=200, N=100
train_2000_2= createDataset(2000,10*2) # generate dataset for K=2000, N=100
train_20_3 = createDataset(20,10*3) # generate dataset for K=20, N=1000
train_200_3 = createDataset(200,10*3) # generate dataset for K=200, N=1000
train_2000_3 = createDataset(2000,10*3) # generate dataset for K=2000, N=1000
test_50 = createDataset(50,10*3) # generate testset for K=50, N=1000
```

The shape the datasets generated:

In [337]:
```
print(train_20_2.shape,train_200_2.shape,train_2000_2.shape,train_20_3.shape,train_200_3.shape,train_2000_3.s
hape,test_50.shape,sep='  ')
```

```
(20, 9)  (200, 9)  (2000, 9)  (20, 9)  (200, 9)  (2000, 9)  (50, 9)
```

Save the data as csv file:

In [342]:
```
# convert the result to pd.DataFrame
df_20_2 = pd.DataFrame(train_20_2)
df_200_2 = pd.DataFrame(train_200_2)
df_2000_2 = pd.DataFrame(train_2000_2)
df_20_3 = pd.DataFrame(train_20_3)
df_200_3 = pd.DataFrame(train_200_3)
df_2000_3 = pd.DataFrame(train_2000_3)
df_50 = pd.DataFrame(test_50)

# save as csv file
df_20_2.to_csv('train_20_2.csv')
df_200_2.to_csv('train_200_2.csv')
df_2000_2.to_csv('train_2000_2.csv')
df_20_3.to_csv('train_20_3.csv')
df_200_3.to_csv('train_200_3.csv')
df_2000_3.to_csv('train_2000_3.csv')
df_50.to_csv('test_50.csv')
```

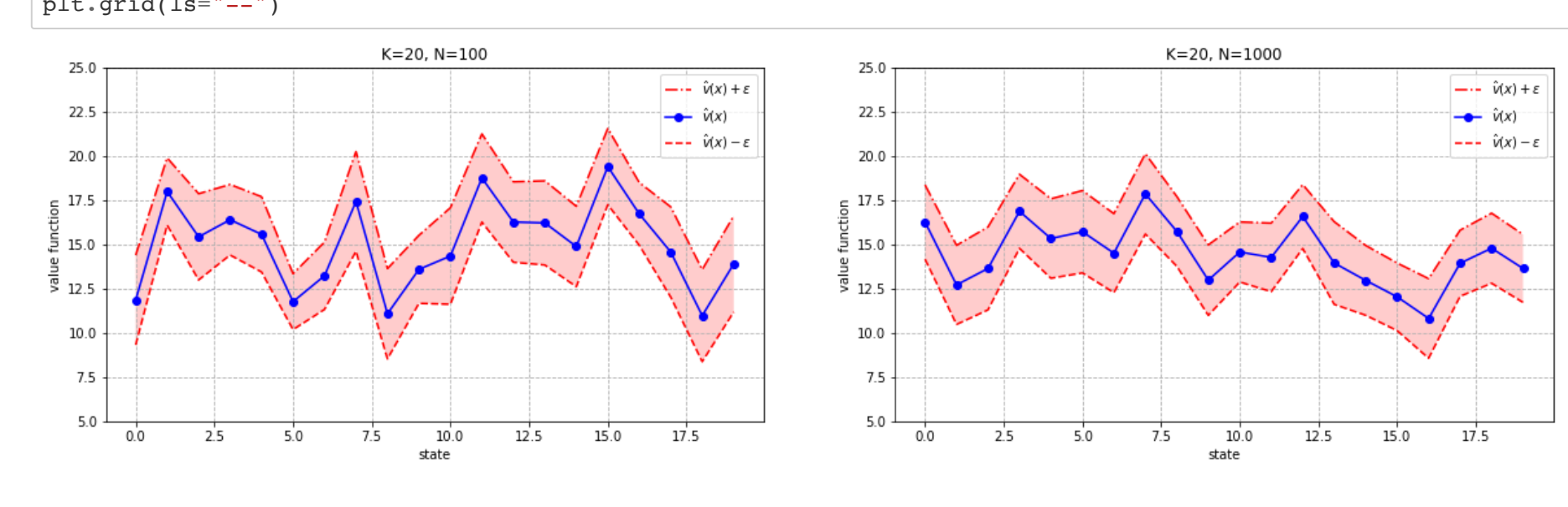
1.4. Plot Value Function

1.4.1. K = 20

In [403]:
```
fig_20 = plt.figure(figsize=(20, 5)) # K = 20

# plot N = 100
sub_1= plt.subplot(1, 2, 1)
sub_1.set_ylim(5, 25)
# the right endpoint of confidence inteval
plt.plot(train_20_2[:, -1], color='red', ls="-.", label='$\hat v(x)+\epsilon$')
# the point estimate of value function
plt.plot(train_20_2[:, -3], color='blue', marker='o', label='$\hat v(x)$')
# the left endpoint of confidence interval
plt.plot(train_20_2[:, -2], color='red', ls="--", label='$\hat v(x)-\epsilon$')
# color the confidence interval red
plt.fill_between(x=np.arange(0, 20, 1), y1=train_20_2[:, -2], y2=train_20_2[:, -1], facecolor="red", alpha=0.
2)
plt.title('K=20, N=100')
plt.xlabel('state')
plt.ylabel('value function')
plt.legend(loc=1)
plt.grid(ls="--")

# plot N = 1000
sub_2=plt.subplot(1, 2, 2)
sub_2.set_ylim(5, 25)
# the right endpoint of confidence inteval
plt.plot(train_20_3[:, -1], color='red', ls="-.", label='$\hat v(x)+\epsilon$')
# the point estimate of value function
plt.plot(train_20_3[:, -3], color='blue', marker = 'o', label='$\hat v(x)$')
# the left endpoint of confidence interval
plt.plot(train_20_3[:, -2], color='red', ls="--", label='$\hat v(x)-\epsilon$')
# color the confidence interval red
plt.fill_between(x=np.arange(0, 20, 1), y1=train_20_3[:, -2], y2=train_20_3[:, -1], facecolor="red", alpha=0.
2)
plt.title('K=20, N=1000')
plt.xlabel('state')
plt.ylabel('value function')
plt.legend(loc=1)
plt.grid(ls="--")
```

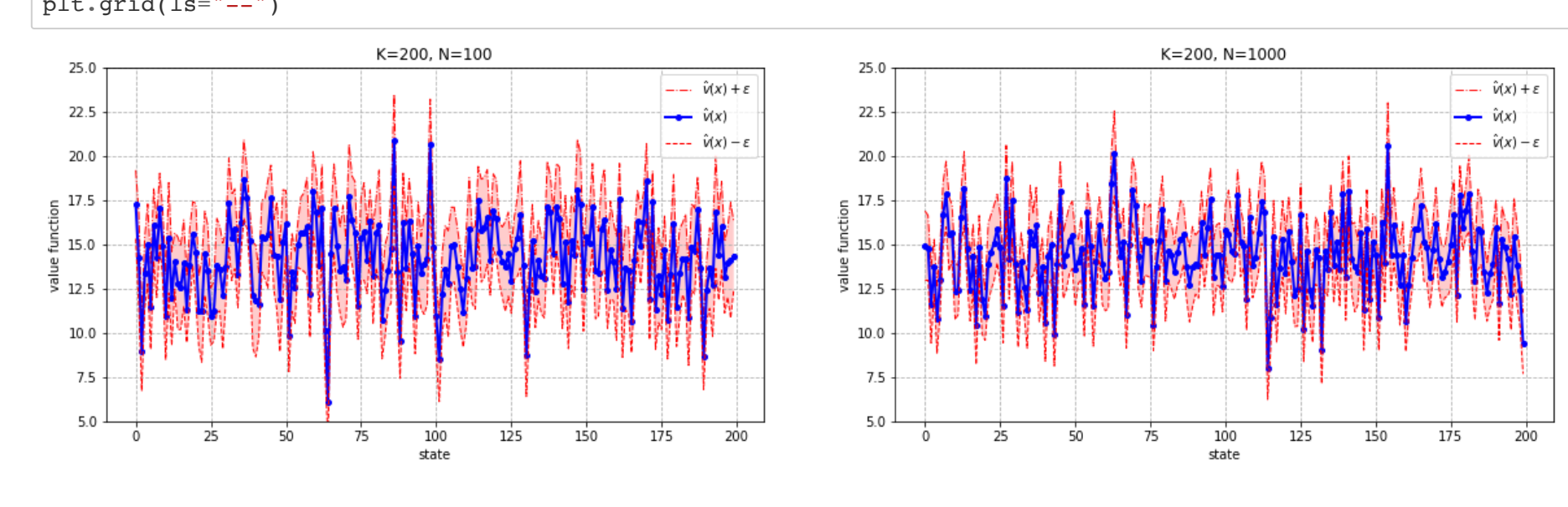


1.4.2. K = 200

In [404]:
```
fig_200 = plt.figure(figsize = (20, 5)) # K = 200

# plot N = 100
sub_1= plt.subplot(1, 2, 1)
sub_1.set_ylim(5, 25)
# the right endpoint of confidence inteval
plt.plot(train_200_2[:, -1], color='red', ls="-.", label='$\hat v(x)+\epsilon$', lw=1)
# the point estimate of value function
plt.plot(train_200_2[:, -3], color='blue', marker = 'o', label='$\hat v(x)$', lw=2, ms=4)
# the left endpoint of confidence interval
plt.plot(train_200_2[:, -2], color='red', ls="--", label='$\hat v(x)-\epsilon$', lw=1)
# color the confidence interval red
plt.fill_between(x=np.arange(0, 200, 1), y1=train_200_2[:, -2], y2=train_200_2[:, -1], facecolor="red", alpha
=0.2)
plt.title('K=200, N=100')
plt.xlabel('state')
plt.ylabel('value function')
plt.legend(loc=1)
plt.grid(ls="--")

# plot N = 1000
sub_2=plt.subplot(1, 2, 2)
sub_2.set_ylim(5, 25)
# the right endpoint of confidence inteval
plt.plot(train_200_3[:, -1], color='red',ls="-.",label='$\hat v(x)+\epsilon$', lw=1)
# the point estimate of value function
plt.plot(train_200_3[:, -3], color='blue', marker = 'o', label='$\hat v(x)$', lw=2, ms=4)
# the left endpoint of confidence interval
plt.plot(train_200_3[:, -2], color='red', ls="--",label='$\hat v(x)-\epsilon$', lw=1)
# color the confidence interval red
plt.fill_between(x=np.arange(0, 200, 1), y1=train_200_3[:, -2], y2=train_200_3[:, -1], facecolor="red", alpha
=0.2)
plt.title('K=200, N=1000')
plt.xlabel('state')
plt.ylabel('value function')
plt.legend(loc=1)
plt.grid(ls="--")
```

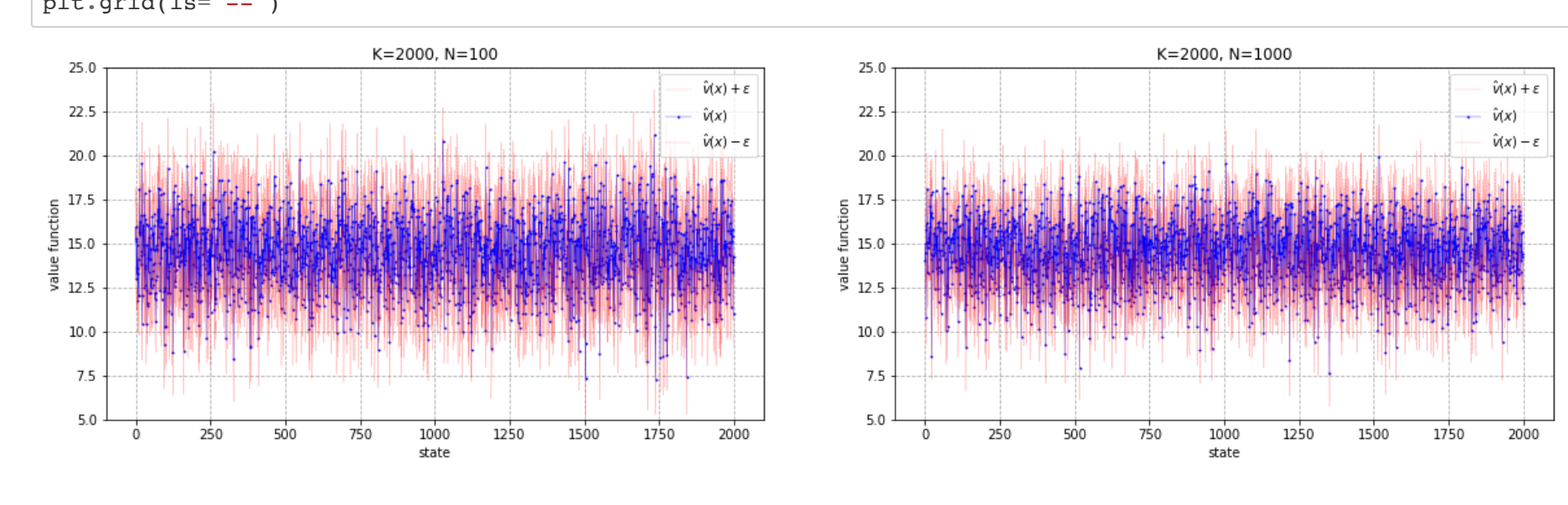


1.4.2. K = 2000

In [443]:
```
fig_2000 = plt.figure(figsize = (20, 5)) # K = 2000

# plot N = 100
sub_1= plt.subplot(1, 2, 1)
sub_1.set_ylim(5, 25)
# the right endpoint of confidence inteval
plt.plot(train_2000_2[:, -1], color='red', ls="-.", label='$\hat v(x)+\epsilon$', lw=0.15)
# the point estimate of value function
plt.plot(train_2000_2[:, -3], color='blue', marker = 'o', label='$\hat v(x)$', lw=0.3, ms=1)
# the left endpoint of confidence interval
plt.plot(train_2000_2[:, -2], color='red', ls="--", label='$\hat v(x)-\epsilon$', lw=0.15)
# color the confidence interval red
plt.fill_between(x=np.arange(0, 2000, 1),y1=train_2000_2[:, -2], y2=train_2000_2[:, -1], facecolor="red", alp
ha=0.2)
plt.title('K=2000, N=100')
plt.xlabel('state')
plt.ylabel('value function')
plt.legend(loc=1)
plt.grid(ls="--")

# plot N = 1000
sub_2=plt.subplot(1, 2, 2)
sub_2.set_ylim(5, 25)
# the right endpoint of confidence inteval
plt.plot(train_2000_3[:, -1], color = 'red', ls="-.", label='$\hat v(x)+\epsilon$', lw=0.15)
# the point estimate of value function
plt.plot(train_2000_3[:, -3], color = 'blue', marker = 'o', label='$\hat v(x)$', lw=0.3, ms=1)
# the left endpoint of confidence interval
plt.plot(train_2000_3[:, -2], color = 'red', ls="--", label='$\hat v(x)-\epsilon$', lw=0.15)
# color the confidence interval red
plt.fill_between(x=np.arange(0, 2000, 1), y1=train_2000_3[:, -2], y2=train_2000_3[:, -1], facecolor="red", al
pha=0.2)
plt.title('K=2000, N=1000')
plt.xlabel('state')
plt.ylabel('value function')
plt.legend(loc=1)
plt.grid(ls="--")
```



1.5. Limitation

Monte Carlo Stimulation is time-cosuming and computationally-demanding. As the state space is large, it is also space consuming to store all the resuls.
To see this:

$S = \{X_t = (X_{t,1}, X_{t,2}, \dots, X_{t,L-1}, X_{t,L}) : X_{t,1} \le X_{t,2} \le \cdots \le X_{t,L-1} \le X_{t,L} \le Q\}$. Cardinality $|S| \le Q^{L-1}$

When $Q = 90$, $L = 7$, $|S| \le 90^6 = 5.31441 \times 10^{11}$

To precisely compute $|S|$, one possible solution is to draw a tree diagram with the help of recursion. However, simple recursion won't work when the height of the tree is very large, which will reuslt in a very deep recursion path and lead to the breakdown of the program. The following code gives an implementation using stack to exhaust all the possible states and returns the cardinality of the state space:

In [394]:
```
def generateStateSpace(Q, L):
    S = set() # the state space
    X = [0 for i in range(L)] # initial state
    TX = tuple(X) # convert X into hashable type to check uniqueness
    Stack = [TX] # Stack record the parent node in the tree
    S.add(TX) # add the initial state into state space
    D = [0]
    while (len(D) > 0): # while the recursion does not end
        d = D[-1] # enumerate d from X[0] to Q
        # update state given demand d
        X[-1] = Q-max(Stack[-1][0], d)
        for i in range(1, L):
            X[i-1] = max(Stack[-1][i]-max(Stack[-1][0], d), 0)
        TX = tuple(X)
        if TX not in S: # if the new state not in state space
            S.add(TX) # add the new state in
            # increase the recursion depth
            Stack.append(TX)
            D.append(X[0])
        elif (D[-1] < Q): # from one child note to another
            D[-1] += 1
        else:
            # cut the leaves and go back to the parent node
            Stack.pop()
            D.pop()
            if (len(D) == 0): # when the recursion terminates, break
                break
            D[-1] += 1
    return(len(S))
```

To give a illustration, when $Q = 20$ and $L = 3$:

In [405]:
```
generateStateSpace(20, 3)
```

Out[405]: `1771`

You won't want to run `generateStateSpace(90,6)` because it takes forever to get the result.

---

2. Neuron Network

In [309]:
```
class neuralNetwork():
    def __init__(self, input_dimension):
        # define the keras model
        self.model = Sequential()
        self.model.add(Dense(5, input_dim=input_dimension,
                        activation='relu'))
        self.model.add(Dense(5, activation='relu'))
        self.model.add(Dense(1, activation='linear'))
        # compile the keras model
        self.model.compile(loss='mean_squared_error', optimizer='adam')

# scale the data
class Scaler:
    # save mean and variance of x, y sets
    def __init__(self, x, y):
        self.x_mean = np.mean(x, axis=0)
        self.y_mean = np.mean(y, axis=0)
        self.x_std = np.std(x, axis=0)
        self.y_std = np.std(y, axis=0)

    def get_x(self):
        # return saved mean and variance of x
        return self.x_std, self.x_mean

    def get_y(self):
        # return saved mean and variance of y
        return self.y_std, self.y_mean
```

2.1. K=20

2.1.1. Fit the Model

In [423]:
```
x_train = train_20_3[:, :L]
y_train = train_20_3[:, L]
x_test = test_50[:, :L]
y_test = test_50[:, L]

# normalize the train and test data
normalizer = Scaler(x_train, y_train)
std_x, mean_x = normalizer.get_x()
x_train_norm = (x_train-mean_x)/std_x
x_test_norm = (x_test-mean_x)/std_x
std_y, mean_y = normalizer.get_y()
y_train_norm = (y_train-mean_y)/std_y

NN1 = neuralNetwork(input_dimension=L)

# fit the data
# set verbose = 0 to silence the processing output
NN1.model.fit(x_train_norm, y_train_norm, epochs=100, batch_size=8, verbose=0)

# generate predicted value function
y_from_nn1_norm = NN1.model.predict(x_test_norm)
y_from_nn1 = y_from_nn1_norm*std_y+mean_y
```

2.1.2. Mean Squared Error

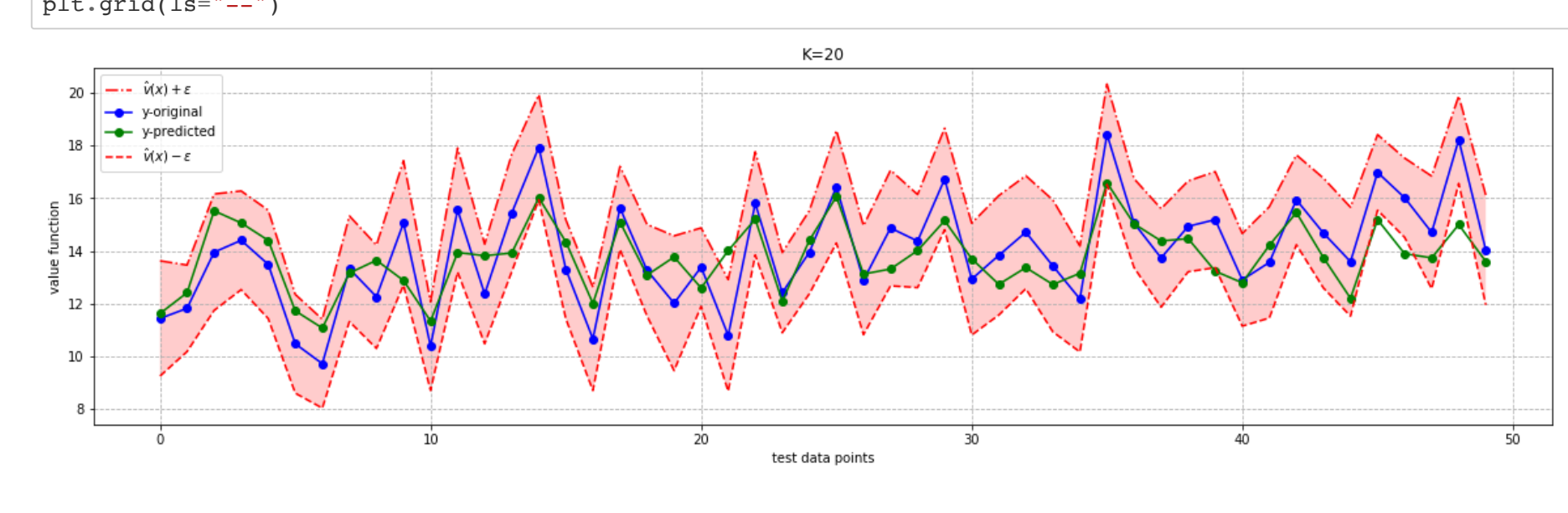
In [424]:
```
mse = mean_squared_error(y_test, y_from_nn1)
print('Mean squared error: ', mse)
```

```
Mean squared error:  1.6899557083422971
```

2.1.3. Test the Model

In [439]:
```
fig_20N = plt.figure(figsize = (20,5)) # K = 20

plt.plot(test_50[:, -1], color = 'red', ls="-.", label='$\hat v(x)+\epsilon$')
plt.plot(test_50[:, -3], color = 'blue',marker = 'o', label='y-original')
plt.plot(y_from_nn1, color = 'green',marker = 'o', label='y-predicted')
plt.plot(test_50[:, -2], color = 'red',ls="--",label='$\hat v(x)-\epsilon$')
plt.fill_between(x=np.arange(0, 50, 1), y1=test_50[:, -2], y2=test_50[:, -1], facecolor="red", alpha=0.2)
plt.title('K=20')
plt.xlabel('test data points')
plt.ylabel('value function')
plt.legend(loc=2)
plt.grid(ls="--")
```



2.1.4. Predict

In [426]:
```
print('Initial State:\t', (np.array([10, 20, 30, 40, 50, 80])))
print('Neural Network:\t', (NN1.model.predict(((np.array([[10, 20, 30, 40, 50, 80]])-mean_x)/std_x))*std_y+me
an_y)[0][0])
MC = MonteCarlo(10**3, np.array([10, 20, 30, 40, 50, 80]))
print('MonteCarlo:\t', MC[0])
print('MonteCarlo CI:\t[', MC[1], ', ', MC[2],']', sep='')
```

```
Initial State:   [10 20 30 40 50 80]
Neural Network:  16.378511
MonteCarlo:      17.49736435397056
MonteCarlo CI:  [17.156497565305322, 17.8382311426358]
```

2.2. K=200

2.2.1. Fit the Model

In [322]:
```
x_train = train_200_3[:, :L]
y_train = train_200_3[:, L]
x_test = test_50[:, :L]
y_test = test_50[:, L]

# normalize the train and test data
normalizer = Scaler(x_train, y_train)
std_x, mean_x = normalizer.get_x()
x_train_norm = (x_train - mean_x) / std_x
x_test_norm = (x_test - mean_x) / std_x
std_y, mean_y = normalizer.get_y()
y_train_norm = (y_train - mean_y) / std_y

NN2 = neuralNetwork(input_dimension=L)

# fit the data
# set verbose = 0 to silence the processing output
NN2.model.fit(x_train_norm, y_train_norm, epochs=100, batch_size=8, verbose=0)

# generate predicted value function
y_from_nn2_norm = NN2.model.predict(x_test_norm)
y_from_nn2 = y_from_nn2_norm * std_y + mean_y
```

2.2.2. Mean Squared Error

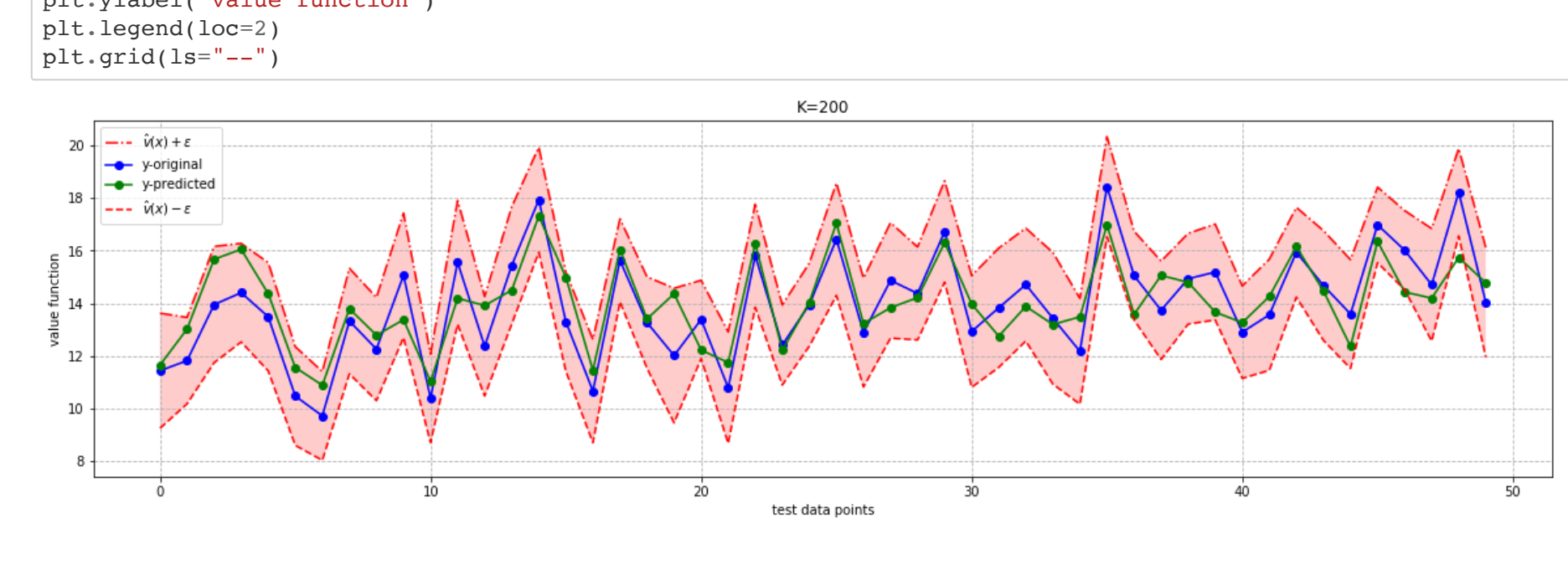
In [323]:
```
mse = mean_squared_error(y_test, y_from_nn2)
print('Mean squared error: ', mse)
```

```
Mean squared error:  1.1659013233236868
```

2.2.3. Test the Model

In [437]:
```
fig_200N = plt.figure(figsize = (20,5)) # K = 200

plt.plot(test_50[:,-1], color = 'red', ls="-.",label='$\hat v(x)+\epsilon$')
plt.plot(test_50[:,-3], color = 'blue',marker = 'o', label='y-original')
plt.plot(y_from_nn2, color = 'green',marker = 'o', label='y-predicted')
plt.plot(test_50[:,-2], color = 'red',ls="--",label='$\hat v(x)-\epsilon$')
plt.fill_between(x=np.arange(0,50,1),y1=test_50[:, -2], y2=test_50[:, -1], facecolor="red", alpha=0.2)
plt.title('K=200')
plt.xlabel('test data points')
plt.ylabel('value function')
plt.legend(loc=2)
plt.grid(ls="--")
```



2.2.4. Predict

In [427]:
```
print('Initial State:\t', (np.array([10, 20, 30, 40, 50, 80])))
print('Neural Network:\t', (NN2.model.predict(((np.array([[10, 20, 30, 40, 50, 80]])-mean_x)/std_x))* std_y+m
ean_y)[0][0])
MC = MonteCarlo(10**3, np.array([10, 20, 30, 40, 50, 80]))
print('MonteCarlo:\t', MC[0])
print('MonteCarlo CI:\t[', MC[1], ', ', MC[2], ']',sep='')
```

```
Initial State:   [10 20 30 40 50 80]
Neural Network:  17.347094
MonteCarlo:      17.188584620132517
MonteCarlo CI:  [16.85684963332085, 17.520319606944184]
```

2.3. K=2000

2.3.1. Fit the Model

In [435]:
```
x_train = train_2000_3[:, :L]
y_train = train_2000_3[:, L]
x_test = test_50[:, :L]
y_test = test_50[:, L]

# normalize the train and test data
normalizer = Scaler(x_train, y_train)
std_x, mean_x = normalizer.get_x()
x_train_norm = (x_train - mean_x) / std_x
x_test_norm = (x_test - mean_x) / std_x
std_y, mean_y = normalizer.get_y()
y_train_norm = (y_train - mean_y) / std_y

NN3 = neuralNetwork(input_dimension=L)

# fit the data
# set verbose = 0 to silence the processing output
NN3.model.fit(x_train_norm, y_train_norm, epochs=100, batch_size=8, verbose=0)

# generate predicted value function
y_from_nn3_norm = NN3.model.predict(x_test_norm)
y_from_nn3 = y_from_nn3_norm * std_y + mean_y
```

2.3.2. Mean Squared Error

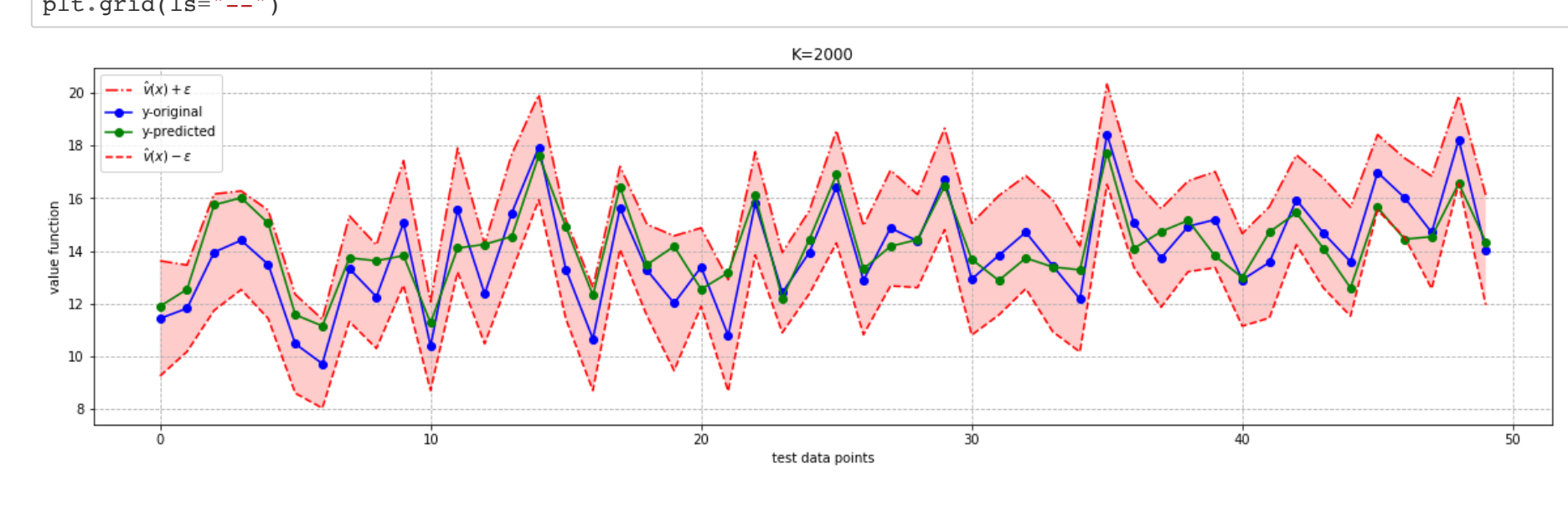
In [436]:
```
mse = mean_squared_error(y_test, y_from_nn3)
print('Mean squared error: ', mse)
```

```
Mean squared error:  1.1904237072409136
```

2.3.3. Test the Model

In [438]:
```
fig_2000N = plt.figure(figsize = (20, 5)) # K = 2000

plt.plot(test_50[:, -1], color = 'red', ls="-.", label='$\hat v(x)+\epsilon$')
plt.plot(test_50[:, -3], color = 'blue', marker = 'o', label='y-original')
plt.plot(y_from_nn3, color = 'green', marker = 'o', label='y-predicted')
plt.plot(test_50[:, -2], color = 'red', ls="--", label='$\hat v(x)-\epsilon$')
plt.fill_between(x=np.arange(0, 50, 1), y1=test_50[:, -2], y2=test_50[:, -1], facecolor="red", alpha=0.2)
plt.title('K=2000')
plt.xlabel('test data points')
plt.ylabel('value function')
plt.legend(loc=2)
plt.grid(ls="--")
```



2.3.4. Predict

In [441]:
```
print('Initial State:\t', (np.array([10, 20, 30, 40, 50, 80])))
print('Neural Network:\t', (NN3.model.predict(((np.array([[10, 20, 30, 40, 50, 80]])-mean_x)/std_x))* std_y +
mean_y)[0][0])
MC = MonteCarlo(10**3, np.array([10, 20, 30, 40, 50, 80]))
print('MonteCarlo:\t', MC[0])
print('MonteCarlo CI:\t[', MC[1], ', ', MC[2], ']', sep='')
```

```
Initial State:   [10 20 30 40 50 80]
Neural Network:  17.063206
MonteCarlo:      17.267245684548296
MonteCarlo CI:  [16.927039819185133, 17.60745154991146]
```

The End of Report