Simple Neural Network

December 9, 2020

[186]:
```
import numpy as np
import matplotlib.pyplot as plt
from numpy import random
```

1 Maths Derivation

[187]:
```
def sigmoid(z):
    a = np.exp(z)
    b = np.exp(-z)
    return (a - b)/(a + b)
```

Derivative of sigmoid

$$(e^x - e^{-x})' = e^x + e^{-x}$$

$$(e^x + e^{-x})' = e^x - e^{-x}$$

$$\phi'(x) = \frac{(e^x - e^{-x})'(e^x + e^{-x}) - (e^x + e^{-x})'(e^x - e^{-x})}{(e^x + e^{-x})^2} = \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} = \frac{4}{(e^x + e^{-x})^2}$$

[188]:
```
def sigmoid_d(z):
    return 4 / ((np.exp(z) + np.exp(-z))**2)
```

1.1 Compute the gradient

$$\nabla_w f_w(x) = \begin{bmatrix} \frac{\partial f_w(x)}{\partial w_1} \\ \frac{\partial f_w(x)}{\partial w_2} \\ \frac{\partial f_w(x)}{\partial w_3} \\ \frac{\partial f_w(x)}{\partial w_4} \\ \frac{\partial f_w(x)}{\partial w_5} \\ \frac{\partial f_w(x)}{\partial w_6} \\ \frac{\partial f_w(x)}{\partial w_7} \\ \frac{\partial f_w(x)}{\partial w_8} \\ \frac{\partial f_w(x)}{\partial w_9} \\ \frac{\partial f_w(x)}{\partial w_{10}} \\ \frac{\partial f_w(x)}{\partial w_{11}} \\ \frac{\partial f_w(x)}{\partial w_{12}} \\ \frac{\partial f_w(x)}{\partial w_{13}} \\ \frac{\partial f_w(x)}{\partial w_{14}} \\ \frac{\partial f_w(x)}{\partial w_{15}} \\ \frac{\partial f_w(x)}{\partial w_{16}} \end{bmatrix} = \begin{bmatrix} \phi(w_2x_1+w_3x_2+w_4x_3+w_5) \\ w_1\phi'(w_2x_1+w_3x_2+w_4x_3+w_5)x_1 \\ w_1\phi'(w_2x_1+w_3x_2+w_4x_3+w_5)x_2 \\ w_1\phi'(w_2x_1+w_3x_2+w_4x_3+w_5)x_3 \\ w_1\phi'(w_2x_1+w_3x_2+w_4x_3+w_5) \\ \phi(w_7x_1+w_8x_2+w_9x_3+w_{10}) \\ w_6\phi'(w_7x_1+w_8x_2+w_9x_3+w_{10})x_1 \\ w_6\phi'(w_7x_1+w_8x_2+w_9x_3+w_{10})x_2 \\ w_6\phi'(w_7x_1+w_8x_2+w_9x_3+w_{10})x_3 \\ w_6\phi'(w_7x_1+w_8x_2+w_9x_3+w_{10}) \\ \phi(w_{12}x_1+w_{13}x_2+w_{14}x_3+w_{15}) \\ w_{11}\phi'(w_{12}x_1+w_{13}x_2+w_{14}x_3+w_{15})x_1 \\ w_{11}\phi'(w_{12}x_1+w_{13}x_2+w_{14}x_3+w_{15})x_2 \\ w_{11}\phi'(w_{12}x_1+w_{13}x_2+w_{14}x_3+w_{15})x_3 \\ w_{11}\phi'(w_{12}x_1+w_{13}x_2+w_{14}x_3+w_{15}) \\ 1 \end{bmatrix}$$

1.2 Compute the Jacobian

$$Dr(w) = \begin{bmatrix} \frac{\partial r_1(w)}{\partial w_1} & \frac{\partial r_1(w)}{\partial w_2} & \dots & \frac{\partial r_1(w)}{\partial w_{16}} \\ \frac{\partial r_2(w)}{\partial w_1} & \frac{\partial r_2(w)}{\partial w_2} & \dots & \frac{\partial r_2(w)}{\partial w_{16}} \\ \dots & \dots & \dots & \dots \\ \frac{\partial r_N(w)}{\partial w_1} & \frac{\partial r_N(w)}{\partial w_2} & \dots & \frac{\partial r_N(w)}{\partial w_{16}} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_w(x^{(1)})}{\partial w_1} & \frac{\partial f_w(x^{(1)})}{\partial w_2} & \dots & \frac{\partial f_w(x^{(1)})}{\partial w_{16}} \\ \frac{\partial f_w(x^{(2)})}{\partial w_1} & \frac{\partial f_w(x^{(2)})}{\partial w_2} & \dots & \frac{\partial f_w(x^{(2)})}{\partial w_{16}} \\ \dots & \dots & \dots & \dots \\ \frac{\partial f_w(x^{(N)})}{\partial w_1} & \frac{\partial f_w(x^{(N)})}{\partial w_2} & \dots & \frac{\partial f_w(x^{(N)})}{\partial w_{16}} \end{bmatrix}$$

$$= \begin{bmatrix} \nabla_w f_w(x^1)^T \\ \nabla_w f_w(x^2)^T \\ \nabla_w f_w(x^3)^T \\ \dots \\ \nabla_w f_w(x^N)^T \end{bmatrix} = \begin{bmatrix} \nabla_w f_w(x^1) & \nabla_w f_w(x^2) & \dots & \nabla_w f_w(x^N) \end{bmatrix}^T$$

1.3 Derive the closed form for the minimizer

Need to find

$$\min_w \left\| r(w^k) + Dr(w^k)(w - w^k) \right\|^2 + \lambda \|w\|^2 + \lambda^k \left\| w - w^k \right\|^2$$

Have $A_1, A_2, A_3, b_1, b_2, b_3$ such that

$$\left\| r(w^k) + Dr(w^k)(w - w^k) \right\|^2 + \lambda \|w\|^2 + \lambda^k \left\| w - w^k \right\|^2 = \|A_1 w - b_1\|^2 + \|A_2 w - b_2\|^2 + \|A_3 w - b_3\|^2$$

$$A_1 = D_r(w^k), b_1 = Dr(w^k)w^k - r(w^k)$$

$$A_2 = I, b_2 = w^k$$

$$A_3 = I, b_3 = 0$$

$$\left\| r(w^k) + Dr(w^k)(w - w^k) \right\|^2 + \lambda \|w\|^2 + \lambda^k \left\| w - w^k \right\|^2 = \left\| \begin{bmatrix} A_1 \\ \sqrt{\lambda^k} A_2 \\ \sqrt{\lambda} A_3 \end{bmatrix} w - \begin{bmatrix} b_1 \\ \sqrt{\lambda^k} b_2 \\ \sqrt{\lambda} b_3 \end{bmatrix} \right\|^2$$

Therefore,

$$A = \begin{bmatrix} Dr(w^k) \\ \sqrt{\lambda^k} I \\ \sqrt{\lambda} I \end{bmatrix}, b = \begin{bmatrix} Dr(w^k)w^k - r(w^k) \\ \sqrt{\lambda^k} w^k \\ 0 \end{bmatrix}$$

To find the minimizer,

$$A^T A w = A^T b$$

$$\begin{bmatrix} Dr(w^k)^T & \sqrt{\lambda^k} I & \sqrt{\lambda} I \end{bmatrix} \begin{bmatrix} Dr(w^k) \\ \sqrt{\lambda^k} I \\ \sqrt{\lambda} I \end{bmatrix} w = \begin{bmatrix} Dr(w^k)^T & \sqrt{\lambda^k} I & \sqrt{\lambda} I \end{bmatrix} \begin{bmatrix} Dr(w^k)w^k - r(w^k) \\ \sqrt{\lambda^k} w^k \\ 0 \end{bmatrix}$$

$$\left(Dr(w^k)^T Dr(w^k) + (\lambda^k + \lambda) I \right) w = Dr(w^k)^T Dr(w^k) w^k - Dr(w^k)^T r(w^k) + \lambda^k w^k$$

$$w^* = \left(Dr(w^k)^T Dr(w^k) + (\lambda^k + \lambda) I \right)^{-1} \left(Dr(w^k)^T Dr(w^k) w^k - Dr(w^k)^T r(w^k) + \lambda^k w^k \right)$$

2 Neural Network Implementation

[189]:
```
# Simple Neural Network (Feed-forward)
def f(w, x):
    return w[0]*sigmoid(w[1]*x[0]+w[2]*x[1]+w[3]*x[2]+w[4]) + \
        w[5]*sigmoid(w[6]*x[0]+w[7]*x[1]+w[8]*x[2]+w[9]) + \
        w[10]*sigmoid(w[11]*x[0]+w[12]*x[1]+w[13]*x[2]+w[14]) + w[15]

# Error on each data point
def r(w, x, y):
    return f(w, x) - y

# Error on the entire dataset
def rn(w, X, Y):
    return np.array([r(w, x, y) for x,y in zip(X, Y)])

# Jacobian
def Dr(w, X):
    ret = np.zeros((N, 16))
    for i,x in enumerate(X):
        phi1 = sigmoid(w[1]*x[0]+w[2]*x[1]+w[3]*x[2]+w[4])
        phi1d = sigmoid_d(w[1]*x[0]+w[2]*x[1]+w[3]*x[2]+w[4])
        phi2 = sigmoid(w[6]*x[0]+w[7]*x[1]+w[8]*x[2]+w[9])
        phi2d = sigmoid_d(w[6]*x[0]+w[7]*x[1]+w[8]*x[2]+w[9])
        phi3 = sigmoid(w[11]*x[0]+w[12]*x[1]+w[13]*x[2]+w[14])
        phi3d = sigmoid_d(w[11]*x[0]+w[12]*x[1]+w[13]*x[2]+w[14])
        ret[i][0] = phi1
        ret[i][1] = w[0]*phi1d*x[0]
        ret[i][2] = w[0]*phi1d*x[1]
        ret[i][3] = w[0]*phi1d*x[2]
        ret[i][4] = w[0]*phi1d
        ret[i][5] = phi2
        ret[i][6] = w[5]*phi2d*x[0]
        ret[i][7] = w[5]*phi2d*x[1]
        ret[i][8] = w[5]*phi2d*x[2]
```

```
        ret[i][9] = w[5]*phi2d
        ret[i][10] = phi3
        ret[i][11] = w[10]*phi3d*x[0]
        ret[i][12] = w[10]*phi3d*x[1]
        ret[i][13] = w[10]*phi3d*x[2]
        ret[i][14] = w[10]*phi3d
        ret[i][15] = 1
    return ret

# Loss: sum of squared errors + regularization error
def l(w, X, Y, lamb):
    loss = 0
    for x,y in zip(X,Y):
        loss += r(w, x, y)**2
    return loss + lamb*np.linalg.norm(w)**2
```

[190]:
```
# Non-linear function to approximate
def g(x):
    return x[0]*x[1] + x[2]
```

[219]:
```
# Levenberg-Marquardt algorithm (Reference: Algorithm 18.3 on textbook Pg.392)
# Choose small residual as the stopping criteria
# Max iteration is 1000 by default
# Threshold for small residual is 0.5 by default
# 0.5 is effective on original g(x), might need adjustments on g2(x)
def LM(X, Y, wk, lamb, lambk, max_iter=1000, thresh=0.5):
    losses = []
    for i in range(max_iter):
        D = Dr(wk, X)  # Jacobian
        err = rn(wk, X, Y)
        loss = l(wk, X, Y, lamb)
        losses.append(loss)  # loss append
#         if i % 100 == 0:
#             print("Iteration: {:3d} with loss: {:.5f}".format(i, loss))

        # Stopping criterium: small residual, have almost solved fw(x) = y
        if np.linalg.norm(err)**2 < thresh:
            break

        # Closed form to find minimizer
        wk_new = np.matmul(np.linalg.inv(np.matmul(D.T, D) + (lambk + lamb)*np.
 ↪identity(16)), (np.matmul(np.matmul(D.T, D), wk) - np.matmul(D.T, err) +␣
 ↪lambk*wk))

        # Check tentaive iterate
        norm = np.linalg.norm(err)**2
        newnorm = np.linalg.norm(rn(wk_new, X, Y))**2
```

```
        if newnorm < norm:
            lambk = 0.8*lambk  # reduce lambda
            wk = wk_new  # accept iterate
        else:
            lambk = 2*lambk  # increase lambda and do not update w

    return wk, losses
```

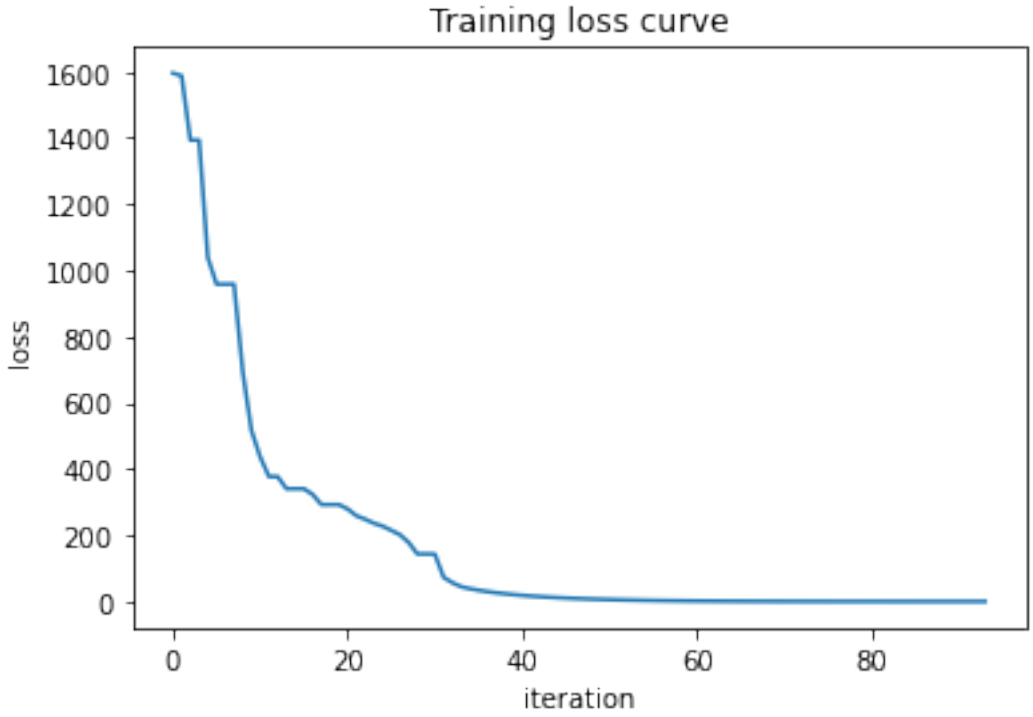
[192]:
```
random.seed(42)
```

[193]:
```
N = 500
X = random.randn(N, 3)
Y = np.array([g(x) for x in X])
```

[81]:
```
w_initial = random.randn(16)
w, losses = LM(X, Y, w_initial, 1e-5, 1)
```

```
Iteration:   0 with loss: 1595.41204
```

[82]:
```
plt.title('Training loss curve')
plt.xlabel('iteration')
plt.ylabel('loss')
plt.plot(losses)
plt.show()
```



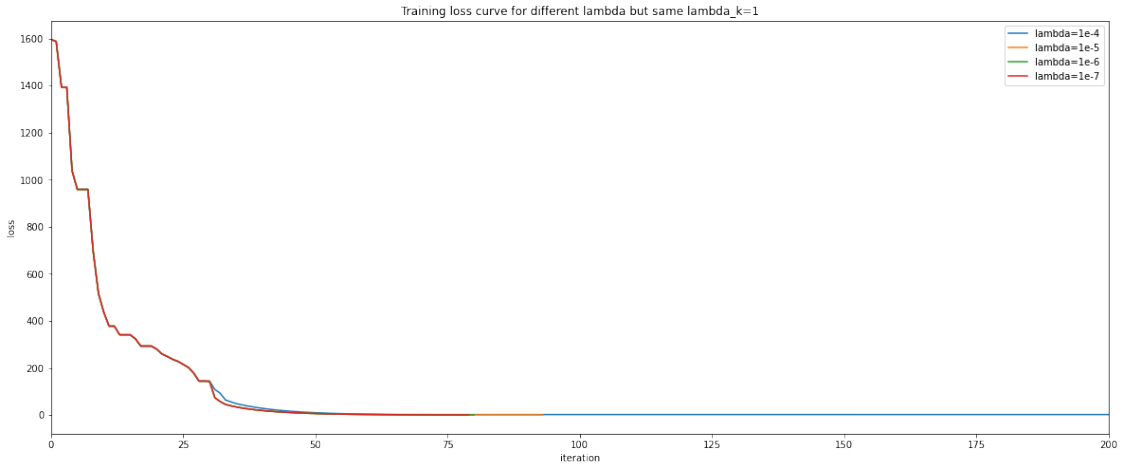
The training loss decreased with iterations and luckily with this initalization of w, it converges within 100 iterations

2.1 Try different initializations & effect on final model training loss error

[84]:
```
w2, losses2 = LM(X, Y, w_initial, 1e-4, 1)
w3, losses3 = LM(X, Y, w_initial, 1e-6, 1)
w4, losses4 = LM(X, Y, w_initial, 1e-7, 1)
```

```
Iteration:   0 with loss: 1595.41320
Iteration: 100 with loss: 2.17376
Iteration: 200 with loss: 2.17375
Iteration: 300 with loss: 2.17375
Iteration: 400 with loss: 2.17375
Iteration: 500 with loss: 2.17375
Iteration: 600 with loss: 2.17375
Iteration: 700 with loss: 2.17375
Iteration: 800 with loss: 2.17375
Iteration: 900 with loss: 2.17375
Iteration:   0 with loss: 1595.41192
Iteration:   0 with loss: 1595.41191
```

[91]:
```
plt.figure(figsize=(20,8))
plt.title('Training loss curve for different lambda but same lambda_k=1')
plt.xlabel('iteration')
plt.ylabel('loss')
plt.xlim(0, 200)
plt.plot(losses2)
plt.plot(losses)
plt.plot(losses3)
plt.plot(losses4)
plt.legend(['lambda=1e-4', 'lambda=1e-5', 'lambda=1e-6', 'lambda=1e-7'])
plt.show()
```



3 Model Test & Tuning

[92]:
```
# Build test set
NT = 100
X_test = random.randn(NT, 3)
Y_test = np.array([g(x) for x in X_test])
```

[194]:
```
# Use mean squared error (MSE) to evaluate the neural network
def calc_error(w, X, Y):
    Y_pred = np.array([f(w, x) for x in X])
    return ((Y_pred - Y)**2).mean()
```

[98]:
```
# Using the weights returned from lambda = 1e-5, lambda_k = 1
calc_error(w, X_test, Y_test)
```

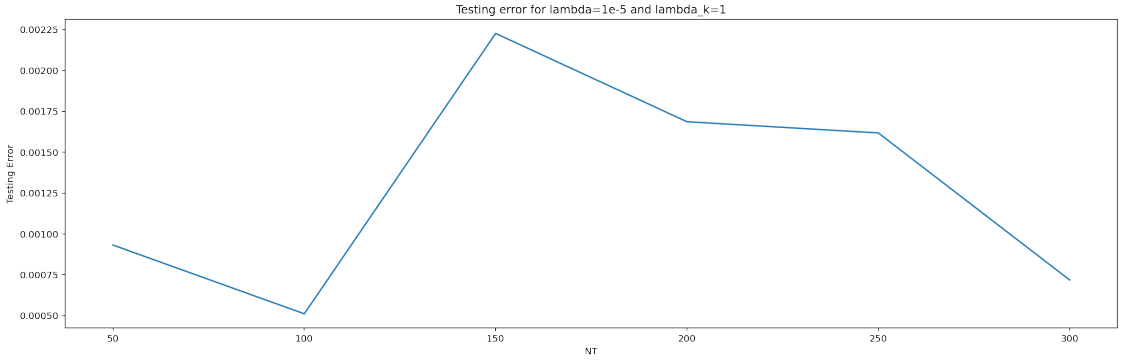
[98]:
```
0.0007839129078666426
```

3.1 Try different values of N_T

[108]:
```
NT_list = [50 + 50 * n for n in range(6)]
errors = []
for NT in NT_list:
    X_test = random.randn(NT, 3)
    Y_test = np.array([g(x) for x in X_test])
    errors.append(calc_error(w, X_test, Y_test))
```

[110]:
```
%config InlineBackend.figure_format = 'retina'
%matplotlib inline
plt.figure(figsize=(20,6))
```

```
plt.plot(NT_list, errors)
plt.xticks(NT_list, [str(s) for s in NT_list])
plt.title('Testing error for lambda=1e-5 and lambda_k=1')
plt.xlabel("NT")
plt.ylabel("Testing Error")
plt.show()
```



3.2 Try different initializations, different choices of λ & λ_k, test on different N_T

Training & Tuning

[167]:
```
from tqdm import notebook  # For loading bar visualization

lambs = [1e-9, 1e-7, 1e-5, 1e-3, 1e-1]
lambks = [1e-6, 1e-4, 1e-2, 1, 1e2]
train_errors = np.zeros((5, 5))

ws = []  # To store the best weight for each permutation of (lambda, lambda_k)
for i,lamb in notebook.tqdm(enumerate(lambs), total=len(lambs), desc='lambda'):

    wss = []  # To store the best weight for each lambda_k
    for j,lambk in notebook.tqdm(enumerate(lambks), total=len(lambks),␣
 ↪desc='lambda_k'):

        # For each permutation of (lambda, lambda_k),
        # I performed three trials with three initializations of w
        avg_train_error = 0  # Average error over three trials
        min_train_error = 100000
        best_w = None  # To store the best weight for each trial
        for _ in range(3):
            w_initial = random.randn(16)
            w, _ = LM(X, Y, w_initial, lamb, lambk, 600)  # Run LM algorithm
            temp = calc_error(w, X, Y)  # Training error for this trial
            if temp < min_test_error:
```

```
                best_w = w  # Store the best w among three trials
                min_train_error = temp
            avg_train_error += temp

        wss.append(best_w)
        train_errors[i,j] = avg_train_error / 3  # Average over three trials
    ws.append(wss)
ws = np.array(ws)
```

```
HBox(children=(FloatProgress(value=0.0, description='lambda', max=5.0, style=ProgressStyle(desc
```

```
HBox(children=(FloatProgress(value=0.0, description='lambda_k', max=5.0, style=ProgressStyle(de
```

```
HBox(children=(FloatProgress(value=0.0, description='lambda_k', max=5.0, style=ProgressStyle(de
```

```
HBox(children=(FloatProgress(value=0.0, description='lambda_k', max=5.0, style=ProgressStyle(de
```

```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:2: RuntimeWarning:
overflow encountered in exp

/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:4: RuntimeWarning:
invalid value encountered in double_scalars
  after removing the cwd from sys.path.
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:3: RuntimeWarning:
overflow encountered in exp
  This is separate from the ipykernel package so we can avoid doing imports
until
```

```
HBox(children=(FloatProgress(value=0.0, description='lambda_k', max=5.0, style=ProgressStyle(de
```

```
HBox(children=(FloatProgress(value=0.0, description='lambda_k', max=5.0, style=ProgressStyle(de
```

[182]:
```
best_train_error = np.where(train_errors == train_errors.min())
best_w = ws[best_train_error[0][0], best_train_error[1][0]]
best_lambda = lambs[best_train_error[0][0]]
best_lambdak = lambks[best_train_error[1][0]]
```

```
print("The best weights after tuning regularization parameters: ")
print(best_w)
print("The best lambda giving the lowest training error: ", best_lambda)
print("The best lambda_k giving the lowest training error: ", best_lambdak)
```

```
The best weights after tuning regularization parameters:
[-1.26630243e+02  4.68676768e-03 -5.19749477e-02 -7.85426137e-03
 -1.10956017e-01  7.12899571e+01  1.21802999e-01 -7.27544926e-02
  1.27533059e-03  7.16264274e-01 -5.17522579e+01  1.43069443e-01
  9.64391942e-02  1.08571561e-03  6.78192026e-01 -2.72481468e+01]
The best lambda giving the lowest training error:  1e-07
The best lambda_k giving the lowest training error:  0.01
```

[258]:
```
import seaborn as sns  # merely for training error visualization

def plotErrors(errors, cmap=None):
    fig = plt.figure(figsize=(10,8))
    ax = fig.add_subplot(111)
    sns.heatmap(errors, annot=True, ax = ax, fmt='g', xticklabels=lambks,␣
 ↪yticklabels=lambs, cmap=cmap)
    ax.set_xlabel('lambda_k')
    ax.set_ylabel('lambda')
    ax.set_title('Training Errors')
```

[256]:
```
# Plot training errors in a heatmap
# To better visualize different permutation of lambda and lambda_k
plotErrors(train_errors)
```

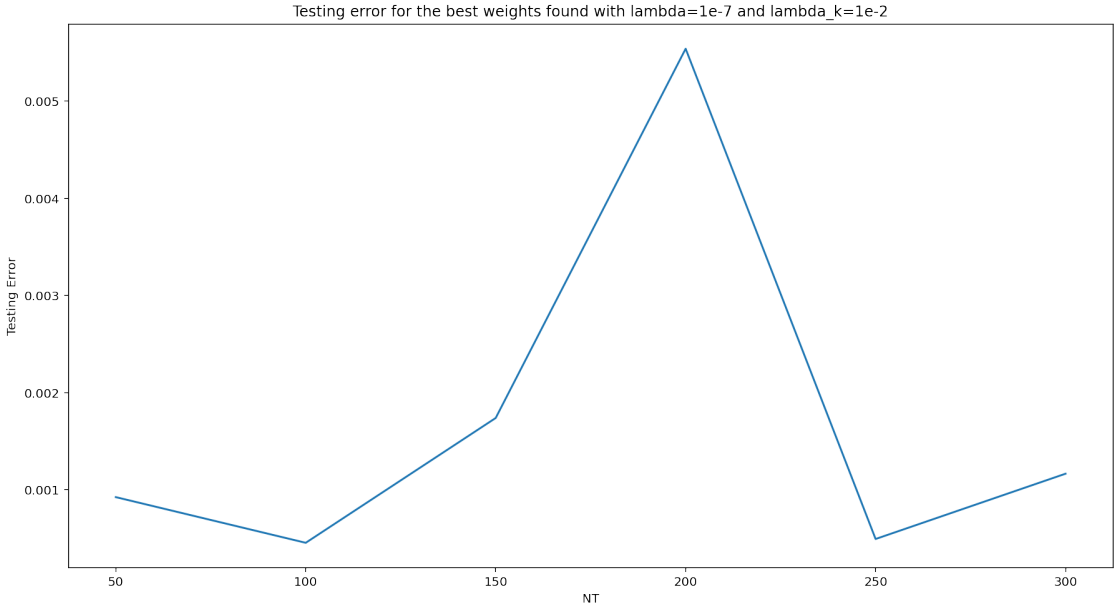
Test the best regularization parameter on test sets with different size

[198]:
```
# NT from 50 to 300 with a step size of 50
NT_list = [50 + 50 * n for n in range(6)]

errors = []
for NT in NT_list:
    X_test = random.randn(NT, 3)
    Y_test = np.array([g(x) for x in X_test])
    errors.append(calc_error(best_w, X_test, Y_test))  # Use the best weights
```

[272]:
```
%config InlineBackend.figure_format = 'retina'
%matplotlib inline
plt.figure(figsize=(15,8))
plt.plot(NT_list, errors)
plt.xticks(NT_list, [str(s) for s in NT_list])
```

```
plt.title('Testing error for the best weights found with lambda=1e-7 and␣
 ↪lambda_k=1e-2')
plt.xlabel("NT")
plt.ylabel("Testing Error")
plt.show()
```



[204]:
```
print("Average Testing Error: ", sum(errors)/len(errors))
```

```
Average Testing Error:  0.0017187574459052715
```

3.3 Summary & Comments

To reiterate:
λ is the regularization constant for the weight itself, to prevent the weights from blowing up.
λ_k is the regularization coefficient to control how big of a jump the algorithm make from one iteration to the next.
λ is constant across one run of the LM algorithm while λ_k is updated during each iteration in a run of the LM algorithm.

Error Metrics
I used Mean Squared Error (MSE) to evaluate the trained neural network. It's defined as

$$\frac{1}{N}\sum_{n=1}^{N}\left(f_w(x^{(n)}) - y^{(n)}\right)^2$$

And it's defined as function calc_error at the very top of this section.

Training Method
I tried λ from 10^{-9} to 10^{-1} with a multiplication step of 100, along with λ_k from 10^{-6} to 10^{2} with

a multiplication step of 100. In addition, I performed three trials per permutation of λ and λ_k because I initialized the starting weights using random numbers of normal distribution. To avoid bias, three trials of random initializations were conducted every permutation and the training error was averaged over three trials, such that the training is more reasonable since it won't be affected by a single "bad" starting point.

I also tried warm starting to start the LM algorithm at the solution of the previously solved problem, i.e. the minimzer w of the previous run of the LM algorithm. However, the training errors were close to each other for different permutations of λ and λ_k, which was not ideal to tune and pick the best regularization parameters. Although warming start greatly reduce the number of iterations required to converge, it didn't provide much helpful insight into our parameter tuning.

In short, as above I have conducted a tuning of our regularization parameters with $5 * 5 * 3 = 75$ iterations, with 25 permutations of (λ, λ_k) and three random initialization trials for each.

Training Results
I have found that the lowest training error 0.001066 was given by $\lambda = 10^{-7}, \lambda_k = 10^{-2}$. I also stored the best weights among the three trials with that initialization of λ and λ_k.
Additionally, In the heatmap drawn, we can find a trend that smaller λ and larger λ_k give smaller training errors, while larger λ and smaller λ_k give bigger training errors.

Testing
Since I found the best weights, I just evaluate our simple neural network with the best weights found on test sets with different sizes, i.e. N_T. I tested it on 6 different N_T, from 50 to 300 with a step size of 50. The testing errors versus N_T has been plotted. Finally, I averaged over six errors and gave the final testing error: 0.001718757.

Comments on my results
I believe I have conducted a thorough tuning of different initializations of λ and λ_k. I tried five different λ with another five different λ_k. Plus, to avoid bias when randomly initializing the starting weights, I performed three trials per permutation. I also tried warm starting but the result did not help much with finding the best parameters. I tested the best weights found on six sets of test points with different N_T, as required. Finally the average testing error of the best weights is 0.00172, a little higher than the training error 0.001066, which is reasonable and intuitive.

4 Different Non-linear Function $g_2(x)$

The non-linear function I chose is:

$$g_2(\mathbf{x}^{(n)}) = \left(x_1^{(n)}\right)^2 + x_2^{(n)} + x_3^{(n)}$$

$$y^{(n)} = g_2(\mathbf{x}^{(n)}), n = 1, 2, \ldots, N$$

Code Re-use
Since I implemented my LM algorithm by modular programming, i.e. with independent, interchangeable modules, I can re-use most of my code without modifications. For example, my $f_w(x)$, $r_n(w)$, $l(w)$ are all independent of the non-linear function $g(x)$ to approximate. The only thing I need to take care of besides just copying and pasting is to find a reasonable max_iteration and threshold for the "small residual" stopping criterium. Therefore, below I will first try a few initializations of λ on g_2, and observe a few things:

- Check if our simple neural network $f_w(x)$ is able to handle the complexity of the new non-linear function g_2
- i.e. see if the LM algorithm is able to converge with a reasonably small "small residual" value
- If it's able to fit g_2, empirically choose the new "small residual" threshold for the LM algorithm (0.5 for g before)
- If it's able to fit g_2, empirically choose a number of maximum iteration for the LM algorithm (1000 for g before)

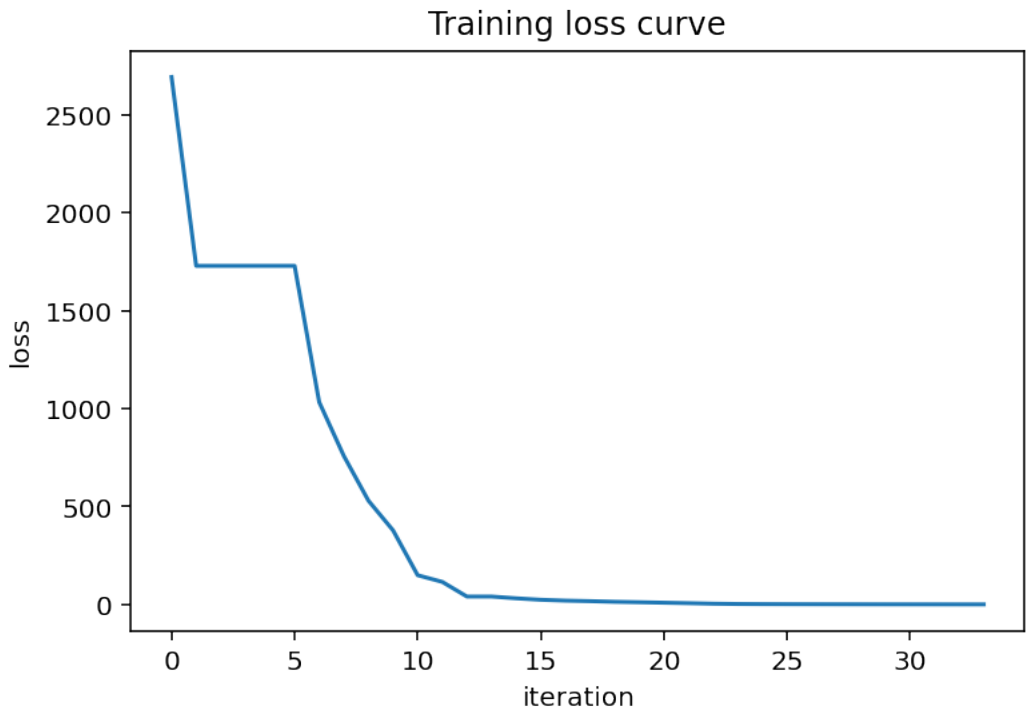
[229]:
```
# Different Non-linear function to approximate
def g2(x):
    return x[0]**2 + x[1] + x[2]
```

[230]:
```
random.seed(42)
```

[231]:
```
N = 500
X2 = random.randn(N, 3)
Y2 = np.array([g2(x) for x in X2])
```

[232]:
```
w_initial = random.randn(16)
w, losses = LM(X2, Y2, w_initial, 1e-5, 1)
```

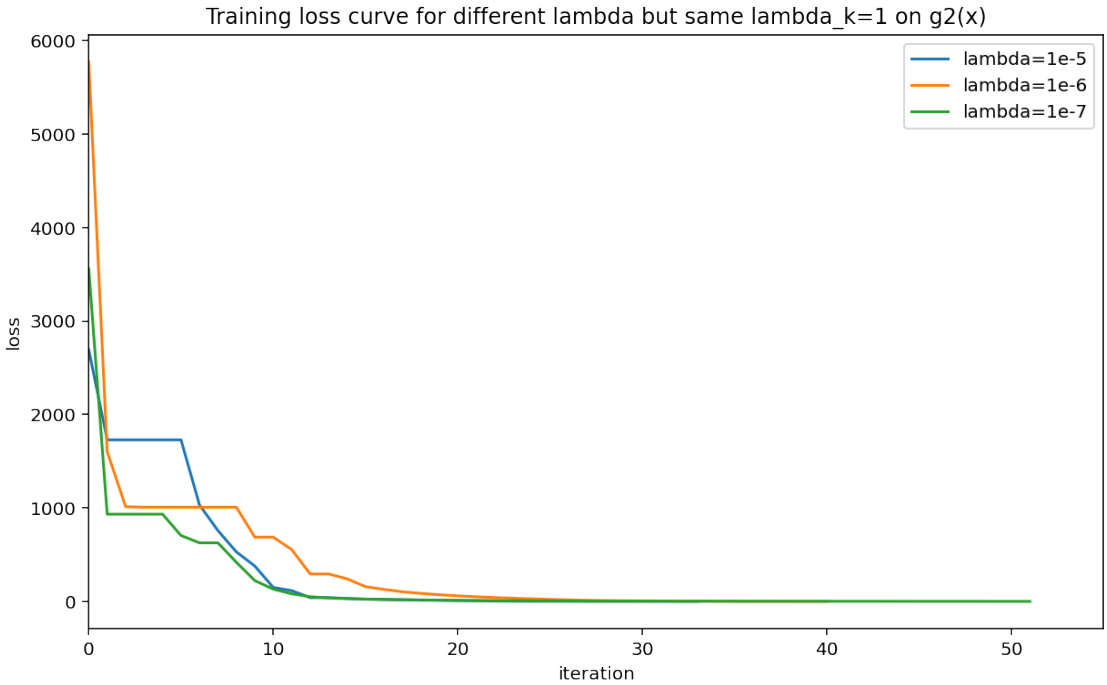
[233]:
```
plt.title('Training loss curve')
plt.xlabel('iteration')
plt.ylabel('loss')
plt.plot(losses)
plt.show()
```



4.1 Try different initializations & effect on final model training loss error

[ ]:
```
w_initial = random.randn(16)
w2, losses2 = LM(X2, Y2, w_initial, 1e-6, 1)
w_initial = random.randn(16)
w3, losses3 = LM(X2, Y2, w_initial, 1e-7, 1)
```

[242]:
```
plt.figure(figsize=(10,6))
plt.title('Training loss curve for different lambda but same lambda_k=1 on␣
 ↪g2(x)')
plt.xlabel('iteration')
plt.ylabel('loss')
plt.xlim(0, 55)
plt.plot(losses)
plt.plot(losses2)
plt.plot(losses3)
plt.legend(['lambda=1e-5', 'lambda=1e-6', 'lambda=1e-7'])
plt.show()
```



Observation

Using the threshold 0.5 as I used for $g(x)$, the LM algorithm successfully converged for $g_2(x)$ as well and actually faster. Therefore, I will decrease the max_iteration argument to 200 and keep

the same thresh argument in my LM algorithm when varying different λ and λ_k below.

4.2 Try different initializations, different choices of λ & λ_k, test on different N_T

Training & Tuning

[243]:
```
from tqdm import notebook  # For loading bar visualization

lambs = [1e-9, 1e-7, 1e-5, 1e-3, 1e-1]
lambks = [1e-6, 1e-4, 1e-2, 1, 1e2]
train_errors2 = np.zeros((5, 5))

ws2 = []  # To store the best weight for each permutation of (lambda, lambda_k)
for i,lamb in notebook.tqdm(enumerate(lambs), total=len(lambs), desc='lambda'):

    wss = []  # To store the best weight for each lambda_k
    for j,lambk in notebook.tqdm(enumerate(lambks), total=len(lambks),␣
 ↪desc='lambda_k'):

        # For each permutation of (lambda, lambda_k),
        # I performed three trials with three initializations of w
        avg_train_error = 0  # Average error over three trials
        min_train_error = 100000
        best_w = None  # To store the best weight for each trial
        for _ in range(3):
            w_initial = random.randn(16)

            # Run LM algorithm with a smaller max iteration (explained above)
            w, _ = LM(X2, Y2, w_initial, lamb, lambk, 200)
            temp = calc_error(w, X2, Y2)  # Training error for this trial
            if temp < min_test_error:
                best_w = w  # Store the best w among three trials
                min_train_error = temp
            avg_train_error += temp

        wss.append(best_w)
        train_errors2[i,j] = avg_train_error / 3  # Average over three trials
    ws2.append(wss)
ws2 = np.array(ws2)
```

```
HBox(children=(FloatProgress(value=0.0, description='lambda', max=5.0, style=ProgressStyle(desc
```

```
HBox(children=(FloatProgress(value=0.0, description='lambda_k', max=5.0, style=ProgressStyle(de
```

```
HBox(children=(FloatProgress(value=0.0, description='lambda_k', max=5.0, style=ProgressStyle(de
```

```
HBox(children=(FloatProgress(value=0.0, description='lambda_k', max=5.0, style=ProgressStyle(de
```

```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:3: RuntimeWarning:
overflow encountered in exp
  This is separate from the ipykernel package so we can avoid doing imports
until
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:4: RuntimeWarning:
invalid value encountered in double_scalars
  after removing the cwd from sys.path.
```

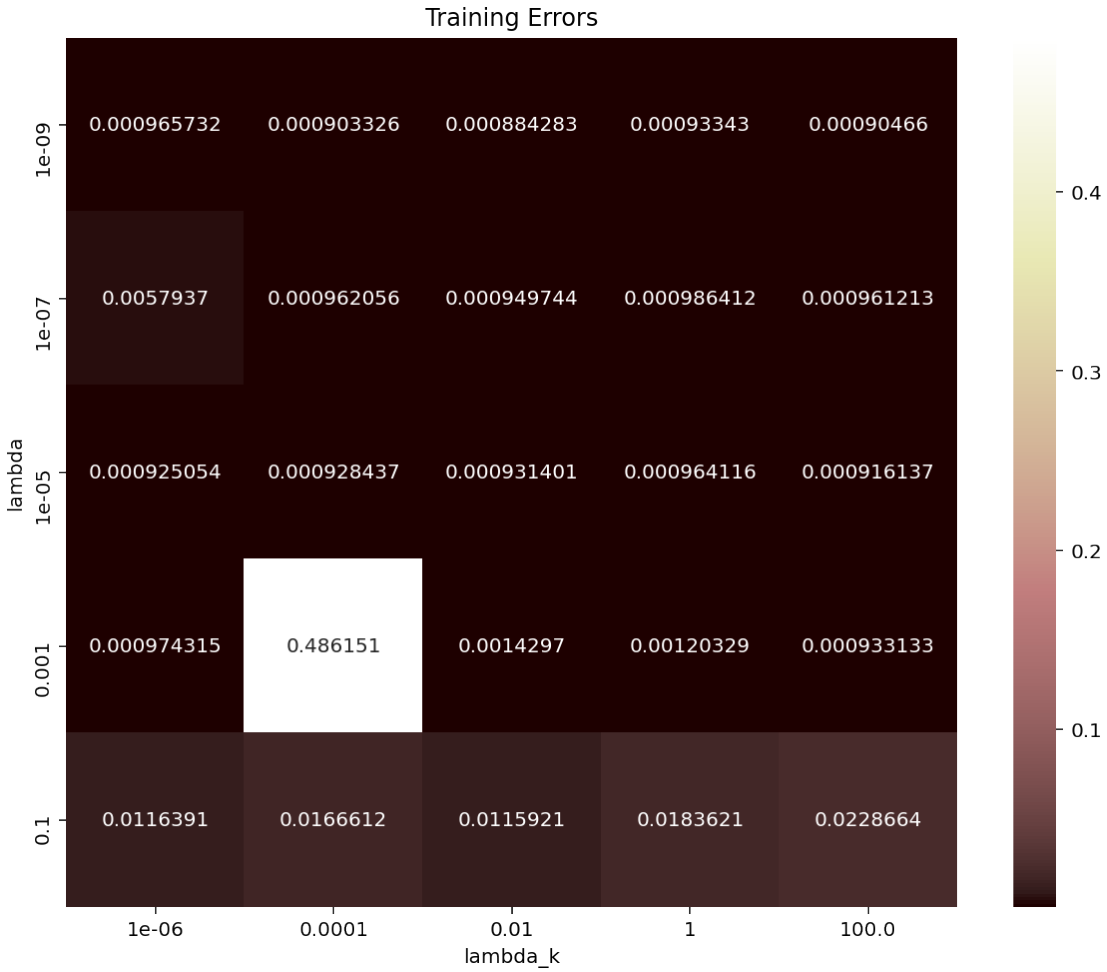
```
HBox(children=(FloatProgress(value=0.0, description='lambda_k', max=5.0, style=ProgressStyle(de
```

```
HBox(children=(FloatProgress(value=0.0, description='lambda_k', max=5.0, style=ProgressStyle(de
```

[267]:
```
best_train_error2 = np.where(train_errors2 == train_errors2.min())
best_w2 = ws2[best_train_error2[0][0], best_train_error2[1][0]]
best_lambda2 = lambs[best_train_error2[0][0]]
best_lambdak2 = lambks[best_train_error2[1][0]]
print("The best weights for the neural network to fit g_2: ")
print(best_w2)
print("The best lambda to fit g_2 with the lowest training error: ",␣
↪best_lambda2)
print("The best lambda_k to fit g_2 with the lowest training error: ",␣
↪best_lambdak2)
```

```
The best weights for the neural network to fit g_2:
[ 7.79713407e+00 -5.88474221e-01  2.59440650e-03  4.72935879e-03
 -1.51180223e+00 -7.59468597e+00 -5.90099612e-01 -1.15932515e-03
 -6.29932070e-03  1.42725847e+00 -1.08368599e+01  1.11589204e-02
 -9.29500939e-02 -9.18220129e-02 -1.00025905e-02  1.37522336e+01]
The best lambda to fit g_2 with the lowest training error:  1e-09
The best lambda_k to fit g_2 with the lowest training error:  0.01
```

[261]:
```
# Plot training errors in a heatmap
# To better visualize different permutation of lambda and lambda_k
plotErrors(train_errors2, "pink")
```

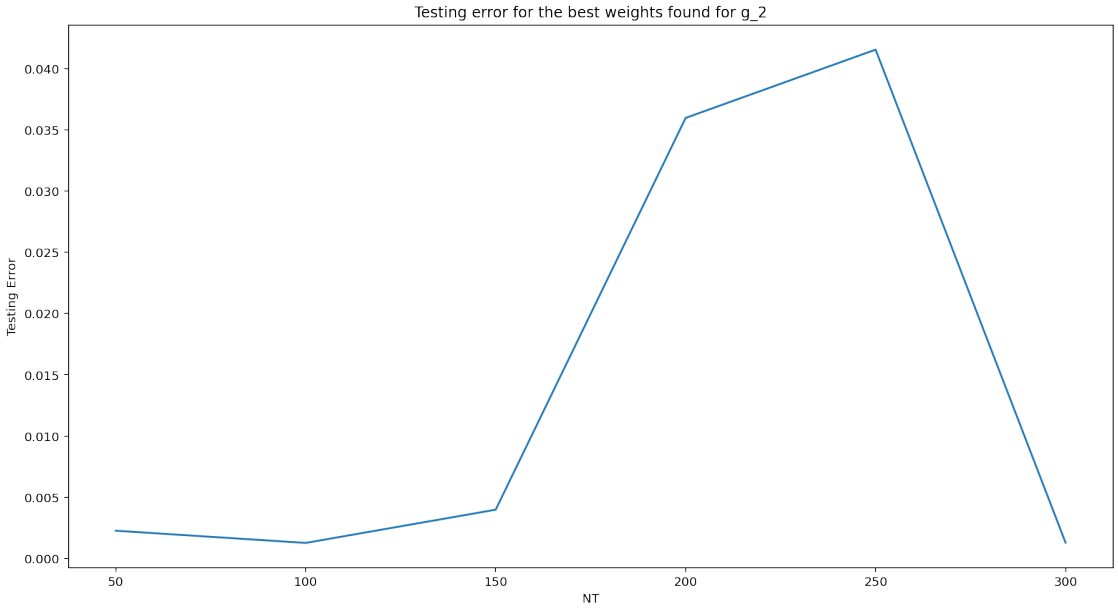


Test the best regularization parameter on test sets with different size

[268]:
```
NT_list = [50 + 50 * n for n in range(6)]
errors2 = []
for NT in NT_list:
    X_test2 = random.randn(NT, 3)
    Y_test2 = np.array([g2(x) for x in X_test2])
    errors2.append(calc_error(best_w2, X_test2, Y_test2))  # Use the best␣
 ↪weights
```

[271]:
```
%config InlineBackend.figure_format = 'retina'
%matplotlib inline
plt.figure(figsize=(15,8))
plt.plot(NT_list, errors2)
plt.xticks(NT_list, [str(s) for s in NT_list])
plt.title('Testing error for the best weights found for g_2')
plt.xlabel("NT")
```

```
plt.ylabel("Testing Error")
plt.show()
```



[270]:
```
print("Average Testing Error: ", sum(errors2)/len(errors2))
```

```
Average Testing Error:  0.014386148541055098
```

4.3 Summary & Comments

Observations
Compared to $g(\mathbf{x})$, our neural network $f_w(\mathbf{x})$ actually performs better to fit $g_2(\mathbf{x})$ because it uses fewer iterations to converge to the same "small residual" threshold. Therefore, it's reasonable to think of g_2 as a less complex non-linear function than the original g.

Error Metrics
Same as before, I used Mean Squared Error (MSE) to evaluate the trained neural network.

Training Method
Same as before, I have conducted a tuning of our regularization parameters with $5 * 5 * 3 = 75$ iterations, with 25 permutations of (λ, λ_k), three 3 initialization trials for each.

Training Results
I have found that the lowest training error 0.000884 was given by $\lambda = 10^{-9}, \lambda_k = 10^{-2}$. I also stored the best weights among the three trials with that initialization of λ and λ_k.
Additionally, In the heatmap drawn, we can find the trend that generally, smaller λ gives smaller training errors, hence better. This is a little different from what we found for $g(\mathbf{x})$.

Testing
Same as before, to evaluate our simple neural network with the best weights found on test sets with different sizes, I tested it on 6 different N_T, from 50 to 300 with a step size of 50. The testing

errors versus N_T has been plotted. Finally, I averaged over six errors and gave the final testing error: 0.01438615.

Comments on my results

I believe I have conducted a thorough tuning of different initializations of λ and λ_k on the new non-linear function $g_2(\mathbf{x})$. Same as before, I tried five different λ with another five different λ_k. Plus, to avoid bias when randomly initializing the starting weights, I performed three trials per permutation. I tested the best weights found on six sets of test points with different N_T, as required. Finally the average testing error of the best weights is 0.014386, a little higher than the training error 0.000884, which is reasonable and intuitive.