5LSL0 Assignment 1: Optimum Linear and Adaptive Filters and Familiarisation with Pytorch

Name	Student ID
Yiling Zhang	1756117
Liyuan Jiang	1704257

Lecturer: Rik Vullings, Ruud van Sloun
Nishith Chennakeshava, Hans van Gorp

Eindhoven, May 12, 2022

Optimum linear and adaptive filters

Exercise 1: Known statistics

1.2.1 Wiener Filter

a

$$\begin{aligned} J = E(e[k]^2) &= E((y[k] - \underline{\mathbf{w}}^t \underline{\mathbf{x}}[k])(y[k] - \underline{\mathbf{x}}^t[k]\underline{\mathbf{w}})) \\ &= E(y^2[k]) - \underline{\mathbf{w}}^t E(y[k]\underline{\mathbf{x}}[k]) - E(y[k]\underline{\mathbf{x}}^t[k])\underline{\mathbf{w}} + \underline{\mathbf{w}}^t E(\underline{\mathbf{x}}[k]\underline{\mathbf{x}}^k[k])\underline{\mathbf{w}} \\ &= E(y^2[k]) - \underline{\mathbf{w}}^t \underline{\mathbf{r}}_{yx} - \underline{\mathbf{r}}_{yx}^t \underline{\mathbf{w}} + \underline{\mathbf{w}}^t R_x \underline{\mathbf{w}} \\ \nabla_{\underline{\mathbf{w}}} J &= 0 - \underline{\mathbf{r}}_{yx} - \underline{\mathbf{r}}_{yx} + 2(R_x \underline{\mathbf{w}}) \\ &= -2(\underline{\mathbf{r}}_{yx} - R_x \underline{\mathbf{w}}) \end{aligned}$$

Let $\nabla_{\underline{\mathbf{w}}} J = 0$, then $\underline{\mathbf{w}} = R_x^{-1}\underline{\mathbf{r}}_{yx}$, so

$$\underline{\mathbf{w}} = \underline{\mathbf{w}}_0 = \begin{bmatrix} 5 & -1 & -2 \\ -1 & 5 & -1 \\ -2 & -1 & 5 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 5.3 \\ -3.9 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 1 \\ -0.5 \end{bmatrix}$$

b

$$\begin{aligned} \underline{\mathbf{r}}_{xe} &= E\{\underline{\mathbf{x}}[k](y[k] - \underline{\mathbf{w}}^t \underline{\mathbf{x}}[k])\} \\ &= E\{\underline{\mathbf{x}}[k]y[k]\} - E\{\underline{\mathbf{x}}[k]\underline{\mathbf{w}}^t \underline{\mathbf{x}}[k]\} \\ &= \underline{\mathbf{r}}_{yx} - E\{\underline{\mathbf{x}}[k]\underline{\mathbf{w}}^t \underline{\mathbf{x}}[k]\} \end{aligned}$$

When $\underline{\mathbf{w}} = \underline{\mathbf{w}}_o$, $\underline{\mathbf{r}}_{yx} = \mathbf{R}_x \underline{\mathbf{w}}$. Substituting $\mathbf{R}_x \underline{\mathbf{w}}$ for $\underline{\mathbf{r}}_{yx}$ in the above equation, we can get the below expression:

$$
\begin{aligned}
\underline{\mathbf{r}}_{xe} &= \mathbf{R}_x\underline{\mathbf{w}} - E\{\underline{\mathbf{x}}[k]\underline{\mathbf{w}}^t\underline{\mathbf{x}}[k]\} \\
&= E\{\underline{\mathbf{x}}[k]\underline{\mathbf{x}}^t[k]\underline{\mathbf{w}}\} - E\{\underline{\mathbf{x}}[k]\underline{\mathbf{w}}^t\underline{\mathbf{x}}[k]\} \\
&= E\{\underline{\mathbf{x}}[k]\hat{y}[k]\} - E\{\underline{\mathbf{x}}[k]\hat{y}[k]\} \\
&= 0
\end{aligned}
$$

It means that the input and residual would be uncorrelated if we use the optimal Wiener filter. This result is explainable as a prior assumption for an effective MMSE model is that the noise is Gaussian distributed(i.e. $\hat{y}[k] = \underline{\mathbf{w}}^t\underline{\mathbf{x}}[k] + \epsilon, \epsilon \sim N(\mu, \sigma^2)$). If the minimum cost is obtained using MMSE, than the error should come from noise, which is uncorrelated to signal.

c

If statistical information is not available, we could estimate these statistics from a series of data samples with the assumption of ergodicity. More specifically, we can collect L data vectors(each of length N), and also the reference signal should contain L samples, then the original MMSE problem with known statistics can be transformed into a LS problem when they are unknown, i.e. $\underline{\mathbf{w}_{ls,o}} = argmin_{\underline{\mathbf{w}}}|\underline{\mathbf{y}} - \mathbf{X}\cdot\underline{\mathbf{w}}|^2$, where

$$
\mathbf{X} = \begin{bmatrix} \underline{x}^t[k] \\ \underline{x}^t[k-1] \\ \vdots \\ \underline{x}^t[k-L+1] \end{bmatrix}, \underline{\mathbf{y}} = \begin{bmatrix} \underline{y}[k] \\ \underline{y}[k-1] \\ \vdots \\ \underline{y}[k-L+1] \end{bmatrix}, \underline{\mathbf{w}} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{L-1} \end{bmatrix}.
$$

Then $\hat{\mathbf{R}_x}$ and $\hat{\underline{\mathbf{r}}_{yx}}$ can be estimated with these data samples.

$$
\hat{\mathbf{R}_x} = \frac{1}{L}\overline{\mathbf{R}_x} = \frac{1}{L}\mathbf{X}^t\mathbf{X}
$$

$$
\hat{\underline{\mathbf{r}}_{yx}} = \frac{1}{L}\overline{\underline{\mathbf{r}}_{yx}} = \frac{1}{L}\mathbf{X}^t\underline{\mathbf{y}}
$$

In the processing of estimation, there is a tradeoff between accuracy and computation cost. We know that a longer time series recording contains more statistics and distribution information, which leads to higher estimation accuracy. However, the processing speed would be slower as the estimation needs more computation cost and operations.

1.2.2 Steepest Gradient Descent

d

SGD algorithm can be expressed as

$$\underline{\mathbf{w}}[k+1] = \underline{\mathbf{w}}[k] + 2\alpha(\underline{\mathbf{r}}_{yx} - \mathbf{R}_x\underline{\mathbf{w}}[k])$$

where $\nabla = -2(\underline{\mathbf{r}}_{yx} - \mathbf{R}_x\underline{\mathbf{w}}[k])$ is the gradient of cost function.

We know that the gradient would be 0 when the minimum cost is achieved, so SGD converges to Wiener solution, i.e. $\lim_{k\to\infty} \underline{\mathbf{w}}[k] \simeq \mathbf{R}_x^{-1} \cdot \underline{\mathbf{r}}_{yx}$. Therefore, when SGD reaches steady-state, $\underline{\mathbf{w}}[k]$ should be similar or equal to $\mathbf{R}_x^{-1} \cdot \underline{\mathbf{r}}_{yx}$.

e

$$det(\mathbf{R}_x - \lambda\mathbf{I}) = \begin{vmatrix} 5-\lambda & -1 & -2 \\ -1 & 5-\lambda & -1 \\ -2 & -1 & 5-\lambda \end{vmatrix} = (\lambda-7)(\lambda+\sqrt{3}-4)(\lambda-\sqrt{3}-4) = 0$$

We can get $\lambda_1 = 7, \lambda_2 = -\sqrt{3}+4 \approx 2.27, \lambda_3 = \sqrt{3}+4 \approx 5.73$ from the above equation. The GD recursion would be stable iff: $\lim_{k\to\infty}(\mathbf{I} - 2\alpha\wedge)^k = 0 \Leftrightarrow |1-2\alpha\lambda_i| < 1$ for each eignvalue $\Leftrightarrow 0 < \alpha < \frac{1}{\lambda_{max}}$, so $\alpha \in (0, \frac{1}{7}) \approx (0, 0.143)$

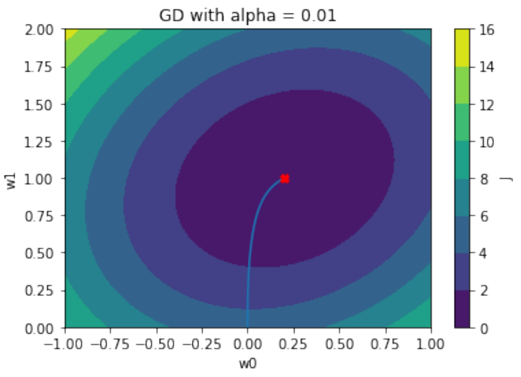
f

As shown in Fig.1(a), the upgrade of two filter coefficients follows the gradient of cost function and they finally converge at the minimum, while the filter weights do not converge uniformly as the eigenvalue spread of $\mathbf{R}_x$ influences the rate of convergence of individual weights. For example, with w_1 moving from 0 to 0.5 approximately, there is not obvious adjustment upon w_0 so that the cost is barely changed in the horizontal direction. Additionally, the movement does not follow the fastest direction towards the minimum as coefficients move through a curve line to achieve convergence.

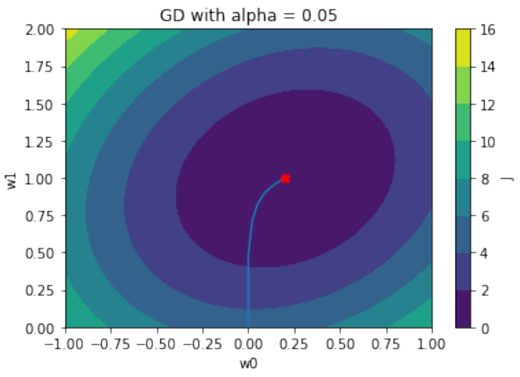
1.2.3 Newton's Method

g

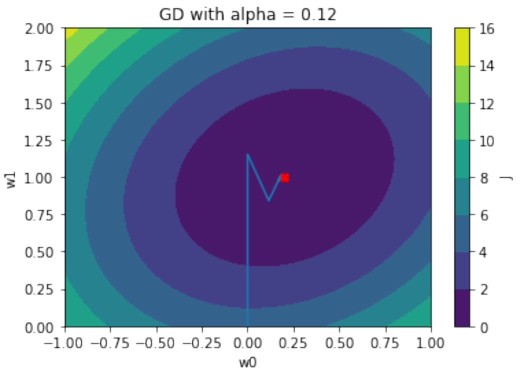
Newton's method can be written as $\underline{\mathbf{w}}[k+1] = \underline{\mathbf{w}}[k] + 2\alpha\mathbf{R}_x^{-1}(\underline{\mathbf{r}}_{yx} - \mathbf{R}_x\underline{\mathbf{w}}[k])$, then we can get



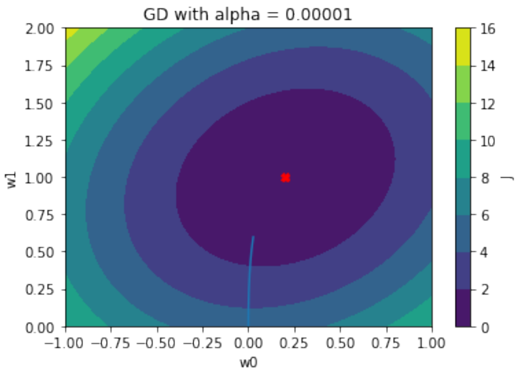
(a) $\alpha = 0.01$



(b) $\alpha = 0.05$



(c) $\alpha = 0.12$



(d) $\alpha = 10^{-5}$

Figure 1: The trajectory of the filter coefficients(SGD)

$$
\begin{aligned}
\underline{\mathbf{d}}[k+1] &= \underline{\mathbf{w}}[k+1] - \underline{\mathbf{w}}_0 \\
&= \underline{\mathbf{w}}[k] + 2\alpha \mathbf{R}_x^{-1}(\underline{\mathbf{r}}_{yx} - \mathbf{R}_x \underline{\mathbf{w}}[k]) - \mathbf{R}_x^{-1}\underline{\mathbf{r}}_{yx} \\
&= (\mathbf{I} - 2\alpha \mathbf{R}_x^{-1}\mathbf{R}_x)\underline{\mathbf{w}}[k] - (\mathbf{R}_x^{-1}\underline{\mathbf{r}}_{yx} - 2\alpha \mathbf{R}_x^{-1}\underline{\mathbf{r}}_{ex}) \\
&= (\mathbf{I} - 2\alpha)\underline{\mathbf{w}}[k] - (\mathbf{I} - 2\alpha)\mathbf{R}_x^{-1}\underline{\mathbf{r}}_{yx} \\
&= (\mathbf{I} - 2\alpha)(\underline{\mathbf{w}}[k] - \underline{\mathbf{w}}_0) \\
&= (\mathbf{I} - 2\alpha)\underline{\mathbf{d}}[k] \qquad (1)
\end{aligned}
$$

From the equation (1), we can get that Newton's method make all weights have same convergence as the convergence rates for all directions are same, i.e. $(1 - 2\alpha)$. There is no shape influence from $\mathbf{R}_x$ in contrast to SGD.

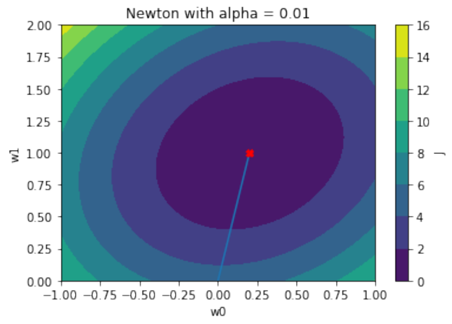
h

According to equation (1), $\underline{\mathbf{d}}[k] = (1-2\alpha)^k \underline{\mathbf{d}}[0]$. Newton's method would be stable iff:

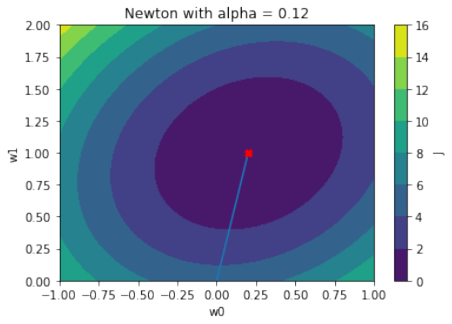
$$\lim_{k\to\infty}(1-2\alpha)^k = 0 \Leftrightarrow |1-2\alpha| < 1 \Rightarrow 0 < \alpha < 1$$

i

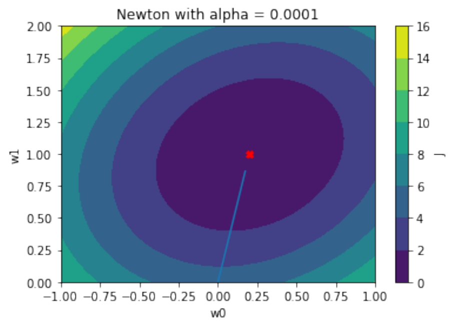
As shown in Fig.2, with the whitening of input process, the convergence rate along each direction are corrected to be same so that the filter coefficients goes to the optimal solution directly along the straight path from the initial position to the optimal destination.



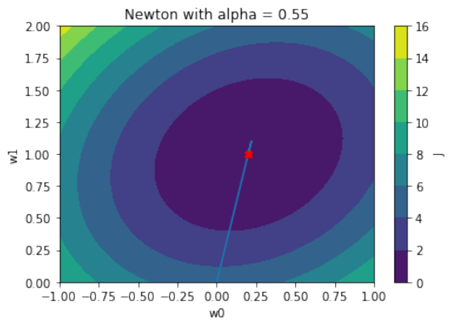
(a) $\alpha = 0.01$



(b) $\alpha = 0.12$



(c) $\alpha = 10^{-4}$



(d) $\alpha = 0.55$

Figure 2: The trajectory of the filter coefficients(Newton)

```python
def Newton(alpha,w_init,N):
    wk = w_init
    w = []
    for i in range(N):
        w.append(wk)
        wk = wk + 2*alpha*np.linalg.inv(Rx)@(ryx-Rx@wk)
    return np.array(w)
```

Exercise 2: Unknown Statistics

1.3.1 LMS and NLMS

j

The weights (Fig.3) converge quickly to the optimal with oscillation. And it keeps oscillating around the optimal weights w_0. In a word, the convergence curve with a larger α contains more fluctuation as a larger α may make the weights go cross the optimal solution over and over again. The function even can not be stable if α is too large.

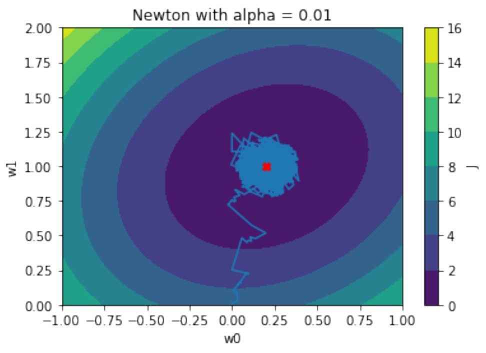
```
x_stack = np.vstack((x, shift(x, 1), shift(x, 2))).T
```

```
def LMS(alpha,w_init,N):
    wk = w_init
    w = []
    for i in range(N):
        w.append(wk)
        x_k = x_stack[i]
        y_h = wk@x_k
        e_k = y[i] - y_h
        wk = wk + 2 * alpha * e_k * x_k
    return np.array(w)
```

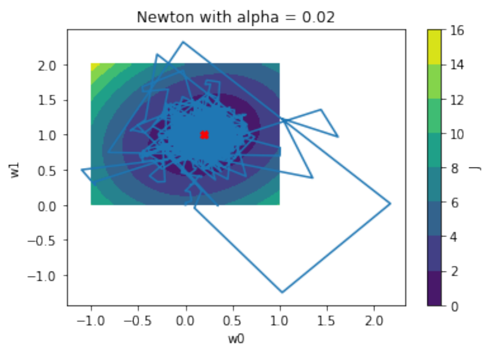
k

NLMS (Fig.4) has nearly the same pattern with LMS, while the oscillation of NLMS is smaller for a same learning rate α.

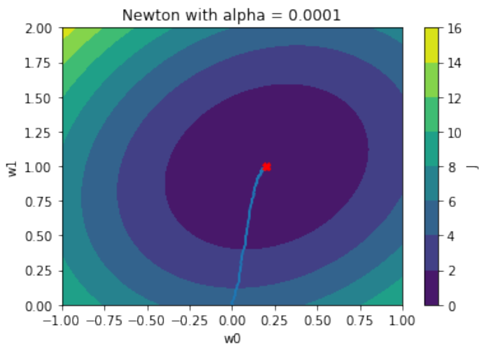
```
def NLMS(alpha,w_init,N):
    wk = w_init
    w = []
    for i in range(N):
        w.append(wk)
        x_k = x_stack[i]
        y_h = wk@x_k
        e_k = y[i] - y_h
        sigma = x_k@x_k/3 + 1e-8
        wk = wk + 2 * alpha * e_k * x_k / sigma
    return np.array(w)
```



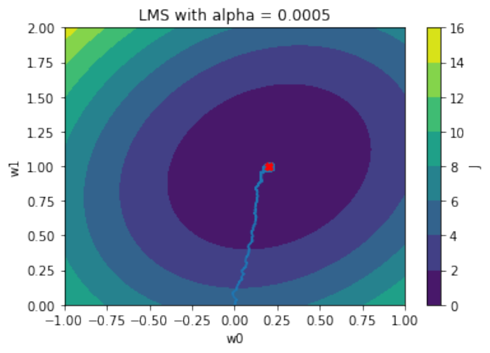
(a) $\alpha = 0.01$



(b) $\alpha = 0.02$



(c) $\alpha = 10^{-4}$



(d) $\alpha = 5E^{-4}$

Figure 3: The trajectory of the filter coefficients(LMS)

1.3.2 RLS

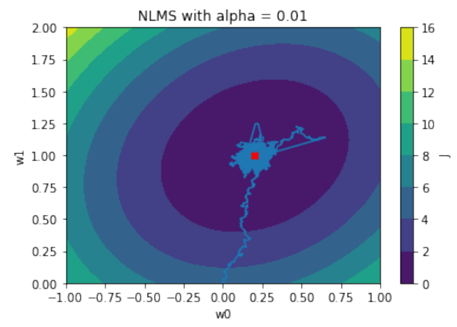
1

• RLS:

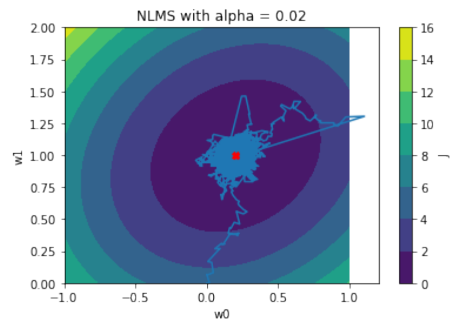
RLS basically finds an efficient recursive solution for LS problem from k $\rightarrow$ k+1. For data block length fixed, LS problem becomes

$$\underline{\mathbf{w}}_{LS}[k] = (\mathbf{X}^t[k]\mathbf{X}[k])^{-1}(\mathbf{X}^t[k]\underline{\mathbf{y}}[k])$$

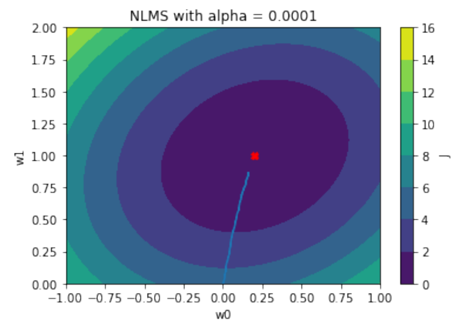
With a exponential sliding window, the data can be scaled down by a forgetting factor γ(i.e. control the memory size). Then $\mathbf{X}^t$ and $\underline{\mathbf{y}}[k]$ in the above LS problem can be written as below.



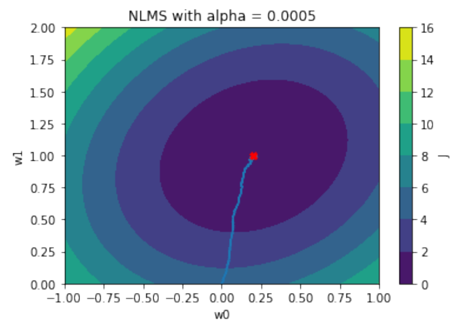
(a) $\alpha = 0.01$



(b) $\alpha = 10^{-4}$



(c) $\alpha = 0.02$



(d) $\alpha = 5E^{-4}$

Figure 4: The trajectory of the filter coefficients(NLMS)

$$\mathbf{X}[k] = \begin{bmatrix} \gamma^0 \underline{x}^t[k] \\ \gamma^1 \underline{x}^t[k-1] \\ \vdots \\ \gamma^k \underline{x}^t[0] \end{bmatrix} \quad \underline{\mathbf{y}}[k] = \begin{bmatrix} \gamma^0 \underline{y}[k] \\ \gamma^1 \underline{y}[k-1] \\ \vdots \\ \gamma^k \underline{y}[0] \end{bmatrix}$$

We can see from the expression that the basic issue behind RLS is LS, while RLS uses a recursive method to solve it. With a forgetting factor, previous estimations(i.e.$\overline{\mathbf{R}}_x^{-1}[k]$ and $\overline{\mathbf{r}}_{yx}[k]$) and current values(i.e.$\mathbf{x}[k+1]$ and $y[k+1]$), the current weights can be estimated(i.e. $\underline{\mathbf{w}}[k+1]$).

In practice, LS requires L sample vectors(each of length N) to estimate the current filter parameters, while RLS does not rely on the previous input and output(i.e.$\mathbf{x}[k]$ and $y[k]$) but just rely on the previous estimation and current values, which requires less memory.

- FDAF:

FDAF basically transform LMS to frequency domain using a multiply update algorithm by $\mathbf{F}^{-1}$. Then LMS update can be transformed into the below expression:

$$\mathbf{F}^{-1}\underline{\mathbf{w}}[k+1] = \mathbf{F}^{-1}\underline{\mathbf{w}}[k] + 2\alpha\mathbf{F}^{-1}\underline{\mathbf{x}}[k]r[k] \Leftrightarrow \underline{\mathbf{W}}[k+1] = \underline{\mathbf{W}}[k] + \frac{2\alpha}{N}\underline{\mathbf{W}}[k]$$

Futhermore, FDAF uses the Newton update rule instead of SGD by a multiplication with $\mathbf{P}^{-1}$, which corresponding the $\mathbf{R}^{-1}$ in Newton's method. Similarly, $\mathbf{P}^{-1}$ achieves the whitening in Fourier domain, i.e. taking all of elements and making them flat in Fourier domain. In other words, FDAF is equivalent to NLMS with white noise input. With power normalisation, FDAF algorithm can be written as

$$\underline{\mathbf{W}}[k+1] = \underline{\mathbf{W}}[k] + 2\alpha\mathbf{P}^{-1}\underline{\mathbf{W}}[k]$$

With the combination of LMS and Newton, FDAF is more practical than Newton as we do not need to know the statistics, also its convergence is more stable than LMS as it performs whitening on input to eliminate the influence from coloration. Additionally. the computation of FDAF is simple as it takes the advantage of DFT symmertry and the inversion of diagonal PSD matrix.

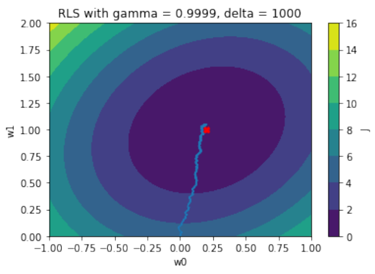
m

The resulting plot is shown in fig 5, and the corresponding core code is shown as above. The forgetting factor γ controls the memory of history data, i.e. the influence from previous data. A larger γ means more memory which leads to a lower sensitivity to the current input, not only to noise but also to the input data, and a smaller γ has the opposite influence. The upgrade does not have any memory if $\gamma \to 0$. In a conclusion, an appropriate forgetting factor can improve convergence.

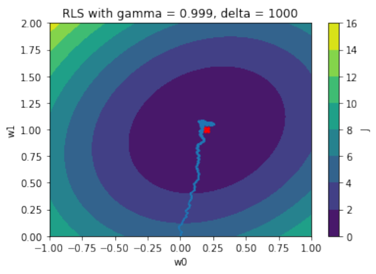
```
def RLS(gamma,w_init,ryx_init,inv_△):
    wk = w_init
    w = []
    inv_Rx = inv_△ * np.identity(3)
    r_yx = ryx_init
    for i in range(N):
        w.append(wk)
        x_k = x_stack[[i]].T
        g = inv_Rx@x_k / (gamma**2 + x_k.T@inv_Rx@x_k)
        inv_Rx = gamma**(-2)*(inv_Rx - g@x_k.T*inv_Rx)
        r_yx = gamma**2*r_yx+x_k*y[i]
        wk = ((inv_Rx@r_yx).T).reshape(-1)
    return np.array(w)
```

n

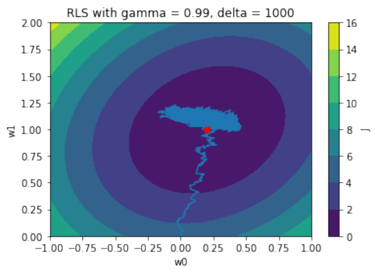
Computational complexity per time upgrade can be quantified as in table 1. It is obvious that LMS is the simplest one as it uses instantaneous estimate of gradient, and NLMS adds normalisation on the basis of LMS. Both RLS and FDAF are more complex than LMS and NLMS as the former two contain matrix calculation with complexity $O(N^2)$, but we are not



(a) $\gamma = 0.9999$



(b) $\gamma = 0.999$



(c) $\gamma = 0.99$

Figure 5: The trajectory of the filter coefficients(RLS)

Table 1: My rank of our methods

	Complexity rank	Accuracy rank
LMS	1(O(2N))	4
NLMS	2(O(3N))	3
RLS	4(O(N^2))	2
FDAF	3(O(N^2))	1

sure which one is better. A guess is that FDAF has lower complexity as the advantage of diagonal matrix and DFT symmetry.

In terms of accuracy, FDAF is supposed to give the best performance as it combines the idea of LMS and Newton, which actually achieve NLMS with white noise input in frequency domain. Although the estimation of **P** may not be exact, input vector with a large length would improve it as frequency bins are "uncorrelated" in this case. Then, the second one is RLS as decorrelation takes place in algorithm and it takes the influence of previous data into account. With the recursion, the convergence would perform well despite this method exhibits unstable roundoff error accumulation. Finally, NLMS has higher accuracy than LMS as it uses normalization to eliminate the influence from input power, while LMS is a simple algorithm trying to "decorrelate" signals and residual.

Familiarisation with Pytorch

Exercise 1

a

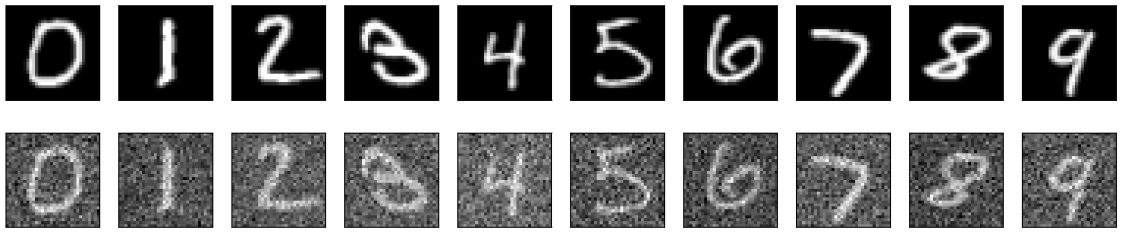


Figure 6: Data samples

b

The FC network with linear layers is defined as below:

```
class LinearNet(nn.Module):
    def __init__(self, img_size, hidden_size1, hidden_size2):
        super(LinearNet, self).__init__()
        self.fc1 = nn.Linear(img_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, img_size)

    def forward(self, x):
        x = x.view(-1, 32*32)
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x
```

A too complex network architecture can introduce the problems of overfitting, gradient vanishing and explosion, and degradation.

- A network model would fit too much on the features of the training set if its architecture is too complex. As the result, it may fail to fit to additional data or predict future observations reliably, i.e. overfitting.
- The gradient would vanish or explode due to the continuous multiplication of small gradients or large gradients in the backpropagation in a too deep network.

- As we increase network depth, accuracy gets saturated. If we still continue to increase the depth, then the accuracy would degraded and even worse than the shallower network because deeper networks lead to higher training error.

c

Optimiser is simply a weight update method according to the difference between target and output. The goal of the optimiser is to tune the parameter used in the network in order to minimize the losses during training.
For SGD (Stochastic Gradient Descent) and Steepest Gradient Descent algorithm:

- Similarity: In both Steepest Gradient Descent algorithm and stochastic gradient descent (SGD), we update a set of parameters in an iterative manner (with gradient) to minimize the loss function.
- Difference: In Steepest Gradient Descent, we have to run through all the samples in the training set to do a single update for a parameter in a particular iteration, while in SGD we only use one or a subset of training sample to do the update procedure. If we use subset, it is called Minibatch Stochastic Gradient Descent.

d

Untrained model will use the default weights to generate the prediction. Refer to Figure 7, the prediction is really bad since the untrained model learned nothing from the data, or say, the prediction is irrelevant with the training data.

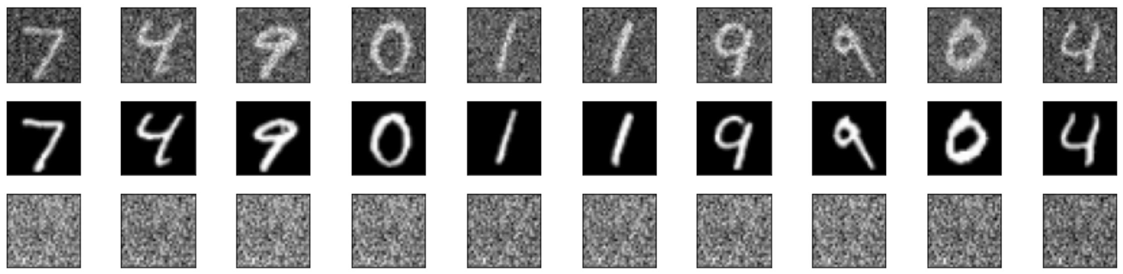


Figure 7: Linear model: untrained prediction

e

The gap between training and validation loss can give an indication of the model performance, e.g., whether overfitting (Huge train/val gap means overfitting) or underfitting (No gap between train/val means underfitting) occurs.

It is also possible to do loss curve analysis to tune the hyper parameter, e.g., loss plateaus means we are using a large learning rate, small gradient at the beginning means a bad initialization, loss still going down means we need to train longer.
In Figuer 8, we found there is no gap between train and validation loss, both val and train saturates at a particular loss value, so the model is underfitting. This is because it doesn't have activations, it's essentially a combined linear model, hence it is not complex enough to model the input information properly, we need to use a more complex model.

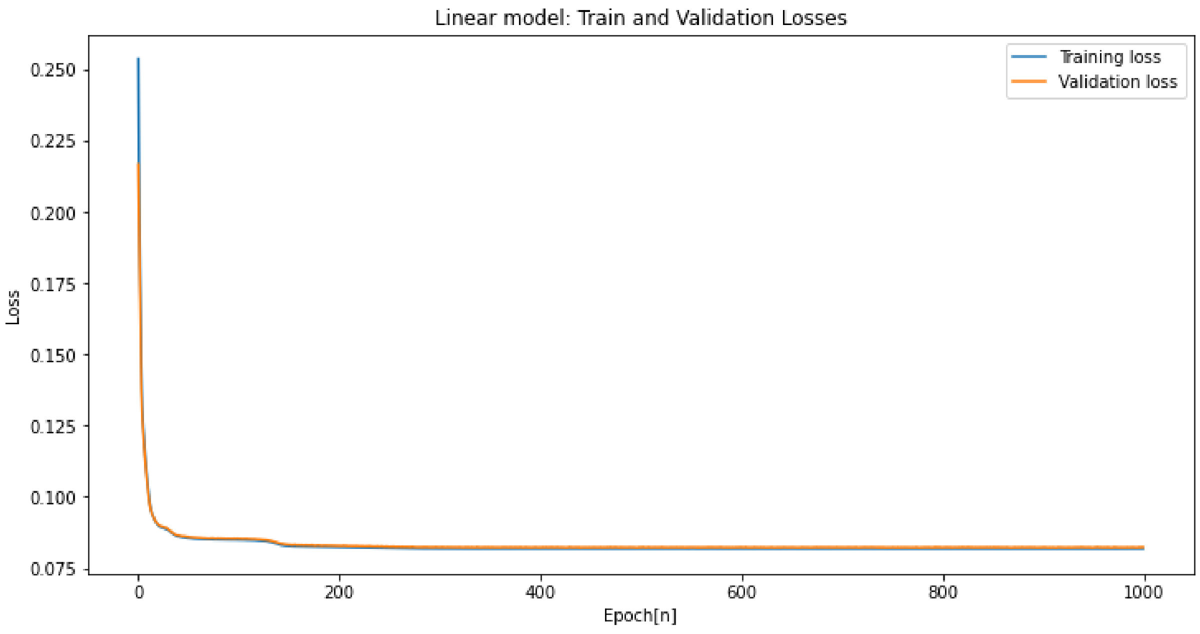


Figure 8: Linear model loss curve

f

The prediction (third row in Figure 9) is much better than what we got from untrained model, while the images are still blurred compared with clean example (third row in Figure 9).

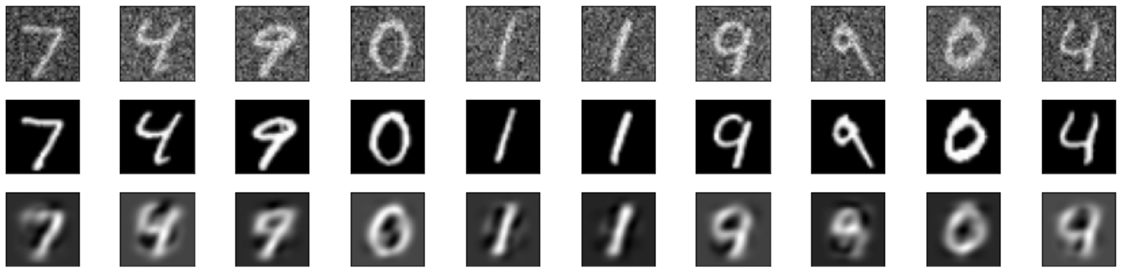


Figure 9: Linear model: trained prediction

Exercise 2

a

Figure 10 shows the visualisation of ReLU function. It matches what we expected: $y = 0, \forall x < 0$ and $y = x, \forall x \geq 0$. The activation function has to be implemented after each Linear layer (except for the last Linear layer, which is the output layer).

```python
def Relu(x):
    a = torch.zeros_like(x)
    return torch.max(x, a)

class NonLinearNet(nn.Module):
    def __init__(self, img_size, hidden_size1, hidden_size2):
        super(LinearNet, self).__init__()
        self.fc1 = nn.Linear(img_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, img_size)

    def forward(self, x):
        x = x.view(-1, 32*32)
        x = self.fc1(x)
        x = Relu(x)
        x = self.fc2(x)
        x = Relu(x)
        x = self.fc3(x)
        return x
```

b

GD have many problems:

- Getting stuck in local minima. This can be relexed by adding momentum, e.g, Adam.
- No independent parameter update, same learning rate for all parameters, makes iteration less efficient. The problem can be solved by AdaGrad and RMSProp. AdaGrad adapts learning rate for individual features, but learning rates decay very quickly. RMSProp impoves AdaGrad by combining the adaptive step size and moving average of the squared gradient.
- Saddle points, all partial derivatives are zero and not minima or maxima in all dimension. In this case, GD would move very slowly. AdaGrad and RMSProp can escape quickly.
- GD is updated with first order, curvature information is not used.

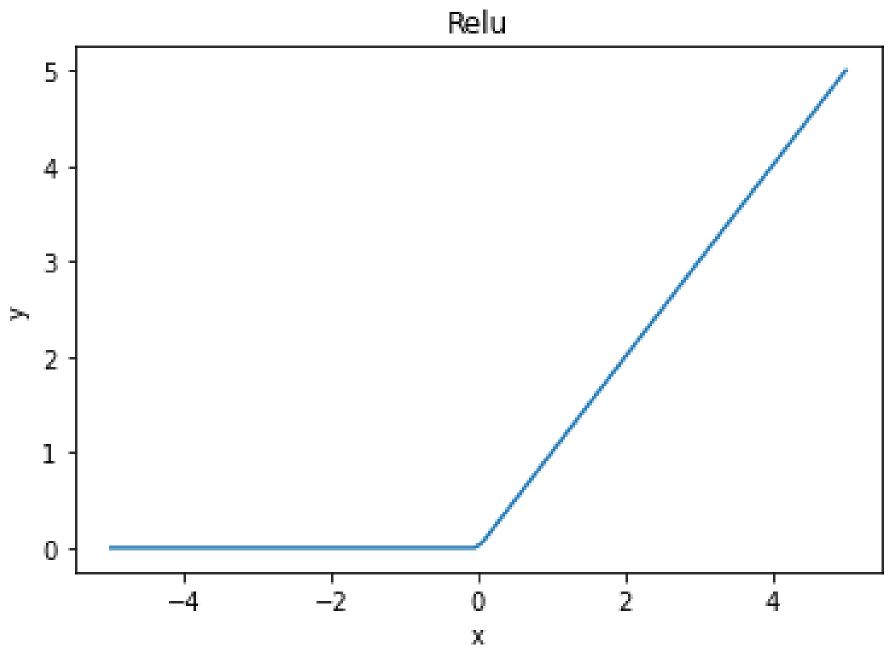


Figure 10: ReLU

In this part, we used Adam instead of SGD. Adam improves RMSProp by introducing a momentum term and a bias correction for the momentum. Another reason is that Adam is relatively easy to configure where the default configuration parameters do well on most problems.

```
learning_rate = 0.001
nonlinear_model = build_model(NonLinearNet())
optimizer = optim.Adam(nonlinear_model.parameters(), lr=learning_rate)
criterion = nn.MSELoss()
train_loss, val_loss = train(nonlinear_model, optimizer, criterion, 1000)
```

c

Figure 11 shows the prediction with untrained non-linear model. As expected, the prediction is still very bad. No matter the model is linear model or not, if no training has taken place, the model would learning nothing from the data, so the prediction will also be irrelevant with data.

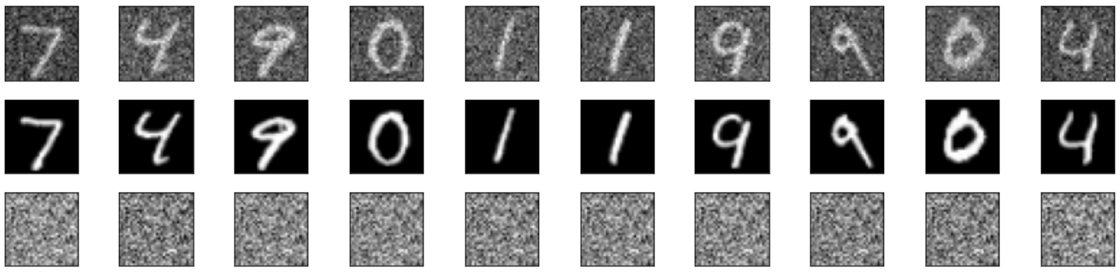


Figure 11: Non-linear model: untrained prediction

d

The loss curve is much more reasonable after adding a non-linear activation.
The results in Figure 12 shows that our model didn't experience overfitting or underfitting, and we don't need to train longer or tune the learning rate.

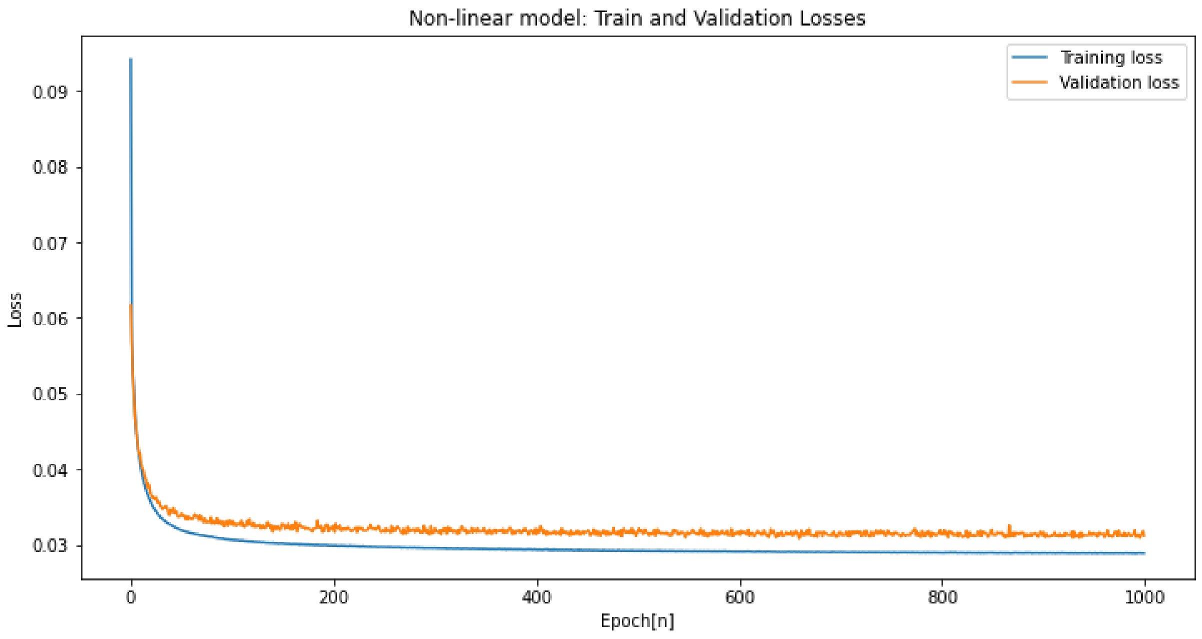


Figure 12: Non-linear model loss curve

e

In Figure 13, the second row is the prediction from Non-linear model, the third is the target, the forth is the prediction from Linear model.
The only differences between two model is that there exists ReLU activation after each Linear layer. The non-linearity introduced by ReLU means the network can no longer be expressed in one linear function, which allows more complex feature could be learned in the hidden layer, resulting in a closer approximation of the target.

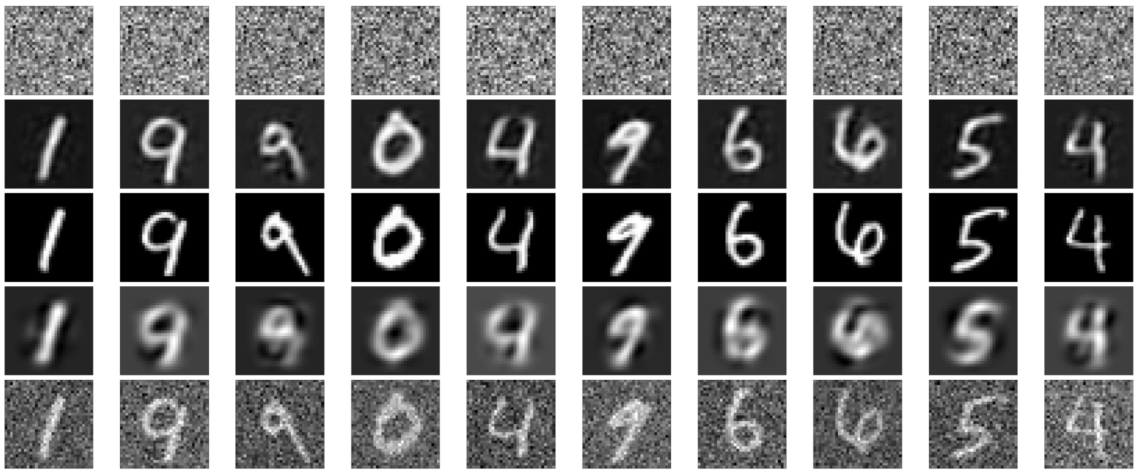


Figure 13: Total comparison

f

The benefits from additional layers for Linear model is limited. In Figure 14, third row and fourth row comes from 4 hidden and 8 hidden layer respectively, however, additional layer doesn't help to improve the lack of non-linearity. The concatenation of linear operations reduces several linear operation to a single linear operation, which just leads to another simple linear model, it naturally can't capture the complexity of the desired feature.

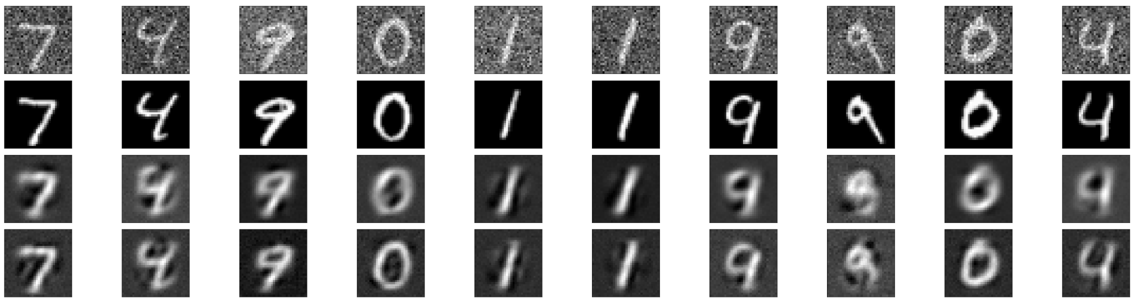


Figure 14: More layers in Linear model