CSC2529-hw5

1010171181 Xinran Zhang

November 2, 2023

1 Task1

First of all, I defined the derivative of the ReLU function as $u(g_i)$, and set $u(0) = 1$. Then, I derived the process of forward and backward propagation, along with the dimensions of the process variables, as shown in figure 1.

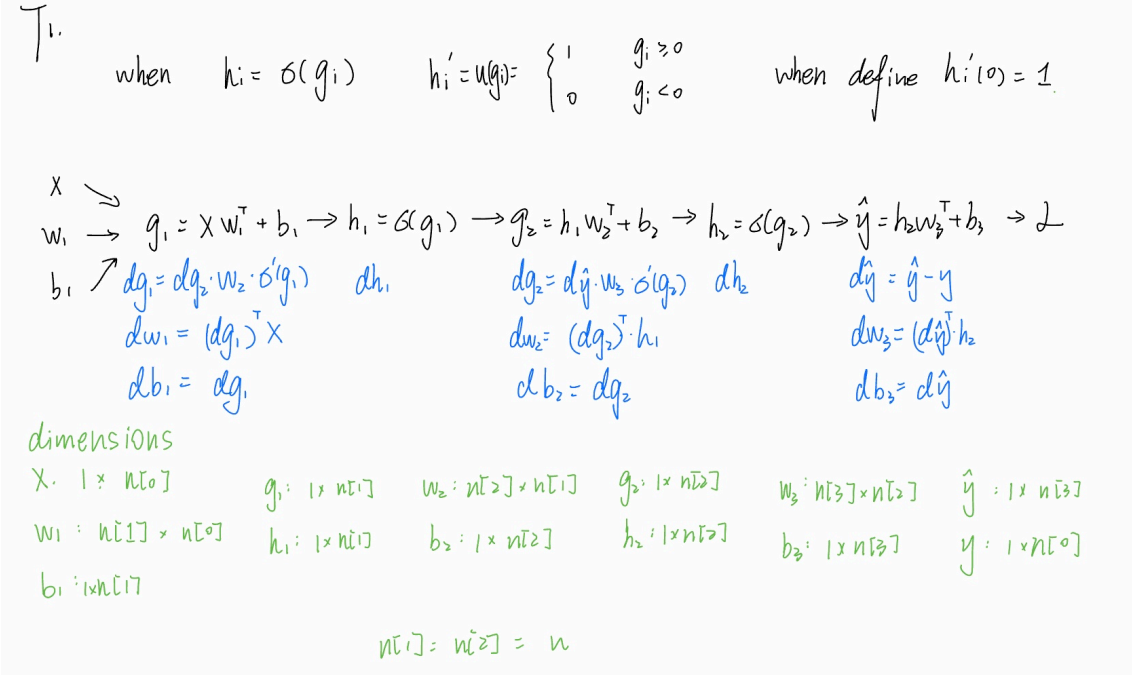


Figure 1: Derivation of Backpropagation

1.1 Q1

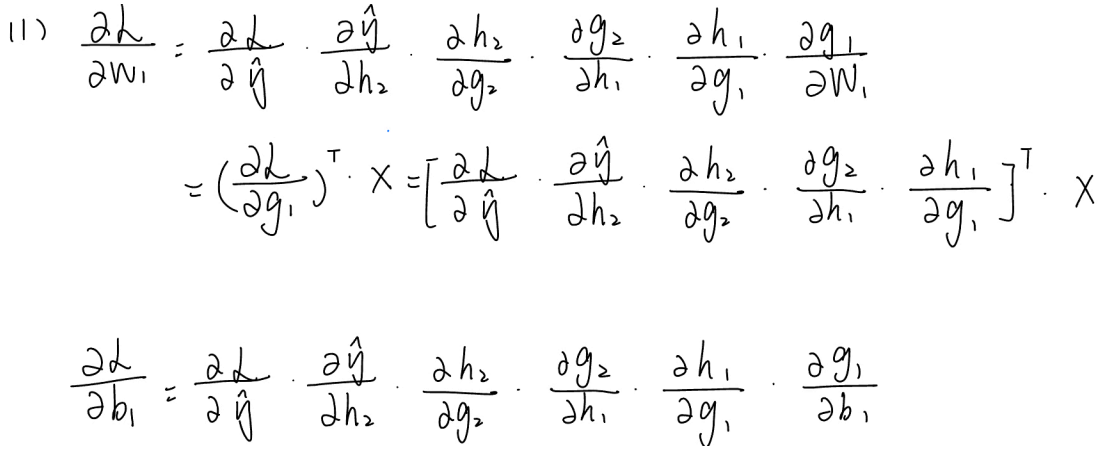


Figure 2: the expression for $\frac{\partial \mathcal{L}}{\partial W1}$ and $\frac{\partial \mathcal{L}}{\partial b1}$

1.2 Q2

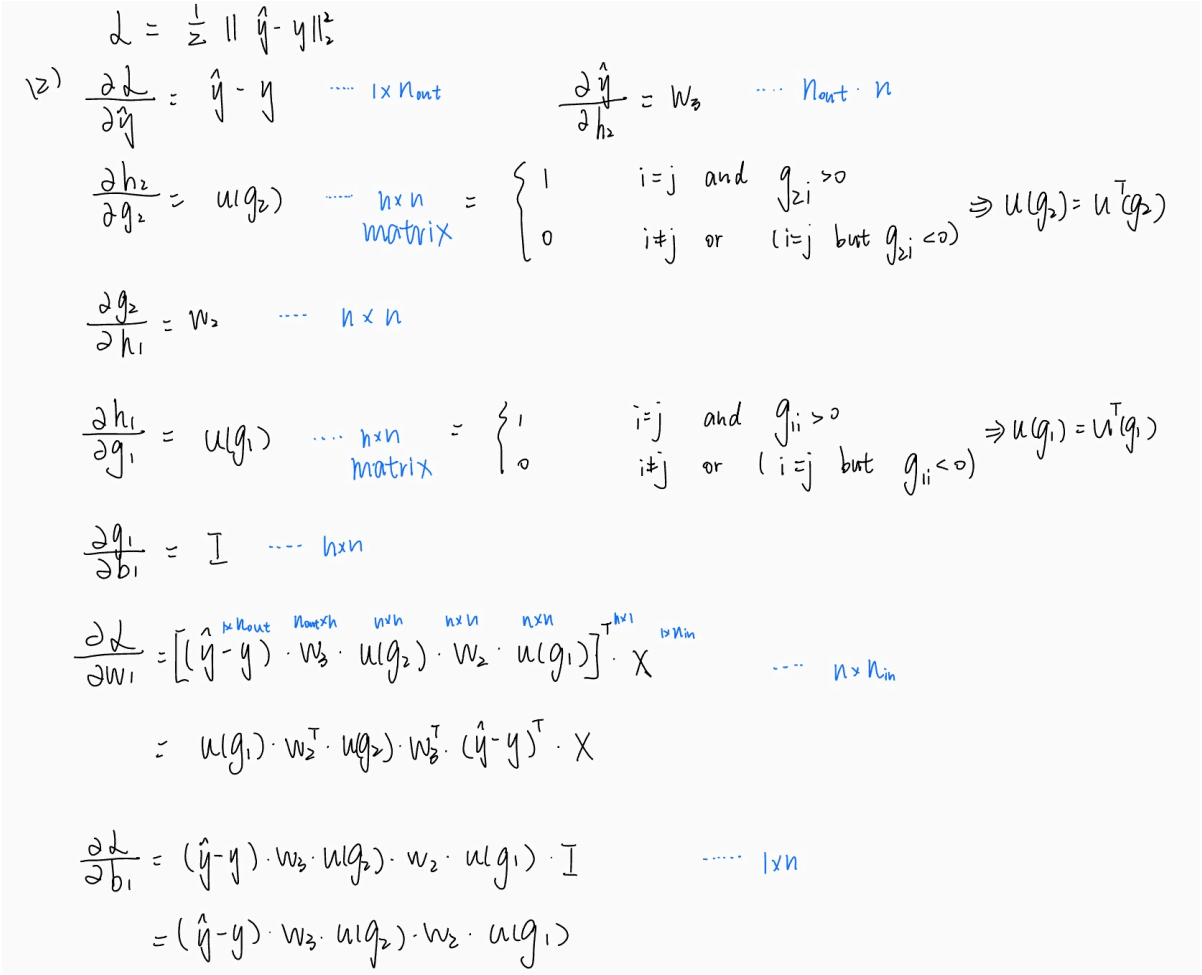


Figure 3: analytical expressions for $\frac{\partial \mathcal{L}}{\partial W1}$ and $\frac{\partial \mathcal{L}}{\partial b1}$

1.3 Q3

α is the learning rate here.

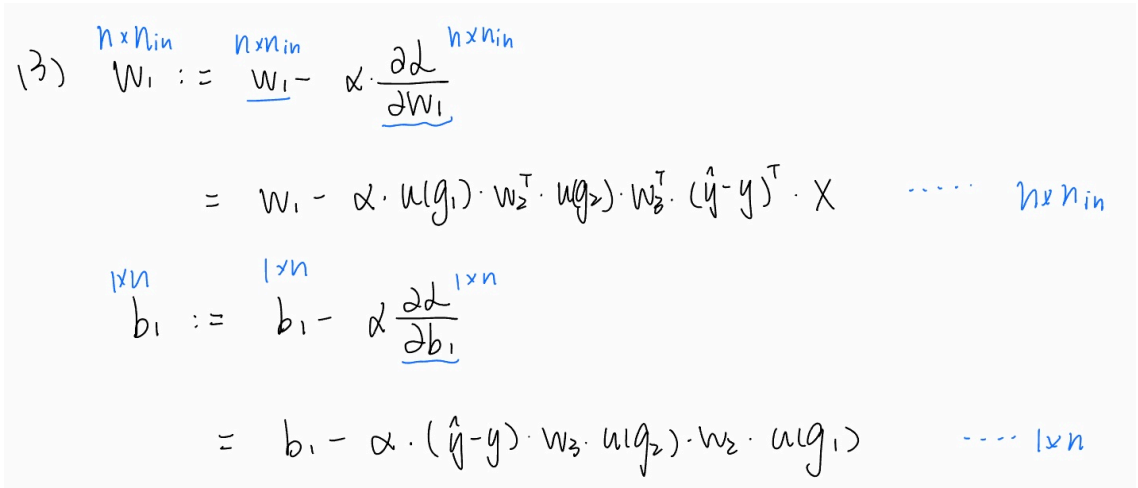


Figure 4: gradient descent update rule for W1 and b1

1.4 Q4

Complete code is shown in Appendix.

1.4.1 (a) automatic differentiation

We have these formulas for backpropagation of linear layer.

$$g_i = h_{i-1}W_i^T + b_i \tag{1}$$

$$grad\_output = \frac{\partial \mathcal{L}}{\partial g_i} \tag{2}$$

$$\frac{\partial \mathcal{L}}{\partial h_i} = \frac{\partial \mathcal{L}}{\partial g_i}\frac{\partial g_i}{\partial h_{i-1}} = grad\_output@\frac{\partial g_i}{\partial h_{i-1}} \tag{3}$$

$$\frac{\partial \mathcal{L}}{\partial h_i} = \frac{\partial \mathcal{L}}{\partial g_i}\frac{\partial g_i}{\partial w_i} = (\frac{\partial \mathcal{L}}{\partial g_i})^T h_{i-1} = grad\_output^T@h_{i-1} \tag{4}$$

$$\frac{\partial \mathcal{L}}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial g_i} = grad\_output \tag{5}$$

Here is a piece of **code** to realize it.

```
grad_input =grad_output @ weight # 1*n
grad_weight =grad_output.T @ input # n*n
grad_bias =grad_output # 1*n
```

Also, here are the formulas for ReLU layer.

$$h_i = \sigma(g_i) \tag{6}$$

$$grad\_output = \frac{\partial \mathcal{L}}{\partial h_i} \tag{7}$$

$$\frac{\partial \mathcal{L}}{\partial g_i} = \frac{\partial \mathcal{L}}{\partial h_i}\frac{\partial h_i}{\partial g_i} = grad\_output@\frac{\partial h_i}{\partial g_i} \tag{8}$$

In realization of backpropagation of ReLU layer, I initially planned to create new matrix as the derivation of ReLU (theoretically, the derivation $\frac{\partial h_i}{\partial g_i}$ is a n*n matrix). However, in the test examples, there are several batches for one example. Therefore, creating a matrix is not a good choice.

Then I cloned the grad_output and added some changes on the cloned matrix. In this way, we can confirm the consistency of dimensions. Here is the **code** to realize ReLU backpropagation.

```
grad_input =grad_output.clone() #
grad_input[input <0] =0
```

1.4.2 (b) compute gradients analytically

As is shown in 1.2, we use the analytical expressions for each of the partial derivative terms to compute the gradients analytically. Here is the piece of **code**.

```
# TODO: write your analytical expression for dloss/dW1 from the gradient descent
# update in part 3
# HINT: your expression will depend on the variables W3, g2, W2, g1, and x
g2_g =g2.clone()
g2_g[g2 >=0] =1
g2_g[g2 <0] =0
g1_g =g1.clone()
g1_g[g1 >=0] =1
g1_g[g1 <0] =0

W1_grad =((yhat -y) @ W3 *g2_g @ W2 *g1_g).T @ x # replace this line

# TODO: write your analytical expression for dloss/db1 from the gradient descent
# update in part 3
# HINT: your expression will depend on the variables W3, g2, W2, g1, and x

b1_grad =(yhat -y) @ W3 *g2_g @ W2 *g1_g # replace this line
```

Although both of these two methods are based on chain rule, there are still some differences between them. Computing gradients analytically, we need to perform a large number of matrix operations, for each weight and each bias which will reduce efficiency. For example, when the model is large with a large number of layers and a large number of parameters, the computation will be very expensive.

As to automatic differentiation, it is more suitable for large model, working through the network's layers to calculate gradients accurately and quickly. Also, we can use the gradient of previous layer to calculate gradient of current layer, which means we can reuse some of the results.

Concurrently, automatic differentiation need more memory because we need store the gradients in the process of forward propagation.

In conclusion, I think automatic differentiation is more efficiently.

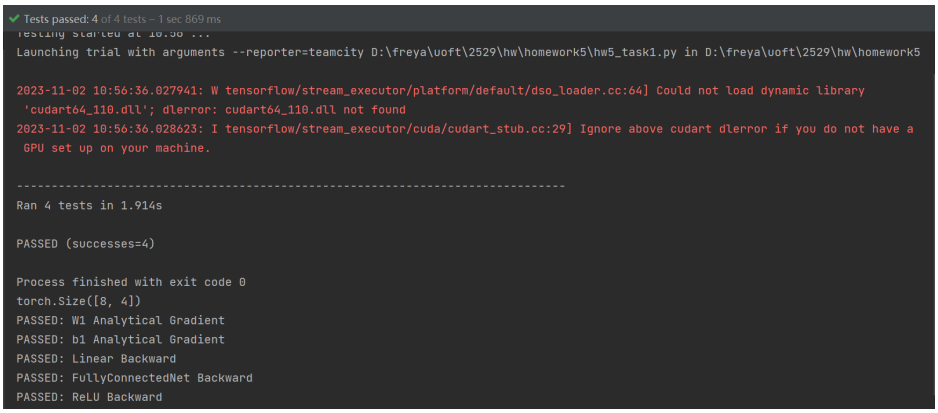


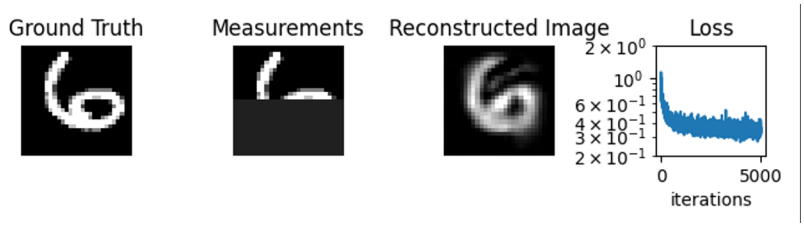
Figure 5: All test passed

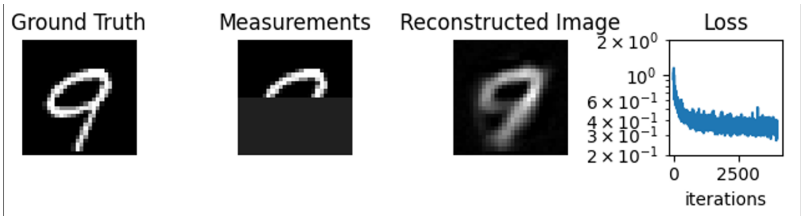
1.4.3 (c)

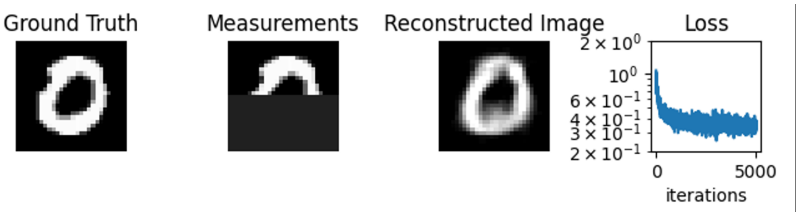
In this part, I use gradient descent to train the network for an image inpainting task. Here is the **code** of gradient decent.

```
        # TODO: write the gradient descent update rule
        # HINT: update p.data using the learning rate and the
        # gradient stored in p.grad
        with torch.no_grad():
            for p in net.parameters():
                # raise NotImplementedError('Need to write gradient descent rule') # remove this line
                p.data =p.data -lr *p.grad # write gradient descent update rule here
```

Here are some resulting pictures.







2 Task2

I trained four models separately based on the presence or absence of bias and different hidden channels when sigma was set to 0.1. Subsequently, I evaluated the models using PSNR as a metric across three levels of sigma [0.05, 0.1, 0.2].

Here are the results.

use bias?	hidden channels	PSNR(dB) sigma=0.05	sigma=0.1	sigma=0.2
√	32	31.614	29.581	22.092
√	64	31.233	**29.942**	21.198
×	32	32.260	29.212	24.944
×	64	**33.229**	29.734	**25.644**

Table 1: PSNRs of adding bias or not and different hidden channels

From results in table1, we can conclude some results. Initially, the four models trained with sigma=0.1 demonstrate similar performance when re-evaluated at sigma=0.1.

Secondly, all four models trained with sigma=0.1 exhibit better performance when sigma=0.05. Hence, we can infer that a model trained on higher noise levels can also effectively handle lower noise levels. Conversely, these same four models show decreased performance when sigma increases to 0.2, indicating that models trained on lower noise levels may not effectively handle higher noise levels.

When considering biases, at sigma=0.05 or sigma=0.2, models without biases outperform models with biases.

I think this kind of difference arises from the models being trained at sigma=0.1, potentially resulting in overfitting of the biases to the specific models and rendering them unsuitable for other sigma values.

Finally, in terms of hidden channels, with a few exceptions (such as sigma=0.2 with bias), models with more hidden channels usually outperform those with fewer hidden channels. In my interpretation, models with more hidden channels possess superior explanatory capabilities.

Here is a piece of code. Complete code is shown in the appendix.

```
# TODO: Your code goes here!
################################################################################

# use the 'train' function with proper parameters for using biases and the
# number of hidden layers
model1 =train(sigma=0.1, use_bias=True, hidden_channels=32, epochs=2, batch_size=32, plot_every=200)
model2 =train(sigma=0.1, use_bias=True, hidden_channels=64, epochs=2, batch_size=32, plot_every=200)
model3 =train(sigma=0.1, use_bias=False, hidden_channels=32, epochs=2, batch_size=32, plot_every=200)
model4 =train(sigma=0.1, use_bias=False, hidden_channels=64, epochs=2, batch_size=32, plot_every=200)
# after training pass the model to the 'evaluate_model' function to run
# on the validation dataset
for noise in [0.05, 0.1, 0.2]:
    psnr1 =evaluate_model(model1, sigma=noise, output_filename=f'out_bias_32_{noise}.png')
    psnr2 =evaluate_model(model2, sigma=noise, output_filename=f'out_bias_64_{noise}.png')
    psnr3 =evaluate_model(model3, sigma=noise, output_filename=f'out_nobias_32_{noise}.png')
    psnr4 =evaluate_model(model4, sigma=noise, output_filename=f'out_nobias_64_{noise}.png')
    print(f'noise={noise}', psnr1, psnr2, psnr3, psnr4)
```

```
C:\Users\Freya\AppData\Local\Programs\Python\Python39\python.exe D:/freya/uoft/2529/hw/homework5/hw5_task2.py
==> Training on noise level 0.10 | use_bias: True | hidden_channels: 32
2023-11-02 12:30:17.338078: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dyn
 .dll not found
2023-11-02 12:30:17.338539: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror i
100%|██████████| 1874/1875 [01:55<00:00, 16.27it/s]
==> Training on noise level 0.10 | use_bias: True | hidden_channels: 64
100%|██████████| 1874/1875 [05:31<00:00,  5.66it/s]
==> Training on noise level 0.10 | use_bias: False | hidden_channels: 32
100%|██████████| 1874/1875 [01:50<00:00, 16.99it/s]
==> Training on noise level 0.10 | use_bias: False | hidden_channels: 64
100%|██████████| 1874/1875 [03:36<00:00,  8.66it/s]
noise=0.05 31.61446836701397 31.232611315474575 32.25992329897346 33.229025812446395
noise=0.1 29.581042575451853 29.942123863722564 29.212180697904774 29.733673343182897
noise=0.2 22.0915820686448 21.19828432973995 24.94408565663587 25.64441035952183
```

Figure 6: 4 models trained

3 Task3

In this task, we evaluate and compare 3 different methods. Complete code is shown in appendix.

3.1 Q1

Evaluate the performance of each approach by computing the average PSNR across the whole validation set images for three different noise levels: $\sigma \in \{0.005, 0.01, 0.02\}$. Here are the resulting PSNRs .

	PSNR(dB)		
	sigma=0.005	sigma=0.01	sigma=0.02
method1 (Wiener)	**32.172**	28.146	22.862
method2 (Neural Network)	29.232	28.585	25.351
method3 (Wiener+NN)	30.997	**30.374**	**27.641**

Table 2: PSNRs of using different denoising methods

Here is a piece of code. Complete code is attached in Appendix.

```
for sigma in [0.005, 0.01, 0.02]:
    psnr1 =0
    psnr2 =0
    psnr3 =0
    i =0
    for image, gt, kernel in dataset:
        img_noise =image +torch.randn_like(image) *sigma
        result_wiener =wiener_deconv(img_noise, kernel)
        result_net1 =model_deblur_denoise(img_noise)
        result_net2 =model_denoise(result_wiener)

        # save the psnrs
        psnr1 +=calc_psnr(result_wiener, gt)
        psnr2 +=calc_psnr(result_net1, gt)
        psnr3 +=calc_psnr(result_net2, gt)
        i +=1
        # save out sample images to include in your writeup
        if i ==10 or i ==24:
            skimage.io.imsave(f'original_image_{i}.png', (img_to_numpy(img_noise) *255).astype(np.uint8))
            skimage.io.imsave(f'image_{i}_method1_sigma={sigma}.png',
                              (img_to_numpy(result_wiener) *255).astype(np.uint8))
            skimage.io.imsave(f'image_{i}_method2_sigma={sigma}.png',
                              (img_to_numpy(result_net1) *255).astype(np.uint8))
            skimage.io.imsave(f'image_{i}_method3_sigma={sigma}.png',
                              (img_to_numpy(result_net2) *255).astype(np.uint8))
            print(i, f'when sigma = {sigma}', psnr1 /i, psnr2 /i, psnr3 /i)

    print(f'when sigma = {sigma}', psnr1 /i, psnr2 /i, psnr3 /i)
```

Also, I will show some image outputs of each method and their average PSNR across validation set images before this image. The first row of images includes: the blurred images from the validation set, and images with 3 different levels of added noise.

	PNSR(dB)		
	sigma=0.005	sigma=0.01	sigma=0.02
			
method1 (Wiener)	**32.303**	28.215	22.883
			
method2 (Neural Network)	29.347	28.755	25.475
			
method3 (Wiener+NN)	31.380	**30.723**	**27.878**
			

Table 3: PSNRs and image outputs of image10

	PNSR(dB) sigma=0.005	sigma=0.01	sigma=0.02
			
method1 (Wiener)	**32.151**	28.146	22.861
			
method2 (Neural Network)	29.094	28.493	25.368
			
method3 (Wiener+NN)	30.813	**30.234**	**27.657**
			

Table 4: PSNRs and image outputs of image24

3.2 Q2

When σ = **0.005**, Wiener deconvolution has best performance. I think this is because Wiener's reliance on SNR.

$$H^{'} = \frac{1}{H} * \frac{|H|^2}{|H|^2 + \frac{1}{SNR}} \tag{9}$$

With smaller noise, the resulting higher SNR leads to a smaller damping factor ($\frac{1}{SNR}$), ultimately yielding a larger H', signifying the heightened strength of the Wiener filter.

Wiener is most effective in scenarios where the signal greatly outweighs the noise. Therefore, in low-noise settings, the Wiener filter outperforms the other two methods. However, as noise increases, the efficacy of the Wiener filter weakens, and it cannot surpass the performance of the Neural network. This is primarily due to Wiener's stronger emphasis on deblurring convolution blur, limiting its effectiveness in higher noise environments.

3.3 Q3

When σ =**0.01** or **0.02**, Neural Network outperform Wiener deconvolution. Method 3, Wiener + Neural Network, has a better performance than method2 in these two scenarios.

When the noise level increases, the capability of the Wiener filter becomes constrained. However, the neural network can learn and denoise the noisy components, hence achieving better performance.

In addition, method three combines both the Wiener filter and neural network approaches, utilizing the neural network on the results obtained by the Wiener filter. As a result, method three not only utilizes the Wiener filter to eliminate the blurring caused by convolution but also applies the neural network to learn and denoise additional noise components. This combined strategy achieves the best performance.

A task1

```
import torch
import torch.nn as nn
from torch.autograd import Function
from torch.autograd import gradcheck
from torchvision.datasets import MNIST
from torch.utils.data import DataLoader
from torchvision.transforms import Compose, ToTensor, Normalize, Lambda
from torchvision.utils import make_grid
import unittest
import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm

import warnings

warnings.filterwarnings("ignore", "UserWarning")

"""
HW 5 Task 1 programming exercise

This exercise implements a small fully-connected neural network from scratch.
We implement custom layers with forward and backward passes for Linear and ReLU
layers, learn how to use autograd to evaluate gradients after a forward pass,
and use gradient descent to train the network to inpaint an image

Specifically you will need to complete the following tasks.

1. Fill in the code to calculate the derivatives for the backward pass for the
   LinearFunction and ReLUFunction.

2. Check your analytical gradient solutions from Task 1 part 3 of the homework
   against the gradients calcualated by AutoGrad. You will need to fill in the
   indicated portions of the "check_analytical" function. The gradients should
   match!

   How does autograd save computation in calculating these gradients
   compared to calculating the analytical gradients for W1 and b1 separately?

3. Use gradient descent to train your network to overfit an image inpainting
   task and learn to inpaint the missing values of an image. Complete and run the
   "train_network" function to do this.

"""


class LinearFunction(Function):

    @staticmethod
    def forward(ctx, input, weight, bias):
        # we will save the input, weight, and bias to help us calculate the
        # gradients in the backward pass
        ctx.save_for_backward(input, weight, bias)

        # return the output of the linear layer
        return input.mm(weight.T) +bias[None, :]

    @staticmethod
    def backward(ctx, grad_output):
        # retrieve the saved variables from the context
        input, weight, bias =ctx.saved_tensors

        ###########################################################################
        # TODO: complete these lines, replacing "None" with the correct
        # calculations
        ###########################################################################

        # We need to return the gradients with respect to the input, the weight
        # matrix, and the bias vector.
        #
        # This can be done in two steps. (1) We need to compute the gradient of
        # the layer output with respect to the input, the weight matrix, and the
```

```
        # bias vector. These correspond to partial derivative terms in the chain
        # rule that you already derived in your homework. (2) We need to
        # multiply this by the "upstream" gradient (grad_output), which is the
        # result of multiplying all the previous terms in the chain rule that
        # have already been calculated in the backward pass (starting at the
        # loss function and flowing backwards).
        #
        # The backward function then returns each of these gradient values. The
        # gradients with respect to the weight and bias parameters will be used
        # for updating the parameters during training, and the value returned
        # for grad_input will become the new grad_output for the next term in
        # the chain rule as we continue the backward pass.

        grad_input =grad_output @ weight # 1*n
        grad_weight =grad_output.T @ input # n*n
        grad_bias =grad_output # 1*n
        ##########################################################################

        return grad_input, grad_weight, grad_bias


class Linear(nn.Module):
    def __init__(self, input_features, output_features):
        super(Linear, self).__init__()

        self.input_features =input_features
        self.output_features =output_features

        self.weight =nn.Parameter(torch.empty(output_features, input_features))
        self.bias =nn.Parameter(torch.empty(output_features))

        # initialize weights
        self.weight.data.uniform_(-0.1, 0.1)
        self.bias.data.uniform_(-0.1, 0.1)

    def forward(self, input):
        return LinearFunction.apply(input, self.weight, self.bias)


class ReLUFunction(Function):

    @staticmethod
    def forward(ctx, input):
        # we will save the input, weight, and bias to help us calculate the
        # gradients in the backward pass
        ctx.save_for_backward(input)

        # return the output of the linear layer
        return torch.clamp(input, 0)

    @staticmethod
    def backward(ctx, grad_output):
        # retrieve the saved variables from the context
        input, =ctx.saved_tensors

        ##########################################################################
        # TODO: complete these lines, replacing "None" with the correct
        # calculations
        ##########################################################################

        # The gradient with respect to the input will become the new
        # "grad_output" in the next backward operation as we continue the
        # backward pass. Also see comments above in the LinearFunction part of
        # the homework.
        # grad_output ... 1*n
        # matrix = np.zeros((len(grad_output), len(grad_output)))
        # print(grad_output.shape)
        # for i in range(len(grad_output)):
        #     for j in range(len(grad_output)):
        #         if i == j and input[i] >= 0:
        #             matrix[i][j] = 1
        #
        # grad_input = grad_output @ matrix # 1*n
        # print(grad_output.shape, matrix.shape, grad_output.shape)
```

```
        # error: there are some batches here, we need to consider the dimensions of batches in each test
        # gradient of relu == gradient_output * relu'
        # relu' means when the input value>0 ==> relu'=1, else relu'=0
        # therefore the dimension of grad_input should be the same with that of grad_output
        # print(grad_output.shape, input.shape)
        grad_input =grad_output.clone() #
        grad_input[input <0] =0
        # print(grad_input.shape)
        ###########################################################################

        return grad_input


class ReLU(nn.Module):
    def __init__(self):
        super(ReLU, self).__init__()

    def forward(self, input):
        return ReLUFunction.apply(input)


class FullyConnectedNet(nn.Module):
    def __init__(self, din=4, dout=3, n=8):
        super(FullyConnectedNet, self).__init__()

        self.din =din
        self.dout =dout
        self.n =n

        self.L1 =Linear(self.din, self.n)
        self.N1 =ReLU()
        self.L2 =Linear(self.n, self.n)
        self.N2 =ReLU()
        self.L3 =Linear(self.n, self.dout)

    def forward(self, x):
        self.g1 =self.L1(x)
        self.h1 =self.N1(self.g1)
        self.g2 =self.L2(self.h1)
        self.h2 =self.N2(self.g2)
        self.yhat =self.L3(self.h2)

        return self.yhat


def check_analytical_gradients():
    """
    Now that you've implemented autograd in PyTorch, let's see how we can use
    this to easily calculate gradients for all the network weights

    """

    # create the fully connected neural network
    net =FullyConnectedNet()

    # use double precision for our numerical checks
    net.double()

    # declare the input and "ground truth" output (arbitrary) we will compare to
    x =torch.randn(1, net.din, dtype=torch.double)
    y =torch.rand(1, net.dout, dtype=torch.double)

    # run the forward pass through the network
    yhat =net(x)

    # we'll use MSE loss
    loss =1 /2 *torch.sum((y -yhat) **2)

    # running .backward() will calculate the gradient of all the weights and
    # biases with respect to the loss. These are stored in the .grad property of
    # each parameter
    loss.backward()

    # Grab all the values we'll need to calculate the gradients analytically
```

```
    W3 =net.L3.weight
    g2 =net.g2
    W2 =net.L2.weight
    g1 =net.g1
    W1 =net.L1.weight
    b1 =net.L1.bias
    print(W1.shape)
    ###############################################################################
    # TODO: write your analytical expression for dloss/dW1 from the gradient descent
    # update in part 3
    # HINT: your expression will depend on the variables W3, g2, W2, g1, and x
    g2_g =g2.clone()
    g2_g[g2 >=0] =1
    g2_g[g2 <0] =0
    g1_g =g1.clone()
    g1_g[g1 >=0] =1
    g1_g[g1 <0] =0

    W1_grad =((yhat -y) @ W3 *g2_g @ W2 *g1_g).T @ x # replace this line

    # TODO: write your analytical expression for dloss/db1 from the gradient descent
    # update in part 3
    # HINT: your expression will depend on the variables W3, g2, W2, g1, and x

    b1_grad =(yhat -y) @ W3 *g2_g @ W2 *g1_g # replace this line

    ###############################################################################

    # check to make sure that it matches autograd
    W1_autograd =W1.grad
    b1_autograd =b1.grad

    assert torch.allclose(W1_grad, W1_autograd), \
        'FAILED: Incorrect W1 analytical gradient!'
    print('PASSED: W1 Analytical Gradient')

    assert torch.allclose(b1_grad, b1_autograd), \
        'FAILED: Incorrect b1 analytical gradient!'
    print('PASSED: b1 Analytical Gradient')


def train_network(lr=2):
    # set up inpainting dataset, normalize by mean and std
    mnist_mean =0.1307
    mnist_std =0.3081
    dataset =MNIST('./', train=True, download=True,
                   transform=Compose([Lambda(lambda x: np.array(x)),
                                      ToTensor(),
                                      Normalize((mnist_mean,), (mnist_std,)),
                                      Lambda(lambda x: torch.flatten(x))
                                      ]))

    dataloader =DataLoader(dataset, batch_size=32, shuffle=True)

    # instantiate neural network
    net =FullyConnectedNet(784, 784, n=32)

    # overfit the network to inpaint this image using the ground truth
    losses =[]
    iteration =0
    N_epochs =3

    def mask_pixels(x):
        x =x.reshape(-1, 28, 28)
        x[:, 14:, :]=0.
        return x.reshape(-1, 28 **2)

    pbar =tqdm(total=N_epochs *len(dataloader))
    for epoch in range(N_epochs):
        for idx, (y, _) in enumerate(dataloader):

            # mask out pixels, we'll need to inpaint them
            x =y.clone()
            x =mask_pixels(x)
```

```
        # run a forward pass through the network
        yhat =net(x)

        # calculate the loss
        loss =torch.mean((yhat -y) **2)
        losses.append(loss.item())

        # run backward pass to calculate gradients for all the parameters
        loss.backward()

        ###########################################################################
        # TODO: write the gradient descent update rule
        # HINT: update p.data using the learning rate and the
        # gradient stored in p.grad
        with torch.no_grad():
            for p in net.parameters():
                # raise NotImplementedError('Need to write gradient descent rule') # remove this line
                p.data =p.data -lr *p.grad # write gradient descent update rule here

        ###########################################################################

        # set the gradients to zero (otherwise they will be accumulated in the
        # .grad array of each parameter during the iterations)
        net.zero_grad()

        if iteration % 1000 ==0:
            plt.ion()
            plt.clf()

            # plot ground truth
            plt.subplot(141)
            y_plot =(y[0].reshape(28, 28).detach().cpu().numpy() *mnist_std) +mnist_mean
            plt.imshow(y_plot, aspect='equal', cmap='gray')
            plt.clim((0, 1))
            plt.axis('off')
            plt.title('Ground Truth')

            # plot measurements
            plt.subplot(142)
            x_plot =(x[0].reshape(28, 28).detach().cpu().numpy() *mnist_std) +mnist_mean
            plt.imshow(x_plot, aspect='equal', cmap='gray')
            plt.clim((0, 1))
            plt.axis('off')
            plt.title('Measurements')

            # plot reconstructed image
            plt.subplot(143)
            yhat_plot =(yhat[0].reshape(28, 28).detach().cpu().numpy() *mnist_std) +mnist_mean
            plt.imshow(yhat_plot, aspect='equal', cmap='gray')
            plt.clim((0, 1))
            plt.axis('off')
            plt.title('Reconstructed Image')

            # plot loss
            plt.subplot(144)
            plt.plot(losses)
            plt.yscale('log')
            plt.title('Loss')
            plt.xlabel('iterations')
            plt.ylim((2e-1, 2))
            plt.gca().set_aspect(1. /plt.gca().get_data_ratio())

            plt.tight_layout()
            plt.pause(0.1)

        iteration +=1
        pbar.update(1)

# print some examples
N_examples =10
list_y =[]
list_x =[]
list_yhat =[]
```

```
    dataloader =DataLoader(dataset, batch_size=N_examples, shuffle=True)

    y, _ =iter(dataloader).next()

    # mask out pixels, we'll need to inpaint them
    x =y.clone()
    x =mask_pixels(x)

    # run a forward pass through the network
    list_y.append(y.reshape(-1, 1, 28, 28))
    list_yhat.append(net(x).reshape(-1, 1, 28, 28))
    list_x.append(x.reshape(-1, 1, 28, 28))

    # plot reconstructed image
    images =torch.cat([*list_y, *list_x, *list_yhat], dim=0) *mnist_std +mnist_mean
    grid =make_grid(images, nrow=N_examples, padding=2, pad_value=1)
    grid =(grid.detach().cpu().numpy()[0])
    plt.imshow(grid, aspect='equal', cmap='gray')
    plt.clim((0, 1))
    plt.axis('off')
    plt.title('Reconstructed Images')


class HW5Checker(unittest.TestCase):

    def test_linear(self):
        input =(torch.randn(64, 16, dtype=torch.double, requires_grad=True),
                torch.randn(32, 16, dtype=torch.double, requires_grad=True),
                torch.randn(32, dtype=torch.double, requires_grad=True))

        result =gradcheck(LinearFunction.apply, input, eps=1e-6, atol=1e-4,
                          raise_exception=False)
        assert result, "Incorrect Linear backward pass"
        print('PASSED: Linear Backward')

    def test_relu(self):
        input =(torch.randn(64, 16, dtype=torch.double, requires_grad=True))

        result =gradcheck(ReLUFunction.apply, input, eps=1e-6, atol=1e-4,
                          raise_exception=False)
        assert result, "Incorrect ReLU backward pass"
        print('PASSED: ReLU Backward')

    def test_net(self):
        net =FullyConnectedNet()
        net.double()
        input =torch.randn(16, net.din, dtype=torch.double).requires_grad_()

        result =gradcheck(net, (input,), eps=1e-6, atol=1e-4, raise_exception=False)
        assert result, "Incorrect FullyConnectedNet backward pass"
        print('PASSED: FullyConnectedNet Backward')

    def test_analytical_grad(self):
        check_analytical_gradients()


def check_part_a():
    suite =unittest.TestSuite()
    suite.addTest(HW5Checker('test_linear'))
    suite.addTest(HW5Checker('test_relu'))
    suite.addTest(HW5Checker('test_net'))

    result =unittest.TextTestRunner(verbosity=0).run(suite)
    if len(result.failures) ==0:
        print('ALL TESTS PASSED')
        return True
    else:
        return False


def check_part_b():
    suite =unittest.TestSuite()
    suite.addTest(HW5Checker('test_analytical_grad'))
```

```
    result =unittest.TextTestRunner(verbosity=0).run(suite)
    if len(result.failures) ==0:
        print('ALL TESTS PASSED')
        return True
    else:
        return False


if __name__ =='__main__':
    check_part_a()
    check_part_b()
    train_network()
```

B task2

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from glob import glob
import os
import skimage.io
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
from tqdm import tqdm

# set random seeds
torch.manual_seed(1)
torch.use_deterministic_algorithms(True)

matplotlib.rcParams['figure.raise_window'] =False

device =torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


class BSDS300Dataset(Dataset):
    def __init__(self, root='./BSDS300', patch_size=32, split='train', use_patches=True):
        files =sorted(glob(os.path.join(root, 'images', split, '*')))

        self.use_patches =use_patches
        self.images =self.load_images(files)
        self.patches =self.patchify(self.images, patch_size)
        self.mean =torch.mean(self.patches)
        self.std =torch.std(self.patches)

    def load_images(self, files):
        out =[]
        for fname in files:
            img =skimage.io.imread(fname)
            if img.shape[0] >img.shape[1]:
                img =img.transpose(1, 0, 2)
            img =img.transpose(2, 0, 1).astype(np.float32) /255.
            out.append(torch.from_numpy(img))
        return torch.stack(out)

    def patchify(self, img_array, patch_size):
        # create patches from image array of size (N_images, 3, rows, cols)
        patches =img_array.unfold(2, patch_size, patch_size).unfold(3, patch_size, patch_size)
        patches =patches.reshape(patches.shape[0], 3, -1, patch_size, patch_size)
        patches =patches.permute(0, 2, 1, 3, 4).reshape(-1, 3, patch_size, patch_size)
        return patches

    def __len__(self):
        if self.use_patches:
            return self.patches.shape[0]
        else:
            return self.images.shape[0]

    def __getitem__(self, idx):
        if self.use_patches:
```

```
            return self.patches[idx]
        else:
            return self.images[idx]


class DnCNN(nn.Module):
    """
    Network architecture from this reference. Note that we omit batch norm
    since we are using a shallow network to speed up training times.

    @article{zhang2017beyond,
      title={Beyond a {Gaussian} denoiser: Residual learning of deep {CNN} for image denoising},
      author={Zhang, Kai and Zuo, Wangmeng and Chen, Yunjin and Meng, Deyu and Zhang, Lei},
      journal={IEEE Transactions on Image Processing},
      year={2017},
      volume={26},
      number={7},
      pages={3142-3155},
    }
    """

    def __init__(self, in_channels=3, out_channels=3, hidden_channels=32, kernel_size=3,
                 hidden_layers=3, use_bias=True):
        super(DnCNN, self).__init__()

        self.use_bias =use_bias

        layers =[]
        layers.append(torch.nn.Conv2d(in_channels, hidden_channels, kernel_size, padding='same', bias=use_bias))
        layers.append(torch.nn.ReLU(inplace=True))
        for i in range(hidden_layers):
            layers.append(torch.nn.Conv2d(hidden_channels, hidden_channels, kernel_size, padding='same',
                                                                        bias=use_bias))
            layers.append(torch.nn.ReLU(inplace=True))
        layers.append(torch.nn.Conv2d(hidden_channels, out_channels, kernel_size, padding='same', bias=use_bias))

        self.net =nn.Sequential(*layers)

    def forward(self, x):
        return x -self.net(x)


def add_noise(x, sigma=0.1):
    return x +torch.randn_like(x) *sigma


def img_to_numpy(x):
    return np.clip(x.detach().cpu().numpy().squeeze().transpose(1, 2, 0), 0, 1)


def calc_psnr(x, gt):
    out =10 *np.log10(1 /((x -gt) **2).mean().item())
    return out


def plot_summary(idx, model, sigma, losses, psnrs, baseline_psnrs,
                 val_losses, val_psnrs, val_iters, train_dataset,
                 val_dataset, val_dataloader):
    with torch.no_grad():
        model.eval()

        # evaluate on training dataset sample
        train_dataset.use_patches =False
        train_image =train_dataset[0][None, ...].to(device)
        train_dataset.use_patches =True

        noisy_train_image =add_noise(train_image, sigma=sigma)
        denoised_train_image =model(noisy_train_image)

        # evaluate on validation dataset sample
        val_dataset.use_patches =False
        val_image =val_dataset[6][None, ...].to(device)
        val_dataset.use_patches =True
        val_patch_samples =next(iter(val_dataloader)).to(device)
```

```
        # calculate validation metrics
        noisy_val_patch_samples =add_noise(val_patch_samples, sigma=sigma)
        denoised_val_patch_samples =model(noisy_val_patch_samples)
        val_loss =torch.mean((val_patch_samples -denoised_val_patch_samples) **2)
        val_psnr =calc_psnr(denoised_val_patch_samples, val_patch_samples)

        val_losses.append(val_loss.item())
        val_psnrs.append(val_psnr)
        val_iters.append(idx)

        noisy_val_image =add_noise(val_image, sigma=sigma)
        denoised_val_image =model(noisy_val_image)

    plt.clf()
    plt.subplot(241)
    plt.plot(losses, label='Train loss')
    plt.plot(val_iters, val_losses, '.', label='Val. loss')
    plt.yscale('log')
    plt.legend()
    plt.title('loss')

    plt.subplot(245)
    plt.plot(psnrs, label='Train PSNR')
    plt.plot(val_iters, val_psnrs, '.', label='Val. PSNR')
    plt.plot(baseline_psnrs, label='Baseline PSNR')
    plt.ylim((0, 32))
    plt.legend()
    plt.title('psnr')

    plt.subplot(242)
    plt.imshow(img_to_numpy(train_image))
    plt.ylabel('Training Set')
    plt.title('GT')

    plt.subplot(243)
    plt.imshow(img_to_numpy(noisy_train_image))
    plt.title('Noisy Image')

    plt.subplot(244)
    plt.imshow(img_to_numpy(denoised_train_image))
    plt.title('Denoised Image')

    plt.subplot(246)
    plt.imshow(img_to_numpy(val_image))
    plt.ylabel('Validation Set')
    plt.title('GT')

    plt.subplot(247)
    plt.imshow(img_to_numpy(noisy_val_image))
    plt.title('Noisy Image')

    plt.subplot(248)
    plt.imshow(img_to_numpy(denoised_val_image))
    plt.title('Denoised Image')

    plt.tight_layout()
    plt.pause(0.1)


def train(sigma=0.1, use_bias=True, hidden_channels=32, epochs=2, batch_size=32, plot_every=200):
    print(f'==> Training on noise level {sigma:.02f} | use_bias: {use_bias} | hidden_channels: {hidden_channels}')

    # create datasets
    train_dataset =BSDS300Dataset(patch_size=32, split='train', use_patches=True)
    val_dataset =BSDS300Dataset(patch_size=32, split='test', use_patches=True)

    # create dataloaders & seed for reproducibility
    train_dataloader =DataLoader(train_dataset, batch_size=batch_size, shuffle=True, drop_last=True)
    val_dataloader =DataLoader(val_dataset, batch_size=batch_size, shuffle=True, drop_last=True)

    model =DnCNN(use_bias=use_bias, hidden_channels=hidden_channels).to(device)
    optim =torch.optim.Adam(model.parameters(), lr=1e-3)
```

```
    losses =[]
    psnrs =[]
    baseline_psnrs =[]
    val_losses =[]
    val_psnrs =[]
    val_iters =[]
    idx =0

    pbar =tqdm(total=len(train_dataset) *epochs //batch_size)
    for epoch in range(epochs):
        for sample in train_dataloader:

            model.train()
            sample =sample.to(device)

            # add noise
            noisy_sample =add_noise(sample, sigma=sigma)

            # denoise
            denoised_sample =model(noisy_sample)

            # loss function
            loss =torch.mean((denoised_sample -sample) **2)
            psnr =calc_psnr(denoised_sample, sample)
            baseline_psnr =calc_psnr(noisy_sample, sample)

            losses.append(loss.item())
            psnrs.append(psnr)
            baseline_psnrs.append(baseline_psnr)

            # update model
            loss.backward()
            optim.step()
            optim.zero_grad()

            # plot results
            if not idx % plot_every:
                plot_summary(idx, model, sigma, losses, psnrs, baseline_psnrs,
                             val_losses, val_psnrs, val_iters, train_dataset,
                             val_dataset, val_dataloader)

            idx +=1
            pbar.update(1)

    pbar.close()
    return model


def evaluate_model(model, sigma=0.1, output_filename='out.png'):
    dataset =BSDS300Dataset(patch_size=32, split='test', use_patches=False)
    model.eval()

    psnrs =[]
    for idx, image in enumerate(dataset):
        image =image[None, ...].to(device) # add batch dimension
        noisy_image =add_noise(image, sigma)
        denoised_image =model(noisy_image)
        psnr =calc_psnr(denoised_image, image)
        psnrs.append(psnr)

        # include the tiger image in your homework writeup
        if idx ==6:
            skimage.io.imsave(output_filename, (img_to_numpy(denoised_image) *255).astype(np.uint8))

    return np.mean(psnrs)


if __name__ =='__main__':
    ################################################################################
    # TODO: Your code goes here!
    ################################################################################

    # use the 'train' function with proper parameters for using biases and the
    # number of hidden layers
```

```
    model1 =train(sigma=0.1, use_bias=True, hidden_channels=32, epochs=2, batch_size=32, plot_every=200)
    model2 =train(sigma=0.1, use_bias=True, hidden_channels=64, epochs=2, batch_size=32, plot_every=200)
    model3 =train(sigma=0.1, use_bias=False, hidden_channels=32, epochs=2, batch_size=32, plot_every=200)
    model4 =train(sigma=0.1, use_bias=False, hidden_channels=64, epochs=2, batch_size=32, plot_every=200)
    # after training pass the model to the 'evaluate_model' function to run
    # on the validation dataset
    for noise in [0.05, 0.1, 0.2]:
        psnr1 =evaluate_model(model1, sigma=noise, output_filename=f'out_bias_32_{noise}.png')
        psnr2 =evaluate_model(model2, sigma=noise, output_filename=f'out_bias_64_{noise}.png')
        psnr3 =evaluate_model(model3, sigma=noise, output_filename=f'out_nobias_32_{noise}.png')
        psnr4 =evaluate_model(model4, sigma=noise, output_filename=f'out_nobias_64_{noise}.png')
        print(f'noise={noise}', psnr1, psnr2, psnr3, psnr4)
```

C task3

```
import imageio.v2
import skimage
import torch
from torch.fft import fft2, ifft2
import numpy as np
from hw5_task2 import BSDS300Dataset
from torchvision.datasets import MNIST
from torchvision.transforms import Compose, ToTensor, Lambda
from models import Unet
from PIL import Image
import torchvision.transforms as transforms
from skimage.metrics import peak_signal_noise_ratio as psnr

device =torch.device("cuda" if torch.cuda.is_available() else "cpu")


class BlurredBSDS300Dataset(BSDS300Dataset):
    def __init__(self, root='./BSDS300', patch_size=32, split='train', use_patches=True,
                 kernel_size=7, sigma=2, return_kernel=True):
        super(BlurredBSDS300Dataset, self).__init__(root, patch_size, split)

        # trim images to even size
        self.images =self.images[..., :-1, :-1]
        self.kernel_size =kernel_size
        self.return_kernel =return_kernel

        # extract blur kernel (use an MNIST digit)
        self.kernel_dataset =MNIST('./', train=True, download=True,
                                   transform=Compose([Lambda(lambda x: np.array(x)),
                                                      ToTensor(),
                                                      Lambda(lambda x: x /torch.sum(x))]))

        kernels =torch.cat([x[0] for (x, _) in zip(self.kernel_dataset, np.arange(self.images.shape[0]))])
        kernels =torch.nn.functional.interpolate(kernels[:, None, ...], size=2 *(kernel_size,))
        kernels =kernels /torch.sum(kernels, dim=(-1, -2), keepdim=True)
        self.kernel =kernels[[0]].repeat(kernels.shape[0], 1, 1, 1)

        # blur the images
        H =psf2otf(self.kernel, self.images.shape)
        self.blurred_images =ifft2(fft2(self.images) *H).real
        self.blurred_patches =self.patchify(self.blurred_images, patch_size)

        # save which blur kernel is used for each image
        self.patch_kernel =self.kernel.repeat(1, len(self.blurred_patches) //len(self.images), 1, 1)
        self.patch_kernel =self.patch_kernel.view(-1, *self.kernel.shape[-2:])

        # reshape kernel
        self.kernel =self.kernel.squeeze()

    def get_kernel(self, kernel_size, sigma):
        kernel =self.gaussian(kernel_size, sigma)
        kernel_2d =torch.matmul(kernel.unsqueeze(-1), kernel.unsqueeze(-1).t())
        return kernel_2d

    def __getitem__(self, idx):
```

```
        out = [self.blurred_images[idx][None, ...].to(device),
               self.images[idx][None, ...].to(device)]
        if self.return_kernel:
            out.append(self.kernel[[idx]].to(device))

        return out


def img_to_numpy(x):
    return np.clip(x.detach().cpu().numpy().squeeze().transpose(1, 2, 0), 0, 1)


def psf2otf(psf, shape):
    inshape = psf.shape
    psf = torch.nn.functional.pad(psf, (0, shape[-1] - inshape[-1], 0, shape[-2] - inshape[-2], 0, 0))

    # Circularly shift OTF so that the 'center' of the PSF is [0,0] element of the array
    psf = torch.roll(psf, shifts=(-int(inshape[-1] /2), -int(inshape[-2] /2)), dims=(-1, -2))

    # Compute the OTF
    otf = fft2(psf)

    return otf


def calc_psnr(x, gt):
    out = 10 * np.log10(1 /((x - gt) **2).mean().item())
    return out


def wiener_deconv(x, kernel):
    snr = 100 # use this SNR parameter for your results
    H = psf2otf(kernel, x.shape).to(device)
    G = torch.conj(H) *1 /(1 /snr + H * torch.conj(H)).to(device)
    return ifft2(fft2(x) * G).real


def load_models():
    model_deblur_denoise = Unet().to(device)
    model_deblur_denoise.load_state_dict(torch.load('pretrained/deblur_denoise.pth', map_location=device))

    model_denoise = Unet().to(device)
    model_denoise.load_state_dict(torch.load('pretrained/denoise.pth', map_location=device))

    return model_deblur_denoise, model_denoise


def evaluate_model():
    # create the dataset
    dataset = BlurredBSDS300Dataset(split='test')

    # load the models
    model_deblur_denoise, model_denoise = load_models()

    # put into evaluation mode
    model_deblur_denoise.eval()
    model_denoise.eval()

    for sigma in [0.005, 0.01, 0.02]:
        psnr1 = 0
        psnr2 = 0
        psnr3 = 0
        i = 0
        for image, gt, kernel in dataset:
            img_noise = image + torch.randn_like(image) * sigma
            result_wiener = wiener_deconv(img_noise, kernel)
            result_net1 = model_deblur_denoise(img_noise)
            result_net2 = model_denoise(result_wiener)

            # save the psnrs
            psnr1 += calc_psnr(result_wiener, gt)
            psnr2 += calc_psnr(result_net1, gt)
            psnr3 += calc_psnr(result_net2, gt)
            i += 1
```

```
            # save out sample images to include in your writeup
            if i ==10 or i ==24:
                skimage.io.imsave(f'original_image_{i}.png', (img_to_numpy(img_noise) *255).astype(np.uint8))
                skimage.io.imsave(f'image_{i}_method1_sigma={sigma}.png',
                                  (img_to_numpy(result_wiener) *255).astype(np.uint8))
                skimage.io.imsave(f'image_{i}_method2_sigma={sigma}.png',
                                  (img_to_numpy(result_net1) *255).astype(np.uint8))
                skimage.io.imsave(f'image_{i}_method3_sigma={sigma}.png',
                                  (img_to_numpy(result_net2) *255).astype(np.uint8))
                print(i, f'when sigma = {sigma}', psnr1 /i, psnr2 /i, psnr3 /i)

        print(f'when sigma = {sigma}', psnr1 /i, psnr2 /i, psnr3 /i)
        # HINT: use the calc_psnr function to calculate the PSNR, and use the
        # wiener_deconv function to perform wiener deconvolution


if __name__ =='__main__':
    evaluate_model()
```