Object recognition and computer vision 2020/2021

Jean Ponce, Ivan Laptev, Cordelia Schmid and Josef Sivic

Assignment 2: Neural networks

Adapted from practicals from Nicolas le Roux,
Andrea Vedaldi and Andrew Zisserman and Rob Fergus
by Gul Varol and Ignacio Rocco

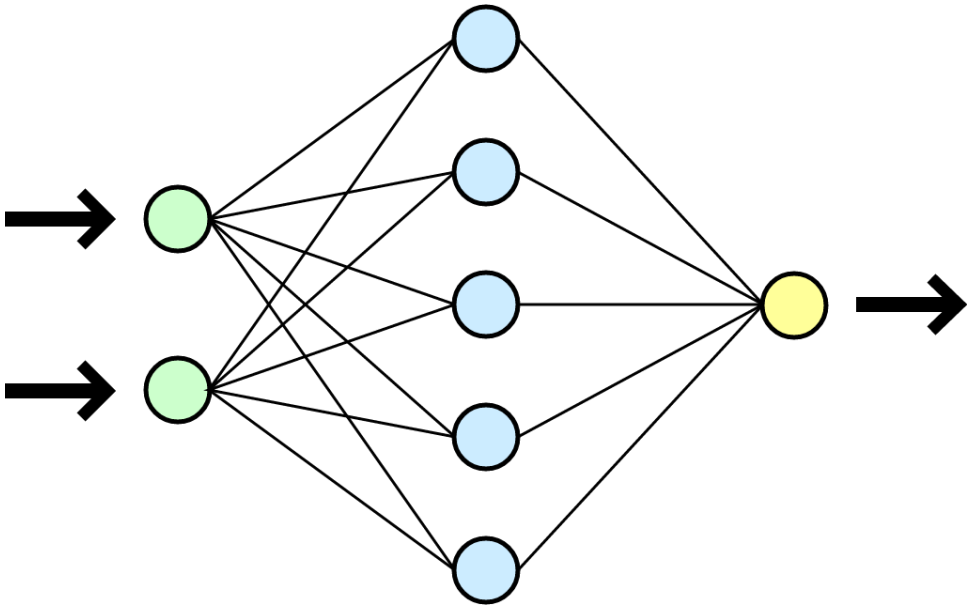


Figure 1

STUDENT: LACOMBE Yoach

EMAIL: yoach.lacombe@gmail.com

yoach.lacombe@telecom-paris.fr

Guidelines

The purpose of this assignment is that you get hands-on experience with the topics covered in class, which will help you understand these topics better. Therefore, ** it is imperative that you do this assignment yourself. No code sharing will be tolerated. **

Once you have completed the assignment, you will submit the `ipynb` file containing **both** code and results. For this, make sure to **run your notebook completely before submitting**.

The `ipynb` must be named using the following format: **A2_LASTNAME_Firstname.ipynb**, and

Goal

The goal of this assignment is to get basic knowledge and hands-on experience with training and using neural networks. In Part 1 of the assignment you will implement and experiment with the training and testing of a simple two layer fully-connected neural network, similar to the one depicted in Figure 1 above. In Part 2 you will learn about convolutional neural networks, their motivation, building blocks, and how they are trained. Finally, in part 3 you will train a CNN for classification using the CIFAR-10 dataset.

Part 1 - Training a fully connected neural network

Getting started

You will be working with a two layer neural network of the following form

$$H = \mathrm{ReLU}(W_i X + B_i) \tag{1}$$
$$Y = W_o H + B_o$$

where X is the input, Y is the output, H is the hidden layer, and W_i, W_o, B_i and B_o are the network parameters that need to be trained. Here the subscripts i and o stand for the *input* and *output* layer, respectively. This network was also discussed in the class and is illustrated in the above figure where the input units are shown in green, the hidden units in blue and the output in yellow. This network is implemented in the function `nnet_forward_logloss`.

You will train the parameters of the network from labelled training data $\{X^n, Y^n\}$ where X^n are points in $\mathbb{R}^2$ and $Y^n \in \{-1, 1\}$ are labels for each point. You will use the stochastic gradient descent algorithm discussed in the class to minimize the loss of the network on the training data given by

$$L = \sum_n s(Y^n, \bar{Y}(X^n)) \tag{2}$$

where Y^n is the target label for the n-th example and $\bar{Y}(X^n)$ is the network's output for the n-th example X^n. The skeleton of the training procedure is provided in the `train_loop` function.

We will use the logistic loss, which has the following form:

$$s(Y, \bar{Y}(X)) = \log(1 + \exp(-Y . \bar{Y}(X)) \tag{3}$$

where Y is the target label and $\bar{Y}(X)$ is the output of the network for input example X. With the logistic loss, the output of the network can be interpreted as a probability $P(\text{class} = 1|X) = \sigma(X)$, where $\sigma(X) = 1/(1 + \exp(-X))$ is the sigmoid function. Note also that $P(\text{class} = -1|X) = 1 - P(\text{class} = 1|X)$.

```
from IPython import display
import matplotlib.pyplot as plt
import numpy as np
import scipy.io as sio
```

```
def decision_boundary_nnet(X, Y, Wi, bi, Wo, bo):
    x_min, x_max = -2, 4
    y_min, y_max = -5, 3
    xx, yy = np.meshgrid(np.arange(x_min, x_max, .05),
                         np.arange(y_min, y_max, .05))

    XX = np.vstack((xx.ravel(), yy.ravel())).T
    input_hidden = np.dot(XX, Wi) + bi
    hidden = np.maximum(input_hidden, 0)
    Z = np.dot(hidden, Wo) + bo

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z > 0, cmap=plt.cm.Paired)
    plt.axis('off')

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=Y, cmap='winter')
    plt.axis([-2, 4, -5, 3])
    plt.draw()
```

```
def sigm(x):
    # Returns the sigmoid of x.
    small_x = np.where(x < -20)  # Avoid overflows.
    sigm_x = 1/(1 + np.exp(-x))
    if type(sigm_x) is np.ndarray:
        sigm_x[small_x] = 0.0
    return sigm_x
```

```
def nnet_forward_logloss(X, Y, Wi, bi, Wo, bo):
    '''
    Compute the output Po, Yo and the loss of the network for the input X
    This is a 2 layer (1 hidden layer network)

    Input:
        X ... (in R^2) set of input points, one per column
        Y ... {-1,1} the target values for the set of points X
        Wi, bi, Wo, bo ... parameters of the network
```

```
    Output:
       Po ... probabilisitc output of the network P(class=1 | x)
                  Po is in <0 1>.
                  Note: P(class=-1 | x ) = 1 - Po
       Yo ... output of the network Yo is in <-inf +inf>
       loss ... logistic loss of the network on examples X with ground target
                    values Y in {-1,1}
    '''
    # Hidden layer
    hidden = np.maximum(np.dot(X, Wi) + bi, 0)
    # Output of the network
    Yo = np.dot(hidden, Wo) + bo
    # Probabilistic output
    Po = sigm(Yo)
    # Logistic loss
    loss = np.log(1 + np.exp( -Y * Yo))
    return Po, Yo, loss
```

```
# Load the training data
!wget -q http://www.di.ens.fr/willow/teaching/recvis_orig/assignment2/double_moon_train100
train_data = sio.loadmat('./double_moon_train1000.mat', squeeze_me=True)
Xtr = train_data['X']
Ytr = train_data['Y']
# Load the validation data
!wget -q http://www.di.ens.fr/willow/teaching/recvis_orig/assignment2/double_moon_val1000.
val_data = sio.loadmat('./double_moon_val1000.mat', squeeze_me=True)
Xval = val_data['X']
Yval = val_data['Y']
```

Computing gradients of the loss with respect to network parameters

:: TASK 1.1 ::

Derive the form of the gradient of the logistic loss (3) with respect to the parameters of the network W_i, W_o, B_i and B_o. *Hint:* Use the chain rule as discussed in the class.

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

WRITE YOUR ANSWER HERE

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

Je considère ici les vecteurs comme des matrices de taille (vectorSize,1).

$$\frac{\partial S}{\partial W_o} = -Y\sigma(-Y(W_o H + B_o))H^T$$

$$\frac{\partial S}{\partial B_o} = -Y\sigma(-Y(W_oH + B_o))$$

$$\frac{\partial S}{\partial W_i} = -POS(H, W_o^T)Y\sigma(-Y(W_oH + B_o))X^T$$

$$\frac{\partial S}{\partial B_i} = -POS(H, W_o^T)Y\sigma(-Y(W_oH + B_o))$$

où $POS(M, N) = [\delta_{m_{i,j}>0} * n_{i,j}]_{i,j}$.

:: TASK 1.2 ::

Following your derivation, implement the gradient computation in the function `gradient_nn`. See the code for the description of the required inputs / outputs of this function.

```
def gradient_nn(X, Y, Wi, bi, Wo, bo):
    '''
    Compute gradient of the logistic loss of the neural network on example X with
    target label Y, with respect to the parameters Wi,bi,Wo,bo.

    Input:
        X ... 2d vector of the input example
        Y ... the target label in {-1,1}
        Wi,bi,Wo,bo ... parameters of the network
        Wi ... [dxh]
        bi ... [h]
        Wo ... [h]
        bo ... 1
        where h... is the number of hidden units
              d... is the number of input dimensions (d=2)

    Output:
        grad_s_Wi [dxh] ... gradient of loss s(Y,Y(X)) w.r.t  Wi
        grad_s_bi [h]   ... gradient of loss s(Y,Y(X)) w.r.t. bi
        grad_s_Wo [h]   ... gradient of loss s(Y,Y(X)) w.r.t. Wo
        grad_s_bo 1     ... gradient of loss s(Y,Y(X)) w.r.t. bo
    '''

    ############################
    #                          #
    #  WRITE YOUR CODE HERE    #
    #                          #
    ############################

    #valeurs intermédiaires



    H_int = Wi.T@X + bi
    H = (H_int>0)*H_int
    Y_pred = np.dot(Wo,H) + bo

    derivate_loss = -Y*sigm(-Y*Y_pred)
```

```
    grad_s_Wi = derivate_loss* np.outer(X,((H>0)*(Wo)))
    grad_s_bi = derivate_loss* ((H>0)*(Wo))
    grad_s_Wo = derivate_loss*H
    grad_s_bo = derivate_loss


    return grad_s_Wi, grad_s_bi, grad_s_Wo, grad_s_bo
    ##########################
```

Numerically verify the gradients

Here you will numerically verify that your analytically computed gradients in function `gradient_nn` are correct.

Double-cliquez (ou appuyez sur Entrée) pour modifier

:: TASK 1.3 ::

Write down the general formula for numerically computing the approximate derivative of the loss $s(\theta)$, with respect to the parameter θ_i using finite differencing. *Hint: use the first order Taylor expansion of loss $s(\theta + \Delta\theta)$ around point θ. *

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

WRITE YOUR ANSWER HERE

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

On a :

$$s(\theta + \Delta\theta) - s(\theta) = (\nabla s(\theta))^T \Delta\theta + o(||\theta||_2)$$

Donc:

$$\frac{\partial s}{\partial \theta_i} \approx \frac{s(\theta + \Delta\theta) - s(\theta)}{\Delta\theta_i}$$

On a également :

$$\frac{\partial s}{\partial \theta_i} \approx -\frac{s(\theta - \Delta\theta) - s(\theta)}{\Delta\theta_i}$$

D'où

$$\frac{\partial s}{\partial \theta_i} \approx \frac{s(\theta + \Delta\theta) - s(\theta - \Delta\theta)}{2\Delta\theta_i}$$

Following the general formula, `gradient_nn_numerical` function numerically computes the derivatives of the loss function with respect to all the parameters of the network W_i, W_o, B_i

and B_o:

```
def gradient_nn_numerical(X, Y, Wi, bi, Wo, bo):
    '''
    Compute numerical gradient of the logistic loss of the neural network on
    example X with target label Y, with respect to the parameters Wi,bi,Wo,bo.

    Input:
       X ... 2d vector of the input example
       Y ... the target label in {-1,1}
       Wi, bi, Wo, bo ... parameters of the network
       Wi ... [dxh]
       bi ... [h]
       Wo ... [h]
       bo ... 1
       where h... is the number of hidden units
             d... is the number of input dimensions (d=2)

    Output:
       grad_s_Wi_numerical [dxh] ... gradient of loss s(Y,Y(X)) w.r.t  Wi
       grad_s_bi_numerical [h]   ... gradient of loss s(Y,Y(X)) w.r.t. bi
       grad_s_Wo_numerical [h]   ... gradient of loss s(Y,Y(X)) w.r.t. Wo
       grad_s_bo_numerical 1     ... gradient of loss s(Y,Y(X)) w.r.t. bo
    '''


    eps = 1e-8
    grad_s_Wi_numerical = np.zeros(Wi.shape)
    grad_s_bi_numerical = np.zeros(bi.shape)
    grad_s_Wo_numerical = np.zeros(Wo.shape)

    for i in range(Wi.shape[0]):
        for j in range(Wi.shape[1]):
            dummy, dummy, pos_loss = nnet_forward_logloss(X, Y, sumelement_matrix(Wi, i, j
            dummy, dummy, neg_loss = nnet_forward_logloss(X, Y, sumelement_matrix(Wi, i, j
            grad_s_Wi_numerical[i, j] = (pos_loss - neg_loss)/(2*eps)

    for i in range(bi.shape[0]):
        dummy, dummy, pos_loss = nnet_forward_logloss(X, Y, Wi, sumelement_vector(bi, i, +
        dummy, dummy, neg_loss = nnet_forward_logloss(X, Y, Wi, sumelement_vector(bi, i, -
        grad_s_bi_numerical[i] = (pos_loss - neg_loss)/(2*eps)

    for i in range(Wo.shape[0]):
        dummy, dummy, pos_loss = nnet_forward_logloss(X, Y, Wi, bi, sumelement_vector(Wo,
        dummy, dummy, neg_loss = nnet_forward_logloss(X, Y, Wi, bi, sumelement_vector(Wo,
        grad_s_Wo_numerical[i] = (pos_loss - neg_loss)/(2*eps)

    dummy, dummy, pos_loss = nnet_forward_logloss(X, Y, Wi, bi, Wo, bo+eps)
    dummy, dummy, neg_loss = nnet_forward_logloss(X, Y, Wi, bi, Wo, bo-eps)
    grad_s_bo_numerical = (pos_loss - neg_loss)/(2*eps)

    return grad_s_Wi_numerical, grad_s_bi_numerical, grad_s_Wo_numerical, grad_s_bo_numeri


def sumelement_matrix(X, i, j, element):
    Y = np.copy(X)
```

```
    Y[i, j] = X[i, j] + element
    return Y


def sumelement_vector(X, i, element):
    Y = np.copy(X)
    Y[i] = X[i] + element
    return Y
```

:: TASK 1.4 ::

Run the following code snippet and understand what it is doing. `gradcheck` function checks that the analytically computed derivative using function `gradient_nn` (e.g. `grad_s_bo`) at the same training example $\{X, Y\}$ is the same (up to small errors) as your numerically computed value of the derivative using function `gradient_nn_numerical` (e.g. `grad_s_bo_numerical`). Make sure the output is `SUCCESS` to move on to the next task.

```
def gradcheck():
    '''
    Check that the numerical and analytical gradients are the same up to eps
    '''
    h = 3 # number of hidden units
    eps = 1e-6
    for i in range(1000):
        # Generate random input/output/weight/bias
        X  = np.random.randn(2)
        Y  = 2* np.random.randint(2) - 1 # {-1, 1}
        Wi = np.random.randn(X.shape[0], h)
        bi = np.random.randn(h)
        Wo = np.random.randn(h)
        bo = np.random.randn(1)
        # Compute analytical gradients
        grad_s_Wi, grad_s_bi, grad_s_Wo, grad_s_bo = gradient_nn(X, Y, Wi, bi, Wo, bo)
        # Compute numerical gradients
        grad_s_Wi_numerical, grad_s_bi_numerical, grad_s_Wo_numerical, grad_s_bo_numerical
        # Compute the difference between analytical and numerical gradients
        delta_Wi = np.mean(np.abs(grad_s_Wi - grad_s_Wi_numerical))
        delta_bi = np.mean(np.abs(grad_s_bi - grad_s_bi_numerical))
        delta_Wo = np.mean(np.abs(grad_s_Wo - grad_s_Wo_numerical))
        delta_bo = np.abs(grad_s_bo - grad_s_bo_numerical)
        # Difference larger than a threshold
        if ( delta_Wi > eps or delta_bi > eps or delta_Wo > eps or delta_bo > eps):
            return False

    return True


# Check gradients
if gradcheck():
    print('SUCCESS: Passed gradcheck.')
```

```
else:
    print('FAILURE: Fix gradient_nn and/or gradient_nn_aprox implementation.')
```

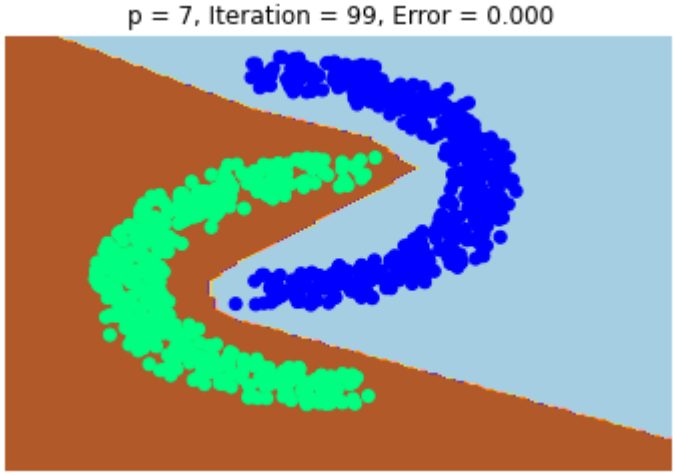
```
 SUCCESS: Passed gradcheck
```

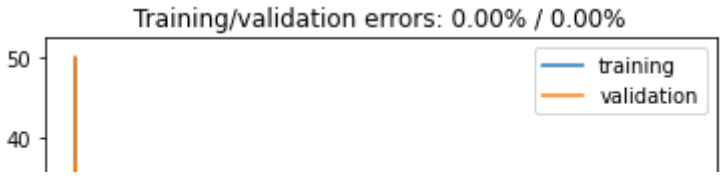
Training the network using backpropagation and experimenting with different parameters

Use the provided code below that calls the `train_loop` function. Set the number of hidden units to 7 by setting $h = 7$ in the code and set the learning rate to 0.02 by setting `lrate = 0.02`. Run the training code. Visualize the trained hyperplane using the provided function `plot_decision_boundary(Xtr,Ytr,Wi,bi,Wo,bo)`. Show also the evolution of the training and validation errors. Include the decision hyper-plane visualization and the training and validation error plots.

```
def train_loop(Xtr, Ytr, Xval, Yval, h, lrate, vis='all', nEpochs=100):
    '''
    Check that the numerical and analytical gradients are the same up to eps

    Input:
        Xtr ... Nx2 matrix of training samples
        Ytr ... N dimensional vector of training labels
        Xval ... Nx2 matrix of validation samples
        Yval ... N dimensional vector validation labels
        h ... number of hidden units
        lrate ... learning rate
        vis ... visulaization option ('all' | 'last' | 'never')
        nEpochs ... number of training epochs

    Output:
        tr_error ... nEpochs*nSamples dimensional vector of training error
        val_error ... nEpochs*nSamples dimensional vector of validation error
    '''

    nSamples = Xtr.shape[0]
    tr_error = np.zeros(nEpochs*nSamples)
    val_error = np.zeros(nEpochs*nSamples)

    # Randomly initialize parameters of the model
    Wi = np.random.randn(Xtr.shape[1], h)
    Wo = np.zeros(h)
    bi = np.zeros(h)
    bo = 0.

    if(vis == 'all' or vis == 'last'):
        plt.figure()

    for i in range(nEpochs*nSamples):
        # Draw an example at random
        n = np.random.randint(nSamples)
```

```
            X = Xtr[n]
            Y = Ytr[n]

            # Compute gradient
            grad_s_Wi, grad_s_bi, grad_s_Wo, grad_s_bo = gradient_nn(X, Y, Wi, bi, Wo, bo)

            # Gradient update
            Wi -= lrate*grad_s_Wi
            Wo -= lrate*grad_s_Wo
            bi -= lrate*grad_s_bi
            bo -= lrate*grad_s_bo

            # Compute training error
            Po, Yo, loss    = nnet_forward_logloss(Xtr, Ytr, Wi, bi, Wo, bo)
            Yo_class        = np.sign(Yo)
            tr_error[i]     = 100*np.mean(Yo_class != Ytr)

            # Compute validation error
            Pov, Yov, lossv = nnet_forward_logloss(Xval, Yval, Wi, bi, Wo, bo)
            Yov_class       = np.sign(Yov)
            val_error[i]    = 100*np.mean(Yov_class != Yval)

            # Plot (at every epoch if visualization is 'all', only at the end if 'last')
            if(vis == 'all' and i%nSamples == 0) or (vis == 'last' and i == nEpochs*nSamples -
                # Draw the decision boundary.
                plt.clf()
                plt.title('p = %d, Iteration = %.d, Error = %.3f' % (h, i/nSamples, tr_error[i
                decision_boundary_nnet(Xtr, Ytr, Wi, bi, Wo, bo)
                display.display(plt.gcf(), display_id=True)
                display.clear_output(wait=True)

        if(vis == 'all'):
            # Plot the evolution of the training and test errors
            plt.figure()
            plt.plot(tr_error, label='training')
            plt.plot(val_error, label='validation')
            plt.legend()
            plt.title('Training/validation errors: %.2f%% / %.2f%%' % (tr_error[-1], val_error
        return tr_error, val_error

    # Run training
    h = 7
    lrate = .02
    tr_error, val_error = train_loop(Xtr, Ytr, Xval, Yval, h, lrate)
```

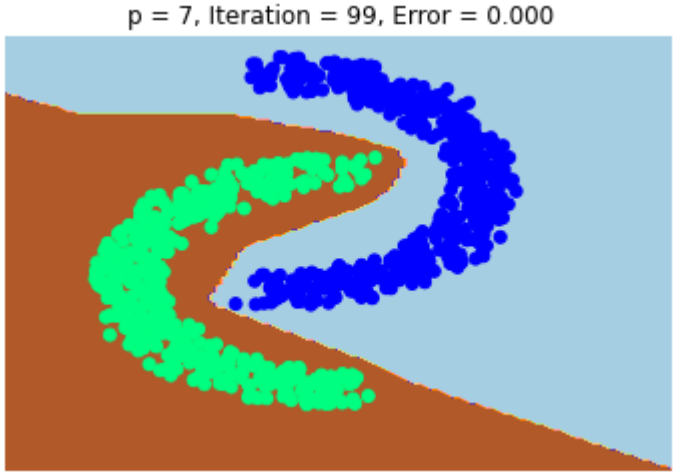


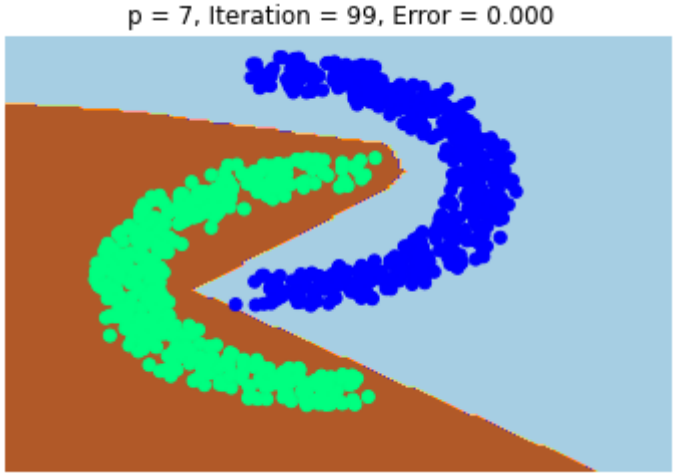


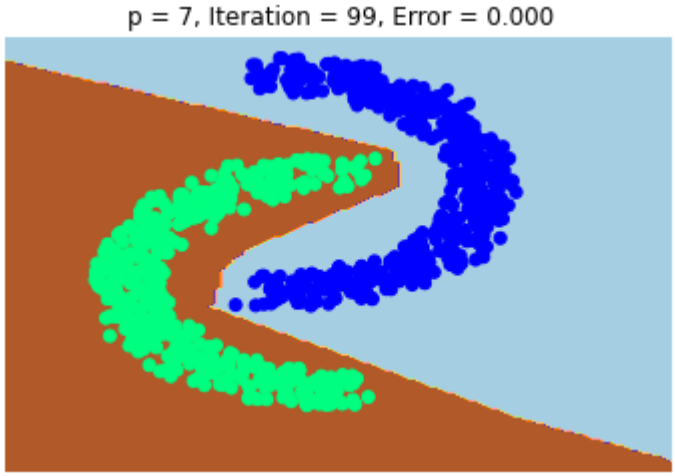
:: TASK 1.6 ::

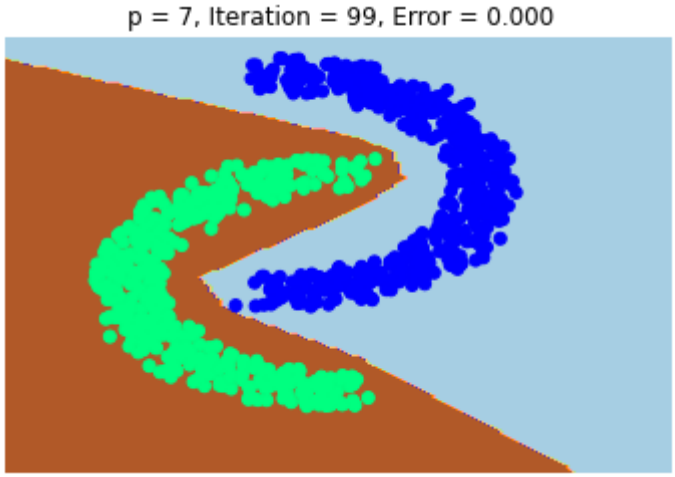
Random initializations. Repeat this procedure 5 times from 5 different random initializations. Record for each run the final training and validation errors. Did the network always converge to zero training error? Summarize your final training and validation errors into a table for the 5 runs. You do not need to include the decision hyper-plane visualizations. Note: to speed-up the training you can plot the visualization figures less often (or never) and hence speed-up the training.

```
############################
#                          #
#  WRITE YOUR CODE HERE    #
#                          #
############################

# Run training
h = 7
lrate = .02
L = [train_loop(Xtr, Ytr, Xval, Yval, h, lrate, vis='last', nEpochs=100) for i in range(5)
```










```
print([·(i[-1],·j[-1])·for·(i,j)·in·L])
```

```
[(0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0)]
```

##########

WRITE YOUR ANSWER HERE

##########

The training and validation errors are each time zeros. For these initial randomization, the algorithm always worked perfectly.

:: SAMPLE TASK ::

For this task, the answer is given. Run the given code and answer Task 1.8 similarly.

Learning rate:

Keep $h = 7$ and change the learning rate to values $\text{lrate} = \{2, 0.2, 0.02, 0.002\}$. For each of these values run the training procedure 5 times and observe the training behaviour. You do not need to include the decision hyper-plane visualizations in your answer.

- Make one figure where *final* error for (i) training and (ii) validation sets are superimposed. x-axis should be the different values of the learning rate, y-axis the error *mean* across 5 runs. Show the standard deviation with error bars and make sure to label each plot with a legend.

- Make another figure where *training error evolution* for each learning rate is superimposed. x-axis should be the iteration number, y-axis the training error *mean* across 5 runs for a given learning rate. Show the standard deviation with error bars and make sure to label each curve with a legend.

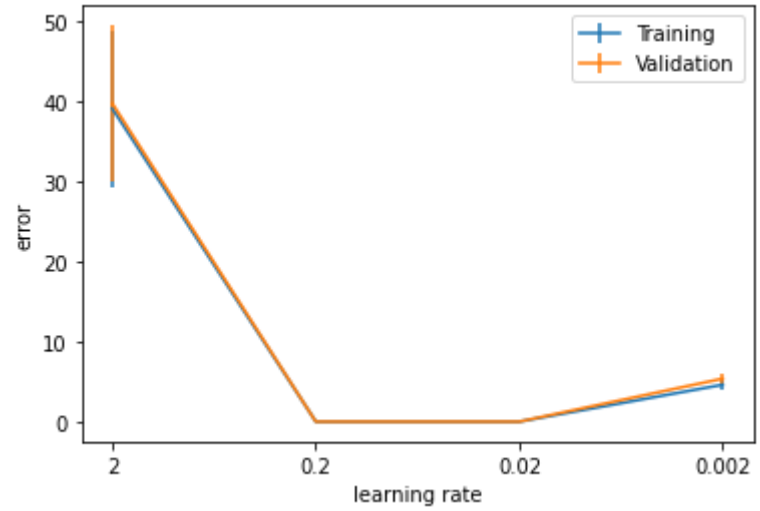
```
nEpochs = 40
trials = 5
lrates = [2, 0.2, 0.02, 0.002]
plot_data_lr = np.zeros((2, trials, len(lrates), nEpochs*1000))
h = 7
for j, lrate in enumerate(lrates):
    print('LR = %f' % lrate)
    for i in range(trials):
        tr_error, val_error = train_loop(Xtr, Ytr, Xval, Yval, h, lrate, vis='never', nEpo
        plot_data_lr[0, i, j, :] = tr_error
        plot_data_lr[1, i, j, :] = val_error
```

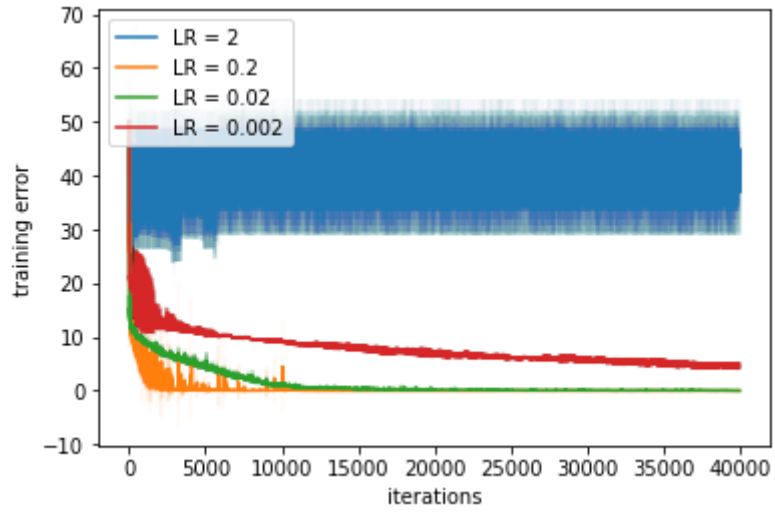
```
LR = 2.000000
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:32: RuntimeWarning: over
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:63: RuntimeWarning: over
LR = 0.200000
LR = 0.020000
LR = 0.002000
```

```
plt.errorbar(np.arange(len(lrates)), plot_data_lr[0, :, :, -1].mean(axis=0), yerr=plot_dat
plt.errorbar(np.arange(len(lrates)), plot_data_lr[1, :, :, -1].mean(axis=0), yerr=plot_dat
```

```
plt.xticks(np.arange(len(lrates)), lrates)
plt.xlabel('learning rate')
plt.ylabel('error')
plt.legend()

# Plot the evolution of the training loss for each learning rate
plt.figure()
for j, lrate in enumerate(lrates):
    x = np.arange(plot_data_lr.shape[3])
    # Mean training loss over trials
    y = plot_data_lr[0, :, j, :].mean(axis=0)
    # Standard deviation over trials
    ebar = plot_data_lr[0, :, j, :].std(axis=0)
    # Plot
    markers, caps, bars = plt.errorbar(x, y, yerr=ebar, label='LR = ' + str(lrate))
    # Make the error bars transparent
    [bar.set_alpha(0.01) for bar in bars]
plt.legend()
plt.xlabel('iterations')
plt.ylabel('training error')
```

```
Text(0, 0.5, 'training error')
```





:: TASK 1.7 ::

- Briefly discuss the different behaviour of the training for different learning rates. How many iterations does it take to converge or does it converge at all? Which learning rate is better and why?

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

WRITE YOUR ANSWER HERE

###################### The behaviour is as expected : the neural network can't learn if the learning rate is too high (because the gradient descent are too rough, it never goes toward a minima). The neural network learns too slowly if the learning rate is too slow, but it does seem to converge even though there are not enough iterations to do so. The best learning rate seem to be 0.02 because it has the lowest training and validation errors (alongside with 0.2) but doesn't have the bumps of 0.2 in training error during the training. The learning is thus more smooth than with 0.2.

:: TASK 1.8 ::

The number of hidden units:

Set the learning rate to 0.02 and change the number of hidden units $h = \{1, 2, 5, 7, 10, 100\}$. For each of these values run the training procedure 5 times and observe the training behaviour

-Visualize one decision hyper-plane per number of hidden units.

-Make one figure where *final* error for (i) training and (ii) validation sets are superimposed. x-axis should be the different values of the number of hidden units, y-axis the error *mean* across 5 runs. Show the standard deviation with error bars and make sure to label each plot with a legend.

-Make another figure where *training error evolution* for each number of hidden units is superimposed. x-axis should be the iteration number, y-axis the training error *mean* across 5 runs for a given learning rate. Show the standard deviation with error bars and make sure to label each curve with a legend.

-Briefly discuss the different behaviours for the different numbers of hidden units.

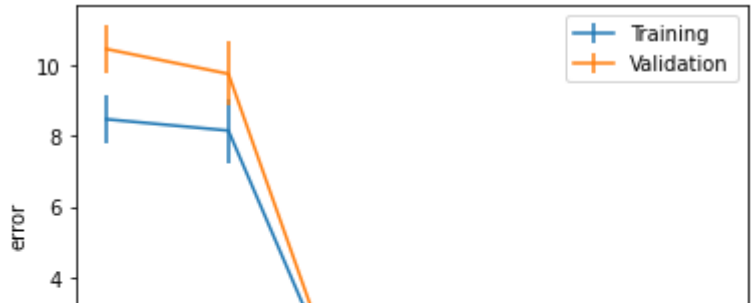
```
##########################
#                        #
#  WRITE YOUR CODE HERE  #
#                        #
##########################
```

```
nEpochs = 40
trials = 5
hs = [1,2,5,7,10,100]
lr = 0.02
```

```
plot_data_lr = np.zeros((2, trials, len(hs), nEpochs*1000))
for j, h in enumerate(hs):
    print('h = %f' % h)
    for i in range(trials):
        tr_error, val_error = train_loop(Xtr, Ytr, Xval, Yval, h, lr, vis='never', nEpochs
        plot_data_lr[0, i, j, :] = tr_error
        plot_data_lr[1, i, j, :] = val_error
```

```
plt.errorbar(np.arange(len(hs)), plot_data_lr[0, :, :, -1].mean(axis=0), yerr=plot_data_lr
plt.errorbar(np.arange(len(hs)), plot_data_lr[1, :, :, -1].mean(axis=0), yerr=plot_data_lr
plt.xticks(np.arange(len(hs)), hs)
plt.xlabel('h')
plt.ylabel('error')
plt.legend()

# Plot the evolution of the training loss for each learning rate
plt.figure()
for j, h in enumerate(hs):
    x = np.arange(plot_data_lr.shape[3])
    # Mean training loss over trials
    y = plot_data_lr[0, :, j, :].mean(axis=0)
    # Standard deviation over trials
    ebar = plot_data_lr[0, :, j, :].std(axis=0)
    # Plot
    markers, caps, bars = plt.errorbar(x, y, yerr=ebar, label='h = ' + str(h))
    # Make the error bars transparent
    [bar.set_alpha(0.01) for bar in bars]
plt.legend()
plt.xlabel('iterations')
plt.ylabel('training error')
```

```
Text(0, 0.5, 'training error')
```




```
plot_data_lr[:,:,:,-1]

array([[[ 8.6,  8.5,  0.2,  0. ,  0. ,  0. ],
        [ 7.8,  6.5,  0. ,  0. ,  0. ,  0. ],
        [ 7.8,  9.2,  0.1,  0. ,  0. ,  0. ],
        [ 8.5,  8.1,  0.2,  0. ,  0.1,  0. ],
        [ 9.6,  8.4,  0. ,  0. ,  0. ,  0. ]],

       [[11. ,  9.6,  0.1,  0. ,  0. ,  0. ],
        [ 9.7,  7.5,  0. ,  0. ,  0.1,  0. ],
        [10.3, 11.7,  0. ,  0. ,  0. ,  0. ],
        [10.9,  8.7,  0. ,  0. ,  0. ,  0. ],
        [10.3, 11.2,  0. ,  0. ,  0. ,  0. ]]])
```

```
######################
```

WRITE YOUR ANSWER HERE

```
######################
```

Here we are confronted to a classical problem in NN optimization, that is to say avoiding under or overfitting (bias-variance trade-off). We can see that if the NN has too little hidden units, it will severly underfit the data, because the model is too simple ('dumb') to be able to solve the problem.

In contrario, if the NN has too much hidden units, we would expect it to severly overfit the model, because the model has enough complexity to learn to distinguish each training sample. As a result, it shouldn't be able to deal with validation samples. **But here,** it does not overfit (training and validation errors are both low (almost always 0 actually).

The right number of hidden units seems to be between 5 and 100, as we can see on the plots. In such case, the NN has a smooth training error (quick learning) and in the end, it has low validation and training errors.

If we expect new data to be as simple as the validation set, we can set h=5, as it does converge and has a lowest complexity as compare to higher h.

Part 2 - Building blocks of a CNN

This part introduces typical CNN building blocks, such as ReLU units and linear filters. For a motivation for using CNNs over fully-connected neural networks, see [Le Cun, et al, 1998].

Install PyTorch

```
!pip install torch torchvision
import torch
print(torch.__version__)
print(torch.cuda.is_available())
```

```
Requirement already satisfied: torch in /usr/local/lib/python3.7/dist-packages (1.9.0
Requirement already satisfied: torchvision in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: pillow>=5.3.0 in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from
1.9.0+cu111
False
```

Convolution

A feed-forward neural network can be thought of as the composition of number of functions

$$f(\mathbf{x}) = f_L(\dots f_2(f_1(\mathbf{x}; \mathbf{w}_1); \mathbf{w}_2)\dots), \mathbf{w}_L).$$

Each function f_l takes as input a datum $\mathbf{x}_l$ and a parameter vector $\mathbf{w}_l$ and produces as output a datum $\mathbf{x}_{l+1}$. While the type and sequence of functions is usually handcrafted, the parameters $\mathbf{w} = (\mathbf{w}_1, \dots, \mathbf{w}_L)$ are *learned from data* in order to solve a target problem, for example classifying images or sounds.

In a *convolutional neural network* data and functions have additional structure. The data $\mathbf{x}_1, \dots, \mathbf{x}_n$ are images, sounds, or more in general maps from a lattice[1] to one or more real numbers. In particular, since the rest of the practical will focus on computer vision applications, data will be 2D arrays of pixels. Formally, each $\mathbf{x}_i$ will be a $M \times N \times K$ real array of $M \times N$ pixels and K channels per pixel. Hence the first two dimensions of the array span space, while the last one spans channels. Note that only the input $\mathbf{x} = \mathbf{x}_1$ of the network is an actual image, while the remaining data are intermediate *feature maps*.

The second property of a CNN is that the functions f_l have a *convolutional structure*. This means that f_l applies to the input map $\mathbf{x}_l$ an operator that is *local and translation invariant*. Examples of convolutional operators are applying a bank of linear filters to $\mathbf{x}_l$.

In this part we will familiarise ourselves with a number of such convolutional and non-linear operators. The first one is the regular *linear convolution* by a filter bank. We will start by focusing our attention on a single function relation as follows:

$$f : \mathbb{R}^{M\times N\times K} \to \mathbb{R}^{M'\times N'\times K'}, \qquad \mathbf{x} \mapsto \mathbf{y}.$$

[1] A two-dimensional *lattice* is a discrete grid embedded in R^2, similar for example to a checkerboard.

```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import torch
import torchvision

# Download an example image
!wget -q http://www.di.ens.fr/willow/teaching/recvis_orig/assignment3/images/peppers.png
# Read the image
x = np.asarray(Image.open('peppers.png'))/255.0
# Print the size of x. Third dimension (=3) corresponds to the R, G, B channels
print(x.shape)
# Visualize the input x
plt.imshow(x)
# Convert to torch tensor
x = torch.from_numpy(x).permute(2, 0, 1).float()
# Prepare it as a batch
x = x.unsqueeze(0)
```

```
(384, 512, 3)
```

This should display an image of bell peppers.

Next, we create a convolutional layer with a bank of 10 filters of dimension $5 \times 5 \times 3$ whose coefficients are initialized randomly. This uses the `torch.nn.Conv2d` module from PyTorch:

```
# Create a convolutional layer and a random bank of linear filters
conv = torch.nn.Conv2d(3, 10, kernel_size=5, stride=1, padding=0, bias=False)
print(conv.weight.size())
```

```
torch.Size([10, 3, 5, 5])
```

Remark: You might have noticed that the `bias` argument to the `torch.nn.Conv2d` function is the empty matrix `false`. It can be otherwise used to pass a vector of bias terms to add to the output of each filter.

Note that `conv.weight` has four dimensions, packing 10 filters. Note also that each filter is not flat, but rather a volume containing three slices. The next step is applying the filter to the image.

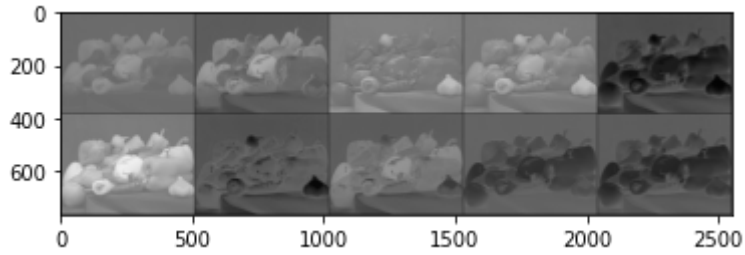
```
# Apply the convolution operator
y = conv(x)
# Observe the input/output sizes
print(x.size())
print(y.size())
```

```
torch.Size([1, 3, 384, 512])
torch.Size([1, 10, 380, 508])
```

The variable `y` contains the output of the convolution. Note that the filters are three-dimensional. This is because they operate on a tensor $\mathbf{x}$ with K channels. Furthermore, there are K' such filters, generating a K' dimensional map $\mathbf{y}$.

We can now visualise the output `y` of the convolution. In order to do this, use the `torchvision.utils.make_grid` function to display an image for each feature channel in `y`:

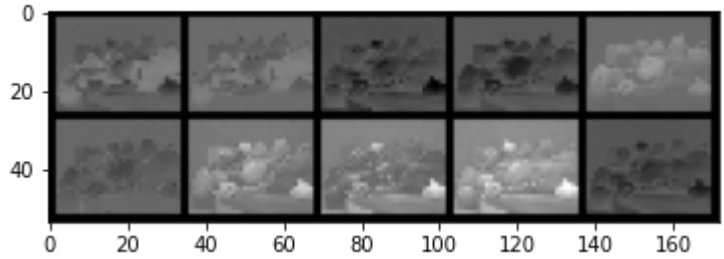
```
# Visualize the output y
def vis_features(y):
    # Organize it into 10 grayscale images
    out = y.permute(1, 0, 2, 3)
    # Scale between [0, 1]
    out = (out - out.min().expand(out.size())) / (out.max() - out.min()).expand(out.size()
    # Create a grid of images
    out = torchvision.utils.make_grid(out, nrow=5)
    # Convert to numpy image
    out = np.transpose(out.detach().numpy(), (1, 2, 0))
    # Show
    plt.imshow(out)
    # Remove grid
    plt.gca().grid(False)

vis_features(y)
```



So far filters preserve the resolution of the input feature map. However, it is often useful to *downsample the output*. This can be obtained by using the `stride` option in `torch.nn.Conv2d`:

```
# Try again, downsampling the output
conv_ds = torch.nn.Conv2d(3, 10, kernel_size=5, stride=16, padding=0, bias=False)
y_ds = conv_ds(x)
print(x.size())
print(y_ds.size())
vis_features(y_ds)
```

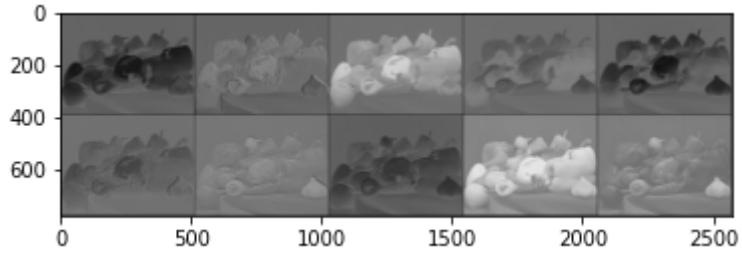
```
torch.Size([1, 3, 384, 512])
torch.Size([1, 10, 24, 32])
```



Applying a filter to an image or feature map interacts with the boundaries, making the output map smaller by an amount proportional to the size of the filters. If this is undesirable, then the input array can be padded with zeros by using the `pad` option:

```
# Try padding
conv_pad = torch.nn.Conv2d(3, 10, kernel_size=5, stride=1, padding=2, bias=False)
y_pad = conv_pad(x)
print(x.size())
print(y_pad.size())
vis_features(y_pad)
```

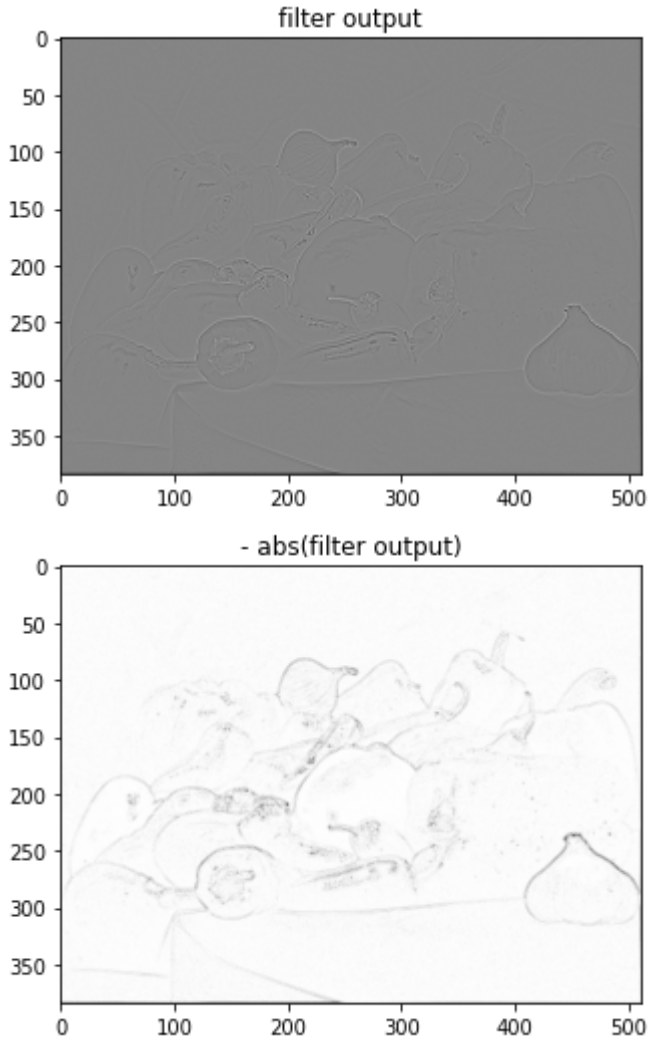
```
torch.Size([1, 3, 384, 512])
torch.Size([1, 10, 384, 512])
```



In order to consolidate what has been learned so far, we will now design a filter by hand:

```
w = torch.FloatTensor([[0,  1, 0 ],
                       [1, -4, 1 ],
                       [0,  1, 0 ]])
w = w.repeat(3, 1).reshape(1, 3, 3, 3)
conv_lap = torch.nn.Conv2d(3, 3, kernel_size=3, stride=1, padding=1, bias=False)
conv_lap.weight = torch.nn.Parameter(w)
y_lap = conv_lap(x)
print(x.size())
print(y_lap.size())

plt.figure()
vis_features(y_lap)
plt.title('filter output')

plt.figure()
vis_features(-torch.abs(y_lap))
plt.title('- abs(filter output)') ;
```

```
torch.Size([1, 3, 384, 512])
torch.Size([1, 1, 384, 512])
```



:: TASK 2.1 ::

- i. What filter have we implemented?
- ii. How are the RGB colour channels processed by this filter?
- iii. What image structure are detected?

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

WRITE YOUR ANSWER HERE

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

- i. The filter is a derivation filter (it detect the edges, that is the regions with high gradient (les contours)).
- ii. It applies the same derivation filter to each channel and sum over the channel (K' = 1).
- iii. It detects the edges, that is the regions with high gradient (les contours). In practice, here, it detects the peppers' edges.

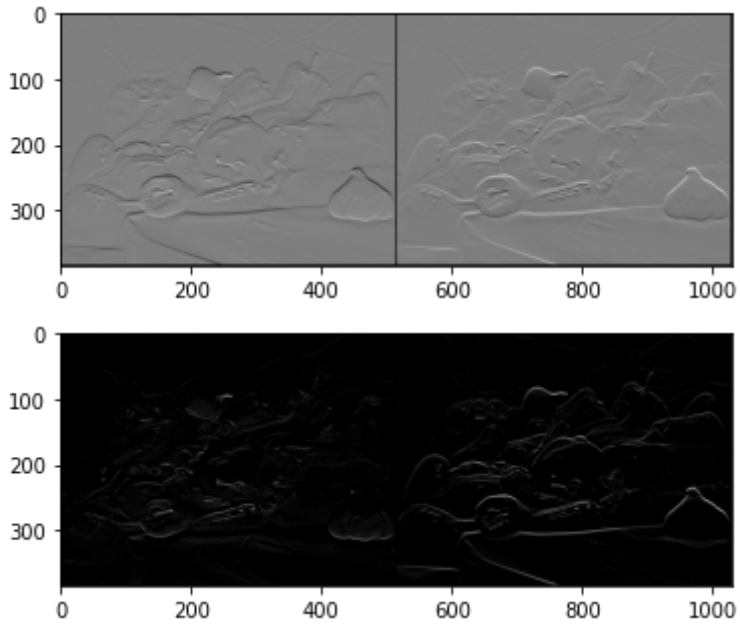
Non-linear activation functions

The simplest non-linearity is obtained by following a linear filter by a *non-linear activation function*, applied identically to each component (i.e. point-wise) of a feature map. The simplest such function is the *Rectified Linear Unit (ReLU)*

$$y_{ijk} = \max\{0, x_{ijk}\}.$$

This function is implemented by `torch.nn.ReLU()`. Run the code below and understand what the filter $\mathbf{w}$ is doing.

```
w = torch.FloatTensor([[1], [0], [-1]]).repeat(1, 3, 1, 1)
w = torch.cat((w, -w), 0)

conv = torch.nn.Conv2d(3, 2, kernel_size=(3, 1), stride=1, padding=0, bias=False)
conv.weight = torch.nn.Parameter(w)
relu = torch.nn.ReLU()

y = conv(x)
z = relu(y)

plt.figure()
vis_features(y)
plt.figure()
vis_features(z)
```



Pooling

There are several other important operators in a CNN. One of them is *pooling*. A pooling operator operates on individual feature channels, coalescing nearby feature values into one by the application of a suitable operator. Common choices include max-pooling (using the max operator) or sum-pooling (using summation). For example, *max-pooling* is defined as:

$$y_{ijk} = \max\{y_{i'j'k} : i \leq i' < i + p, j \leq j' < j + p\}$$

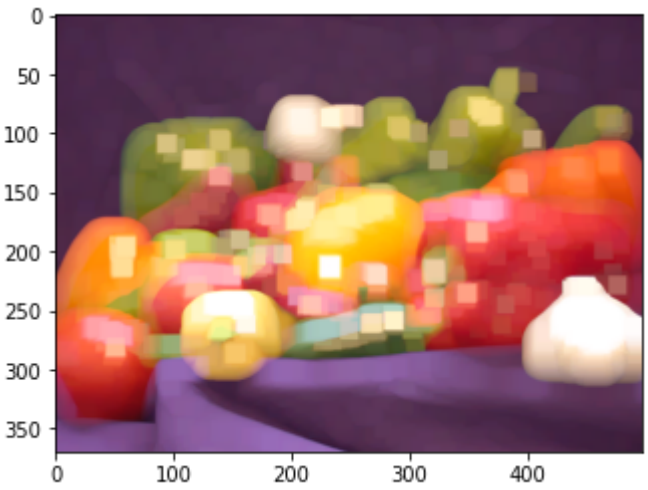
Max-pooling is implemented by `torch.nn.MaxPool2d()`.

:: TASK 2.2 ::

Run the code below to try max-pooling. Look at the resulting image. Can you interpret the result?

```
mp = torch.nn.MaxPool2d(15, stride=1)
y = mp(x)
plt.imshow(y.squeeze().permute(1, 2, 0).numpy())
plt.gca().grid(False)
```

```
/usr/local/lib/python3.7/dist-packages/torch/nn/functional.py:718: UserWarning: Named
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)
```



##########

WRITE YOUR ANSWER HERE

##########

It simply performs down-sampling by taking the maximum value among the 15x15 squares of the original picture.

Part 3 - Training a CNN

This part is an introduction to using PyTorch for training simple neural net models. CIFAR-10 dataset will be used.

Imports

```
from __future__ import print_function
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
```

```
from torch.autograd import Variable
import torchvision


#### I add GPU handling

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

print(device)
```

```
cuda
```

Parameters

The default values for the learning rate, batch size and number of epochs are given in the "options" cell of this notebook. Unless otherwise specified, use the default values throughout this assignment.

```
batch_size = 64   # input batch size for training
epochs = 10       # number of epochs to train
lr = 0.01         # learning rate
```

Warmup

It is always good practice to visually inspect your data before trying to train a model, since it lets you check for problems and get a feel for the task at hand.

CIFAR-10 is a dataset of 60,000 color images (32 by 32 resolution) across 10 classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). The train/test split is 50k/10k.

```
# Data Loading
# Warning: this cell might take some time when you run it for the first time,
#          because it will download the dataset from the internet
dataset = 'cifar10'
data_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])
trainset = datasets.CIFAR10(root='.', train=True, download=True, transform=data_transform)
testset = datasets.CIFAR10(root='.', train=False, download=True, transform=data_transform)

train_loader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True,
test_loader  = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False,
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./cifar-10-pyt
                                                                    170499072/? [00:11<00:00, 16882662.82it/s]
Extracting ./cifar-10-python.tar.gz to .
```

:: TASK 3.1 ::

Use `matplotlib` and ipython notebook's visualization capabilities to display some of these images. Display 5 images from the dataset together with their category label. See this PyTorch tutorial page for hints on how to achieve this.

```
############################
#                          #
#  WRITE YOUR CODE HERE    #
#                          #
############################

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')


import matplotlib.pyplot as plt
import numpy as np

# functions to show an image


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(train_loader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images[:5]))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(5)))
```


```
frog  frog plane plane   dog
```

Training a Convolutional Network on CIFAR-10

Start by running the provided training code below. By default it will train on CIFAR-10 for 10 epochs (passes through the training data) with a single layer network. The loss function cross_entropy computes a Logarithm of the Softmax on the output of the neural network, and then computes the negative log-likelihood w.r.t. the given `target`. Note the decrease in training loss and corresponding decrease in validation errors.

```
def train(epoch, network):
    network.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = network(data)
        loss = F.cross_entropy(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 100 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))

def test(network):
    network.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = network(data)
            test_loss += F.cross_entropy(output, target, size_average=False).item() # sum
            pred = output.data.max(1, keepdim=True)[1] # get the index of the max log-prob
            correct += pred.eq(target.data.view_as(pred)).cpu().sum()

    test_loss /= len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))



# Single layer network architecture

class Net(nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super(Net, self).__init__()
        self.linear = nn.Linear(num_inputs, num_outputs)
        self.num_inputs = num_inputs

    def forward(self, input):
        input = input.view(-1, self.num_inputs) # reshape input to batch x num_inputs
```

```
        output = self.linear(input)
        return output

# Train
network = Net(3072, 10)
network.to(device)

optimizer = optim.SGD(network.parameters(), lr=lr)
for epoch in range(1, 11):
    train(epoch, network)
    test(network)
```

```
Test set: Average loss: 1.7429, Accuracy: 4017/10000 (40%)

Train Epoch: 3 [0/50000 (0%)]   Loss: 2.072908
Train Epoch: 3 [6400/50000 (13%)]       Loss: 1.764110
Train Epoch: 3 [12800/50000 (26%)]      Loss: 1.477228
Train Epoch: 3 [19200/50000 (38%)]      Loss: 1.763885
Train Epoch: 3 [25600/50000 (51%)]      Loss: 1.696771
Train Epoch: 3 [32000/50000 (64%)]      Loss: 1.588085
Train Epoch: 3 [38400/50000 (77%)]      Loss: 1.787011
Train Epoch: 3 [44800/50000 (90%)]      Loss: 1.732143

Test set: Average loss: 1.7355, Accuracy: 3971/10000 (40%)

Train Epoch: 4 [0/50000 (0%)]   Loss: 1.790166
Train Epoch: 4 [6400/50000 (13%)]       Loss: 1.808102
Train Epoch: 4 [12800/50000 (26%)]      Loss: 1.711747
Train Epoch: 4 [19200/50000 (38%)]      Loss: 1.489907
Train Epoch: 4 [25600/50000 (51%)]      Loss: 1.953131
Train Epoch: 4 [32000/50000 (64%)]      Loss: 1.936282
Train Epoch: 4 [38400/50000 (77%)]      Loss: 1.509994
Train Epoch: 4 [44800/50000 (90%)]      Loss: 1.688193

Test set: Average loss: 1.7355, Accuracy: 4043/10000 (40%)

Train Epoch: 5 [0/50000 (0%)]   Loss: 1.561134
Train Epoch: 5 [6400/50000 (13%)]       Loss: 1.724658
Train Epoch: 5 [12800/50000 (26%)]      Loss: 1.679430
Train Epoch: 5 [19200/50000 (38%)]      Loss: 1.780755
Train Epoch: 5 [25600/50000 (51%)]      Loss: 1.725283
Train Epoch: 5 [32000/50000 (64%)]      Loss: 1.691037
Train Epoch: 5 [38400/50000 (77%)]      Loss: 1.765711
Train Epoch: 5 [44800/50000 (90%)]      Loss: 1.606187

Test set: Average loss: 1.7471, Accuracy: 3914/10000 (39%)

Train Epoch: 6 [0/50000 (0%)]   Loss: 1.804735
Train Epoch: 6 [6400/50000 (13%)]       Loss: 1.724678
Train Epoch: 6 [12800/50000 (26%)]      Loss: 1.785710
Train Epoch: 6 [19200/50000 (38%)]      Loss: 1.636965
Train Epoch: 6 [25600/50000 (51%)]      Loss: 1.642264
Train Epoch: 6 [32000/50000 (64%)]      Loss: 1.431316
Train Epoch: 6 [38400/50000 (77%)]      Loss: 1.594812
Train Epoch: 6 [44800/50000 (90%)]      Loss: 1.621047

Test set: Average loss: 1.7328, Accuracy: 3976/10000 (40%)

Train Epoch: 7 [0/50000 (0%)]   Loss: 1.616710
Train Epoch: 7 [6400/50000 (13%)]       Loss: 1.288997
```

```
Train Epoch: 7 [12800/50000 (26%)]      Loss: 1.590927
Train Epoch: 7 [19200/50000 (38%)]      Loss: 1.637883
Train Epoch: 7 [25600/50000 (51%)]      Loss: 1.854328
Train Epoch: 7 [32000/50000 (64%)]      Loss: 1.717840
Train Epoch: 7 [38400/50000 (77%)]      Loss: 1.756205
Train Epoch: 7 [44800/50000 (90%)]      Loss: 1.910950

Test set: Average loss: 1.7347, Accuracy: 3965/10000 (40%)
```

:: TASK 3.2 ::

Add code to create a convolutional network architecture as below.

- Convolution with 5 by 5 filters, 16 feature maps + Tanh nonlinearity.
- 2 by 2 max pooling.
- Convolution with 5 by 5 filters, 128 feature maps + Tanh nonlinearity.
- 2 by 2 max pooling.
- Flatten to vector.
- Linear layer with 64 hidden units + Tanh nonlinearity.
- Linear layer to 10 output units.

```
class ConvNet(nn.Module):
    ###########################
    #                         #
    #  WRITE YOUR CODE HERE   #
    #                         #
    ###########################

    def __init__(self):
        super(ConvNet, self).__init__()

        self.conv1 = torch.nn.Conv2d(3, 16, kernel_size=(5,5), stride = 1, padding = 0)
        self.conv2 = torch.nn.Conv2d(16, 128, kernel_size=(5,5), stride = 1, padding = 0)

        self.maxPool1 = torch.nn.MaxPool2d(2)
        self.maxPool2 = torch.nn.MaxPool2d(2)

        self.linear1 = nn.Linear(128*5*5, 64)
        self.linear2 = nn.Linear(64, 10)

    def forward(self, input):
        x = self.conv1(input)
        x = torch.tanh(self.maxPool1(x))

        x = self.conv2(x)
        x = torch.tanh(self.maxPool2(x))

        x = x.view(-1, 128*5*5)

        x = self.linear1(x)
        x = torch.tanh(x)
```

```
        output = self.linear2(x)


        return output
```

:: TASK 3.3 ::

Some of the functions in a CNN must be non-linear. Why?

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

WRITE YOUR ANSWER HERE

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

Otherwise, the CNN is simply a linear function, so it doesn't have any added value as compared to linear algorithm.

:: TASK 3.4 ::

Train the CNN for 20 epochs on the CIFAR-10 training set.

```
# Train
network=ConvNet()
network = network.to(device)
optimizer = optim.SGD(network.parameters(), lr=lr)
for epoch in range(1, 21):
    train(epoch, network)
    test(network)
```

```
Train Epoch: 1 [0/50000 (0%)]   Loss: 2.321141
Train Epoch: 1 [6400/50000 (13%)]       Loss: 2.102130
Train Epoch: 1 [12800/50000 (26%)]      Loss: 2.139366
Train Epoch: 1 [19200/50000 (38%)]      Loss: 1.953847
Train Epoch: 1 [25600/50000 (51%)]      Loss: 1.976726
Train Epoch: 1 [32000/50000 (64%)]      Loss: 1.790375
Train Epoch: 1 [38400/50000 (77%)]      Loss: 1.770024
Train Epoch: 1 [44800/50000 (90%)]      Loss: 1.854737
/usr/local/lib/python3.7/dist-packages/torch/nn/_reduction.py:42: UserWarning: siz
  warnings.warn(warning.format(ret))

Test set: Average loss: 1.8099, Accuracy: 3707/10000 (37%)

Train Epoch: 2 [0/50000 (0%)]   Loss: 1.668498
Train Epoch: 2 [6400/50000 (13%)]       Loss: 1.793822
Train Epoch: 2 [12800/50000 (26%)]      Loss: 1.840079
Train Epoch: 2 [19200/50000 (38%)]      Loss: 1.649476
Train Epoch: 2 [25600/50000 (51%)]      Loss: 1.747168
Train Epoch: 2 [32000/50000 (64%)]      Loss: 1.813799
Train Epoch: 2 [38400/50000 (77%)]      Loss: 1.864312
Train Epoch: 2 [44800/50000 (90%)]      Loss: 1.566545
```

```
Test set: Average loss: 1.6091, Accuracy: 4280/10000 (43%)

Train Epoch: 3 [0/50000 (0%)]   Loss: 1.568528
Train Epoch: 3 [6400/50000 (13%)]       Loss: 1.652087
Train Epoch: 3 [12800/50000 (26%)]      Loss: 1.674756
Train Epoch: 3 [19200/50000 (38%)]      Loss: 1.809367
Train Epoch: 3 [25600/50000 (51%)]      Loss: 1.522225
Train Epoch: 3 [32000/50000 (64%)]      Loss: 1.340142
Train Epoch: 3 [38400/50000 (77%)]      Loss: 1.341987
Train Epoch: 3 [44800/50000 (90%)]      Loss: 1.726366

Test set: Average loss: 1.4951, Accuracy: 4689/10000 (47%)

Train Epoch: 4 [0/50000 (0%)]   Loss: 1.381467
Train Epoch: 4 [6400/50000 (13%)]       Loss: 1.352101
Train Epoch: 4 [12800/50000 (26%)]      Loss: 1.121957
Train Epoch: 4 [19200/50000 (38%)]      Loss: 1.311195
Train Epoch: 4 [25600/50000 (51%)]      Loss: 1.458536
Train Epoch: 4 [32000/50000 (64%)]      Loss: 1.306559
Train Epoch: 4 [38400/50000 (77%)]      Loss: 1.280850
Train Epoch: 4 [44800/50000 (90%)]      Loss: 1.378633

Test set: Average loss: 1.4212, Accuracy: 4942/10000 (49%)

Train Epoch: 5 [0/50000 (0%)]   Loss: 1.416595
Train Epoch: 5 [6400/50000 (13%)]       Loss: 1.649141
Train Epoch: 5 [12800/50000 (26%)]      Loss: 1.302985
Train Epoch: 5 [19200/50000 (38%)]      Loss: 1.255085
Train Epoch: 5 [25600/50000 (51%)]      Loss: 1.211102
Train Epoch: 5 [32000/50000 (64%)]      Loss: 1.401728
Train Epoch: 5 [38400/50000 (77%)]      Loss: 1.241055
Train Epoch: 5 [44800/50000 (90%)]      Loss: 1.389104

Test set: Average loss: 1.3933, Accuracy: 5062/10000 (51%)

Train Epoch: 6 [0/50000 (0%)]   Loss: 1.606238
```

Note : The performance is clearly better (63% against 41%).

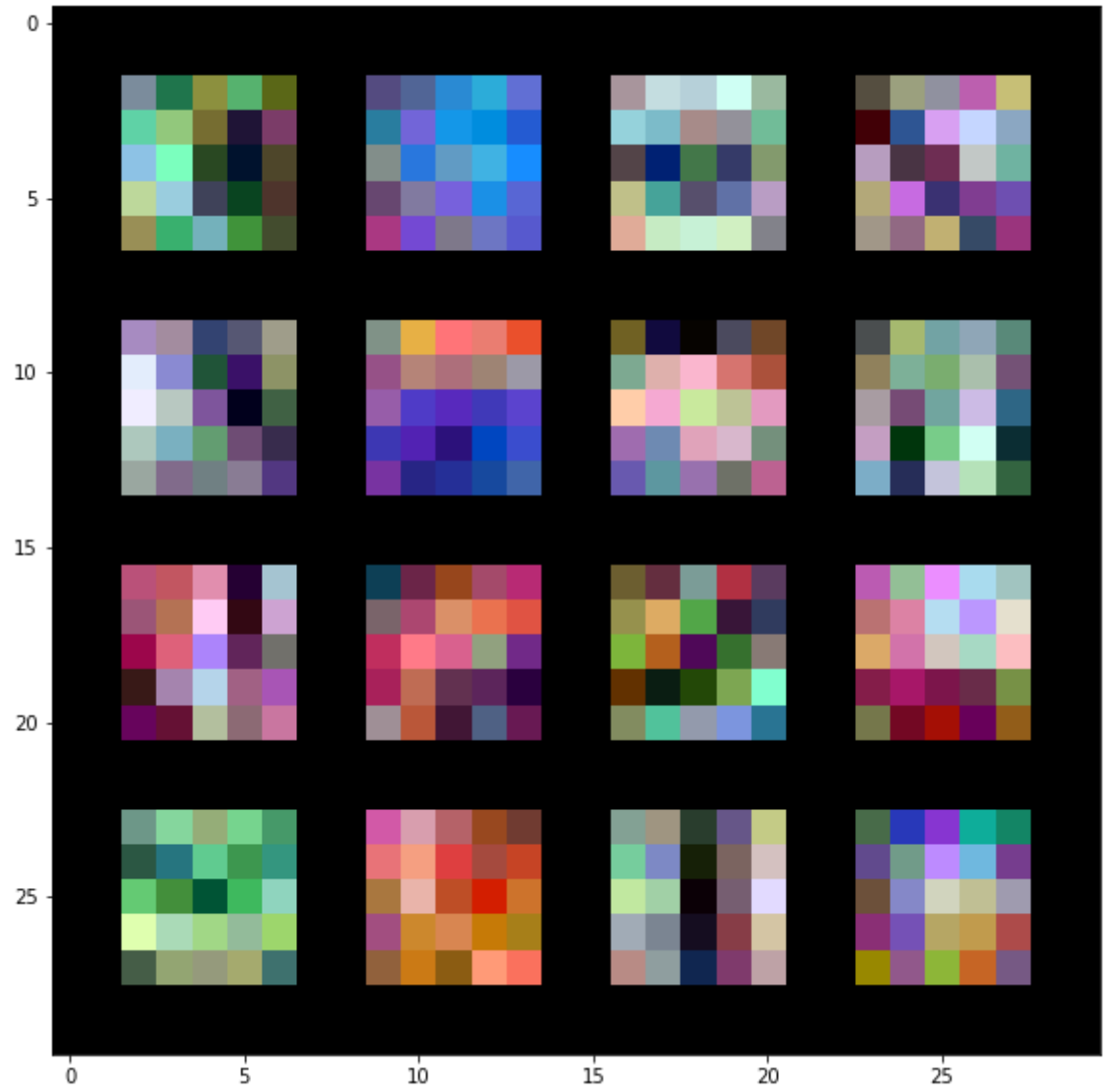
:: TASK 3.5 ::

Plot the first convolutional layer weights as images after the last epoch. (Hint threads: #1 #2)

```
############################
#                          #
#  WRITE YOUR CODE HERE    #
#                          #
############################

#Taken and modified from https://discuss.pytorch.org/t/understanding-deep-network-visualiz

w = network.conv1.weight.data.cpu()
grid = torchvision.utils.make_grid(w, nrow=4, normalize=True, scale_each=True)
```

```
plt.figure(figsize=(10, 10))
plt.imshow(grid.permute(1, 2, 0))
```

```
<matplotlib.image.AxesImage at 0x7fba832ffe50>
```



:: TASK 3.6 ::

What is the dimensionality of the weights at each layer? How many parameters are there in total in this CNN architecture?

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

WRITE YOUR ANSWER HERE

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

I didn't forget to add the bias.

First CNN layer -> weight shape = (3, 16, 5, 5) -> nb weights = 3x16x5x5 + 16 = 1216

Second CNN layer -> weight shape = (16, 128, 5, 5) -> nb weights = 51 328

First linear layer -> weight shape = (128x5x5, 64) -> nb weights = 128x5x5x64 + 64 = 204 864

Second linear layer -> weight shape = (64, 10) -> nb weights = 650

At the end of the day, we have 258058 weights (sum of the weights of each layer).

Useful resources

- PyTorch tutorial
- MNIST example

AUTHORSHIP STATEMENT

I declare that the preceding work was the sole result of my own effort and that I have not used any code or results from third-parties.

LACOMBE Yoach