COMPSCI 670: Computer Vision Fall 22
Homework 4

Prachi Jain

1 Short answers

1.1

Task where the Histogram-of-Oriented gradient representation might be more effective than a Bag-of-Words representation
The application which made the Histogram-of-Oriented gradient (HoG) technique mainstream is Human(Pedestrian) Detection in static images. This task was later expanded to common animals and vehicle detection in static images.

Recognizing three-dimensional objects in complex scenes like higher level representations in strong geometric models are computed from the 3D data in the form of surface features, parametric surface patches, or representations of parts, such as generalized cylinders, and are then compared to similar representations for the shape models.

Here, shape recognition techniques like HoG can accurately extract features while preserving the overall relative structure of the image.

For example, mobile robotic manipulation must reliably recognize objects over hundreds of images and accurately compute their positions. In addition, because this system is to be used by workers unfamiliar with the underlying technology, a high level of recognition accuracy is needed. Hence the HoG technique can be used here as it captures the overall shape . Recognition is performed on outdoor scenery with little control over the illumination and viewpoints. HoG is invariant to lights and small deformation, making it an ideal candidate.

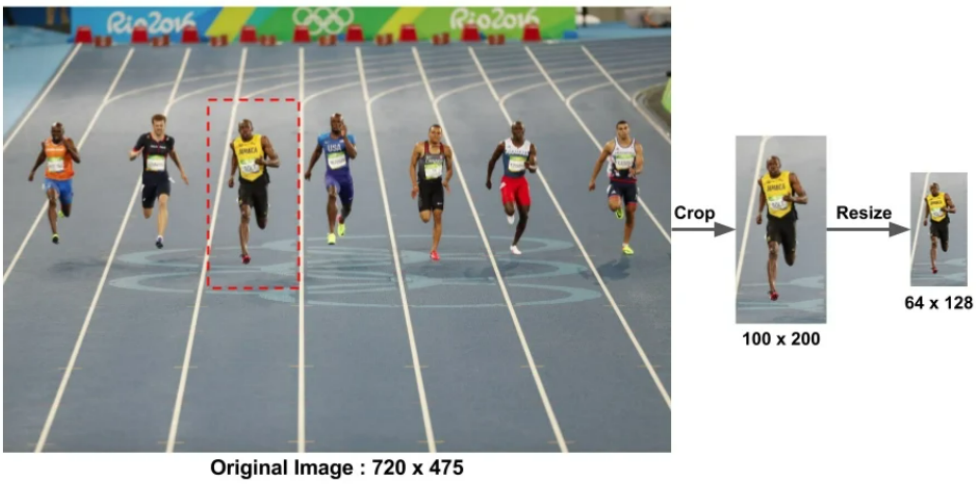
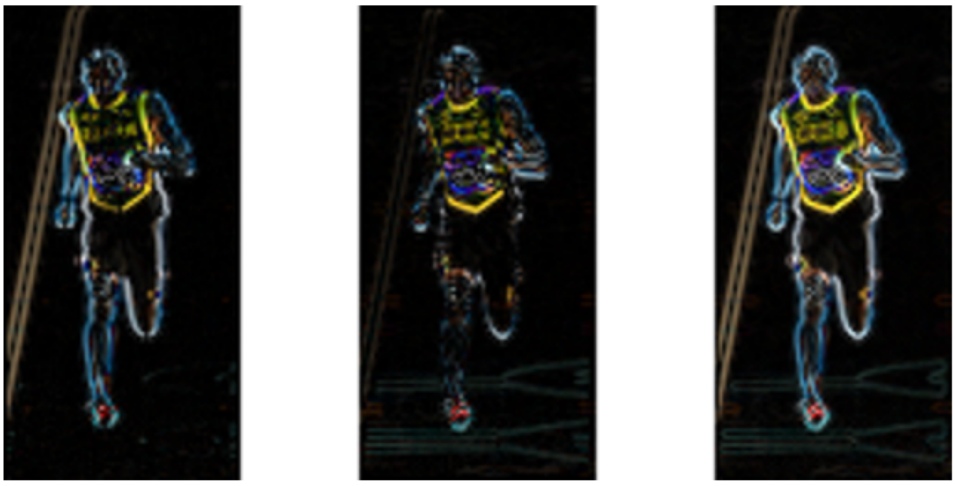
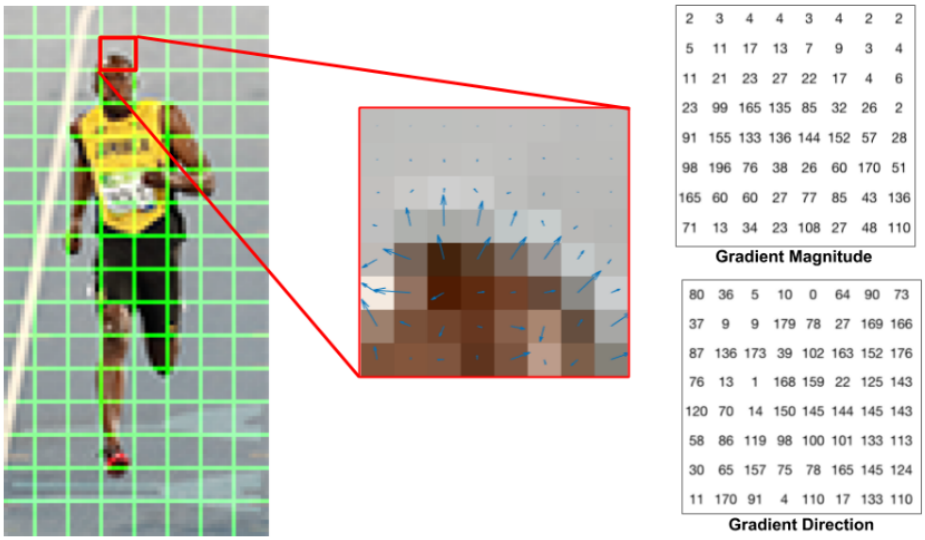


Figure 1: Image of size 720×475 is selected with a patch of size 100×200 for calculating HOG feature descriptor. The patch is cropped out of an image and resized to 64×128.



Left : Absolute value of x-gradient. Center : Absolute value of y-gradient. Right : Magnitude of gradient.

Figure 2: HoG Feature Descriptor Gradients Calculation



Center : The RGB patch and gradients represented using arrows. Right : The gradients in the same patch represented as numbers

Figure 3: Calculate Histogram of Gradients in 8×8 cells

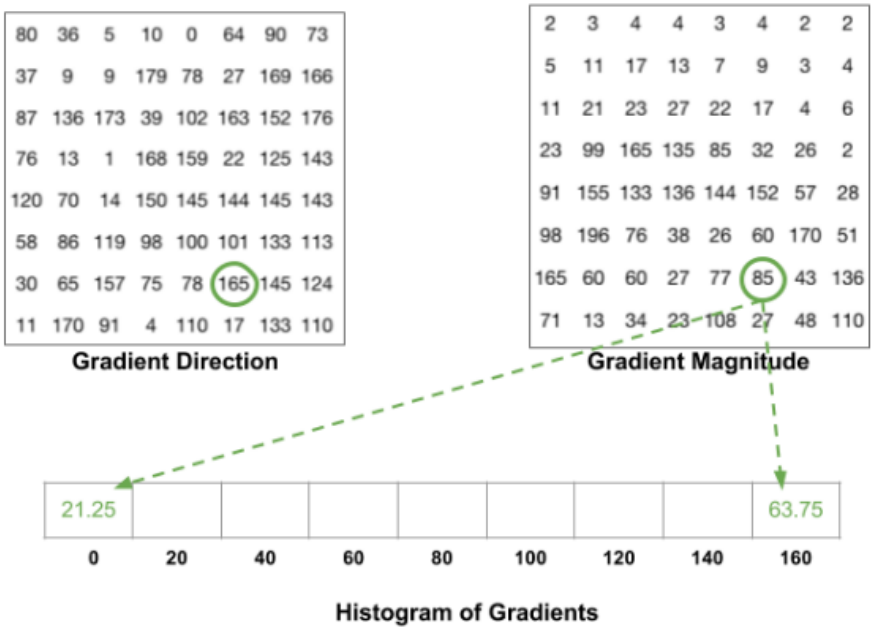


Figure 4: HoG Feature Descriptor Binning

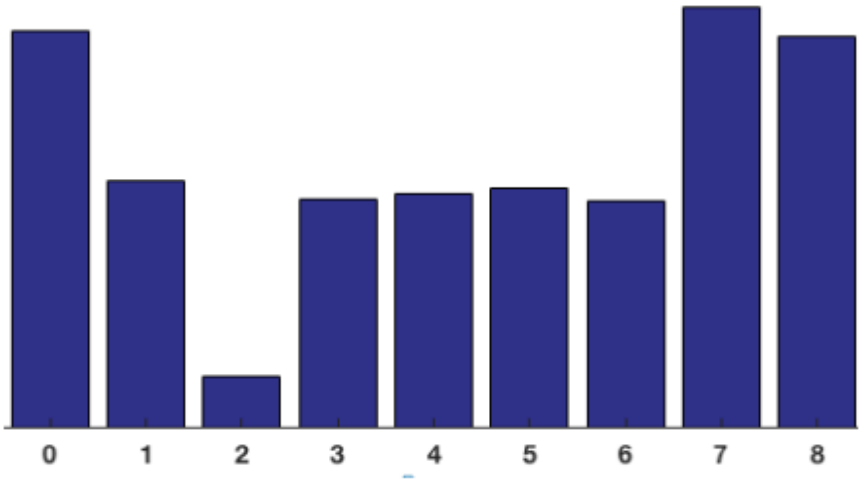


Figure 5: HoG Feature Descriptor Histogram Computation

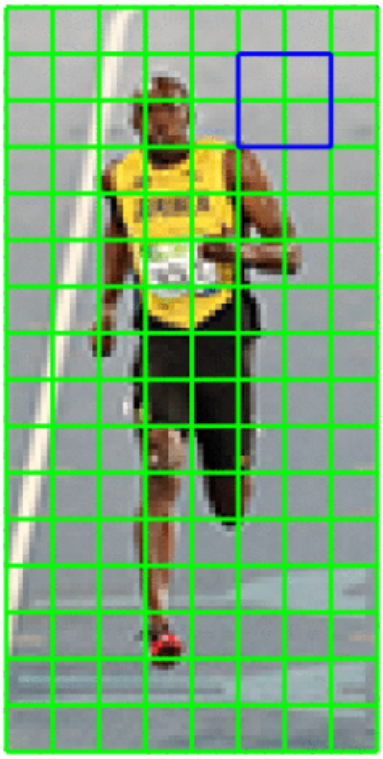


Figure 6: HoG Feature Descriptor 16×16 Block Normalization

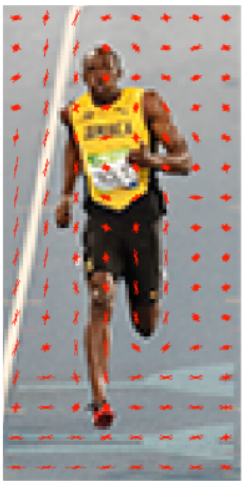


Figure 7: Visualizing Histogram of Oriented Gradients

Bag of visual words (BOVW) (also known as a bag of features or bag of key points) is commonly used in image classification. Its concept is adapted from information retrieval and NLP's bag of words (BOW). In bag of words (BOW), we count the number of each word that appears in a document, use the frequency of each word to know the keywords of the document, and make a frequency histogram from it. We treat a document as a bag of words (BOW). We have the same concept in the bag of visual words (BOVW), but instead of words, we use image features as the "words." Image features are unique patterns that we can find in an image.

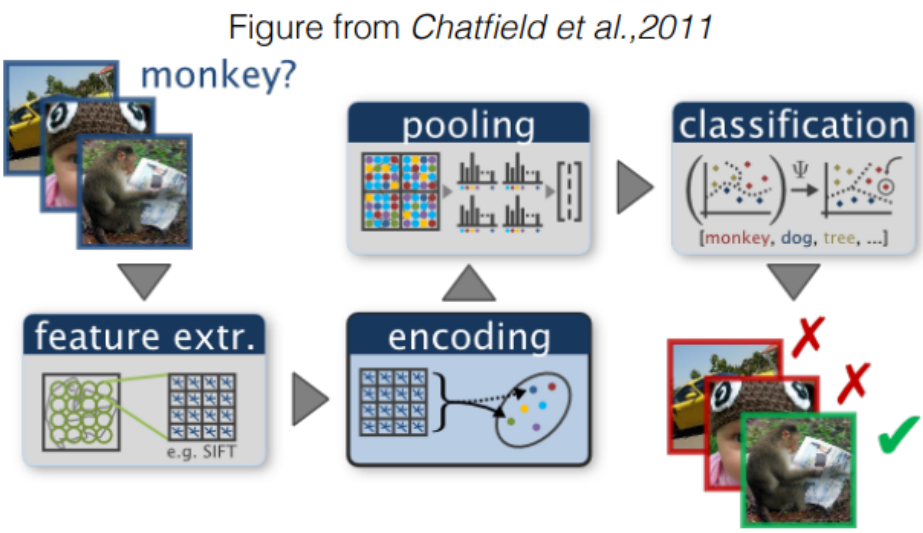


Figure 8: BOVW Algorithm pipeline : 1. Extracting local features 2. Learning a dictionary — clustering using k-means 3. Encoding methods — hard vs. soft assignment 4. Spatial pooling — pyramid representations

BOVW computes the distribution (histogram) of visual words (features) found in the query image and compares this distribution to those found in the training image. The biggest difference between instance recognition like the one using HoG is the absence of a geometric

verification stage since individual instances of generic visual categories have relatively little spatial coherence to their features.

The general idea of the bag of visual words (BOVW) is to represent an image as a set of features. Features consist of keypoints and descriptors. Keypoints are the "stand out" points in an image, so whether the image is rotated, shrunk, or expanded, its keypoints will always be the same. And descriptor is the description of the keypoint. We use the keypoints and descriptors to construct vocabularies and represent each image as a frequency histogram of features in the image. From the frequency histogram, later, we can find other similar images or predict the category of the image.

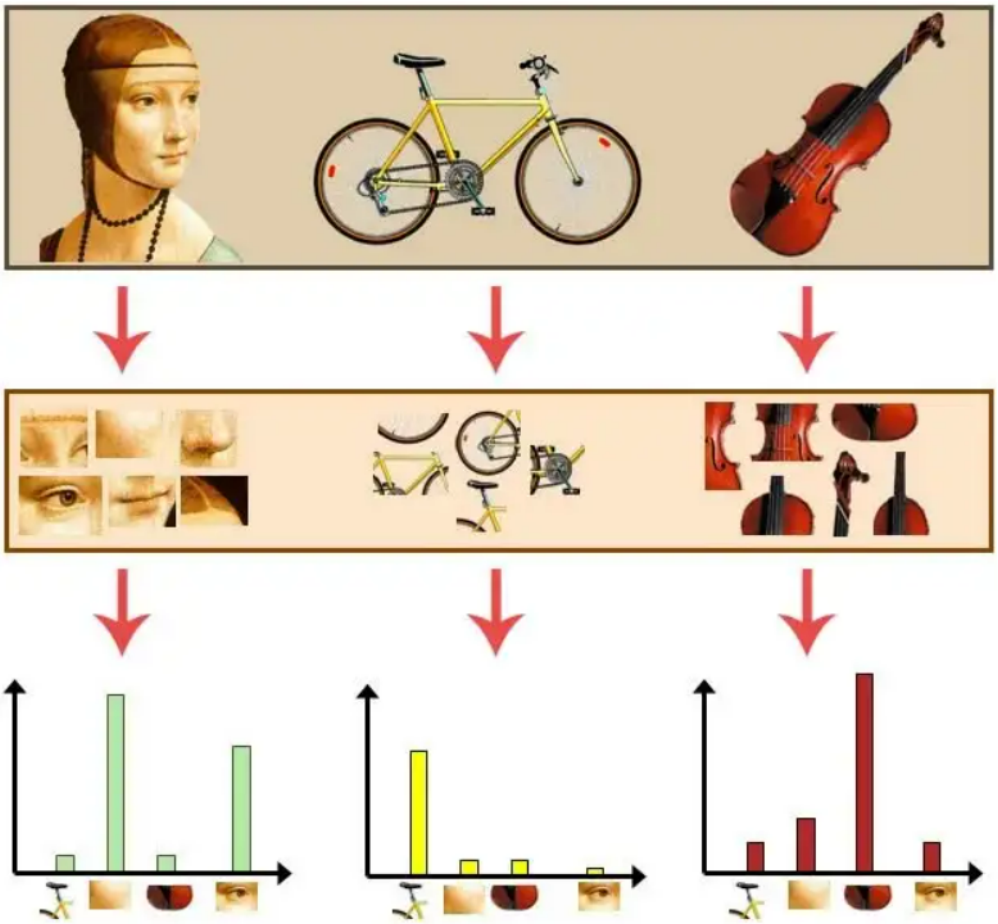


Figure 9: Histogram of visual words

Thus, the bag of visual words captures an image's local shape and texture. It does not preserve the spatial structure and is invariant to large translations.

HoG doesn't recognize textures because its implementation depends on gradients that aren't used much in detecting texture but only for detecting edges.

BOVW can be used in medical or biological images where segmentation of specific anatomical structures and the detection of lesions to differentiate between pathological and healthy tissue in different organs is required. Here HoG doesn't work for object recognition where recognition depends on different textures, which BOVW can easily do.

Many applications such as image compression, pre-processing, or segmentation require some information from the regions composing an image. Each region is characterized in terms of homogeneity (region composed with the same grey level or a single texture) and its type (textured or uniform). Again, in tasks where the decision criterion is based on the use of classical texture attributes (co-occurrence matrix and gray-levels moments), HoG cannot be used, and BOVW comes to the rescue.

1.2

Problem: Training linear classifiers and neural networks with gradient descent results in a practical issue with neural networks but not with linear models.

1. In Linear Classifier Models, weights initialization doesn't matter because there is a guaranteed convergence to global minima as long as the step size is suitably chosen since the objective function is convex.

2. Neural networks are sensitive to weights initialization

a. There exist many local minima, and Neural Networks tend to get stuck in these, so we need to be careful with weight initialization so that we reach the global minima. Hence, it is better to initialize them with many different (random) parameter values with enough variance. That "jumping out of a local minima" is not possible if we have initialized all weights to the same value(like zeros).

b. Symmetries: We need to reorder the hidden units and change the weights accordingly to get another network that produces identical outputs. If the learning process starts with the same weights, then all the neurons will follow the same gradient, same descent, and same weight (parameter value) updates. All the neurons in a layer will end up doing the same thing as/ imitating each other. That, of course, wouldn't train the neural network to classify between different inputs.

To elaborate on the second point, if we consider weights as priors in a Bayesian Network, we will multiply them to get a posterior distribution probability. That multiplication will result in all zeros if the priors are all zeros.

Back-prop identifies the set of updated weights/parameters that minimizes loss with prediction and actual output/objective function; with all weights to zero, we cannot orient the gradient descent algorithm in terms of determining the direction of the system as we will be placing ourselves on a saadle point of the parameter space. Back-propagation-based optimization algorithms will usually stop immediately at the saddle point.

To calculate the gradient, we multiply deltas with weights, and the result will always be zero. In each iteration of the back-propagation algorithm, we will update the weights by multiplying the existing weight by a delta determined by back-propagation. For example, if the initial weight value is 0, multiplying it by any value for delta will change the weights, but they all will change together the same, which means each iteration has no effect on the descent.

Initializing a network with all 0's in basic fully connected, feed-forward the network is equivalent to having one node per layer. Because in the hidden layer, all the nodes would have exactly the same inputs and would therefore stay the same as each other.

Solution: Basically, different weights/ parameter values; can break the hidden layer neuron symmetry. Hence we can train multiple networks with randomly initialized weights in the range [-1,1], HE Initialization (multiplies random initialization with sqrt(2/length of weight vector)) or Xavier Initilaization (multiplies random initialization with sqrt(1/length of weight vector))

1.3

Transfer Learning is reusing a pre-trained model to generalize from one task to another. Here the model uses the knowledge learned from a prior task to increase prediction about a new task that is closely related. In Transfer Learning training and testing distributions or tasks can be different but one can't expect much when the distributions are too different. Transfer Learning is a way to learn parameter-rich models on small datasets where the number of parameters is much greater than the number of data points. In short, Transfer Learning can train deep neural networks with a small amount of data.

Transfer Learning is highly valuable, as most real-world situations do not have millions of labeled data points to train complex Deep learning models. Data collection and labeling is an expensive and time-consuming task. Transfer learning can help develop rapid prototype deep learning models, saving lots of time and resource usage (like expensive GPUs).

Deep CNNs typically aim to detect edges in the first layer, forms in the middle layer, and task-specific features in the latter layers. The early and central layers of Deep CNNs are employed in transfer learning which outputs "off-the-shelf" features, and the latter layers are only retrained from those outputs.
Basically, very deep CNNs are trained on tasks where there are few labeled examples in the following way:
1. Train a model on ImageNet (Development of an Open Source Pre-trained Model by a Third Party on a particular related task)
2. Take outputs of an intermediate layer as features (Repurposing the Model)
3. Train linear classifier on these features (Generalization to another task/Fine Tuning for the task)

Transfer learning Advantages:
1. Training time reduction by reusing trained weights of deep CNNs
2. Simpler learning and more efficient neural network performance
3. There is no need for large datasets (Solving data scarcity and buying expensive proprietary data problems).

Disadvantage:
1. No end-to-end learning

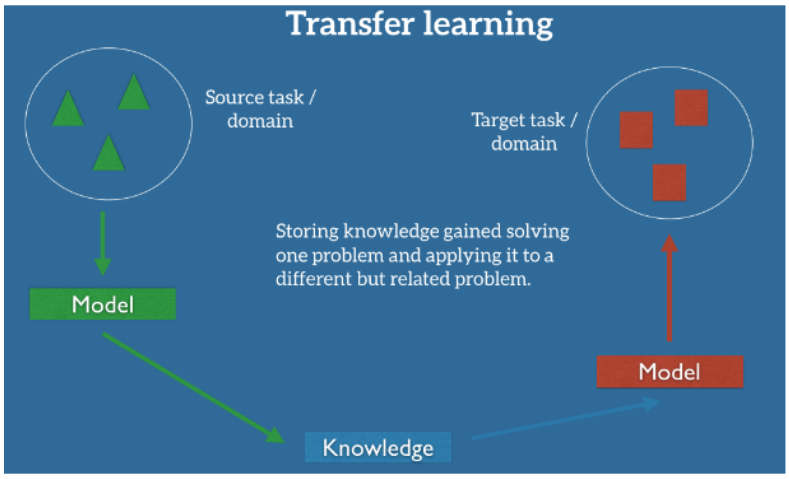


Figure 10: Transfer Learning

2 Classification on MNIST Dataset

2.1

Implementation of Handcrafted Representations -Pixel features, Histogram of oriented gradients (HoG) and Local binary patterns (LBP)

Please refer to implementation of *digitFeatures.py*

```
import numpy as np

#Helper Functions
def computeL2Norm(features):
    sumF = 0
    for i in range(len(features)):
        square = np.square(features[i])
        sumF += square
    features = features/(np.sqrt(sumF)+1e-6)
    return features

def calculateGradient(img, template):
    ts = template.size #Number of elements in the template

    #New padded array to hold the resultant gradient image
    newImg = np.zeros((img.shape[0]+ts-1, img.shape[1]+ts-1))
    newImg[np.uint16((ts-1)/2.0):img.shape[0]+np.uint16((ts-1)/2.0), np.uint16((ts-1)/2.0):img.shape[1]+np.uint16((ts-1)/2.0)] = img
    result = np.zeros((newImg.shape))

    for r in np.uint16(np.arange((ts-1)/2.0, img.shape[0]+(ts-1)/2.0)):
        for c in np.uint16(np.arange((ts-1)/2.0, img.shape[1]+(ts-1)/2.0)):
            currentRegion = newImg[r-np.uint16((ts-1)/2.0):r+np.uint16((ts-1)/2.0)+1, c-np.uint16((ts-1)/2.0):c+np.uint16((ts-1)/2.0)+1]
            currentResult = currentRegion * template
            score = np.sum(currentResult)
            result[r, c] = score

    #Result of the same size as the original image after removing the padding.
    resultImg = result[np.uint16((ts-1)/2.0):result.shape[0]-np.uint16((ts-1)/2.0), np.uint16((ts-1)/2.0):result.shape[1]-np.uint16((ts-1)/2.0)]
```

```python
    return resultImg

def gradientMagnitude(horizontalGradient, verticalGradient):
    gradMagnitude = np.sqrt(np.power(horizontalGradient, 2) + np.power(verticalGradient, 2))
    return gradMagnitude

def gradientDirection(horizontalGradient, verticalGradient):
    gradDirection = np.arctan(verticalGradient/(horizontalGradient + 1e-8))
    gradDirection = np.rad2deg(gradDirection)
    return gradDirection

def getFeature(x, featureType):
    if featureType == 'pixel':
        features = pixelFeatures(x)

    elif featureType == 'hog':
        features = hogFeatures(x)

    elif featureType == 'lbp':
        features = lbpFeatures(x)

    return features

def pixelFeatures(x):
    '''
    Takes x images as input and returns their pixel values as a feature
    Output is an array of size 784 x N where N is the number of images in x
    '''
    features = np.reshape(x, (784, x.shape[2]))

    #Chosen method works best - outperforms accuracy as mentioned in HW4
    method = 0

    if method == 0:
    #Performing standardization so that the raw pixel values when fed into the network does not
    #slows down the learning due to the erratic input nature.
        meanF = features.mean().astype(np.float32)
        stdF = features.std().astype(np.float32)
        features = (features - meanF)/(stdF)

    elif method == 1:
    #Square-root scaling
        features = np.sqrt(abs(features))

    elif method == 2:
    #L2-normalization
        features = computeL2Norm(features)
    return features

def hogFeatures(x):
    '''
    Histogram of oriented gradients (HoG)
    Takes x images as input and returns their HoG features
    '''
    #Define Parameters
    binSize = 4
```

```
    numOri = 8
    horizontalMask = np.array([-1, 0, 1])
    verticalMask = np.array([[-1],[0],[1]])

    numFeats = x.shape[0]
    numImages = x.shape[2]

    #Store features of all images
    features = []

    # For all images
    for image in range(numImages):

        y = x[:,:,image]

        #According to HOG paper of Dalal and Triggs
        #STEP 1: Logarithm of the intensity values(Non-linear Mapping)
        #Additional step for more invariance
        y = np.log(y + 1)

        #STEP 2: Compute Gradients Gx, Gy, magnitude and angle

        #Calculate Horizontal and Vertical Gradients
        Gx = calculateGradient(y, horizontalMask)
        Gy = calculateGradient(y, verticalMask)

        # Getting Magnitude and Direction of H and V Gradients
        # Angle lies between [-pi/2, pi/2]

        GMagnitude = gradientMagnitude(Gx, Gy)
        GDirection = gradientDirection(Gx, Gy)

        # Store features of each image, then concatenate it with other image features to form a single matrix
        blockFeature = []

        #STEP 3: Weighted Vote into spatial and orientation cells
        # For the grid of 7 x 7 => numFeats//binSize x numFeats//binSize => 28//4 x 28//4
        for i in range(numFeats//binSize):
            for j in range(numFeats//binSize):

                # For each cell get the magnitudes and directions for that cell
                magPatch = GMagnitude[binSize*i:binSize*i+binSize, binSize*j:binSize*j+binSize]
                angPatch = GDirection[binSize*i:binSize*i+binSize, binSize*j:binSize*j+binSize]

                # Store histogram for each cell here
                histogram  = np.zeros([8,1])

                # Binning Process (Bi-Linear Interpolation)
                for p in range(binSize):
                    for q in range(binSize):
                        mag = magPatch[p,q]
                        angle = angPatch[p,q]

```

```
                        #Assigning each pixel to the nearest orientation in numOri
                        #With vote proportional to the gradient magnitude

                        if angle>=-90 and angle<-67.5:
                            histogram[0] = histogram[0] + mag *(-67.5 - angle)/22.5
                            histogram[1] = histogram[1] + mag *(angle + 90)/22.5
                        elif angle>=-67.5 and angle<-45:
                            histogram[1] = histogram[1] + mag * (-45 - angle)/22.5
                            histogram[2] = histogram[2] + mag * (angle + 67.5)/22.5
                        elif angle>=-45 and angle<-22.5:
                            histogram[2] = histogram[2] + mag * (-22.5 - angle)/22.5
                            histogram[3] = histogram[3] + mag * (angle + 45)/22.5
                        elif angle>=-22.5 and angle<0:
                            histogram[3] = histogram[3] + mag * (0 - angle)/22.5
                            histogram[4] = histogram[4] + mag * (angle + 22.5)/22.5
                        elif angle>=0 and angle<22.5:
                            histogram[4] = histogram[4] + mag * (22.5 - angle)/22.5
                            histogram[5] = histogram[5] + mag *(angle - 0)/22.5
                        elif angle>=22.5 and angle<45:
                            histogram[5] = histogram[5] + mag *(45-angle)/22.5
                            histogram[6] = histogram[6] + mag *(angle-22.5)/22.5
                        elif angle>=45 and angle<67.5:
                            histogram[6] = histogram[6] + mag *(67.5 - angle)/22.5
                            histogram[7] = histogram[7] + mag *(angle - 45)/22.5
                        elif angle>=67.5 and angle<90:
                            histogram[7]= histogram[7] + mag *(90 - angle)/22.5
                            histogram[0]= histogram[0] + mag *(angle - 67.5)/22.5

                #Concatenate histograms
                blockFeature.append(histogram)

        #STEP 4: Local contrast normalization  of feature vector over overlapping spatial blocks
        #Additional step for more invariance
        blockFeature = np.array(blockFeature)
        #Chosen method works best - outperforms accuracy as mentioned in HW4
        method = 1

        if method == 0:
        #Performing standardization so that the raw pixel values when fed into the network does not
        #slows down the learning due to the erratic input nature.
            meanF = blockFeature.mean().astype(np.float32)
            stdF = blockFeature.std().astype(np.float32)
            blockFeature = (blockFeature - meanF)/(stdF)

        elif method == 1:
        #Square-root scaling
```

```
            blockFeature = np.sqrt(abs(blockFeature))

        elif method == 2:
        #L2-normalization
            blockFeature = computeL2Norm(blockFeature)

        #Concatenate HoG features of each image
        features.append(blockFeature.flatten())

    featuresT = np.array(features).T
    return featuresT

def lbpFeatures(x):
    '''
    Takes x images as input and returns a local binary pattern (LBP) histogram for each image
    '''
    height = x.shape[0]
    width = x.shape[1]
    channel = x.shape[2]
    patchSize = 3

    localBinaryPatternMap = np.array([[1, 2, 4],[8, 0, 16], [32, 64, 128]])
    localBinaryPattern = np.zeros((256, channel))

    #Each 3x3 patch is mapped to a number between 0 and 255 as follows:
    #Consider the pixels in a 3x3 patch.
    #Subtract from each pixel the value of the center pixel and assign them a bit value of
    #1 if the value of the pixel is greater than zero, and a bit value of 0 otherwise.
    #This gives us a 8-bit representation (excluding the center which is always 0) of every 3x3 patch,
    #corresponding to an integer between 0-255.
    for n in range(channel):
        for h in range(height-patchSize+1):
            for w in range(width-patchSize+1):
                localBinaryPattern[np.sum(localBinaryPatternMap * np.array(x[h:h+3,w:w+3,n] > x[h+1,w+1,n])), n] += 1

    features = localBinaryPattern
    #Chosen method works best - outperforms accuracy as mentioned in HW4
    method = 1

    if method == 0:
    #Performing standardization so that the raw pixel values when fed into the network does not
    #slows down the learning due to the erratic input nature.
        meanF = features.mean().astype(np.float32)
        stdF = features.std().astype(np.float32)
        features = (features - meanF)/(stdF)

    elif method == 1:
    #Square-root scaling
        features = np.sqrt(abs(features))

    elif method == 2:
    #L2-normalization
```

```
        features = computeL2Norm(features)

    return features

def zeroFeatures(x):
    return np.zeros((10, x.shape[2]))
```

2.1.1

Feature Normalization - Square-root scaling and L2-normalization
Please refer to implementation of *digitFeatures.py* in the above section

2.2

Multiclass linear model

2.2.1

Derive the expression for the ith component in the vector gradient of L with respect to w_k and write down the update rule for w_k using η for the step size

Assumption:
1. A Regularized Multi-Class Logistic Regression, has K different classes.
2. D is the number of training examples and N is the number of dimensions.
3. x is a n dimensional vector.
4. Let I_{dk} be an indicator function, where $I_{dk} = 1$ if $Y^d = k$, otherwise $I_{dk} = 0$ for a given class k.
5. For a K-way multiclass linear model (aka. logistic regression) the predicted probability of class k for an input x is given by:

$$\hat{y} = P(Y = k|X = x) = \left(\frac{exp(w_k^T x)}{\sum_{i=1}^{K} exp(w_i^T x)}\right)$$

where $w_1, w_2, ..., w_k$ are the model parameters.
6. Training minimizes the cross-entropy loss over the training set $f(x_i; y_i)$ plus a regularization term:

$$\min_{w_1,...,w_k} L = \min_{w_1,...,w_k} \sum_i l(y_i, \hat{y}_i) + \lambda \sum_k \|w_k\|^2$$

where $\hat{y}_i = P(Y \mid X = x_i)$ and $l(y, \hat{y})$ is the cross-entropy loss.
Thus, the Likelihood can be written as:

$$\min_{w_1,...,w_k} L = \min_{w_1,...,w_k} \prod_{d=1}^{D} \prod_{k=1}^{K} P(Y^d = k|X^d = x; w)^{I_{dk}}$$
$$= \min_{w_1,...,w_k} \prod_{d=1}^{D} \prod_{k=1}^{K} \left(\frac{exp(w_k^T x^d)}{\sum_i exp(w_i^T x^d)}\right)$$

Taking Natural Logarithm:

$$\min_{w_1,...,w_k} L = \min_{w_1,...,w_k} \sum_{d=1}^{D} \sum_{k=1}^{K} I_{dk} \left[ln \sum_i exp(w_i^T x^d) - w_k^T x^d \right]$$

Adding L_2 Regularization Term:

$$\min_{w_1,\dots,w_k} L = \min_{w_1,\dots,w_k} \sum_{d=1}^{D}\sum_{k=1}^{K} I_{dk}\left[ln \sum_i exp(w_i^T x^d) - w_k^T x^d\right] + \frac{\lambda}{2}\|w_k\|^2$$

Taking Derivative wrt w_i:

$$\frac{\partial l(w_1,\dots,w_k)}{\partial w_i} = \sum_{d=1}^{D}\left[\frac{x^d exp(w_i^T x^d)}{\sum_i exp(w_i^T x^d)} - I_{di}x^d\right] + \lambda w_i$$

$$\frac{\partial l(w_1,\dots,w_k)}{\partial w_i} = \sum_{d=1}^{D}\left[P(Y^d = i \mid X^d) - I_{di}\right]x^d + \lambda w_i$$

Therefore Update rule for weight vector wrt gradient descent is:

$$w_i = w_i - \eta \sum_{d=1}^{D}\left[P(Y^d = i \mid X^d) - I_{di}\right]x^d + \eta\lambda w_i$$

Thus the given solution converges to global minimum since it is a convex function.

2.2.2

Implementation of the linear model

Please refer to implementation of *linearModel.py*:

```
import numpy as np
import utils

def softmax(z):
  exp = np.exp(z - np.max(z))
  for i in range(len(z)):
    exp[i]/=np.sum(exp[i])
  return exp

def oneHotEncoding(y, classLabel):
    #Assigning one to relevant class labels each belonging to respective digit
    yEncode = np.zeros((len(y), classLabel))
    yEncode[np.arange(len(y)), y] = 1
    return yEncode

def KFoldCV(x,y):
    classLabels = np.unique(y)
    classID = []
    classSamples = []

    for i in range(max(classLabels)+1):
        indices = np.where(y==i)[0]
        classID.append(indices)
        classSamples.append(len(indices))

    trainFold = []
    valFold = []
    for fold in range(5):
```

```
        tempTrain = []
        tempVal = []
        for i in range(max(classLabels)+1):
            stepSize = int(classSamples[i]/5)
            start = fold * (stepSize)
            valClassID = classID[i][start:start + stepSize]
            trainClassID = list((set(classID[i]) - set(valClassID)))
            tempVal.extend(valClassID)
            tempTrain.extend(trainClassID)
        trainFold.append(tempTrain)
        valFold.append(tempVal)

    lambdaS = [0.1, 0.01]
    maxiterS = [2000, 2500, 3000]
    etaS = [0.1, 0.01, 0.05]
    output = {'maxiter': [],'lambda': [], 'eta': [], 'accuracy':[]}

    for m in maxiterS:
        for l in lambdaS:
            for e in etaS:
                modelParams = {'maxiter': m, 'lambda': l, 'eta': e}
                output['maxiter'].append(m)
                output['lambda'].append(l)
                output['eta'].append(e)
                avgAccuracy= 0
                for fold in range(5):
                    xTrain = np.apply_along_axis(lambda cv: cv[trainFold[fold]], 1, x)
                    yTrain = np.apply_along_axis(lambda cv: cv[trainFold[fold]], 0, y)
                    xTest = np.apply_along_axis(lambda cv: cv[valFold[fold]], 1, x)
                    yTest = np.apply_along_axis(lambda cv: cv[valFold[fold]], 0, y)
                    model = train(xTrain, yTrain, param = modelParams)
                    ypred = predict(model, xTest)
                    (acc, conf) = utils.evaluateLabels(yTest, ypred, False)
                    avgAccuracy += acc
                avgAccuracy = np.mean(avgAccuracy)
                output['accuracy'].append(avgAccuracy)
                print('5 Fold Cross Validation Accuracy ={}'.format(avgAccuracy))

    indices = np.argmax(np.array(output['accuracy']))
    bestParam = {'maxiter':output['maxiter'][indices],'lambda': output['lambda'][indices],'eta': output['eta'][indices]}
    return output, bestParam

def train(x, y, param = None):
    if param is None:
        param = {}
        param['lambda'] = 0.1       # Regularization term
        param['maxiter'] = 2500     # Number of iterations
        param['eta'] = 0.05         # Learning rate
        param['addBias'] = True     # Weather to add bias to features
    else:
        param['lambda'] = param['lambda']
        param['maxiter'] = param['maxiter']
```

```
        param['eta'] = param['eta']
        param['addBias'] = True
    return multiclassLRTrain(x, y, param)

def predict(model, x):
    #Returns highest probability class
    yHat = softmax(x.T@model['w'].T + model['b'])
    return np.argmax(yHat, axis=1)

def multiclassLRTrain(x, y, param):
    classLabels = np.unique(y)
    numClass = classLabels.shape[0]
    numFeats = x.shape[0]
    numData = x.shape[1]

    # Randomly initializes a model
    model = {}
    model['w'] = np.random.randn(numClass,numFeats) * 0.01
    if  param['addBias'] == True:
        model['b'] = np.random.randn(numClass) * 0.01
    else:
        model['b'] = np.zeros(numClass)

    model['classLabels'] = classLabels
    w = model['w']
    b = model['b']
    lossEpoch = []

    for epoch in range(param['maxiter']):
        z = x.T@w.T + b
        dW = (1/numData) * np.dot(x, softmax(z) - oneHotEncoding(y, numClass)).T + (2*param['lambda']/numData)*np.linalg.norm(w)
        dB = (1/numData) * np.sum(softmax(z) - oneHotEncoding(y, numClass))

        w = w - param['eta'] * dW
        b = b - param['eta'] * dB

        loss = -(np.log(softmax(z)[np.arange(len(y)), y])).mean() + ((1/numData)*param['lambda'] * np.square(np.linalg.norm(w)))
        lossEpoch.append(loss)
        #print('Epoch:{}.....................Loss:{:.4f}'.format(epoch+1, loss))

    model['w'] = w
    model['b'] = b
    return model
```

2.2.3

Training, Cross-Validation, Testing

Please refer to implementation of *testHandcrafted.py* below and *linearModel.py* in above secton.KFold cross validation (5 fold code in linearModel.py)is performed on training dataset and the parameters are hyper tuned to get maxiter = 2500 , eta = 0.05 and lamda = 0.1 from [2000, 2500, 3000], [0.1, 0.01, 0.05] and [0.1, 0.01] respectively.

```
{'maxiter': 2500, 'lambda': 0.1, 'eta': 0.05}
```

Figure 11: Handcrafted Features Best Grid Search -Parameter Estimation

```python
import numpy as np
import os
import utils
import time

import digitFeatures
import linearModel

# There are three versions of MNIST dataset
dataTypes = ['digits-normal.mat', 'digits-scaled.mat', 'digits-jitter.mat']

# You have to implement three types of features
featureTypes = ['pixel', 'hog', 'lbp']

# Accuracy placeholder
accuracy = np.zeros((len(dataTypes), len(featureTypes)))
trainSet = 1
testSet = 2

for i in range(len(dataTypes)):
    dataType = dataTypes[i]

    #Load data
    path = os.path.join('C:/Users/14136/OneDrive/Desktop/670 Computer vision/hw4_release/data', dataType)
    data = utils.loadmat(path)
    print('+++ Loading dataset: {} ({} images)'.format(dataType, data['x'].shape[2]))

    # To montage the digits in the val set use
    #utils.montageDigits(data['x'][:, :, data['set']==2])

    for j in range(len(featureTypes)):
        featureType = featureTypes[j]

        # Extract features
        tic = time.time()
        features = digitFeatures.getFeature(data['x'], featureType)

        print('{:.2f}s to extract {} features ({} dim).'.format(time.time()-tic, featureType, features.shape[0]))

        # Train model
        tic = time.time()

        #output, bestParam = linearModel.KFoldCV(features[:, data['set']==trainSet], data['y'][data['set']==trainSet])
        #print(bestParam) #{'maxiter': 2500, 'lambda': 0.1, 'eta': 0.05}

        model = linearModel.train(features[:, data['set']==trainSet], data['y'][data['set']==trainSet])
        print('{:.2f}s to train model.'.format(time.time()-tic))

```

```
        # Test the model
        tic = time.time()
        ypred = linearModel.predict(model, features[:, data['set']==testSet])
        print('{:.2f}s to test model.'.format(time.time()-tic))
        y = data['y'][data['set']==testSet]

        # Measure Accuracy and Display Confusion Matrix
        (acc, conf) = utils.evaluateLabels(y, ypred, True)
        print('Accuracy [testSet={}] {:.2f} %\n'.format(testSet, acc*100))
        accuracy[i, j] = acc


# Print the results in a table
print('+++ Accuracy Table [trainSet={}, testSet={}]'.format(trainSet, testSet))
print('--------------------------------------------------')
print('dataset\t\t\t', end="")
for j in range(len(featureTypes)):
    print('{}\t'.format(featureTypes[j]), end="")

print()
print('--------------------------------------------------')
for i in range(len(dataTypes)):
    print('{}\t'.format(dataTypes[i]), end="")
    for j in range(len(featureTypes)):
        print('{:.2f}\t'.format(accuracy[i, j]*100), end="")
    print()

# Once you have optimized the hyperparameters, you can report test accuracy
# by setting testSet=3. Do not optimize your hyperparameters on the
# test set. That would be cheating!
```

Results

```
+++ Accuracy Table [trainSet=1, testSet=2]
--------------------------------------------------
dataset                 pixel   hog     lbp
--------------------------------------------------
digits-normal.mat       86.20   92.80   10.00
digits-scaled.mat       82.60   93.00   10.00
digits-jitter.mat       15.20   38.00   10.00
```

Figure 12: Handcrafted Features Implementation on testset2 with no normalization learning rate =0.05 , Regularization =0.1 and maximum iterations = 2500

```
+++ Accuracy Table [trainSet=1, testSet=2]
--------------------------------------------------
dataset                 pixel   hog     lbp
--------------------------------------------------
digits-normal.mat       85.40   92.60   53.80
digits-scaled.mat       83.20   92.20   54.20
digits-jitter.mat       20.40   36.40   48.20
```

Figure 13: Handcrafted Features Implementation on testset2 with mean centred scaling normalization with learning rate =0.05 , Regularization = 0.1 and maximum iterations = 2500

```
+++ Accuracy Table [trainSet=1, testSet=2]
-------------------------------------------------
dataset                 pixel   hog     lbp
-------------------------------------------------
digits-normal.mat       87.40   93.40   73.00
digits-scaled.mat       82.20   93.60   73.40
digits-jitter.mat       15.60   38.20   68.60
```

Figure 14: Handcrafted Features Implementation on testset2 with square root scaling normalization with learning rate =0.05 , Regularization = 0.1 and maximum iterations = 2500

```
+++ Accuracy Table [trainSet=1, testSet=2]
-------------------------------------------------
dataset                 pixel   hog     lbp
-------------------------------------------------
digits-normal.mat       76.80   92.00   37.60
digits-scaled.mat       67.40   91.20   38.20
digits-jitter.mat       14.80   29.20   32.80
```

Figure 15: Handcrafted Features Implementation on testset2 with l2 normalization with learning rate =0.05 , Regularization = 0.1 and maximum iterations = 2500

```
+++ Accuracy Table [trainSet=1, testSet=2]
-------------------------------------------------
dataset                 pixel   hog     lbp
-------------------------------------------------
digits-normal.mat       85.60   93.40   73.00
digits-scaled.mat       83.40   93.40   73.20
digits-jitter.mat       21.40   38.40   69.00
```

Figure 16: Handcrafted Features Implementation on testset 2 with mean centred scaling normalization on pixel , square root scaling normalization HoG and Lbp with learning rate =0.05 , Regularization = 0.1 and maximum iterations = 2500

```
+++ Accuracy Table [trainSet=1, testSet=3]
-------------------------------------------------
dataset                 pixel   hog     lbp
-------------------------------------------------
digits-normal.mat       82.60   94.80   72.20
digits-scaled.mat       79.40   91.80   71.80
digits-jitter.mat       19.60   34.60   67.00
```

Figure 17: Handcrafted Features Implementation on testset 3 with mean centred scaling normalization on pixel , square root scaling normalization HoG and Lbp with learning rate =0.05 , Regularization = 0.1 and maximum iterations = 2500

2.3

Convolutional Neural Nets

2.3.1

Organize the data set to torch tensors compatible with neural network training
Please refer to implementation of *testCNN.py*:

```
import numpy as np
import os
import utils
import time
import digitFeatures
import linearModel
from CNNModel import CNN
from torch import optim
import torch.nn as nn
import torch
import pdb
import random

# There are three versions of MNIST dataset
dataTypes = ['digits-normal.mat', 'digits-scaled.mat', 'digits-jitter.mat']

# Accuracy placeholder
accuracy = np.zeros(len(dataTypes))
trainSet = 1
testSet = 2

iterations = 1000
batchSize = [40]#[40, 50, 100, 200]

for i in range(len(dataTypes)):
    dataType = dataTypes[i]

    #Load data
    path = os.path.join('..', 'data', dataType)
    data = utils.loadmat(path)
    print('+++ Loading dataset: {} ({} images)'.format(dataType, data['x'].shape[2]))

    # Organize into numImages x numChannels x width x height
    x = data['x'].transpose([2,0,1])
    x = np.reshape(x,[x.shape[0], 1, x.shape[1], x.shape[2]])
    y = data['y']
    # Convert data into torch tensors
    x = torch.tensor(x).float()
    y = torch.tensor(y).long() # Labels are categorical

    # Define the model
    model = CNN()
    model.train()
    #print(model)

    # Define loss function and optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr = 0.01)

    # Start training
    xTrain = x[data['set']==trainSet,:,:,:]
    yTrain = y[data['set']==trainSet]
```

```

    # Loop over training data in some batches
    # Implement this
    for epoch in range(iterations):
        batch = batchSize[random.randint(0, len(batchSize) - 1)]
        totalTrain = int(xTrain.shape[0]/batch)
        count = 0
        for k in range(0, xTrain.shape[0], batch):
          count = count + 1
          #Forward Prpogation
          optimizer.zero_grad()
          #Using permutations without replacement
          indices = torch.randperm(xTrain.shape[0])[k:k + batch]
          xBatch, yBatch = xTrain[indices], yTrain[indices]
          output = model(xBatch)
          loss = criterion(output, yBatch)

          #Backward Pass
          loss.backward()
          optimizer.step()
          #Logging
          if epoch%10 == 0:
            print("Epoch:{} Batch:{}/{}.............Loss={}".format(epoch+1, count, totalTrain, loss.item()))

    # Test model
    xTest = x[data['set']==testSet,:,:,:]
    yTest = y[data['set']==testSet]

    yPred = np.zeros(yTest.shape[0])
    model.eval() # Set this to evaluation mode

    # Loop over xTest and compute labels (implement this)
    yPredict = []
    with torch.no_grad():
      testOutput = model(xTest)
      predict, index = torch.max(testOutput.data, 1)
      yPredict = list(index)

    yPred = np.array(yPredict)

    # Map it back to numpy to use our functions
    yTest = yTest.numpy()
    (acc, conf) = utils.evaluateLabels(yTest, yPred, False)
    print('Accuracy [testSet={}] {:.2f} %\n'.format(testSet, acc*100))
    accuracy[i] = acc

# Print the results in a table
print('+++ Accuracy Table [trainSet={}, testSet={}]'.format(trainSet, testSet))
print('--------------------------------------------------')
print('dataset\t\t\t', end="")
print('{}\t'.format('cnn'), end="")
print()
print('--------------------------------------------------')
for i in range(len(dataTypes)):
    print('{}\t'.format(dataTypes[i]), end="")
    print('{:.2f}\t'.format(accuracy[i]*100))

```

```
# Once you have optimized the hyperparameters, you can report test accuracy
# by setting testSet=3. Do not optimize your hyperparameters on the
# test set. That would be cheating.
```

2.3.2

Implement a CNN architecture

Please refer to implementation of *CNNModel.py*:

```
import torch.nn as nn

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        #Convolution Blocks -> Convolution - BatchNorm - ReLU - MaxPool
        self.conv1 = nn.Sequential(
                        nn.Conv2d(in_channels = 1, out_channels = 16, kernel_size = 5, stride = 1, padding = 2),
                        nn.BatchNorm2d(num_features = 16),
                        nn.ReLU(),
                        nn.MaxPool2d(kernel_size = 2, stride = 2),
        )
        self.conv2 = nn.Sequential(
                        nn.Conv2d(in_channels = 16, out_channels = 32, kernel_size = 5, stride = 1, padding = 2),
                        nn.BatchNorm2d(num_features = 32),
                        nn.ReLU(),
                        nn.MaxPool2d(kernel_size = 2, stride = 2)
        )
        self.conv3 = nn.Sequential(
                        nn.Conv2d(in_channels = 32, out_channels = 64, kernel_size = 4, stride = 1, padding = 2),
                        nn.BatchNorm2d(num_features = 64),
                        nn.ReLU(),
                        nn.MaxPool2d(kernel_size = 2, stride = 2)
        )
        self.conv4 = nn.Sequential(
                        nn.Conv2d(in_channels = 64, out_channels = 128, kernel_size = 3, stride = 1, padding = 2),
                        nn.BatchNorm2d(num_features = 128),
                        nn.ReLU(),
                        nn.MaxPool2d(kernel_size = 2, stride = 2)
        )
        #Fully Connected layer
        self.fc = nn.Linear(1152, 10)

    def forward(self, x):
        h1 = self.conv1(x)
        h2 = self.conv2(h1)
        h3 = self.conv3(h2)
        h4 = self.conv4(h3)
        h5 = h4.view(h4.size(0), -1) # flatten the output of conv4 to (batch_size, 1152)
        output = self.fc(h5)
        return output
```

2.3.3

Define training loss and optimization procedure

1. The training loss metric assesses how a deep learning model fits the training data. It assesses the error of the model on the training set. Note that the training set is a portion of a dataset used to initially train the model. Computationally, the training loss is calculated by taking the sum of errors for each example in the training set. It is also important to note that the training loss is measured after each batch.

2. For optimizing the function torch.optim, was used. torch.optim is a package that implements numerous optimization algorithms. Several commonly used methods are already supported, and the interface is general enough so that more practical ones can also be easily integrated in the future.

3. The following optimizer was used as described in the homework - Adam with a learning rate of 0.01

4. Adam optimizer is an optimization technique used for machine learning and deep learning and comes under the gradient descent algorithm. - The optimizer is a combination of momentum and RMSP algorithms, which are gradient descent methodologies.

5. When working with a large problem that involves a lot of data, this method is really efficient for it.

6. It requires less memory and is efficient

7. The optimizer is relatively easy to configure where the default configuration parameters do well on most problems.

2.3.4

Iterate over training data in mini-batches and optimize the loss function to train the model

Mini-batches of 40, 50, 10 and 200 are used.
Please refer to implementation of *testCNN.py* in section 2.3.1

2.3.5

Model Evaluation

Please refer to implementation of *testCNN.py* in section 2.3.1

```
+++ Accuracy Table [trainSet=1, testSet=2]
-------------------------------------------------
dataset                 cnn
-------------------------------------------------
digits-normal.mat       93.00
digits-scaled.mat       91.60
digits-jitter.mat       90.20
```

Figure 18: CNN Evaluation with Test Set = 2 and Batch Size = 40

```
+++ Accuracy Table [trainSet=1, testSet=2]
-------------------------------------------------
dataset                 cnn
-------------------------------------------------
digits-normal.mat       96.20
digits-scaled.mat       91.40
digits-jitter.mat       88.80
```

Figure 19: CNN Evaluation with Test Set = 2 and Batch Size = 50

```
+++ Accuracy Table [trainSet=1, testSet=2]
-------------------------------------------------
dataset                 cnn
-------------------------------------------------
digits-normal.mat       93.20
digits-scaled.mat       82.40
digits-jitter.mat       88.60
```

Figure 20: CNN Evaluation with Test Set = 2 and Batch Size = 100

```
+++ Accuracy Table [trainSet=1, testSet=2]
-------------------------------------------------
dataset                 cnn
-------------------------------------------------
digits-normal.mat       95.40
digits-scaled.mat       90.60
digits-jitter.mat       85.20
```

Figure 21: CNN Evaluation with Test Set = 2 and Batch Size = 200

```
+++ Accuracy Table [trainSet=1, testSet=3]
-------------------------------------------------
dataset                 cnn
-------------------------------------------------
digits-normal.mat       95.20
digits-scaled.mat       90.40
digits-jitter.mat       88.00
```

Figure 22: CNN Evaluation with Test Set = 3 - Learning Rate = 0.01 Batch Size = 50 as it performed best than the other chosen mini batches

2.4

Answer the Following Questions:

2.4.1

Describe your experiments with handcrafted features. In particular, which ones worked well, what hyperparameters were effective, how important was normalization, and why particular features were effective on each type of MNIST
The following experiments were performed

1. K Fold Cross Validation (5 Fold) and Best Grid Search on training set = 1 and test set =2 (validation set) to get the following most effective hyperparamters maxiter = 2500 , eta = 0.05 and lamda = 0.1 from [2000, 2500, 3000], [0.1, 0.01, 0.05] and [0.1, 0.01] respectively.
Normalization can help training of our neural networks as the different features are on a similar scale, which helps to stabilize the gradient descent step, allowing us to use larger learning rates or help models converge faster for a given learning rate. Therfore, normalisation/scaling is typically recommended and sometimes very important. Especially for neural networks, normalisation can be very crucial because when you input unnormalised inputs to activation functions, you can get stuck in a very flat region in the domain and may not learn at all.
Following Normalization were performed:
1. Mean Centred Normalization: Performed standardization so that the raw pixel values when fed into the network does not slows down the learning due to the erratic input nature.

2. Sqaure Root Scaling: A square root transformation can be useful for:
1. Normalizing a skewed distribution
2. Transforming a non-linear relationship between 2 variables into a linear one
3. Reducing heteroscedasticity of the residuals in neural network
4. Focusing on visualizing certain parts of your data

3. L2 Normalisation: We scale the values so that if they were all squared and summed, the value would be 1 (Unit scaling)
We found Mean Centred Normalization to work well on pixel features and square root scaling to work well on HoG and LBP features.However l2 normalisation did not work well because it performs unit scaling and therefore the distinctive features of the dataset is lost in that process.
HoG features were more effective on normal and scaled dataset because HoG can accurately extract features while preserving the overall relative structure of the image and it is able to capture well the overall shape of the image and is invariant to small deformation(scaling in this sense) and lightning.
LBP features respond well to jitter dataset because in jitter dataset there is translation horizontally , vertically and sometimes in as well which is beyond the scope of HoG feature detection. Moreover LBP is gray-scale invariant texture measure, encode geometric features of an image by detecting edges, corners, raised or flat areas and hard line as it works on the finer levels of the features and not just an overall shape.

2.4.2

Describe your experiments with CNNs. What architecture worked well, how important was batch normalization, etc.
After much exploration the following architecture worked well:
1. Convolution2D(in_channels = 1, out_channels = 16, kernel_size = 5, stride = 1, padding = 2) -> Batch Normalization -> ReLU Activation Function -> Max Pooling
2. Convolution2D(in_channels = 16, out_channels = 32, kernel_size = 5, stride = 1, padding = 2) -> Batch Normalization -> ReLU Activation Function -> Max Pooling
3. Convolution2D(in_channels = 32, out_channels = 64, kernel_size = 4, stride = 1, padding = 2) -> Batch Normalization -> ReLU Activation Function -> Max Pooling
4. Convolution2D(in_channels = 64, out_channels = 128, kernel_size = 3, stride = 1, padding = 2) -> Batch Normalization -> ReLU Activation Function -> Max Pooling
5. Fully Connected layer(128*3*3, 10)

As observed each convolution block above has a layer of batch normalization. Deeper Architectures involving 5 or 6 convolution blocks were attempted but those architectures gave lower accuracy on all three types of datasets. Shallow architectures outperformed accuracy in normalised and scaled datasets with 97 percent and 91.80 percent respectively but failed at jitter dataset achieving only 78 percent.

Batch Normalization is a technique done between the layers of a Neural Network instead of in the raw data. The key idea is to normalize so that each layer output has zero mean and unit variance. It is done along mini-batches instead of the entire data set. It is performed during training by computing the mean and variance for each channel and aggregating over batch, followed by subtracting from the mean and dividing by standard deviation.
The advantage of using different mini-batch sizes, such as 40,50, 100 and 200 in our case, was to speed up training and explore higher learning rates, such as 0.1 and 1.0 (which gave bad results, a deviation of more than 10 percent from reported accuracies in the homework), but it made learning easier.

3 Image Denoising with a Deep Image Prior

3.1

Network architecture and random image
Implementing the Deep Image Prior by *dip.py*

```
import numpy as np
import torch
import torch.nn as nn

class EncDec(nn.Module):
    def __init__(self):
        super(EncDec, self).__init__()
        self.ConvBlocks = nn.Sequential(
            nn.Conv2d(in_channels = 1, out_channels = 16, kernel_size = 3, stride = 2, padding = 1),
            nn.BatchNorm2d(num_features = 16),
            nn.ReLU(True),

            nn.Conv2d(in_channels = 16, out_channels = 32, kernel_size = 3, stride = 2, padding = 1),
            nn.BatchNorm2d(num_features = 32),
            nn.ReLU(True),

            nn.Conv2d(in_channels = 32, out_channels = 64, kernel_size = 3, stride = 2, padding = 1),
            nn.BatchNorm2d(num_features = 64),
            nn.ReLU(True)
        )

        self.DeconvBlocks = nn.Sequential(
            nn.Upsample(mode = 'bilinear', scale_factor = 2),
            nn.Conv2d(in_channels = 64,out_channels = 32, kernel_size = 3, stride = 1, padding = 1),
            nn.BatchNorm2d(num_features = 32),
            nn.ReLU(True),
```

```
            nn.Upsample(mode='bilinear', scale_factor = 2),
            nn.Conv2d(in_channels = 32, out_channels = 16, kernel_size = 3, stride = 1, padding = 1),
            nn.BatchNorm2d(num_features = 16),
            nn.ReLU(True),

            nn.Upsample(mode='bilinear', scale_factor =2),
            nn.Conv2d(in_channels = 16, out_channels = 1, kernel_size = 3, stride = 1, padding = 1),
            nn.ReLU(True)
        )

    def forward(self, x):
        x = self.ConvBlocks(x)
        x = self.DeconvBlocks(x)
        return x
```

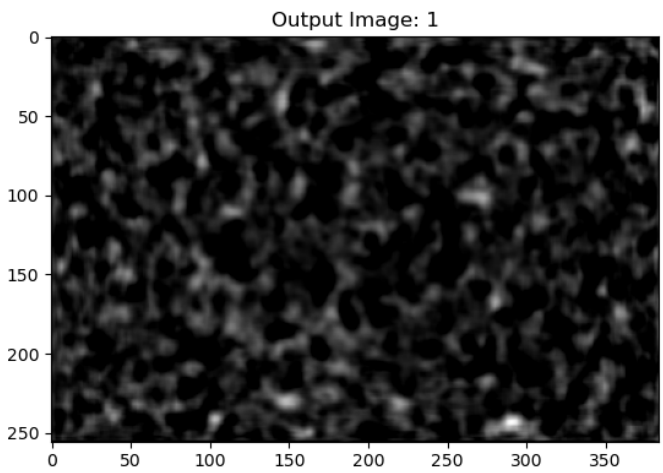


Figure 23: Random Saturn Image 1 by sampling different random inputs and network parameters.

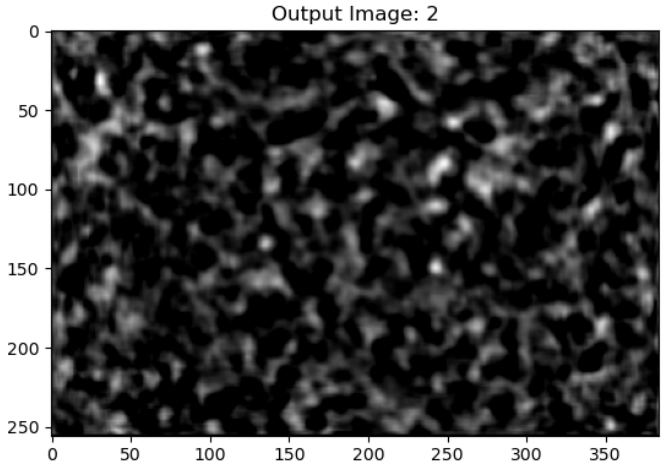


Figure 24: Random Saturn Image 2 by sampling different random inputs and network parameters.

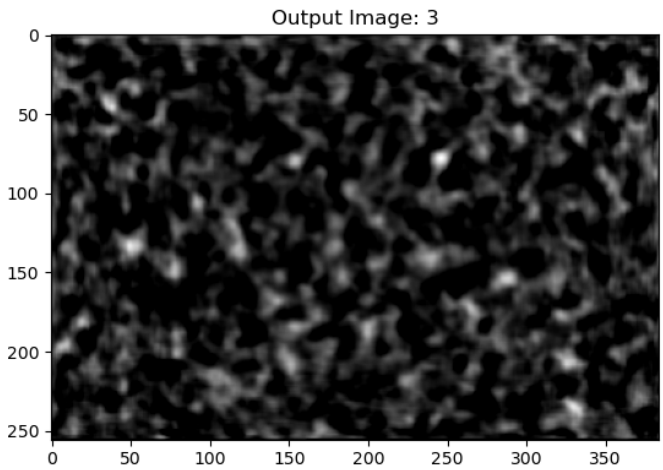


Figure 25: Random Saturn Image 3 by sampling different random inputs and network parameters.

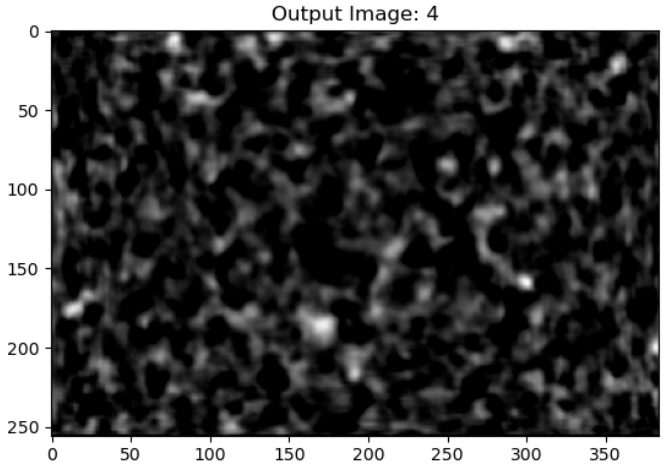


Figure 26: Random Saturn Image 4 by sampling different random inputs and network parameters.

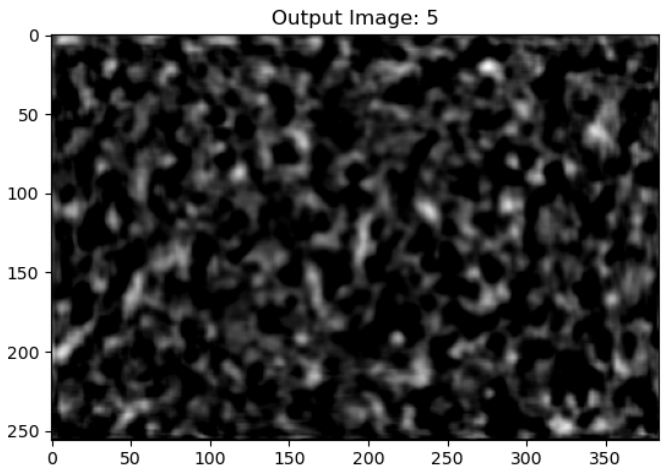


Figure 27: Random Saturn Image 5 by sampling different random inputs and network parameters.

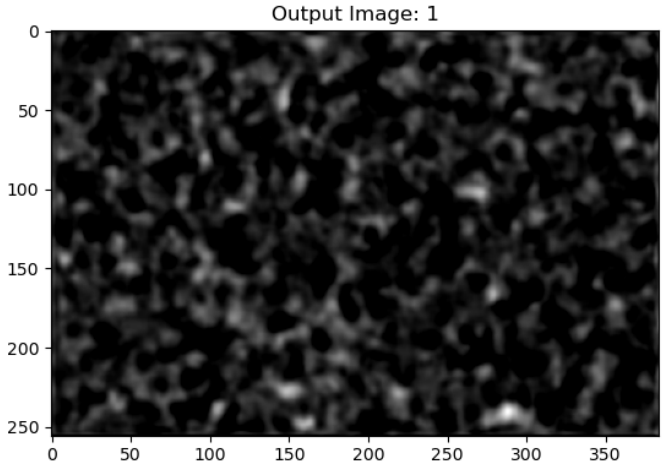


Figure 28: Random Lena Image 1 by sampling different random inputs and network parameters.

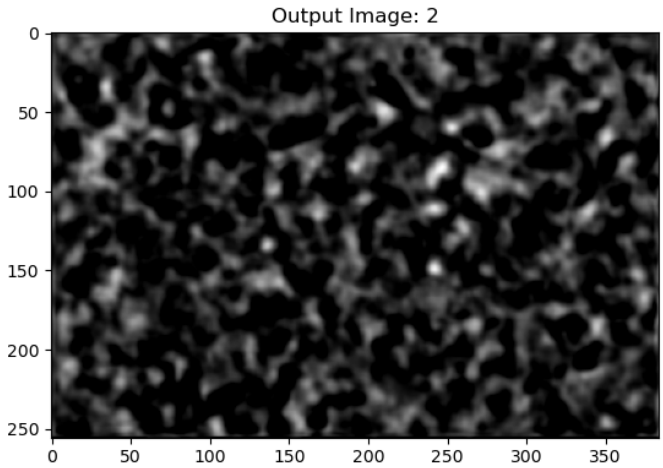


Figure 29: Random Lena Image 2 by sampling different random inputs and network parameters.

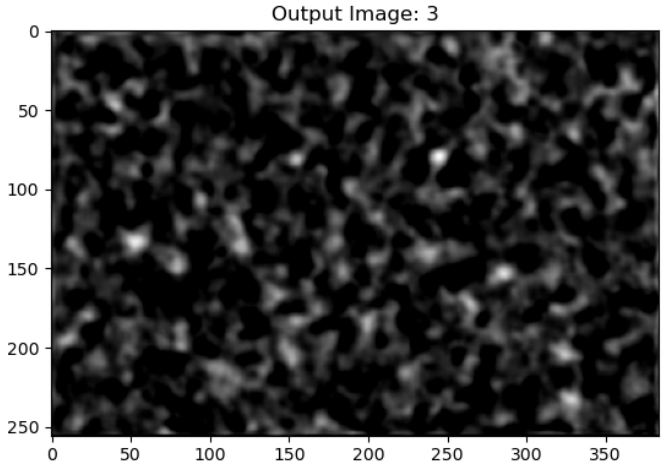


Figure 30: Random Lena Image 3 by sampling different random inputs and network parameters.

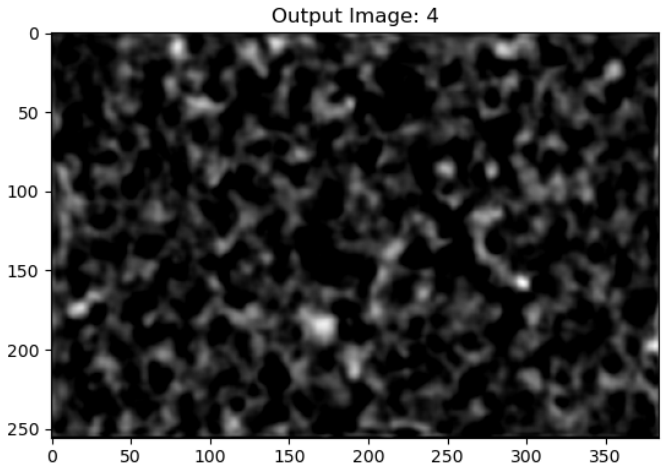


Figure 31: Random Lena Image 4 by sampling different random inputs and network parameters.

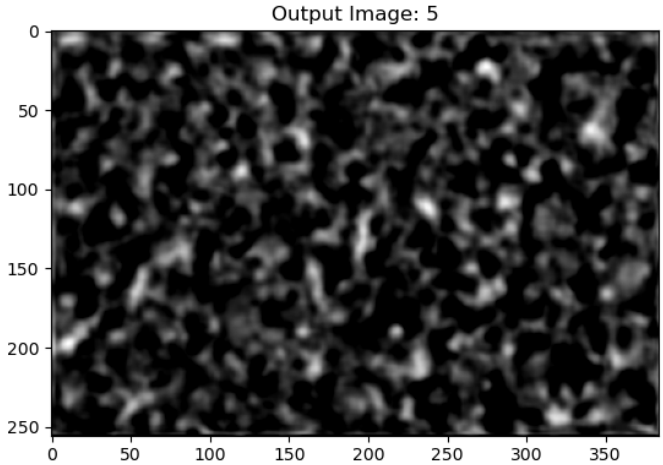


Figure 32: Random Lena Image 5 by sampling different random inputs and network parameters.

3.2

Using the deep image prior

Implementing the Deep Image Prior by *evalDenoising.py*

```
import os
import time
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import sys
import torch
import torch.optim as optim
import torch.nn as nn
from torch.autograd import Variable
from dip import EncDec
from utils import imread
from utils import gaussian
from scipy import ndimage
from scipy import signal

torch.manual_seed(123)

# Load clean and noisy image
#im = imread('../data/denoising/saturn.png')
#noise1 = imread('../data/denoising/saturn-noisy.png')
im = imread('../data/denoising/lena.png')
noise1 = imread('../data/denoising/lena-noisy.png')

error1 = ((im - noise1)**2).sum()
print('Noisy image SE: {:.2f}'.format(error1))

plt.figure(1)

# Original image
plt.subplot(141)
```

```
plt.imshow(im, cmap='gray')
plt.title('Input')

# Noisy version
plt.subplot(142)
plt.imshow(noise1, cmap='gray')
plt.title('Noisy image SE {:.2f}'.format(error1))

# Apply Gaussian filter
plt.subplot(143)
gaussian_filter = gaussian(7, 2)
im_gauss = ndimage.convolve(noise1, gaussian_filter, mode="nearest")
error_gauss = ((im - im_gauss)**2).sum()
plt.imshow(im_gauss, cmap='gray')
plt.title('Gaussian SE {:.2f}'.format(error_gauss))

# Apply Median filter
plt.subplot(144)
im_med = signal.medfilt(noise1, 7)
error_med = ((im - im_med)**2).sum()
plt.imshow(im_med, cmap='gray')
plt.title('Median SE {:.2f}'.format(error_med))
plt.show(block=True)

##############################################################################
# Denoising algorithm (Deep Image Prior)
##############################################################################

#Create network
net = EncDec()

# Loads noisy image and sets it to the appropriate shape
noisy_img = torch.FloatTensor(noise1).unsqueeze(0).unsqueeze(0).transpose(2, 3)
clean_img = torch.FloatTensor(im).unsqueeze(0).unsqueeze(0).transpose(2,3)

#Show five images generated by the network in your report by sampling different
    random inputs and network parameters
for i in range(5):
    eta = torch.randn(*noisy_img.size())
    eta.detach()
    out = net(eta)
    out_img = out[0, 0, :, :].transpose(0,1).detach().numpy()
    plt.imshow(out_img, cmap='gray')
    plt.title('Output Image: {}'.format(i+1))
    plt.show()

#Creates eta (noisy input)
eta = torch.randn(*noisy_img.size())
eta.detach()

###
# Your training code goes here.
###

iteration = 500
lr = 0.01 #0.01 for Saturn and 0.005 for Lena
criterion = nn.MSELoss(reduction = 'sum')
optimizer = torch.optim.Adam(net.parameters(), lr = lr)
```

```
eTrain = np.zeros((iteration, 1))
eTest = np.zeros((iteration, 1))

def monitorTestLoss(net, imgInput, imgDestination):
    net.eval()
    lossTest = 0
    with torch.no_grad():
        output = net(imgInput)
        lossTest = criterion(output, imgDestination)
    return lossTest

for epoch in range(iteration):
    net.train()
    eta = Variable(eta)
    output = net(eta)
    loss = criterion(output, noisy_img)
    eTrain[epoch] = loss.data
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    eTest[epoch] = monitorTestLoss(net, eta, clean_img)
    print('Epoch {}.....................Train Loss:{:.4f}'.format(epoch + 1,
    loss.data.item()))
    earlyStop = 405 #405 for saturn and 480 for lena
    #if epoch == earlyStop: #Uncomment to perform Early Stopping
    #   break               #Uncomment to perform Early Stopping

print('Early Stopping Indices:', np.where(eTest == np.amin(eTest)))
print('Early Stopping Error Value:', np.amin(eTest))

out = net(eta)
out_img = out[0, 0, :, :].transpose(0,1).detach().numpy()

plt.figure(5)
plt.axis('off')

plt.subplot(151)
plt.imshow(im, cmap='gray')
plt.title('Input Image')

plt.subplot(152)
plt.imshow(noise1, cmap='gray')
error1 = ((im - noise1)**2).sum()
plt.title('Noisy Image Squared Error {:.2f}'.format(error1))

plt.subplot(153)
plt.imshow(out_img, cmap='gray')
error2 = ((im - out_img)**2).sum()
plt.title('Output Image Squared Error {:.2f}'.format(error2))

plt.subplot(154)
gaussian_filter = gaussian(7, 2)
im_gauss = ndimage.convolve(noise1, gaussian_filter, mode="nearest")
error_gauss = ((im - im_gauss)**2).sum()
plt.imshow(im_gauss, cmap='gray')
plt.title('Gaussian Squared Error {:.2f}'.format(error_gauss))

plt.subplot(155)
```

```
im_med = signal.medfilt(noise1, 7)
error_med = ((im - im_med)**2).sum()
plt.imshow(im_med, cmap='gray')
plt.title('Median Squared Error {:.2f}'.format(error_med))
plt.show()

fig = plt.gcf()
sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
plt.xscale("log")
plt.plot(np.arange(1, iteration + 1), eTrain, marker='o', linestyle='dashed',
    label='Training Error')
plt.plot(np.arange(1, iteration + 1), eTest, marker='o', linestyle='dashed',
    label='Testing Error')
plt.legend(prop={'size': 22})
plt.xlabel('Iterations', fontsize=22)
plt.ylabel('Loss Function: Squared Error', fontsize=22)
plt.title(r'Training & Testing Error', fontsize=22)
plt.show()
```

Results for both the images as well as the error with respect to the reference and comparison to denoising using Gaussian and Median Filtering on the same images

1. Early Stopping:

By using early stopping, the encoder decoder model is able to prevent the **No denoising** situation where the training error of the model tends to go to zero and we get the minimum squared error on the testing error. This results into making the output image least noisy and at the same time as close to the clean image as is the capcaity of the model. The output images of both Saturn and Lena can confirm least noisiness and minimum squared error characteristics while retaining their respective distinctive features.

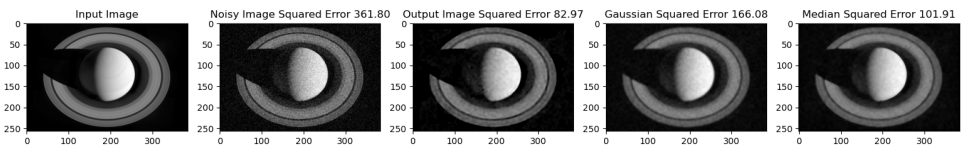


Figure 33: Saturn Image optimized and comparison with denoising using Gaussian and Median Filtering using learning rate = 0.01 and early stopping at 405 iteration

Figure 34: Saturn image at 405 epoch Plot: Training and Testing error as a function of the number of iterations.

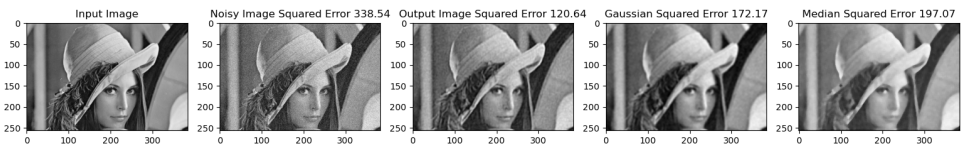


Figure 35: Lena Image optimized and comparison with denoising using Gaussian and Median Filtering using learning rate = 0.005 and early stopping at 480 iteration

Figure 36: Lena image at 480 epoch Plot: Training and Testing error as a function of the number of iterations.

2. 500 Iterations:
By running gradient descent for 500 iterations with learning rate 0.01, the Saturn output image has some noise(squared error) which is greater than that of median filter and less than of Gaussian filter which is shown by the some amount of blurriness and less distinctive nature than the output generated by Median filter.
By running gradient descent for 500 iterations with learning rate 0.01, the Lena output image has most noise(squared error) which is greater than that of Gaussian filter and that of Median filter which is shown by the highest amount of roughness in texture than the output generated by both of the Gaussian and Median filter.

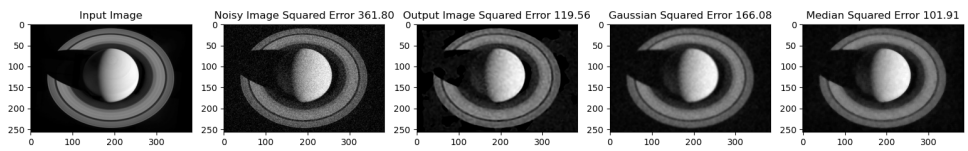


Figure 37: Saturn Image output and comparison with denoising using Gaussian and Median Filtering using learning rate = 0.01 and 500 iterations

Figure 38: Saturn image 500 epochs Plot: Training and Testing error as a function of the number of iterations.

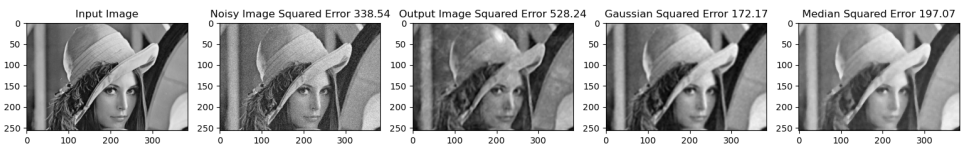


Figure 39: Lena Image output and comparison with denoising using Gaussian and Median Filtering using learning rate = 0.01 and 500 iterations

Figure 40: Lena image 500 epochs Plot: Training and Testing error as a function of the number of iterations.

3. 5000 Iterations:
By running gradient descent for 5000 iterations with learning rate 0.01, the Saturn output image has some noise(squared error) which is greater than that of median filter and less than of Gaussian filter which is shown by the some amount of blurriness and less distinctive nature than the output generated by Median filter. The Lena output image has most noise(squared error) which is greater than that of median filter and that of Gaussian filter which is shown by the highest amount of roughness in texture than the output generated by both of the Gaussian and Median filter.

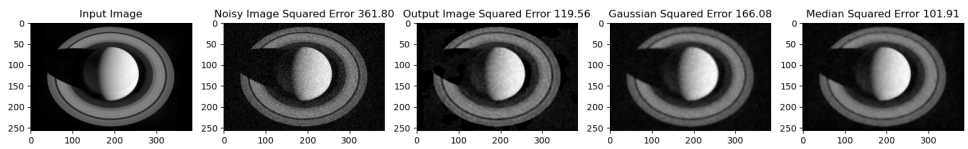


Figure 41: Saturn Image output and comparison with denoising using Gaussian and Median Filtering using learning rate = 0.01 and 5000 iterations

Figure 42: Saturn image 5000 epochs Plot: Training and Testing error as a function of the number of iterations.

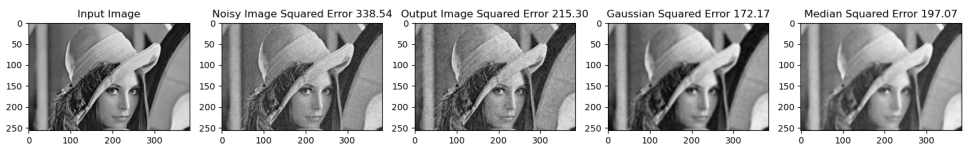


Figure 43: Lena Image output and comparison with denoising using Gaussian and Median Filtering using learning rate = 0.01 and 5000 iterations

Figure 44: Lena image 5000 epochs Plot: Training and Testing error as a function of the number of iterations.

3.3

Run gradient descent for 5000 iterations. Are the images generated after 5000 iterations better than the ones generated after 500 iterations? Why?
Please refer to the figures in the before section for results on Saturn and Lena Images on running gradient descent for 5000 iterations.
Overall images generated after 5000 iterations are not better than the ones generated after 500 iterations.
The squared error after 5000 iterations for both Saturn and Lena images is not better than 500 iterations because after a certain number of iterations, let's say T, the model, instead of minimizing the squared error and fitting the input image to the clean image, starts optimizing the input image in such a way that it fits the input image as close as possible to the target image which in this case is a noisy image. This makes the output images look similar to noisy images after 5000 iterations. As observed from the results above, a **"No denoising"** situation occurs i.e., the neural network tends to overfit the single image after running the gradient descent for a long time (5000 iterations) and makes the training error tend to go to zero. To avoid this situation, early stopping is performed, which halts at T^{th} iteration, which is the iteration where the testing squared error is minimum.

4 Extra Credit

4.1

Improving the results using ideas from Deep Image Prior CVPR paper
Two Encoder Decoder architectures are defined and the following observations are noted:
1. Pixel Shuffle Upsampling
No speckles, however there is a hotspot (in the out of focus region towards the saturn's right side in the sphere) that becomes a different shade spot. The appearance of these hotspots seems

commonplace through both architectures, but the extra smoothness given by the convolution transpose layers repairs these more effectively.
2. Convolution Transpose Upsampling:
Note the grid like speckles during training. These are caused by convolutional kernels overlapping with one another during upsampling. **Results**

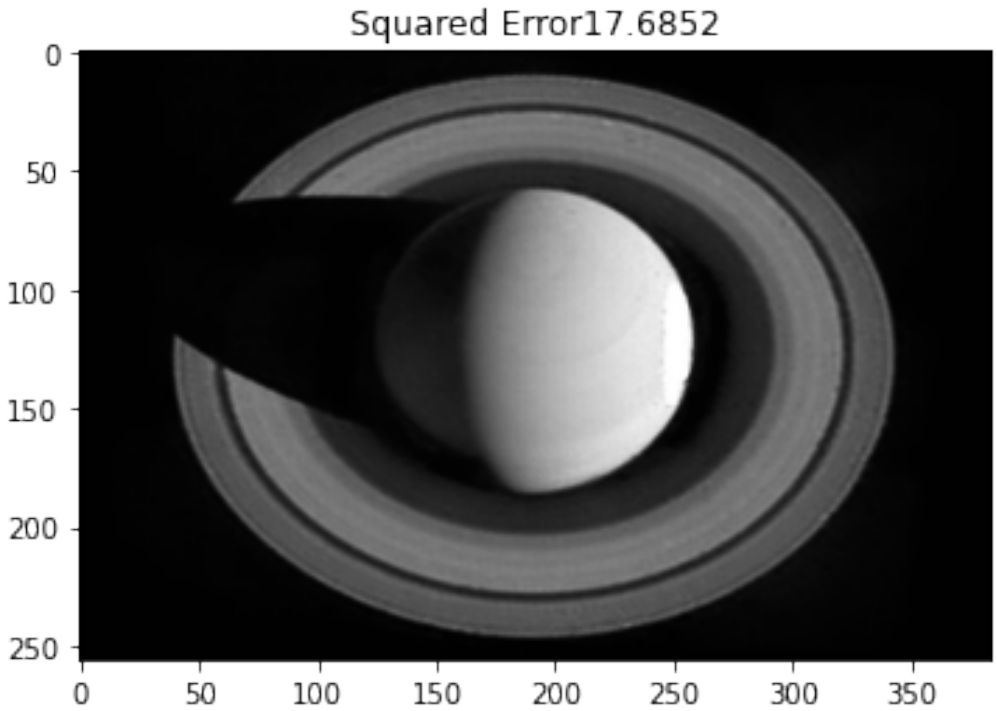


Figure 45: Pixel Shuffle Upsampling Saturn image 4001 epochs and learning rate = 1e-2

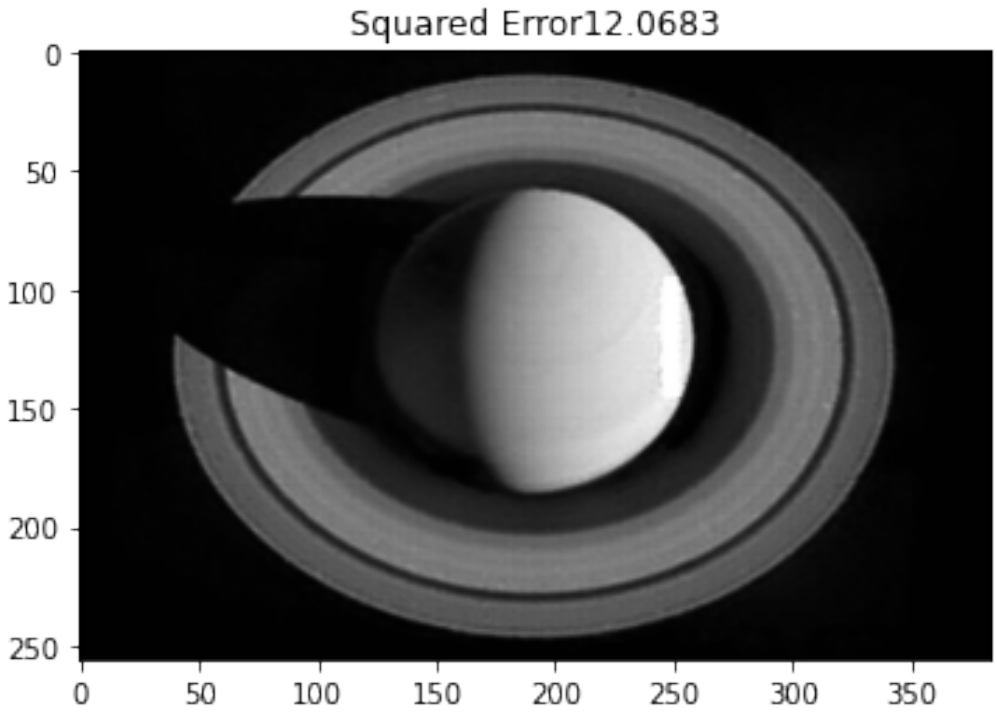


Figure 46: Convolution Transpose Upsampling Saturn image 4001 epochs and learning rate = 1e-2

Figure 47: Pixel Shuffle Upsampling Lena image 4001 epochs and learning rate = 1e-2

Figure 48: Convolution Transpose Upsampling Lena image 4001 epochs and learning rate = 1e-2

Please refer to implementation of *DeepImagePrior.py*:

```
# -*- coding: utf-8 -*-
"""DeepImagePrior.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1D96G49AIB6e1_48C7BtnBektM-5GjaXK
"""

import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.nn.functional as F
import torchvision
from PIL import Image
import matplotlib.pyplot as plt
from tqdm.notebook import tqdm

#Define an encoder decoder network with pixel shuffle upsampling
class PixelShuffleHourglass(nn.Module):
    def __init__(self):
        super(PixelShuffleHourglass, self).__init__()
        self.d_conv_1 = nn.Conv2d(3, 8, 5, stride=2, padding=2)
        self.d_bn_1 = nn.BatchNorm2d(8)

        self.d_conv_2 = nn.Conv2d(8, 16, 5, stride=2, padding=2)
        self.d_bn_2 = nn.BatchNorm2d(16)

```

```
        self.d_conv_3 = nn.Conv2d(16, 32, 5, stride=2, padding=2)
        self.d_bn_3 = nn.BatchNorm2d(32)
        self.s_conv_3 = nn.Conv2d(32, 4, 5, stride=1, padding=2)

        self.d_conv_4 = nn.Conv2d(32, 64, 5, stride=2, padding=2)
        self.d_bn_4 = nn.BatchNorm2d(64)
        self.s_conv_4 = nn.Conv2d(64, 4, 5, stride=1, padding=2)

        self.d_conv_5 = nn.Conv2d(64, 128, 5, stride=2, padding=2)
        self.d_bn_5 = nn.BatchNorm2d(128)
        self.s_conv_5 = nn.Conv2d(128, 4, 5, stride=1, padding=2)

        self.d_conv_6 = nn.Conv2d(128, 256, 5, stride=2, padding=2)
        self.d_bn_6 = nn.BatchNorm2d(256)

        self.u_conv_5 = nn.Conv2d(68, 128, 5, stride=1, padding=2)
        self.u_bn_5 = nn.BatchNorm2d(128)

        self.u_conv_4 = nn.Conv2d(36, 64, 5, stride=1, padding=2)
        self.u_bn_4 = nn.BatchNorm2d(64)

        self.u_conv_3 = nn.Conv2d(20, 32, 5, stride=1, padding=2)
        self.u_bn_3 = nn.BatchNorm2d(32)

        self.u_conv_2 = nn.Conv2d(8, 16, 5, stride=1, padding=2)
        self.u_bn_2 = nn.BatchNorm2d(16)

        self.u_conv_1 = nn.Conv2d(4, 16, 5, stride=1, padding=2)
        self.u_bn_1 = nn.BatchNorm2d(16)

        self.out_conv = nn.Conv2d(4, 3, 5, stride=1, padding=2)
        self.out_bn = nn.BatchNorm2d(3)


    def forward(self, noise):
        down_1 = self.d_conv_1(noise)
        down_1 = self.d_bn_1(down_1)
        down_1 = F.leaky_relu(down_1)

        down_2 = self.d_conv_2(down_1)
        down_2 = self.d_bn_2(down_2)
        down_2 = F.leaky_relu(down_2)

        down_3 = self.d_conv_3(down_2)
        down_3 = self.d_bn_3(down_3)
        down_3 = F.leaky_relu(down_3)
        skip_3 = self.s_conv_3(down_3)

        down_4 = self.d_conv_4(down_3)
        down_4 = self.d_bn_4(down_4)
        down_4 = F.leaky_relu(down_4)
        skip_4 = self.s_conv_4(down_4)

        down_5 = self.d_conv_5(down_4)
        down_5 = self.d_bn_5(down_5)
        down_5 = F.leaky_relu(down_5)
        skip_5 = self.s_conv_5(down_5)

```

```
        down_6 = self.d_conv_6(down_5)
        down_6 = self.d_bn_6(down_6)
        down_6 = F.leaky_relu(down_6)

        up_5 = F.pixel_shuffle(down_6, 2)
        up_5 = torch.cat([up_5, skip_5], 1)
        up_5 = self.u_conv_5(up_5)
        up_5 = self.u_bn_5(up_5)
        up_5 = F.leaky_relu(up_5)

        up_4 = F.pixel_shuffle(up_5, 2)
        up_4 = torch.cat([up_4, skip_4], 1)
        up_4 = self.u_conv_4(up_4)
        up_4 = self.u_bn_4(up_4)
        up_4 = F.leaky_relu(up_4)

        up_3 = F.pixel_shuffle(up_4, 2)
        up_3 = torch.cat([up_3, skip_3], 1)
        up_3 = self.u_conv_3(up_3)
        up_3 = self.u_bn_3(up_3)
        up_3 = F.leaky_relu(up_3)

        up_2 = F.pixel_shuffle(up_3, 2)
        up_2 = self.u_conv_2(up_2)
        up_2 = self.u_bn_2(up_2)
        up_2 = F.leaky_relu(up_2)

        up_1 = F.pixel_shuffle(up_2, 2)
        up_1 = self.u_conv_1(up_1)
        up_1 = self.u_bn_1(up_1)
        up_1 = F.leaky_relu(up_1)

        out = F.pixel_shuffle(up_1, 2)
        out = self.out_conv(out)
        out = self.out_bn(out)
        out = F.sigmoid(out)
        return out

#Define an encoder decoder network with convolution transpose upsampling.
class DeconvHourglass(nn.Module):
    def __init__(self):
        super(DeconvHourglass, self).__init__()
        self.d_conv_1 = nn.Conv2d(3, 8, 5, stride=2, padding=2)
        self.d_bn_1 = nn.BatchNorm2d(8)

        self.d_conv_2 = nn.Conv2d(8, 16, 5, stride=2, padding=2)
        self.d_bn_2 = nn.BatchNorm2d(16)

        self.d_conv_3 = nn.Conv2d(16, 32, 5, stride=2, padding=2)
        self.d_bn_3 = nn.BatchNorm2d(32)
        self.s_conv_3 = nn.Conv2d(32, 4, 5, stride=1, padding=2)

        self.d_conv_4 = nn.Conv2d(32, 64, 5, stride=2, padding=2)
        self.d_bn_4 = nn.BatchNorm2d(64)
        self.s_conv_4 = nn.Conv2d(64, 4, 5, stride=1, padding=2)

        self.d_conv_5 = nn.Conv2d(64, 128, 5, stride=2, padding=2)
        self.d_bn_5 = nn.BatchNorm2d(128)
```

```
        self.s_conv_5 = nn.Conv2d(128, 4, 5, stride=1, padding=2)

        self.d_conv_6 = nn.Conv2d(128, 256, 5, stride=2, padding=2)
        self.d_bn_6 = nn.BatchNorm2d(256)

        self.u_deconv_5 = nn.ConvTranspose2d(256, 124, 4, stride=2, padding=1)
        self.u_bn_5 = nn.BatchNorm2d(128)

        self.u_deconv_4 = nn.ConvTranspose2d(128, 60, 4, stride=2, padding=1)
        self.u_bn_4 = nn.BatchNorm2d(64)

        self.u_deconv_3 = nn.ConvTranspose2d(64, 28, 4, stride=2, padding=1)
        self.u_bn_3 = nn.BatchNorm2d(32)

        self.u_deconv_2 = nn.ConvTranspose2d(32, 16, 4, stride=2, padding=1)
        self.u_bn_2 = nn.BatchNorm2d(16)

        self.u_deconv_1 = nn.ConvTranspose2d(16, 8, 4, stride=2, padding=1)
        self.u_bn_1 = nn.BatchNorm2d(8)

        self.out_deconv = nn.ConvTranspose2d(8, 3, 4, stride=2, padding=1)
        self.out_bn = nn.BatchNorm2d(3)


    def forward(self, noise):
        down_1 = self.d_conv_1(noise)
        down_1 = self.d_bn_1(down_1)
        down_1 = F.leaky_relu(down_1)

        down_2 = self.d_conv_2(down_1)
        down_2 = self.d_bn_2(down_2)
        down_2 = F.leaky_relu(down_2)

        down_3 = self.d_conv_3(down_2)
        down_3 = self.d_bn_3(down_3)
        down_3 = F.leaky_relu(down_3)
        skip_3 = self.s_conv_3(down_3)

        down_4 = self.d_conv_4(down_3)
        down_4 = self.d_bn_4(down_4)
        down_4 = F.leaky_relu(down_4)
        skip_4 = self.s_conv_4(down_4)

        down_5 = self.d_conv_5(down_4)
        down_5 = self.d_bn_5(down_5)
        down_5 = F.leaky_relu(down_5)
        skip_5 = self.s_conv_5(down_5)

        down_6 = self.d_conv_6(down_5)
        down_6 = self.d_bn_6(down_6)
        down_6 = F.leaky_relu(down_6)

        up_5 = self.u_deconv_5(down_6)
        up_5 = torch.cat([up_5, skip_5], 1)
        up_5 = self.u_bn_5(up_5)
        up_5 = F.leaky_relu(up_5)

        up_4 = self.u_deconv_4(up_5)
```

```
        up_4 = torch.cat([up_4, skip_4], 1)
        up_4 = self.u_bn_4(up_4)
        up_4 = F.leaky_relu(up_4)

        up_3 = self.u_deconv_3(up_4)
        up_3 = torch.cat([up_3, skip_3], 1)
        up_3 = self.u_bn_3(up_3)
        up_3 = F.leaky_relu(up_3)

        up_2 = self.u_deconv_2(up_3)
        up_2 = self.u_bn_2(up_2)
        up_2 = F.leaky_relu(up_2)

        up_1 = self.u_deconv_1(up_2)
        up_1 = self.u_bn_1(up_1)
        up_1 = F.leaky_relu(up_1)

        out = self.out_deconv(up_1)
        out = self.out_bn(out)
        out = F.sigmoid(out)

        return out

#use cuda, or not? be prepared for a long wait if you don't have cuda capabilities.
use_cuda = True

#input image.
groundTruth = 'lena.png'

#proportion of pixels to black out.
prop = 0.5

#standard deviation of added noise after each training set
sigma = 1./30

#number of training iterations
num_steps = 4001

#number of steps to take before saving an output image
save_frequency = 250

#where to put the output
output_name = ''

#choose either 'pixel_shuffle' or 'deconv' as the architecture used.
method = 'deconv'

#accept a file path to a png, return a torch tensor
def pngToTensor(filepath=groundTruth):
    pil = Image.open(groundTruth)
    pil_to_tensor = torchvision.transforms.Compose([torchvision.transforms.ToTensor()])
    if use_cuda:
        tensor = pil_to_tensor(pil).cuda()
    else:
        tensor = pil_to_tensor(pil)
    return tensor.view([1]+list(tensor.shape))
```

```
#accept a torch tensor, convert it to a jpg at a certain path
def tensorToPng(tensor, filename, error):
    tensor = tensor.view(tensor.shape[1:])
    if use_cuda:
        tensor = tensor.cpu()
    tensor_to_pil = torchvision.transforms.Compose([torchvision.transforms.
    ToPILImage()])
    pil = tensor_to_pil(tensor)
    if error == 0:
        plt.imshow(pil, cmap='gray')
        plt.show()
    else:
        plt.imshow(pil, cmap='gray')
        plt.title('Squared Error{:.4f}'.format(error))
        plt.show()
    plt.savefig(filename)

#function which zeros out a random proportion of pixels from an image tensor.
def zeroOutPixels(tensor, prop=prop):
    if use_cuda:
        mask = torch.rand([1]+[1] + list(tensor.shape[2:])).cuda()
    else:
        mask = torch.rand([1]+[1] + list(tensor.shape[2:]))
    mask[mask<prop] = 0
    mask[mask!=0] = 1
    mask = mask.repeat(1,3,1,1)
    deconstructed = tensor * mask
    return mask, deconstructed

if __name__=='__main__':
    #import image
    truth = pngToTensor(groundTruth)
    #deconstruct image
    mask, deconstructed = zeroOutPixels(truth)
    #save the deconstructed image
    tensorToPng(deconstructed, 'deconstructed.png', 0)
    #convert the image and mask to variables.
    mask = Variable(mask)
    deconstructed = Variable(deconstructed)

    #input of the network is noise
    if use_cuda:
        noise = Variable(torch.randn(deconstructed.shape).cuda())
    else:
        noise = Variable(torch.randn(deconstructed.shape))

    #initialise the network with the chosen architecture
    if method=='pixel_shuffle':
        net = PixelShuffleHourglass()
    elif method=='deconv':
        net = DeconvHourglass()

    #bind the network to the gpu if cuda is enabled
    if use_cuda:
        net.cuda()
    #network optimizer set up
    optimizer = torch.optim.Adam(net.parameters(), lr=1e-2)
```

```
    #dummy index to provide names to output files
    saveImgIndex = 0
    for step in range(num_steps):
        #get the network output
        output = net(noise)
        #we are only concerned with the output where we have the image available
    .
        maskedOutput = output * mask
        # calculate the l2_loss over the masked output and take an optimizer
    step
        optimizer.zero_grad()
        loss = torch.sum((maskedOutput - deconstructed)**2)
        loss.backward()
        optimizer.step()
        print('At step {}, loss is {}'.format(step, loss.data.cpu()))
        #every save_frequency steps, save a jpg
        if step % save_frequency == 0:
            tensorToPng(output.data,output_name+'_{}.png'.format(saveImgIndex),
    loss)
            saveImgIndex += 1
        if use_cuda:
            noise.data += sigma * torch.randn(noise.shape).cuda()
        else:
            noise.data += sigma * torch.randn(noise.shape)

    #clean up any mess we're leaving on the gpu
    if use_cuda:
        torch.cuda.empty_cache()
```

Please find attached Colab Notebook Implementation below.