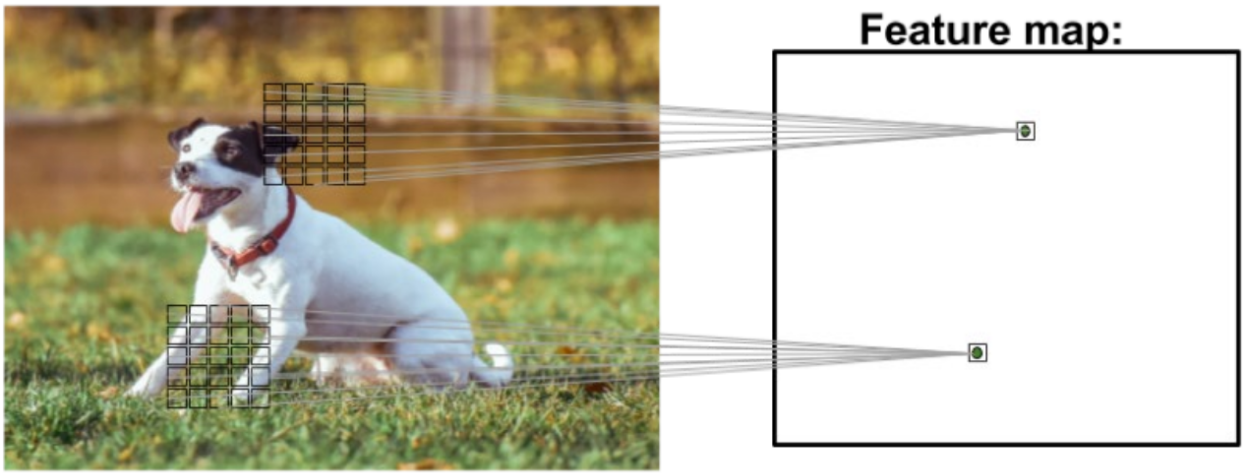
CERI - 8104 – HW #3

ML Seminar – Convolutional Neutral Networks

Reading: "Python Machine Learning", Raschka, Chapter 15
Submit a short report with figures that illustrate your results! Explain your observations and respond to all questions below!

From Raschka, Chapter 15:

As you can see in the following image, a CNN computes feature maps from an input image, where each element comes from a local patch of pixels in the input image.



This local patch of pixels is referred to as the local receptive field. CNNs will usually perform very well for image-related tasks, and that's largely due to two important ideas:

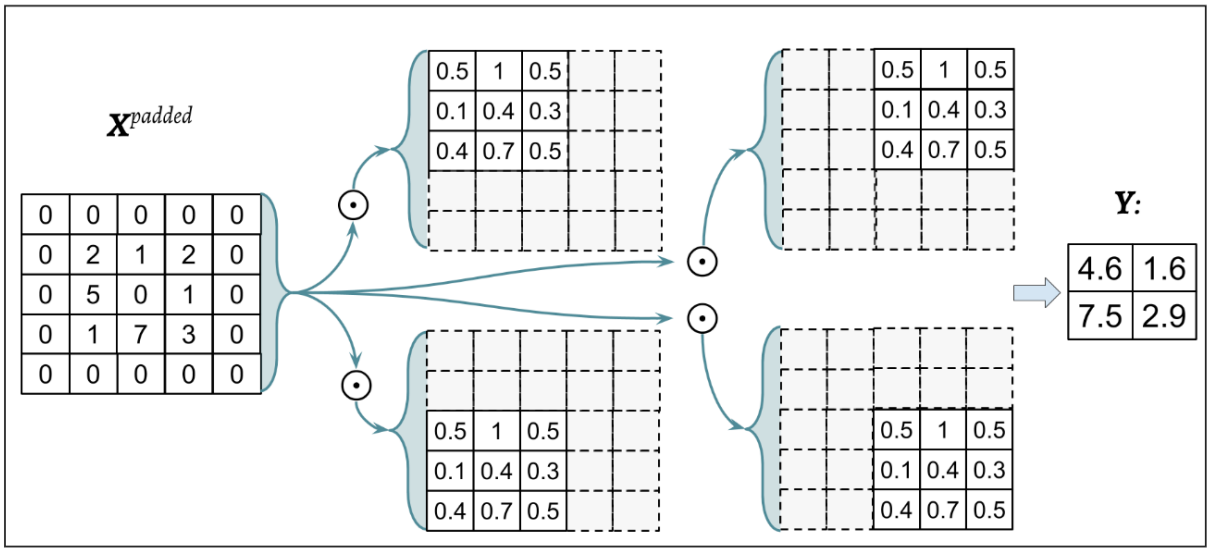
1. Sparse-connectivity: A single element in the feature map is connected to only a small patch of pixels. This is very different from connecting to the whole input image, in the case of perceptrons.
2. Parameter-sharing: The same weights are used for different patches of the input image.

As a direct consequence of these two ideas, the number of weights (parameters) in the network decreases dramatically, and we see an improvement in the ability to capture salient features. Intuitively, it makes sense that nearby pixels are probably more relevant to each other than pixels that are far away from each other. Typically, CNNs are composed of several Convolutional (conv) layers and subsampling (also known as Pooling (P)) layers that are followed by one or more Fully Connected (FC) layers at the end. The fully connected layers are essentially a multilayer perceptron, where every input unit i is connected to every output unit j with weight w_{ij}.

A discrete convolution is mathematically defined as follows:

$$y = x * w \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i-k]w[k]$$

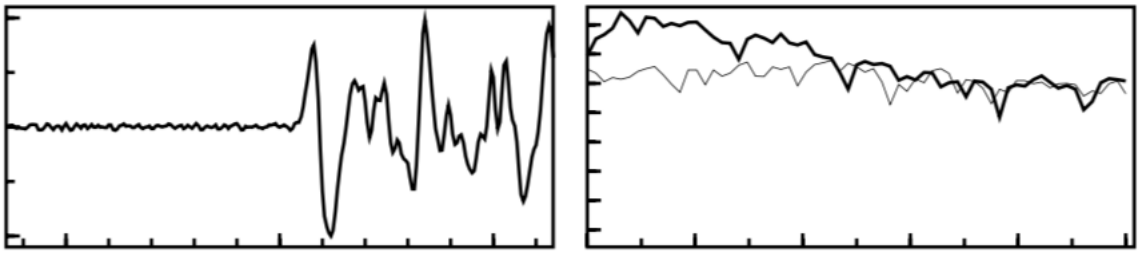
An example of convolution for 2D tensors is give in the following image where two 3x3 matrices are convolved with padding, p=1 and stride, s=2:



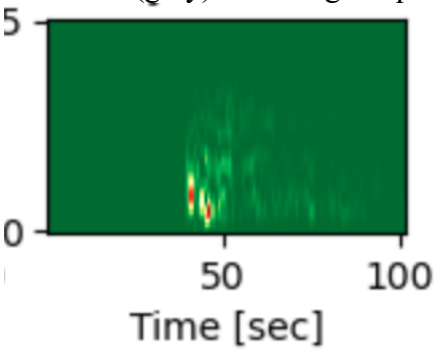
You can think of as X^{padded} as the 'signal' and the matrix: [0.5 1 0.5; 0.1 0.4 0.3; 0.4 0.7 0.5] as a digital filter.

For this homework, you will try to classify a large data set of seismic waveforms. For this purpose, you will convert the waveform time series into the frequency domain by computing the spectrogram for each waveform. There are convenient functions in *scipy* or *obspy* to do this. You can then tread the classification of waveforms in the same way you would approach any images classification problem.

Example waveform with low SNR.



Example waveform centered on P-pick (left) and amplitude difference between signal (black) and noise (gray) in the log-frequency domain (right)



Example event spectrogram at a frequency of 0.1 to 5 Hz.

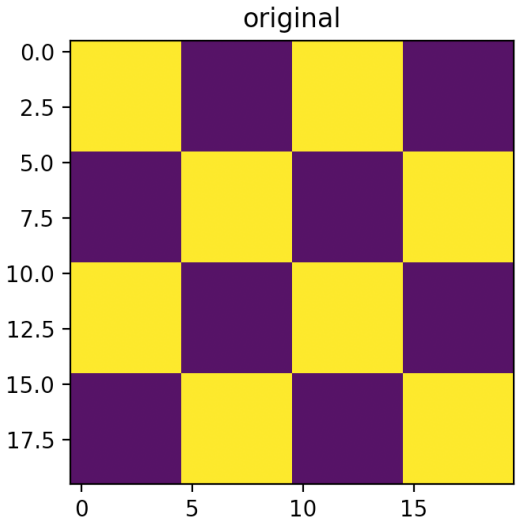
You can follow the standard workflow for machine learning problems that we established in previous activities. However, there will be a few additional tasks for you in this homework:

1. You will have to prepare the dataset through a series of processing steps to convert time series data to something resamples on image or a 2D matrix.
2. You will perform more in-depth validation and performance tests to verify how well your network has learned characteristic patterns of seismic events in the frequency domain.
3. You will compare the performance of a fully-connected neural net with the performance of a convolutional neutral net. Comparing these two architectures will be insightful both in terms of differences in performance as well as differences in computational efficiency.

Part I: Building intuition through exploring edge detection kernels

This first part will focus on building some basic intuition of how convolutional neural networks work and why they are effective in recognizing 2 or more-D patterns e.g. in images. For this simple exercise, you will convolve a 'signal' matrix with different 3x3 edge detection filters to high light a) vertical, b) horizontal, and c) all edges.

1. Start by creating a simple, binary matrix will alternating bright and dark patches analogous to the image below:



There are many ways to do this but you could simply use numpy's concatenate function:

```
np.concatenate((m_1, m_2), axis=0)
np.concatenate((m_1, m_2), axis=1)
```

It should also be useful to define the following variables at the beginning of your program so you can easily change the dimensions of your checkerboard matrix:

```
patch_dim = 5, (patch size in pixel)
n_patch   = 4,  (number of patches in x and y)
i_bright  = 10, (integer value for bright squares)
i_dark    = 0, (integer value for dark squares)
```

To visualize the matrix you can use: `plt.pcolormesh, or plt.imshow or plt.pcolor`

2. Now let's continue with the edge detection task by convolving the matrix with different edge detection filters. Use the following three 3x3 kernels to isolate horizontal, vertical and all edges in your matrix:

```
hor_kernel  = np.array([[3,10,3],
                        [0,0,0],
                        [-3,-10,-3]])
ver_kernel  = np.array([[1,0,-1],
                        [1,0,-1],
                        [1, 0,-1]])
all_kernel = np.array([[-1, -1, -1],
                       [-1, 8, -1],
                       [-1, -1, -1]])
```

   Describe your observations!

3. The results from the initial analysis where probably not very satisfying. Why? Try increasing the parameter, `patch_dim`, by a factor of 10 to `patch_dim = 50`. Describe your observations!

This basic example was hopefully somewhat insightful in demonstrating how convolution operations can effectively highlight and isolate the basic characteristics of an image. Let's now move on to the actual image classification tasks with MLPs and CNNs.

Part II: Prepare the seismic data

In this part, you will convert the seismic waveforms into 2D matrices and save all instances and features in one giant matrix that also contains all class labels. It is certainly good practice to break-down complicated tasks into smaller sub-tasks which can more easily be tested and benchmarked. Preparing the spectrograms and saving them in a separate file is one of such tasks. You will probably also want to create separate programs to solve the classification problem with MLPs and CNNs.

1. Download the zipped data file and inspect the different folders in the unpacked file. Confirm that there are three different types of seismic events called 'AE' (labquakes), 'UP' (active source records) and noise. You probably want to look at the data using some basic python command line functions such as:

```
>>>from obspy.core import read
>>>SEIS = read( file)
>>>SEIS.plot()
```

2. **Computing spectrograms:**

Take a look at this url to better understand how spectrograms are computed and what the required function arguments are:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.spectrogram.html

   a. Create a large matrix using `mSpec = np.zeros()` with `shape= (nx*ny+1, len([all_files]))`. nx and ny will be the shape of your spectrograms which are a function of overlap, nperseg as well as of the nfft parameter. You could probably use something like:

```
'nperseg' : 20, 'noverlap' : 10,
'nfft'    : 128,
'nx'      : 101,
'ny'      :  65, # = nfft*.5 (because of Nyquist)
```

You could store all of these parameters in a python dictionary which could easily be saved in a separate file.

Be careful when setting the *nfft* parameter to compute the Fourier transform within `scipy.signal.spectrogram`. Nfft should be a factor of two (e.g. 1024, 2048 etc.). This parameter will control the resolution and number of pixels, ny. I recommend starting with nfft = 128 so you can still do the classification with MLPs. If the matrices get too large it will no longer be feasible to use fully-connected NNs.

b. Now write a *for -loop* over all three directories and all `.mseed` files within these directories to read and process the waveforms. You can use: `l_files = glob.glob( '*.mseed')` to create a list of files that follow the same pattern, here all `.mseed` files.
c. Read each file with obspy, trim all waveforms to the same length, and detrend the waveforms.
d. Compute the spectrograms using: `scipy.signal.spectrogram`
e. Inspect your results by plotting a number of exemplary spectrograms. What are the differences between the three event types? Are the difference more apparent on linear or log- scales? Depending on your observations here, you may want to save the log-transformed spectrograms.

3. Save the spectrograms as .mat binary.
    a. Flatten the matrix output from `scipy.signal.spectrogram`.
    b. Add the class label to each flattened vector. You will have to encode the string class labels using a system of integer numbers.
    c. Store the vectors in the previously created mSpec matrix.
    d. Save your final matrix with: `scipy.io.savemat( [file_out', {'spec': mSpec}, do_compression=True)`

Part III: Construct and train the neural net

We will start with a simple MLP for the classification task and then compare results with a CNN.

1. Import the following scikit modules:

```
import scipy.io
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier as MLP
from sklearn.metrics import accuracy_score
# learning curve
from sklearn.model_selection import learning_curve
```

2. Load and prepare the data:

```
dSpec = scipy.io.loadmat(  [file_in], squeeze_me = True, struct_as_record = False)
print( dSpec.keys())
X    = dSpec['mSpec'][:,0:-1]
y    = dSpec['mSpec'][:,-1]
Ntot  = X.shape[0]
print( 'total data set: ', Ntot, len( y), len( X[:,0]))
# -----------------train, test split----------------------
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = test_size, shuffle   =True)
```

3. Successfully classifying the data set will require tuning a large number of parameters. Here are some initial suggestions, which we will revisit and modify once we reach the validation and testing step:

```
test_size = 0.2
solver   = 'sgd' #
n_iter   = 20 # max. iterations
d_hidden = (20)
#L2_reg   = 0.01 #factor for regularization term
L2_reg   = .001 #factor for regularization term
n_minibatch = 50
eta        = 0.001 # learning rate
```

4. Create an MLP estimator object using the above parameters. Fit all weights and biases using the training data and report your initial result using:
   `estimator.score( X_test, y_test)`.

Part IV: Performance evaluation and hyper-parameter tuning

Although, the first performance metric may look encouraging, we cannot place too much confidence on a single value. To really understand whether our NN was trained successful, we need to do a suite of blind tests (cross-validation) and look at the functional form of the loss curve (misfit as a function of epoch, i.e. cycle of the training data). These more elaborate tests will help us to see whether the NN learned general data features or solely memorized the training data. It will also help to prevent over (high variance) and underfitting (high bias).

5. Plot the loss function over 100 to 200 epochs. You may have to force the program to run to the maximum number of iterations.

   ```
   n_iter_no_change  = n_iter
   early_stopping    = False
   ```

   The loss curve can be plotted with:

   `estimator.loss_curve_`
6. Now also add learning curves that show the change in accuracy of the predictions using training and validation datasets:
   It is good practice to use k-fold cross-validation to see how variable the predictions are for different data input batches. If you see a high-standard deviation for results from different randomly sampled data batches, you should go back and improve your training behavior by tuning hyperparameters or changing your network architecture.
   There is a very convenient function for learning curves and cross-validation tests in scikit learn that will take care of creating random k-fold sub-samples and computing test scores as a function of training data size:

```
train_size, train_score, test_score = learning_curve( estimator, X, y, cv = cv, shuffle= False,
                                        train_sizes=a_train_size, n_jobs=-1,)
```

```
train_score_mean = train_score.mean( axis =1)
train_score_std  = train_score.std( axis =1)
test_score_mean  = test_score.mean( axis =1)
test_score_std   = test_score.std( axis =1)
print( train_size, train_score_mean, test_score_mean)
```

https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html

7. Lastly, compute the accuracy for the testing data and compare training and testing scores. Create a few plots of false predictions to evaluate when and why your neural net is failing.

8. The initial performance of your neural net is probably not very satisfying. Two probable causes are learning rate and NN architecture. Try changing the optimization to a variable-size learning rate solver:
   `solver = 'adam'` (instead of 'sgd').
   Increase the size of your neural net:
   `d_hidden = (50,2)`
   For this and the following tasks, you may want to allow for early stoppage during training if the loss-function converges to a constant value.
   Determine the number of weights and biases for each hidden layer for your two different MLPs with `d_hidden = (20)`, and `d_hidden = (50,2)`.

9. Conceivably, there may be another issue with the input data that we have not yet considered. Peak amplitudes generally scale as ~10^Mag so we expect a large variability of absolute values in the input data features. It may be a good idea to again test if your training performance and general behavior can be improved by log-transforming the input data and/or by standardizing each input data sample.