COMP 4331 Tut 4

Jiaxin Xie

Outline

- Brief introduction to assignment1
- Brief introduction to MLP
- Brief introduction to CNN

Brief introduction to assignment1

Q1:

- 1. boxplot (tutorial 1)
- 2. z-score normalization (find mean and std of every feature, implement it from scratch)
- 3. numerical measure and plot
- 4. decision tree
- 5. cross-validation

Brief introduction to assignment1

Q2

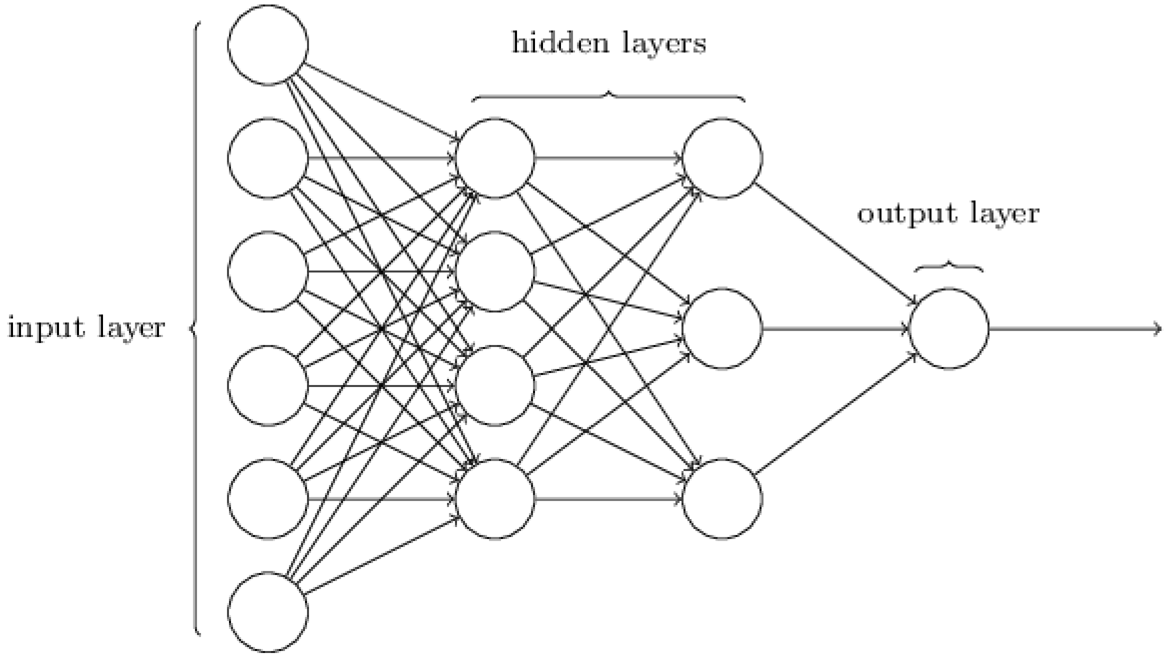
- Naïve Bayes
- Laplacian correction

Q3

- We give normalized data, you can use it directly.
- 1 nearest neighbor, implement it from scratch.
- How to calculate test accuracy: find test samples' nearest neighbor in training set.
- After PCA, we get a transform matrix. For example, reducing 784 dims feature to 200 dims. We get a transform matrix whose size is 784×200. Our requirement is when you get this transform matrix from training set, you don't need calculate on test set again. Directly matrix multiply test data and this transform matrix.

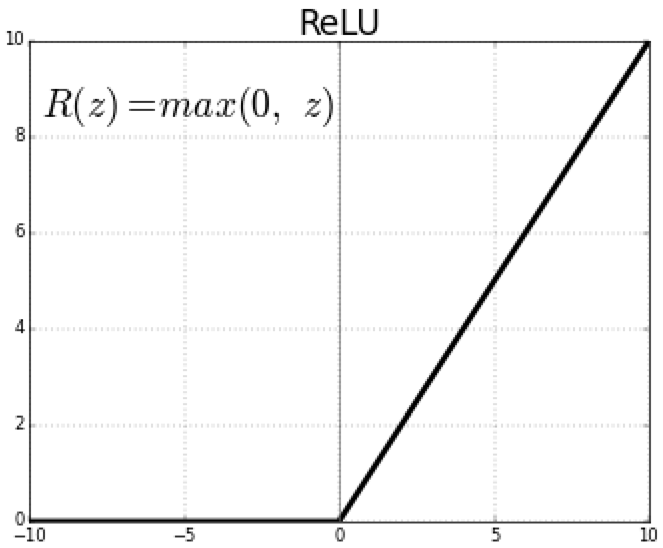
MLP

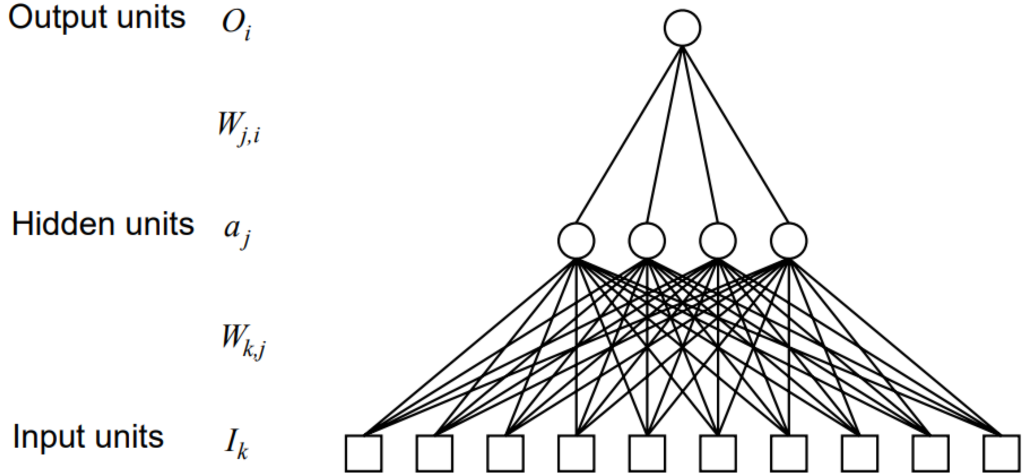
A multilayer perceptron (MLP) is a fully connected neural network. All the nodes from the current layer are connected to the next layer. A MLP consisting in 3 or more layers: an input layer, an output layer and one or more hidden layers. Note that the activation function for the nodes in all the layers (except the input layer) is a non-linear function.



MLP

Example : MNIST Data which has 10 classes
I: 10000 ×784 10000 samples and 784 dims features
W1: 784×100
a: ReLU (I×W1)   10000×100
W2: 100*10
O: ReLU (a×W2) 10000*10 – predicted one hot labels





MLP

- initialize weights *W1,W2* to some small random values

```
# Randomly initialize weights
w1 = initalize_weights_relu(784, 100)
w2 = initalize_weights_relu(100, 10)
```

- propagate the input forward through the network

```
a1 = X
z2 = a1.dot(w1)
a2 = np.maximum(z2, 0)
z3 = a2.dot(w2)
a3 = np.maximum(z3, 0)
Y_hat = a3
```

MLP

- propagate the errors backward through the network

  Gradient descent

```
d3 = Y_hat - Y
grad2 = a2.T.dot(d3) / n_examples
d2_tmp = d3.dot(w2.T)
d2 = d2_tmp.copy()
d2[z2 <= 0] = 0 #d2 = d2 * derivate of ReLU function
grad1 = a1.T.dot(d2) / n_examples
```

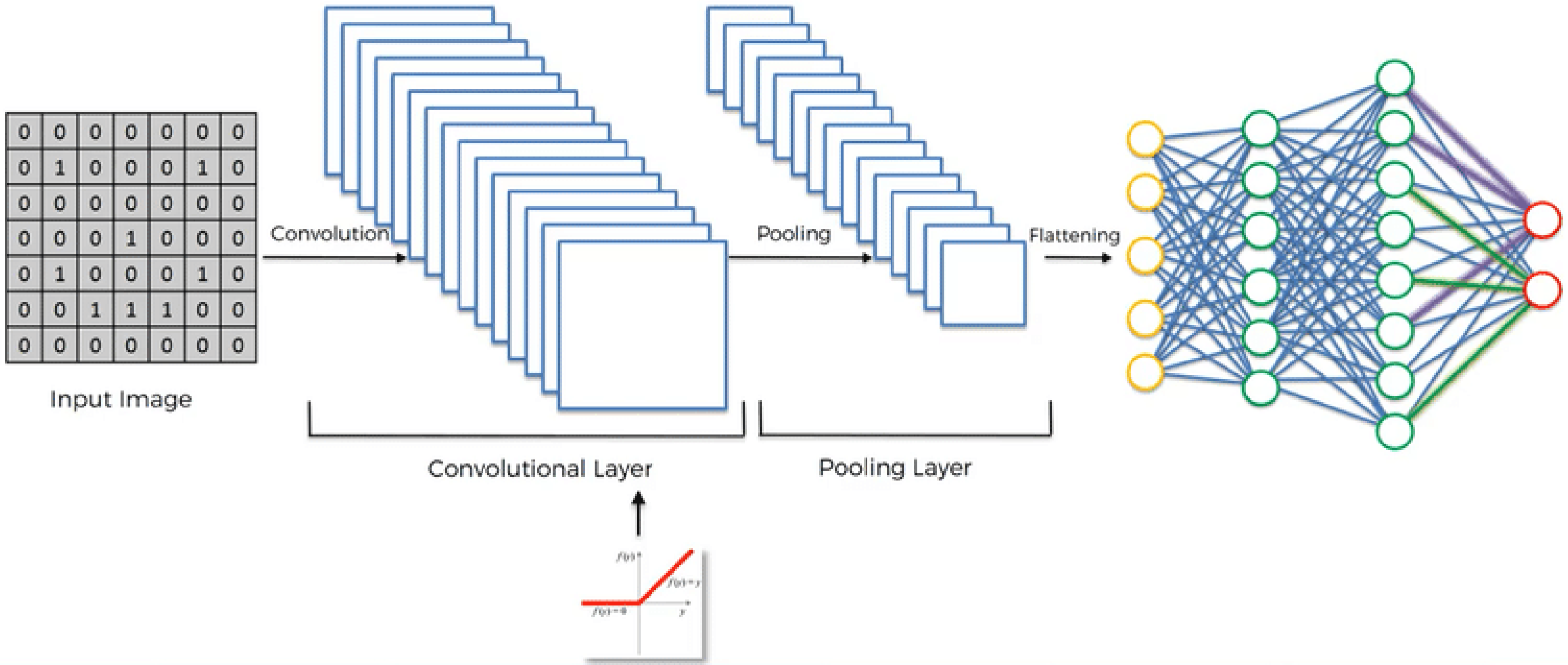
- for each network weight *W* *W<--W+δW*

```
# Update weights
w1 = w1 - grad1
w2 = w2 - grad2
```

CNN

CNNs are regularized versions of multilayer perceptrons. Multilayer perceptrons usually mean fully connected networks, that is, each neuron in one layer is connected to all neurons in the next layer. The "fully-connectedness" of these networks makes them prone to overfitting data.

CNNs take a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns.



A simple CNN

```python
def predict(image, f1, f2, w3, w4, b1, b2, b3, b4, conv_s = 1, pool_f = 2, pool_s = 2):
    '''
    Make predictions with trained filters/weights.
    '''
    conv1 = convolution(image, f1, b1, conv_s) # convolution operation
    conv1[conv1<=0] = 0 #relu activation

    conv2 = convolution(conv1, f2, b2, conv_s) # second convolution operation
    conv2[conv2<=0] = 0 # pass through ReLU non-linearity
    print('conv2:',conv2.shape)
    pooled = maxpool(conv2, pool_f, pool_s) # maxpooling operation
    (nf2, dim2, _) = pooled.shape
    fc = pooled.reshape((nf2 * dim2 * dim2, 1)) # flatten pooled layer

    z = w3.dot(fc) + b3 # first dense layer
    z[z<=0] = 0 # pass through ReLU non-linearity

    out = w4.dot(z) + b4 # second dense layer
    probs = softmax(out) # predict class probabilities with the softmax activation function

    return np.argmax(probs), np.max(probs)
```

CNN

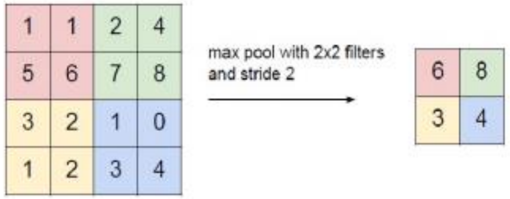
Convolutional Layer

```python
def convolution(image, filt, bias, s=1):
    '''
    Confolves `filt` over `image` using stride `s`
    '''
    (n_f, n_c_f, f, _) = filt.shape # filter dimensions
    n_c, in_dim, _ = image.shape # image dimensions

    out_dim = int((in_dim - f)/s)+1 # calculate output dimensions

    assert n_c == n_c_f, "Dimensions of filter must match dimensions of input image"

    out = np.zeros((n_f,out_dim,out_dim))

    # convolve the filter over every part of the image, adding the bias at each step.
    for curr_f in range(n_f):
        curr_y = out_y = 0
        while curr_y + f <= in_dim:
            curr_x = out_x = 0
            while curr_x + f <= in_dim:
                out[curr_f, out_y, out_x] = np.sum(filt[curr_f] * image[:,curr_y:curr_y+f, curr_x:curr_x+f]) + bias[curr_f]
                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1

    return out
```

CNN

Max-pooling

- for each such sub-region (e.g., over a 2×2 area in the previous layer), outputs the maximum value




```python
def maxpool(image, f=2, s=2):
    '''
    Downsample `image` using kernel size `f` and stride `s`
    '''
    n_c, h_prev, w_prev = image.shape

    h = int((h_prev - f)/s)+1
    w = int((w_prev - f)/s)+1

    downsampled = np.zeros((n_c, h, w))
    for i in range(n_c):
        # slide maxpool window over each part of the image and assign the max value at each step to the output
        curr_y = out_y = 0
        while curr_y + f <= h_prev:
            curr_x = out_x = 0
            while curr_x + f <= w_prev:
                downsampled[i, out_y, out_x] = np.max(image[i, curr_y:curr_y+f, curr_x:curr_x+f])
                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1
    return downsampled
```

Other tools

- PyTorch
- Tensorflow
- MXNet

These tools&Libraries can help you fast implement MLP and CNN. Their function cover most of operation you need.