## Lecture: Intro to CNN

CNNs are currently the best algorithms we have for processing images.

They are used **to identify objects in images**.

Keras is python-based library for building CNNs (a type of ML algorithm)

To display an image use:

Import matplotlib-pyplot as plt

Data = plt.imread(‘stop\_sign.jpg’)

Plt.imshow(data)

Plt.show()

The computer sees an array of numbers of the image. Color images are stored in 3-D. First two dimensions are the height and width of the image in pixels, the last dimension are the number of channels (R,G,B) present in each pixel.

Data[1000,1500] then you get rgb of the pixel

We set the green and blue pixels to 0 in the image:

Data[:,:,1] = 0

Data[:,:,2] = 0

Plt.imshow(data)

Plot.show()

The result is an image that only contains the information in the red channel.

If we set red and blue to zero, the resulting pixel area in the image will be green such as a green square in the image.

Data[200:1200,200:1200,:] = [0,1,0]

Plt.imshow(data)

Plt.show()

We can change part of the array using the assignment.

## Classifying images

Three different classes of images: t-shirt, dress and shoe

Training from samples with images from these three classes together with the class label, the algorithm learns to classify, how well did the classifier do? T-shirt classified as a shoe? Test vs. prediction

How do we represent data for classification? OHE

Labels = [[0,0,1],[0,1,0]]

# The number of image categories

n\_categories = 3

# The unique values of categories in the data

categories = np.array(["shirt", "dress", "shoe"])

# Initialize ohe\_labels as all zeros

ohe\_labels = np.zeros((len(labels), n\_categories))

# Loop over the labels

for ii in range(len(labels)):

# Find the location of this label in the categories variable

jj = np.where(categories == labels[ii])

# Set the corresponding zero to one

ohe\_labels[ii, jj] = 1

# Calculate the number of correct predictions

number\_correct = np.sum(np.multiply(test\_labels, predictions))

print(number\_correct)

# Calculate the proportion of correct predictions

proportion\_correct = number\_correct/len(predictions)

print(proportion\_correct)

## Keras for image classification

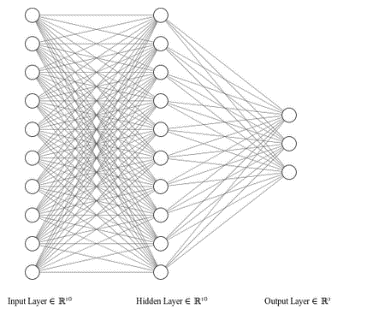
From keras.layer import Dense

Train\_data.shape

(50,28,28,1) – 50 examples of T-shirt images of 28x28 pixels of 1 channel

Model.add(Dense(10,activation=’relu’, input\_shape=(784,)))

28x28 = 784 for the input

Model.add(Dense(10, activation=”relu”))

Model.add(Dense(3, activation=”softmax”))

Output 3 different classes

Model.compile(optimizer=”adam”,

Loss=”categorical\_crossentropy”,

Metrics=[“accuracy”])

Train\_data = train\_data.reshape((50,784))

Model.fit(train\_data, train\_labels, validation\_split=0.2, epochs=3)

At the end of every epoch, we test the model on the validation set of 20%.

Test\_data=test\_data.reshape((10, 784))

Model.evaluate(test\_data, test\_labels)

Do another evaluation on a separate test set

# Imports components from Keras

from keras.models import Sequential

from keras.layers import Dense

# Initializes a sequential model

model = Sequential()

# First layer

model.add(Dense(10, activation='relu', input\_shape=(784,)))

# Second layer

model.add(Dense(10, activation='relu'))

# Output layer

model.add(Dense(3, activation='softmax'))

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Reshape the data to two-dimensional array

train\_data = train\_data.reshape(50,784)

# Fit the model

model.fit(train\_data, train\_labels, validation\_split=0.2, epochs=3)

# Reshape test data

test\_data = test\_data.reshape(10, 784)

# Evaluate the model

model.evaluate(test\_data, test\_labels)

Most pixels in images are not independent from their neighbors.

What is a convolution?

Array = np.array([0,0,0,0,0,1,1,1,1,1]) with an edge in the middle where values go from 0 to 1

Kernel = np.array([-1,1]) The kernel defines the feature, change from small values on left to large values on the right

Conv = np.array([0,0,0,0,0,0,0,0,0,0]) starting point

We slide the kernel along the array

Conv[0] = (kernel \* array[0:2]).sum()

Conv[1] = (kernel \* array[1:3]).sum()

…

We multiply the array with the convolution kernel

For ii in range(8):

Conv[ii] = (kernel \* array[ii:ii+2]).sum()

Conv

2-D convolution

Kernel = np.array([[-1,1],[-1,1]])

Conv = np.zeros(27,27)

For ii in range(27):

For jj in range(27):

Window= image[ii:ii+2, jj:jj+2]

Conv[ii, jj] = np.sum(window \* kernel)

array = np.array([1, 0, 1, 0, 1, 0, 1, 0, 1, 0])

kernel = np.array([1, -1, 0])

conv = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

# Output array

for ii in range(8):

conv[ii] = (kernel \* array[ii:ii+3]).sum()

# Print conv

print(conv)

kernel = np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]])

result = np.zeros(im.shape)

# Output array

for ii in range(im.shape[0] - 3):

for jj in range(im.shape[1] - 3):

result[ii, jj] = (im[ii:ii+3, jj:jj+3] \* kernel).sum()

# Print result

print(result)

from keras.models import Sequential

from keras.layers import Dense, Conv2D, Flatten

model = Sequential()

model.add(Conv2D(10, kernel\_size=3, activation=”relu”, input\_shape=(img\_rows, img\_cols, 1)))

model.add(Flatten())

model.add(Dense(3, activation=”softmax”) )

# Import the necessary components from Keras

from keras.models import Sequential

from keras.layers import Dense, Conv2D, Flatten

# Initialize the model object

model = Sequential()

# Add a convolutional layer

model.add(Conv2D(10, kernel\_size=3, activation='relu',

input\_shape=(img\_rows ,img\_cols, 1)))

# Flatten the output of the convolutional layer

model.add(Flatten())

# Add an output layer for the 3 categories

model.add(Dense(3, activation='softmax'))

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Fit the model on a training set

model.fit(train\_data, train\_labels,

validation\_split=0.2,

epochs=3, batch\_size=10)

# Evaluate the model on separate test data

model.evaluate(test\_data, test\_labels, 10)

### Padding

Model.add(Conv2D(10, kernel\_size=3, activation=”relu”, input\_shape=(img\_rows, img\_cols, 1)), padding=”valid”))

No zero padding is added with padding set to valid which is also the default

Model.add(Conv2D(10, kernel\_size=3, activation=”relu”, input\_shape=(img\_rows, img\_cols, 1)), padding=”valid”))

0-Padding is included such that the input has the same size as the output

### Strides

Strided by 2 pixels which means the output is smaller than the input size

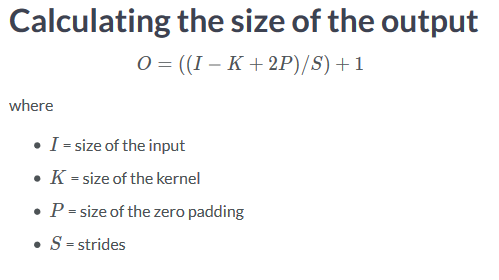
The default is 1

Model.add(Conv2D(10, kernel\_size=3, activation=”relu”, input\_shape=(img\_rows, img\_cols, 1)), strides=1))

The kernel slides along the image and is multiplied at each pixel location.

Model.add(Conv2D(10, kernel\_size=3, activation=”relu”, input\_shape=(img\_rows, img\_cols, 1)), strides=2))

The kernel skips 1 pixel when sliding



Input 28, kernel 3, padding is 1

So 28=28-3+2 /1 +1

### Dilation

Tweak the spacing between the pixels affected by the kernel

Useful if you need to aggregate information across multiple scales

Model.add(Conv2D(10, kernel\_size=3, activation=”relu”, input\_shape=(img\_rows, img\_cols, 1)), dilation\_rate=2))

model = Sequential()

# Add the convolutional layer

model.add(Conv2D(10, kernel\_size=3, activation='relu',

input\_shape=(img\_rows, img\_cols, 1),

padding="same"))

# Feed into output layer

model.add(Flatten())

model.add(Dense(3, activation='softmax'))

# Initialize the model

model = Sequential()

# Add the convolutional layer

model.add(Conv2D(10, kernel\_size=3, activation='relu',

input\_shape=(img\_rows, img\_cols, 1),

strides=2))

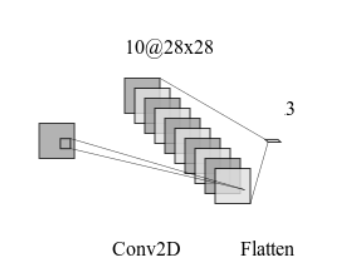
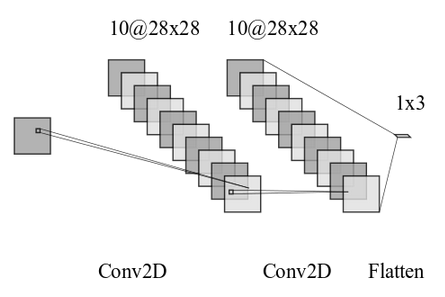
# Feed into output layer

model.add(Flatten())

model.add(Dense(3, activation='softmax'))

### Strenghts

* Building multiple layers of CNN
* 22 layers of convolutions, and other layers such as pooling
* At the top of the fully-connected network
* Gradually build up representation of objects in the image from simple to complex features and has the sensitivity to distinguish the objects
* Deeper networks require more data



# Initialize the model

model = Sequential()

# Add the convolutional layer

model.add(Conv2D(10, kernel\_size=2, activation='relu',

input\_shape=(img\_rows, img\_cols, 1),

padding=”equal”))

model.add(Conv2D(10, kernel\_size=2, activation='relu'))

model.add(Flatten())

model.add(Dense(3, activation=”softmax”))

# Compile model

model.compile(optimizer="adam",

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Fit the model to training data

model.fit(train\_data, train\_labels,

validation\_split=0.2,

epochs=3, batch\_size=10)

# Evaluate the model on test data

model.evaluate(test\_data, test\_labels, batch\_size=10)

model.summary()

### Counting parameters

model.add(Dense(3, activation=”relu”, input\_shape=(784,)))

parameters = 784\*10 + 10 = 7850

that’s for every pixel in the image, times the number of units plus 10 bias terms for each unit

model.add(Dense(10, activation=”relu”))

2nd layer: 1 parameter for every unit in layer 1, times the number of units in this layer plus 10 bias terms for each unit in this layer

model.add(Dense(3, activation=”softmax”))

last layer: 1 parameter for each unit in layer 2, times the number of units in this layer plus 3 bias terms for each unit in this layer

that is why the total number of parameters is 7,993

the number of parameters in a CNN:

model.add(Conv2D(10, kernel\_size=3, activation='relu',

input\_shape=(img\_rows, img\_cols, 1),

padding=”same”))

there are 9 \* 10 + 10 parameters, 9 for the 3x3 kernel, 10\* the kernel unit and the bias term for each kernel unit

model.add(Conv2D(10, kernel\_size=3, activation='relu',

input\_shape=(img\_rows, img\_cols, 1),

padding=”same”))

there are 10\* 9 \* 10 + 10 parameters, 10 kernel unit of the previous layer times 9 for the 3x3 kernel of this layer, 10\* the kernel unit and the bias term for each kernel unit

model.add(Flatten())

no parameters for flatten layer, it takes the output and flattens them to one big array

model.add(Dense(3, activation=”softmax”))

parameters: 7840\*3 +3

28x28 pixels in each feature map of which we have 10, 28x28x10 = 7840 pixels times 3 units n the last layer + 3 bias terms

In total we have 23,523 parameters.

### Pooling

Summarize a group of pixels with some aggregation function: max, min, avg

Retaining only the brightest parts of the image or the darkest parts

Result = np.zeros((im.shape[0]//2, im.shape[1]//2))

Result[0,0] = np.max(im[0:2, 0:2])

Result[0,1] = np.max(im[0:2, 2:4])

Etc.

Result[1,0] = np.max(im[2:4, 0:2])

Result[1,1] = np.max(im[2:4, 2:4])

Etc.

Sliding the window along

Looping through rows and columns, performing operations quickly

For ii in range(result.shape[0]):

For jj in range(result.shape[1]):

Result[ii, jj]= np.max(im[ii\*2:ii\*2+2, jj\*2:jj\*2+2])

Max pooling in Keras

For keras.models import Sequential

From keras.layers import Dense, Conv2D, Flatten, MaxPool2D

Model = Sequential()

Model.add(Conv2D())

Model.add(MaxPooling2(())

After each convolutional layer we add a pooling layer

Pooling will reduce the number of parameters by a large margin

# Result placeholder

result = np.zeros((im.shape[0]//2, im.shape[1]//2))

# Pooling operation

for ii in range(result.shape[0]):

for jj in range(result.shape[1]):

result[ii, jj] = np.max(im[ii\*2:ii\*2+2, jj\*2:jj\*2+2])

# Add a convolutional layer

model.add(Conv2D(15, kernel\_size=2, activation='relu',

input\_shape=(img\_rows, img\_cols, 1)))

# Add a pooling operation

model.add(MaxPool2D(2))

# Add another convolutional layer

model.add(Conv2D(5, kernel\_size=2, activation='relu'))

# Flatten and feed to output layer

model.add(Flatten())

model.add(Dense(3, activation='softmax'))

model.summary()

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

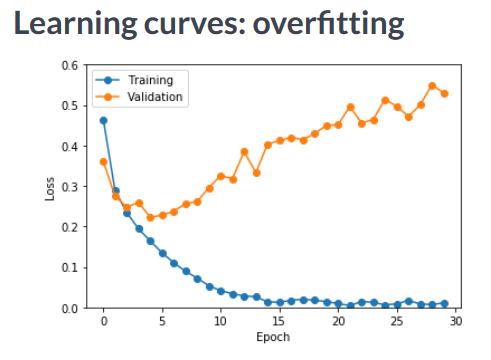
metrics=['accuracy'])

# Fit to training data

model.fit(train\_data, train\_labels, validation\_split=0.2, epochs=3, batch\_size=10)

# Evaluate on test data

model.evaluate(test\_data, test\_labels, 10)



Training = model.fit(train\_data, train\_labels, epochs=3, validation\_split=0.2)

Import matplotlib.pyplot as plt

Plt.plot(training.history[‘loss’])

Plt.plot(training.history[‘val\_loss’])

Plt.show()

From keras.callbacks import ModelCheckpoint

Checkpoint = ModelCheckpoint(‘weights.hdf5’, monitor=’val\_loss’, save\_best\_only=True)

The model is saved before the model overfits

Callbacks\_list = [checkpoint]

Model.fit(train\_data, train\_labels, validation\_split= 0.2, epochs=3, callbacks=callbacks\_list)

The file contains the best weights, load them:

Model.load\_weights(‘weights.hdf5’)

Model.predict\_classes(test\_data)

Array([2,2,…1,2,0])

import matplotlib.pyplot as plt

# Train the model and store the training object

training = model.fit(train\_data, train\_labels, epochs=3, validation\_split=0.2, batch\_size=10)

# Extract the history from the training object

history = training.history

# Plot the training loss

plt.plot(history['loss'])

# Plot the validation loss

plt.plot(history['val\_loss'])

# Show the figure

plt.show()

# Load the weights from file

model.load\_weights('weights.hdf5')

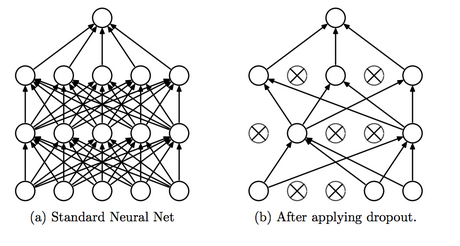
# Predict from the first three images in the test data

model.predict(test\_data[:3])

## Regularization

### Dropout

2014 introduction of the idea



Network is randomly chosen from the full network, other parts compensate for the part that is getting to sensitive to the data

Prevents parts of the network to become overcorrelated in their activity, because one unit perform horizontal the other would prefer vertical ones

From keras.models import Sequential

From keras.layers import Dense, Conv2D, Flatten, Dropout

Model = Sequential()

Model.add(Conv2D(5, kernel\_size=3, activation=”relu”, input\_shape=(img\_rows, img\_cols, 1)))

Model.add(Dropout(0.25))

Model.add(Conv2D(15, kernel\_size=3, activation=”relu”))

Model.add(Flatten())

Model.add(Dense(3, activation=”softmax”))

### Batch normalization

Takes output of a layer and rescales it so that it has always mean of 0 stddeviation of 1 in every batch of training

2015 Sergey jofeh

Solves the idea that different batches of input might generate totally different distributions of outputs in any given layer in the network

Batch normalization can be added after each layer whose output should be normalized

Model = Sequential()

Model.add(Conv2D(5, kernel\_size=3, activation=”relu”, input\_shape=(img\_rows, img\_cols, 1)))

Model.add(BatchNormalization())

Model.add(Conv2D(15, kernel\_size=3, activation=”relu”))

Model.add(Flatten())

Model.add(Dense(3, activation=”softmax”))

Warning! Sometimes dropout (slows down) and batch normalization (learning is faster) will not go together, because together it would function as well as if neither were used

= disharmony between dropout and batch normalization

# Add a convolutional layer

model.add(Conv2D(15, kernel\_size=2, activation='relu',

input\_shape=(img\_rows, img\_cols, 1)))

# Add a dropout layer

model.add(Dropout(0.2))

# Add another convolutional layer

model.add(Conv2D(5, kernel\_size=2, activation='relu'))

# Flatten and feed to output layer

model.add(Flatten())

model.add(Dense(3, activation='softmax'))

## Disadvantage

* CNNs are black boxes, if they work well, not able to understand why they work
* Visualize what different parts of the network are doing

Look at the first CNN layer:

Conv1 = Model.layers[0]

Weights1 = conv1.get\_weights()

Len(weights1)

Kernels1 = weights1[0]

Kernels1.shape

Output: (3,3,1,5) // (kernel size1, kernel size2, channel nb in kernel, nb of kernels in this layer)

Kernel1\_1 =kernels1[:,:,0,0] // pull out first kernel

Kernel1\_1.shape

3x3 array oft he kernel, then visualize the kernel

Plt.imshow(kernel1\_1)

### Visualizing the kernel responses

Test\_image = test\_data[3, :, :, 0]

Plt.imshow(test\_image)

Forth image from the test set, convolve it with th e kernel and create a filtered image:

Filtered\_image = convolution(test\_image, kernel1\_1)

Plt.imshow(filtered\_image)

Run the convolution over another image

Test\_image = test\_data[4, :, :, 1]

Plt.imshow(test\_image)

Filtered\_image = convolution(test\_image, kernel1\_1)

Plt.imshow(filtered\_image)

Kernel1\_2 = kernels[:,:,0,1]

Filtered\_image = convolution(test\_image, kernel1\_2)

Plt.imshow(filtered\_image)

# Load the weights into the model

model.load\_weights('weights.hdf5')

# Get the first convolutional layer from the model

c1 = model.layers[0]

# Get the weights of the first convolutional layer

weights1 = c1.get\_weights()

# Pull out the first channel of the first kernel in the first layer

kernel = weights1[0][...,0, 0]

print(kernel)

import matplotlib.pyplot as plt

# Convolve with the fourth image in test\_data

out = convolution(test\_data[3, :, :, 0], kernel)

# Visualize the result

plt.imshow(out)

plt.show()

## Summary

* CNNs excel at image classification
* Convolutions
* CNNs have a very large number of parameters
  + Tweak convolutions to adapt them to your problem
* See Residual networks (skip layers) for classification
* See transfer learning, if few data
* See fully convolutional networks (input image output another image)
* See generative adversarial networks (create new images)