Lecture 4: AutoEncoders and Convolutional Neural Networks (CNN)

TIES4911 Deep-Learning for Cognitive Computing for Developers
Spring 2019

by:
Dr. Oleksiy Khriyenko
IT Faculty
University of Jyväskylä
I am grateful to all the creators/owners of the images that I found from Google and have used in this presentation.
UNLABELED data
- Feature Extraction
- Unsupervised Learning
- Pattern Recognition
  - Unsupervised Learning (extraction of the patterns from a set of unlabeled data):
    - Restricted Boltzmann Machine (RBM)
    - Autoencoders

LABELED data
- Supervised Learning
  - Supervised Learning to build a classifiers, generators, etc. Depending on application use:
    - Recurrent Neural Net (RNN) or Recursive Neural Tensor Network (RNTN) for text processing (sentiment analysis, parsing, and named entity recognition)
    - Deep Belief Network or Convolutional Net (CNN) for image recognition
    - CNN or RNTN for object recognition
    - Recurrent Neural Net (RNN) for speech recognition and any language model that operates on the character level

In general, use …
- Deep Belief Networks and Multilayer Perceptrons with Rectified Linear Units (ReLU) for classification
- Recurrent Net for time series analysis

A visual and intuitive understanding of deep learning:
https://www.youtube.com/watch?v=Oqm9vsf_hvU

24/01/2019
Autoencoders
Autoencoders in Deep Learning

Data compression and Dimensionality reduction help us to increase the performance by converting our data into a smaller representation that we can recreate to a degree of quality.

Autoencoders outperform linear dimensionality reduction approaches (e.g. Principle Component Analysis (PCA)) applying non-linear transformations using multiple layers and non-linear activation functions...

Relevant links:
https://www.youtube.com/watch?v=nTt_ajul8NY
http://www.deeplearningbook.org/contents/autoencoders.html
Autoencoders are an important family of neural networks that are designed to recognize inherent patterns in data and sort of encode their own structure. Autoencoder is a neural net that takes a set of typically unlabeled inputs, and after encoding them, tries to reconstruct them as accurately as possible. As a result, the net decides which of the data features are the most important, essentially acting as a feature extraction engine.

Autoencoders are typically very shallow, and are usually comprised of an input layer, an output layer and a hidden layer. However, there are deep autoencoders that are extremely useful tools for dimensionality reduction.

**Autoencoder training parameters:**
- **Code size** – size of compressed data (amount of nodes in the middle layer)
- **Number of Layers** – could be as deep as you wish
- **Loss Function** – mean square error or binary cross entropy (if the input values in the range [0,1])
- **Number of Nodes per Layers** – usually layers structure is symmetric for encoder and decoder, and amount of nodes decrease the “code” (compressed data)

Relevant links:
- https://www.youtube.com/watch?v=nTt_ajul8NY
- http://www.deeplearningbook.org/contents/autoencoders.html
Autoencoders in Deep Learning

**Loss function** (usually consists of two terms: one encourage the model to be sensitive to the inputs (**reconstruction loss**), another one discourages memorization/overfitting (added **regularizer**).

\[ \mathcal{L}(x, \hat{x}) + \text{regularizer} \]

For **binary inputs** calculate distance as cross-entropy (sum of Bernoulli cross-entropies):

\[ l(f(x)) = -\sum_k (x_k \log(\hat{x}_k) + (1 - x_k) \log(1 - \hat{x}_k)) \]

For **real-values inputs** use a linear activation function at the output and calculate distance as sum of squared differences (squared Euclidian distance):

\[ l(f(x)) = \frac{1}{2} \sum_k (\hat{x}_k - x_k)^2 \]

Relevant links:
https://www.jeremyjordan.me/autoencoders/
https://www.youtube.com/watch?v=xTU79Zs4XY
Autoencoders in Deep Learning

**Overcomplete Hidden Layers** – a concept in which autoencoders may recognize more features, having a hidden layer that is equal to the amount of inputs nodes or greater. However, it might bring a huge problem when trained and make it useless…

Both types of autoencoders can be shallow or deep...

**Sparse autoencoders** – is an approach towards regularization via regularization of the activations, not the weights of a network. They offer an alternative method for introducing an information bottleneck without requiring a reduction in the number of nodes at our hidden layers. Here loss function supposed to penalize activations within a layer.

**Denoising autoencoders** – is an approach towards developing a generalizable model by slightly corrupting the input data but still maintaining the uncorrupted data as a target output.

**Contractive autoencoders** add a penalty to the backpropagation of the autoencoder preventing it from cheating. Here we explicitly encouraging the model to learn an encoding in which similar inputs have similar encodings. Denoising autoencoders make the reconstruction function (ie. decoder) resist small but finite-sized perturbations of the input, while contractive autoencoders make the feature extraction function (ie. encoder) resist infinitesimal perturbations of the input.

**Autoencoders** learn how to compress the data based on attributes (ie. correlations between the input feature vector) discovered from data during training, these models are typically only capable of reconstructing data similar to the class of observations of which the model observed during training.
Autoencoders Use-cases

- Data compression
- Dimensionality Reduction
- Image reconstruction
- Image coloring
- Feature variation and Denoising
- Generating higher resolution images
- Image search and information retrieval
- ...

Relevant links:
https://www.edureka.co/blog/autoencoders-tutorial/
import numpy as np
from keras.layers import Input, Dense
from keras.models import Model
from keras.datasets import mnist
import matplotlib.pyplot as plt

# size of encoded representations
encoding_dim = 32  # compression (vs. the input is 784)
# input placeholder
input_img = Input(shape=(784,))
# encoded representation of the input
encoded = Dense(encoding_dim, activation='relu')(input_img)
# lossy reconstruction of the input
decoded = Dense(784, activation='sigmoid')(encoded)
# model that maps an input to its reconstruction
autoencoder = Model(input_img, decoded)
# model that maps an input to its encoded representation
encoder = Model(input_img, encoded)

# create a placeholder for an encoded (32-dimensional) input
encoded_input = Input(shape=(encoding_dim,))
# retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# decoder model
decoder = Model(encoded_input, decoder_layer(encoded_input))
# configure the model
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')

# read dataset
(x_train, _), (x_test, _) = mnist.load_data()
# normalize all values between 0 and 1 and flatten
# the 28x28 images into vectors of size 784.
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

# fit autoencoder
autoencoder.fit(x_train, x_train, epochs=50, batch_size=256, shuffle=True, validation_data=(x_test, x_test))

# encode and decode some digits, taking them from the 'test' set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)

# visualise the results
n = 20  # how many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original
    ax = plt.subplot(3, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    # display code
    ax = plt.subplot(3, n, i + 1 + n)
    plt.imshow(encoded_imgs[i].reshape(8, 4))
    plt.gray()
    # display reconstruction
    ax = plt.subplot(3, n, i + 1 + 2*n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
plt.show()
### Sparse Deep Net

def getDeepAE():
    # input layer
    input_layer = Input(shape=(784,))
    # encoding architecture
    encode_layer1 = Dense(1024, activation='relu')(input_layer)
    encode_layer2 = Dense(1024, activation='relu')(encode_layer1)
    # latent view
    latent_view = Dense(1024, activation='sigmoid',
                       activity_regularizer=regularizers.l1(1e-5))(encode_layer2)
    # decoding architecture
    decode_layer1 = Dense(1024, activation='relu')(latent_view)
    decode_layer2 = Dense(1024, activation='relu')(decode_layer1)
    # output layer
    output_layer = Dense(784)(decode_layer2)
    # model
    model = Model(input_layer, output_layer)
    return model

### Vanila Net

def getVanilaAE():
    # input layer
    input_layer = Input(shape=(784,))
    # latent view
    latent_view = Dense(64, activation='sigmoid')(input_layer)
    # output layer
    output_layer = Dense(784)(latent_view)
    # model
    model = Model(input_layer, output_layer)
    return model

### Deep Net

def getDeepAE():
    # input layer
    input_layer = Input(shape=(784,))
    # encoding architecture
    encode_layer1 = Dense(512, activation='relu')(input_layer)
    encode_layer2 = Dense(256, activation='relu')(encode_layer1)
    encode_layer3 = Dense(128, activation='relu')(encode_layer2)
    # latent view
    latent_view = Dense(64, activation='sigmoid')(encode_layer3)
    # decoding architecture
    decode_layer1 = Dense(128, activation='relu')(latent_view)
    decode_layer2 = Dense(256, activation='relu')(decode_layer1)
    decode_layer3 = Dense(512, activation='relu')(decode_layer2)
    # output layer
    output_layer = Dense(784)(decode_layer3)
    # model
    model = Model(input_layer, output_layer)
    return model

Different architectures of the models…
Deep Learning

Restricted Boltzmann Machine (RBM) is a very popular example of an autoencoder) allows automatically find patterns in data by reconstructing the input. It makes decisions about which input features are important and how they should be combined to form patterns. An RBM is an example of an autoencoder with only two layers.

Deep Belief Net (DBN) could be considered as a stack (a combination) of several RBMs that detect patterns based on unlabeled data, and need relatively small amount of labeled data to organize them for classification purpose.

Each RBM layer learns the entire input. After this initial training, the RBMs have created a model that can detect inherent patterns in the data. To finish training, we need to introduce labels to the patterns and fine-tune the net with supervised learning.

- DBN only needs a small labelled data set, which is important for real-world applications.
- the training process can also be completed in a reasonable amount of time through the use of GPUs.
- the resulting net will be very accurate compared to a shallow net

Relevant links:
https://www.edureka.co/blog/restricted-boltzmann-machine-tutorial/

24/01/2019 TIES4911 – Lecture 4
Convolutional Neural Networks (CNN)
Our expectation: Computer takes an array of pixels (w x h x 3) with RGB values (0..255) and produces output number(s) that describe the probability of the image being a certain class (.80 for dog, .15 for cat, .05 for car, etc).
Fully connected DNN is very rarely used in computer vision tasks…

- Too many parameters require too much training data, computation power and time

- No generalization, no spatial invariance for images
**CNNs**

**Approach to be used:** In a similar way as humans classify pictures with a dog if it has identifiable features such as paws or 4 legs, the computer is supposed to be able to perform image classification by looking for low-level features such as edges and curves, and then building up to more abstract concepts through a series of convolutional layers. Therefore, we have to train a computer to differentiate between all the images it’s given and by figuring out the unique features that make a dog a dog or that make a cat a cat, etc.

**Convolutional Neural Networks (CNN)** are probably the most popular deep learning architecture for the moment.

**CNN progressed from 8 layer AlexNet in 2012 towards 152 layer ResNet in 2015.**

**Applied for:**
- image related problem
- recommender systems
- natural language processing
- etc.

**The main advantages:**
- Automatic detection of important features without any human supervision (e.g. given many pictures of two different classes it learns distinctive features for each class by itself).
- Using special convolution and pooling operations and parameter sharing, CNN becomes computationally efficient and enables models to be run on any device.
**CNN model** is kind of a combination of two components:

**Feature extraction.** The convolution + pooling layers perform feature extraction and make possible to detect features like two eyes, four legs, road sign, two wheels, building, face, etc. The convolution layers learn these complex features by building on top of each other. The first layers detect edges, the next layers combine them to detect shapes, to following layers merge this information to infer complete objects.

**Classification.** The fully connected layers act as a classifier on top of the extracted features, and assign a probability for the input image being a representative of certain class.
A convolution is an integral that expresses the amount of overlap of one function $g$ as it is shifted over another function $f$. It therefore "blends" one function with another. In another words, *Convolution* - is a mathematical operation to merge two sets of information.

Convolution on signal processing is used for the following use cases:
- Filter signals (1D audio, 2D image processing)
- Check how much a signal is correlated to another
- Find patterns in signals

Relevant links:
- https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/convolution.html

24/01/2019
2D convolutions are used as image filters, and when you would like to find a specific patch on an image. We use a convolution filter on top of the input data to produce a feature map.

Convolution operation on matrices (images context):
- slide the filter over the input
- do element-wise matrix multiplication and sum the result at every sliding location (receptive field)

Nice visual explanation of Image Kernels: http://setosa.io/ev/image-kernels/
**2D Convolution**

*Stride* specifies how much we move the convolution filter at each step (e.g. stride 1 ([1,1]), stride 2 ([2,2]), etc.). We can have bigger strides if we want less overlap between the receptive fields making the resulting feature map smaller. We can use *padding* to maintain the feature map’s dimensionality similar to the input by surrounding input with zeros or the values on the edge.
Convolution

# Importing
import numpy as np
from scipy import signal
from scipy import misc
import matplotlib.pyplot as plt
from PIL import Image

### Load image of your choice on the notebook
im = Image.open('bird.jpg')  # type here your image's name
# uses the ITU-R 601-2 Luma transform (there are several
# ways to convert an image to grey scale)
image_gr = im.convert("L")
print("\n Original type: %r \n\n" % image_gr)

# convert image to a matrix with values from 0 to 255 (uint8)
arr = np.asarray(image_gr)
print("After conversion to numerical representation: \n\n %r\n\n" % arr)

### Activating matplotlib for Ipython
%matplotlib inline

### Plot image
imgplot = plt.imshow(arr)
# you can experiment different colormaps (Greys,winter,autumn)
imgplot.set_cmap('gray')
print("\n Input image converted to gray scale: \n\n")
plt.show(imgplot)
kern = np.array([[ 0, 1, 0],
                 [ 1,-4, 1],
                 [ 0, 1, 0],])

24/01/2019 TIES4911 – Lecture 4
Usually, an image is represented as a 3D matrix with dimensions of height, width and depth (color channels - RGB). In turn, a convolution filter become 3D as well having a specific height and width (e.g. 3x3 or 5x5), and by design covering the entire depth of the input.

Using different filters, we perform multiple convolutions on an input and stack all the distinct feature maps together forming the final output of the convolution layer. Each of these channels (distinct feature maps) will end up being trained to detect certain key features in the image (e.g. lines, edges or other distinctive shapes). Therefore, the output of the convolutional layer will actually be 4 dimensional (for a 3D image).
2D Convolution

To be powerful, neural network (that basically has just been computing linear operations during the conv layers) needs to contain non-linearity. It is achieved by passing the weighted sum of its inputs through an activation function.

In CNN we pass the result of the convolution operation through ReLU activation function, that works far better (comparably to tanh and sigmoid) and makes the network able to train a lot faster (due to the computational efficiency) without making a significant difference to the accuracy. So the values in the final feature maps are not actually the sums, but the ReLU function applied to them.

Short summary on Convolution Layer:

There are 4 important hyperparameters to decide on:

- **Filter size.** 3x3 filters are typically used, but 5x5 or 7x7 are also applicable depending on the application. Even 1x1 filters have sense, since filters are 3D and have a depth dimension as well.
- **Filters amount.** This is the most variable parameter, it’s a power of two anywhere between 32 and 1024. More filters make a model more powerful, but we risk overfitting due to increased amount of parameters. One of the strategies is to start with a small number of filters at the initial layers, and progressively increase number of them as go deeper into the network.
- **Stride.** Usually default value is 1 (stride [1,1]).
- **Padding.** Usually use padding.

Relevant links:

http://www.cs.toronto.edu/~fritz/absps/reluICML.pdf
Pooling

To reduce the number of parameters, which both shortens the training time and combats overfitting, we reduce the dimensionality of convolution operation result by performing a **pooling** (sub-sampling) operation. It is also a “sliding window” technique, that applies some sort of statistical function over the values within the window.

**max pooling** - the most common type of pooling operation that just slides a window over the input, and simply takes the max value in the window.

- There are some other alternatives e.g. mean (average) pooling or L2-norm pooling. Similar to a filter size and sliding step in convolution, we specify the window size and stride for pooling operation.
- Pooling layers reduce the height and width, keeping the depth of convolution layer output intact. It down-sample each feature map independently, while keeping the important information.
- The intuition behind the pooling layer is that once there is a high activation value, there will be a specific feature in the original input volume and its exact location is not as important as its relative location to the other features.

In CNN architectures, **pooling** is typically performed with **2x2 windows, stride 2 (stride [2,2]) and no padding**. While **convolution** is done with **3x3 windows, stride 1 and with padding**.
Pooling in CNN:

- reduces the dimensionality of the output (called down-sampling), and reduces the number of parameters (or weights) in the model, thus lessening the computation cost.
- by controlling overfitting, makes the detection of certain features in the input invariant to scale and orientation changes.

Pooling acts as a generalizer of the lower level information and so enables us to move from high resolution data to lower resolution information. In other words, pooling coupled with convolutional filters attempt to detect objects within an image.
In CNN architecture, a chain of the convolution + pooling layers is wrapped up with a couple of fully connected layers.

At the output of the convolutional-pooling layers we have moved from high resolution low level data about the pixels to representations of objects within the image. The purpose of these final, fully connected layers is to make classifications regarding these objects – so, we insert a standard neural network classifier onto the end of a trained object detector.

Since the output of both convolution and pooling layers are 3D volumes and a fully connected layer accepts input in a form of a 1D vector of numbers, the output of the final pooling layer should be flatten to a vector.
CNNs compare images based on generalized features rather than on pixel basis. Comparing images piece by piece, CNNs try to find rough feature matches in roughly the same positions in two images.

Relevant links:
https://www.edureka.co/blog/convolutional-neural-network/
**CNN simple example**

**Convolution** makes a filtering of the image by calculating the match to a feature across it.

To calculate the match of a feature to a patch of the image, simply multiply each pixel in the feature by the value of the corresponding pixel in the image. Then add up the answers and divide by the total number of pixels in the feature.

Next repeat the convolution process in its entirety for each of the other features and get a set of filtered images, one for each of our filters.

Relevant links:
https://www.edureka.co/blog/convolutional-neural-network/

24/01/2019 TIES4911 – Lecture 4
**Pooling** is a way to take large images and shrink them down while preserving the most important information in them.

The math behind pooling consists of stepping a small window across an image and taking the maximum value (max pooling) from the window at each step.

After pooling, an image has about a quarter as many pixels as it started with. Because it keeps the maximum value from each window, it preserves the best fits of each feature within the window. This means that it doesn’t care so much exactly where the feature fit as long as it fit somewhere within the window.

Relevant links:
https://www.edureka.co/blog/convolutional-neural-network/
**Rectified Linear Unit (ReLU)** helps the CNN stay mathematically healthy by keeping learned values from getting stuck near 0 or blowing up toward infinity.

Wherever a negative number occurs, swap it out for a 0.

Layers could be stacked like Lego bricks.

As a result, raw images get filtered, rectified and pooled to create a set of shrunken, feature-filtered images. Each time, the features become larger and more complex, and the images become more compact. This lets lower layers represent simple aspects of the image (e.g. edges and bright spots). Higher layers can represent increasingly sophisticated aspects of the image, such as shapes and patterns.

Relevant links:
- https://www.edureka.co/blog/convolutional-neural-network/
**Fully connected layers** take the high-level filtered images and translate them into votes.

Instead of treating inputs as a two-dimensional array, Fully connected layers are treated as a single list and all treated identically. Every value gets its own vote on whether the current image is an X or an O. Some values are much better than others at knowing when the image is an X, and some are particularly good at knowing when the image is an O. These get larger votes than the others. These votes are expressed as weights, or connection strengths, between each value and each category.

Relevant links:
- [https://www.edureka.co/blog/convolutional-neural-network/](https://www.edureka.co/blog/convolutional-neural-network/)
Training…
- What values filters in each layer should have?
- How do the filters in the initial conv layers know to what edges and curves they should look for?
- How filters on further layers know which edges and curves aggregate to construct higher level features?
- How does the fully connected layer know what activation maps to look at?
- etc.

To address these questions, CNN is trained the same way as ordinary ANN using Backpropagation with gradient descent to adjust its filter values (weights):

- forward pass: on the first training example (W x H x 3 array of numbers that represents image) all of the weights/filter values were randomly initialized and the output will not give preference to any classification class in particular (e.g. [0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1] – probability distribution).

- loss function: having a training data with both an image and a label, we will use labels to compute a loss against the output of previous step (forward pass). One of the common ways to define a loss function we want to minimize is cross entropy between target (e.g. [0 0 0 0 0 0 1 0 0] that represents digit “7”) and actual output:

- backward pass: solving an optimization problem, we find out which weights most directly contributed to the loss of the network. We have to calculate a derivative of the loss dE/dW (where W are the weights at a particular layer) to determining which weights contributed most to the loss and find ways to adjust them so that the loss decreases.

- weight update: update all the weights of the filters so that they change in the opposite direction of the gradient. Where the learning rate is a parameter chosen by the programmer to define a step in weight change.

Relevant links:
https://en.wikipedia.org/wiki/Backpropagation
http://neuralnetworksanddeeplearning.com/chap2.html
https://brilliant.org/wiki/backpropagation/
We may face a case, when training loss keeps going down but the validation loss starts increasing after certain epoch. It is a sign of **overfitting**. It tells us that our model is memorizing the training data, but it’s failing to generalize to new instances.

The most popular regularization technique for deep neural networks is **Dropout**. It is used to prevent overfitting via temporarily “dropping”/disabling a neuron with probability $p$ (a hyperparameter called the *dropout-rate* that is typically a number around 0.5) at each iteration during the training phase.

- The dropped-out neurons are resampled with probability $p$ at every training step (a dropped out neuron at one step can be active at the next one). (Dropout is not applied during test time after the network is trained).
- Dropout can be applied to input or hidden layer nodes but not the output nodes.

**Reason.** Dropout forces every neuron to be able to operate independently and prevents the network to be too dependent on a small number of neurons.

Relevant links:

24/01/2019 TIES4911 – Lecture 4
Batch Normalization is a popular regularization technique…

Batch Normalization is a method to reduce internal covariate shift in neural networks, improving the performance and stability of neural networks. To speed up learning, it is reasonable to normalize not only the input layer, but similarly, to apply normalization for the values in the hidden layers as well, improving training speed and overall accuracy. Batch normalization allows each layer of a network to learn by itself a little bit more independently of other layers.

- **Speeds up training**: it should converge much quicker, even though each training iteration will be slower because of the extra normalisation calculations during the forward pass and the additional hyperparameters to train during back propagation.
- **Allows higher learning rates**: as networks get deeper, initially small gradients get even smaller during back propagation, and so require even more iterations. Batch normalisation allows much higher learning rates, increasing the speed at which networks train.
- **Simplifies weights initialization**: batch normalisation helps reduce the sensitivity to the initial starting weights.
- **Makes activation functions viable**: as batch normalisation regulates the values going into each activation function, nonlinearities that don't work well in deep networks tend to become viable again.
- **Reduces overfitting**: it has a slight regularization effects and, similarly to dropout, it adds some noise to each hidden layer’s activations. If we use batch normalization, we will use less dropout, which is a good thing because we are not going to lose a lot of information. However, we should not depend only on batch normalization for regularization; we should better use it together with dropout.

**Batch normalization** adds two trainable parameters to each layer, so the normalized output is multiplied by a “standard deviation” parameter and add a “mean” parameter. Being used before the activation function and dropout, Batch Normalization layer applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.

---

**Relevant links:**
- https://keras.io/layers/normalization/

---

24/01/2019 TIES4911 – Lecture 4
Overfitting happens when we are training on very few examples (e.g. 1000 images per category), and even use of dropout does not help us much. To start thing about Deep Learning, we have to operate with at least 100K training examples. On a small dataset we will overfit no matter which regularization technique is applied.

Fortunately, there is a **data augmentation** method which enables us to train deep models on small datasets. It allows us to artificially boost the size of the training set through enrichment or “augmentation” of the training data by generating new examples via random transformation of existing ones.

Data augmentation is done dynamically during training time. To generate realistic images, the common transformations are: rotation, shifting, resizing, exposure adjustment, contrast change etc.

Also, some data cleaning tricks on images can be applied (e.g. whitening and mean normalization).

http://ufldl.stanford.edu/tutorial/unsupervised/PCAWhitening/

Relevant links:
https://keras.io/preprocessing/image/
https://machinelearningmastery.com/image-augmentation-deep-learning-keras/
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import matplotlib.pyplot as plt
import numpy as np
import math

# Import data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)

# Weight variable
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

# Bias variable
def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)

# Convolution
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

# Max pool
def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

# Get activations
def getActivations(layer, stimuli):
    units = sess.run(layer, feed_dict={x: np.reshape(stimuli, [1, 784], order='F'), keep_prob: 1.0})
    plotNNFilter(units)

# Plot NN filter
def plotNNFilter(units):
    filters = units.shape[3]
    plt.figure(1, figsize=(20, 20))
    n_columns = math.ceil(filters / 6)
    n_rows = math.ceil(filters / n_columns) + 1
    for i in range(filters):
        plt.subplot(n_rows, n_columns, i + 1)
        plt.title('Filter ' + str(i))
        plt.imshow(units[0, :, :, i], interpolation='nearest', cmap='gray')

# Create the model
x = tf.placeholder(tf.float32, [None, 784])
x_image = tf.reshape(x, [-1, 28, 28, 1])
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
W_pool1 = max_pool_2x2(h_conv1)
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
W_pool2 = max_pool_2x2(h_conv2)
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
W_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
W_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
W_pool1 = max_pool_2x2(h_conv1)
W_conv2 = tf.nn.relu(conv2d(W_pool1, W_conv2) + b_conv2)
W_pool2 = max_pool_2x2(h_conv2)
W_fc1 = tf.nn.relu(tf.matmul(W_pool2, W_fc1) + b_fc1)
W_fc2 = tf.nn.relu(tf.matmul(W_pool2, W_fc2) + b_fc2)

# Create the model
x = tf.placeholder(tf.float32, [None, 784])
x_image = tf.reshape(x, [-1, 28, 28, 1])
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
W_pool1 = max_pool_2x2(h_conv1)
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
W_pool2 = max_pool_2x2(h_conv2)
W_fc1 = weight_variable([7*7*64, 1024])
b_fc1 = bias_variable([1024])
W_pool2 = tf.reshape(h_pool2, [-1, 7*7*64])
W_pool2 = tf.nn.relu(W_pool2[1, 7*7*64])
W_fc1 = tf.placeholder(tf.float32, [None, 10])
W_fc2 = bias_variable([10])
W_conv1 = tf.nn.relu(W_conv1 + b_conv1)
W_pool1 = max_pool_2x2(h_conv1)
W_conv2 = tf.nn.relu(W_pool1 + b_conv2)
W_pool2 = max_pool_2x2(h_conv2)
W_fc1 = tf.nn.relu(W_pool2 + b_fc1)
W_fc2 = tf.nn.relu(W_pool2 + b_fc2)

# Create the model
x = tf.placeholder(tf.float32, [None, 784])
x_image = tf.reshape(x, [-1, 28, 28, 1])
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
W_pool1 = max_pool_2x2(h_conv1)
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
W_pool2 = max_pool_2x2(h_conv2)
W_fc1 = weight_variable([7*7*64, 1024])
b_fc1 = bias_variable([1024])
W_pool2 = tf.reshape(h_pool2, [-1, 7*7*64])
W_pool2 = tf.nn.relu(W_pool2[1, 7*7*64])
W_fc1 = tf.placeholder(tf.float32, [None, 10])
W_fc2 = bias_variable([10])
W_conv1 = tf.nn.relu(W_conv1 + b_conv1)
W_pool1 = max_pool_2x2(h_conv1)
W_conv2 = tf.nn.relu(W_pool1 + b_conv2)
W_pool2 = max_pool_2x2(h_conv2)
W_fc1 = tf.nn.relu(W_pool2 + b_fc1)
W_fc2 = tf.nn.relu(W_pool2 + b_fc2)
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])

cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits_v2(labels=y_,
    logits=y_conv))

train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)

correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))

accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# Add ops to save and restore all the variables.
saver = tf.train.Saver()

with tf.Session() as sess:
sess.run(tf.global_variables_initializer())

for i in range(20000):
    batch = mnist.train.next_batch(50)
    if i % 100 == 0:
        train_accuracy = accuracy.eval(feed_dict={
            x: batch[0], y_: batch[1], keep_prob: 1.0})
        print('step %d, training accuracy %g' % (i, train_accuracy))
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print('test accuracy %g' % accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))

save_path = saver.save(sess, "my_models_tmp/model.ckpt")
print("Model saved in file: %s" % save_path)

saver.restore(sess, "my_models_tmp/model.ckpt")
print("Model restored.")

# Get 28x28 image
image_to_use = mnist.test.images[47]
sample_1 = image_to_use.reshape(28,28)

# Get corresponding integer label from one-hot encoded data
sample_label_1 = np.where(mnist.test.labels[47] == 1)[0][0]

# Plot sample
plt.imshow(sample_1, cmap='Greys')
plt.title('label = {}'.format(sample_label_1))
plt.show()

predicted_value = tf.argmax(y_conv, 1)
print("is classified as: " + str(predicted_value.eval(session=sess, feed_dict={
    x:image_to_use.reshape(1,784),
    keep_prob: 1.0})))

print("conv layer:"
getActivations(h_conv2,image_to_use)
The TensorFlow layers module provides a high-level API that makes it easy to construct a neural network.
https://www.tensorflow.org/api_docs/python/tf/layers

It provides methods that facilitate the creation of dense (fully connected) layers and convolutional layers, adding activation functions, and applying dropout regularization.

As an example... tf.layers.conv2d handles activation and bias automatically while you have to write additional codes for these if you use tf.nn.conv2d.

```
tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=None, data_format=None, name=None)
```

```
tf.layers.conv2d(inputs, filters, kernel_size, strides=(1, 1), padding='valid', data_format='channels_last', dilation_rate=(1, 1), activation=None, use_bias=True, kernel_initializer=None, bias_initializer=tf.zeros_initializer(), kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, trainable=True, name=None, reuse=None)
```

Check the tutorial that explains how to build a convolutional neural network model to recognize the handwritten digits in the MNIST data set using layers and creating corresponding Estimator to handle the model. https://www.tensorflow.org/tutorials/layers
Convolution NN on MNIST dataset with TensorFlow-Slim

https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim

```python
import numpy as np
import matplotlib as mp
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import tensorflow.contrib.slim as slim
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("MNIST_data", one_hot=True)

with tf.Graph().as_default():
    x = tf.placeholder(tf.float32, [None, 784], name="x-in")
    true_y = tf.placeholder(tf.float32, [None, 10], name="y-in")
    keep_prob = tf.placeholder("float")

    x_image = tf.reshape(x, [-1, 28, 28, 1])
    hidden_1 = slim.conv2d(x_image, 5, [5, 5])
    pool_1 = slim.max_pool2d(hidden_1, [2, 2])
    hidden_2 = slim.conv2d(pool_1, 10, [5, 5])
    pool_2 = slim.max_pool2d(hidden_2, [2, 2])
    hidden_3 = slim.conv2d(pool_2, 20, [5, 5])
    hidden_3 = slim.dropout(hidden_3, keep_prob)
    out_y = slim.fully_connected(slim.flatten(hidden_3), 10,
                                 activation_fn=tf.nn.softmax)

    cross_entropy = tf.reduce_mean(
        tf.nn.softmax_cross_entropy_with_logits_v2(
            labels=true_y, logits=out_y))
    correct_prediction = tf.equal(tf.argmax(out_y, 1),
                                  tf.argmax(true_y, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)

    batchSize = 50
    sess = tf.Session()
    init = tf.global_variables_initializer()
    sess.run(init)
    for i in range(501):
        batch = mnist.train.next_batch(batchSize)
        sess.run(train_step,
                 feed_dict={x: batch[0], true_y: batch[1],
                            keep_prob: 0.5})
        if i % 100 == 0 and i != 0:
            trainAccuracy = sess.run(accuracy,
                                      feed_dict={x: batch[0], true_y: batch[1],
                                                 keep_prob: 1.0})
            print("step %d, training accuracy %g" % (i, trainAccuracy))

    testAccuracy = sess.run(accuracy,
                             feed_dict={x: mnist.test.images,
                                        true_y: mnist.test.labels, keep_prob: 1.0})
    print("test accuracy %g" % (testAccuracy))

    print('original image:')
    imageToUse = mnist.test.images[47]
    sample_img = imageToUse.reshape(28, 28)
    sample_label_1 = np.where(mnist.test.labels[47] == 1)[0][0]
    # Plot sample
    plt.imshow(sample_img, interpolation="nearest", cmap='Greys')
    plt.title('label = {}'.format(sample_label_1))
    plt.show()
```

24/01/2019

TIES4911 – Lecture 4
Convolution NN on MNIST dataset with Keras

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
get_ipython().magic(u'matplotlib inline')
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.optimizers import Adam
from keras.layers.normalization import BatchNormalization
from keras.utils import np_utils
from keras.layers import Conv2D, MaxPooling2D, ZeroPadding2D, GlobalAveragePooling2D
from keras.layers.advanced_activations import LeakyReLU
from keras.preprocessing.image import ImageDataGenerator

(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

number_of_classes = 10
Y_train = np_utils.to_categorical(y_train, number_of_classes)
Y_test = np_utils.to_categorical(y_test, number_of_classes)

# Three steps to create a CNN
# 1. Convolution
# 2. Activation
# 3. Pooling
# Repeat Steps 1,2,3 for adding more hidden layers

# Four steps to create a CNN
# 1. Convolution
# 2. Activation
# 3. Pooling
# 4. After that make a fully connected network
# This fully connected network gives ability to the CNN to classify the samples

model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(28,28,1)))
model.add(BatchNormalization(axis=-1))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())

model.add(Dense(512))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(10))
model.add(Activation('softmax'))
```

24/01/2019 TIES4911 – Lecture 4
Convolution NN on MNIST dataset with Keras

```python
model.compile(loss='categorical_crossentropy', optimizer=Adam(), metrics=['accuracy'])
gen = ImageDataGenerator(rotation_range=8, width_shift_range=0.08, shear_range=0.3, height_shift_range=0.08, zoom_range=0.08)
test_gen = ImageDataGenerator()
train_generator = gen.flow(X_train, Y_train, batch_size=64)
test_generator = test_gen.flow(X_test, Y_test, batch_size=64)
model.fit_generator(train_generator, batch_size=64, epochs=5, validation_data=test_generator)
```

The CIFAR-10 and CIFAR-100 are labeled subsets of the 80 million tiny images dataset, collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.
https://www.cs.toronto.edu/~kriz/cifar.html

CNN on CIFAR-10 tutorial: https://www.tensorflow.org/tutorials/deep_cnn

Some datasets: http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html
MNIST CNN based on csv data: https://github.com/tgjeon/kaggle-MNIST/blob/master/mnist.py

Relevant links:
https://elitedatascience.com/keras-tutorial-deep-learning-in-python
http://cv-tricks.com/tensorflow-tutorial/keras/
http://adventuresinmachinelearning.com/keras-tutorial-cnn-11-lines/

24/01/2019 TIES4911 – Lecture 4
CNN on Fashion-MNIST dataset...

```python
from __future__ import print_function
import keras
from keras.datasets import fashion_mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Conv2D, MaxPooling2D, Flatten
from keras.optimizers import RMSprop, SGD
import matplotlib.pyplot as plt
%matplotlib inline

batch_size = 128
num_classes = 10
epochs = 20

# Data Preparation
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

---

**Label Description**
- 0 T-shirt
- 1 Trouser
- 2 Pullover
- 3 Dress
- 4 Coat
- 5 Sandal
- 6 Shirt
- 7 Sneaker
- 8 Bag
- 9 Ankle boot

---

```python
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# Model Building
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
... model.add(Dense(num_classes, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))

score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```
Spatial Transformation

**Spatial Transformer Networks** (by a group at Google Deepmind)

Even though CNNs define an exceptionally powerful class of models, they are still limited by the lack of ability to be spatially invariant to the input data in a computationally and parameter efficient manner. To show good results, model should be trained on a huge variety of “transformed” samples...

**Spatial Transformer** is a new learnable module, which explicitly allows the spatial manipulation of data within the network. This differentiable module can be inserted into existing convolutional architectures, giving neural networks the ability to actively spatially transform feature maps, conditional on the feature map itself, without any extra training supervision or modification to the optimization process. Authors has shown that the use of spatial transformers results in models which learn invariance to translation, scale, rotation and more generic warping, resulting in state-of-the-art performance on several benchmarks, and for a number of classes of transformations.

- Instead of making changes to the main CNN architecture itself, the authors offer making changes to the image before it is fed into the specific conv layer.
- The module transforms the input image in a way so that the subsequent layers have an easier time making a classification...
- The module can be dropped into a CNN at any point and helps the network learn how to transform feature maps in a way that minimizes the cost function during training...

Relevant links:
- Video from Deepmind: https://drive.google.com/file/d/0B1nQa_sA3W2iN3RQLXVFRkNXN0k/view
- https://gist.github.com/kvn219/b42d382a06e6f3254bf00e780e9b8e0f

24/01/2019 TIES4911 – Lecture 4
**Capsule Net** as an improvement of the CNN...

CNNs are awesome and can do amazing things. Nonetheless, they have their limits and they have fundamental drawbacks. Internal data representation of a convolutional neural network does not take into account important spatial hierarchies between simple and complex objects...

Geoffrey Hinton and his team introduced a completely new type of neural network based on so-called *capsules*. In addition to that, the team published an algorithm, called **dynamic routing between capsules** (October 2017), that allows to train such a network. Capsule is introduced as a group of neurons (subnet) that are nested inside the convolutional layer. Authors also introduced a corresponding new non-linearity function to be applied to the capsule...

- **High accuracy on MNIST (but not yet so good on CIFAR10). Is not tested yet on larger images (e.g. ImageNet)**
- **Requires less training data, but more training time (due to the inner loop)**
- **Position and pose information are preserved (equivariance), that is promising for image segmentation and object detection.**
- **Offers robustness to affine transformations (translation, scale, shear, rotation, etc.)**
- **A CapsNet cannot see two very close identical objects (it is called “crowding”).**

Relevant links:
- [https://www.youtube.com/watch?v=pPN8d0E3900](https://www.youtube.com/watch?v=pPN8d0E3900)
- [https://www.youtube.com/watch?v=6S1_WqE55UQ](https://www.youtube.com/watch?v=6S1_WqE55UQ)
- [https://hackernoon.com/what-is-a-capsnet-or-capsule-network-2bfbe48769cc](https://hackernoon.com/what-is-a-capsnet-or-capsule-network-2bfbe48769cc)
- [https://medium.com/botsupply/running-capsulenet-on-tensorflow-1099f5c67189](https://medium.com/botsupply/running-capsulenet-on-tensorflow-1099f5c67189)
- [https://www.youtube.com/watch?v=2Kawrd5szHE](https://www.youtube.com/watch?v=2Kawrd5szHE)
- [https://www.youtube.com/watch?v=VKCLvGnq15RM](https://www.youtube.com/watch?v=VKCLvGnq15RM)
- [https://github.com/jaesik817/adv_attack_capsnet](https://github.com/jaesik817/adv_attack_capsnet)
- [https://github.com/naturomics/CapsNet-Tensorflow](https://github.com/naturomics/CapsNet-Tensorflow)
- [https://github.com/llSourcell/capsule_networks](https://github.com/llSourcell/capsule_networks)
- [https://github.com/bourdakos1/capsule-networks](https://github.com/bourdakos1/capsule-networks)
- [https://github.com/XifengGuo/CapsNet-Keras](https://github.com/XifengGuo/CapsNet-Keras)
- [https://openreview.net/pdf?id=HJWLlGWRb](https://openreview.net/pdf?id=HJWLlGWRb)
- [https://becominghuman.ai/understand-and-apply-capsnet-on-traffic-sign-classification-a592e2d4a4ea](https://becominghuman.ai/understand-and-apply-capsnet-on-traffic-sign-classification-a592e2d4a4ea)