

# Introduction to Deep Learning with Keras

Ian J. Watson

ian.james.watson@cern.ch

University of Seoul

Kyungpook National University  
September 25, 2019



**KRF** KOREA RESEARCH FELLOWSHIP  
해외 우수신진연구자 유치사업



- Slides available from:

<https://github.com/watson-ij/mltute>

- In particular, the talk should be rendered as a webpage at:

<https://github.com/watson-ij/mltute/blob/master/20180907-mltute-iwatson.org>

- Other Resources:

- Keras documentation: <http://keras.io/>
- Large repository of various tutorials, examples, and applications curated by the original author of Keras.  
<https://github.com/fchollet/keras-resources>
- Another tutorial presentation:  
[https://uwaterloo.ca/data-science/sites/ca.data-science/files/uploads/files/keras\\_tutorial.pdf](https://uwaterloo.ca/data-science/sites/ca.data-science/files/uploads/files/keras_tutorial.pdf)

- If you're already a deep learning master, not the talk for you!
- Get up and running quickly with Deep Learning
  - In particular, the goal is to build neural networks you can take home today!
- Therefore, use *Keras* to get up and running quickly
- Outline of the session:
  - Basic Usage of Keras (Iris)
  - Convolutional Neural Net in Keras (MNIST)
  - GANs (using MNIST)
- For this lecture, I recommend using *google colaboratory*
  - Machine learning education and research tool setup by google, all the packages are installed, just need a google account to sign in

<https://colab.research.google.com>

*However,*

If you want to follow along with a local setup:

With python and pip installed, you can pull the dependencies by pip installing (you might need to add ‘-user’ to the end of the command lines):

```
# Optional, setup a separate virtualenv to keep everything clean
virtualenv ENV
source ENV/bin/activate
# Download dependencies, tensorflow will be the CPU version
pip install matplotlib tensorflow seaborn scikit-learn h5py jupyter
# Then you could start a notebook
jupyter notebook
```

For a GPU tensorflow, usually best to build yourself (out of scope)

- Now, let's setup a new workspace

# Google Colab / Jupyter Basic Usage

- <https://colab.research.google.com/notebook>
- Offers free jupyter-notebook-as-a-service in the cloud
  - Even offers free access to cloud-based GPUs
- Has all the packages we'll need for today pre-installed
- Demo
  - Basic jupyter usage

```
!ls # execute external commands
import os
?os # help at your fingertips
pi = 3.14159 # interpreter persists over cells
pi*2

def area(radius):
    return pi*radius**2

area(1)

# inline plotting
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-3.14, 3.14, 100)
y = np.sin(x)
plt.plot(x, y)
plt.show()
```



- Deep learning framework built by Google engineer François Chollet
- High-level interface built allowing eg Theano or Tensorflow as a backend
  - Has been accepted into mainline Tensorflow, so always accessible there
- Library written in python, user-friendly interface
- Easy to get started building networks
- Highly modular and easily expandable
  - Can drop down into the underlying library when complex/bespoke operations are needed
- Quickly build and train serious models

- Follow along either on the web-based service, or your own machine
- Lets pull in all the imports and definitions we'll need

```
import h5py
import matplotlib
# matplotlib.use("AGG") # To batch graphics
import matplotlib.pyplot as plt
import os
import seaborn as sns
from sklearn.model_selection import train_test_split
import sklearn
import numpy as np
import tensorflow as tf

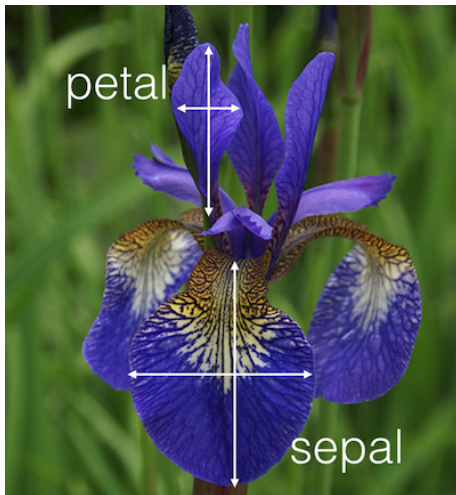
keras = tf.keras
Sequential = keras.Sequential
Activation = keras.layers.Activation
Dense = keras.layers.Dense
LeakyReLU = keras.layers.LeakyReLU
BatchNormalization = keras.layers.BatchNormalization
Reshape = keras.layers.Reshape
UpSampling2D = keras.layers.UpSampling2D
Dropout = keras.layers.Dropout
Conv2D = keras.layers.Conv2D
MaxPooling2D = keras.layers.MaxPooling2D
Flatten = keras.layers.Flatten
SGD = keras.optimizers.SGD
mnist = keras.datasets.mnist
```

# Overarching Idea of (Supervised) Machine Learning

- Framework for Machine Learning: given a set of data, and set of expected outputs, build a system which learns how to connect data to output
- Neural Network is one type, connect stacks of tensor operators with fixed linear and non-linear transformations
- Optimize transformation parameters so as to approximate expected outputs



# The iris dataset and a basic network with Keras



- The iris dataset is a classic classification task, first studied by Fisher in 1936.
- The goal is, given features measured from a particular iris, classify it into one of three species
  - Iris setosa, virginica, versicolor.
- The variables are: Sepal width and length, petal width and length (all in cm).

# Iris dataset

We begin by loading the iris dataset, helpfully available from the seaborn package, which also lets us create plots showing the correlations between the variables.

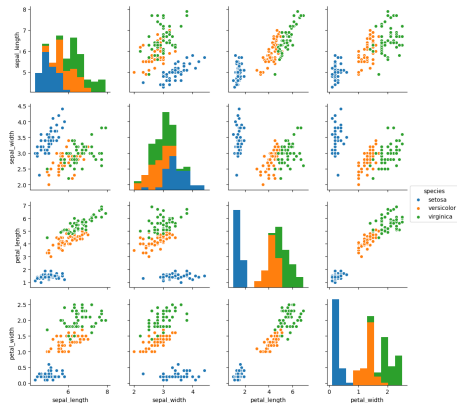
```
iris = sns.load_dataset("iris")  
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

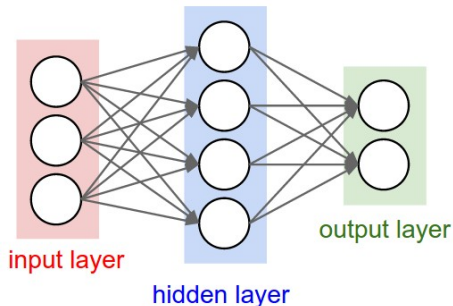
# Iris Variables

Lets view the basic variables we have. Setosa (blue) looks easily separable by the petal length and width, but versicolor and virginica are a little tricky.

```
plot = sns.pairplot(iris, hue="species")  
plot.savefig('iris.png');
```



# Neural Networks



- Example shown: input vector  $\vec{x}$ , goes through  $\vec{y}_{hidden} = W\vec{x} + \vec{b}$ , then  $\vec{y}_{output} = \sigma(\vec{y}_{hidden})$  ( $\sigma$  is some non-linear turn-on curve)
- I.e. hidden layer combines  $\vec{x}$  by some weights, then if the weighted sum passes a threshold  $\vec{b}$ , we turn on the output (with the  $\sigma(x) = 1/(1 + e^{-x})$  to gate the ops)
- Need to **train** the weight matrix  $W$  and the bias vector  $b$  and optimize a "loss" function that represents a distance from the target output

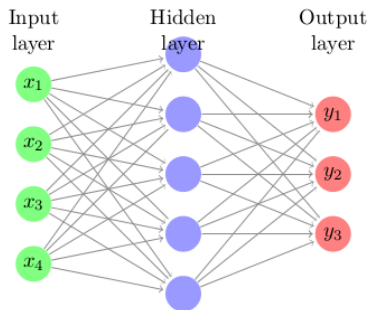
# Backpropagation

- There is a theorem which says that neural networks can well-approximate any (nice enough) function
- To do this, we start with a random network and "train" the network (set the weights)
- The algorithm to train neural networks is called **backpropagation**
- We start with the parameters set to arbitrary values, usually picked from e.g. unit gaussian
- We run a forward pass through the network and calculate the loss
- Using the chain rule, calculate derivatives backward from the loss to the higher layers
- Propagate changes based on the gradient  $\Delta w_i = -\eta \frac{\partial f}{\partial w_i}$

# Keras Networks

In order to classify the irises, we'll build a simple network in Keras.

- The basic network type in Keras is the `Sequential` model.
- The `Sequential` model builds a neural network by stacking layers
  - Keras also has a `Graph` model that allows arbitrary connections
- It builds up like lego, adding one layer on top of another and connecting between the layers
  - Keras comes with a menagerie of pre-built layers for you to use.
- Interface to/from the model with numpy arrays



# Model

- Our model will be a simple NN with a single hidden layer
- We start by building a Sequential model and add a Dense (fully-connected) layer, with sigmoid activation
- Dense: standard layer, all inputs connect to all outputs:  $\hat{y} = W\hat{x} + \hat{b}$ 
  - `keras.layers.Dense(output_dim)`
  - Can also set the initialization, add an activation layer inline, add regularizers inline, etc.
- Activation: essentially acts as a switch for a given node, turns output on/off based on threshold
  - `keras.layers.Activation( type )`
    - Where *type* might be:
  - *sigmoid*:  $f(x) = \frac{1}{1+e^{-x}}$
  - *tanh*:  $f(x) = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - *relu*:  $f(x) = \max(0, x)$ , 'rectified linear unit'
  - *softplus*:  $f(x) = \ln(1 + e^x)$ , smooth approx. to *relu*
  - *softmax*:  $f_k(x) = \frac{e^{-x_k}}{\sum_i e^{-x_i}}$  for the *k*'th output, as last layer of categorical distribution, represents a probability distribution over the outputs

# Build a model: Python code

```
# Build a model
model = Sequential()

model.add(Dense(128, input_shape=(4,)))
model.add(Activation('sigmoid'))
# model.add(Dense(128))
# model.add(Activation('sigmoid'))
model.add(Dense(3))
model.add(Activation('softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy',
              metrics=['accuracy'])
model.summary()
```

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 128)	640
activation_1 (Activation)	(None, 128)	0
dense_2 (Dense)	(None, 3)	387
activation_2 (Activation)	(None, 3)	0
Total params: 1,027		



# More on model building

- When =add=ing layers, keras takes care of input/output size details
  - Except for the input layer, which must be specified
- The final layer we make size 3 after a softmax activation
  - This will output the network probability for each of the potential iris classes as a numpy array (`nsamples, ( psetosa, pvirginica, pversicolor )`)
- We compile the model with an optimizer and loss function
  - The loss function will be minimized during the training phase
- We can give auxilliary metrics which will be calculated with the loss
- Keras automatically takes care of calculating derivatives through the network for the backprop phase
- We could be more explicit in creating the functions if we want more control over hyperparameters:

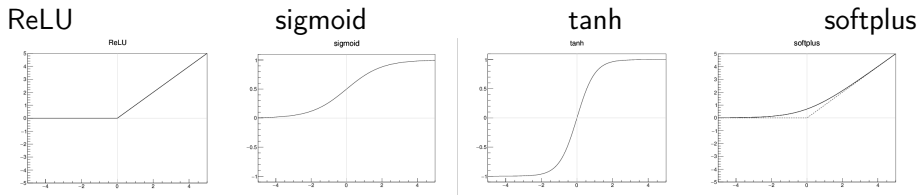
```
model.compile(loss=keras.losses.mean_squared_error,  
              optimizer=keras.optimizers.SGD(lr=0.0005, momentum=0.9, nesterov=True)
```

# More on model building

Here we used the adam optimizer which automatically updates the step sizes used for parameter optimization, with a categorical cross-entropy loss, which measures  $-\sum_i t_i \log p_i$  where  $t_i$  is 1 for the true label and  $p_i$  is the probability of the  $i$ th label assigned by the model. As the model assigns higher probability to the correct label, the cross-entropy goes to 0.

- Other options to consider:

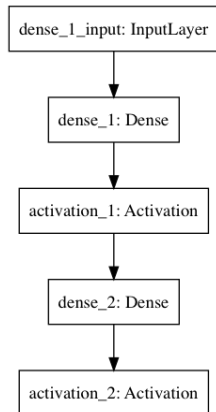
- Activation: *sigmoid*, *softmax*, *linear*, *tanh*, *relu*, ...
- Optimizer: *SGD*, *RMSprop*, *Adagrad*, *Adadelata*, *Adam*, ...
- Loss: *categorical\_crossentropy*, *binary\_crossentropy*, *mean\_squared\_error*, ...



# Model picture

If pydot is installed we can also output a picture of the network

```
keras.utils.plot_model(model, to_file='iris_model.png')  
'iris_model.png'
```



# Training Code

```
# Split the variables to train, and the target
variables = iris.values[:, :4]
species = iris.values[:, 4]

# One hot encode the species target
smmap = {'setosa' : 0, 'versicolor' : 1, 'virginica' : 2}
species_enc = np.eye(3)[list(smmap[s] for s in species)]

# To show we are simply passing numpy arrays of the data
print(variables[0], species[0], species_enc[0])

train_X, test_X, train_y, test_y = \
    train_test_split(variables, species_enc, train_size=0.8, random_state=0)
model.fit(train_X, train_y, epochs=15, batch_size=1, verbose=1)

[5.1 3.5 1.4 0.2] setosa [ 1.  0.  0.]
Epoch 1/15
120/120 [=====] - 0s - loss: 0.2873 - acc: 0.9500

...

Epoch 15/15
120/120 [=====] - 0s - loss: 0.1477 - acc: 0.9583
```

- Now we fit to the training data.
- We can set the number of epochs, batch\_size, and verbose'ity
  - Epochs: number of training passes through the complete dataset
  - Batch size: number of datapoints to consider together when updating the network
- We pass through the input data as a numpy array (nsamples, 4)
- We pass the output as (nsamples, 3) where for each sample one of the positions is 1, corresponding to the correct class.
- We use the `np.eye` identity matrix creator to help us transform the raw species information (which labels classes setosa, virginica, versicolor) to the expected format
  - Setosa = (1, 0, 0)
  - Versicolor = (0, 1, 0)
  - Virginica = (0, 0, 1)
- We fit the model to a labelled dataset simply by calling `fit` with the dataset `train_X` and the true labels `train_y`

# Evaluation

- After running the model, we can evaluate how well it works on the labelled *test* data we kept aside for *overfitting* evaluation purposes.
  - Overfitting is when the model fits to the training set in a way that doesn't generalize to unseen samples
  - One usually also has a separate *validation* set, use the *test* set on a single model, choose a model you like, then check the *hyperparameters* didn't cause bias by checking the *validation*

```
# The evaluation passes out the overall loss,  
# as well as any other metrics you included  
# when compiling the model  
loss, accuracy = model.evaluate(test_X, test_y, verbose=0)  
print("Loss={:.2f}\nAccuracy = {:.2f}".format(loss, accuracy))
```

```
Loss=0.11  
Accuracy = 0.97
```

# Prediction

- And we can ask the model to predict some unlabelled data
  - For illustration, we just use our test data, and compare the true label against the 'prediction'
  - In the output, I stack the true answers (first rows), and the prediction, which can basically be interpreted as the model's probability for each category (second rows)

```
pred_y = model.predict(test_X)
print(np.stack([test_y, pred_y], axis=1)[:10])
```

```
[[[ 0.00000000e+00  0.00000000e+00  1.00000000e+00]
  [ 2.63856982e-05  8.96630138e-02  9.10310626e-01]]

 [[ 0.00000000e+00  1.00000000e+00  0.00000000e+00]
  [ 1.57812089e-02  9.63519156e-01  2.06995625e-02]]

 [[ 1.00000000e+00  0.00000000e+00  0.00000000e+00]
  [ 9.96497989e-01  3.50204227e-03  1.25929889e-09]]

 [[ 0.00000000e+00  0.00000000e+00  1.00000000e+00]
  [ 4.74178378e-05  1.32592529e-01  8.67359996e-01]]

 [[ 1.00000000e+00  0.00000000e+00  0.00000000e+00]
```

# MNIST digit recognition and Convolutional Networks

- Another, more recent, classic classification task.
- Given a 28x28 image of a handwritten digit, can you train a classifier to recognize the numbers from 0 to 9?
- Keras has the ability to download the dataset and parse it into numpy arrays. We use `to_categorical` to one hot encode the true labels (which number did they write?) as for the irises

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

```
from keras.utils.np_utils import to_categorical
```

```
# or to_categorical = tf.keras.utils.np_utils.to_categorical
```

```
print(y_train[:4])
```

```
y_train_enc = np.eye(10)[y_train]
```

```
y_test_enc = to_categorical(y_test) # many ways to do the same thing
```

```
print(y_train_enc[:4])
```

```
[5 0 4 1]
```

```
[[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
```

```
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

```
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]
```

```
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

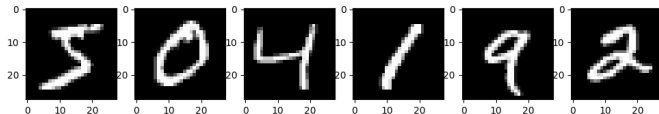


# Examples

- We can use `matplotlib.pyplot` to show a few example digits
- In *jupyter*, `matplotlib` results will show automatically, so you don't need to print it out (or resize it for that matter)

```
print(x_train.shape, y_train_enc.shape)
plt.clf()
for i in range(6):
    plt.subplot(1,6,i+1)
    plt.imshow(x_train[i], cmap='gray')

F = plt.gcf(); F.set_size_inches((14,2))
plt.savefig('mnist-examples.png'); 'mnist-examples.png'
```



# Simple Network

- We can start by simply trying a basic neural network as before.
- 'Flatten' takes the 2D input and concatenates the rows together to a 1D form suitable for passing to a 'Dense' layer.

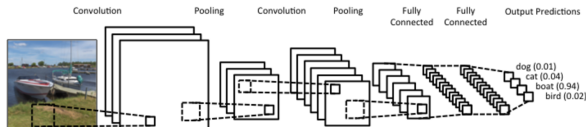
```
model = Sequential()  
model.add(Flatten(input_shape=(28,28)))  
model.add(Dense(128))  
model.add(Activation('sigmoid'))  
model.add(Dense(128))  
model.add(Activation('sigmoid'))  
model.add(Dense(10))  
model.add(Activation('softmax'))  
  
model.compile(optimizer='adam', loss='categorical_crossentropy',  
              metrics=['accuracy'])
```

- And fit and evaluate as we did before

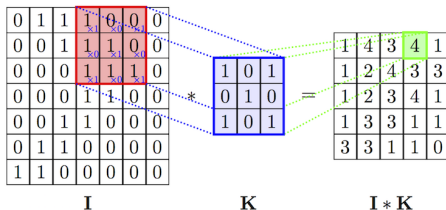
```
model.fit(x_train, y_train_enc, epochs=3, verbose=1)
loss, accuracy = model.evaluate(x_test, y_test_enc, verbose=0)
print("Loss={:.2f}\nAccuracy = {:.2f}".format(loss, accuracy))

Epoch 1/3
60000/60000 [=====] - 4s - loss: 0.5373 - acc: 0.8
Epoch 2/3
60000/60000 [=====] - 4s - loss: 0.3729 - acc: 0.8
Epoch 3/3
60000/60000 [=====] - 4s - loss: 0.3207 - acc: 0.9
Loss=0.30
Accuracy = 0.91
```

# A Convolutional Network



- One of the great advances in image classification in recent times
- We have some filter kernel  $K$  of size  $n \times m$  which we apply to every  $n \times m$  cell on the original image to create a new filtered image.
- It has been seen that applying these in multiple layers of a network can build up multiple levels of abstraction to classify higher-level features.
  - And, importantly, is trainable many, many layers deep



Reference: <http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/>

# Reshaping data for Keras

- Convolution of this type in Keras is provided by the Conv2D layer
- Conv2D requires passing an array of *width x height x channels*
  - Where channels might represent colors of an image
- We have black and white images so we'll just reshape it into the required form with a single channel.
- We plot the image just check show the shaping is correct

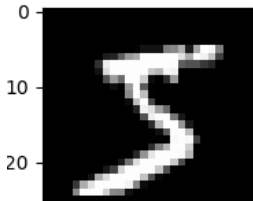
```
x_train_dense = x_train.reshape((len(x_train), 28,28,1))
```

```
x_test_dense = x_test.reshape((len(x_test), 28,28,1))
```

```
plt.clf()
```

```
plt.imshow(x_train_dense[0,:,:,:], cmap="gray")
```

```
F = plt.gcf().F.set_size_inches((2,2)); plt.savefig("testing.png"); "testi
```



# Building a Convolutional Neural Network in Keras

- Now, lets build a convolutional neural network!
- Generally, `Conv2D` will be stacked on top of each other with `MaxPooling2D` layers and learn edge detection at lower layers and higher level feature extraction in subsequent layers.
- But just to show how to use them in keras, we'll just create one convolution layer with 32 filters, then `Flatten` it into a 1D array and pass it into a `Dense` hidden layer before the output.
- We can set the `kernel_size` ( $m \times n$  size of the filter), and the number of filters used

# Building a Convolutional Neural Network in Keras

- We can set the `kernel_size` ( $m \times n$  size of the filter), and the number of filters used

```
model = Sequential()

model.add(Conv2D(32, kernel_size=(3,3), input_shape=(28,28,1)))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(128))
model.add(Activation('sigmoid'))
model.add(Dense(10))
model.add(Activation('softmax'))

model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

# Training

And train the model. This is already starting to get to the point where a GPU would be extremely helpful!

```
model.fit(x_train_dense, y_train_enc, epochs=4, verbose=1)
```

```
Epoch 1/4
```

```
60000/60000 [=====] - 65s - loss: 0.4544 - acc: 0.
```

```
Epoch 2/4
```

```
60000/60000 [=====] - 70s - loss: 0.1745 - acc: 0.
```

```
Epoch 3/4
```

```
60000/60000 [=====] - 68s - loss: 0.1369 - acc: 0.
```

```
Epoch 4/4
```

```
60000/60000 [=====] - 69s - loss: 0.1227 - acc: 0.
```

```
<keras.callbacks.History object at 0x11d742390>
```

```
loss, accuracy = model.evaluate(x_test_dense, y_test_enc, verbose=0)
```

```
print("Loss={:.3f}\nAccuracy = {:.3f}".format(loss, accuracy))
```

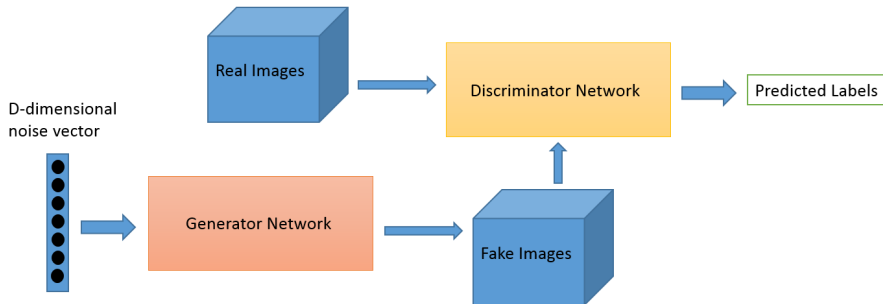
```
Loss=0.117
```

```
Accuracy = 0.964
```



# A Convolution GAN

- The idea is to train two adversarial networks,
  - One is trying to create images equivalent to the MNIST dataset
    - Given an input of noise, the *latent space*
  - The other trying to label the images as either from the dataset or fake
    - Fake = generated by the opposing dataset



- References:
  - For more on GANs and their uses: <https://arxiv.org/pdf/1701.00160.pdf>
  - Code based on: <https://github.com/jacobgil/keras-dcgan>
  - Some tricks for training GANs <https://github.com/soumith/ganhacks>

# Idea: Image generator network

- We start with the image generation network
- Essentially a image classifier in reverse.
- The top layer is for high-level feature inputs which we'll randomly set during the training.
- We then pass through Dense layers and then reshape into a  $7 \times 7 \times \text{channels}$  image-style layer.
- We Upsampling2D and pass through convolutional filters until the last layer which outputs a  $28 \times 28 \times 1$  image as expected of an MNIST greyscale image.
  - Essentially we're *adding* features as we go up, instead of *extracting* features as we go down
- BatchNormalization is a technique to improve the network stability by providing the next layer inputs with zero mean and unit variance

```
# Complete code for the generator model
nfeatures = 100

generate = Sequential()
generate.add(Dense(1024, input_dim=nfeatures))
generate.add(Activation('tanh'))
generate.add(Dense(128*7*7))
generate.add(BatchNormalization())
generate.add(Activation('tanh'))
generate.add(Reshape((7, 7, 128)))
generate.add(UpSampling2D(size=(2,2)))
generate.add(Conv2D(64, (5,5), padding='same'))
generate.add(Activation('tanh'))
generate.add(UpSampling2D(size=(2,2)))
generate.add(Conv2D(1, (5, 5), padding='same'))
generate.add(Activation('sigmoid'))
generate.compile(loss="binary_crossentropy", optimizer="SGD")
```

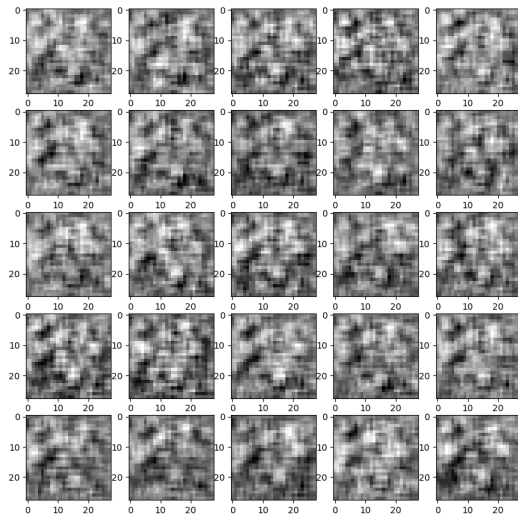
Now, just to check everything's put together properly, randomly pass some data through the network and check we get image outputs as expected.

```
nim = 25
pred = generate.predict(np.random.uniform(0, 1, (nim,nfeatures)))

plt.clf()
for i in range(nim):
    plt.subplot(np.sqrt(nim),np.sqrt(nim),i+1)
    plt.imshow(pred[i,:,:,:0], cmap='gray')

pred[0].shape, np.average(pred[0])
F = plt.gcf(); F.set_size_inches((10,10)); plt.savefig("genimg_no.pr
```

# Example images, pre-training



- Next, we create the discriminating network, with an image input
- As for classification, we have a convolutional layer attached to Dense layers.
- For the output, we now have a single sigmoid with interpretation:
  - 0: The network thinks its definitely a generated image
  - 1: The network thinks its definitely a real MNIST dataset image

# Discriminator

```
# Complete code for the discriminator network
discr = Sequential()
discr.add(Conv2D(64, (5,5), input_shape=(28,28,1), padding='same'))
discr.add(Activation('tanh'))
discr.add(MaxPooling2D((2,2)))
discr.add(Conv2D(128, (5,5)))
discr.add(Activation('tanh'))
discr.add(MaxPooling2D((2,2)))
discr.add(Dropout(0.5))
discr.add(Flatten())
discr.add(Dense(1024))
discr.add(Activation('tanh'))
discr.add(Dense(1))
discr.add(Activation('sigmoid'))
discr.compile(loss='binary_crossentropy',
              optimizer=SGD(lr=0.0005, momentum=0.9, nesterov=True))
```

# Test the discriminator

- Test the network with a few MNIST images and some random images.
- Since the network isn't trained we don't yet expect any differences in the output.

```
x_prepared = np.concatenate(  
    [x_train[:5, :, :].reshape(5, 28, 28, 1) / 256.,  
     np.random.uniform(0, 1, (5, 28, 28, 1))], axis=0)  
discr.predict(x_prepared)
```

0.53229088 0.53476292 0.53820759 0.5288614 0.52571678 0.57405



- Now we set up a network which will be used to train the generation network.
- Keras allows us to simply add the models we just created together into a Sequential like they were ordinary layers.
- So, we feed the generator output into the discriminator input and set up an optimizer which will try to drive the generator to produce MNIST-like images (i.e. to fool the discriminator).
- Keras allows us to turn layer training on and off through the "trainable" variable attached to a layer, so when we train the generator we can easily turn training for the discriminator off.

# Setup GAN

```
gen_discr = Sequential()  
gen_discr.add(generate)  
discr.trainable = False  
gen_discr.add(discr)  
gen_discr.compile(loss='binary_crossentropy',  
                  optimizer=SGD(lr=0.0005, momentum=0.9, nesterov=True),  
                  metrics=['accuracy'])  
discr.trainable = True
```

# Training the GAN

- Finally, we have the actual training
- Here, we setup the batches ourselves and alternate between training the discriminator and generator
  - `model.train_on_batch`
  - This was previously put together by Keras itself
- We start by taking a batch of MNIST images (labeled 1), and generator images (labeled 0) and run a training batch on the discriminator network
- Then, we turn off training off the discriminator and run training on the generator+discriminator network with random high-level feature inputs to the generator
- We try to drive all the outputs to 1, i.e. train the generator to more MNIST-like images (as according to the discriminator network)
- Last remark: we are saving the networks after each epoch with `model.save`
  - Load with `keras.models.load_model`

# Training the GAN

```
batch_size = 100
n_epochs = 10
print_every_nth_epoch = 50
x_tru_all = x_train.reshape(len(x_train), 28, 28, 1) / 256.

zeros = np.array([0]*batch_size)
ones = np.array([1]*batch_size)
oneszeros = np.array([1]*batch_size + [0]*batch_size)

losses_d = []
losses_g = []
for epoch in range(n_epochs):
    print ("Epoch", epoch)
    discr.save("/discr-"+str(epoch))
    generate.save("/generate-"+str(epoch))
    for i in range(0, len(x_train), batch_size):
        x_gen = generate.predict(np.random.uniform(0, 1, (batch_size, nfeatures)))
        x_tru = x_tru_all[i:i+batch_size]
        # Train the discriminator by taking example MNIST and generator-produced images
        discr.trainable=True
        loss_d = discr.train_on_batch(np.concatenate([x_tru, x_gen], axis=0), oneszeros)
        # Now, turn discriminator training off, so we can train the generator
        discr.trainable=False
        loss_g = gen_discr.train_on_batch(np.random.uniform(0, 1, (batch_size, nfeatures)), ones)
    if i % (print_every_nth_epoch*batch_size) == 0:
        print (i / batch_size, "discr", loss_d, "--", "gen", loss_g[0], "( acc.", loss_g[1], ")")
    losses_g.append(loss_g)
    losses_d.append(loss_d)
```

# Checking results

- Lets see how we did, lets just generate a bunch of images

```
nim = 25
```

```
pred = generate.predict(np.random.uniform(0, 1, (nim,nfeatures)))
```

```
plt.clf()
```

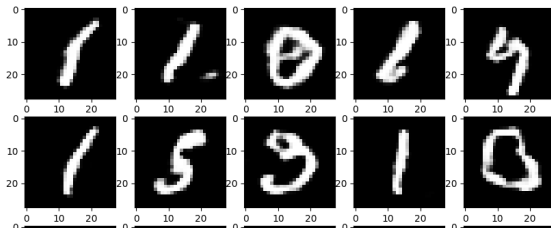
```
for i in range(nim):
```

```
    plt.subplot(np.sqrt(nim),np.sqrt(nim),i+1)
```

```
    plt.imshow(pred[i,:,:,:0], cmap='gray')
```

```
pred[0].shape, np.average(pred[0])
```

```
F = plt.gcf(); F.set_size_inches((10,10)); plt.savefig("genimg_after.png"); "genimg"
```



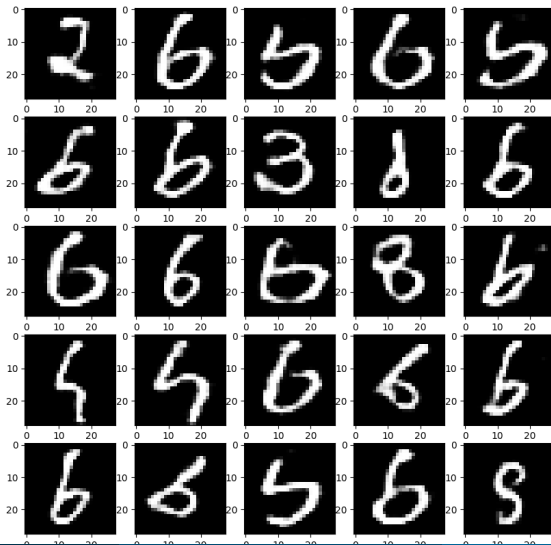
- Whats the "best" being produced by the GAN?
- Only accept above 0.9 from discriminator

```
nim = 25
target = .9

plt.clf()
for i in range(nim):
    best = 0; pred=None
    while best < target:
        pred = generate.predict(np.random.uniform(0, 1, (1,nfeatures)))
        best = discr.predict(pred)[0][0]
    plt.subplot(np.sqrt(nim),np.sqrt(nim),i+1)
    plt.imshow(pred[0,:,:], cmap='gray')

pred[0].shape, np.average(pred[0])
F = plt.gcf(); F.set_size_inches((10,10)); plt.savefig("genimg40_best.9.png"); "genimg40_best.9.png"
```

# Good Images



- Whats the "worst" being produced by the GAN?
- Only accept below 0.1 from discriminator

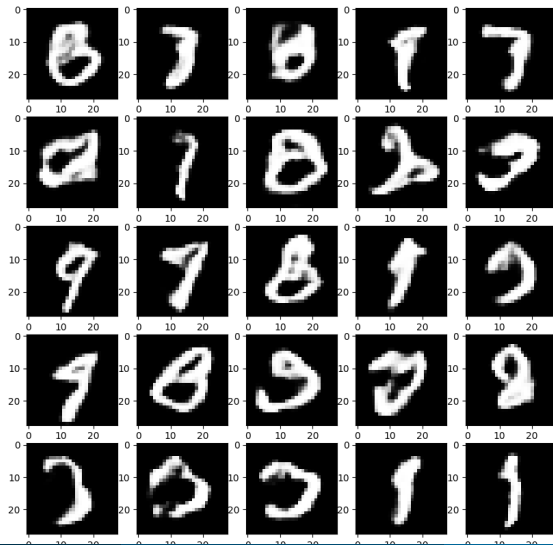
```
nim = 25
target = .1

plt.clf()
for i in range(nim):
    best = 1; pred=None
    while best > target:
        pred = generate.predict(np.random.uniform(0, 1, (1,nfeatures)))
        best = discr.predict(pred)[0][0]
    plt.subplot(np.sqrt(nim),np.sqrt(nim),i+1)
    plt.imshow(pred[0,:,:], cmap='gray')

pred[0].shape, np.average(pred[0])
F = plt.gcf(); F.set_size_inches((10,10)); plt.savefig("genimg40_worst.1.png"); "genimg40_worst.1.png"
```



# Bad images



- Try different networks, what works well, what fails badly?
- Add another set of inputs hot-one encoding the number you want to generate,
  - The discriminator will need to say which number it believes its seeing as well as how likely it is to be real
  - The generator will need to train with the number output as a loss also

# Train requiring GAN to also output the correct number

```
nfeatures = 100

generate = Sequential()
generate.add(Dense(1024, input_dim=(nfeatures + 10)))
generate.add(Activation('tanh'))
generate.add(Dense(128*7*7))
generate.add(BatchNormalization())
generate.add(Activation('tanh'))
generate.add(Reshape((7, 7, 128)))
generate.add(UpSampling2D(size=(2,2)))
generate.add(Conv2D(64, (5,5), padding='same'))
generate.add(Activation('tanh'))
generate.add(UpSampling2D(size=(2,2)))
generate.add(Conv2D(1, (5, 5), padding='same'))
generate.add(Activation('sigmoid'))
generate.compile(loss="binary_crossentropy", optimizer="SGD")
```

# Create the Discriminator

```
discr = Sequential()
discr.add(Conv2D(128, (5,5), input_shape=(28,28,1), padding='same'))
discr.add(Activation('relu'))
discr.add(MaxPooling2D((2,2)))
discr.add(Conv2D(256, (5,5)))
discr.add(Activation('relu'))
discr.add(MaxPooling2D((2,2)))
discr.add(Dropout(0.5))
discr.add(Flatten())
discr.add(Dense(1024))
discr.add(Activation('tanh'))
discr.add(Dense(11)) # 1 for real or fake, then 10 for which number
discr.add(Activation('sigmoid'))
discr.compile(loss='categorical_crossentropy', optimizer=SGD(lr=0.0005, momentum=0.9, nesterov=True),
              metrics=['accuracy'])
```

# Create the combined network

```
gen_discr = Sequential()
gen_discr.add(generate)
discr.trainable = False
gen_discr.add(discr)
gen_discr.compile(loss='categorical_crossentropy', optimizer=SGD(lr=0.0005, momentum=0.9, nesterov=True),
                  # optimizer='adam',
                  metrics=['accuracy'])
discr.trainable = True

batch_size = 100
n_epochs = 50
print_every_nth_epoch = 50
x_tru_all = x_train.reshape(len(x_train), 28, 28, 1) / 256.

zeros = np.array([0]*batch_size)
ones = np.array([1]*batch_size)
oneszeros = np.array([1]*batch_size + [0]*batch_size)
```

# Pre-train the discriminator on the (untrained) generator output and real MNIST

```
# pre train the gan to be able to distinguish numbers
pre_losses_d = []
for epoch in range(5):
    print ("Epoch", epoch)
    for i in range(0, len(x_train), batch_size):
        one_hot_gen = np.eye(10)[np.random.random_integers(0, 9, size=(batch_size,))]
        x_inp = np.concatenate([np.random.uniform(0, 1, (batch_size, nfeatures)), one_hot_gen], axis=1)
        x_gen = generate.predict(x_inp)
        x_tru = x_tru_all[i:i+batch_size]
        y_tru = y_train_enc[i:i+batch_size]
        discr.trainable = True
        for_d_tru = np.concatenate([np.zeros((batch_size,1)), y_tru], axis=1)
        for_d_gen = np.concatenate([np.ones((batch_size,1)), np.zeros((batch_size,10))], axis=1)
        loss_d = discr.train_on_batch(np.concatenate([x_tru, x_gen], axis=0),
                                     np.concatenate([for_d_tru, for_d_gen], axis=0))
        if i % (print_every_nth_epoch*batch_size) == 0:
            print (i / batch_size, "discr", loss_d)
        pre_losses_d.append(loss_d)

loss, accuracy = discr.evaluate(x_test_dense,
                               np.concatenate([np.zeros((len(y_test_enc),1)), y_test_enc], axis=1), verbose=0)
print("Loss={:.3f}\nAccuracy = {:.3f}".format(loss, accuracy))
```

# Train the generator and discriminator together

```
losses_d = []
losses_g = []
for epoch in range(n_epochs):
    print ("Epoch", epoch)
    discr.save("discr-num-"+str(epoch))
    generate.save("generate-num-"+str(epoch))
    for i in range(0, len(x_train), batch_size):
        one_hot_gen = np.eye(10)[np.random.random_integers(0, 9, size=(batch_size,))]
        x_inp = np.concatenate([np.random.uniform(0, 1, (batch_size, nfeatures)), one_hot_gen], axis=1)
        x_gen = generate.predict(x_inp)
        x_tru = x_tru_all[i:i+batch_size]
        y_tru = y_train_enc[i:i+batch_size]
        discr.trainable = True
        for_d_tru = np.concatenate([np.zeros((batch_size,1)), y_tru], axis=1)
        for_d_gen = np.concatenate([np.ones((batch_size,1)), np.zeros((batch_size,10))], axis=1)
        loss_d = discr.train_on_batch(np.concatenate([x_tru, x_gen], axis=0),
                                      np.concatenate([for_d_tru, for_d_gen], axis=0))
        discr.trainable=False
        for_g = np.concatenate([np.zeros((batch_size,1)), one_hot_gen], axis=1)
        new_inp_g = np.concatenate([np.random.uniform(0, 1, (batch_size, nfeatures)), one_hot_gen], axis=1)
        loss_g = gen_discr.train_on_batch(new_inp_g, for_g)
        if i % (print_every_nth_epoch*batch_size) == 0:
            print (i / batch_size, "discr", loss_d, "--", "gen", loss_g[0], "( acc.", loss_g[1], ")")
        losses_g.append(loss_g)
        losses_d.append(loss_d)

print ("done")
```

# Check the output of the labelled GAN

```
# generate = tf.keras.models.load_model('generate-num-41')

nim = 25
numb = 1
pred = generate.predict(np.concatenate([np.random.uniform(0, 1, (nim,nfeatures)), np.eye(10)[[numb,]*nim] ], axis=0))

plt.clf()
for i in range(nim):
    plt.subplot(np.sqrt(nim),np.sqrt(nim),i+1)
    plt.imshow(pred[i,:,:,:0], cmap='gray')

pred[0].shape, np.average(pred[0])
F = plt.gcf(); F.set_size_inches((10,10)); plt.savefig("gen-num-img_after-%d.png" % numb); "gen-num-img_after-%d.png"
```



# Some examples from labelled GAN

