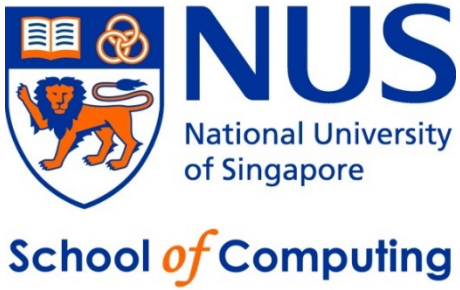
CS3237

Lecture 4

Some Deep Learning Techniques

colintan@nus.edu.sg



Motivation

- **In Lecture 3 we looked at neural networks:**
  - Unsupervised and supervised techniques.
  - Selection of hyperparameters.
  - Overfitting.
- **The neural networks of the previous lecture have an issue:**
  - The uniform, fully connected layers mean that there are many many parameters to train.
    - ✓**Compounded by complex data requiring more complex layers.**
  - "Curse of dimensionality" means that we need exponentially more data to train these parameters – and data is hard to get.
    - ✓**We have a tough tradeoff between the learning ability of the neural network, and overfitting.**

Motivation

- **Deep learning neural networks are designed to address these constraints:**
  - Consist of many layers, hence "deep".
  - Layers are not uniform, and can have different functions. E.g. act as filters, feature extractors, etc.
  - These transform and simplify the data so that less dense networks can be used to learn them.

NUS National University of Singapore
School of Computing

Motivation

- **Architectures we will look at:**
  - Long-Short Term Memory (LSTM)
  - Autoencoders (AEs)
  - Generative Adversarial Networks (GANs)
  - Convolutional Neural Networks (CNNs)

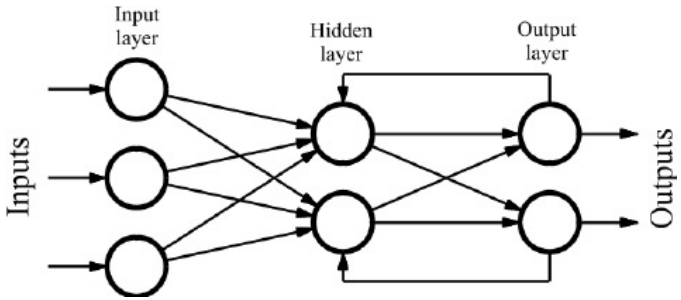
NUS National University of Singapore
School of Computing

SWS3009 Embedded Systems and Deep Learning.

LONG-SHORT-TERM MEMORIES (LSTM)

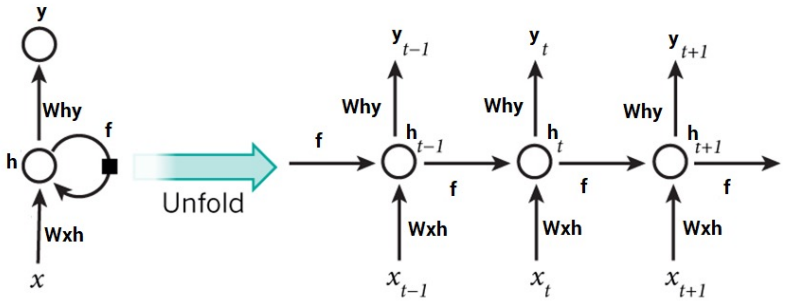
Deep Learning
Recurrent Neural Networks

- **Key issue with MLPs:**
  - When we present training data we assume that all the samples are independent.
  - This is often WRONG:
    - ✓**E.g. the word "often" above usually follows "is" or "are". Likewise "is" or "are" often follows "this". THEY ARE NOT INDEPENDENT!**
  - Solution:
    - ✓**Use past (and sometimes future) information to learn with current data.**
- **In RNNs we feed the output of the MLP back to the hidden layer (and maybe backwards to previous hidden layers)**



Deep Learning
Recurrent Neural Networks

- **Training is carried out using standard gradient descent that we just talked about.**
- **If we "unroll" an RNN in time, we get:**



- **We see that learning each new piece of data is now affected by past pieces.**

Deep Learning
Recurrent Neural Networks

- **Recurrent neural networks are good at learning from patterns in past data:**
  - Predicting next word in a sentence.
  - Predicting stock performance based on past data.
  - Trajectory prediction for a robot (e.g. learning how to pick up an object)
- **Standard RNNs have a problem:**
  - Take an RNN node and current input x_t. The output at time t is given by:

$$h_t = \sigma(Ux_t + Vh_{t-1})$$

Here U and V are weights for the input and previous output respectively.

- We can expand h_{t-1} giving us:

$$h_t = \sigma(Ux_t + V(\sigma(Ux_{t-1} + V(h_{t-2}))))$$

Deep Learning
Recurrent Neural Networks

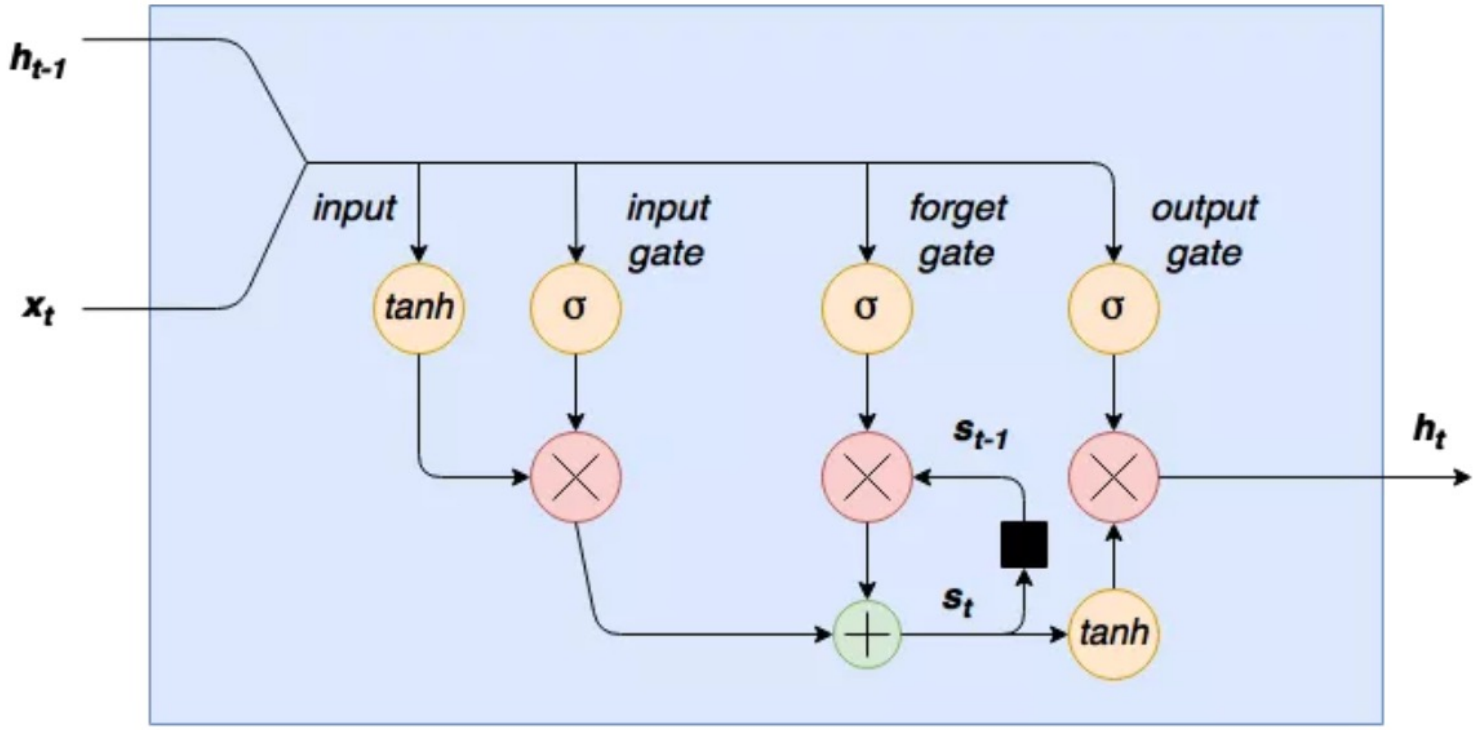
- **We can continue expanding this further. But what we want to do is to take the gradient $\frac{\delta E_t}{\delta U}$ to do our gradient descent:**

$$\frac{\delta E_t}{\delta U} = \frac{\delta E_t}{\delta out_t} \frac{\delta h_t}{\delta h_{t-1}} \frac{\delta h_{t-1}}{\delta h_{t-2}} \cdots \frac{\delta h_1}{\delta U}$$

- **As the gradient of a sigmoid is always small (≤0.25), this series of products will decay to 0 very quickly:**
  - This means that over a longer time series, no adjustments will be made to *U*. The network stops learning!
- **To solve this problem we introduce a special RNN called a "Long Short Term Memory" or LSTM Neural Network.**

LSTM Cell

- **The diagram below shows a typical LSTM cell:**



LSTM cell diagram

LSTM Cell
Input Stage

- **The cell receives two inputs: x_t, which is the current input, and h_{t-1} which is the previous output. Both are multiplied by weights, and put through a *tanh* function to squeeze it between -1 and 1:**

$$in_t = tanh(b_t^{in} + x_t U_t^{in} + h_{t-1} V_t^{in})$$

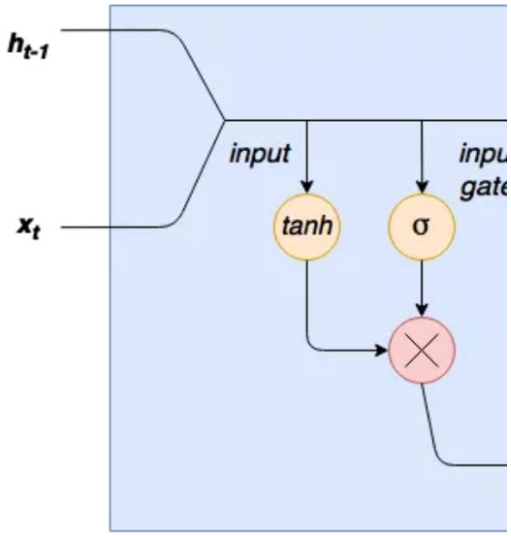
- **There is an input gate that selects which inputs will affect the output:**

$$g_t^{in} = \sigma(x_t U_t^{gin} + h_{t-1} V_t^{gin})$$

- **Finally the output of the input stage is given by:**

$$o_t^{in} = in_t \odot g_t^{in}$$

Where $\odot$ is the element-wise multiply operation.



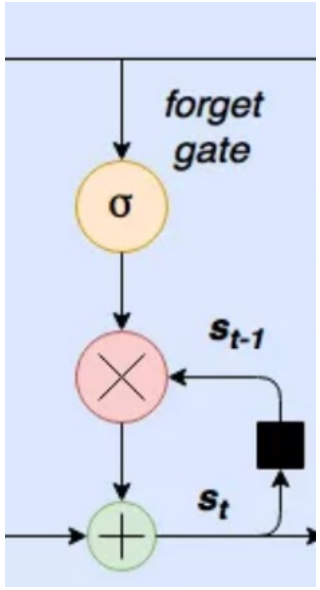
LSTM Cell
Forget Stage

- **In this stage we have a state variable s_t that maintains recurrence information, and a "forget" gate that decides what is remembered:**

$$g_t^{forget} = \sigma\left(b_t^{forget} + x_t U_t^{forget} + h_{t-1} V_t^{forget}\right)$$

- **We again do a element-wise multiply to control which states are remembered (g^{forget} =1) and which are forgotten (g^{forget}=0). We add this state to the output of the input stage (confusing, I know):**

$$s_t = g_t^{forget} \odot s_{t-1} + o_t^{in}$$



LSTM Cell
Output Stage

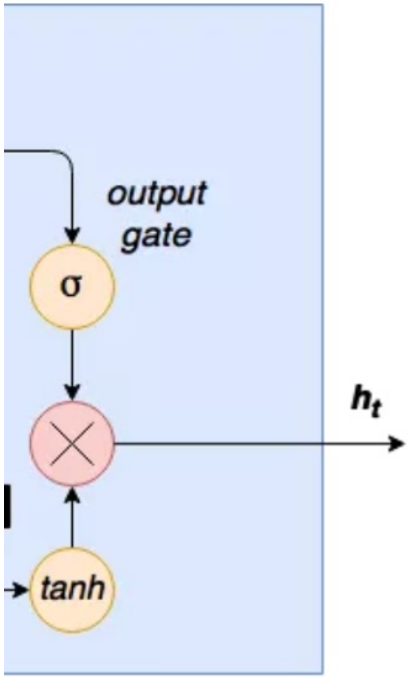
- **Our LSTM cell output is actually just based on the state variable s_t, but once again controlled through a gate called the "output gate" that decides what output to pass and what to suppress (whether $g^{output}=1$ or $g^{output}=0$)**

$$g_t^{output} = \sigma(b_t^{output} + x_t U_t^{output} + h_{t-1} V_t^{output})$$

- **Now we can get the output of the LSTM cell:**

$$h_t = tanh(s_t) \odot g_t^{output}$$

- **The U and V weights at every gate (and the input) are trained using gradient descent.**

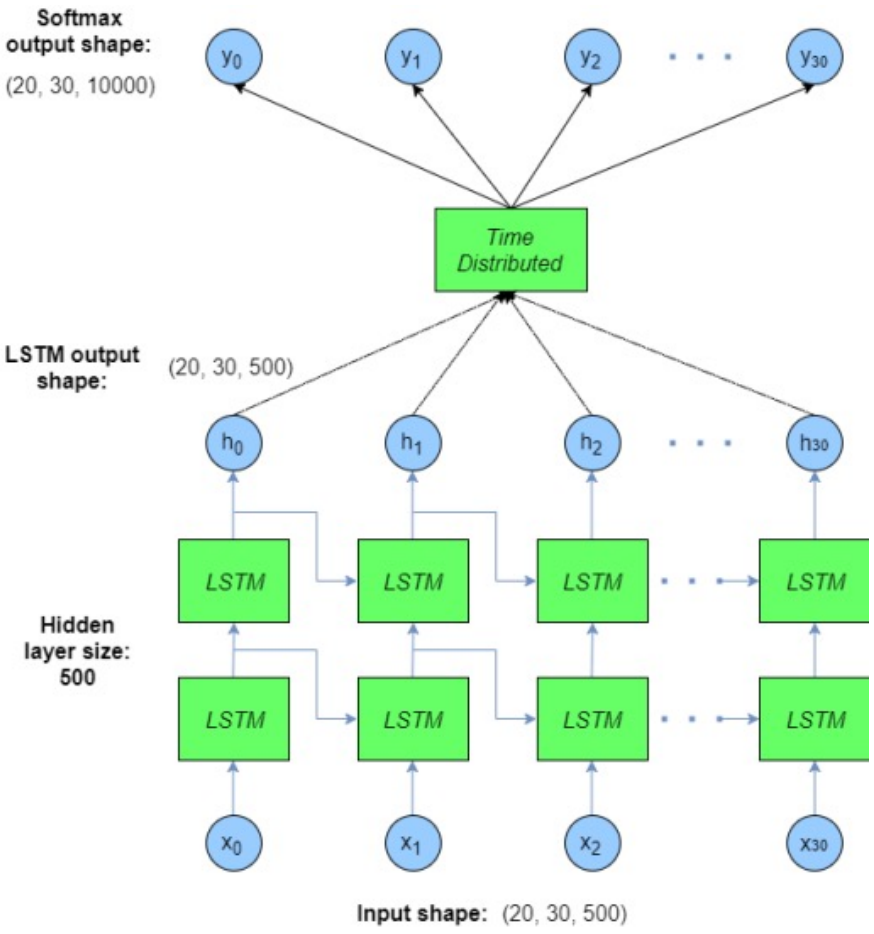


LSTM Hidden Layer Size

- **Each "gate" appears to be a single node, but they are actually a collection of nodes.**
  - The number of nodes in each gate is called the "hidden layer size" of the LSTM.
- **If we have *m* hidden nodes in the LSTM:**
  - Given an n dimension input, our matrix U has a dimension of $n * m$
  - Given a p dimension output, our matrix V has a dimension of $m * p$, and we actually have p $\boldsymbol{\sigma}(.)$ functions, not a single function.

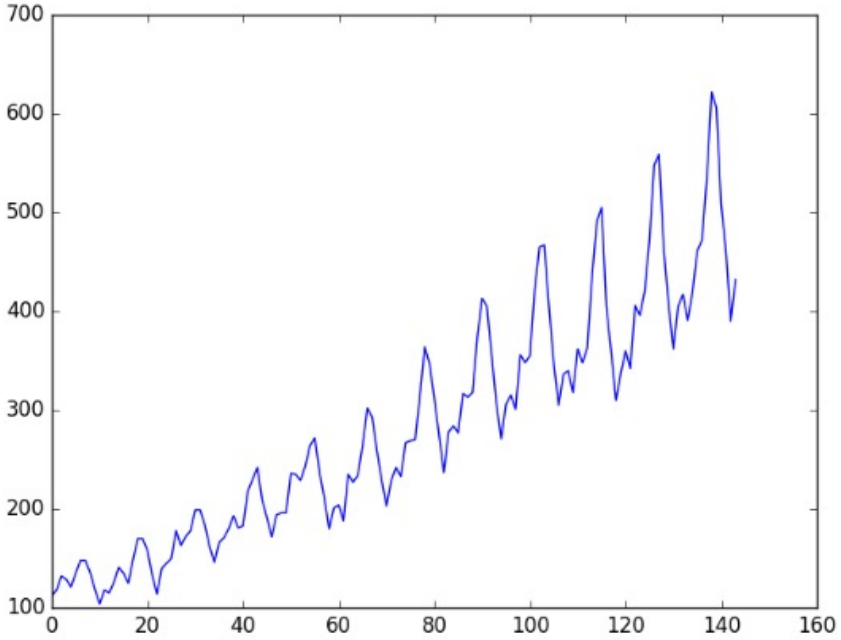
Using the LSTM Cells

- **The diagram below shows one way we can use an LSTM cell:**
  - Each cell produces one time step, and feeds the next cell, together with the input for that time step.
  - We can stack LSTMs to increase learning power, and connect Dense layers to learn patterns, etc. from the time-stepped data.



Using LSTMs in Keras

- **To see how to use LSTMs in Keras, we will use LSTMs to model the growth in airline passenger traffic from January 1949 to December 1960, shown below:**
  - The idea is to learn the data to make predictions about passenger traffic in the next month.



Plot of the Airline Passengers Dataset

```
"Month","Passengers"
"1949-01",112
"1949-02",118
"1949-03",132
"1949-04",129
"1949-05",121
"1949-06",135
"1949-07",148
"1949-08",148
"1949-09",136
"1949-10",119
```

NUS National University of Singapore
School of Computing

LSTM Hands On

Load up lstm.ipynb in Jupyter

Using LSTMs in Keras
Imports

- **Additional explanation for the Python notebook.**
  - We begin firstly with all the imports:

~/Dropbox/Modules/Summer Programme/2019/Practic

```
import numpy
import pandas
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
```

  - Here we bring in the LSTM layer for us to build our LSTM network, as well as a Dense layer that we will use to predict growth of airline passenger traffic based on past data.
  - We also bring in a MinMaxScaler that will scale our data into the [0, 1] range.

Using LSTMs in Keras
Training Hyperparameters

- **Now we create some constants, and seed the random number generator so that we get reproducible random numbers:**

```
batch_size = 1
look_back = 5
skip = 1
hidden_size = 128
num_epochs = 500
TRAIN_PERCENT = 0.67

numpy.random.seed(7)
```

- **The constants are:**
  - Batch_size = # of pieces of data presented each time for training, look_back = the time span we are learning from (i.e. current time and previous 5 time periods), skip = # of time steps to advance each time, hidden_size = # of neurons in each part of the LSTM. The rest are self-explanatory.

Using LSTMs in Keras
Creating the Dataset

- **We now load the data using Pandas, scale the data to between 0 and 1, then partition the data into training and testing data. We also make a "create_dataset" function.**

```
dataframe = pandas.read_csv('airline-passengers.csv', usecols=[1], engine='python')
dataset = dataframe.values
dataset=dataset.astype('float32')

scaler=MinMaxScaler(feature_range=(0,1))
dataset=scaler.fit_transform(dataset)

train_size = int(len(dataset) * TRAIN_PERCENT)
test_size = len(dataset) - train_size
train, test  = dataset[0:train_size], dataset[train_size:len(dataset)]

def create_dataset(dataset, look_back=1):
    dataX, dataY=[], []

    for i in range(len(dataset) - look_back - 1):
        a=dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return numpy.array(dataX), numpy.array(dataY)
```

Using LSTMs in Keras
Creating the Dataset

- **The "create_data" function creates a sliding window of data, depending on the value of look_back. If we use a look_back of 1 on the data series "1, 7, 3, 5, 2, 15, 12, 6, 3", then we get a table like:**

dataX	dataY
1	7
7	3
3	5
5	2
2	15
15	12
12	6
6	3

- **So dataY is always one step after the current position in dataX.**

Using LSTMs in Keras
Creating the Dataset

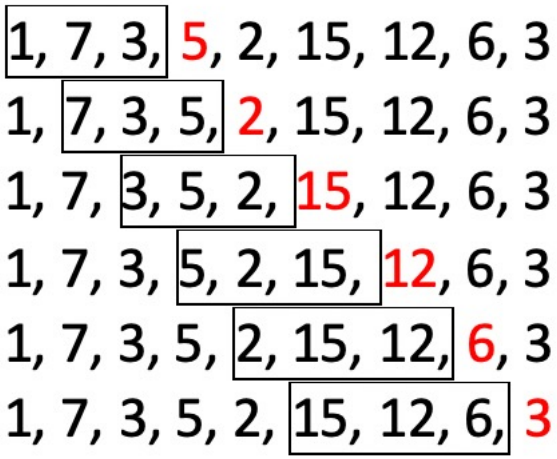
- **If we use a look_back of 3, we get:**

dataX	dataY
1, 7, 3	5
7, 3, 5	2
3, 5, 2	15
5, 2, 15	12
2, 15, 12	6
15, 12, 6	3

- **Notice that dataX is now a vector of 3 values, while dataY is the next value.**

Using LSTMs in Keras
Creating the Dataset

- **The diagram below gives a clearer picture:**



- **The window shows the current X values being considered, while the red values are the next value, which are to be predicted.**

Using LSTMs in Keras
Reshaping the Datasets

- **Since our LSTM is using just a single input (passenger data), our LSTM will have a single input, each input consisting of "look_back" entries. Our training data must be reformatted to match this:**

```
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)

trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
```

Using LSTMs in Keras
Building the LSTM

- **Now we are finally able to build our LSTM. We will use the LSTM layer in Keras to do this. We also need a dense layer to take the LSTM outputs and produce a single output – our prediction of passenger volumes. Then we add an Adam optimizer and call fit to train the LSTM:**

```
model = Sequential()
model.add(LSTM(hidden_size, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=num_epochs, batch_size=batch_size, verbose=2)
```

- **Each gate of the LSTM it itself a neural network, each with "hidden_size" nodes. It expects inputs to be of the shape (1, look_back):**
  - This means it expects a single variable input, each consisting of "look_back" entries.

Using LSTMs in Keras Evaluating

- **Finally we call predict to predict the next value, and use reverse_transform to map back from the [0, 1] range to the range of our original data. We then compare against the training and test data to see the root mean squared error (RMSE):**

```
trainPredict=model.predict(trainX)
testPredict = model.predict(testX)

trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])

trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
print('Train Score: %.2f RMSE' % (trainScore))

testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:, 0]))
print('Test Score: %.2f RMSE' % (testScore))
```

Using LSTMs in Keras Evaluating

- **When run you will see:**

```
 - 0s - loss: 0.0016
Epoch 496/500
 - 0s - loss: 0.0016
Epoch 497/500
 - 0s - loss: 0.0015
Epoch 498/500
 - 0s - loss: 0.0017
Epoch 499/500
 - 0s - loss: 0.0016
Epoch 500/500
 - 0s - loss: 0.0015
Train Score: 20.06 RMSE
Test Score: 53.28 RMSE
```

Using LSTMs in Keras
Swapping Dimensions

- **One of LSTM's strengths is its ability to remember across training batches. We will now see how to do this. Before we start:**
  - It is more intuitive to think of look_back time steps of a single input, rather than single inputs of look_back time steps. So locate these lines:

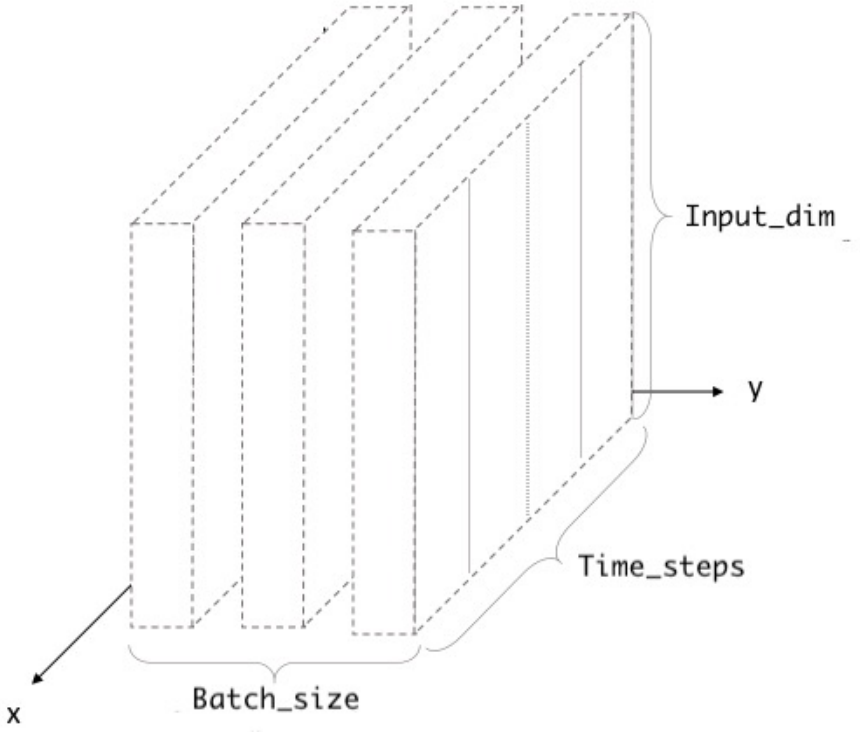
```
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
```

  - And change them to:

```
trainX = numpy.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = numpy.reshape(testX, (testX.shape[0], testX.shape[1], 1))
```

Using LSTMs in Keras
Why swap dimensions?

- **LSTM expects the third dimension t be the input features, NOT the look back.**
  - In our original code we had swappe this around so that the third dimensic was the look back, and the second dimension was the input features.
  - We swapped this back the right wa in our second version of the code.



Using LSTMs in Keras
Using Memory

- **To use memory we need to organize the data into batches, and to turn on the 'stateful' flag in LSTM. So change:**

```
model = Sequential()
model.add(LSTM(hidden_size, input_shape=(1, look_back)))
```

- **To:**

```
model = Sequential()
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1),
stateful = True))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
```

- **Note that we have also flipped the input from 1, look_back to look_back, 1, and reduced the number of hidden nodes in the LSTM.**

Using LSTMs in Keras
Using Memory

- **We also need to change the model.fit call to run one epoch at the time, because we need to call reset_states(), so that the LSTM knows it is getting a fresh batch of inputs.**
  - Otherwise the LSTM will wrongly tie the end of the previous epoch with the start of the next epoch.
- **Change:**

```
model.fit(trainX, trainY, epochs=num_epochs, batch_size=batch_size, verbose=2)
```

- **To:**

```
for i in range(num_epochs):
    print("Iter: %d of %d" % (i, iters))
    model.fit(trainX, trainY, epochs=1, batch_size = batch_size, verbose = 2, shuffle = False)
    model.reset_states()
```

- **Note that you MUST turn off shuffling, as we need to present the data sequentially as they occur for the LSTM to learn across them properly.**

Using LSTMs in Keras
Using Memory

- **When we run our new lstm1.py program, we see:**

```
Epoch 1/1
 - 0s - loss: 0.0013
Iter: 499 of 500
Epoch 1/1
 - 0s - loss: 0.0013
Train Score: 19.50 RMSE
Test Score: 48.75 RMSE
```

- **We see that the results have improved from 20.06 RMSE and 53.28 RSME for inside and outside testing respectively, to 19.50 and 48.75.**
- **In the next lab we will look at how to stack LSTMs to get better results.**

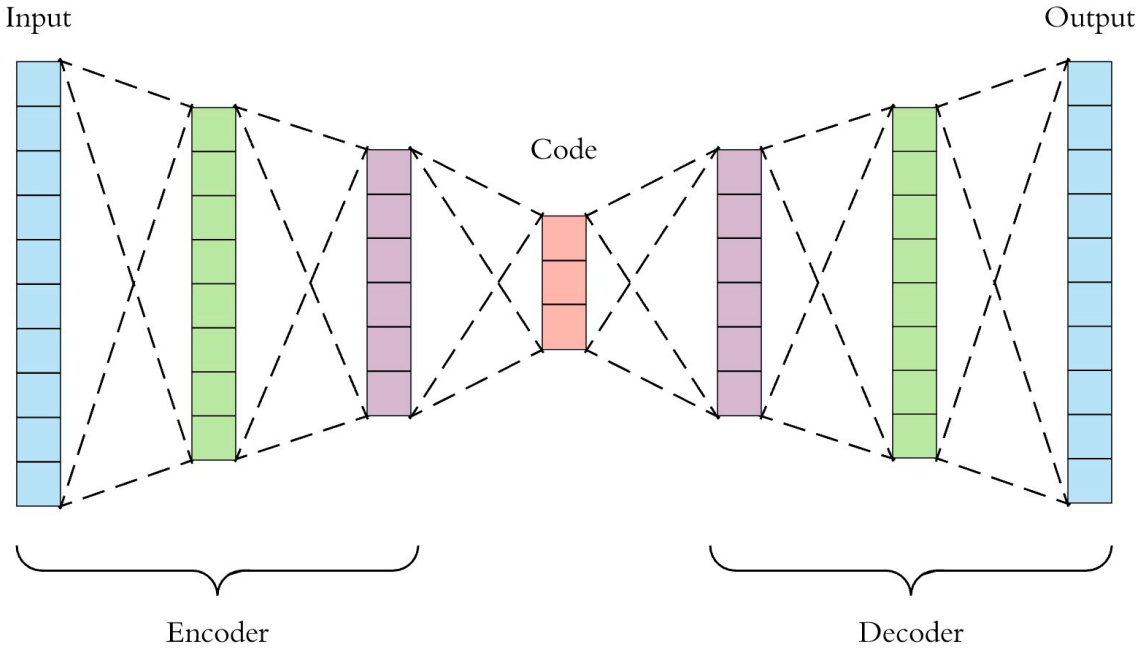
SWS3009 Embedded Systems and Deep Learning.

AUTOENCODERS

Autoencoders
What Are They?

- **We will now look at another class of deep-learning networks called "autoencoders".**
  - Idea: We train a neural network to take an input x, and generate the same output x. I.e. we will build a deep learning network that does:

$$f(x) \approx x$$



Autoencoders
Motivation

- **Why would we do this? Two reasons:**
  - We can achieve "compression":
    - ✓**If the "code layer" is substantially smaller than the input, we can feed in the input and use the resultant "code" as a compressed version of the input.**
    - ✓**We can feed this code into the decoder network to recover the original.**
    - ✓**However AEs are very bad at this: the recovered data will be highly "lossy", and the AE can only reconstruct data of the type it was trained on.**
  - We can detect anomalies:
    - ✓**When the AE sees "typical" data it was trained on, it can reconstruct the input with minimal error.**
    - ✓**When the data becomes atypical, reconstruction error rises. We can flag anomalies when this error exceeds a threshold.**

A Simple Autoencoder

- **We are going to build a simple autoencoder to learn a simple function:**

$$f(x) = 2x + \varepsilon$$

  - **This is a simple linear function but with white noise ε.**
- **We will test it against two pieces of data:**
  - **A noisier version of *f(x)***
  - **An abnormal version of *f(x)* that does:**

$$f(x) = 1.5x + \varepsilon$$

Autoencoder Hands On

Load up autoencoder.ipynb in Jupyter

A Simple Autoencoder Imports

- **We begin with the imports:**

```
from keras.layers import Dense, Input
from keras.models import Model
from keras.optimizers import SGD
from sklearn.metrics import mean_squared_error
import numpy as np

# For generating our fake data
import random
```

A Simple Autoencoder
Training Hyperparameters

- **We now create some constants to configure our dummy data, AE, etc.**

```
# Number of epochs to train
EPOCHS = 200

# Number of fake datapoints to create

DATA_SIZE = 10000
DATA_NOISE = 0.1
VAL_SIZE = int(0.2 * DATA_SIZE)

# Test Noise level
TEST_NOISE = 0.3
```

- **Here our training data will have 10,000 samples. A white noise of between -0.1 and 0.1 will be added. The data itself ranges from -1 to 1.**
- **We add white-noise of -0.3 to 0.3 for the "noisy" version.**

A Simple Autoencoder
Building the Autoencoder

- **Now we can build our neural network. We will use the functional API to build a simple 4-layer MLP:**

```
# We use the Functional API to create our autoencoder
aeinput = Input(shape = (2, ), name = 'input')
encoder = Dense(units = 32, activation = 'relu')(aeinput)
encoder = Dense(units = 16, activation = 'relu')(encoder)
decoder = Dense(units = 32, activation = 'relu')(encoder)
aeoutput = Dense(units = 2, activation = 'tanh')(decoder)

ae = Model(aeinput, aeoutput)
ae.compile(loss = 'mean_squared_error', optimizer = 'sgd')
```

- **In the output we use a "tanh" activation function so that our data can range from -1 to 1.**
- **Notice the change in name from "encoder" to "decoder".**
  - This lets us build separate encoder and decoder models if we want. Useful if we want to do compression.

A Simple Autoencoder
Generating Data

- **We now create our training, noisy and "wrong" data:**

```
# We set a fixed seed so that our experiments are reproducible
random.seed(24601)

# This adds white noise of -scale to scale
def noise(scale):
    return (2 * random.uniform(0, 1) - 1) * scale

def gen_X(data_size):
    return [random.uniform(0, 1) for i in range(data_size)]

def gen_Y(X):
    return [[x, (2*x + noise(DATA_NOISE) - 1)] for x in X]

# We create our dummy dataset and output, with 1000 numbers
X_in = gen_X(DATA_SIZE)
X_test = gen_X(VAL_SIZE)

# We create a simple linear function with noise for the network to learn
Y = gen_Y(X_in)
Y_out = np.array(Y)
Y_test = np.array(gen_Y(X_test))

# We create a noisy version of Y_out to check the anomaly detection
Y_noisy = np.array([[x, 2 * x + noise(TEST_NOISE) - 1] for x in X_in])

# We create an incorrect version of Y_out
Y_wrong = np.array([[x, 1.5*x + noise(DATA_NOISE) - 1]for x in X_in])
```

A Simple Autoencoder
Training the Autoencoder

- **Now we do the training. Notice that the target ("y = …") and input ("x = …") are exactly the same.**
  - This is because we are training the AE to recreate the input.

```
ae.fit(x = Y_out, y = Y_out, batch_size = 100,
epochs = EPOCHS, validation_data = (Y_test, Y_test))
```

A Simple Autoencoder Evaluation

- **Finally we call the AE's evaluate function to:**
  - Reconstruct the given input.
  - Find the MSE of the reconstructed input with the actual input, and print it out.

```
clean_loss = ae.evaluate(x = Y_out, y = Y_out)
test_loss = ae.evaluate(x = Y_test, y = Y_test)
noisy_loss = ae.evaluate(x = Y_noisy, y = Y_noisy)
wrong_loss = ae.evaluate(x = Y_wrong, y = Y_wrong)

print("Clean loss = %3.4f, Test loss = %3.4f Noisy loss = %3.4f, Wrong loss = %3.4f" %
(clean_loss, test_loss, noisy_loss, wrong_loss))
```

A Simple Autoencoder Evaluation

- **We run and get the following:**

```
10000/10000 [==============================] - 0s 11us/step - loss: 8.1981e-04 - v
Epoch 199/200
10000/10000 [==============================] - 0s 11us/step - loss: 8.1825e-04 - v
Epoch 200/200
10000/10000 [==============================] - 0s 11us/step - loss: 8.1653e-04 - v
10000/10000 [==============================] - 0s 13us/step
2000/2000 [==============================] - 0s 13us/step
10000/10000 [==============================] - 0s 13us/step
10000/10000 [==============================] - 0s 13us/step
Clean loss = 0.0008, Test loss = 0.0008 Noisy loss = 0.0042, Wrong loss = 0.0082
```

- **We can see that the losses for the noisy version (noisy loss) and abnormal version (wrong loss) are significantly higher than the correct data (clean loss) and validation loss (test loss).**
  - This is because the network's internal representation is meant only for the correct data.
  - When presented with incorrect data it will reconstruct the input poorly.

SWS3009 Embedded Systems and Deep Learning.

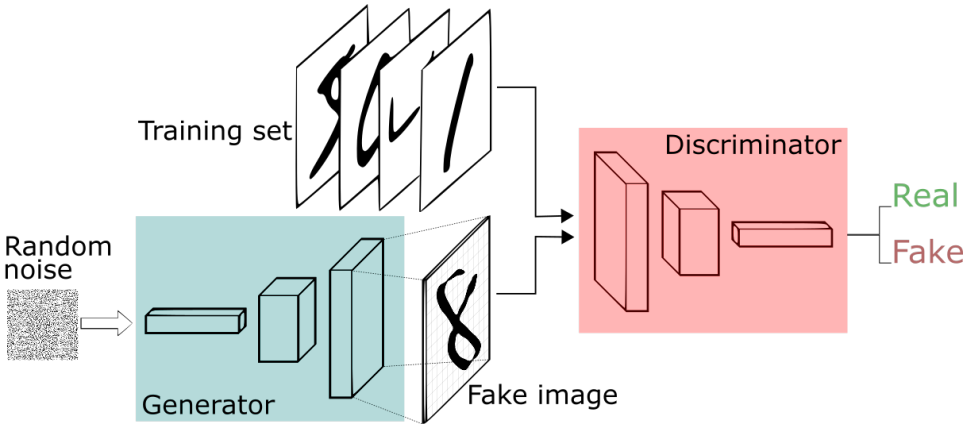
GENERATOR ADVERSARIAL NETWORKS

Generative Adversarial Networks
What Are They?

- **We now look at Generative Adversarial Networks, or GANs.**
- **Motivation:**
  - We often don't have enough training data.
  - We want to build a neural network that will produce "fake" data from the real data.
- **Consists of two networks:**
  - A "Generator" network that learns how to "counterfeit" the data.
    - ✓**Like a forger learning how to make better and better counterfeit money.**
  - A "Discriminator" network that learns how to differentiate between the real and the fake data.
    - ✓**Like a cop learning to differentiate between the increasingly good counterfeit money from the real money.**
  - We can use GANs to do very cool things like:
    - ✓**Producing images of people who don't exist (!!).**

Generative Adversarial Networks
What Are They?

- **The diagram below shows this idea:**
  - Generator:
    - ✓**Random noise is presented to the generator.**
    - ✓**The generator uses this to create its fake data.**
  - Discriminator:
    - ✓**Both fake and real data are presented to the discriminator.**
    - ✓**Discriminator is trained how to differentiate them.**



Generative Adversarial Networks
How Do They Work?

- Training the Generator:
  - ✓**Weights for the Discriminator are frozen.**
  - ✓**The GAN is now trained on the assumption that the fake data is real.**
  - ✓**End result: The Generator adjusts its weights to try to maximize the "realness" of the fake data.**
  - ✓**The process repeats.**
- So training alternates between:
  - ✓**Maximizing the Discriminator's ability to tell fake data from real.**
  - ✓**Maximizing the quality of the fake data to minimize the Discriminator's ability to differentiate.**

- **This is why it is called an "adversarial network":**
  - The two networks try to fight each other.

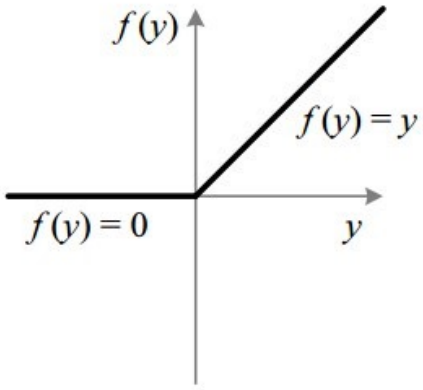
Generative Adversarial Networks
Imports

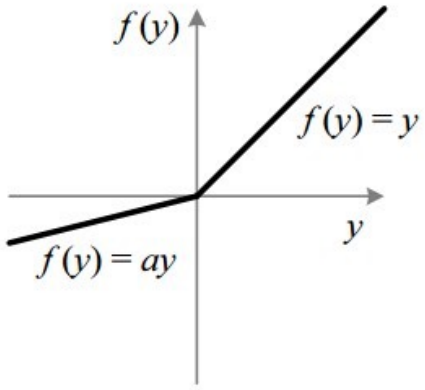
- **We will now build a GAN to generate fake MNIST data. Create a new file called "gan.py" and key in the following.**
- **As always we begin with the imports:**

```
import numpy as np
import matplotlib.pyplot as plt
import keras
from keras.layers import Dense, Dropout, Input
from keras.models import Model, Sequential
from keras.datasets import mnist
from tqdm import tqdm
from keras.layers.advanced_activations import LeakyReLU
from keras.optimizers import Adam
```

Generative Adversarial Networks
LeakyReLU

- **Most of this is pretty standard, except for:**
  - Matplotlib: This is a plotting library. Does not work properly on MacOS but we can still use it to generate PNG files.
  - Tqdm: This is a very cool progress bar library.
  - We are also importing a new activation function called LeakyReLU





Generative Adversarial Networks
Preparing the Data

- **We load the MNIST data, pretty much the same as before.**
- **Also since we are using the Adam optimizer twice (once each for the Generator and Discriminator), we create a function to create the optimizer.**

```
from keras.optimizers import Adam

def load_data():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train = (x_train.astype(np.float32) - 127.5) / 127.5

    # Reshape from 28x28 image to 784 element vector
    x_train = x_train.reshape(x_train.shape[0], 784)

    return (x_train, y_train, x_test, y_test)

(x_train, y_train, x_test, y_test) = load_data()

def adam_optimizer():
    return Adam(lr = 0.0002, beta_1 = 0.5)
```

Generative Adversarial Networks
Building the Generator

- **Now we create the Generator network. This is a simple MLP with:**
  - A 100 element input, which will be filled with 100 random numbers.
  - Two hidden layers.
  - A 784 element output since our image is 28x28.

```
def create_generator():
    generator = Sequential()
    generator.add(Dense(units = 256, input_dim = 100))
    generator.add(LeakyReLU(0.2))
    generator.add(Dense(units = 512))
    generator.add(LeakyReLU(0.2))
    generator.add(Dense(units = 784, activation = 'tanh'))
    generator.compile(loss = 'binary_crossentropy', optimizer = adam_optimizer())

    return generator
```

  - Here we use LeakyReLU with a = 0.2, so we have $f(x) = 0.2x$ when $x < 0$ and $f(x) = x$ when $x > 0$. $f(0)$ is defined to be 0.

Generative Adversarial Networks
Building the Discriminator

- **We similarly create the Discriminator. It has:**
  - 784 inputs, which is our 28x28 real or fake image.
  - Three hidden layers.
  - A single output which will be set to either 0 (fake) or 1 (true).

```
    return generator

def create_discriminator():
    discriminator = Sequential()
    discriminator.add(Dense(units = 1024, input_dim = 784))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(units = 512))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(units = 256))
    discriminator.add(LeakyReLU(0.2))

    discriminator.add(Dense(units = 1, activation = 'sigmoid'))

    discriminator.compile(loss = 'binary_crossentropy', optimizer = adam_optimizer())
    return discriminator
```

Generative Adversarial Networks
Building the GAN

- **Now we create the GAN. This is relatively simple:**
  - Take as inputs the Generator and Discriminator.
  - Create an input layer.
  - Pass the input to the Generator to create the fake image.
  - Pass this image to the Discriminator.
  - Set the GAN's input to the new input layer, and the GAN's output to the output of the Discriminator.
  - We use a binary cross-entropy loss since we are doing a binary 0/1 classification.

Generative Adversarial Networks
Building the GAN

```
# Create the GAN
def create_gan(discriminator, generator):
    discriminator.trainable = False
    gan_input = Input(shape = (100,))
    x = generator(gan_input)
    gan_output = discriminator(x)
    gan = Model(inputs = gan_input, output = gan_output)
    gan.compile(loss = 'binary_crossentropy', optimizer = 'adam')
    return gan
```

Generative Adversarial Networks Helper Function

- **Now we create a little helper function just to plot the generated images.**

```
# Display images
def plot_generated_images(epoch, generator, examples = 100, dim = (10, 10), figsize = (10, 10)):
    noise = np.random.normal(loc = 0, scale = 1, size = [examples, 100])
    generated_images = generator.predict(noise)
    generated_images = generated_images.reshape(100, 28, 28)
    plt.figure(figsize = figsize)

    for i in range(generated_images.shape[0]):
        plt.subplot(dim[0], dim[1], i + 1)
        plt.imshow(generated_images[i], interpolation = 'nearest')
        plt.axis('off')

    plt.tight_layout()
    plt.savefig('gan_generated_image %d.png' %epoch)
```

Generative Adversarial Networks Training

- **Finally we come to the fun part: Training!**
  - Create the Generator, Discriminator and GAN networks, and load the data.
  - In each epoch, for the Discriminator:
    - ✓**Generate fake data using random numbers.**
    - ✓**Randomly choose real data, and combine with fake data,**
    - ✓**Tag the real data with "1" and the fake data with "0".**
    - ✓**Train the Discriminator.**
  - In each epoch, for the Generator:
    - ✓**Freeze the Discriminator's weights.**
    - ✓**Generate the fake data using random numbers.**
    - ✓**Train the GAN with the assumption that the fake data is real.**
    - ✓**This forces the Generator to optimize its weights to maximize its score from the Discriminator (i.e. "convincing" the Discriminator to output a 1 instead of 0)**
    - ✓**The Discriminator itself is not affected since its weights are frozen.**

```
def training(epochs = 1, batch_size = 128):
    # Load the training data
    (X_train, y_train, X_test, y_test) = load_data()
    batch_count = X_train.shape[0] / batch_size

    # Create the GAN
    generator = create_generator()
    discriminator = create_discriminator()
    gan = create_gan(discriminator, generator)

    for e in range(1, epochs + 1):
        print("Epoch %d." % e)
        for _ in tqdm(range(batch_size)):

            # Generate random noise
            noise = np.random.normal(0, 1, [batch_size, 100])

            # Generate fake images
            generated_images = generator.predict(noise)

            # Get a random batch of real images
            image_batch = X_train[np.random.randint(low = 0, high = X_train.shape[0], size = batch_size)]
            X = np.concatenate([image_batch, generated_images])

            # Labels for the real and fake data
            y_dis = np.zeros(2 * batch_size)

            # Real data labels
            y_dis[:batch_size] = 0.9

            # Pretrain the discriminator on real and fake images
            # before starting the GAN

            discriminator.trainable = True
            discriminator.train_on_batch(X, y_dis)

            # Create fake data and fake labels
            noise = np.random.normal(0, 1, [batch_size, 100])
            y_gen = np.ones(batch_size)

            # We freeze the discriminator while training the GAN
            # So we alternate between training the discriminator
            # and the GAN

            discriminator.trainable = False
            gan.train_on_batch(noise, y_gen)

        if e == 1 or e % 20 == 0:
            plot_generated_images(e, generator)
```

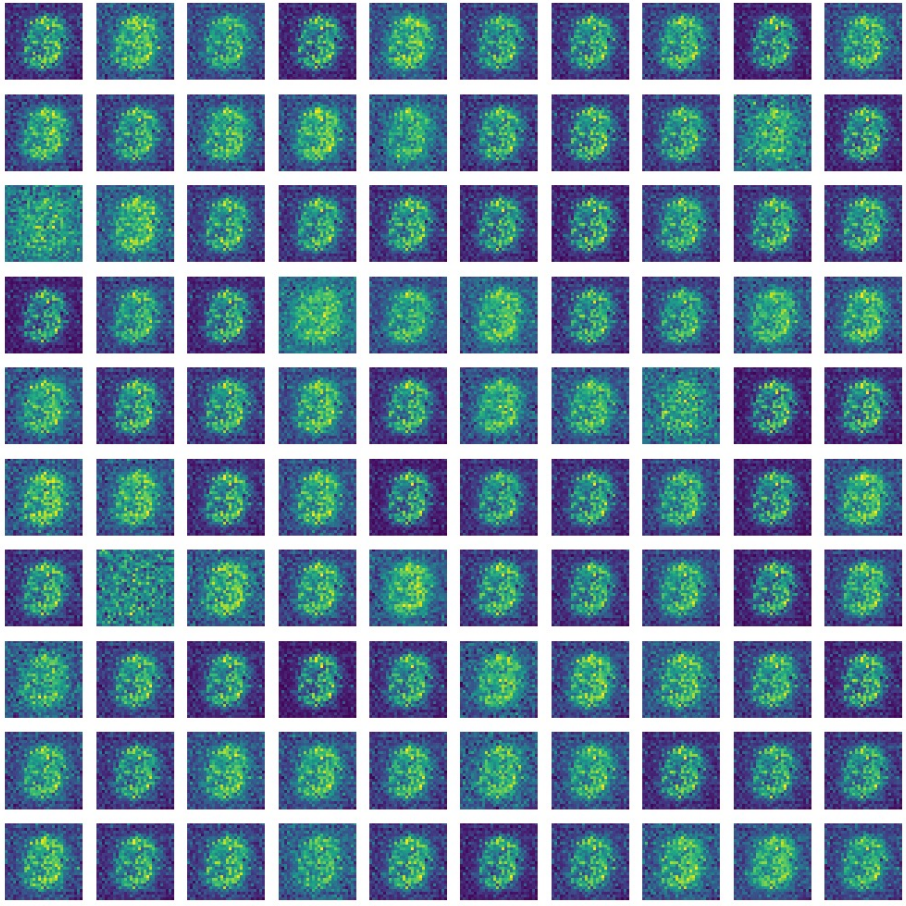
Generative Adversarial Networks

- **We end with just a single call to train for 400 epochs, with batch size of 128:**

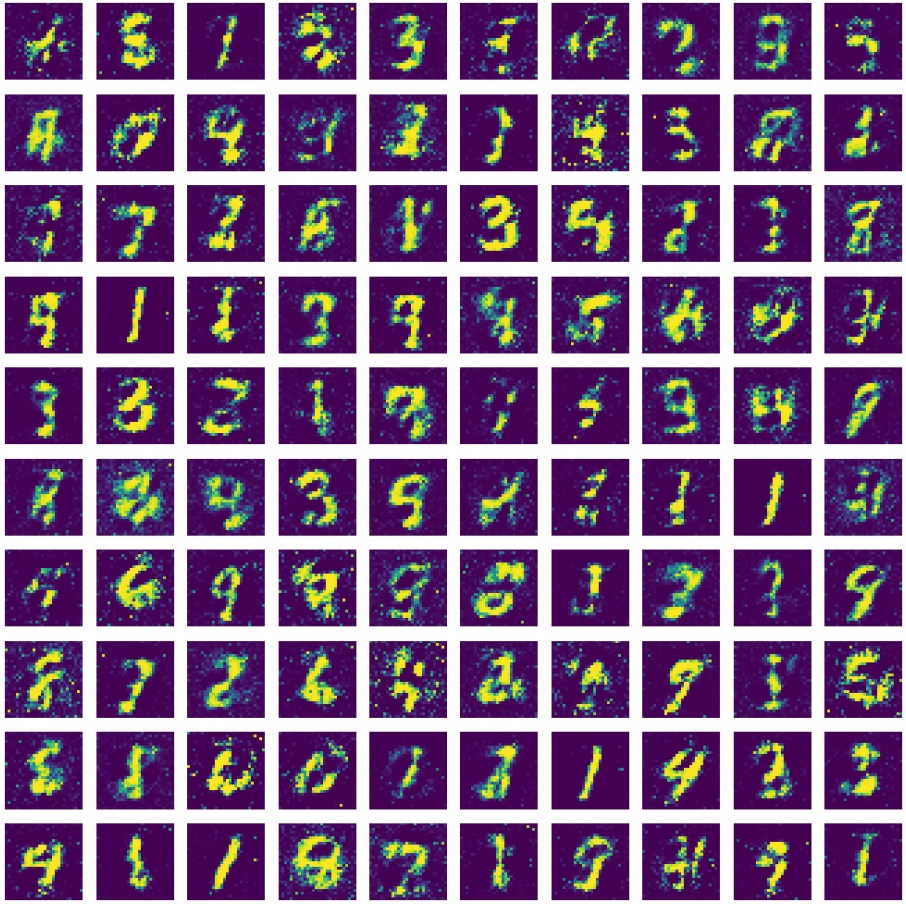
```
training(epochs = 400, batch_size = 128)
~
```

- **When run, the program will display the generated images on the screen (does not work with MacOS) and output to the PNG file with the filename "gan_generated_image xxx.jpg" where xxx is the epoch number.**
- **Next slide shows the results at various epochs.**

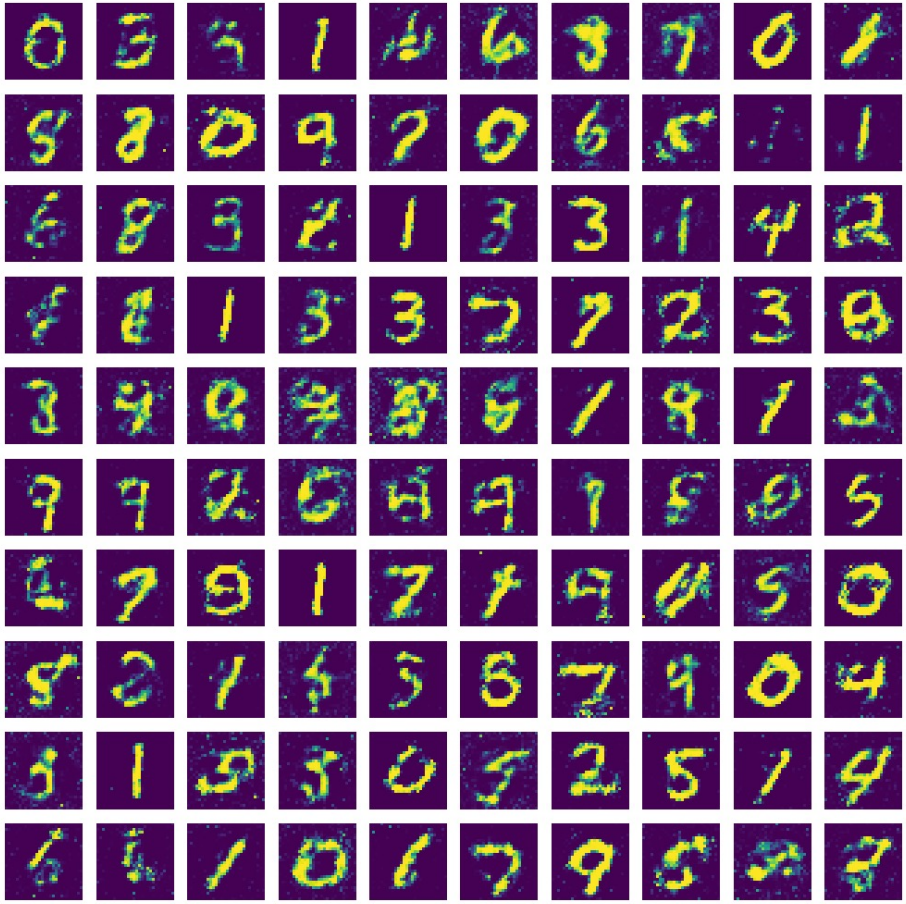
Generative Adversarial Networks
Epoch 1



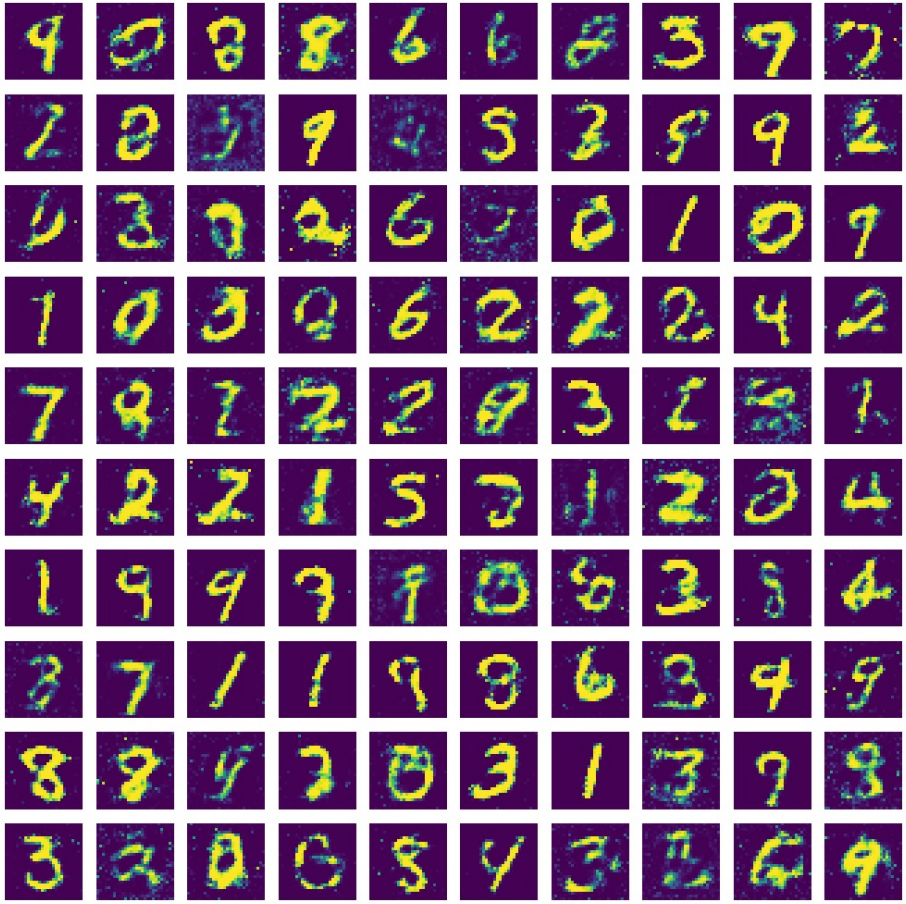
Generative Adversarial Networks
Epoch 20



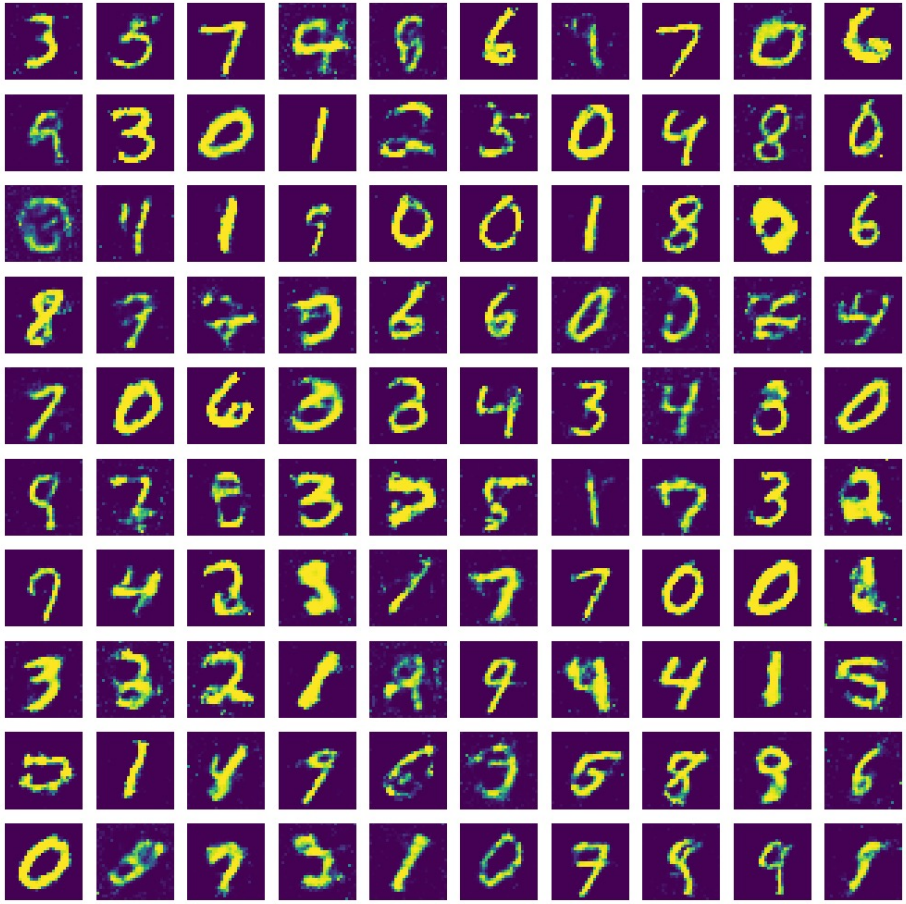
Generative Adversarial Networks
Epoch 60



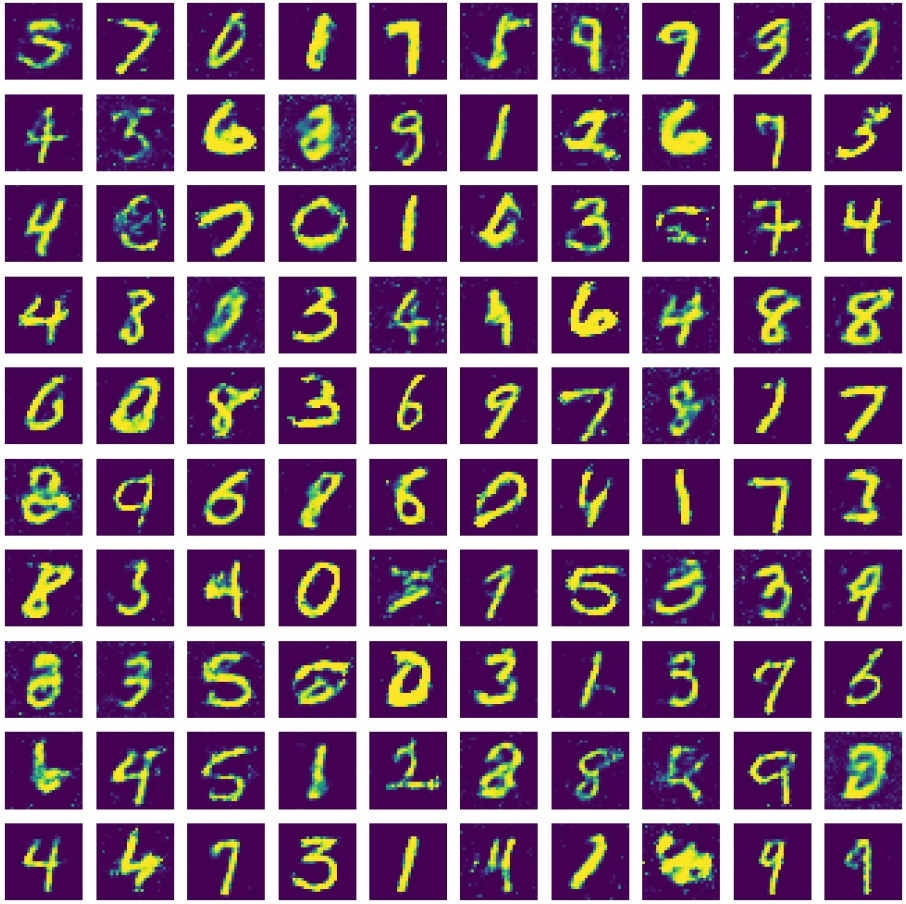
Generative Adversarial Networks
Epoch 100



Generative Adversarial Networks
Epoch 200



Generative Adversarial Networks
Epoch 400



Motivation

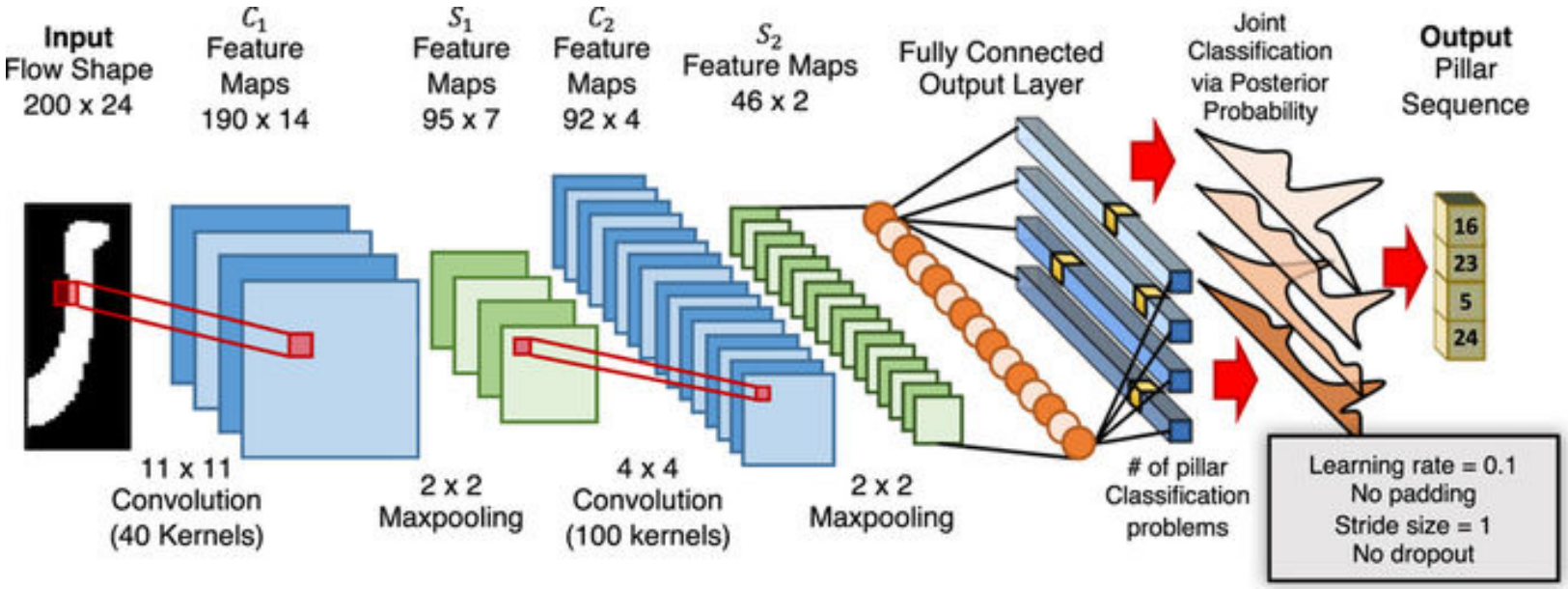
- **In the previous lecture we looked at LSTMs, Autoencoders and Generative-Adversarial Networks, and how to build and use them.**
- **In this lecture we look at:**
  - Convolutional Neural Networks: Traditionally used for image recognition, but can be used elsewhere as well.
  - An exploration of activations, loss functions and optimizers.
  - Dealing with Overfitting.
  - Introductory neural network theory:
    - ✓**Kohonen Self-Organizing Networks**
    - ✓**Perceptrons**
    - ✓**Multilayer Perceptrons**

NUS National University of Singapore
School of Computing

SWS3009 Embedded Systems and Deep Learning.

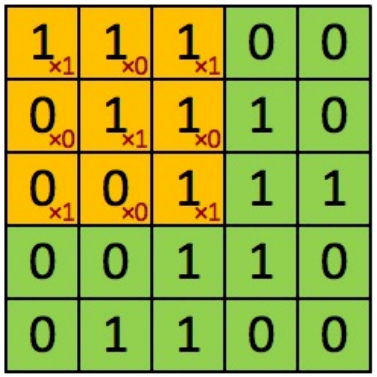
CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks
Revision: What is a CNN?

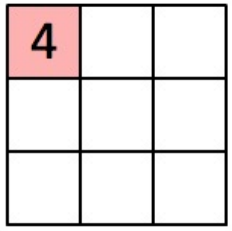


Convolutional Neural Networks
Convolution Layers

- **The CNN's primary distinguishing feature is the convolution layer.**
  - Made up of kernels that convolve over the input and over each other.



Image



Convolved Feature

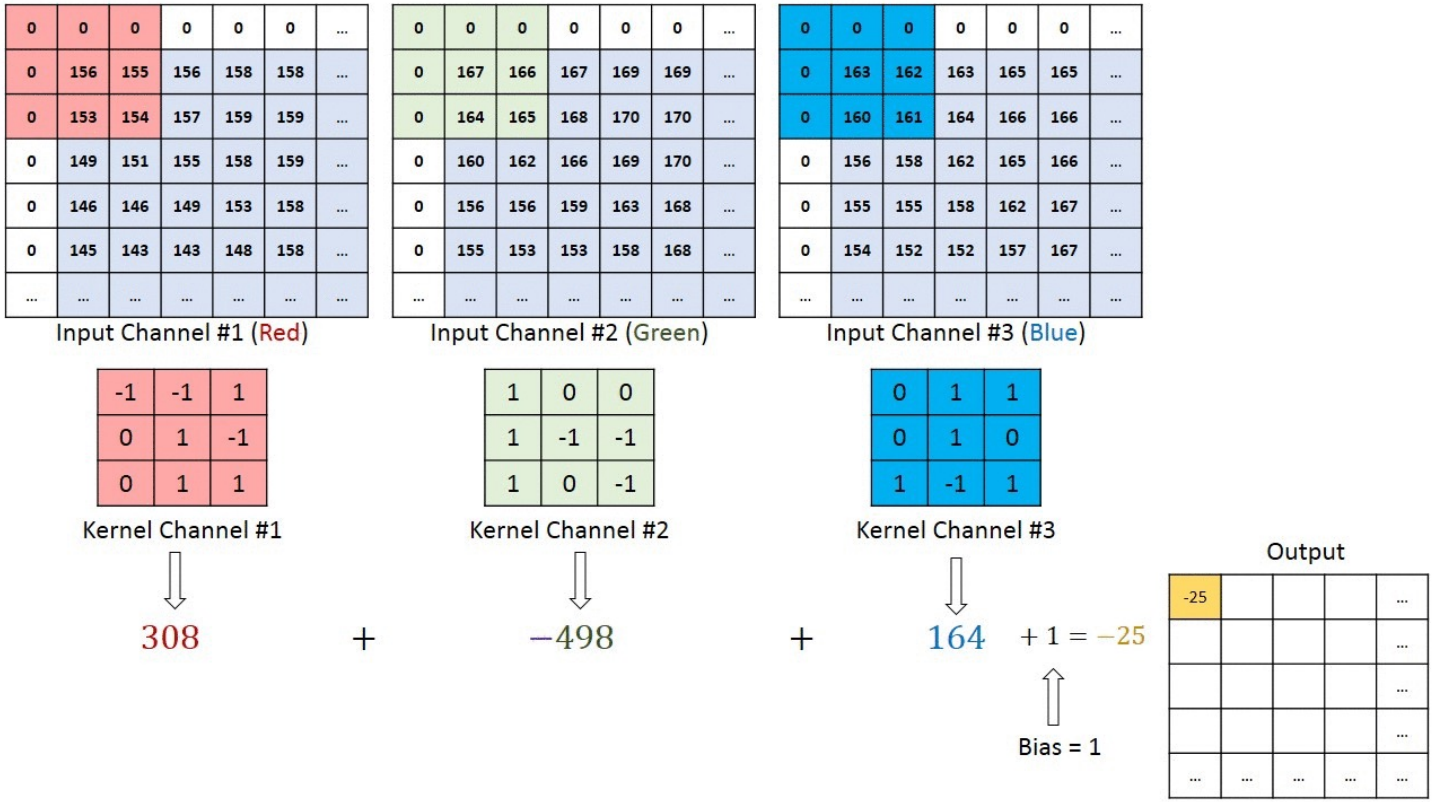
```
Kernel/Filter, K =

1  0  1
0  1  0
1  0  1
```

$$(I * K)_{xy} = \sum_{i=1}^{h} \sum_{j=1}^{w} K_{ij} \cdot I_{x+i-1,y+j-1}$$

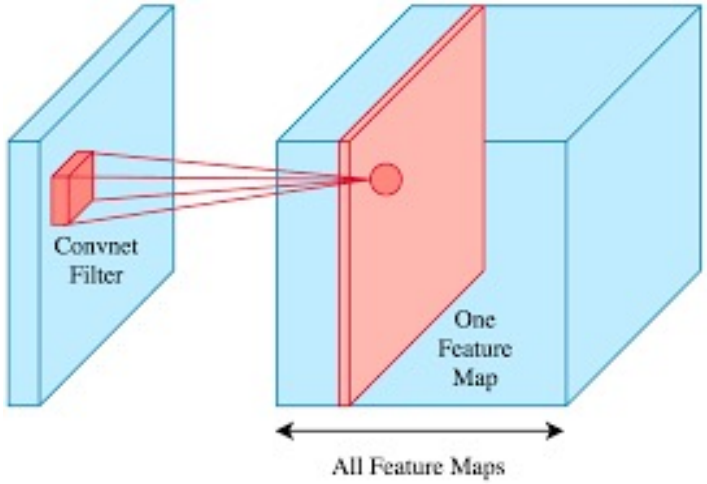
Convolutional Neural Networks
Convolution Layers

- **The depth of the kernel is equal to the depth of the data. E.g. with color images there are 3 channels, and the kernel is 3 layers deep:**



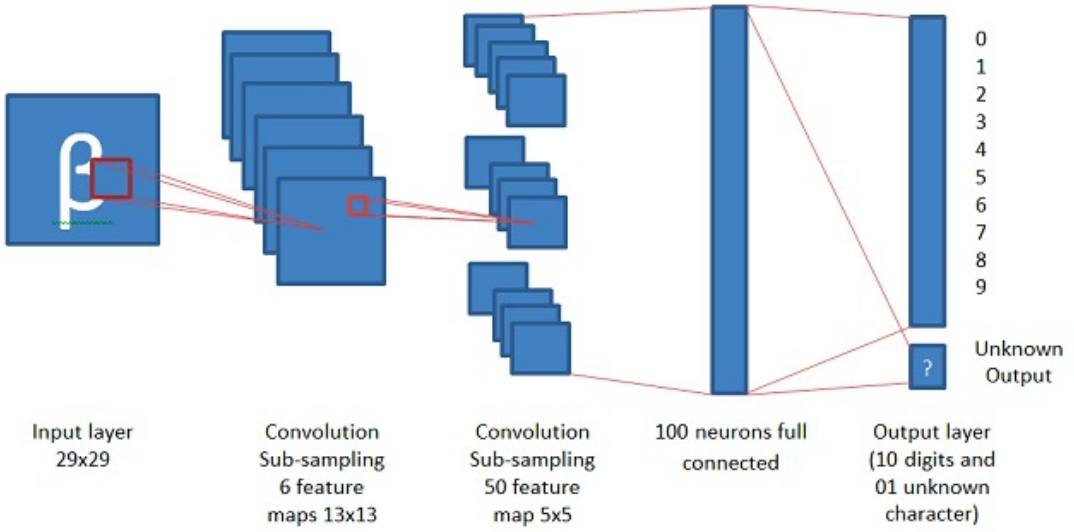
Convolutional Neural Networks
Convolution Layers

- **The kernels function as feature extractors and the output from the convolution operation is called a "feature map"**
  - As training progresses the kernels become optimized to extract key information like edges, repeating patterns, etc. in the input data.
- **Many kernels can be used on one layer.**
  - This will produce a volume of feature maps instead of a single feature map.
  - Due to the randomness of the initial kernel, each feature map ends up extracting a different feature of the input.



Convolutional Neural Networks
Convolution Layers

- **You can convolve kernels over earlier kernels.**
  - The feature maps generated will represent higher level features of the input.
  - E.g. the lower kernel might extract lines, the higher kernel might extract shapes.



Convolutional Neural Networks
Convolution Layers

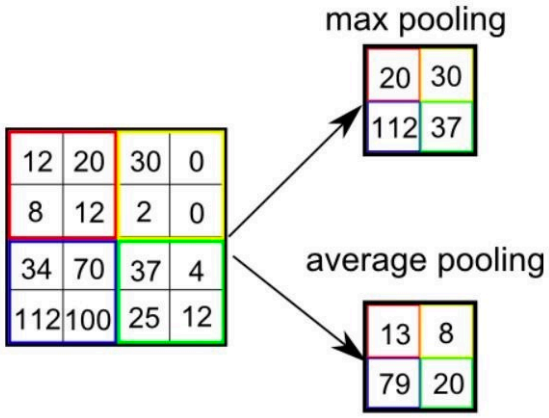
- **Convolution Kernel Hyper-Parameters:**
  - Size:
    - ✓**E.g. 3x3, 5x5, 9x9**
    - ✓**Typically smaller kernels extract more refined features at the expense of higher computational costs. Can be more sensitive to noise.**
    - ✓**Larger kernels use less computational power but produce features of lower resolution.**
  - Stride:
    - ✓**The # of steps a kernel moves to the right or down each time.**
    - ✓**Smaller strides capture more data but also produce larger feature maps.**
    - ✓**Larger strides produce smaller feature maps but lose more data.**
  - Padding:
    - ✓**What do we do when we reach the end of the input data and part of the kernel "hangs out" over the edge?**
    - ✓**Keras supports a "same" padding that preserves the size of the input. Other choices may increase or decrease the size of the input.**

Convolutional Neural Networks
Convolution Layers in Keras

- **Keras offers many types of convolution layers:**
  https://keras.io/layers/convolutional/
  - padding:
    - ✓**'valid': Convolution stops when the right/bottom edge of the filter hits the right/bottom edge of the input data. Will create a feature map smaller than the input**
    - ✓**'same': Input is padded with '0' to allow the feature map to be the same size as the input.**
  - activation:
    - ✓**If None, Keras will use an identify activation. I.e. id(x) = x, where x is the feature map computed by the convolution operation.**
    - ✓**Otherwise can specify other activations like ReLU, tanh etc.**
  - kernel_size:
    - ✓**Size of the kernel window. E.g. (3, 3) makes a 3x3 filter**
  - filters: Number of kernels to make.

Convolutional Neural Networks
Pooling Layers

- **The pooling layer takes a n x n region of the feature map and either picks the maximum or takes an average:**



- **Pooling layers:**
  - Reduce dimensionality and hence computing power.
  - Extracts the most dominant features (max pooling):
    - ✓**Makes the model shift invariant and less noise sensitive**
  - averages the features (average pooling)
    - ✓**Reduces the effect of noise by averaging it out – white noise has an average of 0.**

Convolutional Neural Networks
Pooling Layers

- **Pooling layer hyperparameters:**
  - stride: As with the convolution layer, this determines the number of steps the pool moves each time to the right or down.
  - size: Controls the dimensionality reduction.
    - ✓**Generally if stride = size, the feature map is reduced by 1/size in each dimension.**
  - Keras pooling API: https://keras.io/layers/pooling/
    - ✓**pool_size: The length or dimensions of the pool.**
    - ✓**strides: As described above.**
- **Generally the first few (or many?) layers of the CNN consist of alternating convolution and pooling layers:**
  - However stacking too many pooling layers can result in losing all data.

Convolutional Neural Networks
Flatten Layer / Example

- **The Flatten layer turns the feature maps into a 1D vector for feeding into the Dense layers.**
- **Example:**

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(5,5),
activation='relu',
input_shape=(28, 28, 1), padding='same'))

model.add(MaxPooling2D(pool_size=(2,2), strides=2))
model.add(Conv2D(64, kernel_size=(5,5), activation='relu'))
model.add(Conv2D(128, kernel_size=(5,5), activation='relu'))
model.add(Conv2D(64, kernel_size=(5,5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=2))
model.add(Flatten())
model.add(Dense(1024, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(10, activation='softmax'))
```

Summary

- **Over the past few lectures we have looked at various methods of machine learning:**
  - Neural Networks: Slow learning, potentially very powerful, can learn very complex patterns. Can be prone to overfitting due to high density of the parameters.
  - Deep Networks: Very large complex networks, but made up of many often heterogeneous types of networks that try to simplify the inputs, so that we can use a relatively small dense network to perform final decisions.
- **We've also seen how these have been applied to solve various problems.**
  - Think about what sort of problem you'd like to solve for your project.
  - Think about the kind of machine learning architectures.
  - Start collecting potentially relevant data (from places like Kaggle) and experiment.