**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING,**
**COLLEGE OF ENGINEERING, GUINDY,**
**ANNA UNIVERSITY, CHENNAI - 25.**

**CS6301 - MACHINE LEARNING**

# HANDWRITING GENERATION

PRANAVA RAMAN BMS - 2019103555
PREETI KRISHNAVENI RA - 2019103560
ANUSREE V - 2019103507

## CONTENTS

# INTRODUCTION

We know that in handwriting recognition models it learns that a series of pen points represents, say, the letter ‘a’. A much more difficult challenge is to reverse the process, ie. to train a model that takes the letter ‘a’ as an input and produces a series of points that we can connect to make the letter ‘a.’

Handwriting synthesis is the generation of handwriting for a given text. The resulting sequences are sufficiently convincing that they often cannot be distinguished from real handwriting. Furthermore, this realism is achieved without sacrificing the diversity in writing style. This is because the number of coordinates used to write each character varies greatly according to style, size, pen speed etc.

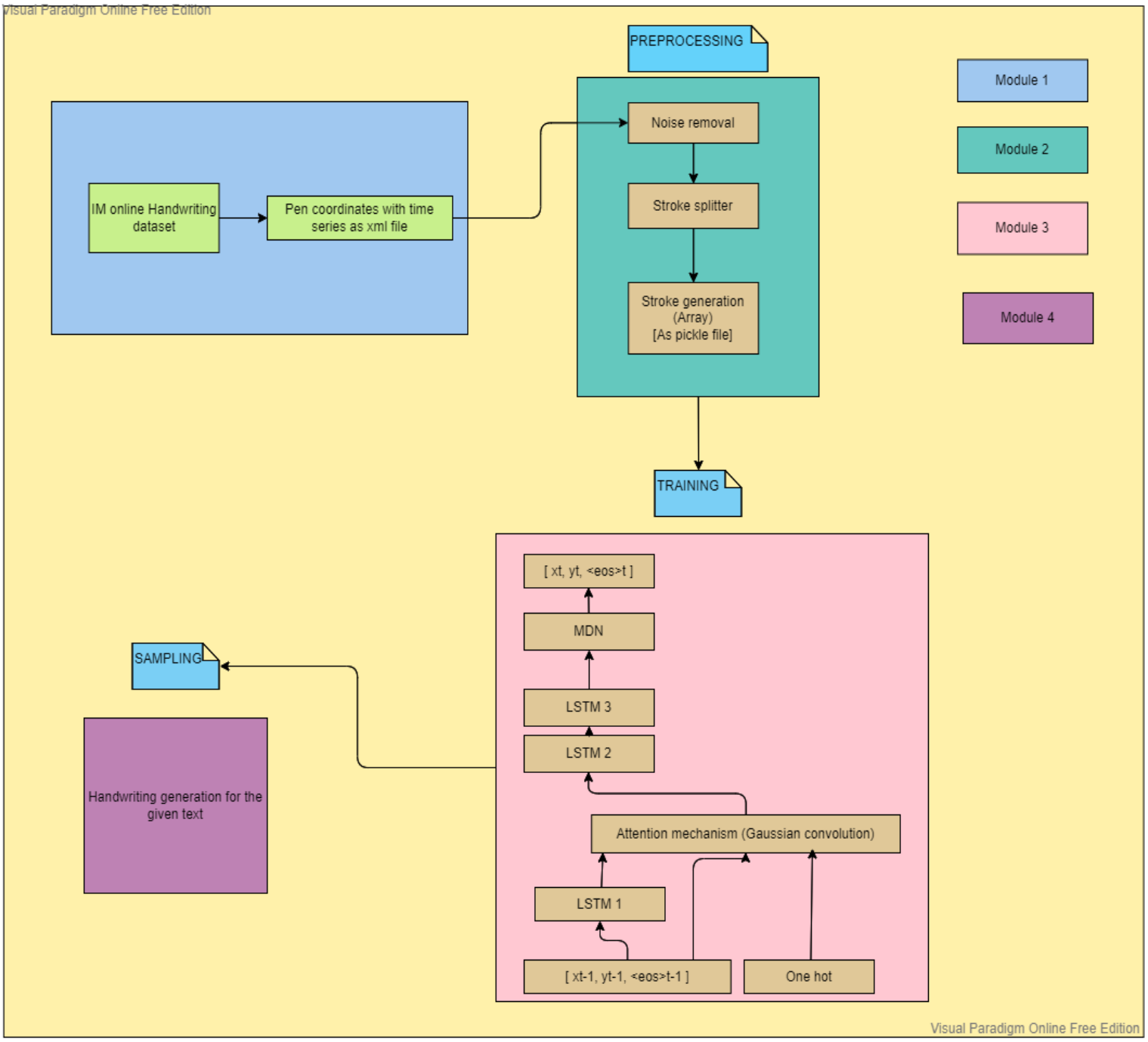
Deep generative models have been able to produce realistic handwritten text. The power of deep learning can be demonstrated by **recurrent neural networks** (RNNs) which can be viewed as a top class of dynamic models that is used to generate sequences in multiple domains such as machine translations, speech recognition, music, generating captions for input images and determining emotional tone behind a piece of text. RNNs are used to generate sequences by processing real data sequences and then predicting the next sequence. A large enough RNN is capable of generating sequences of subjective complexity. The **Long Short-Term Memory networks** (LSTMs) are a part of RNN architecture which is capable of storing and accessing information more efficiently than regular RNNs. In this paper, we have used the LSTM memory and the **Mixture Density Network** (MDN) to generate realistic sequences.

**Recurrent neural network (RNN) -** The major difference between a traditional neural network and an RNN is that the traditional neural networks cannot use its reasoning about previous events in the process to inform the later neurons or events. Traditional neural networks start the thinking process from scratch again. RNNs address this issue, by using loops in their network, thus making information persist. The passing of information from one step of the network to the next is allowed by a loop. An RNN network can be taken as one of many copies where each one of the copies passes information to the previous. In RNNs, each word in an input sequence in RNN will be mapped with a particular time step.

## RELATED WORKS

| Author name,Year | Type of Work | Methodology |
|---|---|---|
| Alex Graves Department of Computer Science University of Toronto “Generating Sequences With Recurrent Neural Networks” 2014 | Research Paper | RNN, LSTM, MDN |
| Suraj Bodapati, Sneha reddy “Realistic Handwriting Generation Using Recurrent Neural Networks and Long Short-Term Networks. ” 2020 | Conference Paper | RNN, LSTM, MDN |
| Sam Greydanus “Generating Realistic Handwriting with Tensorflow” 2016 | Implementation | RNN, LSTM, MDN |

# SYSTEM ARCHITECTURE



## MODULE 1 - DATASET

We used the IAM Handwriting Database to train the model. As far as datasets go, Although it's very small in size (less than 50 MB once parsed), preprocessing it generates a huge dataset. A total of 657 writers contributed to the dataset and each has a unique handwriting style.

The data itself is a three-dimensional time series. The first two dimensions are the (x, y) coordinates of the pen tip and the third is a binary 0/1 value where 1 signifies the end of a stroke. Each line has around 500 pen points and an annotation of ascii characters.

Suppose if we have to train the model with our own handwriting, we need to build a module to convert our handwriting into pen strokes with timeline data.

**SAMPLE DATA:**

```
    <Text>
By Trevor Williams. A move
to stop Mr. Gaitskell from
nominating any more Labour
life Peers is to be made at a
meeting of Labour M Ps
tomorrow. Mr. Michael Foot has
put down a resolution on the
subject and he is to be backed
by Mr. Will Griffiths, MP for
    </Text>
```

```
<TextLine id="a01-001z-01" text="By Trevor Williams. A move">
  <Word id="a01-001z-01-01" text="By">
    <Char id="a01-001z-01-01-01" text="B"/>
    <Char id="a01-001z-01-01-02" text="y"/>
  </Word>
```

```
<Stroke colour="black" start_time="13090871.35" end_time="13090871.78">
  <Point x="895" y="992" time="13090871.35"/>
  <Point x="890" y="987" time="13090871.37"/>
  <Point x="894" y="993" time="13090871.39"/>
  <Point x="893" y="992" time="13090871.40"/>
  <Point x="892" y="992" time="13090871.42"/>
  <Point x="892" y="1001" time="13090871.43"/>
  <Point x="892" y="1017" time="13090871.44"/>
  <Point x="890" y="1037" time="13090871.46"/>
  <Point x="885" y="1060" time="13090871.47"/>
  <Point x="880" y="1092" time="13090871.49"/>
  <Point x="873" y="1126" time="13090871.50"/>
  <Point x="863" y="1166" time="13090871.52"/>
  <Point x="859" y="1223" time="13090871.53"/>
  <Point x="851" y="1282" time="13090871.55"/>
  <Point x="842" y="1346" time="13090871.56"/>
  <Point x="833" y="1404" time="13090871.58"/>
  <Point x="828" y="1469" time="13090871.59"/>
  <Point x="821" y="1525" time="13090871.60"/>
  <Point x="816" y="1577" time="13090871.62"/>
  <Point x="813" y="1625" time="13090871.64"/>
  <Point x="813" y="1669" time="13090871.65"/>
  <Point x="814" y="1706" time="13090871.66"/>
  <Point x="815" y="1727" time="13090871.67"/>
  <Point x="819" y="1739" time="13090871.69"/>
  <Point x="823" y="1748" time="13090871.71"/>
  <Point x="827" y="1753" time="13090871.72"/>
  <Point x="830" y="1749" time="13090871.73"/>
  <Point x="836" y="1746" time="13090871.75"/>
  <Point x="838" y="1734" time="13090871.76"/>
</Stroke>
```

# MODULE 2 - PREPROCESSING

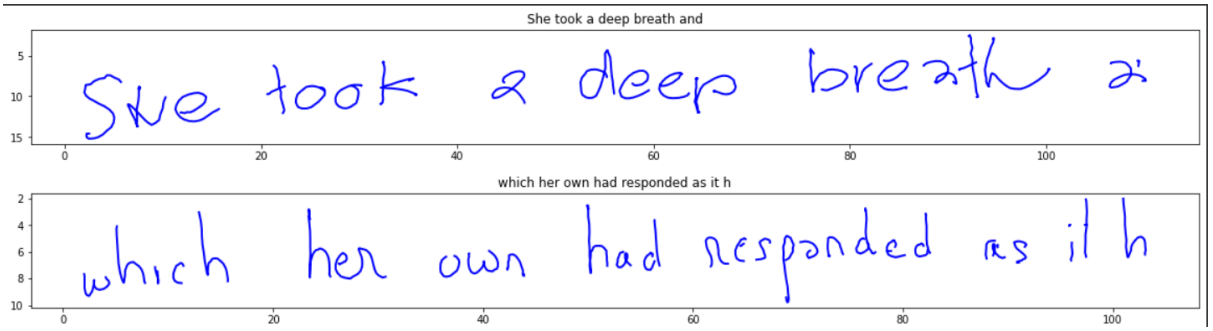
- The time series data is converted into stroke data with corresponding ascii input labels.
- We first take the inputs and find the distance between the adjacent pen co-ordinates.
- We remove noises at this part by removing points that are too distant.
- Then the strokes are converted into arrays and along with their labels, they are given output as pickle files. This is used as input for training purposes.
- Multiple styles of writing are saved as different pickle files.

## CODE SNIPPET:

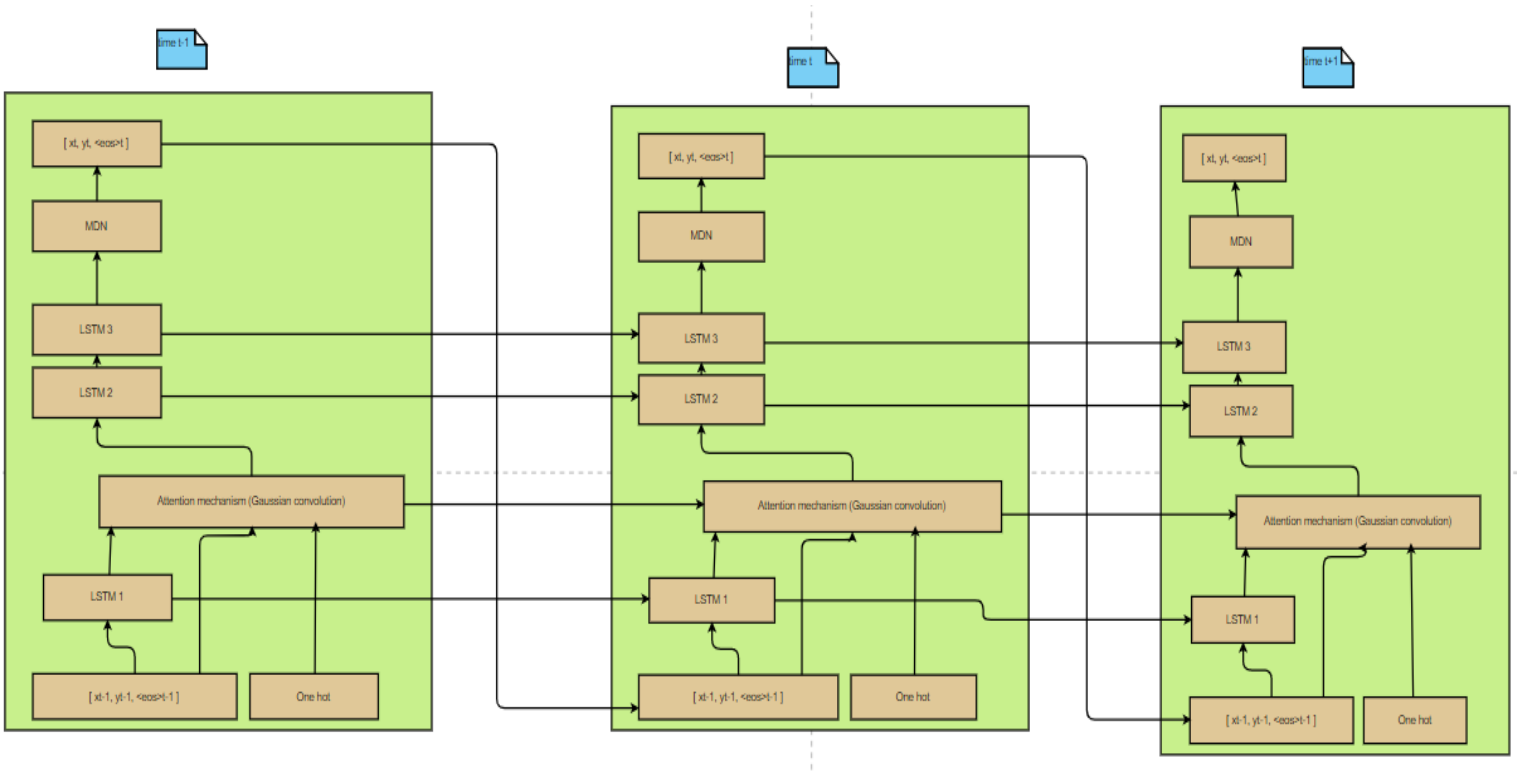
```
def preprocess(self, stroke_dir, ascii_dir, data_file):
    self.logger.write("\tparsing dataset...")

    filelist = []
    rootDir = stroke_dir
    for dirName, subdirList, fileList in os.walk(rootDir):
        for fname in fileList:
            filelist.append(dirName+"/"+fname)

    def getStrokes(filename):
        tree = ET.parse(filename)
        root = tree.getroot()

        result = []

        x_offset = 1e20
        y_offset = 1e20
        y_height = 0
        for i in range(1, 4):
            x_offset = min(x_offset, float(root[0][i].attrib['x']))
            y_offset = min(y_offset, float(root[0][i].attrib['y']))
            y_height = max(y_height, float(root[0][i].attrib['y']))
        y_height -= y_offset
        x_offset -= 100
        y_offset -= 100

        for stroke in root[1].findall('Stroke'):
            points = []
            for point in stroke.findall('Point'):
                points.append([float(point.attrib['x'])-x_offset,float(point.attrib['y'])-y_offset])
            result.append(points)
        return result
```

```python
def convert_stroke_to_array(stroke):
    n_point = 0
    for i in range(len(stroke)):
        n_point += len(stroke[i])
    stroke_data = np.zeros((n_point, 3), dtype=np.int16)

    prev_x = 0
    prev_y = 0
    counter = 0

    for j in range(len(stroke)):
        for k in range(len(stroke[j])):
            stroke_data[counter, 0] = int(stroke[j][k][0]) - prev_x
            stroke_data[counter, 1] = int(stroke[j][k][1]) - prev_y
            prev_x = int(stroke[j][k][0])
            prev_y = int(stroke[j][k][1])
            stroke_data[counter, 2] = 0
            if (k == (len(stroke[j])-1)):
                stroke_data[counter, 2] = 1
            counter += 1
    return stroke_data
```

## OUTPUT:

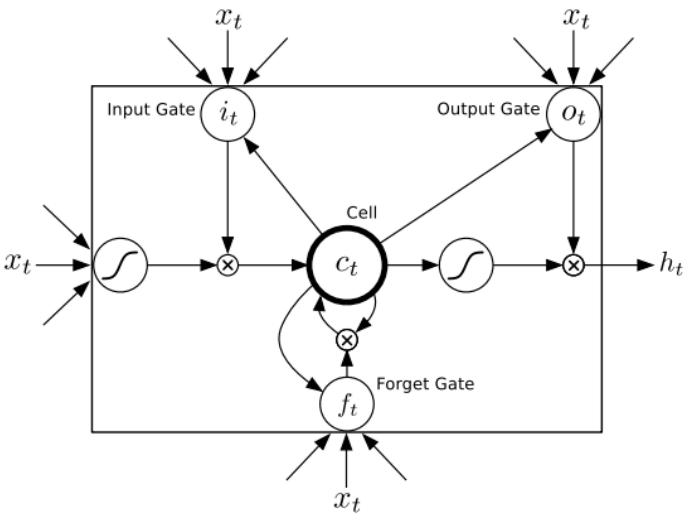


# MODULE 3 - TRAINING




The backbone of the model is three **LSTM** cells . There is a **custom attention mechanism** which digests a one-hot encoding of the sentence we want the model to write. The **Mixture Density Network** on top chooses appropriate Gaussian distributions from which to sample the next pen point, adding some natural randomness to the model.

**The Long Short-Term Memory (LSTM) Cell:**



- At the core of the Graves handwriting model are three Long Short-Term Memory (LSTM) Recurrent Neural Networks (RNNs).
- They keep a trail of time-independent patterns by using a differentiable memory.
- They are capable of learning long-term dependencies.
- In these LSTMs, we use three different tensors for performing each of the “erase”, “write” and “read” operations on the “memory” tensor. This helps in deciding what details to forget and what to remember.
- TensorFlow’s seq2seq API is used to build the model.
- RNNs are extremely good at modeling sequential data.

```
# ——— build the basic recurrent network architecture
cell_func = tf.contrib.rnn.LSTMCell # could be GRUCell or RNNCell
self.cell0 = cell_func(args.rnn_size, state_is_tuple=True, initializer=self.graves_initializer)
self.cell1 = cell_func(args.rnn_size, state_is_tuple=True, initializer=self.graves_initializer)
self.cell2 = cell_func(args.rnn_size, state_is_tuple=True, initializer=self.graves_initializer)

if (self.train and self.dropout < 1): # training mode
    self.cell0 = tf.contrib.rnn.DropoutWrapper(self.cell0, output_keep_prob = self.dropout)
    self.cell1 = tf.contrib.rnn.DropoutWrapper(self.cell1, output_keep_prob = self.dropout)
    self.cell2 = tf.contrib.rnn.DropoutWrapper(self.cell2, output_keep_prob = self.dropout)

self.input_data = tf.placeholder(dtype=tf.float32, shape=[None, self.tsteps, 3])
self.target_data = tf.placeholder(dtype=tf.float32, shape=[None, self.tsteps, 3])
self.istate_cell0 = self.cell0.zero_state(batch_size=self.batch_size, dtype=tf.float32)
self.istate_cell1 = self.cell1.zero_state(batch_size=self.batch_size, dtype=tf.float32)
self.istate_cell2 = self.cell2.zero_state(batch_size=self.batch_size, dtype=tf.float32)

#slice the input volume into separate vols for each tstep
inputs = [tf.squeeze(input_, [1]) for input_ in tf.split(self.input_data, self.tsteps, 1)]
#build cell0 computational graph
outs_cell0, self.fstate_cell0 = tf.contrib.legacy_seq2seq.rnn_decoder(inputs, self.istate_cell0, self.cell0, \
    loop_function=None, scope='cell0')
```

**The Attention Mechanism:**

- To get the information about what characters are used to make up the sentence, the model uses a differentiable attention mechanism.
- These are Gaussian convolutions over one-hot ascii encodings.
- This convolution operations as a soft-window through which the model can look into a small subset of characters (For instance, ‘wo’ in ‘world’)
- Since all the parameters are differentiable, the model shifts the window from character to character as it writes them.
- The model learns to control the window parameters remarkably well.

These three parameters control the character window wt according to:-

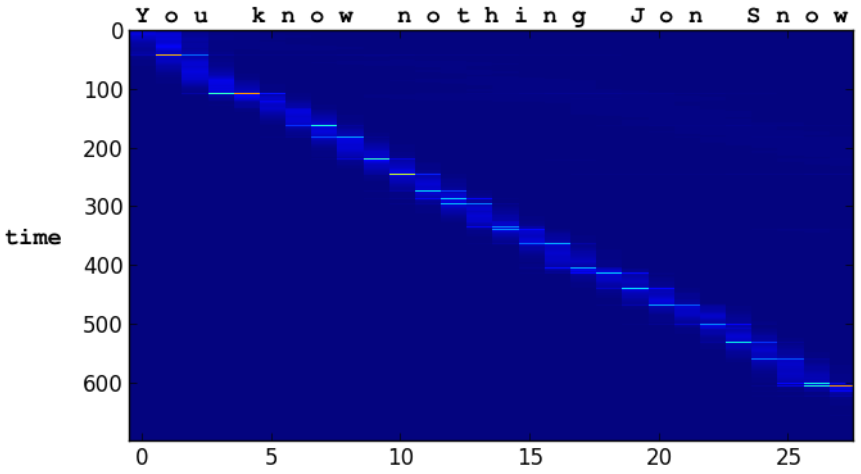
$$(\hat{\alpha}_t, \hat{\beta}_t, \hat{\kappa}_t) = W_{h^1p} h_t^1 + b_p$$

Each of these parameters are outputs from a dense layer on top of the first LSTM which we then transform according to:

$$\alpha_t = \exp(\hat{\alpha}_t) \qquad \beta_t = \exp(\hat{\beta}_t) \qquad \kappa_t = \kappa_{t-1} + \exp(\hat{\kappa}_t)$$

From these parameters we can construct the window as a convolution:

$$w_t = \sum_{u=1}^{U} \phi(t,u) c_u \qquad \phi(t,u) = \sum_{k=1}^{K} \alpha_t^k \exp\left(-\beta_t^k (\kappa_t^k - u)^2\right)$$

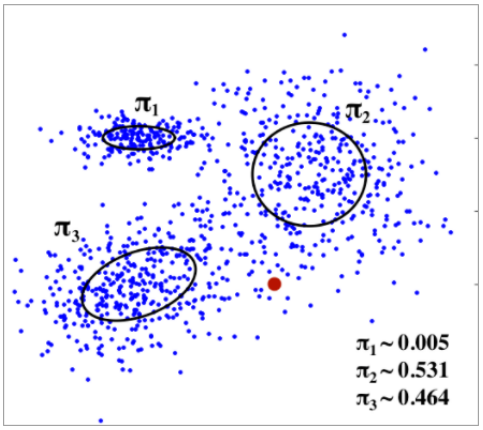



## Code:-

```python
# ——— build the gaussian character window
def get_window(alpha, beta, kappa, c):
    # phi -> [? x 1 x ascii_steps] and is a tf matrix
    # c -> [? x ascii_steps x alphabet] and is a tf matrix
    ascii_steps = c.get_shape()[1].value #number of items in sequence
    phi = get_phi(ascii_steps, alpha, beta, kappa)
    window = tf.matmul(phi,c)
    window = tf.squeeze(window, [1]) # window ~ [?,alphabet]
    return window, phi

#get phi for all t,u (returns a [1 x tsteps] matrix) that defines the window
def get_phi(ascii_steps, alpha, beta, kappa):
    # alpha, beta, kappa -> [?,kmixtures,1] and each is a tf variable
    u = np.linspace(0,ascii_steps-1,ascii_steps) # weight all the U items in the sequence
    kappa_term = tf.square( tf.subtract(kappa,u))
    exp_term = tf.multiply(-beta,kappa_term)
    phi_k = tf.multiply(alpha, tf.exp(exp_term))
    phi = tf.reduce_sum(phi_k,1, keep_dims=True)
    return phi # phi ~ [?,1,ascii_steps]

def get_window_params(i, out_cell0, kmixtures, prev_kappa, reuse=True):
    hidden = out_cell0.get_shape()[1]
    n_out = 3*kmixtures
    with tf.variable_scope('window',reuse=reuse):
        window_w = tf.get_variable("window_w", [hidden, n_out], initializer=self.graves_initializer)
        window_b = tf.get_variable("window_b", [n_out], initializer=self.window_b_initializer)
    abk_hats = tf.nn.xw_plus_b(out_cell0, window_w, window_b) # abk_hats ~ [?,n_out]
    abk = tf.exp(tf.reshape(abk_hats, [-1, 3*kmixtures,1])) # abk_hats ~ [?,n_out] = "alpha, beta, kappa hats"

    alpha, beta, kappa = tf.split(abk, 3, 1) # alpha_hat, etc ~ [?,kmixtures]
    kappa = kappa + prev_kappa
    return alpha, beta, kappa # each ~ [?,kmixtures,1]
```

**The Mixture Density Network (MDN):**

- Randomness and Styling in handwriting in this model are generated by using Mixture Density Networks
- MDNs can parametrize probability distributions, so they are very good at capturing the randomness in the data.
- We predict the mixture of Gaussian distributions by estimating their means and covariance with the output of the dense neural networks.
- In this model, MDN will choose the Gaussian with diffused shapes at the beginning of strokes and Gaussians with peaky shapes when in the middle of strokes.
- Here π can be thought of as the probability that the output values were drawn from that particular component's distribution. We can see that the π values for region 2 & 3 are similar whereas region 1 is very far away and hence very less. So we could say the red dot could belong to either of 2 or 3rd region.



```
# ——— finish building LSTMs 2 and 3
    outs_cell1, self.fstate_cell1 = tf.contrib.legacy_seq2seq.rnn_decoder(outs_cell0, self.istate_cell1, self.cell1, l

    outs_cell2, self.fstate_cell2 = tf.contrib.legacy_seq2seq.rnn_decoder(outs_cell1, self.istate_cell2, self.cell2, l

# ——— start building the Mixture Density Network on top (start with a dense layer to predict the MDN params)
    n_out = 1 + self.nmixtures * 6 # params = end_of_stroke + 6 parameters per Gaussian
    with tf.variable_scope('mdn_dense'):
        mdn_w = tf.get_variable("output_w", [self.rnn_size, n_out], initializer=self.graves_initializer)
        mdn_b = tf.get_variable("output_b", [n_out], initializer=self.graves_initializer)

    out_cell2 = tf.reshape(tf.concat(outs_cell2, 1), [-1, args.rnn_size]) #concat outputs for efficiency
    output = tf.nn.xw_plus_b(out_cell2, mdn_w, mdn_b) #data flows through dense nn


# ——— build mixture density cap on top of second recurrent cell
    def gaussian2d(x1, x2, mu1, mu2, s1, s2, rho):
        # define gaussian mdn (eq 24, 25 from http://arxiv.org/abs/1308.0850)
        x_mu1 = tf.subtract(x1, mu1)
        x_mu2 = tf.subtract(x2, mu2)
        Z = tf.square(tf.div(x_mu1, s1)) + \
            tf.square(tf.div(x_mu2, s2)) - \
            2*tf.div(tf.multiply(rho, tf.multiply(x_mu1, x_mu2)), tf.multiply(s1, s2))
        rho_square_term = 1-tf.square(rho)
        power_e = tf.exp(tf.div(-Z,2*rho_square_term))
        regularize_term = 2*np.pi*tf.multiply(tf.multiply(s1, s2), tf.sqrt(rho_square_term))
        gaussian = tf.div(power_e, regularize_term)
        return gaussian
```

# TRAINING CODE SNIPPET:

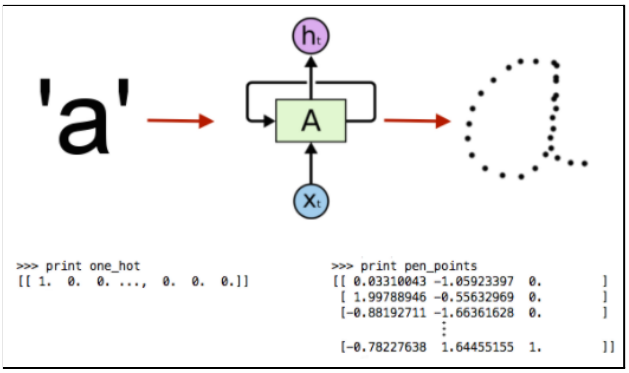
```
def train_model(args):
  logger = Logger(args) # make logging utility
  logger.write("\nTRAINING MODE...")
  logger.write("{}\n".format(args))
  logger.write("loading data...")
  data_loader = DataLoader(args, logger=logger)

  logger.write("building model...")
  model = Model(args, logger=logger)

  logger.write("attempt to load saved model...")
  load_was_success, global_step = model.try_load_model(args.save_path)

  v_x, v_y, v_s, v_c = data_loader.validation_data()
  valid_inputs = {model.input_data: v_x, model.target_data: v_y, model.char_seq: v_c}

  logger.write("training...")
  model.sess.run(tf.assign(model.decay, args.decay ))
  model.sess.run(tf.assign(model.momentum, args.momentum ))
  running_average = 0.0 ; remember_rate = 0.99
  for e in range(global_step//args.nbatches, args.nepochs):
    model.sess.run(tf.assign(model.learning_rate, args.learning_rate * (args.lr_decay ** e)))
    logger.write("learning rate: {}".format(model.learning_rate.eval()))

    c0, c1, c2 = model.istate_cell0.c.eval(), model.istate_cell1.c.eval(), model.istate_cell2.c.eval()
    h0, h1, h2 = model.istate_cell0.h.eval(), model.istate_cell1.h.eval(), model.istate_cell2.h.eval()
    kappa = np.zeros((args.batch_size, args.kmixtures, 1))

    for b in range(global_step%args.nbatches, args.nbatches):
```

```
    i = e * args.nbatches + b
    if global_step is not 0 : i+=1 ; global_step = 0

    if i % args.save_every == 0 and (i > 0):
      model.saver.save(model.sess, args.save_path, global_step = i) ; logger.write('SAVED MODEL')

    start = time.time()
    x, y, s, c = data_loader.next_batch()

    feed = {model.input_data: x, model.target_data: y, model.char_seq: c, model.init_kappa: kappa, \
        model.istate_cell0.c: c0, model.istate_cell1.c: c1, model.istate_cell2.c: c2, \
        model.istate_cell0.h: h0, model.istate_cell1.h: h1, model.istate_cell2.h: h2}

    [train_loss, _] = model.sess.run([model.cost, model.train_op], feed)
    feed.update(valid_inputs)
    feed[model.init_kappa] = np.zeros((args.batch_size, args.kmixtures, 1))
    [valid_loss] = model.sess.run([model.cost], feed)

    running_average = running_average*remember_rate + train_loss*(1-remember_rate)

    end = time.time()
    if i % 10 is 0: logger.write("{}/{}, loss = {:.3f}, regloss = {:.5f}, valid_loss = {:.3f}, time = {:.3f}" \
      .format(i, args.nepochs * args.nbatches, train_loss, running_average, valid_loss, end - start) )
```

## TRAINING OUTPUT:

```
TRAINING MODE...
<__main__.Args object at 0x7f1ad473c190>

loading data...
        loaded dataset:
                11262 train individual data points
                592 valid individual data points
                351 batches
building model...
        using alphabet abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
/tensorflow-1.15.2/python3.7/tensorflow_core/python/client/session.py:1750: Use
  warnings.warn('An interactive session is already active. This can '
attempt to load saved model...
no saved model to load. starting new session
training...
learning rate: 9.999999747378752e-05
0/125000, loss = 3.090, regloss = 0.03090, valid_loss = 3.061, time = 46.975
10/125000, loss = 3.106, regloss = 0.31705, valid_loss = 3.056, time = 0.446
20/125000, loss = 3.039, regloss = 0.57752, valid_loss = 3.046, time = 0.454
30/125000, loss = 2.912, regloss = 0.81331, valid_loss = 3.033, time = 0.446
40/125000, loss = 3.039, regloss = 1.02522, valid_loss = 3.014, time = 0.823
50/125000, loss = 2.982, regloss = 1.21226, valid_loss = 2.990, time = 0.444
60/125000, loss = 3.012, regloss = 1.37867, valid_loss = 2.960, time = 0.446
70/125000, loss = 2.879, regloss = 1.52687, valid_loss = 2.917, time = 0.447
80/125000, loss = 2.799, regloss = 1.65591, valid_loss = 2.855, time = 0.459
90/125000, loss = 2.761, regloss = 1.76637, valid_loss = 2.763, time = 0.446
100/125000, loss = 2.564, regloss = 1.85541, valid_loss = 2.612, time = 0.449
110/125000, loss = 2.413, regloss = 1.91864, valid_loss = 2.374, time = 0.445
120/125000, loss = 2.009, regloss = 1.94675, valid_loss = 2.033, time = 0.448
```

# MODULE 4 - SAMPLING




- Once the model is trained enough epochs (in our case about 27000 times) and the loss is minimized the model is saved. Then each time during sampling the model is loaded.
- The style of the author is used to prime the model. Priming consists of joining the real pen-positions and character sequences to generate and set the synthetic pen coordinates to a vector (the positions are sampled from MDN)
- The attention window helps it in generating pen strokes closer to what humans would do while writing.
- The output is again given as time series data of pen co-ordinates, which then again is processed to show the output in the form of line plots.

Code Snippets:-

```
def sample_text(sess, args_text, translation, style=None):
    fields = ['coordinates', 'sequence', 'bias', 'e', 'pi', 'mu1', 'mu2', 'std1', 'std2',
              'rho', 'window', 'kappa', 'phi', 'finish', 'zero_states']
    vs = namedtuple('Params', fields)(
        *[tf.compat.v1.get_collection(name)[0] for name in fields]
    )

    text = np.array([translation.get(c, 0) for c in args_text])
    coord = np.array([0., 0., 1.])
    coords = [coord]

    # Prime the model with the author style if requested
    prime_len, style_len = 0, 0
    if style is not None:
        # Priming consist of joining to a real pen-position and character sequences the synthetic sequence to generate
        #   and set the synthetic pen-position to a null vector (the positions are sampled from the MDN)
        style_coords, style_text = style
        prime_len = len(style_coords)
        style_len = len(style_text)
        prime_coords = list(style_coords)
        coord = prime_coords[0] # Set the first pen stroke as the first element to process
        text = np.r_[style_text, text] # concatenate on 1 axis the prime text + synthesis character sequence
        sequence_prime = np.eye(len(translation), dtype=np.float32)[style_text]
        sequence_prime = np.expand_dims(np.concatenate([sequence_prime, np.zeros((1, len(translation)))]), axis=0)

    sequence = np.eye(len(translation), dtype=np.float32)[text]
    sequence = np.expand_dims(np.concatenate([sequence, np.zeros((1, len(translation)))]), axis=0)
```

```
phi_data, window_data, kappa_data, stroke_data = [], [], [], []
sess.run(vs.zero_states)
sequence_len = len(args_text) + style_len
for s in range(1, 60 * sequence_len + 1):
    is_priming = s < prime_len

    print('\r[{:5d}] sampling... {}'.format(s, 'priming' if is_priming else 'synthesis'), end='')

    e, pi, mu1, mu2, std1, std2, rho, \
    finish, phi, window, kappa = sess.run([vs.e, vs.pi, vs.mu1, vs.mu2,
                                           vs.std1, vs.std2, vs.rho, vs.finish,
                                           vs.phi, vs.window, vs.kappa],
                                          feed_dict={
                                              vs.coordinates: coord[None, None, ...],
                                              vs.sequence: sequence_prime if is_priming else sequence,
                                              vs.bias: args.bias
                                          })
```

```
    if is_priming:
        # Use the real coordinate if priming
        coord = prime_coords[s]
    else:
        # Synthesis mode
        phi_data += [phi[0, :]]
        window_data += [window[0, :]]
        kappa_data += [kappa[0, :]]
        # ---
        g = np.random.choice(np.arange(pi.shape[1]), p=pi[0])
        coord = sample(e[0, 0], mu1[0, g], mu2[0, g],
                       std1[0, g], std2[0, g], rho[0, g])
        coords += [coord]
        stroke_data += [[mu1[0, g], mu2[0, g], std1[0, g], std2[0, g], rho[0, g], coord[2]]]

        if not args.force and finish[0, 0] > 0.8:
            print('Finished sampling!\n')
            break

coords = np.array(coords)
coords[-1, 2] = 1.

return phi_data, window_data, kappa_data, stroke_data, coords
```

The model thus generated can create legible handwriting for styles that aren't cursive. Cursive is still hard, and many times the generated output is not legible.

Since the model's MDN cap predicts the pen's (x,y) (x,y) coordinates by drawing them from a Gaussian distribution, we can modify that distribution to make the handwriting cleaner or messier. by introducing a bias b

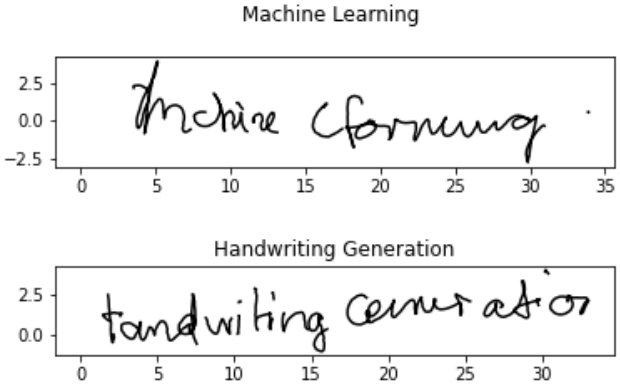
$$\sigma_t^j = \exp(\hat{\sigma}(1+b))$$

$$\pi_t^j = \frac{\exp(\hat{\pi}(1+b))}{\sum_{j'=1}^{M} \exp(\hat{\pi}(1+b))}$$

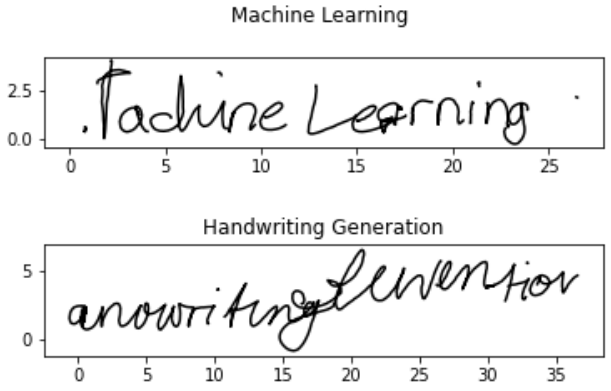
.

On lowering the bias the writing becomes more messier.

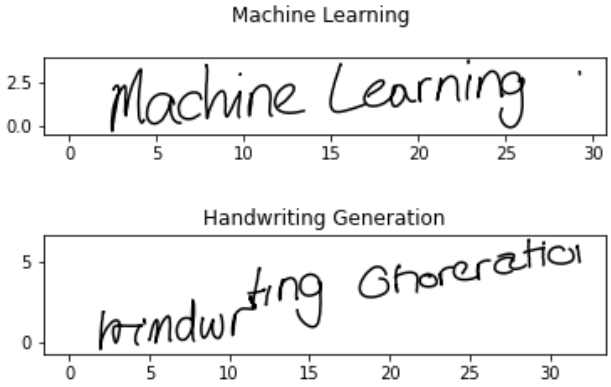
# RESULTS



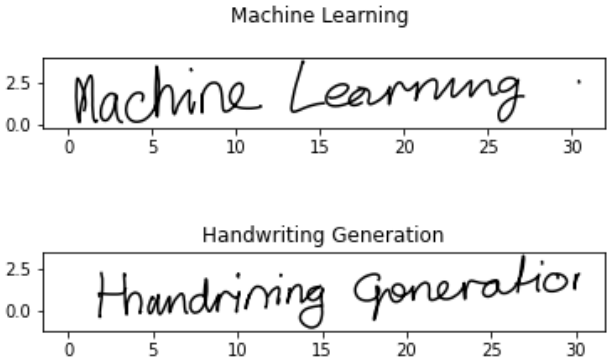

Bias = 0.0




Bias = 0.3




Bias - 0.7

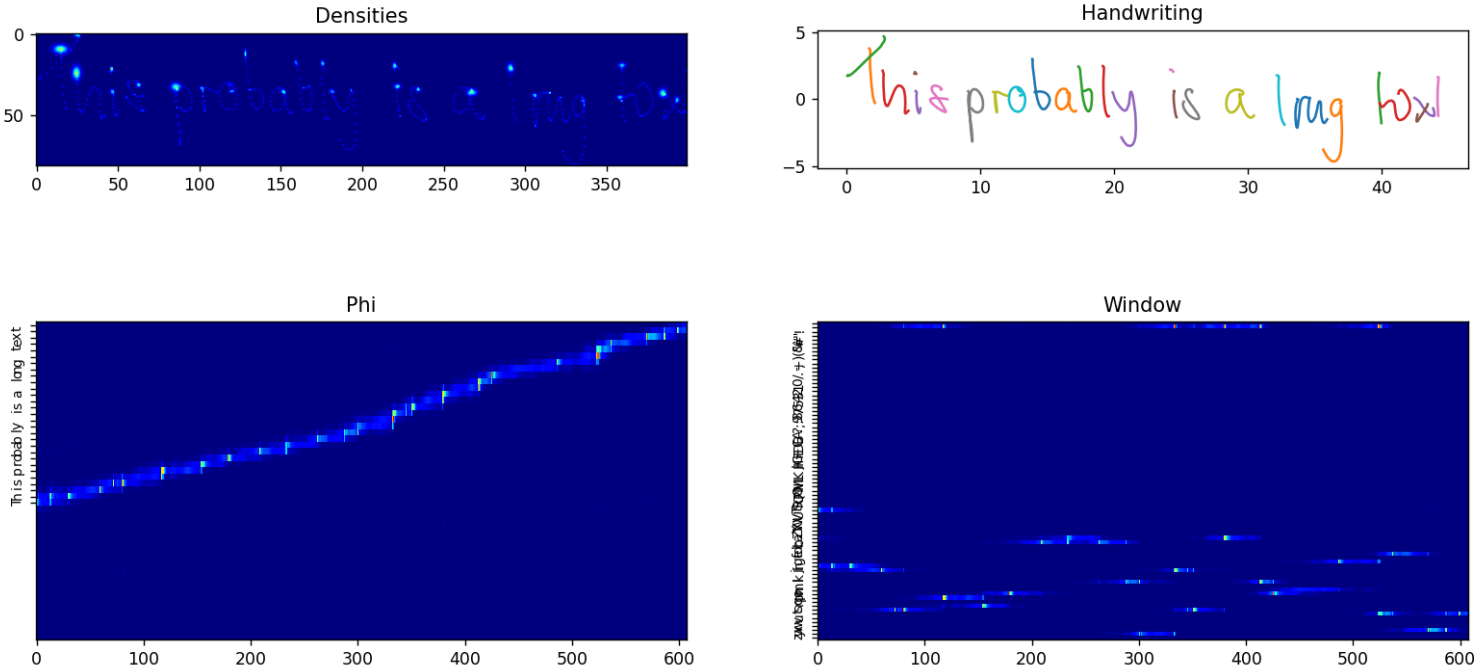



Bias = 1.0

Handwriting samples generated by the prediction network.

Input text file =
Machine Learning
Handwriting Generation

Input = “This is probably a long text”




## CONCLUSION AND FINAL SUMMARY

- This Project has demonstrated the ability of Long Short-Term Memory recurrent neural networks to generate both discrete and real-valued sequences with complex, long-range structure using next-step prediction.
- It has also introduced a novel convolutional mechanism that allows a recurrent network to condition its predictions on an auxiliary annotation sequence, and used this approach to synthesise diverse and realistic samples of online handwriting. Furthermore, it has shown how these samples can be biased towards greater legibility, and how they can be modelled on the style of a particular writer.

## REFERENCES


- Generating Sequences With Recurrent Neural Networks: Alex Graves [1308.0850.pdf (arxiv.org)]
- Realistic Handwriting Generation Using Recurrent Neural Networks and Long Short-Term Networks | SpringerLink
- Grzego/handwriting-generation: Implementation of handwriting generation with use of recurrent neural networks in tensorflow. Based on Alex Graves paper (https://arxiv.org/abs/1308.0850). (github.com)
- greydanus/scribe Realistic Handwriting with Tensorflow (github.com)


**Thanking You:-**