Autobot for Character Generation

This Jupyter notebook contains a standalone demonstration of the AutoBot method/architecture applied to the Omniglot stroke-completion task ("task 1" in the paper). This means, we download the Omniglot stroke dataset, and train to complete character strokes by showing the model the first half of a stroke and training it to predict the second half.

This notebook has the following sections:

1. Downloading the Omniglot dataset and preparing the data
2. Creating a dataloader for Pytorch to prepare the data for training/testing
3. Setting up the Autobot model
4. Setting up the loss functions
5. Training the model (est 45-60min)
6. Plot/test functions to generate qualitative samples from the trained model on the test dataset

This notebook should contain a trained model and if you scroll all the way to the end of the test section, there should be sample images. If that's not the case, please rerun the entire notebook.

Setting Up the Omniglot dataset

In the section, we download and process the Omniglot dataset to generate and training and testing partitions.

In [ ]:

```
!git clone https://github.com/brendenlake/omniglot.git
!unzip "omniglot/python/images_background.zip"
!unzip "omniglot/python/images_evaluation.zip"
!unzip "omniglot/python/strokes_background.zip"
!unzip "omniglot/python/strokes_evaluation.zip"
```

In [2]:

```
import numpy as np
import os
import random
from sys import platform as sys_pf


# Color map for the stroke of index k
def get_color(k):
    scol = ['r', 'g', 'b', 'm', 'c']
    ncol = len(scol)
    if k < ncol:
        out = scol[k]
    else:
        out = scol[-1]
    return out


# convert to str and add leading zero to single digit numbers
def num2str(idx):
    if idx < 10:
        return '0' + str(idx)
    return str(idx)


# Load binary image for a character
#
# fn : filename
def load_img(fn):
    I = plt.imread(fn)
    I = np.array(I, dtype=bool)
    return I


# Load stroke data for a character from text file
#
# Input
#   fn : filename
#
# Output
#   motor : list of strokes (each is a [n x 3] numpy array)
#      first two columns are coordinates
#          the last column is the timing data (in milliseconds)
def load_motor(fn):
    motor = []
    with open(fn, 'r') as fid:
        lines = fid.readlines()
    lines = [l.strip() for l in lines]
    for myline in lines:
        if myline == 'START':  # beginning of character
            stk = []
        elif myline == 'BREAK':  # break between strokes
            stk = np.array(stk)
            motor.append(stk)  # add to list of strokes
            stk = []
        else:
            arr = np.fromstring(myline, dtype=float, sep=',')
            stk.append(arr)
```

```python
    return motor


def space_motor_to_img(pt):
    pt[:, 1] = -pt[:, 1]
    return pt


def space_img_to_motor(pt):
    pt[:, 1] = -pt[:, 1]
    return


for stroke_dir in ['strokes_background', 'strokes_evaluation']:
  num_strokes = 0
  max_num_strokes = 0
  num_valid_strokes = 0  # valid data needs to have more than one stroke and the length of each stroke should be more t
  stroke_lengths = []
  train_valid_fnames = []
  test_valid_fnames = []

  alphabet_names = [a for a in os.listdir(stroke_dir) if a[0] != '.']  # get folder names

  for alpha_name in alphabet_names:  # for each alphabet

      char_dirs = sorted(os.listdir(os.path.join(stroke_dir, alpha_name)))
      for char_dir in char_dirs:
          if "char" not in char_dir:  # protect against useless folders, .DS_STORE
              continue

          char_renditions = os.listdir(os.path.join(stroke_dir, alpha_name, char_dir))
          valid_data = []
          for char_rendition in char_renditions:
              fname_stroke = os.path.join(stroke_dir, alpha_name, char_dir, char_rendition)
              strokes = load_motor(fname_stroke)
              num_strokes += 1

              if len(strokes) > 1:
                  valid_strokes = True
                  for stroke in strokes:
                      if len(stroke) < 10 or len(stroke) > 100:
                          valid_strokes = False
                          break

                  if valid_strokes:
                      num_valid_strokes += 1
                      if len(strokes) > max_num_strokes:
                          max_num_strokes = len(strokes)

                      for stroke in strokes:
                          stroke_lengths.append(len(stroke))

                      valid_data.append(fname_stroke)

          if len(valid_data) > 0:
              if len(valid_data) > 3:
                  num_train = int(0.75*len(valid_data))
                  train_valid_fnames += valid_data[:num_train]
                  test_valid_fnames += valid_data[num_train:]
              else:
                  train_valid_fnames += valid_data

  print("Number of points", num_strokes)
  print("Number of valid points", num_valid_strokes)
  print("Max Number of strokes", max_num_strokes)

  with open(stroke_dir+'_train.txt', 'w') as f:
      for item in train_valid_fnames:
          f.write("%s\n" % item)

  with open(stroke_dir+'_test.txt', 'w') as f:
      for item in test_valid_fnames:
          f.write("%s\n" % item)
```

```
Number of points 19280
Number of valid points 6019
Max Number of strokes 12
Number of points 13180
Number of valid points 4644
Max Number of strokes 11
```

Creating a pytorch dataloader

In [3]:

```python
!pip install -q torch==1.9.0 torchvision==0.10.0
```

```
     |████████████████████████████████| 831.4 MB 2.7 kB/s
     |████████████████████████████████| 22.1 MB 30.4 MB/s
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behavio
ur is the source of the following dependency conflicts.
torchtext 0.11.0 requires torch==1.10.0, but you have torch 1.9.0 which is incompatible.
torchaudio 0.10.0+cu111 requires torch==1.10.0, but you have torch 1.9.0 which is incompatible.
```

In [4]:

```python
import os
from torch.utils.data import Dataset
import numpy as np
from scipy import signal


class OmniGlotDataset(Dataset):
    def __init__(self, dset_path=".", in_seq_len=10, entire_seq_len=20, split_name='train'):
        self.in_seq_len = in_seq_len

        with open(os.path.join(dset_path, "strokes_background_{}.txt".format(split_name)), 'r') as fid:
            lines = fid.readlines()
        char_fnames = [l.strip() for l in lines]

        with open(os.path.join(dset_path, "strokes_evaluation_{}.txt".format(split_name)), 'r') as fid:
            lines = fid.readlines()
        char_fnames += [l.strip() for l in lines]

        max_num_strokes = 0
        self.data = []
        self.fnames = []
        for char_fname in char_fnames:
            self.fnames.append(char_fname)
            char_data = self.load_motor(os.path.join(dset_path, char_fname))
            new_char_data = []
            for i in range(len(char_data)):
                new_char_data.append(signal.resample(char_data[i], num=entire_seq_len))
                new_char_data[-1][0] = char_data[i][0]

            curr_data = np.array(new_char_data)
            curr_data[:, :, 2] = 1.0
            curr_data = self.normalize(curr_data)

            if len(curr_data) > max_num_strokes:
                max_num_strokes = len(curr_data)
            if len(curr_data) < 12:
                new_curr_data = np.zeros((12, entire_seq_len, 3))
                new_curr_data[:len(curr_data)] = curr_data
                self.data.append(new_curr_data)
            else:
                self.data.append(curr_data)

    def normalize(self, data):
        min_x = np.min(data[:, :, 0])
        max_x = np.max(data[:, :, 0])
        min_y = np.min(data[:, :, 1])
        max_y = np.max(data[:, :, 1])

        data[:, :, 0] -= min_x
        data[:, :, 0] /= (max_x - min_x)
        data[:, :, 0] *= 10.0  # factor for stability

        data[:, :, 1] -= min_y
        data[:, :, 1] /= (max_y - min_y)
        data[:, :, 1] *= 10.0  # factor for stability
        return data

    def load_motor(self, fn):
        motor = []
        with open(fn, 'r') as fid:
            lines = fid.readlines()
        lines = [l.strip() for l in lines]
        for myline in lines:
            if myline == 'START':  # beginning of character
                stk = []
            elif myline == 'BREAK':  # break between strokes
                stk = np.array(stk)
                motor.append(stk)  # add to list of strokes
                stk = []
            else:
                arr = np.fromstring(myline, dtype=float, sep=',')
                stk.append(arr)
        return motor

    def __getitem__(self, idx: int):
        data = self.data[idx].transpose((1, 0, 2))
        past = data[:self.in_seq_len]
        future = data[self.in_seq_len:]
        fname = self.fnames[idx]
        return past, future, fname

    def __len__(self):
        return len(self.data)
```

In [5]:

```python
dset = OmniGlotDataset(split_name="train")
past, future, fname = dset[0]
print(past.shape)
print(future.shape)
print(fname)
```

```
(10, 12, 3)
(10, 12, 3)
```

strokes_background/Asomtavruli_(Georgian)/character01/0068_16.txt

Model Code

In [25]:

```
import math

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F


def init(module, weight_init, bias_init, gain=1):
    weight_init(module.weight.data, gain=gain)
    bias_init(module.bias.data)
    return module


class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout=0.1, max_len=20):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0).transpose(0, 1)
        self.register_buffer('pe', pe)

    def forward(self, x):
        '''
        :param x: must be (T, B, H)
        :return:
        '''
        x = x + self.pe[:x.size(0), :]
        return self.dropout(x)


class OutputModel(nn.Module):
    def __init__(self, hidden_size=64):
        super(OutputModel, self).__init__()
        self.hidden_size = hidden_size
        init_ = lambda m: init(m, nn.init.xavier_normal_, lambda x: nn.init.constant_(x, 0), np.sqrt(2))
        self.observation_model = nn.Sequential(
            init_(nn.Linear(hidden_size, hidden_size)), nn.ReLU(),
            init_(nn.Linear(hidden_size, hidden_size)), nn.ReLU(),
            init_(nn.Linear(hidden_size, 5))
        )
        self.min_stdev = 0.01

    def forward(self, agent_latent_state):
        T = agent_latent_state.shape[0]
        BK = agent_latent_state.shape[1]
        pred_obs = self.observation_model(agent_latent_state.reshape(-1, self.hidden_size)).reshape(T, BK, -1)
        x_mean = pred_obs[:, :, 0]
        y_mean = pred_obs[:, :, 1]
        x_sigma = F.softplus(pred_obs[:, :, 2]) + self.min_stdev
        y_sigma = F.softplus(pred_obs[:, :, 3]) + self.min_stdev
        rho = torch.tanh(pred_obs[:, :, 4]) * 0.9  # for stability
        return torch.stack([x_mean, y_mean, x_sigma, y_sigma, rho], dim=2)


class AutoBotJoint(nn.Module):
    def __init__(self, d_k=128, M=5, c=5, T=30, L_enc=1, dropout=0.0, num_heads=16, L_dec=1, tx_hidden_size=384):
        super(AutoBotJoint, self).__init__()
        init_ = lambda m: init(m, nn.init.xavier_normal_, lambda x: nn.init.constant_(x, 0), np.sqrt(2))

        self.d_k = d_k
        self.M = M
        self.c = c
        self.T = T
        self.L_enc = L_enc
        self.dropout = dropout
        self.num_heads = num_heads
        self.L_dec = L_dec
        self.tx_hidden_size = tx_hidden_size

        # INPUT ENCODERS
        self.agents_dynamic_encoder = nn.Sequential(init_(nn.Linear(2, self.d_k)))

        self.social_attn_layers = []
        self.temporal_attn_layers = []
        for _ in range(self.L_enc):
            tx_encoder_layer = nn.TransformerEncoderLayer(d_model=self.d_k, nhead=self.num_heads, dropout=self.dropout,
            self.temporal_attn_layers.append(nn.TransformerEncoder(tx_encoder_layer, num_layers=1))
            tx_encoder_layer = nn.TransformerEncoderLayer(d_model=self.d_k, nhead=self.num_heads, dropout=self.dropout,
            self.social_attn_layers.append(nn.TransformerEncoder(tx_encoder_layer, num_layers=1))

        self.temporal_attn_layers = nn.ModuleList(self.temporal_attn_layers)
        self.social_attn_layers = nn.ModuleList(self.social_attn_layers)
```

```
        # DECODER MODELS
        self.Q = nn.Parameter(torch.Tensor(self.T, 1, self.c, 1, self.d_k), requires_grad=True)  # Decoder seed paramet
        nn.init.xavier_uniform_(self.Q)
        self.social_attn_decoder_layers = []
        self.temporal_attn_decoder_layers = []
        for _ in range(self.L_dec):
            tx_decoder_layer = nn.TransformerDecoderLayer(d_model=self.d_k, nhead=self.num_heads, dropout=self.dropout,
            self.temporal_attn_decoder_layers.append(nn.TransformerDecoder(tx_decoder_layer, num_layers=1))
            tx_encoder_layer = nn.TransformerEncoderLayer(d_model=self.d_k, nhead=self.num_heads, dropout=self.dropout,
            self.social_attn_decoder_layers.append(nn.TransformerEncoder(tx_encoder_layer, num_layers=1))

        self.temporal_attn_decoder_layers = nn.ModuleList(self.temporal_attn_decoder_layers)
        self.social_attn_decoder_layers = nn.ModuleList(self.social_attn_decoder_layers)
        self.pos_encoder = PositionalEncoding(self.d_k, dropout=0.0)

        # OUTPUT MODEL
        self.output_model = OutputModel(hidden_size=self.d_k)

        # Mode Prediction Models
        self.P = nn.Parameter(torch.Tensor(1, self.c, 1, self.d_k), requires_grad=True)
        nn.init.xavier_uniform_(self.P)
        self.prob_decoder = nn.TransformerDecoderLayer(d_model=self.d_k, nhead=self.num_heads, activation="relu", dim_f
        self.prob_predictor = init_(nn.Linear(self.d_k, 1))
        self.train()

    def process_observations(self, in_set_seqs):
        '''
        :param in_set_seqs: (B, T, M, k+1)
        where k is the number of attributes; here it's (x,y, mask).
        '''
        in_set_tensors = in_set_seqs[:, :, :, :2]
        in_set_masks = (1.0 - in_set_seqs[:, :, :, 2]).type(torch.BoolTensor).to(in_set_seqs.device)
        return in_set_tensors, in_set_masks

    def generate_decoder_mask(self, seq_len, device):
        ''' For masking out the subsequent info. '''
        subsequent_mask = (torch.triu(torch.ones((seq_len, seq_len), device=device), diagonal=1)).bool()
        return subsequent_mask

    def temporal_attn_fn(self, agents_emb, agent_masks, layer):
        '''
        :param agents_emb: (t, B, M, d_k)
        :param agent_masks: (B, t, M)
        :return: (t, B, M, d_k)
        '''
        t = agents_emb.size(0)
        B = agent_masks.size(0)
        agent_masks = agent_masks.permute(0, 2, 1).reshape(-1, t)
        agent_masks[:, -1][agent_masks.sum(-1) == t] = False  # Ensures that agent's that don't exist don't cause NaNs.
        agents_temp_emb = layer(self.pos_encoder(agents_emb.reshape(t, B * (self.M), -1)), src_key_padding_mask=agent_m
        return agents_temp_emb.view(t, B, self.M, -1)

    def social_attn_fn(self, agents_emb, agent_masks, layer):
        '''
        :param agents_emb: (t, B, M, d_k)
        :param agent_masks: (B, t, M)
        :return: (t, B, M, d_k)
        '''
        t = agents_emb.size(0)
        B = agent_masks.size(0)
        agents_emb = agents_emb.permute(2, 1, 0, 3).reshape(self.M, B * t, -1)
        agents_soc_emb = layer(agents_emb, src_key_padding_mask=agent_masks.view(-1, self.M))
        agents_soc_emb = agents_soc_emb.view(self.M, B, t, -1).permute(2, 1, 0, 3)
        return agents_soc_emb

    def temporal_attn_decoder_fn(self, agents_emb, context, agent_masks, layer):
        '''
        :param agents_emb: (T, Bc, M, d_k)
        :param context: (t, Bc, M, d_k)
        :param agent_masks: (Bc, T, M)
        :return: (T, Bc, M, d_k)
        '''
        t = context.size(0)
        Bc = agent_masks.size(0)
        time_masks = self.generate_decoder_mask(seq_len=self.T, device=agents_emb.device)
        agent_masks = agent_masks.permute(0, 2, 1).reshape(-1, t)
        agent_masks[:, -1][agent_masks.sum(-1) == t] = False  # Ensure that agent's that don't exist don't make NaN.
        agents_emb = agents_emb.reshape(self.T, -1, self.d_k)  # [T, BxcxM, d_k]
        context = context.view(-1, Bc*self.M, self.d_k)

        agents_temp_emb = layer(agents_emb, context, tgt_mask=time_masks, memory_key_padding_mask=agent_masks)
        agents_temp_emb = agents_temp_emb.view(self.T, Bc, self.M, -1)

        return agents_temp_emb

    def social_attn_decoder_fn(self, agents_emb, agent_masks, layer):
        '''
        :param agents_emb: (T, Bc, M, d_k)
        :param agent_masks: (Bc, T, M)
        :return: (T, Bc, M, d_k)
        '''
        Bc = agent_masks.size(0)
        agent_masks = agent_masks[:, -1:].repeat(1, self.T, 1).view(-1, self.M)  # take last timestep of all agents.
        agents_emb = agents_emb.permute(2, 1, 0, 3).reshape(self.M, Bc * self.T, -1)
```

```
        agents_soc_emb = layer(agents_emb, src_key_padding_mask=agent_masks)
        agents_soc_emb = agents_soc_emb.view(self.M, Bc, self.T, -1).permute(2, 1, 0, 3)
        return agents_soc_emb

    def forward(self, in_set_seqs):
        B = in_set_seqs.size(0)  # batch_size
        t = in_set_seqs.size(1)

        # Encode all input observations
        in_set_tensors, in_set_masks = self.process_observations(in_set_seqs)
        in_sets_emb = self.agents_dynamic_encoder(in_set_tensors).permute(1, 0, 2, 3)  # element-wise MLP

        for i in range(self.L_enc):
            in_sets_emb = self.temporal_attn_fn(in_sets_emb, in_set_masks, layer=self.temporal_attn_layers[i])
            in_sets_emb = self.social_attn_fn(in_sets_emb, in_set_masks, layer=self.social_attn_layers[i])

        # Repeat the tensors for the number of modes.
        in_set_masks_modes = in_set_masks.unsqueeze(1).repeat(1, self.c, 1, 1).view(B*self.c, t, -1)
        context = in_sets_emb.unsqueeze(2).repeat(1, 1, self.c, 1, 1).view(t, B*self.c, self.M, self.d_k)

        # Decoding
        dec_parameters = self.Q.repeat(1, B, 1, self.M, 1).view(self.T, B*self.c, self.M, -1)
        for i in range(self.L_dec):
            dec_parameters = self.temporal_attn_decoder_fn(dec_parameters, context, in_set_masks_modes, layer=self.temp
            dec_parameters = self.social_attn_decoder_fn(dec_parameters, in_set_masks_modes, layer=self.social_attn_dec

        out_dists = self.output_model(dec_parameters.reshape(self.T, -1, self.d_k))
        out_dists = out_dists.reshape(self.T, B, self.c, self.M, -1).permute(2, 0, 1, 3, 4)

        # Mode prediction
        mode_params_emb = self.P.repeat(B, 1, self.M, 1).transpose(0, 1).reshape(self.c, -1, self.d_k)
        mode_par_masks = torch.eye(self.c).to(in_set_seqs.device)
        mode_probs = self.prob_decoder(mode_params_emb, in_sets_emb.reshape(-1, B*self.M, self.d_k), tgt_mask=mode_par_
        mode_probs = self.prob_predictor(mode_probs).squeeze(-1).view(B, self.M, -1).sum(1)
        mode_probs = F.softmax(mode_probs, dim=1)

        # return  # [c, T, B, M, 5], [B, c]
        return out_dists, mode_probs
```

Loss Functions

We define some utility functions for calculating the multimodal loss of the generated characters.

In [38]:

```
import torch
from scipy import special
import torch.distributions as D
from torch.distributions import MultivariateNormal


def get_BVG_distributions(pred):
    B = pred.size(0)
    T = pred.size(1)
    N = pred.size(2)
    mu_x = pred[:, :, :, 0].unsqueeze(3)
    mu_y = pred[:, :, :, 1].unsqueeze(3)
    sigma_x = pred[:, :, :, 2]
    sigma_y = pred[:, :, :, 3]
    rho = pred[:, :, :, 4]

    cov = torch.zeros((B, T, N, 2, 2)).to(pred.device)
    cov[:, :, :, 0, 0] = sigma_x ** 2
    cov[:, :, :, 1, 1] = sigma_y ** 2
    cov_val = rho * sigma_x * sigma_y
    cov[:, :, :, 0, 1] = cov_val
    cov[:, :, :, 1, 0] = cov_val

    biv_gauss_dist = MultivariateNormal(loc=torch.cat((mu_x, mu_y), dim=-1), covariance_matrix=cov)
    return biv_gauss_dist


def nll_pytorch_dist(pred, data, agents_masks):
    biv_gauss_dist = get_BVG_distributions(pred)
    num_active_agents_per_timestep = agents_masks.sum(2)
    loss = (((-biv_gauss_dist.log_prob(data) * agents_masks).sum(2)) / num_active_agents_per_timestep).sum(1)
    return loss


def nll_loss_multimodes(pred, agents_data, modes_pred, entropy_weight=1.0, kl_weight=1.0):
    gt_agents = agents_data[:, :, :, :2]
    modes = len(pred)
    nSteps, batch_sz, M, dim = pred[0].shape

    agents_masks = agents_data[:, :, :, 2]

    modes_pred = modes_pred
    log_lik = np.zeros((batch_sz, modes))

    with torch.no_grad():
        for kk in range(modes):
            nll = nll_pytorch_dist(pred[kk].transpose(0, 1), gt_agents, agents_masks)
            log_lik[:, kk] = -nll.cpu().numpy()
```

```
    priors = modes_pred.detach().cpu().numpy()

    log_posterior_unnorm = log_lik + np.log(priors)
    log_posterior = log_posterior_unnorm - special.logsumexp(log_posterior_unnorm, axis=1).reshape((batch_sz, 1))
    post_pr = np.exp(log_posterior)

    post_pr = torch.tensor(post_pr).float().to(gt_agents.device)
    post_entropy = torch.mean(D.Categorical(post_pr).entropy()).item()

    loss = 0.0
    for kk in range(modes):
        nll_k = nll_pytorch_dist(pred[kk].transpose(0, 1), gt_agents, agents_masks) * post_pr[:, kk]
        loss += nll_k.sum() / float(batch_sz*M)

    kl_loss = torch.nn.KLDivLoss(reduction='batchmean')  # type: ignore
    loss += kl_weight*kl_loss(torch.log(modes_pred), post_pr)

    entropy_vals = []
    for kk in range(modes):
        entropy_vals.append(get_BVG_distributions(pred[kk]).entropy())
    entropy_loss = torch.mean(torch.stack(entropy_vals).permute(2, 0, 3, 1).sum(3).mean(2).max(1)[0])
    loss += entropy_weight*entropy_loss

    return loss, post_entropy
```

Training Loop

The training loop takes about 45-60 minutes on a single-GPU machine.

In [ ]:

```
import torch
from torch import optim
import torch.nn as nn
import time


# Model parameters
d_k = 128
c = 4
M = 12
T = 10
L_enc = 1
L_dec = 1
dropout = 0.2

# Training parameters
batch_size = 64
learning_rate = 5e-4
adam_epsilon = 1e-4
entropy_weight = 1.0
kl_weight = 1.0
num_epochs = 40


if torch.cuda.is_available():
    device = torch.device("cuda")
    torch.cuda.manual_seed(0)
else:
    device = torch.device("cpu")


# Defining models
autobot_model = AutoBotJoint(d_k=d_k, M=M, c=c, T=T, L_enc=L_enc, L_dec=L_dec, dropout=dropout).to(device)
optimiser = optim.Adam(autobot_model.parameters(), lr=learning_rate, eps=adam_epsilon)

# Initialize dataloader
dset = OmniGlotDataset(dset_path=".", split_name="train")
print("Number of Characters:", len(dset))
train_loader = torch.utils.data.DataLoader(dset, batch_size=batch_size, shuffle=True, num_workers=2, drop_last=False, p

start_time = time.time()
total_steps = 0
losses = []
for train_iter in range(0, num_epochs):
    print("Epoch:", train_iter, "Entropy Weight:", entropy_weight, "KL weight:", kl_weight)
    print("time since start:", (time.time() - start_time) / 60.0, "minutes.")
    for i, data in enumerate(train_loader):
        in_set_seqs, out_set_seqs, _ = data
        in_set_seqs = in_set_seqs.float().to(device)
        out_set_seqs = out_set_seqs.float().to(device)

        # Run through model.
        pred_obs, mode_probs = autobot_model(in_set_seqs)

        # Compute loss
        loss, post_entropy = nll_loss_multimodes(pred_obs, out_set_seqs, mode_probs, entropy_weight=entropy_weight, kl_
        sigmas = pred_obs[:, :, :, :, 2:4]
        sigma_magnitude = torch.mean(torch.norm(sigmas, dim=-1))

        # Backprop
        optimiser.zero_grad()
```

```
        loss.backward()
        nn.utils.clip_grad_norm_(autobot_model.parameters(), 5.0)
        optimiser.step()

        losses.append([loss.item()])
        if i % 5 == 0:
            print(i, "Obs_Loss", losses[-1][0], "Sigma Magnitude", sigma_magnitude.item())
```

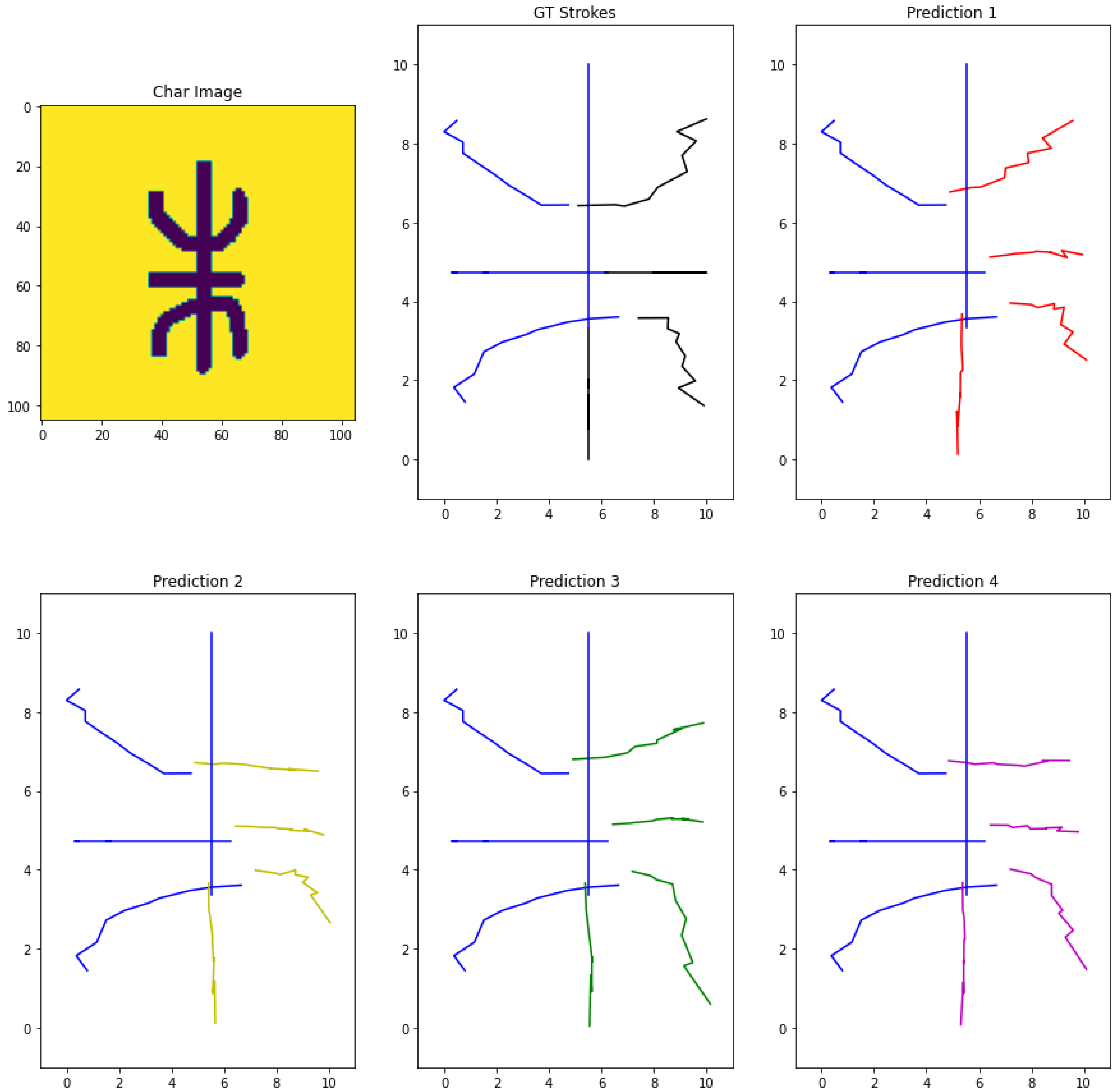
Testing Learned Model

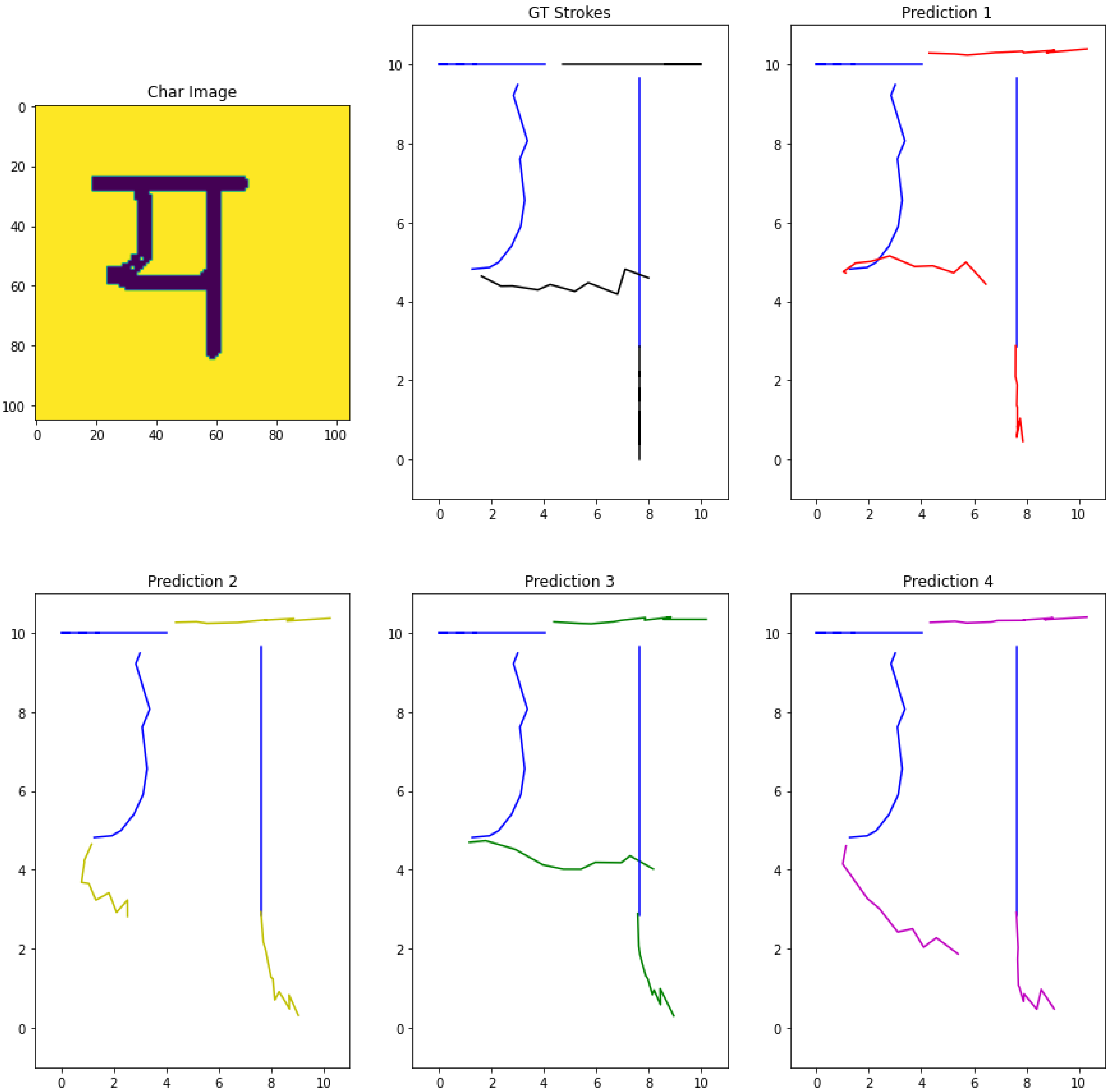
This consistutes the results shown in Figures 3 (left), 13-14 of the paper.

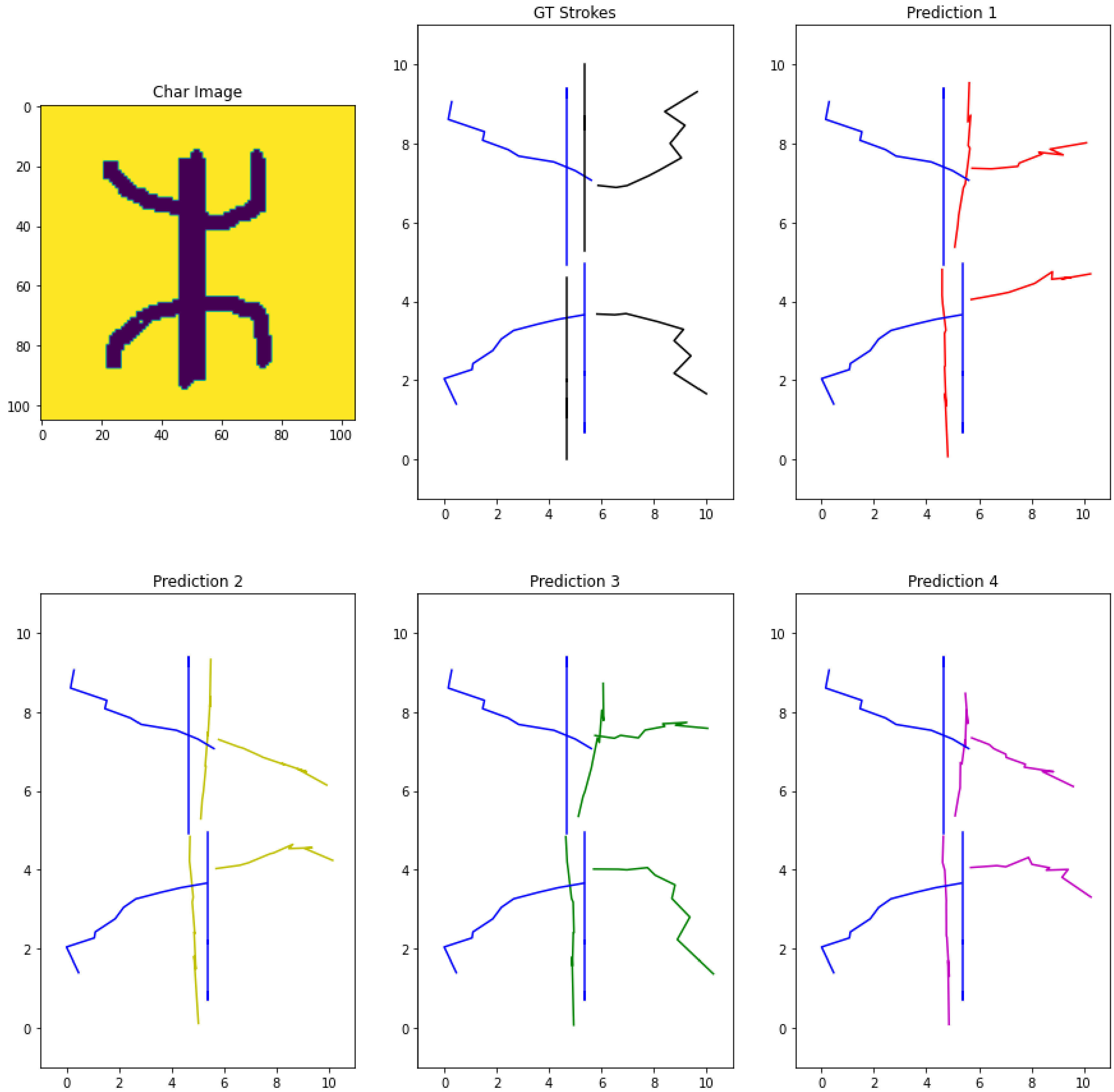
In [44]:

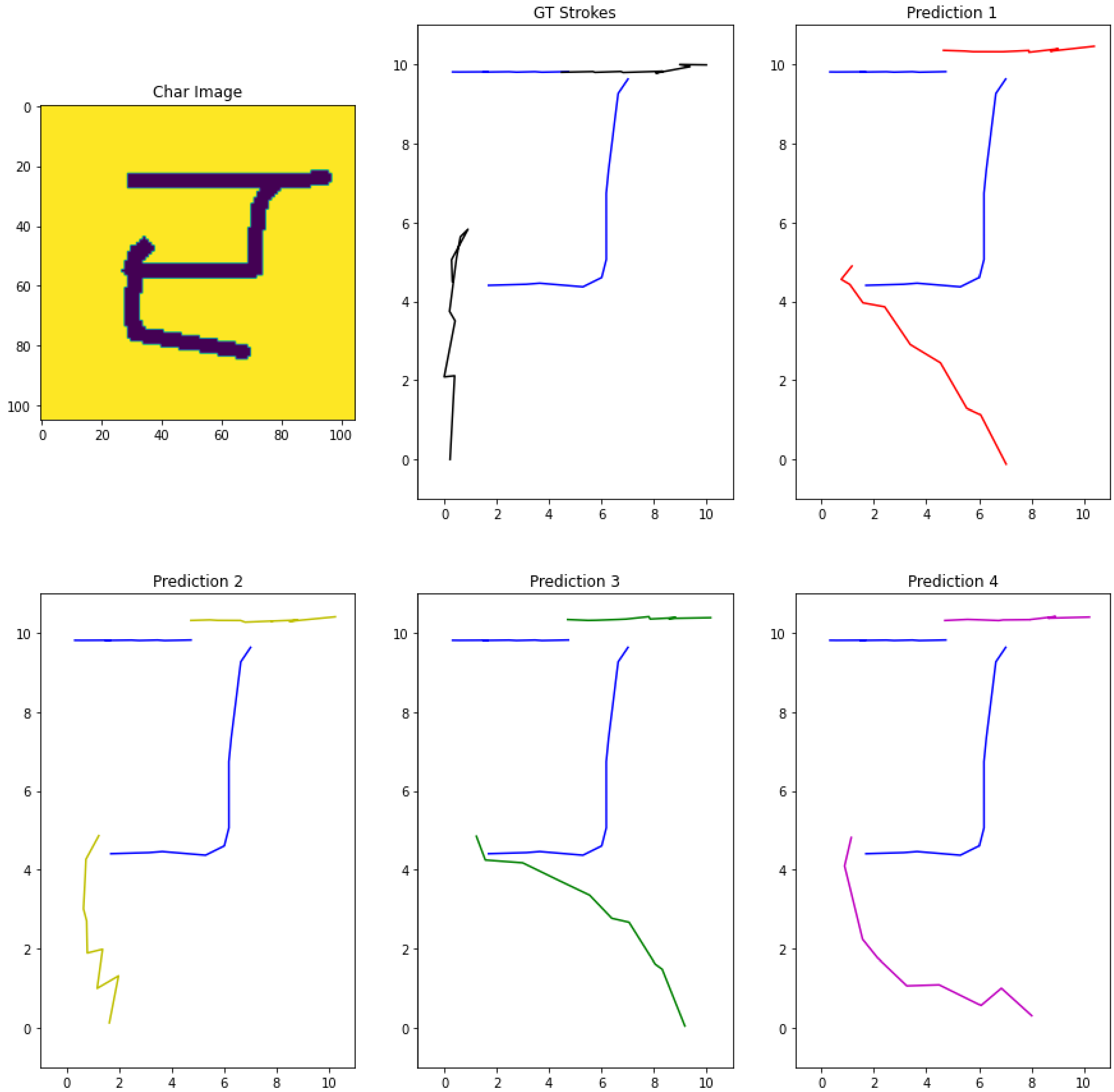
```
import os
from matplotlib import pyplot as plt
import matplotlib.image as mpimg

%matplotlib inline

autobot_model.eval()
dset = OmniGlotDataset(dset_path=".", split_name='test')
test_loader = torch.utils.data.DataLoader(
    dset, batch_size=batch_size, shuffle=True, num_workers=12, drop_last=False, pin_memory=True
)
pred_colors = ['r', 'y', 'g', 'm']
with torch.no_grad():
    for i, data in enumerate(test_loader):
        in_set_seqs, out_set_seqs, fname = data
        in_set_seqs = in_set_seqs.float().to(device)
        out_set_seqs = out_set_seqs.float().to(device)
        pred_obs, mode_probs = autobot_model(in_set_seqs)
        img_fname = fname[0].replace(".txt", ".png").replace("strokes", "images")
        char_image = mpimg.imread(img_fname)

        num_strokes = int(in_set_seqs[0, 0, :, 2].sum())
        gt_past = in_set_seqs[0].cpu().numpy()
        gt_future = out_set_seqs[0].cpu().numpy()
        prediction = pred_obs[:, :, 0].cpu().numpy()

        fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(15,15))
        ax[0, 0].imshow(char_image)
        ax[0,0].title.set_text('Char Image')
        for m in range(num_strokes):
            ax[0, 1].plot(gt_past[:, m, 0], gt_past[:, m, 1], color='b')

        for m in range(num_strokes):
            ax[0, 1].plot(gt_future[:, m, 0], gt_future[:, m, 1], color='k')
        ax[0, 1].axis(xmin=-1, xmax=11, ymin=-1, ymax=11)
        ax[0, 1].title.set_text('GT Strokes')

        row = 0
        for k in range(c):
            col = (k+2) % 3
            if (k + 2) % 3 == 0:
                row += 1
            for m in range(num_strokes):
                ax[row, col].plot(gt_past[:, m, 0], gt_past[:, m, 1], color='b')
                ax[row, col].plot(prediction[k, :, m, 0], prediction[k, :, m, 1], color=pred_colors[k])

            ax[row, col].axis(xmin=-1, xmax=11, ymin=-1, ymax=11)
            ax[row, col].title.set_text('Prediction '+str(k+1))
        plt.show()
        if i == 5:
          break
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarning: This DataLoader will create 12
worker processes in total. Our suggested max number of worker in current system is 2, which is smaller than what this D
ataLoader is going to create. Please be aware that excessive worker creation might get DataLoader running slow or even
freeze, lower the worker number to avoid potential slowness/freeze if necessary.
  cpuset_checked))
```

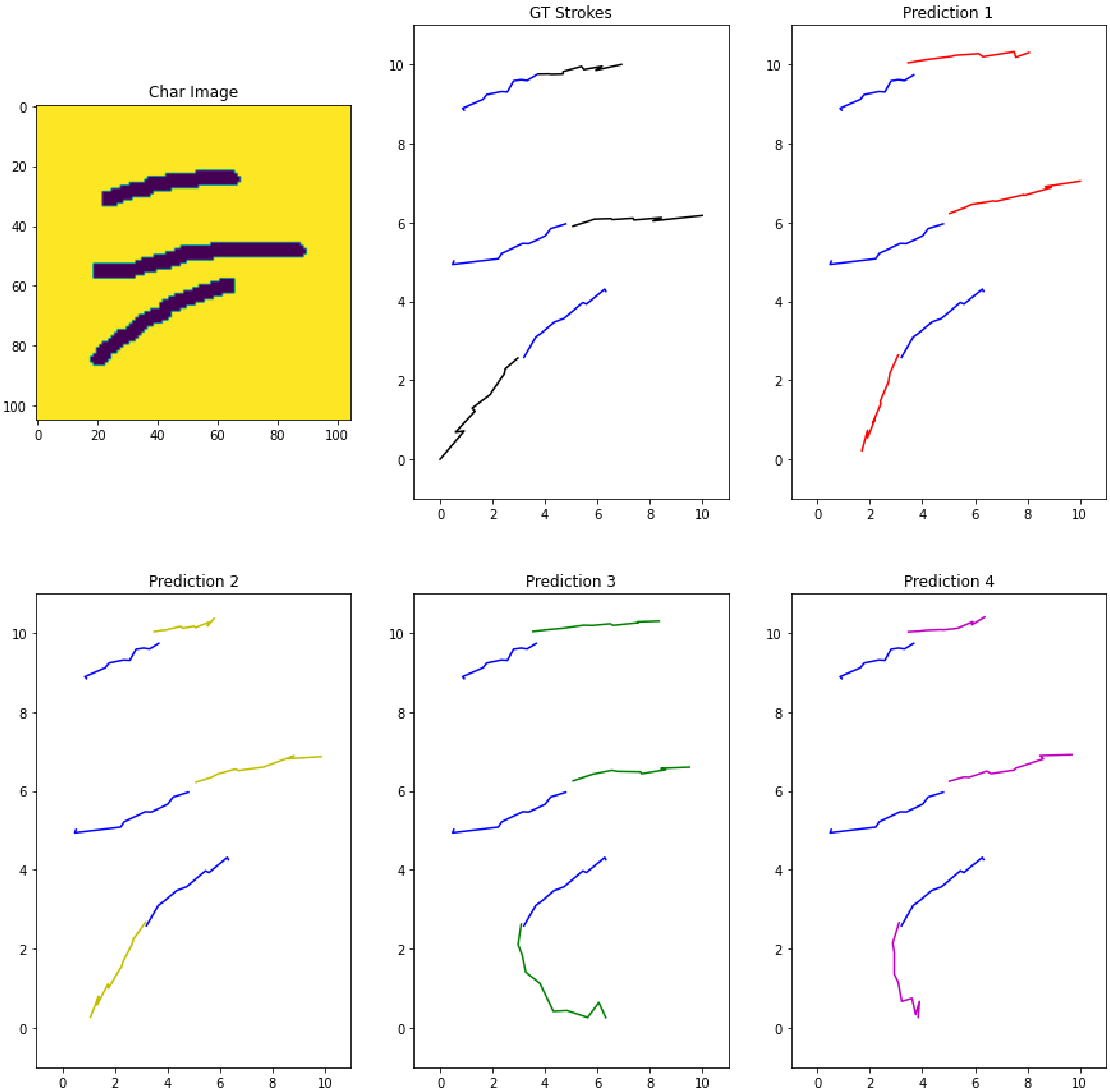

Char Image
GT Strokes
Prediction 1
Prediction 2
Prediction 3
Prediction 4

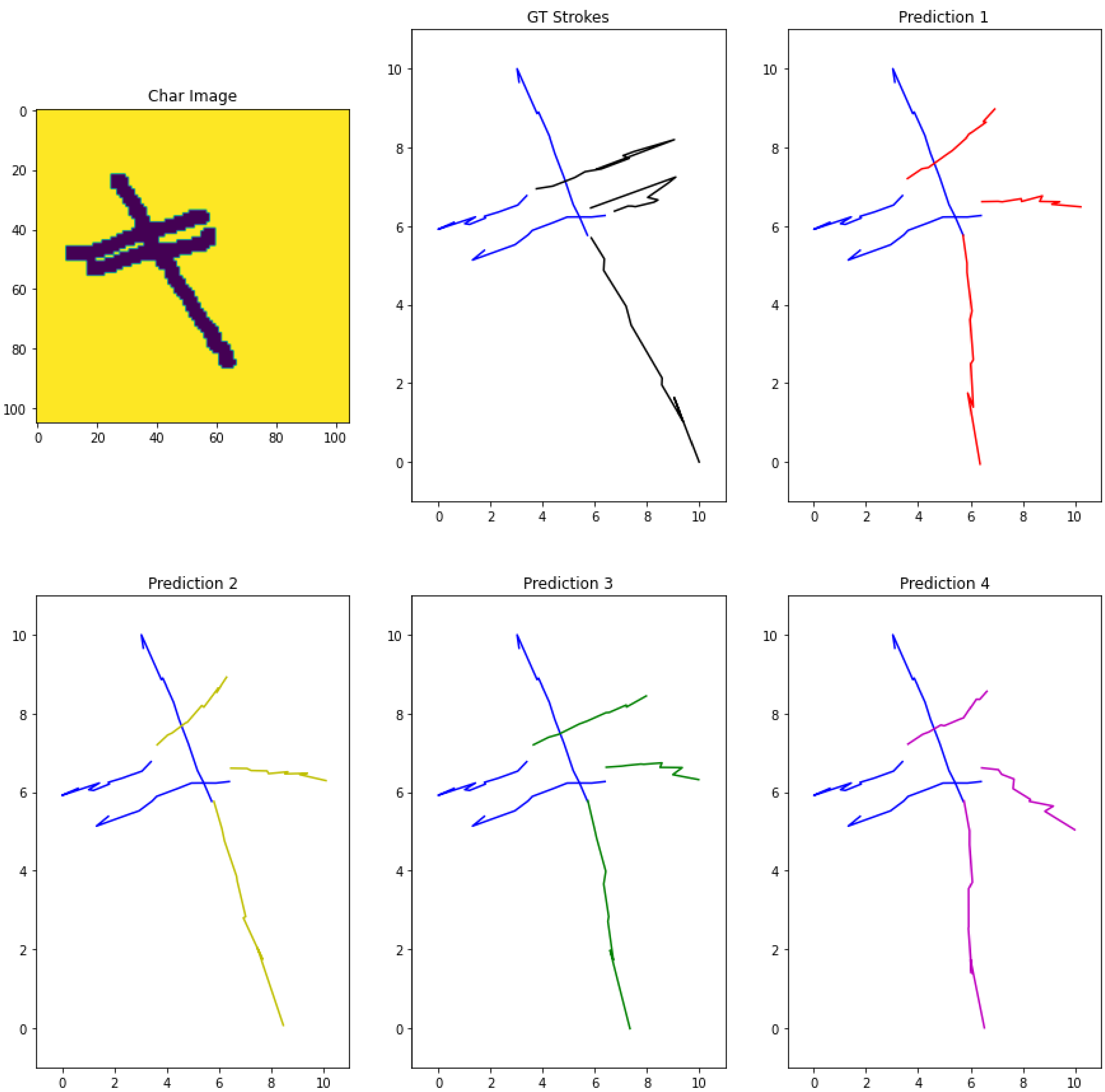

Char Image
GT Strokes
Prediction 1
Prediction 2
Prediction 3
Prediction 4


Char Image
GT Strokes
Prediction 1
Prediction 2
Prediction 3
Prediction 4
0
20
40
60
80
100
2
4
6
8
10



Char Image
GT Strokes
Prediction 1
Prediction 2
Prediction 3
Prediction 4
0
20
40
60
80
100
0
2
4
6
8
10


Char Image
GT Strokes
Prediction 1
Prediction 2
Prediction 3
Prediction 4


Char Image
0
20
40
60
80
100
0
20
40
60
80
100
GT Strokes
Prediction 1
Prediction 2
Prediction 3
Prediction 4
0
2
4
6
8
10