GMMs, 2 Ways

In the following we will optimize a Gaussian mixture model (GMM) using two optimization techniques: 1) directly optimizing the likelihood; and 2) using the expectation maximization (EM) algorithm.

Turn in by November 15th 11:59PM via https://docs.google.com/forms/d/e/1FAIpQLSfU2sqMhLeU0vj6vHzvWQb8iNb-abEJkfmhHeZd3fHKL7MSuA/viewform?usp=pp_url.

Data and Setup *(5 points)*

First, it may help to enable GPUs for the notebook:

- Navigate to Edit→Notebook Settings
- select GPU from the Hardware Accelerator drop-down

Next, confirm that we can connect to the GPU with tensorflow.

(Note, it is fine if you can not connect to GPU, it just might take a little longer to run.)

```
%tensorflow_version 2.x
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

```
---------------------------------------------------------------------------
SystemError                               Traceback (most recent call last)
<ipython-input-67-d1d5f2f639c4> in <module>()
      3 device_name = tf.test.gpu_device_name()
      4 if device_name != '/device:GPU:0':
----> 5   raise SystemError('GPU device not found')
      6 print('Found GPU at: {}'.format(device_name))

SystemError: GPU device not found
```

Let's import additional packages of use.

```
%matplotlib inline
import numpy as np
from scipy.stats import multivariate_normal
```

```
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
```

Next, we prescribe the ground truth parameters to generate data. Recall that the parameters are π_j, the mixing prior coefficients for components, μ_j, the means for components, and σ_j the standard deviation for components. π will be represented with logits in `gt_logits`; i.e. the softmax of `gt_logits` is π. μ is represented by `gt_means`. σ is represented in log space by `gt_lsigmas`; i.e. the exp of `gt_lsigmas` is σ.

```
gt_logits = tf.math.log([1/4, 1/4, 1/6, 1/6, 1/6])
gt_means = tf.convert_to_tensor([1.0, -0.5, -2, .5, 3])
gt_lsigmas = tf.math.log([.5, 1.0, .2, 0.1, .5])


#remove later
#print(tf.nn.softmax(gt_logits)) #pi
#print(tf.math.exp(gt_lsigmas))
```

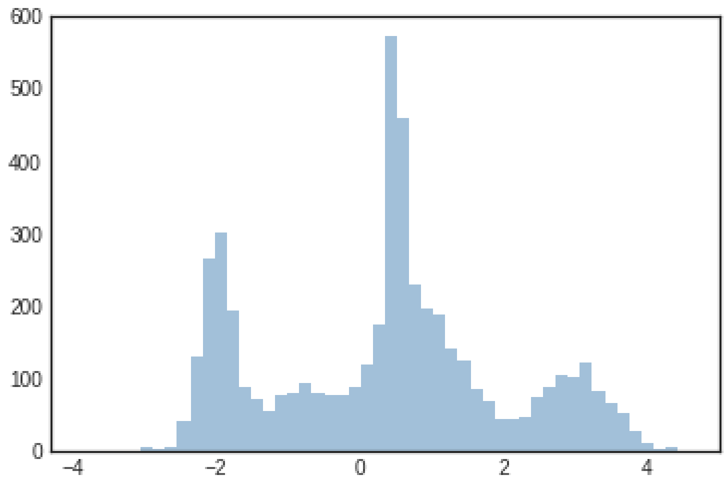
Sample data based on the parameters (5 points).

Fill in the code below to generate samples.

```
def make_data(N, logits, means, lsigmas):
  z = tf.transpose(tf.random.categorical([logits], N))
  y = tf.random.normal((N, 1))
  """
  hint: use tf.gather
  x = .... transform y to get sample
  """
  x = None  # TODO

  std = tf.exp(lsigmas)
  idx_m0 = tf.where(tf.equal(z, 0))[:, 0]
  idx_m1 = tf.where(tf.equal(z, 1))[:, 0]
  idx_m2 = tf.where(tf.equal(z, 2))[:, 0]
  idx_m3 = tf.where(tf.equal(z, 3))[:, 0]
  idx_m4 = tf.where(tf.equal(z, 4))[:, 0]

  x_m0 = (tf.gather(y, idx_m0)*std[0]) + means[0]
  x_m1 = (tf.gather(y, idx_m1)*std[1]) + means[1]
  x_m2 = (tf.gather(y, idx_m2)*std[2]) + means[2]
  x_m3 = (tf.gather(y, idx_m3)*std[3]) + means[3]
  x_m4 = (tf.gather(y, idx_m4)*std[4]) + means[4]

  x = tf.concat([x_m0, x_m1, x_m2, x_m3, x_m4], axis=0)
  return x
```

Now plot the data. (Hint: it should look something like this image.)

```
NUM_EXAMPLES = 5000
training_inputs = make_data(NUM_EXAMPLES, gt_logits, gt_means, gt_lsigmas)
plt.hist(np.reshape(training_inputs, [-1]), bins=50, alpha=0.5,
         histtype='stepfilled', color='steelblue', edgecolor='none');
```



Likelihood based GMMs (57 Points)

Here we shall optimize parameters of an estimated GMM directly with maximum likelihood estimation (MLE).

Likelihood function (25 points)

To do so, we need to write a function that computes the log-likelihood for inputs given our estimated parameters. To be numerically stable *you need to use* `tf.reduce_logsumexp` *(you will lose points if not)*.

```
def univariate_normal(x, mean, stddev):
    variance = stddev**2
    return ((1. / np.sqrt(2 * np.pi * variance)) * np.exp(-(x - mean)**2 / (2 * variar


def log_pdf(x, mean, stddev):
  variance = stddev**2
  return ((-0.5 * np.log(2 * np.pi * variance)) + (-(x - mean)**2 / (2 * variance)))


def mixture_likelihood(x, logits, means, lsigmas):
    """Given log-unnormalized mixture weights, shift, and log scale parameters
```

```
    for mixture components, return the likelihoods for targets.
    Args:
        x: N x 1 tensor of 1d targets to get likelihoods for.
        logits: ncomp tensor of mixing priors of mixture model.
        means: ncomp tensor of means of mixture model.
        lsigmas: ncomp tensor of log std. dev. of mixture model.
    Return:
        likelihoods: N x 1  tensor of likelihoods log p(x).
    """
    # Compute likelihoods per x
    # Write log likelihood with logsumexp.
    LL = tf.zeros(shape=(x.shape[0],0))
    clusters = logits.shape[0]
    stddevs = tf.math.exp(lsigmas)
    weights = tf.nn.softmax(logits)
    for i in range(clusters):
      #x_temp = univariate_normal(x, means[i], stddevs[i])
      x_temp = log_pdf(x, means[i], stddevs[i])
      x_temp = tf.math.log(weights[i]) + x_temp
      #LL = tf.concat([LL, weights[i] * x_temp], axis=-1)
      LL = tf.concat([LL, x_temp], axis=-1)
    #LL = tf.math.log(LL)
    LL = tf.reduce_logsumexp(input_tensor = LL, axis=1)
    return LL  # TODO
```

Let's plot our likelihood using the ground truth parameters. The likelihood should match up to the histogram above.

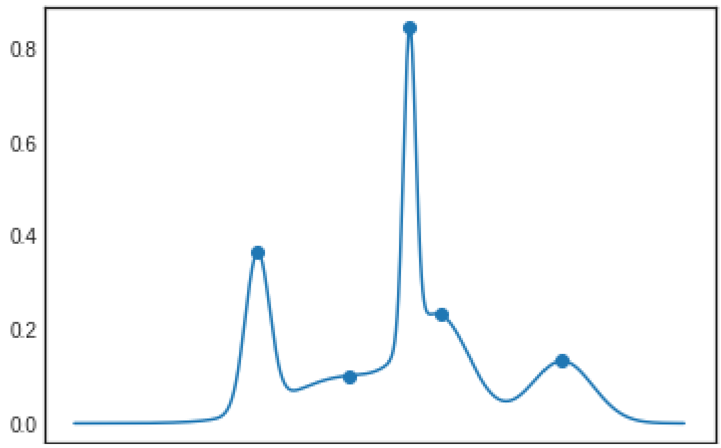
```
def plot_density(logits, means, lsigmas):
  gridx = np.reshape(np.linspace(-5.0, 5.0, 1000), [-1, 1])

  log_px = mixture_likelihood(gridx, logits, means, lsigmas)
  #print(np.mean(np.exp(log_px)))
  plt.plot(gridx, np.exp(log_px))
  plt.scatter(tf.reshape(means, [-1, 1]), np.exp(mixture_likelihood(tf.reshape(means,
```

Side question (4 points): What is the expected value of `np.mean(np.exp(log_px))` above? *Explain why.*

TODO

Answer: 0.09989941 This point can explain maximum of the data points.

```
plot_density(gt_logits, gt_means, gt_lsigmas)
```



Optimization

We will now optimize a model based on its likelihood.

Model (15 points)

First, let's implement a keras model for GMMs.

```
class GMM(tf.keras.Model):
  def __init__(self, k):
    super(GMM, self).__init__()
    """
    Hint: it helps to initialize variables close to zero with a small range
    (about 0.1 standard deviation).

    self.logits = tf.Variable( TODO , name='logits')
    self.means = tf.Variable( TODO, name='means')
    self.lsigmas = tf.Variable( TODO, name='lsigmas')
    """
    logits = tf.math.log(tf.random.uniform(shape=[k, 1], maxval=0.1))
    means = tf.convert_to_tensor(tf.random.uniform(shape=[k, 1], minval= -0.1, maxval=
    lsigmas = tf.math.log(tf.random.uniform(shape=[k, 1], maxval=0.1))

    temp = np.array(np.random.dirichlet(np.ones(k), size=1).transpose())
    s = np.sum(temp)
    temp = temp/s

    self.logits = tf.Variable(shape=(k, 1),
                              initial_value = np.log(temp.astype(np.float32)),
                              name='logits')  # TODO
    self.means = tf.Variable(shape=(k, 1),
                             initial_value = means, #(np.random.randn(k, 1)*0.1).astyp
                             name='means')  # TODO
    self.lsigmas = tf.Variable(shape=(k, 1),
                               initial_value = lsigmas, #np.log(np.abs((np.random.ranc
                               name='lsigmas')  # TODO
  def call(self, inputs):
```

```
    """
    Hint: what should the model return? It should have the same length as
    inputs.
    """
    log_px = mixture_likelihood(inputs, self.logits, self.means, self.lsigmas)
    return log_px  # TODO
```

Loss (4 points)

Now we'll implement the loss to *minimize* according to gradients.

```
def loss(model, inputs):
  """
  Hint: return some_function_of(model(inputs)).
  """
  #print(model(inputs))
  return -tf.reduce_sum(model(inputs))  # TODO

def grad(model, inputs):
  with tf.GradientTape() as tape:
      loss_value = loss(model, inputs)
  grads = tape.gradient(loss_value, model.trainable_variables)
  return grads


#remove later
```

Train (4 points)

Let's train using our training data. Note, you may want to run this several times to observe differences in the resulting model. (No need for minibatches.)

```
K = 5
model = GMM(K)
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

print("Initial loss: {:.3f}".format(loss(model, training_inputs)))
plot_density(model.logits, model.means, model.lsigmas)

#print(model.summary())
#print(training_inputs)
#print(model.logits, model.means, model.lsigmas)

steps = 3000
for i in range(steps):
  """
```

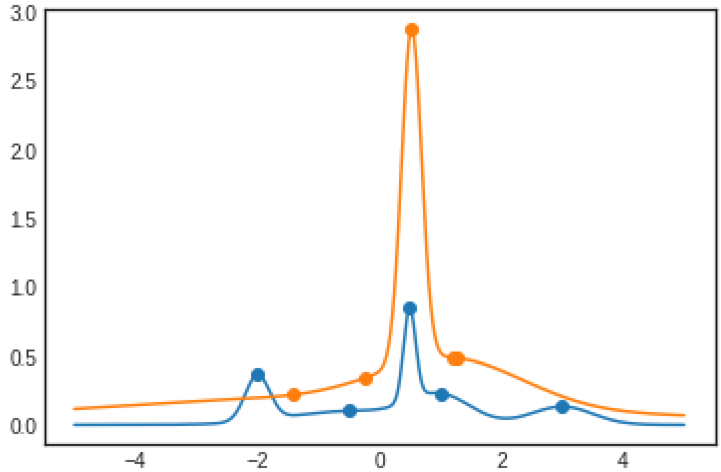
```
    Hint:
    grads = something with training_inputs...
    """
    grads = grad(model, training_inputs)  # TODO

    optimizer.apply_gradients(zip(grads, model.trainable_variables))
    if i % 100 == 0:
      print("Loss at step {:03d}: {:.3f}".format(i, loss(model, training_inputs)))
      plot_density(model.logits, model.means, model.lsigmas)
  #print(training_inputs)
  #print(model.logits, model.means, model.lsigmas)
```

```
Initial loss: 1137574.250
Loss at step 000: 1112341.625
```

Let's plot our estimate versus the ground truth to see how well they match up.

```
Loss at step 400: 3062.073
```

```
plot_density(gt_logits, gt_means, gt_lsigmas)
plot_density(model.logits, model.means, model.lsigmas)
```



Side question (5 points): Why does it not make sense to compare the MSE of `model.means` versus `gt_means`?

```
Loss at step 2800: 3387.703
```

TODO Answer: Because of the nature that GMM does soft assignments to clusters as opposed to hard assignments. Also, MSE is a good choice for convex cost function but the likelihood of GMM is non-convex.

EM GMMs (38 Points)

Now we will train a GMM using the expectation maximation algorithm.

Implement posterior (15 points)

Implement the function which will compute the posterior of latent component indicators z_i's given GMM parameters.

```
def mixture_posterior_logits(x, logits, means, lsigmas):
  """Given log-unnormalized mixture weights, shift, and log scale parameters
  for mixture components, return the posterior of latent z_i's.
  Args:
      x: N x 1 tensor of 1d targets to get posteriors for.
      logits: ncomp tensor of mixing priors of mixture model
```

```
      means: ncomp tensor of means of mixture model.
      lsigmas: ncomp tensor of log std. dev. of mixture model.
  Return:
      posterior_logits: N x ncomp  tensor logits for posterior of z_i's.
  """
  total = np.zeros(shape = (x.shape[0], 1), dtype = np.float64)
  clusters = logits.shape[0]
  gamma_znk = np.zeros(shape=(x.shape[0], 0), dtype=np.float64)
  stddevs = tf.math.exp(lsigmas)
  weights = tf.nn.softmax(logits)
  for i in range(clusters):
    gamma = log_pdf(x, means[i], stddevs[i])
    weighted_gamma = np.log(weights[i]) + gamma
    total = total + weighted_gamma
    gamma_znk = np.concatenate((gamma_znk, weighted_gamma), axis = -1)
  total = tf.reduce_logsumexp(gamma_znk, axis = 1, keepdims=True)
  gamma_znk = gamma_znk - total
  gamma_znk = np.exp(gamma_znk)
  return gamma_znk  # TODO
```

Implement M step (15 points)

Next, we implement the M step, which will update the parameters of the GMM given a current posterior.

```
def mstep(x, posterior):
  """Given log-unnormalized mixture weights, shift, and log scale parameters
  for mixture components, return the posterior of latent z_i's.
  Args:
      x: N x 1 tensor of 1d samples.
      posterior_logits: N x ncomp  tensor of posterior for z_i's; i.e. sums to
        1 along the second axis.
  Return:
      logits: ncomp tensor of new mixing priors of mixture model
      means: ncomp tensor of new means of mixture model.
      lsigmas: ncomp tensor of new log std. dev. of mixture model.
  """
  denominator = np.sum(posterior, axis=0)

  weights = np.sum(posterior, axis=0) / float(posterior.shape[0])
  logits = np.log(weights)
  logits = np.reshape(logits, [-1, 1])
  #print(logits.shape)

  means = np.sum((posterior * x), axis=0) / denominator
  means = np.reshape(means, [-1, 1])
  #print(means.shape)

  variances = np.sum(posterior * ((x - means.T)**2) , axis=0) / denominator
  stddevs = np.sqrt(variances)
```

```
    lsigmas = np.log(stddevs)
    lsigmas = np.reshape(lsigmas, [-1, 1])
    #print(lsigmas.shape)

    return logits, means, lsigmas  # TODO
```

Implement EM Model (8 Points)

Based on the above functions, we now implement a model that updates parameters using the E and M steps when called.

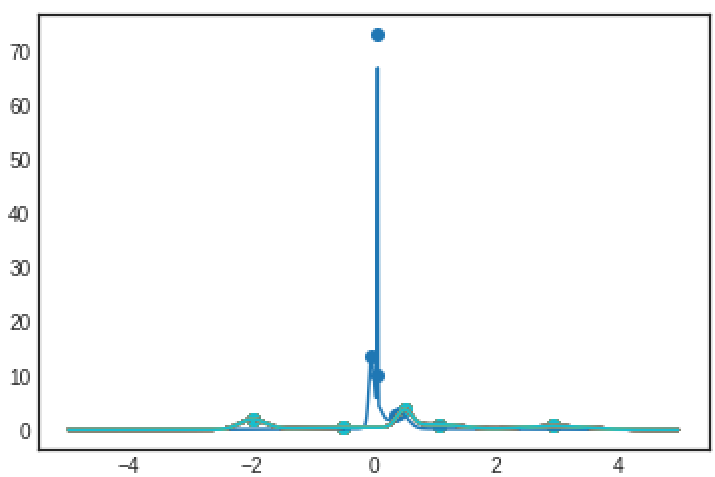
```
class EMGMM(tf.keras.Model):
  def __init__(self, k):
    super(EMGMM, self).__init__()
    """
    Hint: You should be able to initialize as above.
    *Note the trainable=False.*
    self.logits = tf.Variable( TODO , name='logits', trainable=False)
    self.means = tf.Variable( TODO, name='means', trainable=False)
    self.lsigmas = tf.Variable( TODO, name='lsigmas', trainable=False)
    """

    logits = tf.math.log(tf.random.uniform(shape=[k, 1], maxval=0.1))
    means = tf.convert_to_tensor(tf.random.uniform(shape=[k, 1], minval= -0.1, maxval=
    lsigmas = tf.math.log(tf.random.uniform(shape=[k, 1], maxval=0.1))

    temp = np.array(np.random.dirichlet(np.ones(k), size=1).transpose())
    s = np.sum(temp)
    temp = temp/s

    self.logits = tf.Variable(shape = (k, 1),
                              initial_value = np.log(temp.astype(np.float32)),
                              name = 'logits',
                              trainable = False)   # TODO
    self.means = tf.Variable(shape = (k, 1),
                             initial_value = means,
                             name = 'means',
                             trainable = False)  # TODO
    self.lsigmas = tf.Variable(shape = (k, 1),
                               initial_value = lsigmas,
                               name = 'lsigmas',
                               trainable = False)  # TODO

  def call(self, inputs):
    """
    E step
    Hint: think about logits versus probabilities.
    posterior = ...
    """
    posterior = None  # TODO
```

```
    posterior = mixture_posterior_logits(inputs, self.logits, self.means, self.lsigmas
    """
    M step
    """
    new_logits, new_means, new_lsigmas = (None, None, None)  # TODO
    new_logits, new_means, new_lsigmas = mstep(inputs, posterior)  # TODO

    self.logits.assign(new_logits)
    self.means.assign(new_means)
    self.lsigmas.assign(new_lsigmas)
    return None  # Shouldn't return anything
```

Optimization

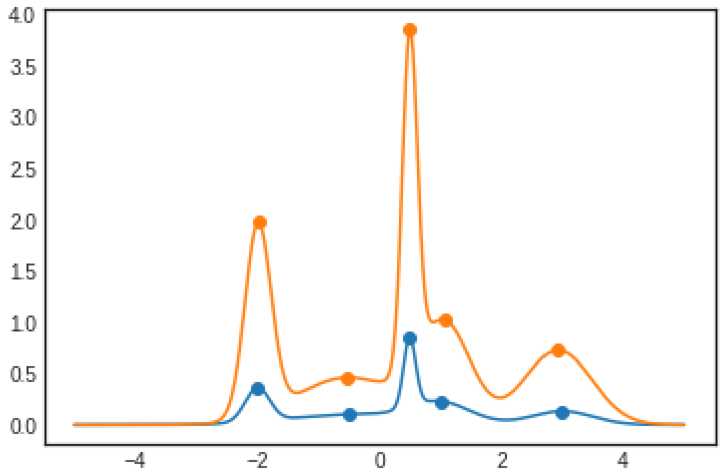
Train the model as follows.

```
K = 5
model = EMGMM(K)

steps = 3000
for i in range(steps):
  model(training_inputs)
  if i % 100 == 0:
    plot_density( model.logits, model.means, model.lsigmas)
```



Compare to ground truth.

```
plot_density(gt_logits, gt_means, gt_lsigmas)
plot_density(model.logits, model.means, model.lsigmas)
```



✓ 0s completed at 3:36 PM