Problem 2: Language Modeling with RNNs

- **Learning Objective:** In this problem, you are going to implement simple recurrent neural networks to deeply understand how RNNs works.
- **Provided Code:** We provide the skeletons of classes you need to complete. Forward checking and gradient checkings are provided for verifying your implementation as well.
- **TODOs:** you will firstly implement a vanilla RNN to warm up, and then implement an LSTM to train a model that can generate text using your own text source (novel, lyrics etc).

In [5]:
```
from lib.rnn.rnn import *
from lib.rnn.layer_utils import *
from lib.rnn.train import *
from lib.grad_check import *
from lib.optim import *
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipythor
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Recurrent Neural Networks

We will use recurrent neural network (RNN) language models for text generation.

Please complete the TODOs in the function `VanillaRNN` of the file `lib/rnn/layer_utils.py` which should contain implementations of different layer types that are needed for recurrent neural networks.

And then, complete the TODOs in the file `lib/rnn/rnn.py` which uses these layers to implement a text generation model.

Vanilla RNN: step forward (4 Pts)

Open the file `lib/rnn/layer_utils.py` . Implement the forward and backward passes for different types of layers that are commonly used in recurrent neural networks.

First complete the implementation of the function `step_forward` which implements the forward pass for a single timestep of a vanilla recurrent neural network.

After doing so run the following code. You should see errors less than 1e-8.

In [2]:
```
%reload_ext autoreload

N, D, H = 3, 10, 4

rnn = VanillaRNN(D, H, init_scale=0.02, name="rnn_test")
x = np.linspace(-0.4, 0.7, num=N*D).reshape(N, D)
prev_h = np.linspace(-0.2, 0.5, num=N*H).reshape(N, H)

rnn.params[rnn.wx_name] = np.linspace(-0.1, 0.9, num=D*H).reshape(D, H)
rnn.params[rnn.wh_name] = np.linspace(-0.3, 0.7, num=H*H).reshape(H, H)
rnn.params[rnn.b_name] = np.linspace(-0.2, 0.4, num=H)

next_h, _ = rnn.step_forward(x, prev_h)
expected_next_h = np.asarray([
  [-0.58172089, -0.50182032, -0.41232771, -0.31410098],
  [ 0.66854692,  0.79562378,  0.87755553,  0.92795967],
  [ 0.97934501,  0.99144213,  0.99646691,  0.99854353]])

print('next_h error: ', rel_error(expected_next_h, next_h))
```

```
next_h error:  6.292421426471037e-09
```

Vanilla RNN: step backward (4 Pts)

In the `VanillaRNN` class in the file `lib/rnn/layer_utils.py` complete the `step_backward` function.

After doing so run the following to numerically gradient check the implementation. You should see errors less than `1e-8` .

In [3]:
```
%reload_ext autoreload

np.random.seed(231)
N, D, H = 4, 5, 6

rnn = VanillaRNN(D, H, init_scale=0.02, name="rnn_test")

x = np.random.randn(N, D)
h = np.random.randn(N, H)
Wx = np.random.randn(D, H)
Wh = np.random.randn(H, H)
b = np.random.randn(H)

rnn.params[rnn.wx_name] = Wx
rnn.params[rnn.wh_name] = Wh
rnn.params[rnn.b_name] = b

out, meta = rnn.step_forward(x, h)

dnext_h = np.random.randn(*out.shape)

dx_num = eval_numerical_gradient_array(lambda x: rnn.step_forward(x, h)[0], x, dr
dprev_h_num = eval_numerical_gradient_array(lambda h: rnn.step_forward(x, h)[0],
dWx_num = eval_numerical_gradient_array(lambda Wx: rnn.step_forward(x, h)[0], Wx,
dWh_num = eval_numerical_gradient_array(lambda Wh: rnn.step_forward(x, h)[0], Wh,
db_num = eval_numerical_gradient_array(lambda b: rnn.step_forward(x, h)[0], b, dr

dx, dprev_h, dWx, dWh, db = rnn.step_backward(dnext_h, meta)

print('dx error: ', rel_error(dx_num, dx))
print('dprev_h error: ', rel_error(dprev_h_num, dprev_h))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

```
dx error:  6.447921686813177e-10
dprev_h error:  2.734803312602142e-10
dWx error:  1.9370107669274995e-10
dWh error:  3.1863456358952086e-10
db error:  6.415505229199452e-11
```

Vanilla RNN: forward (4 Pts)

Now that you have completed the forward and backward passes for a single timestep of a vanilla RNN, you will see how they are combined to implement a RNN that process an entire sequence of data.

In the `VanillaRNN` class in the file `lib/rnn/layer_utils.py`, complete the function `forward`. This is implemented using the `step_forward` function that you defined above.

After doing so run the following to check the implementation. You should see errors less than `1e-7`.

In [4]:
```
%reload_ext autoreload

N, T, D, H = 2, 3, 4, 5

rnn = VanillaRNN(D, H, init_scale=0.02, name="rnn_test")

x = np.linspace(-0.1, 0.3, num=N*T*D).reshape(N, T, D)
h0 = np.linspace(-0.3, 0.1, num=N*H).reshape(N, H)
Wx = np.linspace(-0.2, 0.4, num=D*H).reshape(D, H)
Wh = np.linspace(-0.4, 0.1, num=H*H).reshape(H, H)
b = np.linspace(-0.7, 0.1, num=H)

rnn.params[rnn.wx_name] = Wx
rnn.params[rnn.wh_name] = Wh
rnn.params[rnn.b_name] = b

h = rnn.forward(x, h0)
expected_h = np.asarray([
  [
    [-0.42070749, -0.27279261, -0.11074945,  0.05740409,  0.22236251],
    [-0.39525808, -0.22554661, -0.0409454,   0.14649412,  0.32397316],
    [-0.42305111, -0.24223728, -0.04287027,  0.15997045,  0.35014525],
  ],
  [
    [-0.55857474, -0.39065825, -0.19198182,  0.02378408,  0.23735671],
    [-0.27150199, -0.07088804,  0.13562939,  0.33099728,  0.50158768],
    [-0.51014825, -0.30524429, -0.06755202,  0.17806392,  0.40333043]]])
print('h error: ', rel_error(expected_h, h))
```

```
h error:  7.728466180186066e-08
```

Vanilla RNN: backward (4 Pts)

In the file `lib/rnn/layer_utils.py`, complete the backward pass for a vanilla RNN in the function `backward` in the `VanillaRNN` class. This runs back-propagation over the entire sequence, calling into the `step_backward` function defined above.

You should see errors less than 5e-7.

In [5]:
```
%reload_ext autoreload

np.random.seed(231)

N, D, T, H = 2, 3, 10, 5

rnn = VanillaRNN(D, H, init_scale=0.02, name="rnn_test")

x = np.random.randn(N, T, D)
h0 = np.random.randn(N, H)
Wx = np.random.randn(D, H)
Wh = np.random.randn(H, H)
b = np.random.randn(H)

rnn.params[rnn.wx_name] = Wx
rnn.params[rnn.wh_name] = Wh
rnn.params[rnn.b_name] = b

out = rnn.forward(x, h0)

dout = np.random.randn(*out.shape)

dx, dh0 = rnn.backward(dout)

dx_num = eval_numerical_gradient_array(lambda x: rnn.forward(x, h0), x, dout)
dh0_num = eval_numerical_gradient_array(lambda h0: rnn.forward(x, h0), h0, dout)
dWx_num = eval_numerical_gradient_array(lambda Wx: rnn.forward(x, h0), Wx, dout)
dWh_num = eval_numerical_gradient_array(lambda Wh: rnn.forward(x, h0), Wh, dout)
db_num = eval_numerical_gradient_array(lambda b: rnn.forward(x, h0), b, dout)

dWx = rnn.grads[rnn.wx_name]
dWh = rnn.grads[rnn.wh_name]
db = rnn.grads[rnn.b_name]

print('dx error: ', rel_error(dx_num, dx))
print('dh0 error: ', rel_error(dh0_num, dh0))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

```
dx error:  1.7733540516533993e-08
dh0 error:  1.1185701416558183e-09
dWx error:  2.0806521162853925e-08
dWh error:  1.5030291853141093e-08
db error:  2.735929351595301e-10
```

Word embedding: forward (4 Pts)

In deep learning systems, we commonly represent words using vectors. Each word of the vocabulary will be associated with a vector, and these vectors will be learned jointly with the rest of the system.

In the file `lib/rnn/layer_utils.py`, implement the function `forward` in the `word_embedding` class to convert words (represented by integers) into vectors. Run the following to check the implementation. You should see error around `1e-8`.

In [6]:
```
%reload_ext autoreload

N, T, V, D = 2, 4, 5, 3

we = word_embedding(V, D, name="we")

x = np.asarray([[0, 3, 1, 2], [2, 1, 0, 3]])
W = np.linspace(0, 1, num=V*D).reshape(V, D)

we.params[we.w_name] = W

out = we.forward(x)
expected_out = np.asarray([
 [[ 0.,          0.07142857,  0.14285714],
  [ 0.64285714,  0.71428571,  0.78571429],
  [ 0.21428571,  0.28571429,  0.35714286],
  [ 0.42857143,  0.5,         0.57142857]],
 [[ 0.42857143,  0.5,         0.57142857],
  [ 0.21428571,  0.28571429,  0.35714286],
  [ 0.,          0.07142857,  0.14285714],
  [ 0.64285714,  0.71428571,  0.78571429]]])

print('out error: ', rel_error(expected_out, out))
```

```
out error:  1.0000000094736443e-08
```

Word embedding: backward (4 Pts)

Implement the backward pass for the word embedding function in the function `backward` in the `word_embedding` class. After doing so run the following to numerically gradient check your implementation. You should see errors less than `1e-11`.

In [7]:
```
%reload_ext autoreload

np.random.seed(231)

N, T, V, D = 50, 3, 5, 6

we = word_embedding(V, D, name="we")

x = np.random.randint(V, size=(N, T))
W = np.random.randn(V, D)

we.params[we.w_name] = W

out = we.forward(x)
dout = np.random.randn(*out.shape)
we.backward(dout)

dW = we.grads[we.w_name]

f = lambda W: we.forward(x)
dW_num = eval_numerical_gradient_array(f, W, dout)

print('dW error: ', rel_error(dW, dW_num))
```

```
dW error:  3.2759440934795915e-12
```

Temporal Fully Connected layer

At every timestep we use an affine function to transform the RNN hidden vector at that timestep into scores for each word in the vocabulary. Because this is very similar to the fully connected layer that you implemented in assignment 1, we have provided this function for you in the `forward` and `backward` functions in the file `lib/rnn/layer_util.py`. Run the following to perform numeric gradient checking on the implementation. You should see errors less than 1e-9.

In [8]:
```
%reload_ext autoreload

np.random.seed(231)

# Gradient check for temporal affine layer
N, T, D, M = 2, 3, 4, 5

t_fc = temporal_fc(D, M, init_scale=0.02, name='test_t_fc')

x = np.random.randn(N, T, D)
w = np.random.randn(D, M)
b = np.random.randn(M)

t_fc.params[t_fc.w_name] = w
t_fc.params[t_fc.b_name] = b

out = t_fc.forward(x)

dout = np.random.randn(*out.shape)

dx_num = eval_numerical_gradient_array(lambda x: t_fc.forward(x), x, dout)
dw_num = eval_numerical_gradient_array(lambda w: t_fc.forward(x), w, dout)
db_num = eval_numerical_gradient_array(lambda b: t_fc.forward(x), b, dout)

dx = t_fc.backward(dout)
dw = t_fc.grads[t_fc.w_name]
db = t_fc.grads[t_fc.b_name]

print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
dx error:  3.2269470390098687e-10
dw error:  3.8595619942595054e-11
db error:  1.1455396263586309e-11
```

Temporal Softmax Cross-Entropy loss

When rolling out a RNN language model to generate a sentence, at every timestep we produce a score for each word in the vocabulary, propotional to the predicted likelihood of this word appearing at the particular timestep in the sentence. We know the ground-truth word at each timestep, so we use a softmax cross-entropy loss function to (1) compute a proper probability distribution over the words in the vocabulary at every time step and (2) use this to compute loss and gradient at each timestep. We sum the losses over time and average them over the minibatch.

We provide this loss function for you; look at the `temporal_softmax_CE_loss` function in the file `lib/rnn/layer_utils.py`.

Run the following cell to sanity check the loss and perform numeric gradient checking on the function. You should see an error for dx less than 1e-7.

In [9]:
```
%reload_ext autoreload

loss_func = temporal_softmax_CE_loss()

# Sanity check for temporal softmax loss
N, T, V = 100, 1, 10

def check_loss(N, T, V, p):
    x = 0.001 * np.random.randn(N, T, V)
    y = np.random.randint(V, size=(N, T))
    mask = np.random.rand(N, T) <= p
    print(loss_func.forward(x, y, mask))

check_loss(100, 1, 10, 1.0)   # Should be about 2.3
check_loss(100, 10, 10, 1.0)  # Should be about 23
check_loss(5000, 10, 10, 0.1) # Should be about 2.3

# Gradient check for temporal softmax loss
N, T, V = 7, 8, 9

x = np.random.randn(N, T, V)
y = np.random.randint(V, size=(N, T))
mask = (np.random.rand(N, T) > 0.5)

loss = loss_func.forward(x, y, mask)
dx = loss_func.backward()

dx_num = eval_numerical_gradient(lambda x: loss_func.forward(x, y, mask), x, verb

print('dx error: ', rel_error(dx, dx_num))
```

```
2.3026547279318357
23.026307039328714
2.2989009292538665
dx error:  4.0464746298031226e-08
```

RNN for language modeling

Now that you have the necessary layers, you can combine them to build a language modeling model. Open the file `lib/rnn/rnn.py` and look at the `TestRNN` class.

For now only check the forward and backward pass of the `TestRNN` model and ignore the `TODOs` in the constructor; you will implement these later. After doing so, run the following to check the forward and backward pass using a small test case; you should see error less than `1e-10`.

In [10]:
```
%reload_ext autoreload

N, D, H = 10, 20, 40
V = 4
T = 13

model = TestRNN(D, H, cell_type='rnn')
loss_func = temporal_softmax_CE_loss()

# Set all model parameters to fixed values
for k, v in model.params.items():
    model.params[k] = np.linspace(-1.4, 1.3, num=v.size).reshape(*v.shape)
model.assign_params()

features = np.linspace(-1.5, 0.3, num=(N * D * T)).reshape(N, T, D)
h0 = np.linspace(-1.5, 0.5, num=(N*H)).reshape(N, H)
labels = (np.arange(N * T) % V).reshape(N, T)

pred = model.forward(features, h0)

# You'll need this
mask = np.ones((N, T))

loss = loss_func.forward(pred, labels, mask)
dLoss = loss_func.backward()

expected_loss = 51.0949189134

print('loss: ', loss)
print('expected loss: ', expected_loss)
print('difference: ', abs(loss - expected_loss))
```

```
loss:  51.094918913361184
expected loss:  51.0949189134
difference:  3.881694965457427e-11
```

Run the following cell to perform more detailed gradient checking on the backward pass of the `TestRNN` class; you should errors around `1e-7` or less.

In [11]:
```
%reload_ext autoreload

np.random.seed(231)

batch_size = 2
timesteps = 3
input_dim = 4
hidden_dim = 6
label_size = 4

labels = np.random.randint(label_size, size=(batch_size, timesteps))
features = np.random.randn(batch_size, timesteps, input_dim)
h0 = np.random.randn(batch_size, hidden_dim)

model = TestRNN(input_dim, hidden_dim, cell_type='rnn')
loss_func = temporal_softmax_CE_loss()

pred = model.forward(features, h0)

# You'll need this
mask = np.ones((batch_size, timesteps))

loss = loss_func.forward(pred, labels, mask)
dLoss = loss_func.backward()

dout, dh0 = model.backward(dLoss)

grads = model.grads

for param_name in sorted(grads):
    f = lambda _: loss_func.forward(model.forward(features, h0), labels, mask)
    param_grad_num = eval_numerical_gradient(f, model.params[param_name], verbose
    e = rel_error(param_grad_num, grads[param_name])
    print('%s relative error: %e' % (param_name, e))
```

```
vanilla_rnn_b relative error: 9.451394e-08
vanilla_rnn_wh relative error: 3.221744e-08
vanilla_rnn_wx relative error: 9.508480e-08
```

LSTM

Vanilla RNNs can be tough to train on long sequences due to vanishing and exploding gradiants. LSTMs solve this problem by replacing the simple update rule in the forward step of the vanilla RNN with a gating mechanism as follows.

Similar to the vanilla RNN, at each timestep we receive an input $x_t \in \mathbb{R}^D$ and the previous hidden state $h_{t-1} \in \mathbb{R}^H$. Crucially, the LSTM also maintains an H-dimensional *cell state*, so we also receive the previous cell state $c_{t-1} \in \mathbb{R}^H$. The learnable parameters of the LSTM are an *input-to-hidden* matrix $W_x \in \mathbb{R}^{4H \times D}$, a *hidden-to-hidden* matrix $W_h \in \mathbb{R}^{4H \times H}$ and a *bias vector* $b \in \mathbb{R}^{4H}$.

At each timestep we first compute an *activation vector* $a \in \mathbb{R}^{4H}$ as $a = W_x x_t + W_h h_{t-1} + b$. We then divide this into four vectors $a_i, a_f, a_o, a_g \in \mathbb{R}^H$ where a_i consists of the first H elements of a, a_f is the next H elements of a, etc. We then compute the *input gate* $g \in \mathbb{R}^H$, *forget gate* $f \in \mathbb{R}^H$, *output gate* $o \in \mathbb{R}^H$ and *gate gate* $g \in \mathbb{R}^H$ as

$$i = \sigma(a_i) \qquad f = \sigma(a_f) \qquad o = \sigma(a_o) \qquad g = \tanh(a_g)$$

where σ is the sigmoid function and $\tanh$ is the hyperbolic tangent, both applied elementwise.

Finally we compute the next cell state c_t and next hidden state h_t as

$$c_t = f \odot c_{t-1} + i \odot g \qquad\qquad h_t = o \odot \tanh(c_t)$$

where $\odot$ is the elementwise product of vectors.

In the rest of the notebook we will implement the LSTM update rule and apply it to the text generation task.

In the code, we assume that data is stored in batches so that $X_t \in \mathbb{R}^{N \times D}$, and will work with *transposed* versions of the parameters: $W_x \in \mathbb{R}^{D \times 4H}$, $W_h \in \mathbb{R}^{H \times 4H}$ so that activations $A \in \mathbb{R}^{N \times 4H}$ can be computed efficiently as $A = X_t W_x + H_{t-1} W_h$

LSTM: step forward (6 Pts)

Implement the forward pass for a single timestep of an LSTM in the `step_forward` function in the file `lib/rnn/layer_utils.py`. This should be similar to the `step_forward` function that you implemented above, but using the LSTM update rule instead.

Once you are done, run the following to perform a simple test of your implementation. You should see errors around `1e-8` or less.

In [12]:
```
%reload_ext autoreload

N, D, H = 3, 4, 5

lstm = LSTM(D, H, init_scale=0.02, name='test_lstm')

x = np.linspace(-0.4, 1.2, num=N*D).reshape(N, D)
prev_h = np.linspace(-0.3, 0.7, num=N*H).reshape(N, H)
prev_c = np.linspace(-0.4, 0.9, num=N*H).reshape(N, H)
Wx = np.linspace(-2.1, 1.3, num=4*D*H).reshape(D, 4 * H)
Wh = np.linspace(-0.7, 2.2, num=4*H*H).reshape(H, 4 * H)
b = np.linspace(0.3, 0.7, num=4*H)

lstm.params[lstm.wx_name] = Wx
lstm.params[lstm.wh_name] = Wh
lstm.params[lstm.b_name] = b

next_h, next_c, cache = lstm.step_forward(x, prev_h, prev_c)

expected_next_h = np.asarray([
    [ 0.24635157,  0.28610883,  0.32240467,  0.35525807,  0.38474904],
    [ 0.49223563,  0.55611431,  0.61507696,  0.66844003,  0.7159181 ],
    [ 0.56735664,  0.66310127,  0.74419266,  0.80889665,  0.858299  ]])
expected_next_c = np.asarray([
    [ 0.32986176,  0.39145139,  0.451556,    0.51014116,  0.56717407],
    [ 0.66382255,  0.76674007,  0.87195994,  0.97902709,  1.08751345],
    [ 0.74192008,  0.90592151,  1.07717006,  1.25120233,  1.42395676]])

print('next_h error: ', rel_error(expected_next_h, next_h))
print('next_c error: ', rel_error(expected_next_c, next_c))
```

```
next_h error:  5.7054131185818695e-09
next_c error:  5.8143123088804145e-09
```

LSTM: step backward (6 Pts)

Implement the backward pass for a single LSTM timestep in the function `step_backward` in the file `lib/rnn/layer_utils.py` . Once you are done, run the following to perform numeric gradient checking on your implementation. You should see errors around `1e-6` or less.

In [13]:
```
%reload_ext autoreload

np.random.seed(231)

N, D, H = 4, 5, 6

lstm = LSTM(D, H, init_scale=0.02, name='test_lstm')

x = np.random.randn(N, D)
prev_h = np.random.randn(N, H)
prev_c = np.random.randn(N, H)
Wx = np.random.randn(D, 4 * H)
Wh = np.random.randn(H, 4 * H)
b = np.random.randn(4 * H)

lstm.params[lstm.wx_name] = Wx
lstm.params[lstm.wh_name] = Wh
lstm.params[lstm.b_name] = b

next_h, next_c, cache = lstm.step_forward(x, prev_h, prev_c)

dnext_h = np.random.randn(*next_h.shape)
dnext_c = np.random.randn(*next_c.shape)

fx_h = lambda x: lstm.step_forward(x, prev_h, prev_c)[0]
fh_h = lambda h: lstm.step_forward(x, prev_h, prev_c)[0]
fc_h = lambda c: lstm.step_forward(x, prev_h, prev_c)[0]
fWx_h = lambda Wx: lstm.step_forward(x, prev_h, prev_c)[0]
fWh_h = lambda Wh: lstm.step_forward(x, prev_h, prev_c)[0]
fb_h = lambda b: lstm.step_forward(x, prev_h, prev_c)[0]

fx_c = lambda x: lstm.step_forward(x, prev_h, prev_c)[1]
fh_c = lambda h: lstm.step_forward(x, prev_h, prev_c)[1]
fc_c = lambda c: lstm.step_forward(x, prev_h, prev_c)[1]
fWx_c = lambda Wx: lstm.step_forward(x, prev_h, prev_c)[1]
fWh_c = lambda Wh: lstm.step_forward(x, prev_h, prev_c)[1]
fb_c = lambda b: lstm.step_forward(x, prev_h, prev_c)[1]

num_grad = eval_numerical_gradient_array

dx_num = num_grad(fx_h, x, dnext_h) + num_grad(fx_c, x, dnext_c)
dh_num = num_grad(fh_h, prev_h, dnext_h) + num_grad(fh_c, prev_h, dnext_c)
dc_num = num_grad(fc_h, prev_c, dnext_h) + num_grad(fc_c, prev_c, dnext_c)
dWx_num = num_grad(fWx_h, Wx, dnext_h) + num_grad(fWx_c, Wx, dnext_c)
dWh_num = num_grad(fWh_h, Wh, dnext_h) + num_grad(fWh_c, Wh, dnext_c)
db_num = num_grad(fb_h, b, dnext_h) + num_grad(fb_c, b, dnext_c)

dx, dh, dc, dWx, dWh, db = lstm.step_backward(dnext_h, dnext_c, cache)

print('dx error: ', rel_error(dx_num, dx))
print('dh error: ', rel_error(dh_num, dh))
print('dc error: ', rel_error(dc_num, dc))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

```
dx error:  3.3906726460948097e-10
dh error:  1.048321718944741e-08
dc error:  1.0127280886420228e-08
dWx error:  1.8944292163912603e-07
dWh error:  1.461752914864933e-07
db error:  1.8671697284523756e-08
```

LSTM: forward (6 Pts)

In the class `lstm` in the file `lib/rnn/layer_utils.py`, implement the `forward` function to run an LSTM forward on an entire timeseries of data.

When you are done, run the following to check your implementation. You should see an error around `1e-7`.

In [14]:
```
%reload_ext autoreload

N, D, H, T = 2, 5, 4, 3

lstm = LSTM(D, H, init_scale=0.02, name='test_lstm')

x = np.linspace(-0.4, 0.6, num=N*T*D).reshape(N, T, D)
h0 = np.linspace(-0.4, 0.8, num=N*H).reshape(N, H)
Wx = np.linspace(-0.2, 0.9, num=4*D*H).reshape(D, 4 * H)
Wh = np.linspace(-0.3, 0.6, num=4*H*H).reshape(H, 4 * H)
b = np.linspace(0.2, 0.7, num=4*H)

lstm.params[lstm.wx_name] = Wx
lstm.params[lstm.wh_name] = Wh
lstm.params[lstm.b_name] = b

h = lstm.forward(x, h0)

expected_h = np.asarray([
 [[ 0.01764008,  0.01823233,  0.01882671,  0.0194232 ],
  [ 0.11287491,  0.12146228,  0.13018446,  0.13902939],
  [ 0.31358768,  0.33338627,  0.35304453,  0.37250975]],
 [[ 0.45767879,  0.4761092,   0.4936887,   0.51041945],
  [ 0.6704845,   0.69350089,  0.71486014,  0.7346449 ],
  [ 0.81733511,  0.83677871,  0.85403753,  0.86935314]]])

print('h error: ', rel_error(expected_h, h))
```

```
h error:  8.610537452106624e-08
```

LSTM: backward (6 Pts)

Implement the backward pass for an LSTM over an entire timeseries of data in the function `backward` in the `lstm` class in the file `lib/rnn/layer_utils.py`. When you are done, run the following to perform numeric gradient checking on your implementation. You should see errors around `1e-7` or less.

In [15]:
```
%reload_ext autoreload

np.random.seed(231)

N, D, T, H = 2, 3, 10, 6

lstm = LSTM(D, H, init_scale=0.02, name='test_lstm')

x = np.random.randn(N, T, D)
h0 = np.random.randn(N, H)
Wx = np.random.randn(D, 4 * H)
Wh = np.random.randn(H, 4 * H)
b = np.random.randn(4 * H)

lstm.params[lstm.wx_name] = Wx
lstm.params[lstm.wh_name] = Wh
lstm.params[lstm.b_name] = b

out = lstm.forward(x, h0)

dout = np.random.randn(*out.shape)

dx, dh0 = lstm.backward(dout)
dWx = lstm.grads[lstm.wx_name]
dWh = lstm.grads[lstm.wh_name]
db = lstm.grads[lstm.b_name]

dx_num = eval_numerical_gradient_array(lambda x: lstm.forward(x, h0), x, dout)
dh0_num = eval_numerical_gradient_array(lambda h0: lstm.forward(x, h0), h0, dout)
dWx_num = eval_numerical_gradient_array(lambda Wx: lstm.forward(x, h0), Wx, dout)
dWh_num = eval_numerical_gradient_array(lambda Wh: lstm.forward(x, h0), Wh, dout)
db_num = eval_numerical_gradient_array(lambda b: lstm.forward(x, h0), b, dout)

print('dx error: ', rel_error(dx_num, dx))
print('dh0 error: ', rel_error(dh0_num, dh0))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

```
dx error:  1.4551080281612179e-09
dh0 error:  4.884205769873603e-10
dWx error:  2.903749692081731e-09
dWh error:  1.3098725231893739e-07
db error:  3.323911044905218e-10
```

LSTM model (2 Pts)

Now that you have implemented an LSTM, update the initialization of the `TestRNN` class in the file `lib/rnn/rnn.py` to handle the case where `self.cell_type` is `lstm`.

Once you have done so, run the following to check your implementation. You should see a difference of less than `1e-10`.

In [16]:
```
%reload_ext autoreload

N, D, H = 10, 20, 40
V = 4
T = 13

model = TestRNN(D, H, cell_type='lstm')
loss_func = temporal_softmax_CE_loss()

# Set all model parameters to fixed values
for k, v in model.params.items():
    model.params[k] = np.linspace(-1.4, 1.3, num=v.size).reshape(*v.shape)
model.assign_params()

features = np.linspace(-1.5, 0.3, num=(N * D * T)).reshape(N, T, D)
h0 = np.linspace(-1.5, 0.5, num=(N*H)).reshape(N, H)
labels = (np.arange(N * T) % V).reshape(N, T)

pred = model.forward(features, h0)

# You'll need this
mask = np.ones((N, T))

loss = loss_func.forward(pred, labels, mask)
dLoss = loss_func.backward()

expected_loss = 49.2140256354

print('loss: ', loss)
print('expected loss: ', expected_loss)
print('difference: ', abs(loss - expected_loss))
```

```
loss:  49.21402563544293
expected loss:  49.2140256354
difference:  4.293099209462525e-11
```

Let's have some fun!! (8 Pts)

Now you have everything you need for language modeling. You will work on text generation using RNNs from any text source (novel, lyrics).

The network is trained to predict what word is coming next given a previous word. Once you train the model, by looping the network, you can keep generating new text which is mimicing the original text source.

We will use one of the most frequently downloaded e-books, Alice's Adventures in Wonderland, from Project Gutenberg, where the original link can be found here (https://www.gutenberg.org/ebooks/11).

For simplify training we extracted only the first chapter.

In [3]:
```
%reload_ext autoreload

input_file = open("data/alice.txt", "r", encoding="utf8")
input_text = input_file.readlines()
input_text = ''.join(input_text)
```

Simply run the following code to construct the training dataset.

In [6]:
```
%reload_ext autoreload

import re

text = re.split(' |\n',input_text.lower())  # all words are converted into lower
outputSize = len(text)
word_list = list(set(text))
dataSize = len(word_list)
output = np.zeros(outputSize)
for i in range(0, outputSize):
    index = np.where(np.asarray(word_list) == text[i])
    output[i] = index[0]
data = output.astype(np.int)
gt_labels = data[1:]
input_data = data[:-1]

print('Input text size: %s' % outputSize)
print('Input word number: %s' % dataSize)
```

```
Input text size: 2169
Input word number: 778
```

We defined a LanguageModelRNN class for you in `rnn.py`. Please fill in the TODO block in the constructor and complete the training loop.

- In the constructor, design a recurrent neutral network consisting of a word_embedding layer, recurrent unit, and temporal fully connected layer so that they match the provided dimensions.
- Please read the train.py under lib directory carefully and complete the TODO blocks in the train_net function. Then execute the following code block to train the model.

In [19]:
```
%reload_ext autoreload

# you can change the following parameters.
D = 15  # input dimension
H = 100  # hidden space dimension
T = 50  # timesteps
N = 30  # batch size
max_epoch = 50  # max epoch size

loss_func = temporal_softmax_CE_loss()
# you can change the cell_type between 'rnn' and 'lstm'.
model = LanguageModelRNN(dataSize, D, H, cell_type='lstm')
optimizer = Adam(model, 5e-4)

data = {'data_train': input_data, 'labels_train': gt_labels}

results = train_net(data, model, loss_func, optimizer, timesteps=T, batch_size=N,
```

```
(Iteration 1 / 3600) loss: 332.79380000347584
best performance 3.5977859778597785%
(Epoch 1 / 50) Training Accuracy: 0.035977859778597784
best performance 4.105166051660516%
(Epoch 2 / 50) Training Accuracy: 0.041051660516605165
best performance 4.474169741697417%
(Epoch 3 / 50) Training Accuracy: 0.04474169741697417
best performance 4.843173431734318%
(Epoch 4 / 50) Training Accuracy: 0.04843173431734318
best performance 6.411439114391144%
(Epoch 5 / 50) Training Accuracy: 0.06411439114391145
best performance 7.5184501845018445%
(Epoch 6 / 50) Training Accuracy: 0.07518450184501844
(Iteration 501 / 3600) loss: 233.11054973143752
best performance 9.916974169741698%
(Epoch 7 / 50) Training Accuracy: 0.09916974169741698
best performance 13.099630996309964%
(Epoch 8 / 50) Training Accuracy: 0.13099630996309963
best performance 17.38929889298893%
(Epoch 9 / 50) Training Accuracy: 0.1738929889298893
best performance 22.739852398523986%
(Epoch 10 / 50) Training Accuracy: 0.22739852398523985
best performance 28.64391143911439%
(Epoch 11 / 50) Training Accuracy: 0.2864391143911439
best performance 34.640221402214024%
(Epoch 12 / 50) Training Accuracy: 0.3464022140221402
best performance 41.005535055350556%
(Epoch 13 / 50) Training Accuracy: 0.41005535055350556
(Iteration 1001 / 3600) loss: 131.941596836897
best performance 46.44833948339483%
(Epoch 14 / 50) Training Accuracy: 0.46448339483394835
best performance 53.27490774907749%
(Epoch 15 / 50) Training Accuracy: 0.5327490774907749
best performance 57.656826568265686%
(Epoch 16 / 50) Training Accuracy: 0.5765682656826568
best performance 63.284132841328415%
(Epoch 17 / 50) Training Accuracy: 0.6328413284132841
```

```
best performance 66.46678966789668%
(Epoch 18 / 50) Training Accuracy: 0.6646678966789668
best performance 69.880073800738%
(Epoch 19 / 50) Training Accuracy: 0.6988007380073801
best performance 73.70848708487084%
(Epoch 20 / 50) Training Accuracy: 0.7370848708487084
(Iteration 1501 / 3600) loss: 73.90719154159073
best performance 76.29151291512916%
(Epoch 21 / 50) Training Accuracy: 0.7629151291512916
best performance 79.61254612546126%
(Epoch 22 / 50) Training Accuracy: 0.7961254612546126
best performance 82.380073800738%
(Epoch 23 / 50) Training Accuracy: 0.8238007380073801
best performance 84.59409594095942%
(Epoch 24 / 50) Training Accuracy: 0.8459409594095941
best performance 86.90036900369003%
(Epoch 25 / 50) Training Accuracy: 0.8690036900369004
best performance 88.79151291512916%
(Epoch 26 / 50) Training Accuracy: 0.8879151291512916
best performance 90.17527675276753%
(Epoch 27 / 50) Training Accuracy: 0.9017527675276753
(Iteration 2001 / 3600) loss: 40.70493628671922
best performance 91.65129151291514%
(Epoch 28 / 50) Training Accuracy: 0.9165129151291513
best performance 92.80442804428044%
(Epoch 29 / 50) Training Accuracy: 0.9280442804428044
best performance 93.81918819188192%
(Epoch 30 / 50) Training Accuracy: 0.9381918819188192
best performance 94.55719557195572%
(Epoch 31 / 50) Training Accuracy: 0.9455719557195572
best performance 95.84870848708486%
(Epoch 32 / 50) Training Accuracy: 0.9584870848708487
best performance 96.54059040590406%
(Epoch 33 / 50) Training Accuracy: 0.9654059040590406
best performance 97.18634686346863%
(Epoch 34 / 50) Training Accuracy: 0.9718634686346863
(Iteration 2501 / 3600) loss: 24.382210376550105
best performance 97.73985239852398%
(Epoch 35 / 50) Training Accuracy: 0.9773985239852399
best performance 97.92435424354244%
(Epoch 36 / 50) Training Accuracy: 0.9792435424354243
best performance 98.1549815498155%
(Epoch 37 / 50) Training Accuracy: 0.981549815498155
best performance 98.38560885608855%
(Epoch 38 / 50) Training Accuracy: 0.9838560885608856
best performance 98.57011070110701%
(Epoch 39 / 50) Training Accuracy: 0.9857011070110702
best performance 98.61623616236163%
(Epoch 40 / 50) Training Accuracy: 0.9861623616236163
best performance 98.75461254612546%
(Epoch 41 / 50) Training Accuracy: 0.9875461254612546
(Iteration 3001 / 3600) loss: 17.183596186984285
best performance 98.8929889298893%
(Epoch 42 / 50) Training Accuracy: 0.988929889298893
best performance 99.03136531365314%
(Epoch 43 / 50) Training Accuracy: 0.9903136531365314
best performance 99.07749077490774%
```

```
(Epoch 44 / 50) Training Accuracy: 0.9907749077490775
(Epoch 45 / 50) Training Accuracy: 0.9907749077490775
(Epoch 46 / 50) Training Accuracy: 0.9907749077490775
best performance 99.12361623616236%
(Epoch 47 / 50) Training Accuracy: 0.9912361623616236
best performance 99.16974169741697%
(Epoch 48 / 50) Training Accuracy: 0.9916974169741697
(Iteration 3501 / 3600) loss: 11.548062505269039
best performance 99.2619926199262%
(Epoch 49 / 50) Training Accuracy: 0.992619926199262
best performance 99.30811808118081%
(Epoch 50 / 50) Training Accuracy: 0.9930811808118081
```

Simply run the following code block to check the loss and accuracy curve. (We expect training accuracy to be >80%, you can change all parameters above except `T` to try to improve your training performance. The higher your performance, the better your text samples below will get.)

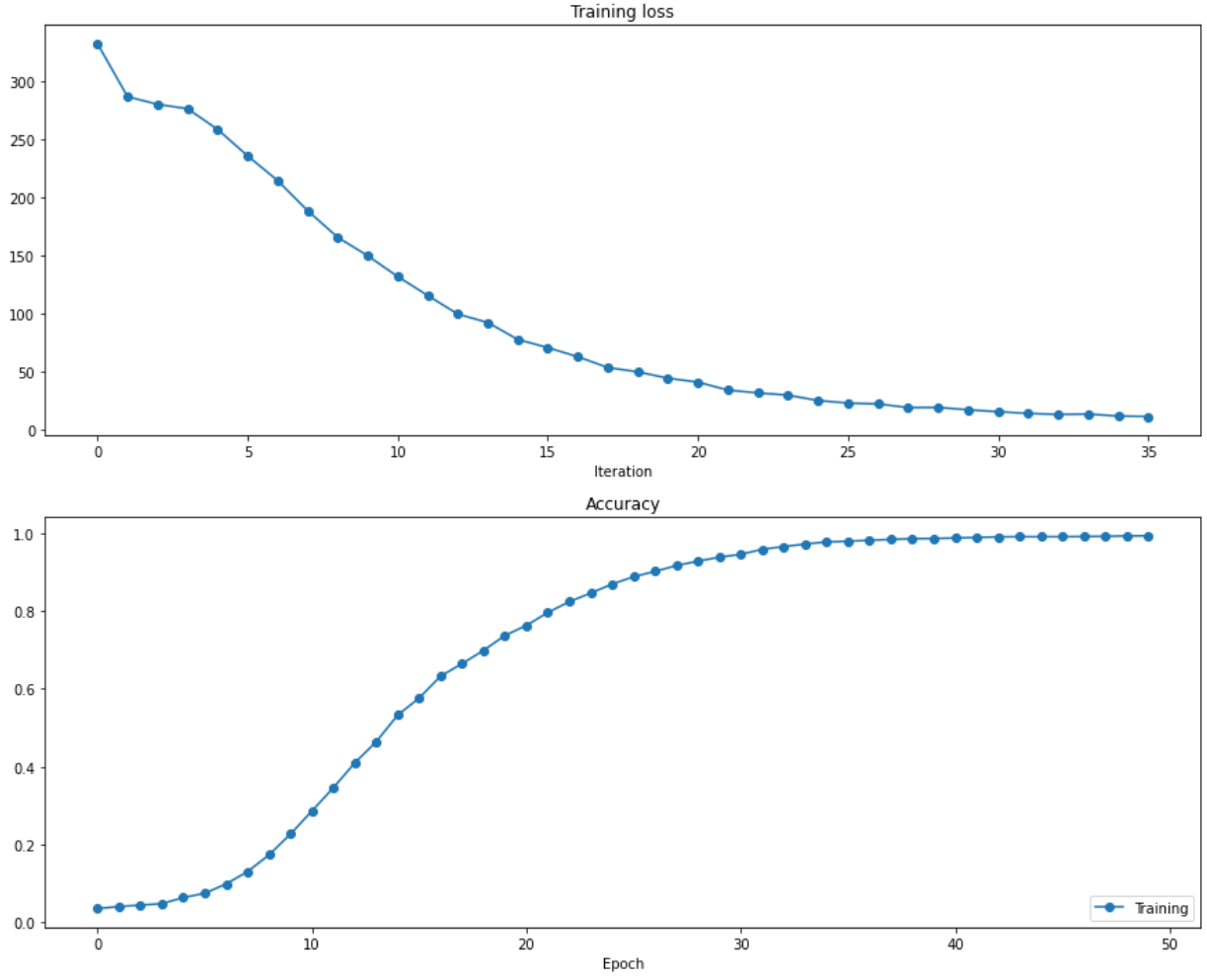
In [20]:
```
%reload_ext autoreload

opt_params, loss_hist, train_acc_hist = results

# Plot the learning curves
plt.subplot(2, 1, 1)
plt.title('Training loss for lstm')
loss_hist_ = loss_hist[1::100]  # sparse the curve a bit
plt.plot(loss_hist_, '-o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy for lstm')
plt.plot(train_acc_hist, '-o', label='Training')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)

plt.show()
```



In [10]:
```
%reload_ext autoreload

# you can change the following parameters.
D = 15  # input dimension
H = 100  # hidden space dimension
T = 50  # timesteps
N = 30  # batch size
max_epoch = 50  # max epoch size

loss_func = temporal_softmax_CE_loss()
# you can change the cell_type between 'rnn' and 'lstm'.
model = LanguageModelRNN(dataSize, D, H, cell_type='rnn')
optimizer = Adam(model, 5e-4)

data = {'data_train': input_data, 'labels_train': gt_labels}

results_1 = train_net(data, model, loss_func, optimizer, timesteps=T, batch_size=
opt_params, loss_hist, train_acc_hist = results_1
```

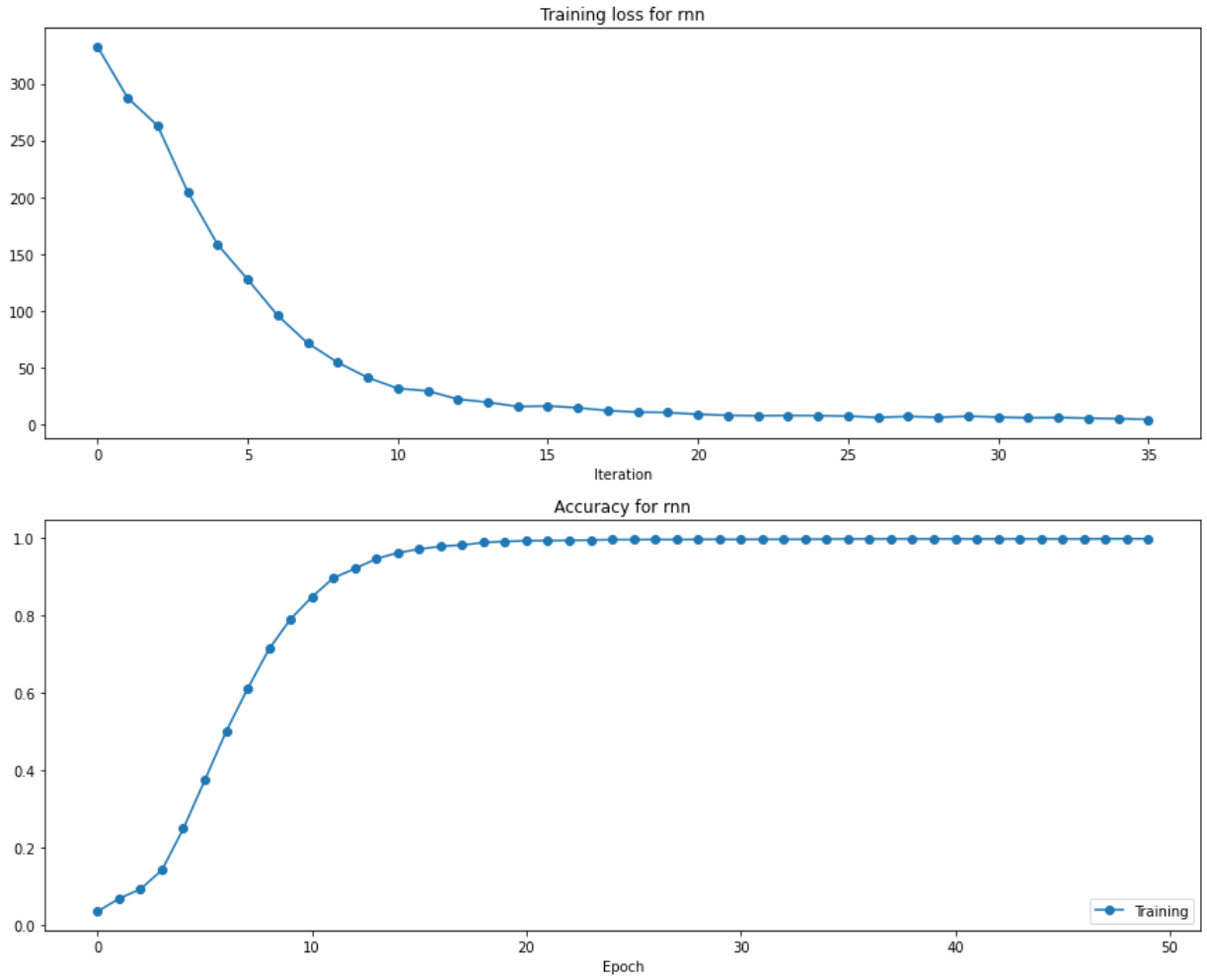
```
(Iteration 1 / 3600) loss: 332.7852621191569
best performance 3.5977859778597785%
(Epoch 1 / 50) Training Accuracy: 0.035977859778597784
best performance 6.918819188191883%
(Epoch 2 / 50) Training Accuracy: 0.06918819188191883
best performance 9.363468634686347%
(Epoch 3 / 50) Training Accuracy: 0.09363468634686346
best performance 14.252767527675278%
(Epoch 4 / 50) Training Accuracy: 0.14252767527675278
best performance 24.95387453874539%
(Epoch 5 / 50) Training Accuracy: 0.24953874538745388
best performance 37.40774907749078%
(Epoch 6 / 50) Training Accuracy: 0.37407749077490776
(Iteration 501 / 3600) loss: 119.44189954439152
best performance 50.04612546125461%
(Epoch 7 / 50) Training Accuracy: 0.5004612546125461
best performance 61.07011070110702%
(Epoch 8 / 50) Training Accuracy: 0.6107011070110702
best performance 71.26383763837639%
(Epoch 9 / 50) Training Accuracy: 0.7126383763837638
best performance 78.96678966789668%
(Epoch 10 / 50) Training Accuracy: 0.7896678966789668
best performance 84.68634686346863%
(Epoch 11 / 50) Training Accuracy: 0.8468634686346863
best performance 89.57564575645756%
(Epoch 12 / 50) Training Accuracy: 0.8957564575645757
best performance 91.97416974169742%
(Epoch 13 / 50) Training Accuracy: 0.9197416974169742
(Iteration 1001 / 3600) loss: 31.8893602103895
best performance 94.51107011070111%
(Epoch 14 / 50) Training Accuracy: 0.9451107011070111
best performance 96.07933579335793%
(Epoch 15 / 50) Training Accuracy: 0.9607933579335793
best performance 97.00184501845018%
(Epoch 16 / 50) Training Accuracy: 0.9700184501845018
best performance 97.73985239852398%
```

```
(Epoch 17 / 50) Training Accuracy: 0.9773985239852399
best performance 98.06273062730627%
(Epoch 18 / 50) Training Accuracy: 0.9806273062730627
best performance 98.70848708487084%
(Epoch 19 / 50) Training Accuracy: 0.9870848708487084
best performance 98.98523985239852%
(Epoch 20 / 50) Training Accuracy: 0.9898523985239852
(Iteration 1501 / 3600) loss: 12.284078295993973
best performance 99.16974169741697%
(Epoch 21 / 50) Training Accuracy: 0.9916974169741697
best performance 99.21586715867159%
(Epoch 22 / 50) Training Accuracy: 0.9921586715867159
best performance 99.2619926199262%
(Epoch 23 / 50) Training Accuracy: 0.992619926199262
best performance 99.35424354243543%
(Epoch 24 / 50) Training Accuracy: 0.9935424354243543
best performance 99.44649446494465%
(Epoch 25 / 50) Training Accuracy: 0.9944649446494465
(Epoch 26 / 50) Training Accuracy: 0.9944649446494465
best performance 99.49261992619927%
(Epoch 27 / 50) Training Accuracy: 0.9949261992619927
(Iteration 2001 / 3600) loss: 9.828653884604442
(Epoch 28 / 50) Training Accuracy: 0.9944649446494465
(Epoch 29 / 50) Training Accuracy: 0.9949261992619927
best performance 99.53874538745387%
(Epoch 30 / 50) Training Accuracy: 0.9953874538745388
(Epoch 31 / 50) Training Accuracy: 0.9949261992619927
(Epoch 32 / 50) Training Accuracy: 0.9953874538745388
(Epoch 33 / 50) Training Accuracy: 0.9953874538745388
(Epoch 34 / 50) Training Accuracy: 0.9953874538745388
(Iteration 2501 / 3600) loss: 6.744764839160829
best performance 99.58487084870849%
(Epoch 35 / 50) Training Accuracy: 0.9958487084870848
best performance 99.63099630996311%
(Epoch 36 / 50) Training Accuracy: 0.996309963099631
(Epoch 37 / 50) Training Accuracy: 0.996309963099631
(Epoch 38 / 50) Training Accuracy: 0.996309963099631
(Epoch 39 / 50) Training Accuracy: 0.996309963099631
(Epoch 40 / 50) Training Accuracy: 0.996309963099631
(Epoch 41 / 50) Training Accuracy: 0.996309963099631
(Iteration 3001 / 3600) loss: 5.269637972426479
(Epoch 42 / 50) Training Accuracy: 0.996309963099631
(Epoch 43 / 50) Training Accuracy: 0.996309963099631
(Epoch 44 / 50) Training Accuracy: 0.996309963099631
(Epoch 45 / 50) Training Accuracy: 0.996309963099631
(Epoch 46 / 50) Training Accuracy: 0.996309963099631
(Epoch 47 / 50) Training Accuracy: 0.996309963099631
best performance 99.67712177121771%
(Epoch 48 / 50) Training Accuracy: 0.9967712177121771
(Iteration 3501 / 3600) loss: 5.143680279836584
(Epoch 49 / 50) Training Accuracy: 0.9967712177121771
(Epoch 50 / 50) Training Accuracy: 0.9967712177121771
```

In [11]:
```
# Plot the learning curves
plt.subplot(2, 1, 1)
plt.title('Training loss for rnn')
loss_hist_ = loss_hist[1::100]  # sparse the curve a bit
plt.plot(loss_hist_, '-o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy for rnn')
plt.plot(train_acc_hist, '-o', label='Training')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)

plt.show()
```



Now you can generate text using the trained model. You can start from a specific word in the original text, such as `she` . (We expect the text to not be too repetitive, i.e. not repeating the same three words over and over. See an example of an acceptable sample below.)

> she was dozing off, and book-shelves; here and she tried to curtsey as she spoke--fancy curtseying as you're falling through the little door into a dreamy sort of way, 'do cats eat bats? do cats eat bats?' and sometimes,

In [12]:
```
%reload_ext autoreload

# you can change the generated text length below.
text_length = 40

idx = 0
# you also can start from specific word.
# since the words are all converted into lower case
idx = int(np.where(np.asarray(word_list) == 'She'.lower())[0])

# sample from the trained model
words = model.sample(idx, text_length-1)

# convert indices into words
output = [word_list[i] for i in words]
print(' '.join(output))
```

```
she was now only ten inches high, and her face brightened up at the thought tha
t she was now the right size for going through the little door into that lovely
garden. first, however, she waited for a few minutes
```

Inline Question (2 Pts): Play around with different settings to get better understanding of its behavior and describe your observation. Make sure to cover at least the following points:

- Vanilla RNN vs LSTM (you can set different training timesteps `T` and `test_length` to test with longer texts.)
- Limitations you observed when training the recurrent language models. What could be causing them? (there's no unique answer. just explain your own opinion from experiments.)

(Please limit your answer to <150 words)

Ans: LSTM is more stable for longer texts compare to RNN but RNN's accuracy is more faster gained than LSTM. RNN cannot deal with missing gradient, thus is not suitable in long texts

Submission

Please prepare a PDF document `problem_2_solution.pdf` in the root directory of this repository with all plots and inline answers of your solution. Concretely, the document should contain the following items in strict order:

1. Training loss / accuracy curves for vanilla RNN and LSTM training
2. Sample text generation from a trained model
3. Answers to inline questions about recurrent net behavior

Note that you still need to submit the jupyter notebook with all generated solutions. We will randomly pick submissions and check that the plots in the PDF and in the notebook are equivalent.

More Text Corpora (Not Graded)

If you want to explore the capabilities of your model further, feel free to train the model on new text corpora! Just bring them in the appropriate format (see above) and then train your model. You can change any of the hyperparameters in the code blocks below. If your model produces some fun outputs you can print them below (including the text corpus the model was trained on).

! Note that this is completely optional and has no influence on the grade of the assignment. !

! Please make sure that all notebook blocks above show results trained on the original text dataset for the assignment as we can only grade those. !

In [ ]:
```
%reload_ext autoreload

################ LOAD YOUR DATA HERE ###################
# word_list: list of unique vocabulary entries, e.g. ['', 'country', 'slippery;',
# data: integer array of shape (num_data_samples,),
# holds index into word_list for each word in the input text, e.g. [649 377 263 .
word_list, data = None, None
########################################################

gt_labels = data[1:]
input_data = data[:-1]
```

In [ ]:
```
%reload_ext autoreload

# you can change the following parameters.
D = 10  # input dimention
H = 20  # hidden space dimention
T = 50  # timesteps
N = 10  # batch size
max_epoch = 50  # max epoch size

loss_func = temporal_softmax_CE_loss()
# you can change the cell_type between 'rnn' and 'lstm'.
model = LanguageModelRNN(dataSize, D, H, cell_type='lstm')
optimizer = Adam(model, 5e-4)

data = {'data_train': input_data, 'labels_train': gt_labels}

results = train_net(data, model, loss_func, optimizer, timesteps=T, batch_size=N,

opt_params, loss_hist, train_acc_hist = results
```

In [ ]:
```
%reload_ext autoreload

# Plot the learning curves
plt.subplot(2, 1, 1)
plt.title('Training loss')
loss_hist_ = loss_hist[1::100]  # sparse the curve a bit
plt.plot(loss_hist_, '-o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(train_acc_hist, '-o', label='Training')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)

plt.show()
```

In [ ]:
```
%reload_ext autoreload

# you can change the generated text length below.
text_length = 15

idx = 0
# you also can start from specific word.
# since the words are all converted into lower case
idx = int(np.where(np.asarray(word_list) == 'She'.lower())[0])

# sample from the trained model
words = model.sample(idx, text_length-1)

# convert indices into words
output = [word_list[i] for i in words]
print(' '.join(output))
```

Fun text generations?

[...]

In [ ]:
```

```