Lab2_2020

March 25, 2020

1 Lab2: Simple Recurrent Network Models for Experimental Predictions

Jennifer Spenader, Giorgos Tziafas, 2020

Completed by Magdalena Bilska (S4086511) and Varun R. Varma (S3893030)

1.1 Learning Goals for this lab:

- Extend practical skills in using Python and Keras to implement simple recurrent networks
- Understand the motivations for the second simulation study in the Mirman, Graf Estes and Magnuson (2010) paper
- Implement the simulation architecture and run the simulation
- Extend knowledge by further building a simulation for the Misyak, Christiansen and Tomblin (2009) study
- Confirm the findings of the two papers by performing a new single reaction experiment with new data
- Reflect on whether you think the model predictions are realisting enough for human participants or not

Note: only questions next to capital letters in `Exercise` sections need to be included in the report. Other questions are rhetorical.

2 Part 1: Recreating Mirman et al.

Let's first focus on these second model in the Mirman, Graf Estes and Magnuson paper. To really understand this you should carefully read the paper, in particular page 4 and pages 9-11. First, make sure you understand what the model is trying to show. Answer the following questions:

2.1 Exercise 1

A) The Mirman et al. paper wants to find evidence linking statistical learning with word learning. In your own words, summarize briefly (but using precise terms) what results with human subjects (adults and children) are the trying to explain better?

They are trying to understand the results of studies conducted by Graf Estes et al. (2007) and Mirman et al. (2008) better. These studies investigated whether people take advantage of transitional probability to support word learning. The study of Graf Estes has shown that (after being exposed to a nonsegmented syllable stream), infants were only able to learn object labels if they were characterized by high probability syllable transitions. Contrary to this finding, the study of Mirman has found that adult learners exposed to a similar task were capable of learning such labels regardless of their syllables' probability transitions (although they learned the high probability transition labels faster).

B) What does Simulation 2 actually show?

Simulation 2 showed that that a simple recurrent network with one hidden layer provides an account of why both infants and adults learned labels with high transitional probability faster - because higher transitional probability syllables have more distinct phonological representations. Therefore, transitional probability improved learning by making the underlying representations more distinct, and therefore resulting in more clear-cut activations.

C) Explain briefly in what way do the results of Simulation 2 add to or clarify the human results.

Simulation 2 provided a possible answer to the "why" question posed by the results of the behavioral experiments. Due to the fact that the NNs used to reproduce human results were fairly straightforward, it was possible to analyze their activation patterns depending on the transitional probabilities of syllables and understand the statistical learning basis of the differences between conditions.

2.2 Model and data files

All the simulation data files can be found on Nestor under a folder named `Mirman_Sim`. (These are the ACTUAL files that the authors used for their paper implementation in Lens!). However, since the purpose of this lab is for you to recreate their simulation in Python, we have provided you with the necessary utilities for parsing the origina data. Just for reference, here are some of the key files with descriptions of their contents: * `exposureSet2Syll35.ex`: Training examples for the exposure phase * `labelSet2SyllW35.ex`: Set of 'Words' for the label learning phase * `labelSet2SyllPW35.ex`: Set of 'Part Words' for the label learning phase * `labelSet2SyllNW35.ex`: Set of 'Non-Words' * `labelSet2SyllNWc35.ex` : Another set of 'Non-Words'

Let's first import all of our necessary utilities for this lab: The `tensorflow.keras` modules for implementing and training our SRNs, some generic Python modules as well as the aforementioned lab utilities from the `mirman_utils.py` script that you can also find in Nestor under `Lab2`. The `LensParser` class takes care of parsing the input data from the Lens format to Python lists and the `Binarizer` class transforms these data to binary `numpy` vectors for interfacing with `keras`.

[68]:
```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, SimpleRNN, TimeDistributed, Flatten
from tensorflow.keras.initializers import RandomUniform
from tensorflow.keras.optimizers import SGD

import random
import numpy as np

from matplotlib import pyplot as plt

from mirman_utils import LensParser, Binarizer, multi_plot
```

The Mirman simulation consists of two phases. First, in the *exposure phase*, the model is presented with 1000-syllable sequences. Our recurrent network takes as an input the first syllable (encoded as a one-hot vector representation) and predicts the next syllable (again encoded as a one-hot vector). The second syllable of the word is then the second input and the third syllable is predicted, etc. Does this model look similar to you as the one that Elman used for lexical class learning? In what ways? The purpose of this phase is to simulate the exposure of human subjects to statistical dependencies between syllables of words in an artificial language.

After the exposure phase, the model is trained on a *label learning* task. The purpose is for our model to activate one out of 5 unique "object" labels after presented with a two-syllable word. This phase simulates the potential of human subjects in learning to map new words to their object representations in the world. Four different label categories, including different overlapping syllable dependencies with the exposure phase are included in order to further research the connection between this statistical learning and word learning. In other words, the research question at hand is whether an SRN that is trained to predict labels for words consisting of syllable transitions never encountered before (or encountered with smaller probabilities) follows the same behaviour in learning them as human subjects do.

After reading the paper fill the code below to define the Mirman SRN model.

Note: Even though in the paper implementation it is explicitely said that the output consists of 15 neurons (the first 10 are for what and the second 5 for what?) in order to follow the traditional `keras` flow and keep things simple we need to define our model's output as 20, the same as the input, even though our model will learn never to activate the last 5.

[2]:
```
## fillin _ with the correct parameter
model = Sequential([
    SimpleRNN(15, input_shape=(None, 20),
              return_sequences=True, #doesn't matter since there's only one␣
 ↪hidden layer
              activation='tanh',
              name='hidden'),
    TimeDistributed(Dense(20, activation='softmax', name='output'))
])

model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=0.05, momentum=0.9),
              metrics=['categorical_accuracy', 'mean_squared_error'])

print(model.summary())
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
hidden (SimpleRNN)           (None, None, 15)          540
_________________________________________________________________
time_distributed (TimeDistri (None, None, 20)          320
=================================================================
Total params: 860
```

```
Trainable params: 860
Non-trainable params: 0
_________________________________________________________________
None
```

2.3 Exercise 2

A) Take a look at the provided Lens file for the exposure phase data. In the Lens format, the syllables are abstract sequences of node activations, with an integer representing the bit of the one-hot encoding that is activated, with each set of two input-output ('I' and 'T') pairs treated as a single 2-syllable word, e.g:

name: {1_0} 2 I: { 1} 3 T: { 1} 7 I: { 1} 7 T: { 1} 1 ;

In this example, the current word consists of the $[3, 7]$ syllables (out of the 10 used for exposure phase) and the first syllable of the next word is 1. There are 1000 such examples of syllable transitions in this file. Explain how this number is arrived at given the Lens format.

Each one and a half-word sequence consists of 4 syllables with 10 possible entries, therefore there should be

$$10 * 10 * 10 * 10 = 10000$$

possible examples of syllable transitions. However, the third entry is always the same as the second (since they represent the same syllable, one as input and one as output), and therefore there are

$$10 * 10 * 1 * 10 = 1000$$

possible sequences.

B) Do the same for the provided label learning data files. In each such file, the same format is used, but for defining the 5 different two-syllable words that the network must learn to distinguish. These are just the description of the labels and not the training data themselves. For example in the word data file:

name: {4} 2 I: { 1} 4 T: { 1} 13 I: { 1} 9 T: { 1} 13 ;

This block tells us that the 2-syllable word $[4, 9]$ should be mapped to label 13. Give the set of all different labels in all such files. What does this set have to do with our network's output activations?

The possible labels are `[10,11,12,13,14]`. These values correspond to 5 of our 20 (5 out of 15 in Mirman) output nodes - the label nodes representing the 5 possible words.

C) Out of the two non-word data files, one contains novel-sequence non-words (syllable transitions never encountered in the exposure phase) and one novel-syllable non-words (new syllables). Which one is which? How can you tell?

The file `labelSet2SyllNWc35.ex` contains the novel-sequence non-words, as its inputs range from 0 to 9, just like in the exposure file. The file `labelSet2SyllNW35.ex` contains the novel-syllable non-words, because its inputs are composed of syllables ranging from 10 to 19, which are not included in the exposure file.

2.4 Exposure Phase

Now that you've understood how our data is structured, let's implement the exposure phase of our simulation! First, use the `parse_sequence` method of the provided parser to parse the Lens files into Python lists. Inspect the data to verify you understand their structure compared to the Lens format. In order to convert our data to vector representations, we will define a `Binarizer` object and make use of the `binarize_sequence` method. Inspect the data shape and contents.

[3]:
```python
parser = LensParser()
bnzer = Binarizer(vocab_size=20)

exposure_data = parser.parse_sequence('./exposureSet2Syll35.ex')

# inspect the first 10 tokens of the sequence
print(exposure_data[:10])

X_exposure, y_exposure = bnzer.binarize_sequence(exposure_data)

# inspect one-hot representations. Everything looks fine?
print(X_exposure.shape, y_exposure.shape)
print('-' * 48)
print(X_exposure[:,:10])
print('-' * 48)
print(y_exposure[:,:10])
```

```
[6, 0, 3, 7, 1, 2, 4, 9, 8, 5]
(1, 1000, 20) (1, 1000, 20)
------------------------------------------------
[[[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
  [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
  [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]]
------------------------------------------------
[[[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
  [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]]
```

Now that our model and data are in place, you are ready to simulate the exposure phase! (**Note:** When attempting to re-train your model remember to run the model definition cell again in order to re-initialize it, otherwise you are training on top of your previous weights!)

[ ]:
```
exposure_phase = model.fit(X_exposure, y_exposure, epochs=75, verbose=2)

plt.plot(exposure_phase.history['loss'])
plt.title('Loss (lower is better)')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.show()

plt.plot(exposure_phase.history['categorical_accuracy'])
plt.title('Categorical Accuracy (higher is better)')
plt.ylabel('Categorical Accuracy')
plt.xlabel('Epoch')
plt.show()

plt.plot(exposure_phase.history['mean_squared_error'])
plt.title('Mean Squared Error (lower is better)')
plt.ylabel('MSE')
plt.xlabel('Epoch')
plt.show()

postexposure_weights = model.get_weights()
```

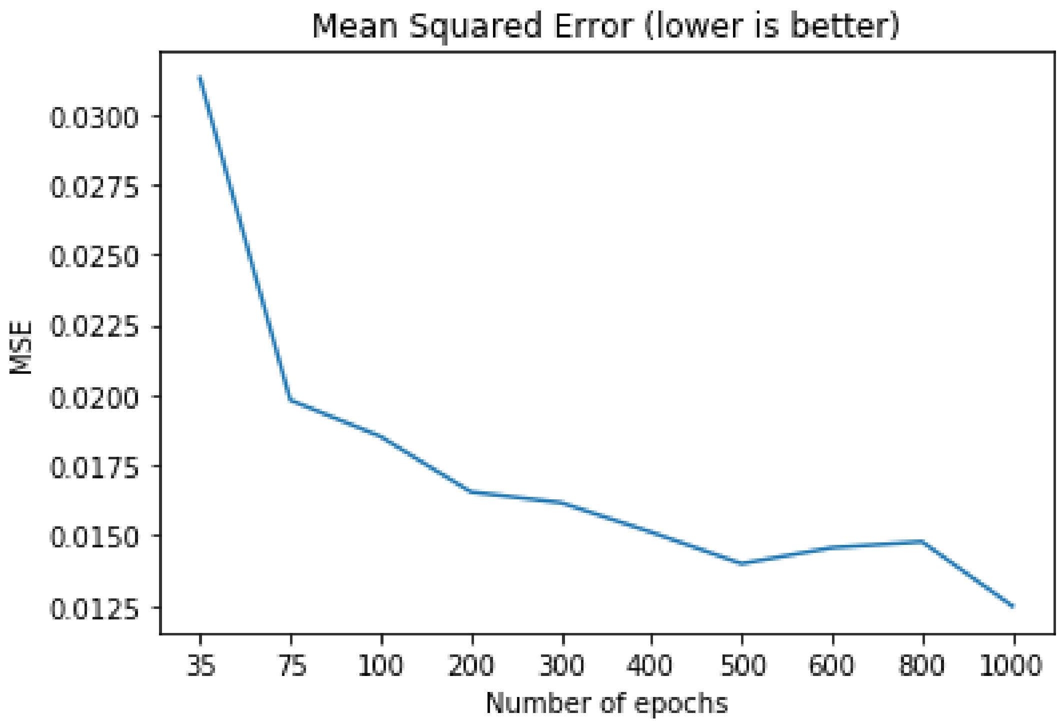
2.5 Exercise 3

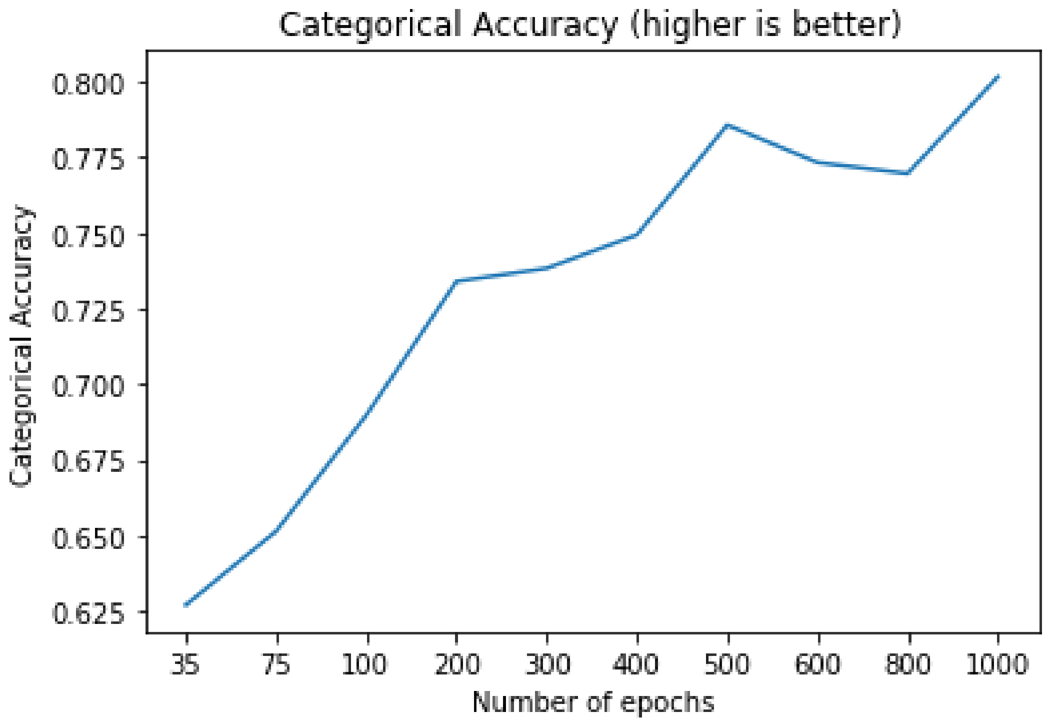
A) Train multiple times and average your results. How much training is needed to get a low error rate? (include an example graph in your report)

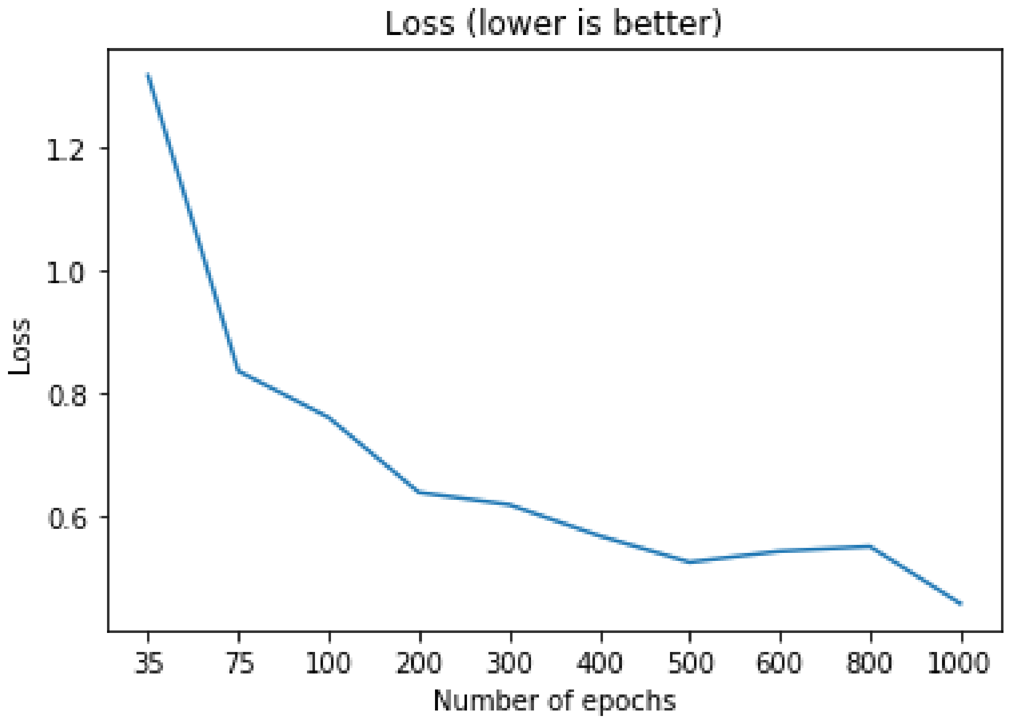
[5]:
```
losses = []
accuracies = []
errors = []
epochs_no = [35, 75, 100, 200, 300, 400, 500, 600, 800, 1000]

for i in range (3):
    for j in range (10):

        model = Sequential([
            SimpleRNN(15, input_shape=(None, 20),
                      return_sequences=True, #doesn't matter since there's only␣
 ↪one hidden layer
                      activation='tanh',
                      name='hidden'),
            TimeDistributed(Dense(20, activation='softmax', name='output'))
        ])

        model.compile(loss='categorical_crossentropy',
                      optimizer=SGD(lr=0.05, momentum=0.9),
                      metrics=['categorical_accuracy', 'mean_squared_error'])

        exposure_phase = model.fit(X_exposure, y_exposure, epochs=epochs_no[j],␣
 ↪verbose=0)

        errors.append(exposure_phase.history['mean_squared_error'][-1])
        accuracies.append(exposure_phase.history['categorical_accuracy'][-1])
        losses.append(exposure_phase.history['loss'][-1])
```

[6]:
```
outcomes = []

index = 0

for epoch_types in range (10):

    #alphabetically - a, e, l

    accu = (accuracies[index]+accuracies[index+10]+accuracies[index+20])/3
    erro = (errors[index]+errors[index+10]+errors[index+20])/3
    loss = (losses[index]+losses[index+10]+losses[index+20])/3

    outcomes.append([accu, erro, loss])

    index += 1
```

[7]:
```
x = [35, 75, 100, 200, 300, 400, 500, 600, 800, 1000]
xi = list(range(len(x)))

plt.plot(xi, [outcomes[0][1], outcomes[1][1], outcomes[2][1], outcomes[3][1],␣
 ↪outcomes[4][1], outcomes[5][1], outcomes[6][1], outcomes[7][1],␣
 ↪outcomes[8][1], outcomes[9][1]])
plt.title('Mean Squared Error (lower is better)')
plt.ylabel('MSE')
plt.xlabel('Number of epochs')
plt.xticks(xi, x)
plt.show()

plt.plot(xi, [outcomes[0][0], outcomes[1][0], outcomes[2][0], outcomes[3][0],␣
 ↪outcomes[4][0], outcomes[5][0], outcomes[6][0], outcomes[7][0],␣
 ↪outcomes[8][0], outcomes[9][0]])
plt.title('Categorical Accuracy (higher is better)')
plt.ylabel('Categorical Accuracy')
plt.xlabel('Number of epochs')
plt.xticks(xi, x)
plt.show()

plt.plot(xi, [outcomes[0][2], outcomes[1][2], outcomes[2][2], outcomes[3][2],␣
 ↪outcomes[4][2], outcomes[5][2], outcomes[6][2], outcomes[7][2],␣
 ↪outcomes[8][2], outcomes[9][2]])
plt.title('Loss (lower is better)')
plt.ylabel('Loss')
plt.xlabel('Number of epochs')
plt.xticks(xi, x)
plt.show()
```




Categorical Accuracy (higher is better)
Categorical Accuracy
0.800
0.775
0.750
0.725
0.700
0.675
0.650
0.625
35
75
100
200
300
400
500
600
800
1000
Number of epochs


Loss (lower is better)
Loss
1.2
1.0
0.8
0.6
35
75
100
200
300
400
500
600
800
1000
Number of epochs

Lowest error was achieved at the value of 800 epochs. However, considering the time required to run these computations, lower numbers (even as low as 35) result in low enough MSEs.

B) How much training did they report in the paper? Would half this training be sufficient? Experiment and explain your results.

200 epochs were reported in the paper (CA 0.713, MSE 0.017, and L 0.660). Halving this number results in a slightly higher MSE (0.019), lower CA (0.663), and higher loss (0.784). Considering the fairly small differences between these values, the training would be sufficient. However, further experimentation would be needed to determine how these differences would affect the label making stage. I,e. the lower number of epochs could result in an insufficient familiarization (and underestimation of the effects of syllable familiarity on label learning), or the NN might not extract the information related to the transitional probabilities and rely on word frequencies instead. The latter would completely sabbotage the investigation of the influence of transitional probabilities on label learning.

2.6 Label Learning Phase

For the label learning phase, we will have to further train our model 4 times, once for every different category of label learning data (word, part-word, novel-syllable non-word and novel-sequence non-word). We will use the `parse_labels` method of our parser to obtain our 5 2-syllable label words. Inspect them to make sure you understand the data format. We then sample 100 examples from such words and binarize. Do our data shapes seem logical?

[24]:
```
# load words label data for label learning phase
word_labels = parser.parse_labels('./labelSet2SyllW35.ex')

# inspect the data. Are they the same as in the input files?
print('Word labels={}'.format(word_labels))

# generate 100 training word examples
word_data = random.choices(word_labels, k=200)
print('First 5 word label examples={}'.format(word_data[:5]))

# convert to one-hot encodings
Xw, yw = bnzer.binarize_labels(word_data)

# inspect the input data shape. Make sense?
print('Word data shape={}, labels shape={}'.format(Xw.shape, yw.shape))
```

```
Word labels=[(1, 2, 10), (8, 5, 11), (6, 0, 12), (4, 9, 13), (3, 7, 14)]
First 5 word label examples=[(8, 5, 11), (4, 9, 13), (1, 2, 10), (1, 2, 10), (1,
2, 10)]
Word data shape=(1, 400, 20), labels shape=(1, 400, 20)
```

Without re-initializing our model (keeping the weights the same as after the exposure phase) we are ready to further train it for the label learning task!

```
[ ]: model.set_weights(postexposure_weights)
     words_learning = model.fit(Xw, yw, epochs=200, verbose=2)

     plt.plot(words_learning.history['loss'])
     plt.title('Loss (lower is better)')
     plt.ylabel('Loss')
     plt.xlabel('Epoch')
     plt.show()

     plt.plot(words_learning.history['categorical_accuracy'])
     plt.title('Categorical Accuracy (higher is better)')
     plt.ylabel('Categorical Accuracy')
     plt.xlabel('Epoch')
     plt.show()

     plt.plot(words_learning.history['mean_squared_error'])
     plt.title('Mean Squared Error (lower is better)')
     plt.ylabel('MSE')
     plt.xlabel('Epoch')
     plt.show()
```

Repeat this process for the other 3 label categories.

```
[ ]: # repeat for the other 3 categories. Don't forget to set the model back to it's␣
      ↪postexposure phase!

     ## implement your code here! ##
     # Learning Part words
     part_words = parser.parse_labels('./labelSet2SyllPW35.ex')
     print('Part Words={}'.format(part_words))
     part_word_data = random.choices(part_words, k=100)
     print('First 5 part word examples={}'.format(part_word_data[:5]))
     Xpw, ypw = bnzer.binarize_labels(part_word_data)

     # Train the model
     model.set_weights(postexposure_weights)
     part_words_learning = model.fit(Xpw, ypw, epochs=200, verbose=2)
```

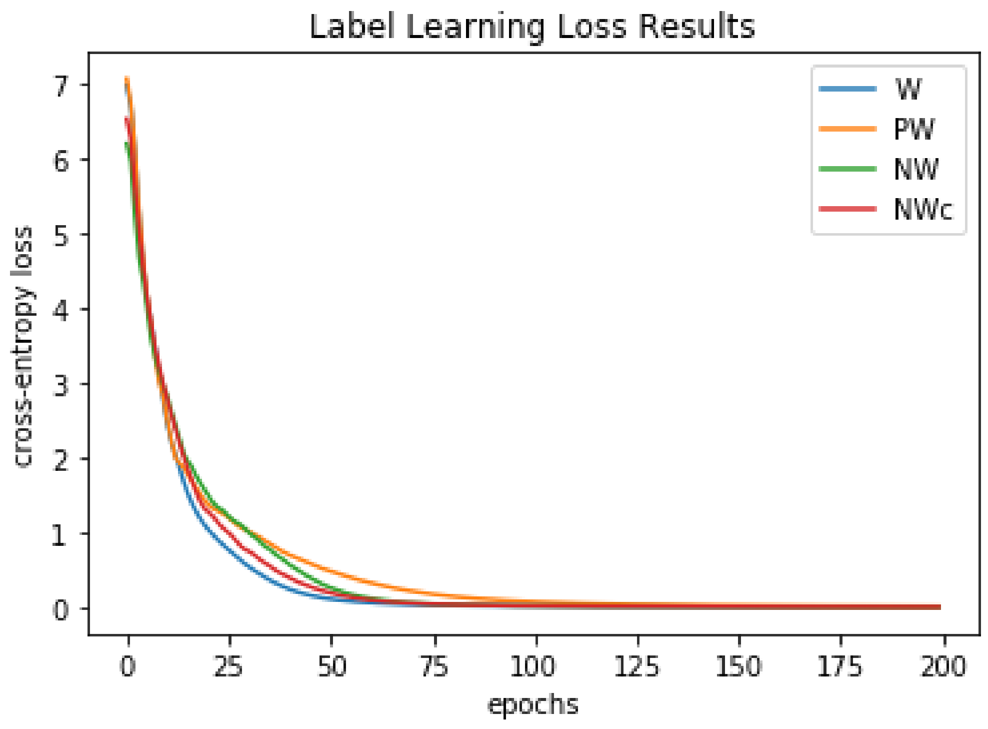
```
[ ]: # Learning Non words
     non_words = parser.parse_labels('./labelSet2SyllNW35.ex')
     print('Non Words = {}'.format(non_words))
     non_word_data = random.choices(non_words, k=100)
     print('First 5 non word examples={}'.format(non_word_data[:5]))
     Xnw, ynw = bnzer.binarize_labels(non_word_data)

     # Train the model
     model.set_weights(postexposure_weights)
```

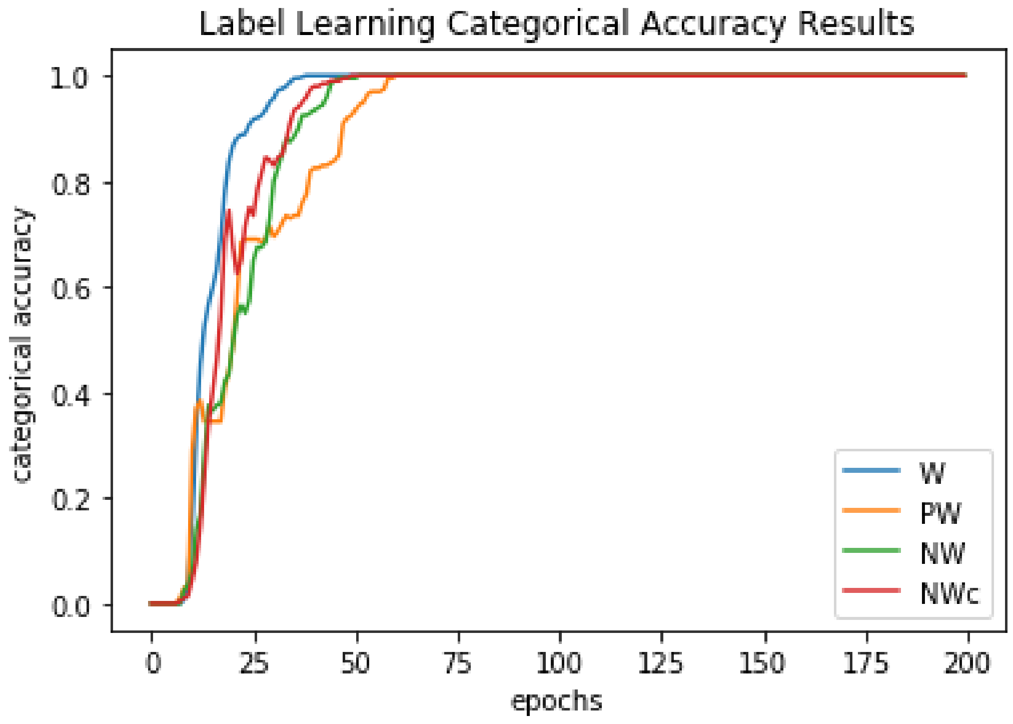
```
non_words_learning = model.fit(Xnw, ynw, epochs=200, verbose=2)
```

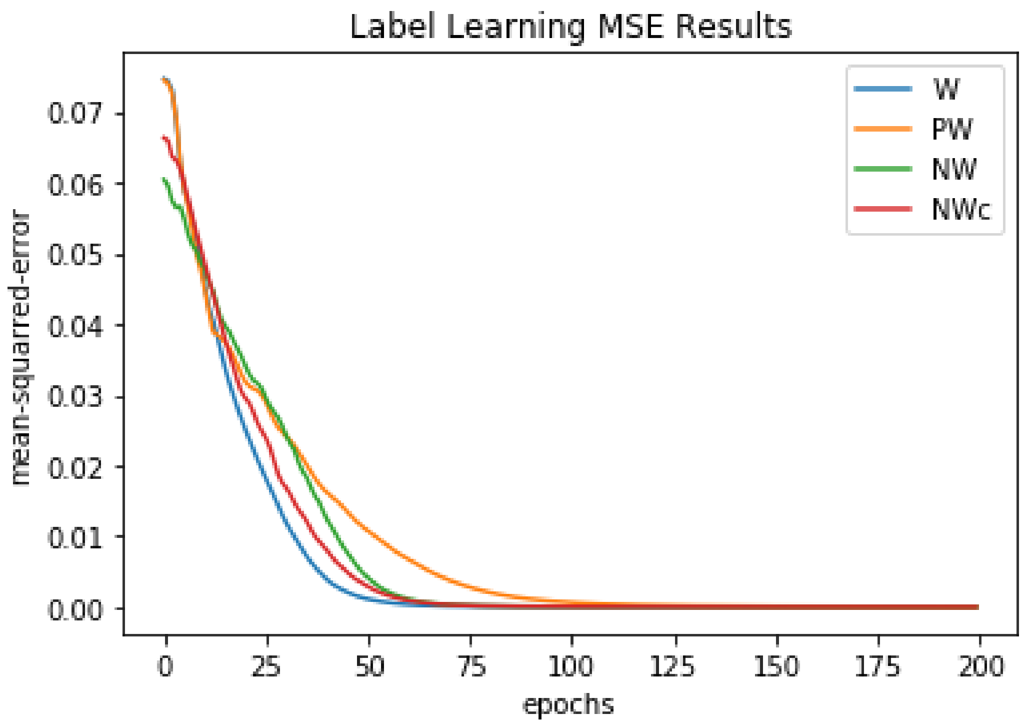
[ ]:
```
# Learning Non words (c)
non_words_c = parser.parse_labels('./labelSet2SyllNWc35.ex')
print('Non Word (c) = {}'.format(non_words_c))
non_word_c_data = random.choices(non_words_c, k=100)
print('First 5 non word (c) examples = {}'.format(non_word_c_data[:5]))
Xnwc, ynwc = bnzer.binarize_labels(non_word_c_data)

# Train the model
model.set_weights(postexposure_weights)
non_words_c_learning = model.fit(Xnwc, ynwc, epochs=200, verbose=2)
```

Then, use the provided `multi_plot` function to compare the learning progress of the four different categories. (**Note:** You might want to take several trials of the four learning steps in order to gain serious insight about the results)

[29]:
```
losses = [words_learning.history['loss'], part_words_learning.history['loss'], \
          non_words_learning.history['loss'], non_words_c_learning.
 ↪history['loss']]

accuracies = [words_learning.history['categorical_accuracy'], \
              part_words_learning.history['categorical_accuracy'], \
              non_words_learning.history['categorical_accuracy'], \
              non_words_c_learning.history['categorical_accuracy']]


MSEs = [words_learning.history['mean_squared_error'], \
        part_words_learning.history['mean_squared_error'], \
        non_words_learning.history['mean_squared_error'], \
        non_words_c_learning.history['mean_squared_error']]


multi_plot(losses, accuracies, MSEs)
```

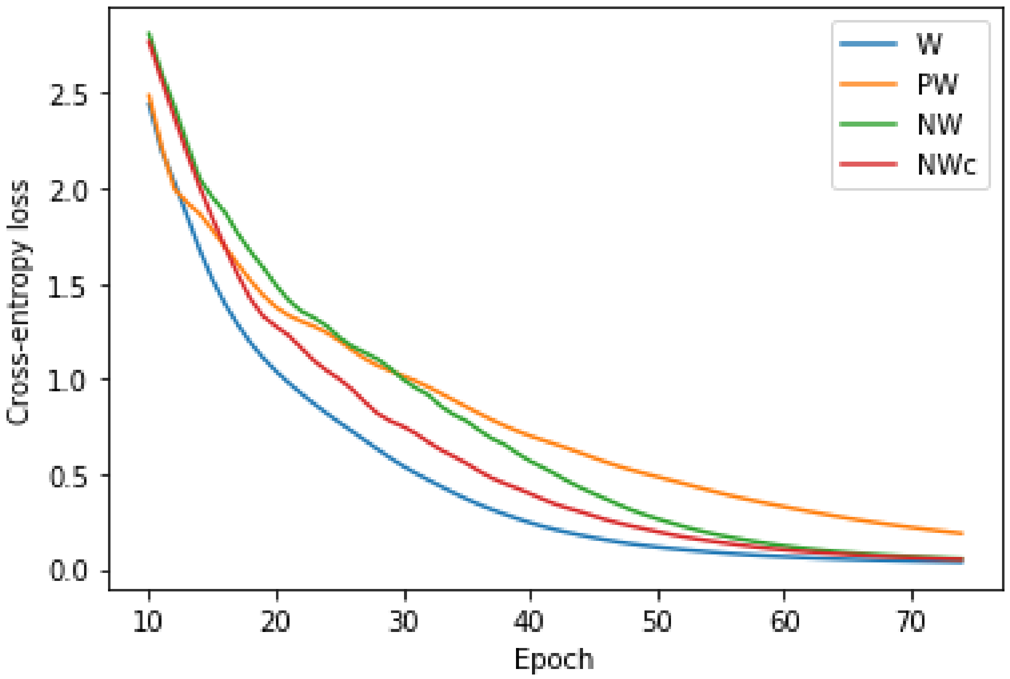

Label Learning Loss Results
cross-entropy loss
epochs
W
PW
NW
NWc


Label Learning Categorical Accuracy Results
categorical accuracy
epochs
W
PW
NW
NWc



[46]:
```
losses_zoom = [words_learning.history['loss'][10:75], part_words_learning.
 ↪history['loss'][10:75], \
          non_words_learning.history['loss'][10:75], non_words_c_learning.
 ↪history['loss'][10:75]]

tick = np.arange(10,75)
plt.plot(tick, losses_zoom[0], label = 'W')
plt.plot(tick, losses_zoom[1], label = 'PW')
plt.plot(tick, losses_zoom[2], label = 'NW')
plt.plot(tick, losses_zoom[3], label = 'NWc')
plt.ylabel('Cross-entropy loss')
plt.xlabel('Epoch')
plt.legend()
```

[46]:
```
<matplotlib.legend.Legend at 0x226eaebcef0>
```



2.7 Exercise 4

A) Why do we need to include novel-syllable nonwords in the simulation? What purpose does it serve?

The two different types of nonwords are included to single out and examine the effects of syllable familiarity on label learning.

B) In the label learning phase, our model must be initialized with the postexposure weights. Explain why.

The purpose of the study was to find new insights relating to the finding that people learn word labels faster if the words have higher transitional probabilities. In order to do so, the humans had to first be exposed to the words in order to learn their transitional probabilities (the simulation had to finish the word based on its first syllable). Only then, after knowing that the network extracted information regarding these probabilities (as opposed to word frequency cues), it could have been investigated how they influence label learning.

C) Report your comparison graph for the different label categories. Do you obtain similar results as those reported in the paper? Discuss any differences.

The most noteworthy part of label learning can be observed in the magnified graph above (epochs 10 to 75). Overall, the results are fairly similar to the ones noted by Mirman - novel syllable nonwords and partwords can be seen to be more difficult to learn than novel sequence nonwords and words. Strikingly however, the order of the first two (NW and PW) is at first reversed (epoch 15 to 25). This has a marginal effect, as between epochs 40 and 75 the trend is identical to Mirman's. The biggest difference between the two graphs (ours and Mirman's seems to occur at the beginning -

in our case the learning seems to go much more smoothly than for Mirman, where a significant "bump" in learning can be observed around epoch 20.

D) Given the distincion between adult-child results in the human experiments and your model's results, would you say that your SRN model is simulating children or adults? Explain your answer.

In the behavioral experiments, infants learned W labels, but not PW or NW, whereas adults learned labels of all three kinds of words. Therefore, our results represent adult learning, since the model learned labels of all kinds of words.

3 Part 2: Modelling Misyak et al. (2010)

Read Misyak, Christiansen and Tomblin (2010). On page 139 you can find information about the human experiments and on page 145 the description of the SRN employed and the constructed training examples.

3.1 Model and training data

Make sure you understand the experiment details, as you will have to create your own data for simulating it. The data in our case are strings consisting of 3 nonword tokens; A starting nonword syllable (a, b or c), a middle bisyllabic nonword $X_1, ..., X_{24}$ and an ending token (d, e or f). What combinations are considered *grammatical* in the paper? The purpose of this simulation is to investigate whether the SRN model can learn to predict the ending token of each string after being exposed to the grammatical dependencies for several examples.

[47]:
```
import itertools

all_tokens = [
 'pel',
 'dak',
 'vot',
 'rid',
 'jic',
 'tood',
 'wadim',
 'kicey',
 'puser',
 'fengle',
 'coomo',
 'loga',
 'gople',
 'taspu',
 'hiftam',
 'deecha',
 'vamey',
 'skiger',
 'benez',
 'gensim',
 'feenam',
 'laelijeen',
 'chila',
 'roosa',
 'plizet',
 'balip',
 'malsig',
 'suleb',
 'nilbo',
 'wiffle'
]

start_tokens = all_tokens[:3]  ## {a,b,c}
end_tokens = all_tokens[3:6]   ## {d,e,f}
middle_tokens = all_tokens[6:] ## {X1,...,X24}

grammar_map = {
    'pel' : 'rid',    ## a->d
    'dak' : 'jic',    ## b->e
    'vot' : 'tood'    ## c->f
}
```

As described in the paper, the simulation consists of three learning phases: * **Phase 1:** During this phase the model is presented with 6 grammatical blocks, each consisting of 72 *unique* strings containing all middle tokens. During this phase we expect the model to learn the inehrent dependencies present in our training exaxmples. * **Phase 2:** In the second phase the model is presented with 1 ungrammatical block containing 24 strings of all middle tokens with violated dependencies. This is meant to simulate the disruption in the predictive capacity of human subjects. * **Phase 3:** During this "recovery" phase, the model is presented again with 1 grammatical block like that of the first phase.

The code below will provide you with all unique combinations of the 24 middle tokens that hold the defined grammatic dependencies for each grammatical block:

```
[48]: def generate_grammatical_block():
          return [(*r, grammar_map[r[0]]) for r in itertools.product(start_tokens,␣
       ↪middle_tokens)]

      # let's inspect a generated block. Does it look grammatical according to the␣
       ↪paper?
      print(generate_grammatical_block())
```

```
[('pel', 'wadim', 'rid'), ('pel', 'kicey', 'rid'), ('pel', 'puser', 'rid'),
('pel', 'fengle', 'rid'), ('pel', 'coomo', 'rid'), ('pel', 'loga', 'rid'),
('pel', 'gople', 'rid'), ('pel', 'taspu', 'rid'), ('pel', 'hiftam', 'rid'),
('pel', 'deecha', 'rid'), ('pel', 'vamey', 'rid'), ('pel', 'skiger', 'rid'),
('pel', 'benez', 'rid'), ('pel', 'gensim', 'rid'), ('pel', 'feenam', 'rid'),
('pel', 'laelijeen', 'rid'), ('pel', 'chila', 'rid'), ('pel', 'roosa', 'rid'),
('pel', 'plizet', 'rid'), ('pel', 'balip', 'rid'), ('pel', 'malsig', 'rid'),
('pel', 'suleb', 'rid'), ('pel', 'nilbo', 'rid'), ('pel', 'wiffle', 'rid'),
('dak', 'wadim', 'jic'), ('dak', 'kicey', 'jic'), ('dak', 'puser', 'jic'),
('dak', 'fengle', 'jic'), ('dak', 'coomo', 'jic'), ('dak', 'loga', 'jic'),
('dak', 'gople', 'jic'), ('dak', 'taspu', 'jic'), ('dak', 'hiftam', 'jic'),
('dak', 'deecha', 'jic'), ('dak', 'vamey', 'jic'), ('dak', 'skiger', 'jic'),
('dak', 'benez', 'jic'), ('dak', 'gensim', 'jic'), ('dak', 'feenam', 'jic'),
('dak', 'laelijeen', 'jic'), ('dak', 'chila', 'jic'), ('dak', 'roosa', 'jic'),
('dak', 'plizet', 'jic'), ('dak', 'balip', 'jic'), ('dak', 'malsig', 'jic'),
('dak', 'suleb', 'jic'), ('dak', 'nilbo', 'jic'), ('dak', 'wiffle', 'jic'),
('vot', 'wadim', 'tood'), ('vot', 'kicey', 'tood'), ('vot', 'puser', 'tood'),
('vot', 'fengle', 'tood'), ('vot', 'coomo', 'tood'), ('vot', 'loga', 'tood'),
('vot', 'gople', 'tood'), ('vot', 'taspu', 'tood'), ('vot', 'hiftam', 'tood'),
('vot', 'deecha', 'tood'), ('vot', 'vamey', 'tood'), ('vot', 'skiger', 'tood'),
('vot', 'benez', 'tood'), ('vot', 'gensim', 'tood'), ('vot', 'feenam', 'tood'),
('vot', 'laelijeen', 'tood'), ('vot', 'chila', 'tood'), ('vot', 'roosa',
'tood'), ('vot', 'plizet', 'tood'), ('vot', 'balip', 'tood'), ('vot', 'malsig',
'tood'), ('vot', 'suleb', 'tood'), ('vot', 'nilbo', 'tood'), ('vot', 'wiffle',
'tood')]
```

Fill the code snippet below to generate ungrammatical blocks of 24 strings for the second phase of our simulation. (***Hint:*** Dependencies now should like aXe, aXf, bXd, bXf, cXd and cXe)

[50]:
```python
non_grammar_map = {
    'pel' : ['jic', 'tood'],
    'dak' : ['rid', 'tood'],
    'vot' : ['rid', 'jic']
}

def generate_ungrammatical_block(num_strings=24):
    return [(*r, non_grammar_map[r[0]][np.random.choice([0,1])]) for r in␣
 ↪itertools.product(start_tokens, middle_tokens)][:num_strings]

# inspect an ungrammatical block.
print(generate_ungrammatical_block())
```

```
[('pel', 'wadim', 'tood'), ('pel', 'kicey', 'tood'), ('pel', 'puser', 'jic'),
('pel', 'fengle', 'tood'), ('pel', 'coomo', 'tood'), ('pel', 'loga', 'jic'),
('pel', 'gople', 'tood'), ('pel', 'taspu', 'jic'), ('pel', 'hiftam', 'tood'),
('pel', 'deecha', 'jic'), ('pel', 'vamey', 'jic'), ('pel', 'skiger', 'jic'),
('pel', 'benez', 'tood'), ('pel', 'gensim', 'jic'), ('pel', 'feenam', 'jic'),
('pel', 'laelijeen', 'jic'), ('pel', 'chila', 'jic'), ('pel', 'roosa', 'jic'),
('pel', 'plizet', 'tood'), ('pel', 'balip', 'jic'), ('pel', 'malsig', 'tood'),
('pel', 'suleb', 'jic'), ('pel', 'nilbo', 'jic'), ('pel', 'wiffle', 'tood')]
```

In our simulation, as we are employing neural network models, all of our data need to be in continuous form. As a result, we will need to encode our vocabulary $\{a, b, c, d, e, f, X_1, ..., X_{24}\}$ into vector representations. What is the size of our vocabulary? Then each nonword syllable or bisyllable should be mapped to a one-hot encoded vector of such size with a one in the index of the present syllable.

[51]:
```
from sklearn.preprocessing import LabelBinarizer

# generate one grammatical block of 72 strings, one ungrammatical of default 24␣
↪strings
# and one "recovery" block of 72 grammatical strings
gram_block = random.sample(generate_grammatical_block(), 72)
ungr_block = random.sample(generate_ungrammatical_block(), 24)
rec_block = random.sample(generate_grammatical_block(), 72)

# define an one-hot encoder
encoder = LabelBinarizer()
encoder.fit(all_tokens)

# flatten the strings and transform all tokens to one-hot vectors
gram_tokens = [token for tokens in gram_block for token in tokens]
ungr_tokens = [token for tokens in ungr_block for token in tokens]
rec_tokens = [token for tokens in rec_block for token in tokens]

gram_vectors = encoder.transform(gram_tokens)
ungr_vectors = encoder.transform(ungr_tokens)
rec_vectors = encoder.transform(rec_tokens)

# define training data and targets by shifting the sequence one token to the␣
↪left
X_gram = gram_vectors[np.newaxis, :]
y_gram = np.append(gram_vectors[1:], gram_vectors[:1], axis=0)[np.newaxis, :]

X_ungr = ungr_vectors[np.newaxis, :]
y_ungr = np.append(ungr_vectors[1:], ungr_vectors[:1], axis=0)[np.newaxis, :]

X_rec = rec_vectors[np.newaxis, :]
y_rec = np.append(rec_vectors[1:], rec_vectors[:1], axis=0)[np.newaxis, :]

# what is the shape of our data? Does this match descriptions in the paper?
print(X_gram.shape, y_gram.shape, X_ungr.shape, y_ungr.shape, X_rec.shape,␣
↪y_rec.shape)
```

```
(1, 216, 30) (1, 216, 30) (1, 72, 30) (1, 72, 30) (1, 216, 30) (1, 216, 30)
```

Right! Now to our model. By now, you should be experienced in defining SRNs. Read the description of the network parameters from the paper and fill the code below to define the simulation model. You might want to take a look at the Elman implementation from Lab 1 if unsure as to which should be the shape of input/output.

[73]:
```
## fillin _ with the correct parameter
model = Sequential([
    SimpleRNN(15, input_shape=(None, 30),
              return_sequences=True,
              activation='tanh',
              recurrent_initializer=RandomUniform(minval=-1, maxval=1),
              name='hidden'),
    TimeDistributed(Dense(30, activation='softmax'), name='output')
])

model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=0.1, momentum=0.8),
              metrics=['categorical_accuracy', 'mean_squared_error'])

print(model.summary())
```

```
Model: "sequential_36"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
hidden (SimpleRNN)           (None, None, 15)          690
_________________________________________________________________
output (TimeDistributed)     (None, None, 30)          480
=================================================================
Total params: 1,170
Trainable params: 1,170
Non-trainable params: 0
_________________________________________________________________
None
```

3.2 Simulating the Three Learning Phases

Run the three learning phases in order and plot your results. Instead of training our model a single epoch in each block, we scale it up to further smooth the learning process and provide better visualization insight.

[74]:
```
scale_factor = 5

print('-'*100)
print('Phase 1: 6 grammatical blocks of 72 strings..\n')
phase_1 = model.fit(X_gram, y_gram, epochs=6*scale_factor, verbose=2)
print('-'*100)

print('Phase 2: 1 ungrammatical block of 24 strings..\n')
phase_2 = model.fit(X_ungr, y_ungr, epochs=1*scale_factor, verbose=2)
print('-'*100)

print('Phase 3: 1 recovery block of 72 strings..\n')
phase_3 = model.fit(X_rec, y_rec, epochs=1*scale_factor, verbose=2)
print('-'*100)
```

```
--------------------------------------------------------------------------------
--------------------
Phase 1: 6 grammatical blocks of 72 strings..

Train on 1 samples
Epoch 1/30
1/1 - 1s - loss: 3.8459 - categorical_accuracy: 0.0139 - mean_squared_error:
0.0339
Epoch 2/30
1/1 - 0s - loss: 3.7171 - categorical_accuracy: 0.0139 - mean_squared_error:
0.0333
Epoch 3/30
1/1 - 0s - loss: 3.4673 - categorical_accuracy: 0.0463 - mean_squared_error:
0.0325
Epoch 4/30
1/1 - 0s - loss: 3.2333 - categorical_accuracy: 0.1111 - mean_squared_error:
0.0318
Epoch 5/30
1/1 - 0s - loss: 3.1424 - categorical_accuracy: 0.0972 - mean_squared_error:
0.0315
Epoch 6/30
1/1 - 0s - loss: 3.0697 - categorical_accuracy: 0.1157 - mean_squared_error:
0.0312
Epoch 7/30
1/1 - 0s - loss: 2.9875 - categorical_accuracy: 0.1435 - mean_squared_error:
0.0309
Epoch 8/30
1/1 - 0s - loss: 2.9568 - categorical_accuracy: 0.1389 - mean_squared_error:
0.0308
Epoch 9/30
1/1 - 0s - loss: 2.9382 - categorical_accuracy: 0.1481 - mean_squared_error:
0.0307
Epoch 10/30
1/1 - 0s - loss: 2.9240 - categorical_accuracy: 0.1759 - mean_squared_error:
0.0306
Epoch 11/30
1/1 - 0s - loss: 2.9089 - categorical_accuracy: 0.1898 - mean_squared_error:
0.0306
Epoch 12/30
1/1 - 0s - loss: 2.8974 - categorical_accuracy: 0.1574 - mean_squared_error:
0.0305
Epoch 13/30
1/1 - 0s - loss: 2.8911 - categorical_accuracy: 0.1157 - mean_squared_error:
0.0305
Epoch 14/30
1/1 - 0s - loss: 2.8833 - categorical_accuracy: 0.1157 - mean_squared_error:
0.0305
Epoch 15/30
```

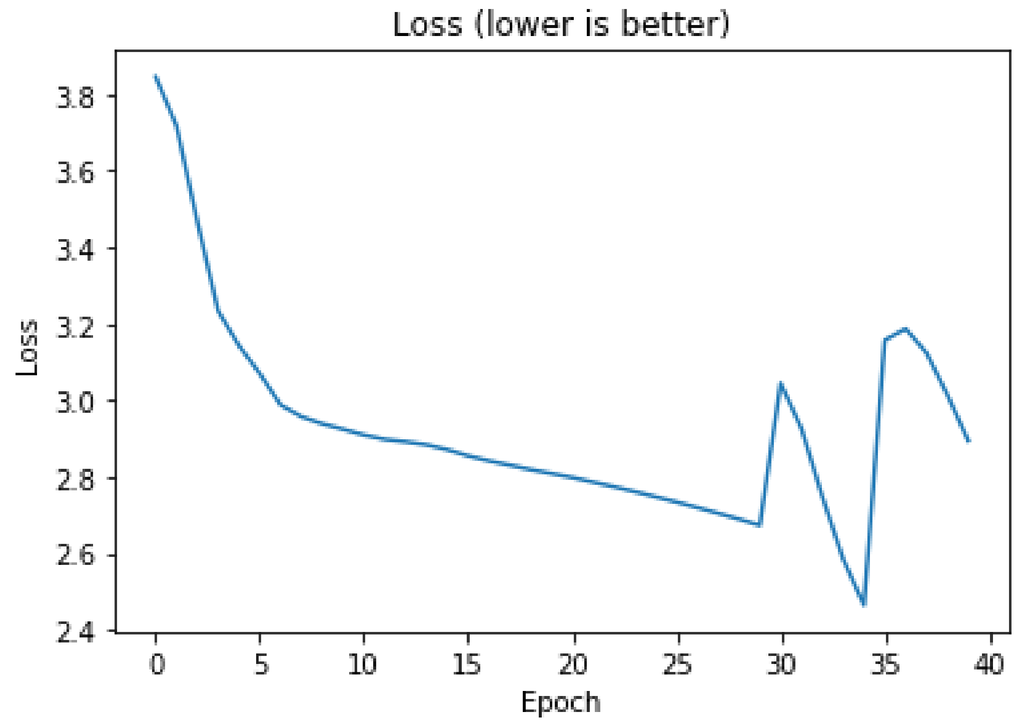
```
1/1 - 0s - loss: 2.8704 - categorical_accuracy: 0.1250 - mean_squared_error:
0.0304
Epoch 16/30
1/1 - 0s - loss: 2.8542 - categorical_accuracy: 0.1944 - mean_squared_error:
0.0303
Epoch 17/30
1/1 - 0s - loss: 2.8411 - categorical_accuracy: 0.3148 - mean_squared_error:
0.0302
Epoch 18/30
1/1 - 0s - loss: 2.8300 - categorical_accuracy: 0.3333 - mean_squared_error:
0.0301
Epoch 19/30
1/1 - 0s - loss: 2.8186 - categorical_accuracy: 0.3009 - mean_squared_error:
0.0300
Epoch 20/30
1/1 - 0s - loss: 2.8085 - categorical_accuracy: 0.3009 - mean_squared_error:
0.0300
Epoch 21/30
1/1 - 0s - loss: 2.7979 - categorical_accuracy: 0.3102 - mean_squared_error:
0.0299
Epoch 22/30
1/1 - 0s - loss: 2.7859 - categorical_accuracy: 0.3148 - mean_squared_error:
0.0298
Epoch 23/30
1/1 - 0s - loss: 2.7738 - categorical_accuracy: 0.3102 - mean_squared_error:
0.0297
Epoch 24/30
1/1 - 0s - loss: 2.7610 - categorical_accuracy: 0.3333 - mean_squared_error:
0.0296
Epoch 25/30
1/1 - 0s - loss: 2.7473 - categorical_accuracy: 0.3565 - mean_squared_error:
0.0295
Epoch 26/30
1/1 - 0s - loss: 2.7336 - categorical_accuracy: 0.3704 - mean_squared_error:
0.0294
Epoch 27/30
1/1 - 0s - loss: 2.7192 - categorical_accuracy: 0.3657 - mean_squared_error:
0.0293
Epoch 28/30
1/1 - 0s - loss: 2.7043 - categorical_accuracy: 0.3611 - mean_squared_error:
0.0292
Epoch 29/30
1/1 - 0s - loss: 2.6890 - categorical_accuracy: 0.3611 - mean_squared_error:
0.0290
Epoch 30/30
1/1 - 0s - loss: 2.6727 - categorical_accuracy: 0.3657 - mean_squared_error:
0.0289
--------------------------------------------------------------------------------
```

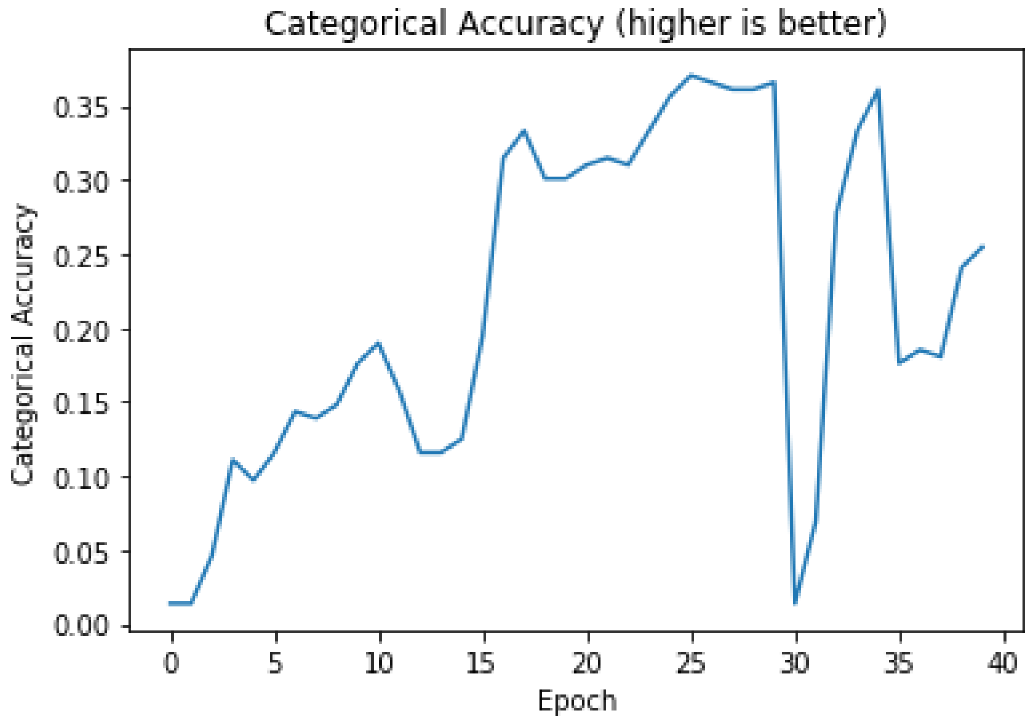
```
--------------------
Phase 2: 1 ungrammatical block of 24 strings..

Train on 1 samples
Epoch 1/5
1/1 - 1s - loss: 3.0446 - categorical_accuracy: 0.0139 - mean_squared_error:
0.0316
Epoch 2/5
1/1 - 0s - loss: 2.9256 - categorical_accuracy: 0.0694 - mean_squared_error:
0.0308
Epoch 3/5
1/1 - 0s - loss: 2.7483 - categorical_accuracy: 0.2778 - mean_squared_error:
0.0295
Epoch 4/5
1/1 - 0s - loss: 2.5846 - categorical_accuracy: 0.3333 - mean_squared_error:
0.0282
Epoch 5/5
1/1 - 0s - loss: 2.4644 - categorical_accuracy: 0.3611 - mean_squared_error:
0.0272
--------------------------------------------------------------------------------
--------------------
Phase 3: 1 recovery block of 72 strings..

Train on 1 samples
Epoch 1/5
1/1 - 0s - loss: 3.1568 - categorical_accuracy: 0.1759 - mean_squared_error:
0.0322
Epoch 2/5
1/1 - 0s - loss: 3.1866 - categorical_accuracy: 0.1852 - mean_squared_error:
0.0321
Epoch 3/5
1/1 - 0s - loss: 3.1233 - categorical_accuracy: 0.1806 - mean_squared_error:
0.0314
Epoch 4/5
1/1 - 0s - loss: 3.0139 - categorical_accuracy: 0.2407 - mean_squared_error:
0.0307
Epoch 5/5
1/1 - 0s - loss: 2.8942 - categorical_accuracy: 0.2546 - mean_squared_error:
0.0300
--------------------------------------------------------------------------------
--------------------
```

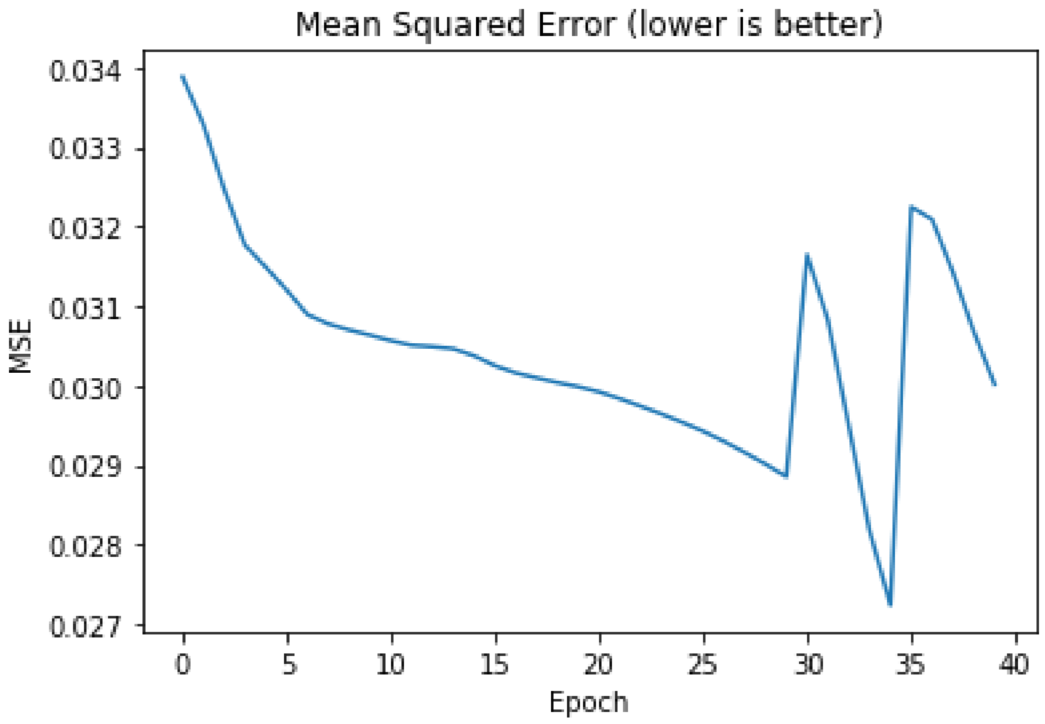
Make use of the `loss`, `categorical_accuracy` and `mean_squarred_error` keys of the the three phases history dictionaries (e.g. `phase_1.history` for phase 1) to plot the learning progress of the model in our simulation in a single plot.

[75]:
```
import matplotlib.pyplot as plt
```

```
loss = phase_1.history['loss'] + phase_2.history['loss'] + phase_3.
↪history['loss']
accu = phase_1.history['categorical_accuracy'] + phase_2.
↪history['categorical_accuracy'] \
          + phase_3.history['categorical_accuracy']
mse = phase_1.history['mean_squared_error'] + phase_2.
↪history['mean_squared_error'] \
          + phase_3.history['mean_squared_error']


plt.plot(loss)
plt.title('Loss (lower is better)')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.show()

plt.plot(accu)
plt.title('Categorical Accuracy (higher is better)')
plt.ylabel('Categorical Accuracy')
plt.xlabel('Epoch')
plt.show()

plt.plot(mse)
plt.title('Mean Squared Error (lower is better)')
plt.ylabel('MSE')
plt.xlabel('Epoch')
plt.show()
```


Loss (lower is better)
Loss
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
0
5
10
15
20
25
30
35
40
Epoch


Categorical Accuracy (higher is better)
Categorical Accuracy
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
5
10
15
20
25
30
35
40
Epoch



3.3 Prediction Task and Network Analysis

For the last phase of the simulation, we must inspect whether the model is indeed capable of predicting the last token for 12 grammatical strings.

[64]:
```
pred_block = random.sample(generate_grammatical_block(), 12)

# lets view our 12 prediction strings
print(pred_block)
```

```
[('pel', 'fengle', 'rid'), ('dak', 'skiger', 'jic'), ('dak', 'taspu', 'jic'),
('vot', 'hiftam', 'tood'), ('vot', 'gensim', 'tood'), ('vot', 'gople', 'tood'),
('vot', 'deecha', 'tood'), ('vot', 'malsig', 'tood'), ('pel', 'coomo', 'rid'),
('vot', 'puser', 'tood'), ('pel', 'plizet', 'rid'), ('pel', 'wadim', 'rid')]
```

In order to know what exactly our model predicts, we will create a sequence consisting of all $12 * 3 = 36$ tokens in the prediction block and pass it to our model for inference. We use our one-hot encoder to decode the softmax model predictions back to the original token strings for inspection:

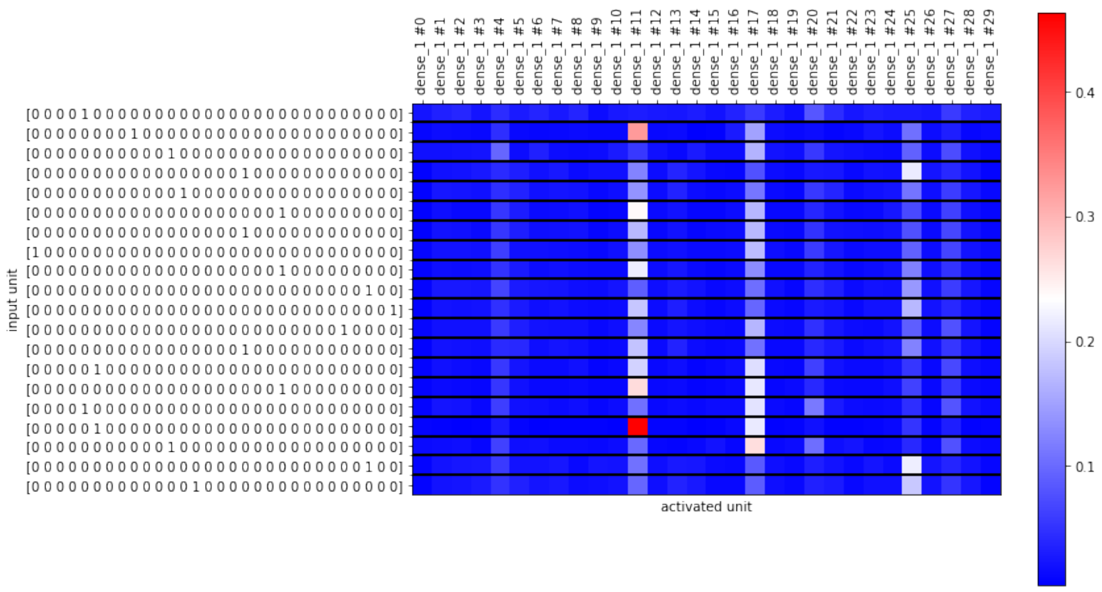
[156]:
```
# convert data to one-hot vectors and make model predictions
pred_tokens = [token for tokens in pred_block for token in tokens]
pred_vectors = encoder.transform(pred_tokens)
X_pred = pred_vectors[np.newaxis, :]
```

```
# decode and inspect
pre = model.predict(X_pred)[0]
predictions = encoder.inverse_transform(np.array(pre))

print('12-item prediction task input tokens:\n{}'.format(pred_tokens))
print('\n12-item prediction task output predictions:\n{}'.format(predictions))
```

```
12-item prediction task input tokens:
['pel', 'fengle', 'rid', 'dak', 'skiger', 'jic', 'dak', 'taspu', 'jic', 'vot',
'hiftam', 'tood', 'vot', 'gensim', 'tood', 'vot', 'gople', 'tood', 'vot',
'deecha', 'tood', 'vot', 'malsig', 'tood', 'pel', 'coomo', 'rid', 'vot',
'puser', 'tood', 'pel', 'plizet', 'rid', 'pel', 'wadim', 'rid']

12-item prediction task output predictions:
['tood' 'pel' 'jic' 'pel' 'jic' 'pel' 'jic' 'jic' 'pel' 'tood' 'tood'
 'pel' 'tood' 'jic' 'pel' 'tood' 'jic' 'pel' 'tood' 'tood' 'pel' 'tood'
 'jic' 'pel' 'jic' 'jic' 'jic' 'pel' 'jic' 'pel' 'jic' 'jic' 'jic' 'jic'
 'jic' 'jic']
```

Inspect the predictions of the model. Is it indeed able to correctly infer the ending syllable token after seeing the starting and the middle ones? What percentage of the 12 strings were correctly classified? Is this close to your resulting accuracy in your model? You can also make use of the visualization utilities implemented for Lab 1 to inspect the state of our SRN for the Misyak et al. simulation. (Remember to copy paste the necessary python files `plot_functions.py` and `utils.py` to your current working directory)

[113]:
```
from plot_functions import plot_activation_matrix

plt.figure(figsize=(10,8))
plot_activation_matrix(model, X_gram, layers=[model.layers[1]],
                       subset=(0, slice(0, 20), slice(None)),
                       show_values=False)
```



3.4 Exercise 5

A) How many weight parameters does the SRN model have in total? How many for the hidden, context and dense layers separately?

As can be seen above in the model summary, the SRN has 1170 connections. In the hidden layer, there are 30x15=450 weights, and 15x15=225 recurrent weights. This is accompanied by 15 bias connections (one per each hidden node). In the dense time distributed layer, there are 15x30=450 weights accompanied by 30 bias edges (one per each dense node).

B) What is the loss function that you utilize for training the model. What are the evaluation metrics? Do these numbers represent in their absolute value how well the model is able to predict the end token for each string? Explain your answer.

We utilize the **Categorical Cross-Entropy** loss function. It returns the probability over multiple subsequent strings for a given input string, in the case of our experiment.

The evaluation metrics are:

```
* Categorical Accuracy: Categorical accuracy computes the number of correct classifications pe
* Mean Squared Error: Computes the mean error of classification over all 30 classes.
```

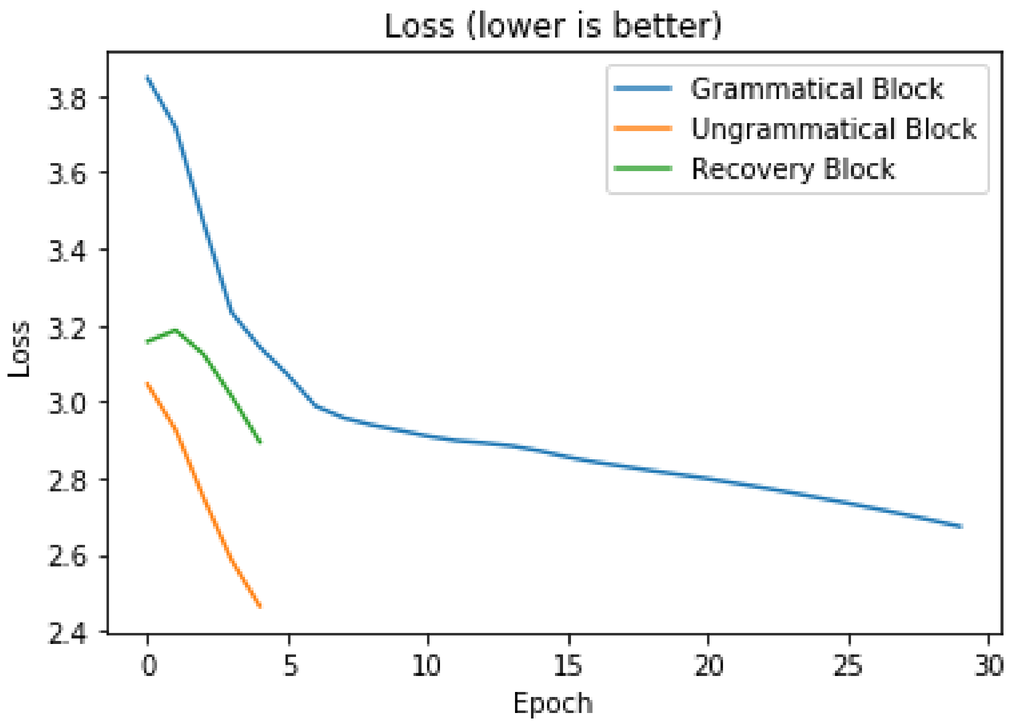
The categorical accuracy is based on the truth values for the end of the string and tells us, for each class, how frequently the model classifies an input into the correct class. The function does not tell us how frequently each class is encountered in the training process (are there are inherent skews to one particular string in the training set).

The MSE returns the error of the model over all classes for a given epoch. This helps us understand if the model is learning in the training process, but we are unable to quantify how well the model learns. We also do not get any information on the class-wise error of the model from this metric.

C) How does learning proceed? Include in your report information (table/plot) about the error metrics after each block and at the end of the training. Do you see the effect of the 24 ungrammatical trials in the error? How does learning in the recovery period proceed? Briefly but precisely, describe how it went.

As in the plot below, we notice that the model learns grammatical representations that it is exposed to in the grammatical block. In the ungrammatical block, the pretrained model from the exposure phase learns ungrammatical representations in a short amount of time, probably due to the learned representations from the first block. This is signified by the sharp decrease in loss (orange line). In the recovery scenario, we notice that the loss is higher for the first epoch as compared to the loss from the grammatical block. This shows us that the ungrammatical block had an effect in updating the model's knowledge. Thus in the recovery period, given the limited number of epochs, the model is only able to recover its learning, achieving almost similar losses as in the grammatical block. This shows that the model is able to retain what it learnt in the grammatical block from the examples it sees in the recovery block.

[76]:
```
plt.plot(phase_1.history['loss'], label="Grammatical Block")
plt.plot(phase_2.history['loss'], label="Ungrammatical Block")
plt.plot(phase_3.history['loss'], label="Recovery Block")
plt.legend()
plt.title('Loss (lower is better)')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.show()
```



D) In the Misyak et al. study they took the mean score after 30 runs with different random weight initilization. What was the purpose of this? (What does it simulate?)

This manipulation was deployed to simulate individual differences between learners.

E) For the 12-item predictiont task, Misyak et al. calculated the Luce ratio difference scores to determine the network's selection. Do this calculation for the 12-item prediction task and include it in your report.

[209]:
```
import pandas as pd

preds = pd.DataFrame(pre[1:,])
cols = ["w/e" for x in range(30)]
cols[25] = "tood"
cols[11] = "jic"
cols[17] = "rid"
preds.columns = cols
preds.drop([0])
preds.index = np.asarray(pred_block).reshape(36,1)[0:35]
```

[234]:
```
final = pd.DataFrame(np.sort(preds.values)[:,-2:],␣
 ↪columns=['2nd-largest','largest'])
final.index = np.asarray(pred_block).reshape(36,1)[0:35]
final['jic'] = preds['jic']
final['rid'] = preds['rid']
final['tood'] = preds['tood']

final.iloc[[0,3,6,9,12,15,18,21,24,27,30,33]]
```

[234]:
```
        2nd-largest   largest       jic       rid      tood
(pel,)     0.094825  0.113297  0.094825  0.113297  0.024695
(dak,)     0.242230  0.358249  0.358249  0.242230  0.055348
(dak,)     0.158424  0.225340  0.225340  0.158424  0.093097
(vot,)     0.155521  0.196527  0.155521  0.083402  0.196527
(vot,)     0.154914  0.173092  0.173092  0.104420  0.154914
(vot,)     0.132055  0.209671  0.209671  0.120728  0.132055
(vot,)     0.176218  0.184711  0.176218  0.109139  0.184711
(vot,)     0.144833  0.164795  0.164795  0.105616  0.144833
(pel,)     0.159360  0.169023  0.169023  0.159360  0.087477
(vot,)     0.152106  0.192699  0.192699  0.124524  0.152106
(pel,)     0.157484  0.177856  0.177856  0.157484  0.086375
(pel,)     0.184297  0.186354  0.186354  0.184297  0.063077
```

In the above dataframe, all the information required to calculate the Luce difference can be seen. First of all, we have to identify the correct trials - this can be done by comparing the value 'largest' to the grammar-specific words from the last three columns. For example, if for a 'pel' row, the values 'largest' and 'rid' are the same, then that trial is counter as a correct one. Below, a new dataframe consisting of correct trials only is extracted:

[239]:
```
final.iloc[[0,3,6,9,18]]
```

[239]:
```
        2nd-largest   largest       jic       rid      tood
(pel,)     0.094825  0.113297  0.094825  0.113297  0.024695
(dak,)     0.242230  0.358249  0.358249  0.242230  0.055348
(dak,)     0.158424  0.225340  0.225340  0.158424  0.093097
(vot,)     0.155521  0.196527  0.155521  0.083402  0.196527
(vot,)     0.176218  0.184711  0.176218  0.109139  0.184711
```

After extracting this information, we can start calculating the difference. Since our NN doesn't perceive stimuli as a human would (there isn't really a 'foil'), we will assume that the foil stimulus is the one with 2nd highest activation. Another important thing to consider is that due to our NN using softmax function, all final activations sum up to 1 (and therefore instead of calculating ratios, and just take the activations as they are. The formula for the Luce difference is:

$$Luce = targetactivation - foilactivation \tag{1}$$

In order to calculate it for the entire task, we will simply calculate all of the Luce differences and take their average:

$$Luce = \frac{(0.113297 - 0.094825) + (0.358249 - 0.242230) + (0.225340 - 0.158424) + (0.166527 - 0.155521) + (0.184}{5} \tag{2}$$

3.5 Part 3: Extending Misyak et al.

Now that you have recreated the simulation from the Misyak et al. paper, you are ready to extend it. What would happen if instead of one word, there were two words in the middle position? Using the same network (don't add addtional hidden layers!) create new training data that differ from the original by having two words in the middle position. Run the new simulation.

[250]:
```
def extended_grammatical_block():
    return [(*r, grammar_map[r[0]]) for r in itertools.product(start_tokens,␣
 ↪middle_tokens, middle_tokens)]

def extended_ungrammatical_block(num_strings=24):
    return [(*r, non_grammar_map[r[0]][np.random.choice([0,1])]) for r in␣
 ↪itertools.product(start_tokens, middle_tokens, middle_tokens)][:num_strings]



# Generate and reshape the data as we did for Misyak simulations
ext_gram_block = random.sample(extended_grammatical_block(), 72)
ext_ungr_block = random.sample(extended_ungrammatical_block(), 24)
ext_rec_block = random.sample(extended_grammatical_block(), 72)

ext_encoder = LabelBinarizer()
ext_encoder.fit(all_tokens)
```

```
ext_gram_tokens = [token for tokens in ext_gram_block for token in tokens]
ext_ungr_tokens = [token for tokens in ext_ungr_block for token in tokens]
ext_rec_tokens = [token for tokens in ext_rec_block for token in tokens]

ext_gram_vectors = ext_encoder.transform(ext_gram_tokens)
ext_ungr_vectors = ext_encoder.transform(ext_ungr_tokens)
ext_rec_vectors = ext_encoder.transform(ext_rec_tokens)

ext_X_gram = ext_gram_vectors[np.newaxis, :]
ext_y_gram = np.append(ext_gram_vectors[1:], ext_gram_vectors[:1], axis=0)[np.
 ↪newaxis, :]

ext_X_ungr = ext_ungr_vectors[np.newaxis, :]
ext_y_ungr = np.append(ext_ungr_vectors[1:], ext_ungr_vectors[:1], axis=0)[np.
 ↪newaxis, :]

ext_X_rec = ext_rec_vectors[np.newaxis, :]
ext_y_rec = np.append(ext_rec_vectors[1:], ext_rec_vectors[:1], axis=0)[np.
 ↪newaxis, :]

print(ext_X_gram.shape, ext_y_gram.shape, ext_X_ungr.shape, ext_y_ungr.shape,␣
 ↪ext_X_rec.shape, ext_y_rec.shape)

# Initialize the model as used for Misyak simulations
ext_model = Sequential([
    SimpleRNN(15, input_shape=(None, 30),
              return_sequences=True,
              activation='tanh',
              recurrent_initializer=RandomUniform(minval=-1, maxval=1),
              name='hidden'),
    TimeDistributed(Dense(30, activation='softmax', name='output'))
])

ext_model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=0.1, momentum=0.8),
              metrics=['categorical_accuracy', 'mean_squared_error'])

print(ext_model.summary())

# Training in 3 phases
scale_factor = 5

print('-'*100)
print('Phase 1: 6 grammatical blocks of 72 strings..\n')
ext_phase_1 = ext_model.fit(ext_X_gram, ext_y_gram, epochs=6*scale_factor,␣
 ↪verbose=2)
```

```
print('-'*100)

print('Phase 2: 1 ungrammatical block of 24 strings..\n')
ext_phase_2 = ext_model.fit(ext_X_ungr, ext_y_ungr, epochs=1*scale_factor,␣
↪verbose=2)
print('-'*100)

print('Phase 3: 1 recovery block of 72 strings..\n')
ext_phase_3 = ext_model.fit(ext_X_rec, ext_y_rec, epochs=1*scale_factor,␣
↪verbose=2)
print('-'*100)


# Generating predictions
ext_pred_block = random.sample(extended_grammatical_block(), 12)
print(ext_pred_block)

ext_pred_tokens = [token for tokens in ext_pred_block for token in tokens]
ext_pred_vectors = encoder.transform(ext_pred_tokens)
ext_X_pred = pred_vectors[np.newaxis, :]

# decode and inspect
ext_predictions = encoder.inverse_transform(np.array(model.
↪predict(ext_X_pred)[0]))

print('12-item prediction task input tokens:\n{}'.format(ext_pred_tokens))
print('\n12-item prediction task output predictions:\n{}'.
↪format(ext_predictions))

plt.figure(figsize=(10,8))
plot_activation_matrix(ext_model, ext_X_gram, layers=[ext_model.layers[1]],
                       subset=(0, slice(0, 20), slice(None)),
                       show_values=False)
```

```
(1, 288, 30) (1, 288, 30) (1, 96, 30) (1, 96, 30) (1, 288, 30) (1, 288, 30)
Model: "sequential_40"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
hidden (SimpleRNN)           (None, None, 15)          690

_________________________________________________________________
time_distributed_38 (TimeDis (None, None, 30)          480
=================================================================
Total params: 1,170
Trainable params: 1,170
Non-trainable params: 0

_________________________________________________________________
```

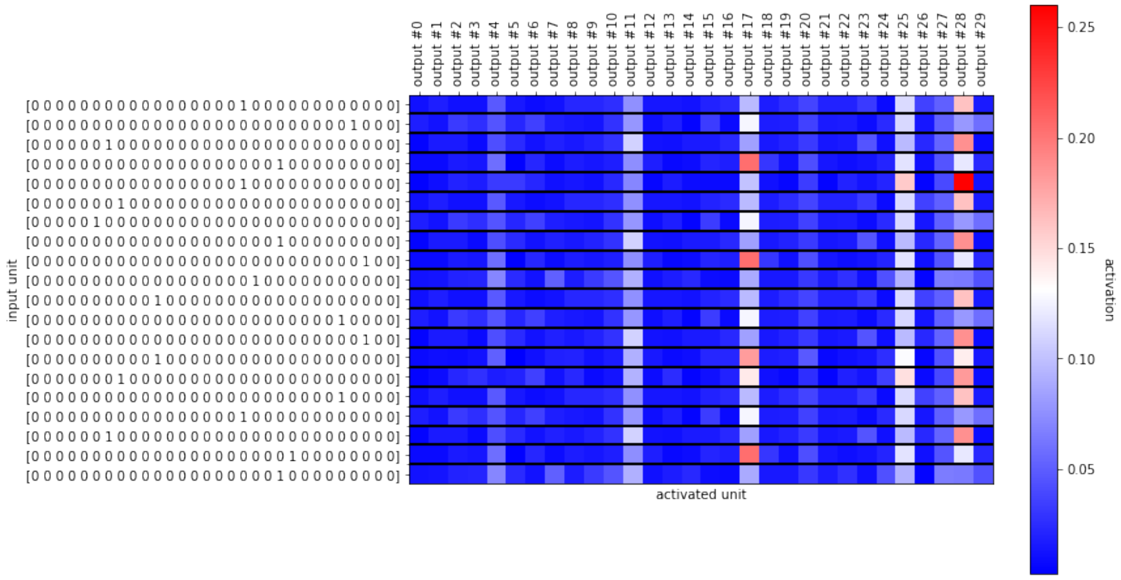
```
None
--------------------------------------------------------------------------------
--------------------
Phase 1: 6 grammatical blocks of 72 strings..

Train on 1 samples
Epoch 1/30
1/1 - 1s - loss: 3.6128 - categorical_accuracy: 0.0278 - mean_squared_error:
0.0329
Epoch 2/30
1/1 - 0s - loss: 3.6306 - categorical_accuracy: 0.0625 - mean_squared_error:
0.0328
Epoch 3/30
1/1 - 0s - loss: 3.6224 - categorical_accuracy: 0.0417 - mean_squared_error:
0.0328
Epoch 4/30
1/1 - 0s - loss: 3.6475 - categorical_accuracy: 0.0208 - mean_squared_error:
0.0329
Epoch 5/30
1/1 - 0s - loss: 3.6009 - categorical_accuracy: 0.0278 - mean_squared_error:
0.0328
Epoch 6/30
1/1 - 0s - loss: 3.6458 - categorical_accuracy: 0.0660 - mean_squared_error:
0.0331
Epoch 7/30
1/1 - 0s - loss: 3.6107 - categorical_accuracy: 0.0660 - mean_squared_error:
0.0330
Epoch 8/30
1/1 - 0s - loss: 3.5617 - categorical_accuracy: 0.0660 - mean_squared_error:
0.0328
Epoch 9/30
1/1 - 0s - loss: 3.5061 - categorical_accuracy: 0.0660 - mean_squared_error:
0.0326
Epoch 10/30
1/1 - 0s - loss: 3.4494 - categorical_accuracy: 0.0694 - mean_squared_error:
0.0324
Epoch 11/30
1/1 - 0s - loss: 3.3955 - categorical_accuracy: 0.0729 - mean_squared_error:
0.0322
Epoch 12/30
1/1 - 0s - loss: 3.3469 - categorical_accuracy: 0.1007 - mean_squared_error:
0.0321
Epoch 13/30
1/1 - 0s - loss: 3.3046 - categorical_accuracy: 0.1146 - mean_squared_error:
0.0319
Epoch 14/30
1/1 - 0s - loss: 3.2682 - categorical_accuracy: 0.1146 - mean_squared_error:
0.0318
```

```
Epoch 15/30
1/1 - 0s - loss: 3.2372 - categorical_accuracy: 0.1146 - mean_squared_error:
0.0318
Epoch 16/30
1/1 - 0s - loss: 3.2111 - categorical_accuracy: 0.1146 - mean_squared_error:
0.0317
Epoch 17/30
1/1 - 0s - loss: 3.1890 - categorical_accuracy: 0.1146 - mean_squared_error:
0.0316
Epoch 18/30
1/1 - 0s - loss: 3.1702 - categorical_accuracy: 0.1111 - mean_squared_error:
0.0316
Epoch 19/30
1/1 - 0s - loss: 3.1541 - categorical_accuracy: 0.1111 - mean_squared_error:
0.0316
Epoch 20/30
1/1 - 0s - loss: 3.1401 - categorical_accuracy: 0.1146 - mean_squared_error:
0.0315
Epoch 21/30
1/1 - 0s - loss: 3.1279 - categorical_accuracy: 0.1146 - mean_squared_error:
0.0315
Epoch 22/30
1/1 - 0s - loss: 3.1170 - categorical_accuracy: 0.1146 - mean_squared_error:
0.0315
Epoch 23/30
1/1 - 0s - loss: 3.1071 - categorical_accuracy: 0.1146 - mean_squared_error:
0.0315
Epoch 24/30
1/1 - 0s - loss: 3.0982 - categorical_accuracy: 0.0972 - mean_squared_error:
0.0314
Epoch 25/30
1/1 - 0s - loss: 3.0901 - categorical_accuracy: 0.0972 - mean_squared_error:
0.0314
Epoch 26/30
1/1 - 0s - loss: 3.0827 - categorical_accuracy: 0.1181 - mean_squared_error:
0.0314
Epoch 27/30
1/1 - 0s - loss: 3.0759 - categorical_accuracy: 0.1181 - mean_squared_error:
0.0314
Epoch 28/30
1/1 - 0s - loss: 3.0697 - categorical_accuracy: 0.1181 - mean_squared_error:
0.0314
Epoch 29/30
1/1 - 0s - loss: 3.0639 - categorical_accuracy: 0.1250 - mean_squared_error:
0.0314
Epoch 30/30
1/1 - 0s - loss: 3.0586 - categorical_accuracy: 0.1250 - mean_squared_error:
0.0313
```

```
--------------------------------------------------------------------------------
-------------------
Phase 2: 1 ungrammatical block of 24 strings..

Train on 1 samples
Epoch 1/5
1/1 - 1s - loss: 3.2612 - categorical_accuracy: 0.0625 - mean_squared_error:
0.0319
Epoch 2/5
1/1 - 0s - loss: 3.2010 - categorical_accuracy: 0.0625 - mean_squared_error:
0.0317
Epoch 3/5
1/1 - 0s - loss: 3.0996 - categorical_accuracy: 0.1562 - mean_squared_error:
0.0314
Epoch 4/5
1/1 - 0s - loss: 2.9729 - categorical_accuracy: 0.1979 - mean_squared_error:
0.0310
Epoch 5/5
1/1 - 0s - loss: 2.8355 - categorical_accuracy: 0.1979 - mean_squared_error:
0.0304
--------------------------------------------------------------------------------
-------------------
Phase 3: 1 recovery block of 72 strings..

Train on 1 samples
Epoch 1/5
1/1 - 0s - loss: 3.3851 - categorical_accuracy: 0.0556 - mean_squared_error:
0.0325
Epoch 2/5
1/1 - 0s - loss: 3.4091 - categorical_accuracy: 0.0521 - mean_squared_error:
0.0327
Epoch 3/5
1/1 - 0s - loss: 3.4163 - categorical_accuracy: 0.0521 - mean_squared_error:
0.0328
Epoch 4/5
1/1 - 0s - loss: 3.4070 - categorical_accuracy: 0.0556 - mean_squared_error:
0.0327
Epoch 5/5
1/1 - 0s - loss: 3.3844 - categorical_accuracy: 0.0556 - mean_squared_error:
0.0326
--------------------------------------------------------------------------------
-------------------
[('pel', 'coomo', 'vamey', 'rid'), ('dak', 'balip', 'taspu', 'jic'), ('pel',
'nilbo', 'benez', 'rid'), ('pel', 'benez', 'hiftam', 'rid'), ('pel', 'benez',
'benez', 'rid'), ('vot', 'laelijeen', 'coomo', 'tood'), ('pel', 'nilbo',
'plizet', 'rid'), ('vot', 'gensim', 'benez', 'tood'), ('vot', 'balip', 'wadim',
'tood'), ('pel', 'benez', 'chila', 'rid'), ('dak', 'kicey', 'vamey', 'jic'),
('dak', 'loga', 'skiger', 'jic')]
```

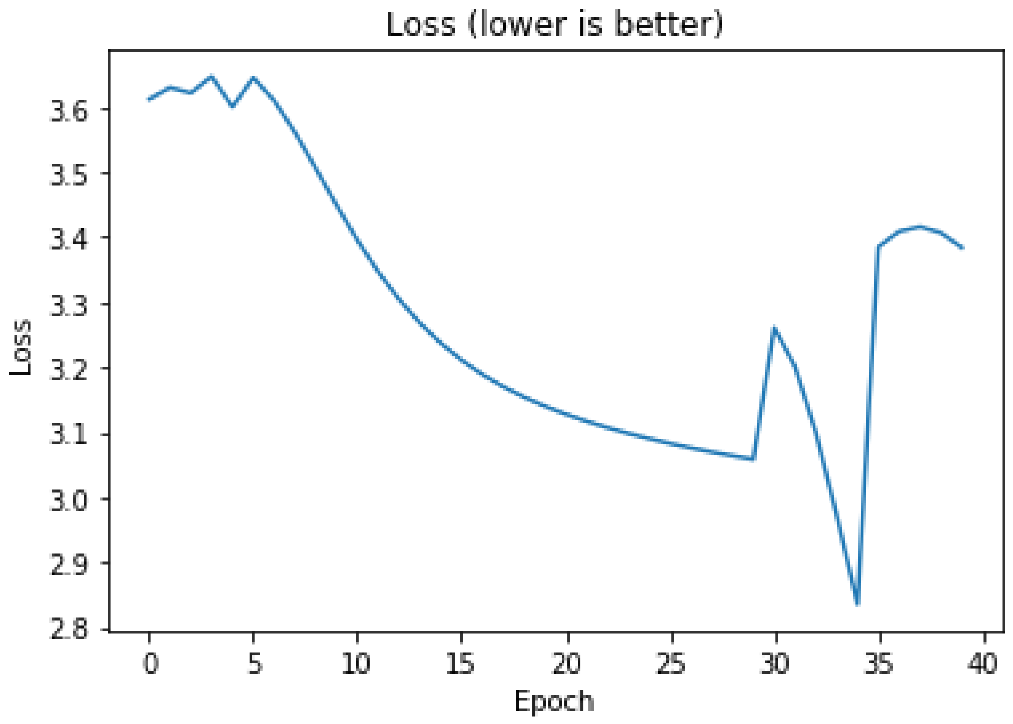
```
12-item prediction task input tokens:
['pel', 'coomo', 'vamey', 'rid', 'dak', 'balip', 'taspu', 'jic', 'pel', 'nilbo',
'benez', 'rid', 'pel', 'benez', 'hiftam', 'rid', 'pel', 'benez', 'benez', 'rid',
'vot', 'laelijeen', 'coomo', 'tood', 'pel', 'nilbo', 'plizet', 'rid', 'vot',
'gensim', 'benez', 'tood', 'vot', 'balip', 'wadim', 'tood', 'pel', 'benez',
'chila', 'rid', 'dak', 'kicey', 'vamey', 'jic', 'dak', 'loga', 'skiger', 'jic']

12-item prediction task output predictions:
['tood' 'pel' 'jic' 'pel' 'jic' 'pel' 'jic' 'jic' 'pel' 'tood' 'tood'
 'pel' 'tood' 'jic' 'pel' 'tood' 'jic' 'pel' 'tood' 'tood' 'pel' 'tood'
 'jic' 'pel' 'jic' 'jic' 'jic' 'pel' 'jic' 'pel' 'jic' 'jic' 'jic' 'jic'
 'jic' 'jic']
```

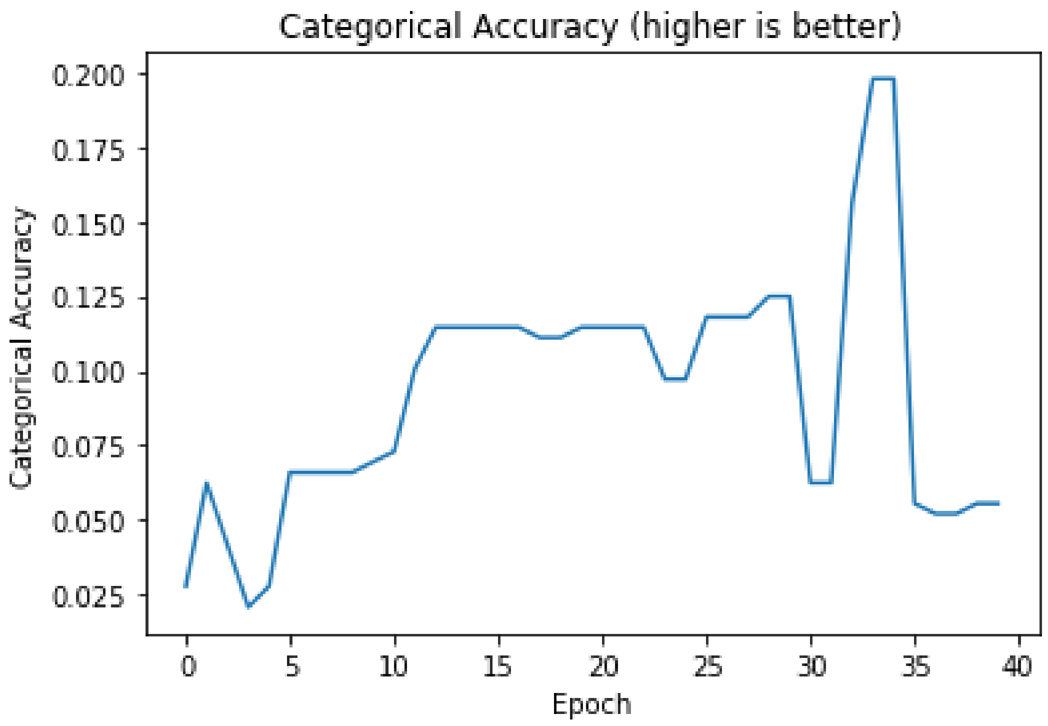
```
C:\Users\magdo\Anaconda3\lib\site-packages\matplotlib\text.py:1150:
FutureWarning: elementwise comparison failed; returning scalar instead, but in
the future will perform elementwise comparison
  if s != self._text:
```

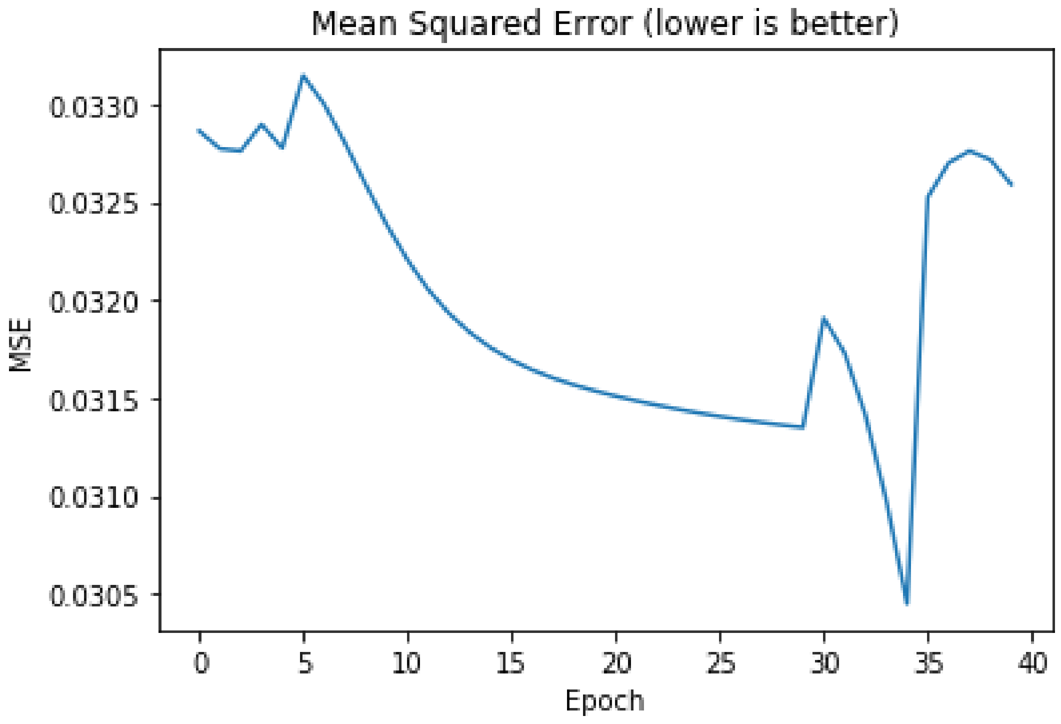


[251]:
```python
ext_loss = ext_phase_1.history['loss'] + ext_phase_2.history['loss'] +␣
 ↪ext_phase_3.history['loss']
ext_accu = ext_phase_1.history['categorical_accuracy'] + ext_phase_2.
 ↪history['categorical_accuracy'] \
          + ext_phase_3.history['categorical_accuracy']
ext_mse = ext_phase_1.history['mean_squared_error'] + ext_phase_2.
 ↪history['mean_squared_error'] \
          + ext_phase_3.history['mean_squared_error']
```

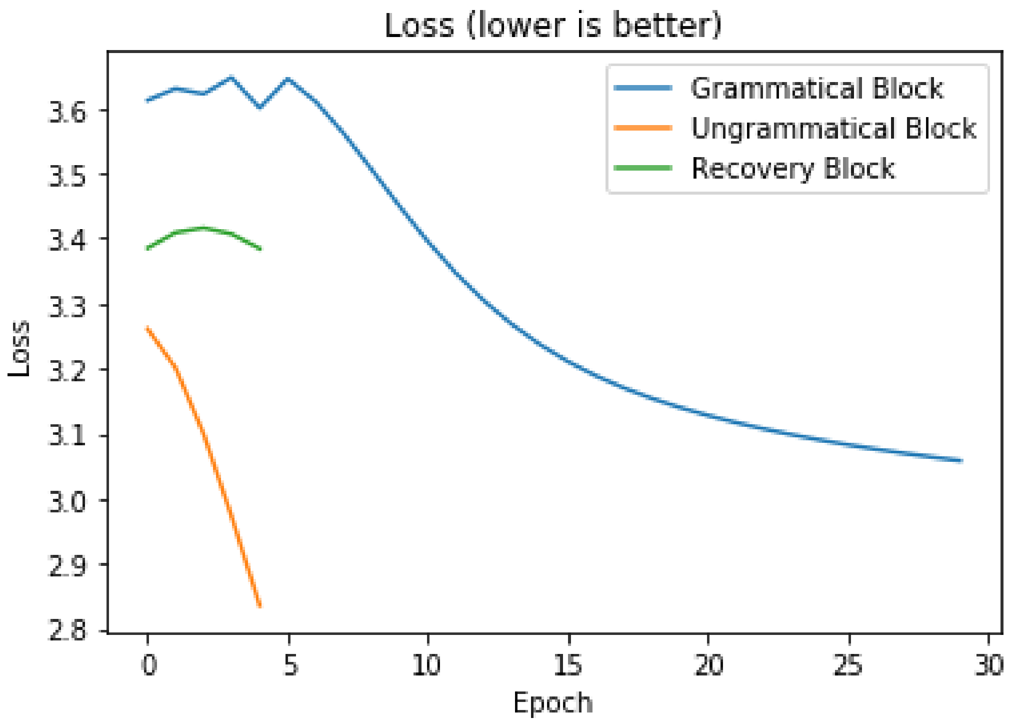
```
plt.plot(ext_loss)
plt.title('Loss (lower is better)')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.show()

plt.plot(ext_accu)
plt.title('Categorical Accuracy (higher is better)')
plt.ylabel('Categorical Accuracy')
plt.xlabel('Epoch')
plt.show()

plt.plot(ext_mse)
plt.title('Mean Squared Error (lower is better)')
plt.ylabel('MSE')
plt.xlabel('Epoch')
plt.show()
```




Categorical Accuracy (higher is better)
Categorical Accuracy
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0
5
10
15
20
25
30
35
40
Epoch


Mean Squared Error (lower is better)
MSE
0.0330
0.0325
0.0320
0.0315
0.0310
0.0305
0
5
10
15
20
25
30
35
40
Epoch

[252]:
```
plt.plot(ext_phase_1.history['loss'], label="Grammatical Block")
plt.plot(ext_phase_2.history['loss'], label="Ungrammatical Block")
plt.plot(ext_phase_3.history['loss'], label="Recovery Block")
plt.legend()
plt.title('Loss (lower is better)')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.show()
```



3.6 Exercise 6

A) How does learning proceed now? Describe learning, how the error metrics develop (especially after blocks 6,7 and 8).

As can be seen from the plots above, the learning is relatively smoother for the extended model, although the values for loss are rleatively higher. The model seems to be more robust, starting the recovery block from about the same loss region as the original grammatical block and displaying similar trends. The model seems to display lower loss for the ungrammatical block, which could be due to the model having learned grammatical representations fairly well in the grammatical stage. Due to that, it would learn the sequences in the ungrammatical block fairly faster. The loss for the ungrammatical block displays a similar sharp drop as seen in the original Misyak simulations. The learning of grammatical sequences is rocky at the beginning and becomes smooth later (~ epoch 5). This trend might be relating to the NN ‘trying out’ different kinds of information before extracting the grammatical rules.

The loss and the trends could be due to the longer input sequences the model is presented with. Longer sequences require the model to be trained on more data in order to identify the underlying patterns. This could also be achieved by creating deeper recurrent models.

B) Compare the prediction capacity of the model in this extended simulation to the one of the previous section, commenting on similarities and differences.

The model working on the original sequences learns the underlying text sequences much better than the same model applied to extended sequences. Both models display similar trends in the decline of the loss metric over the different blocks of training, while the extended model displays more rocky declines in the loss curves, both models express similar loss values.

We notice in the predictions that the extended model predicts the string “jic” more times given different input sequence. The extended model seems to have defaulted to returning “jic” as a continuation for any input sequence. The original model has learned to map the possible sequences better, with outputs averaging a class-wise accuracy of 50% where as the extended model averages a class-wise accuracy of 25%.

C) SRNs with only one context layer can learn more than just one step back in time. But is your SRN successful in this task? Explain your answer.

The higher loss depicted as well as the lower categorical accuracy infer that the model is not as successful in representing longer sequences as compared to the three word sequences in the original Misyak experiment. Longer sequences with smaller datasets (or datasets of the same size as the shorter sequences) mean that the model does not see examples for every possible mapping between the sequences. This causes a skew in the inputs the model receives, in turn affecting the model’s learning of such combinations.

Thus, deeper architectures (increasing the number of context layers) could assist in creating better representations of the sequences of text that the model learns. The model could also perform better given more data with extended sequences.

D) What do you predict human subjects will do, based on the results of your simulation?

Human subjects will find it difficult to predict the ending tokens of the sentences when they are longer, unless they are exposed to significantly longer training. This canbe attributed to the increase in possible choices for the middle sequences. Initially, with the three sequence words, the complexity of choosing the middle sequence was N where N is the number of options for the middle sequence. With the elongated sequence, this complexity is now increased to N^2, which makes it difficult for humans to grasp without substantial training.