Thai Language Tokenization with Multilayer Perceptron

Michael Partridge
Brandeis University
Statistical Approaches to NLP
Ben Welner
mpartrid@brandeis.edu

Abstract

Language tokenization, here referring primarily to separation of character sequences into words is a nontrivial task for many languages including Thai which do not space separate words. Here we present an approach using a Multilayer Perceptron and validate the approach by comparing it to previous approaches to the same task.

1 Introduction

Previous approaches to Thai language tokenization have considered characters separately or as a sequence using various linear models. There are two of note that will be used as baselines for comparison here. One is a Maximum Entropy classifier which considered each character separately, though with features that gave context information. The other is a Linear Chain Conditional Random Field which used the same features as the Maximum Entropy approach, but considered each sequence of characters together and classified them via viterbi decoding. These sequences came from the trivial separation of Thai language into sentences using punctuation.

This paper describes a non-linear approach to the same task and will compare its performance against the best results from each of the two previous approaches. We will use a Multilayer Perceptron from deeplearning[1], though with numerous experimental modifications as described herein.

Following establishing that a non-linear model can outperform the linear classifiers, we will apply techniques for converting our feature space into fixed-width vectors with tools from word2vec[2]. If successful, this approach has two main benefits. First, it allows our approach to be generalized to other languages without requiring language-specific knowledge for featurization. Second, it allows us to capture more context information than allowed by our current feature model due to its compactness in expressing information in dense representations. Our work here will show that this approach is conceptually sound, however improvement is needed before it can replace hand-tuned features.

Since this project is an exploration of neural networks as much as it is an attempt to solve a novel problem, we have conducted a multitude of experiments with different configurations both in an attempt to maximize model performance and to replicate expected network behavior. While neural networks are able to approximate more functions than linear models can, it is suggested that they are prone to over fitting. We have designed experiments without model regularization in an attempt to detect model over fitting.

2 Method Description

This Multilayer Perceptron comes initially from the aforementioned resource at deeplearning.net. It was initially taken unaltered except to alter the `load_data` method to allow this model to work with the Thai tokenization task instead of the image recognition task for which it was designed. After the model was successfully running in its original form, experimental changes were made, first to the hyper parameters and later more fundamental changes to

[1]http://deeplearning.net/tutorial/mlp.html

[2]https://code.google.com/p/word2vec/

the model structure.

2.1 The Model

Taken directly from deeplearning, the model is a standard feed-forward neural net or Multilayer Perceptron (MLP) which consists of an input layer, a hidden layer, and an output layer. The input and output layers are of fixed width as determined by the input. Since dense representations of features are used, the width of the input layer is the dimensionality of the feature space, which will be discussed further in the *Features* section. The output layer for this task has dimensionality of two–one for each label. For simplicity, the softmax function applied over the two output nodes is combined into a single node here. With that simplification, it gives our model diagram shown in Figure 1 the appearance of a Maximum Entropy classifier added to a fully-connected hidden layer as input. This is exactly the architecture of the deeplearning MLP.

The model is written in Python with a library called Theano, designed for machine learning applications. Theano programs are graphical models which are functionally represented in Python for ease of programming them, but compiled into C code at runtime for better performance. Theano employs symbolic algebra and shared variables to achieve very high parallelism and supports GPU-based calculations with CUDA for NVIDIA-made GPUs. For this project, the GPU functionality was not used, but CPU parallelism was leveraged (automatically by the system) which necessitated one significant change over the previous models: in order to instantiate all nodes simultaneously in memory and train them in parallel, Theano requires dense feature vector representation. This causes a fully-featurized dataset to occupy far more memory than the sparse representations used in the Maximum Entropy and Conditional Random Field approaches of the past. Initially this meant that only the simpler *Character* features could be used (described below), however for later experiments a large server was used which had the necessary resources to support any feature sets that were available.

2.2 Hyper-Parameters

A complete discussion of available hyper-parameters would be tedious and uninformative to

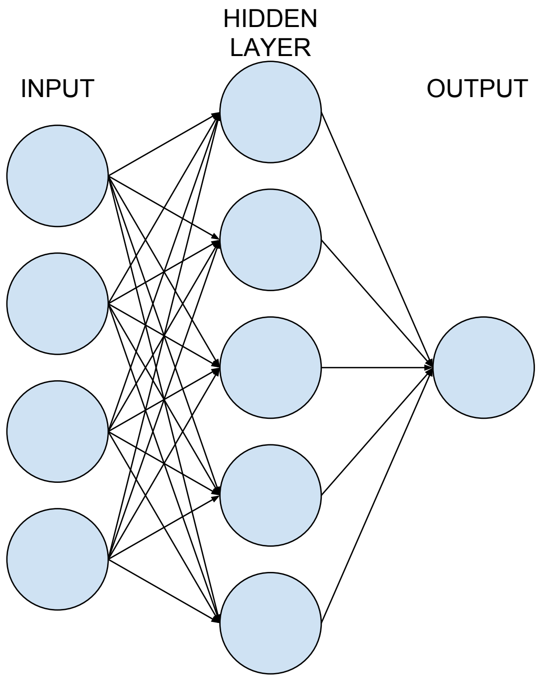

Figure 1: Architecture of standard MLP for this task

readers assumed to have some machine learning background. Instead, we will discuss a few of note, including ones which are most relevant to the experiments performed.

Hidden Layer Size: Most straightforward is the number of hidden nodes, that is the width of the hidden layer. For experiments which employ multiple hidden layers, all layers have the same number of hidden nodes. These nodes are fully connected in all cases and all use the same activation function.

Activation Function: The standard activation function used is the hyperbolic tangent function, *tanh*. All experiments unless otherwise noted use this function for all hidden nodes. The initialization of the parameters, as suggested by *deeplearning*, samples uniformly from a range of this function:

$$\left[-\sqrt{\frac{6}{fan_{in}+fan_{out}}},\sqrt{\frac{6}{fan_{in}+fan_{out}}}\right]$$

Where fan_{in} is the number of units in the input layer (or preceding layer for multiple hidden layers) and similarly for fan_{out}. Initialization of sigmoid activation parameters comes from a range 4 times this interval. In the absence of guidance, values for other activation functions are initialized the same as for *tanh*.

Beyond *tanh* and *sigmoid*, which are built-in functions for Theano, we implemented a rectified linear (*relu*) function and a simple linear function for comparison. Since Theano performs symbolic differentiation to compute the gradient for backpropagation, only the activation function itself needed to be implemented.

Regularizers: Both L1 and L2 regularization (sum of absolute values, sum of squares) are implemented in the package. Literature suggests that L1 regularization is better suited to sparse data and L2 for most other cases. As this data is not sparse (only a small number of unique characters), L2 is preferred. The type and amount of regularization is set using `L1_reg` and `L2_reg` hyper-parameters.

Patience: This model uses stochastic gradient descent (SGD) to optimize the model parameters. Similar to other implementations, it uses a fixed learning rate and a fixed size minibatch. However, this model uses two conditions to determine when the gradient descent has converged: patience and epochs. A limit is placed on the maximum number of iterations over the full training data (broken into however many minibatches as required), called epochs. If this limit is reached, 1000 epochs by default, the training stops regardless of the current likelihood of the verification set (sometimes also refered to as the dev set).

The alternate condition of convergence regards the *patience* of the training process. Initialized to a fixed number of minibatches, patience determines the maximum number of minibatches considered during training. However, if at any point the accuracy against the verification set improves more than a certain threshold over the previous best accuracy (by default, 0.995), patience grows to the maximum of its current value and some factor of the current iteration number, by default twice the number of minibatches considered so far. This allows the training to proceed at least as long as it has so far whenever parameters produce a significantly better likelihood than seen so far. In practice, almost every training run ended with patience running out rather than the maximum number of epochs.

2.3 Alternate Configurations

In addition to the explicit hyper-parameters of the model, we experimented with different numbers of hidden layers in the model. This was done through small modifications to the MLP class by composing more hidden layers together. The 1-layer, 2-layer, and 4-layer models are included in the submitted code. Each uses the same number of hidden nodes as well as the same activation function. Based on outside reading, *relu* activation function was prefered for deeper networks to avoid the vanishing gradient problem[3].

2.4 Corpus

The corpus is the same from the last programming assignment–orchid97. It consists of processed Thai text and consists of 1,500,647 characters in 94,685 sequences (sentences). Each character has been tagged with its type from one of 7 character classes (tone marker, consonant, punctuation, end of sentence, etc.) and has been tagged with B, I, or O to indicate its position at the beginning of a word, inside a word, or outside a word (sentence boundary). Our goal is to learn the B tags, which is equivalent to finding where word boundaries are.

This approach will not consider sequence information beyond local context to a character (described in the next section) and so each character will be trained on and labeled separately. This was accomplished by a simple nested list comprehension to flatten the list of sequences into a list of characters after the sequences have been featurized. Based on previous experience with this approach using the Maximum Entropy classifier, we expect to be able to achieve excellent performance without needing to fully model each sequence. As it will be relevant when evaluating performance on this data set, the

[3]The vanishing gradient problem is common in deep neural networks and simply stated it refers of the tendency of the composition of the activation functions in every layer to drive the overall gradient towards 0 or causing it to blow up. Only a narrow range of weights result in the overall gradient value in a useful range. Gradients outside this useful range prevent useful training from taking place. Proper initialization of weights as well as using rectified linear activation (avoid compressing values towards 1 or -1) have been shown to improve this as discussed here: `neuralnetworksanddeeplearning.com/chap5.html`

frequency of the most common tag, the *I* tag for inside a word is 79%.

2.5 Features

We began initially with the same feature sets used by the Maximum Entropy and Conditional Random Field classifiers when put to this task. We redefine those here for completeness:

Character: The *Character* feature set consists of the type (consonant, vowel, tone marker, etc.) of the current character as well as characters immediately before and after the current. In addition, if the current character is the first of a sequence (as trivially separated by punctuation) or the last of a sequence, it gets an additional feature to indicate that instead of a neighbor character type. On this corpus, *Character* results in a feature set of 20 unique features.

Character2: The *Character2* feature set contains all of the information of *Character*, plus it also includes the actual characters at each position in the window. Additionally, the window is larger–2 characters in front and 2 characters behind the target character. This considerably increases the size of the feature space, up to 757 unique features. As was previously mentioned, one of the restrictions of working with Theano and its parallel computation model was the requirement to work with dense feature vectors. The size of the full training data, fully featurized with *Character2* features consumed almost 4 GB of memory and required pursuing additional computational resources to process. However, as the experiments section will show, this was worthwhile given the high accuracy achieved.

2.6 Word2Vec

As an alternative to hand-tuned linguistic features, which simple as they are here require at least labeling character types with domain knowledge, we also attempt character embedding using Google's Word2Vec open source package. Both Context Bag of Words (CBOW) and skip-gram are considered using a corpus wholly separate from the labeled sequence corpus of the main task. This corpus is the standard Thai language Wikipedia dump with minimal pre-processing. Though this does include some non-Thai characters, the vast majority is natural Thai language. It totals over 8.3 million characters with context.

Parameter	**Value**
Learning Rate	0.01
L1_Reg	0
L2_Reg	0.0001
Activation	`tanh`
Patience	10,000
Patience Incr.	`*=2`
Patience Thresh.	0.995

Parameter	**Value**
Batch Size	200
Hidden Layer	100
Features	Character
Max Epochs	1,000
Training Set	50,000
Validation Set	5,000
Test Set	15,000

Table 1: Standard experiment configuration parameters

Once the character vectors are constructed, yielding about 200 unique characters and embeddings, they are supplied to the classifier in lieu of the already-described features. The width of the character vector varies with the experiment and will be described in detail there. Other aspects of the training and evaluation process using character vectors remain the same.

3 Experiments and Discussion of Results

In total, just under 70 training and classification test runs were conducted using all manner of model configurations, feature configurations, and hyperparameters described in the previous section. The overall goals of the experiments were to confirm successful learning of the model, to compare the model to previous ones applied to this task, to explore tuning of MLP networks, and to examine expected results with MLP networks with regard to overfitting problems.

The standard setup for experiments was as is shown in Table 1. All experiments used these parameters and configuration except where explicitly stated.

3.1 Comparison to Previous Attempts

We'll roll validating the model into this section since beating previous approaches also validates this one. We compared our model to the other approaches using as close to the default setup as was reasonable while allowing some minor tuning to better suit this dataset. Since the *Character* and *Character2* feature sets each had interesting differences in how well the sequence and non-sequence models took to them, we considered our model against the others in both categories.

Figures 2 and 3 show the best performances of our model and previous attempts with each of the feature

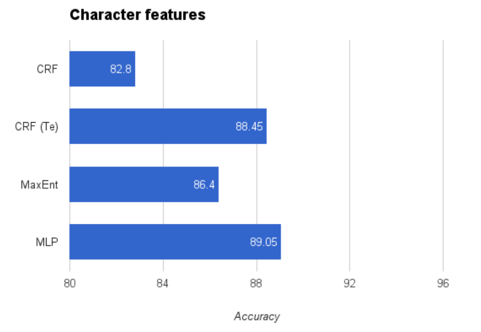

Figure 2: Accuracy with Character Features

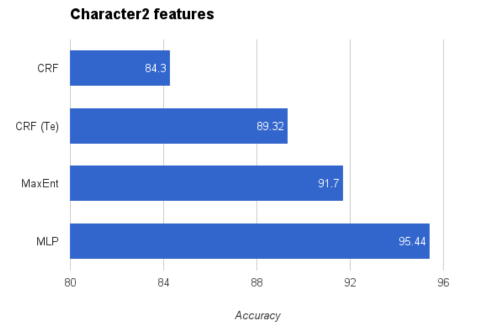

Figure 3: Accuracy with Character2 Features

sets. I was unable to achieve impressive results with the CRF and so I have included Te Rutherford's results as well for that task. As can be seen, the MLP beats all others on both feature sets. It is especially impressive that it beats the best linear classifier by almost 4 percentage points on the more advanced features, though the running times need to be considered. The MaxEnt classifier took under 1 minute to attain its almost 92% on those features. The accuracy shown for MLP required 86 minutes of training and test time. However, a similar run with fewer hidden nodes was able to achieve 94% percent accuracy while keeping total time under 1 minute as well. Thus, while the MLP generally does take longer than the other non-sequence classifier it can compete in time without suffering greatly.

This result is exactly what was hoped and achieving it required minimal tuning of the model. The only deviation from the standard experiment set up for these results was to set `L2_reg` to 0 and set the hidden layer size to 20 for the best performing run and 5 for the fast and well-performing run. Both of these experiment configurations will be made available in the submitted code for result verification.

3.2 Tuning the Model

For the sake of running time, most experimentation in tuning the model was performed on the less performant *Character* features. The results should then be compared to the 89% accuracy best result of the MLP. In experimenting with learning rates both larger and smaller than the standard 0.01, most outcomes were similar to the standard. This required also increasing the *Patience* parameter to allow more training iterations before converging and in the case of learning rates as low as 0.0001, required increasing the maximum number of epochs to 2000 since these experiments generally improved too slowly for the patience increase algorithm to take effect. Since these also performed similarly to the 0.01 learning rate experiments, we did not lower the Patience Threshold parameter form 0.995 to continue in that area.

In past models, performance has been highly sensitive to learning rate due to either divergence for higher rates or failure to improve for very small rates. The MLP appears very robust to differing learning rates as compared to linear models. We speculate that this results from the non-linearity of the activation functions tending to keep adjustments from overshooting local optima too far as well as fairly large batches of 200 records for each adjustment.

Experiments with varying batch size between 20 records and 5000 records produced slightly worse results, but generally did not make a significant difference in accuracy. Likewise choosing a sigmoid or rectified linear activation function produced results very similar to hyperbolic tangent.

We also composed several hidden layers together and ran experiments with 2 hidden layers and 4 hidden layers and compared them to the standard setup with 1 hidden layer. Interestingly, 2 layers performed slightly better than the standard setup, though at the cost of approximately double the running time. Choosing a rectified linear activation function instead of tanh improved running time and accuracy but only slightly. Best standard setup came in at 88.6% accuracy while 2 hidden layers gave 88.7% and with relu 88.82%. Based on the speed increase of relu, the tests with 4 hidden layers were all run with relu and further improved accuracy but only slightly to 88.92%. Since none of these results were better significantly better than the untuned model and not better at all than the best tuned model on the features, we did not proceed any further with modifying network architecture.

3.3 Overfitting

One sentence in the deeplearning tutorial caught our eye: "Unless we employ some regularization scheme (early stopping of L1/L2 penalties), a typical number of hidden units vs. generalization performance graph will be U-shaped." We sought to replicate this result through experimenting with varying hidden layer size with L1 and L2 regularization set to 0. We then compared the accuracy against the test set to the accuracy against the training set in an attempt to see the U-shaped curve described. Figure 4 shows the results with the *Character* feature set. Beyond disabling regularization, the experiment setup was standard.

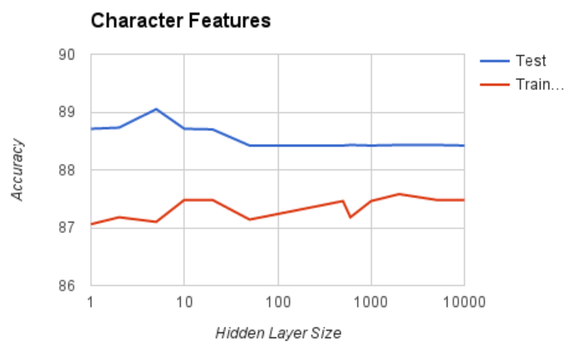

Figure 4: Overfitting with Character Features

As can be seen in the graph, whose x axis is log scale, there was not a significant change in overfitting (test accuracy compared to training accuracy), though there does appear to be a small upward trend in accuracy of the Training set, as predicted. Even more interesting was that our performance did not seem to suffer as low as even 1 node in the hidden layer. That structure is essentially a standard maximum entropy classifier. Good performance on this data would imply that the data was linearly separable and that a sophisticated linear classifier was unnecessary for this task, despite the good results shown so far. To examine that result more closely, we created a linear activation function and with an otherwise standard setup. The results were consistent with what a size 1 hidden layer suggested–very similar performance to non-linear setups.

However, the simplicity of the *Character* features might mask our results due to it driving accuracy changes into such a small range as to be possibly insignificant. We re-ran as much of the experiment as was possible with *Character2* features, though the running times became extreme with even a hidden layer size of 20 nodes taking nearly an hour and a half to converge. We present the results of this experiment to the largest size hidden layer that could be completed in Figure 5.

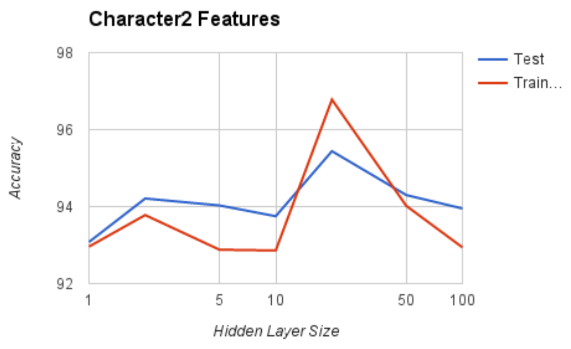

Figure 5: Overfitting with Character Features

Once again, the graph is not exactly what was predicted. However, of particular interest is the test with a hidden layer of 20, where both test and training accuracy peak. This test ran significantly longer than those with smaller hidden layer sizes and than the two with larger sizes. We believe that this gives a clue to our inability to reproduce the results predicted by deeplearning. Since we allowed the Patience convergence check to continue, our experiments with overfitting ran for very different amounts of time (and numbers of epochs). Thus we did not fully remove regularization in its most general form from the model. The model with 20 hidden nodes reached the full 1000 epoch limit, while the other tests on *Character2* all stayed below 25 epochs. Our result then, is more properly interpreted as pointing to a correlation between overfitting and amount of training done in the absence of regularization since the greatest overfitting, at 20 hidden nodes, was also the most time spent training. Unfortunately we do not have enough data from experiments already run nor time to run further ones to confirm this finding. Future work should study this possible correlation and its applications to model regularization, convergence, and generalization.

3.4 Word2Vec

In an attempt to generalize the process of featurization for this task across languages, we implemented character embedding with Google's Word2Vec package. We used the Thai language wikipedia dump as a source of data and space separated all characters with `cat full.txt | sed 's/./& /g' > full-separated.txt`. That sequence of 8.3 million characters was then fed directly into Word2Vec with various configurations and the character vectors generated were then fed into our model as features.

Word2Vec recommends CBOW for data which is not sparse, so our experiments generally used that, though one experiment was run with skip-gram for comparison. Our classifier configuration was standard for all of these experiments. Overall results were not impressive, but did demonstrate the soundness of the technique. Our best accuracy with this method was 84.5%, which compared with the 79% baseline on this data set demonstrates learning, though not to the degree of even the simple *Character* features.

Default parameters for Word2Vec were used except for the following. We experimented with vectors of 100, 300, and 600 dimensions as well as varied the context window for embedding from 5 to 15 characters. For reference, 5 is equivalent to the context information contained in *Character2* features, though those features also include linguistically significant character types. We increased the number of training iterations of Word2Vec to 15 and also performed one test with a large window, 600 dimensions and skip-gram. Vectors of 600 dimensions generally outperformed smaller vectors, though at great cost in running time. Overall, we believe there is promise to this approach, but it is not yet a drop-in replacement for domain knowledge and hand-tuned features.

4 Conclusion

The goals of this project were twofold: learn about neural networks through experimentation and attempt to outperform linear classifiers on the task of tokenizing thai text. We certainly succeed in the latter, producing meaningful improvement in accuracy even when restricted to similar running times. Moreover, we were able to extract better gains from more sophisticated features, suggesting that neural nets make better use from domain knowledge on this task.

In learning about neural nets, we also learned about implementing machine learning tools from packages, an invaluable skill, and were especially impressed with Theano as a set of tools for building them. While the parallel nature of Theano limited what experiments could be run on simple hardware, its possibilities for speedup by running experiments on the GPU would likely make up for that given access to better hardware. Furthermore, it opens the possibilities for developing complex networks using relatively simple symbolic algebra instead of attempts to optimize gradient computation by hand in Python.

As for the nets themselves, it was encouraging to see that high performance essentially comes for free when dropping in one as a replacement for a linear classifier, even when the data as here is questionably non-linear. In exploring the behavior of neural networks in the overfitting experiments, the correlation between training time and generalization was very interesting and suggests a future project in this area. It was disappointing that this discovery came so close to the submission deadline for the project.

I have submitted all related code for this project. The Orchid97 data is the same as what was provided with the last programming assignment and large so it is not included with the project submission. At the bottom of `mlp_thai_tokenization.py`, I have included functions to run some of the best performing tests to see those results. All other tests were run by modifying that function call or occasionally some other parameters clearly labeled in that file. I did not modify the code for Word2Vec and so have not included those files, but they are freely available from the link provided in the introduction. The only preprocessing run on the wikipedia dump was the single `sed` command already included here. I have also included the function call for running a test with word vectors in comments with the others.