Sentiment Analysis of Tweets: Transfer Learning in LSTM Networks

Farid Musayev | LiU-ID: farmu437
Linköping University
Department of Computer and Information Science
The Division of Statistics and Machine Learning
732A81 Text Mining Course
farmu437@student.liu.se

Abstract

This study focuses on the Sentiment Analysis of Tweets shared as a part of for Kaggle Competition in 2020. During the study, a machine learning model, namely SVM classifier, and a deep neural model, namely LSTM network, with three different initiations of word embeddings are compared. Thus, another focus of the study is to analyze transfer learning impact on a given sentiment analysis task. It is revealed that all the trained models detect the sentiment of a given tweet with the accuracy ranging from 0.72 and 0.75. In addition, it is also confirmed that transfer learning and further model training can indeed improve performance of a deep neural model. However, we can also see that a retrained deep neural model does not outperform machine learning model with a significant margin. Evaluation of models on custom-generated tweets make us believe that design choices are equally important and transfer learning do not guarantee a performance boost on its own and there can be other limiting factors in the model.

1 Introduction

The freedom and ease of self-expression that is aided with the modern era of social networks also opens a space for a spread of discrimination, insults and hatred. It becomes vital for social networks to moderate the shared content of the users to let the community thrive in love and peace rather than drown in darkness and evil.

This study focuses on the Sentiment Analysis of Tweets shared as a part of for Kaggle Competition[1] in 2020. Although the initial task for the competition was extraction of words that reflected the sentiment, this study is relatively simplified by trying to predict whether a given tweet has a *negative*, *neutral* or *positive* sentiment. During this study a machine learning (ML) and a deep learning (DL) model with different configurations are compared: Support Vector Machine (SVM) classifier, type of recurrent neural network (RNN) LSTM network with trained word embeddings, LSTM with pretrained word embeddings and LSTM with retrained word embeddings. Word embeddings utilized for neural models are vector representations of words that were learned from two billion tweets in an unsupervised way as a part of GloVe[2] project. SVM classifier initially serves as a baseline model, however, it demonstrates a competitive performance. Although three neural models are mentioned, it is worth stating that, technically, the same neural model with three different initiations of word embeddings is implemented. This allows to demonstrate whether the model performance improves and elaborate on the transfer learning in the scope of this classification task.

2 Theory

In this section, word representations for each of the models, SVM and LSTM, are discussed. Then, SVM classifier is briefly discussed as a baseline machine learning model. Finally, the architecture of implemented LSTM network and its suitability for this classification task are discussed.

2.1 Bag-of-words

Bag-of-words is a numerical representation of text (in our case tweet) as counts of raw words where order of these words is disregarded. Each tweet is transformred into a vector with dimensionality equal to the vocabulary size. Vocabulary size is the total number of unique words in all tweets of the training corpus. In this study, *tf-idf* vectorization of each tweet is implemented to design a feature space for SVM classifier. Tf-idf vectorization allows to account not only for a number of times a word appears in the tweet but also for the frequency at which it appears across all the tweets in

[1] https://www.kaggle.com/competitions/tweet-sentiment-extraction/overview

[2] https://nlp.stanford.edu/projects/glove/

Tf-idf vectorization	
n-grams	(1, 2)
feature-dim	(1, 126954)

Table 1: Specifications of tf-idf vectorization.

the corpus. This operation is implemented with the help of `TfidfVectorizer` class from `Python's` library. From Table 1, we can see that each tweet in our training data will have a dimensionality `(1, 126954` where `126954` is the total number of n-grams in our data. N-grams are limited by unigrams and bigrams (see Table 1). Each word in this tf-idf representation is represented as a one-hot encoded vector with dimensionality equal to the vocabulary size. After summation and normalization of all one-hot encoded vectors of the words in a given tweet, this tweet is transformed into a bag-of-words vector.

2.2 Word embeddings

The problem with tf-idf vectorized text representations is that the order of the words is disregarded. In addition, one-hot encoded vector representations are long and sparse what causes computational complexities. Importantly, the notion of semantic similarity between the words is also lost. An alternative way of representing a word is a *word embedding*. Here, the initial assumption is that there is a vocabulary with size `V` and each word from this vocabulary is mapped into a `d`-dimensional space. Word embeddings have several advantages over one-hot encoded vector representations. In particular, the former ones are shorter, dense, can be learned from the available data in different ways or reused as pre-trained ones. In this study, word embeddings are learned for a given dataset using neural networks. The performance of the trained word embeddings on a given sentiment classification task is compared to the performance of the pre-trained embeddings available as a part of GloVe, an open-source project at Stanford University.

2.3 SVM Classifier

SVM classifier serves as a baseline model in this study. However, the optimization task for this classifier is solved by Stochastic Gradient Descent (Zhang, 2004) algorithm instead of quadratic programming method. A variant of Huber loss called *modified huber* is used as a loss function for SVM classifier. Modified huber loss function demonstrated superior performance when compared to other loss functions during hyperparameter tuning phase.

Modified huber loss function is

$$L(y, f(x)) = \begin{cases} max(0, 1 - y\, f(x))^2, & \text{for } y\, f(x) \geq -1 \\ -4\, y\, f(x), & \text{otherwise} \end{cases}$$

where y corresponds to the class label and $f(x)$ is a SVM classifier score.

Elastic net regularization is applied, and that combines both L_1 and L_2 regularization terms from LASSO and Ridge regression respectively. This method was proven to be effective (Wang et al., 2006) in classification problems for SVM where the number of variables is much larger than training data $(p >> n)$.

2.4 Neural Networks

Neural networks made a significant impact in the field of natural language processing during the last decade. In particular, the arrival of large computational resources, massive amounts of data and architectures such as recurrent neural networks (RNNs) and transformers enabled transition from statistical methods in tasks like language modelling, sequence labelling and machine translation.

In this study, a type of RNN with long short-term memory (LSTM) cells is utilized for the classification task. However before diving into explanation of LSTM network, it is important to understand how RNNs work, what are their limitations, and how LSTM cells address those limitations.

2.4.1 RNN

To introduce LSTMs as a type of RNN, we should first talk about RNNs themselves, and why they are a suitable modelling approach for sequential data. To guide our explanations, we refer to Figure 1 as an example of RNN implemented in this study. However, instead of RNN cells, LSTM cells are utilized in recurrent hidden layer (see Figure 2).

In RNN, we can view our input tweet X as a sequence of T words so that $X = \{X_1, X_2, ..., X_T\}$ where each X_t represents a word. In our case, each X_t is a vector represented by a corresponding n-dimensional word embedding. The output $Y = \{Y_1, Y_2, Y_3\}$ is an (3, 1) - dimensional output of the final *softmax* layer where each of the values represents probability of a given tweet to belong to one of three classes. Although this neural network may look as a feed forward neural network (FNN),

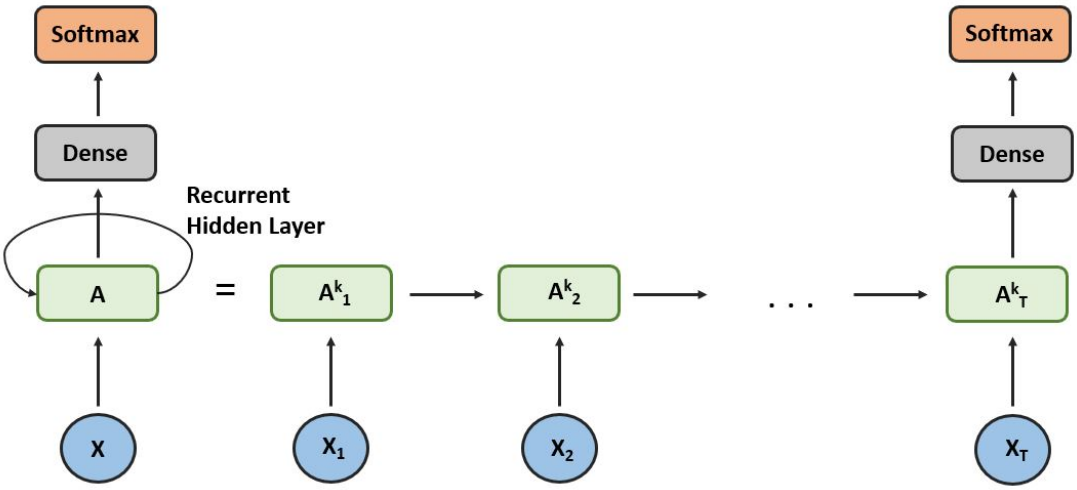

Figure 1: Unrolled recurrent hidden layer followed by a single dense layer.

the inner behavior of recurrent hidden layer is what differentiates it from FNNs.

Our classification task is an example of *many-to-one* architecture of RNNs. This means that network accepts as an input a sequence of words (many) and outputs a (3, 1) - dimensional vector (one) with its values corresponding to the probability of a given tweet X belonging to a certain class. In the hidden layer, X_t is processed at each timestep, and corresponding hidden activation vector with values of hidden units (assume k hidden units), A^k_t, is calculated. When X_{t+1}, the next element of the sequence, is processed, hidden activation vector A^k_{t+1}, is calculated using X_{t+1} and the values from hidden activation vector of the previous timestep, A^k_t. As a result, a series of hidden activation vectors are calculated where each A^k_t corresponds to the hidden activation vector with k hidden units generated at timestep t which also includes information from the previous hidden activation vectors A^k_{t-1} to A^k_0. This inclusion of previous activation values allows network to account for the sequential relation between the words at different timesteps. In many-to-one architecture, A^k_T, only the hidden activation vector calculated after the final step T is passed into the next layer. Then, this vector can be passed through a dense layer (similar to FNN) with some activation function $g(\cdot)$ and followed by the softmax layer to produce the output of the network. The model complexity can be increased further by adding more hidden units, an additional recurrent hidden layer or a several dense layers.

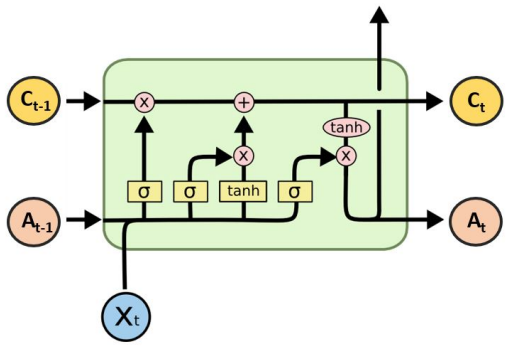

Figure 2: A closer look at LSTM cell and its gating mechanisms. This figure is adopted from Chris Olah's blog[3].

2.4.2 RNN with LSTM cells

The architecture of RNNs provides advantages when working with sequential data in comparison with FNNs and other machine learning algorithms. One of the problems with RNNs (as well as FNNs) is that they are unable to capture long-term dependencies due to the problem of vanishing gradients (Pascanu et al., 2013). Gradients of RNNs are calculated using backpropagation through time, and when the number of layers (or the length of sequence) increases, gradient updates become exponentially smaller which makes it difficult for the network to learn long-term dependencies in the sequence. To solve the problem of vanishing gradients, RNNs are updated with LSTM cells.

LSTM cells (Hochreiter and Schmidhuber, 1997) can be viewed as an enhancement to the architec-

[3] http://colah.github.io/posts/2015-08-Understanding-LSTMs/

ture of RNNs. Here, at a given timstep t, there are two outputs, hidden activation vectors A_t and cell state vectors C_t which carry information from the current and previous steps in the sequence. LSTM cells are using *gating mechanism* to decide how to update A_{t-1} and C_{t-1} values from a previous timestep before outputting A_t and C_t to the next timestep in the sequence. Thus, LSTM cells are designed with three gates:

- *Forget gate* - uses $\sigma(\cdot)$ *sigmoid* function of A_{t-1} and X_t to decide which information will be forgotten from C_{t-1}
- *Update gate* - uses $\sigma(\cdot)$ sigmoid function of A_{t-1} and X_t to decide which values to update in C_{t-1} and $tanh(\cdot)$ function of A_{t-1} and X_t to calculate new candidate values $\hat{C}_t$
- *Output gate* - uses $tanh(\cdot)$ to put values of C_t between -1 and 1 and $\sigma(\cdot)$ function of A_{t-1} and X_t to decide which information from C_t to output as A_t

We can also represent these operations in the form of mathematical expressions:

$$\Gamma_f = \sigma(w_f[A_{t-1}, X_t] + b_f) \quad (1)$$

$$\Gamma_u = \sigma(w_u[A_{t-1}, X_t] + b_u) \quad (2)$$

$$\hat{C}_t = tanh(w_c[A_{t-1}, X_t] + b_c) \quad (3)$$

$$C_t = \Gamma_u \cdot \hat{C}_t + \Gamma_f \cdot C_{t-1} \quad (4)$$

$$\Gamma_o = \sigma(w_o[A_{t-1}, X_t] + b_o) \quad (5)$$

$$A_t = \Gamma_o \cdot tanh\, C_t \quad (6)$$

where $\hat{C}_t$ denotes candidate values for updating cell state C_{t-1}, and Γ_f, Γ_u, Γ_o are forget, update and output gates respectively.

The key part of the above equations that allows to solve vanishing gradients problem is derivative $\frac{\partial C_t}{\partial C_{t-1}}$. When solving it, one can see that there is no exponential weight accumulation (that is present in RNNs) meaning that there is at least one way where the gradient does not vanish.

3 Data

In this study, data is used from 2020 Kaggle Competition. It consists of tweets and corresponding sentiment labels: *negative*, *neutral* and *positive*. Data was already separated into a training and test sets, 27466 and 3529 instances respectively. Thus, training set was further split into a new training and validation set. From Figure 3, it can be seen that sentiment labels in both datasets are relatively balanced. Before feeding it into our models, the data was tokenized using `spacy` library. In particular, the following steps were implemented:

- replacing ‘ character with ’ to let `spacy` apply splitting correctly.
- create tokens with alphabetical letters only
- transform all uppercase letters in a token into lowercase
- merge tokens back to obtain tokenized form of a tweet

As it was mentioned in 2.1, for SVM classifier all the tweets are transformed using tf-idf vectorization. Thus, each tweet from our data is represented by (1, 126954) - dimensional vector where 126954 is vocabulary size. Note that both unigrams and bigrams are used during transformation what obviously results into a large vocabulary size.

For LSTM network, each word is represented in the form of (1, 100) - dimensional word embedding which is either learned or taken in a pre-trained form from GloVe.

4 Methodology

The internal workings and design choice of different parameters for SVM classifer and LSTM network are described in 2.3 and 2.4.2 respectively. In this section, only the modelling pipeline is briefly outlined for each of the models.

4.1 SVM Classifer Model Pipeline

```
Pipeline([
('vectorizer',
TfidfVectorizer(ngram_range = (1, 2))),
('SGD',
SGDClassifier(loss = 'modified_huber',
penalty = 'elasticnet', l1_ratio = 0.9,
random_state = 42))])
```

As you can see, there are two steps in the pipeline. First, each tweet is tf-idf vectorized with parameter `ngram_range` specifying unigrams and bigrams. Second, vectorized representation is fed to `SGDClassifer` which is SVM classifier with SGD solver for optimization. Here, the loss function and type of regularization are specified accordingly. Apart from theoretical ground, the selection of the following parameters was also confirmed by utilizing k-fold cross validation with $k = 5$ parameter.

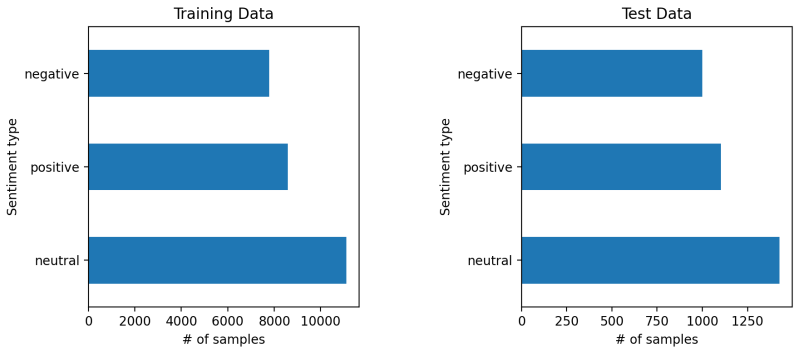

Figure 3: Distribution of classes in training and test datasets.

4.2 LSTM Network Model Pipeline

LSTM network is designed in `Keras`, an open-source software library that provides a `Python` interface for modelling artificial neural networks. LSTM network is implemented in three settings: with *trained embeddings* (TE), with *pre-trained word embeddings* from GloVe (PE) and with *retrained word embeddings* (RE) GloVe. The difference between the second and the third one, is the ability to use pre-trained embeddings from GloVe as initial parameters for embedding matrix in neural network. Thus, the aim is also to see if further training would allow network to customize general pre-trained embeddings to the task at hand.

TE LSTM Model

```
Sequential([
Embedding(input_dim = vocab_length,
output_dim = 100, mask_zero = True),
LSTM(64, activation = 'relu'),
Dense(64, activation = 'relu'),
Dense(3, activation = 'softmax')])
```

PE LSTM Model

```
Sequential([
Embedding(input_dim = vocab_length,
output_dim = emb_dim, mask_zero = True,
embeddings_initializer =
Constant(embedding_matrix),
trainable = False),
LSTM(64, activation = 'relu'),
Dense(64, activation = 'relu'),
Dense(3, activation = 'softmax')])
```

RE LSTM Model

```
Sequential([
Embedding(input_dim = vocab_length,
output_dim = emb_dim, mask_zero = True,
embeddings_initializer =
Constant(embedding_matrix),
trainable = True),
LSTM(64, activation = 'relu'),
Dense(64, activation = 'relu'),
Dense(3, activation = 'softmax')])
```

The above implementations were inspired by the guidelines of `Keras` library developer and AI researcher François Chollet (Chollet, 2017).

There is a common structure in all three LSTM models which was thoroughly described in 2.4.2. In essence, there is an embedding layer `Embedding`, where embeddings for each word in a given sequence are learned, this sequence of words each with its trained embedding vector is fed into `LSTM` layer which is also followed by `Dense` layer. In both of LSTM and Dense layers *relu* activation function is used. Finally, the outputs from a dense layer are fed into layer with softmax activation function (also known as *softmax layer*) to produce (3, 1) - dimensional vector of probabilistic outputs for a given sequence of words. The key difference between TE and PE settings is that, in the latter, embeddings are initialized with `Constant` class with `embedding_matrix` which comes from GloVe and maps each word to its corresponding embedding vector. The difference between PE and RE settings is that the latter has a parameter `trainable = True` which allows to initialize model with `embedding_matrix` and train it further to adjust to the given classification task.

5 Results and Discussion

Model performances are summarized in Table 2. The focus is on accuracy as evaluation metric, because, as it is mentioned in Section 3, the distribution of classes is close to balanced. SVM classifier

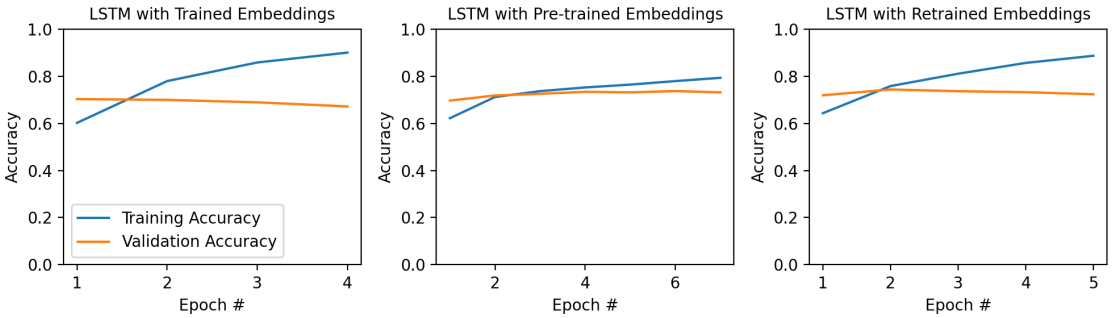

Figure 4: Training and validation accuracy for each of three neural models during the training process.

Model	Accuracy
SVM Classifier	0.72
TE LSTM	0.71
PE LSTM	0.73
RE LSTM	0.75

Table 2: Peformance evaluation results for each of the models.

achieves 0.72 accuracy and outperforms TE LSTM model. SVM classifier looses by a small margin to PE and RE LSTM models (0.73 and 0.75 accuracy respectively). When looking at training and validation curves of LSTM models in Figure 4, we can see that in TE and RE cases, both models overfit to the data, however, this is not the case in PE model. Although dropout regularization was tested for TE and RE models, no performance improvement was observed. In general, neural networks are complex models which need a very good understanding of their behavior to tune hyperparameters accordingly. When analyzing LSTM models by themselves, we can see that using pre-trained embeddings and re-training them further improves model performance. However, it is arguable whether one should still prefer neural models in this task given the performance of SVM classifier. It is also worth mentioning that SVM classifier is tuned with very task specific loss and regularization parameters which help to improve its performance.

5.1 Out-of-hand Sentiment Analysis

In this section, we wanted to demonstrate how RE LSTM network performs, and what it actually learns, using custom-generated out-of-hand examples (we call them out-of-hand because they are not present in training or test data). The structure is the following:

`example » [p-neg, p-neu, p-pos]`

where `p-neg`, `p-neu` and `p-pos` are probabilities of negative, neutral and positive sentiments respectively.

`ex1. had a bad day »`

$$0.856, 0.13, 0.014$$

`ex2. had so bad day »`

$$0.889, 0.099, 0.013$$

`ex3. had a good day »`

$$0.028, 0.07, 0.903$$

`ex4. had so good day »`

$$0.05, 0.086, 0.863$$

`ex5. had sooo good day »`

$$0.033, 0.048, 0.919$$

`ex6. What is the meaning of life? »`

$$0.143, 0.765, 0.092$$

`ex7. Why this world is sometimes so hard to love »`

$$0.29, 0.332, 0.378$$

As we can see model is able to learn sentiments of simple words such as "bad" and "good". It increases its confidence when the emotional aspect of the sentence is increased further by adverbs like "so". However, it does not increase its confidence when adding "*so*" to the example with "good day" but it increases its confidence when we add "sooo". One of the reasons can be that the model learns

from tweets where grammatical rules can be sometimes neglected, and people may emphasize their tweets by adding additional letters to the words such as "sooo", "coool" and etc. Two examples, ex.6 and ex.7, are probably more interesting to analyze. ex.6 is predicted to have a neutral sentiment, while for ex.7 the model allocates a uniform distribution of probabilities to each of the sentiments. One of the reasons behind that could be words "hard" and "love", where the former drives sentiment prediction in a negative direction, and the latter - in the position direction.

Let us also analyze some examples where context becomes important.
ex.8 I am greedy for knowledge »

$$0.616, 0.326, 0.057$$

ex.9 I became stronger after going through many struggles »

$$0.544, 0.382, 0.074$$

In ex.8, the model output is skewed by the word "greedy", and it fails to understand that "greed for knowledge" can hold a positive rather than a negative sentiment. Finally, in ex.9, the model output is skewed by "all these struggles" and does not capture the whole context of the sentence.

Finally, we compare probabilistic outputs of SVM model for ex.8 and ex.9.
The results are the following: ex.8 »

$$0.22, 0.568, 0.212$$

ex.9 »

$$0.175, 0.685, 0.14$$

We note that SVM model is quite confident that ex.9 is neutral one, though it has less confidence in ex.8

6 Conclusion

To sum up, we can observe that our trained models detect the sentiment of a given tweet with the accuracy ranging from 0.72 to 0.75. Such a performance is barely acceptable if one tries to build a strong classifier. However, another focus of our study is to understand whether transfer learning can significantly improve performance of a deep neural model when compared to a machine learning model. It is demonstrated that transfer learning and further retraining can indeed improve performance of a deep neural model. Nevertheless, when compared with well tuned machine learning model, deep neural model does not outperform it by a significant margin. There can be several reasons for that. First, a small amount of available training data can be a limiting factor. Second, hyperparameters of a deep neural model should be probably adjusted in a more nuanced way. Third, an architecture choice of a deep neural model can be elaborated further to capture the contextual information in a given sequence. All of these points showcase that despite using transfer learning, the expectations should not always be that the model will start working much better. One should always be willing to spend enough time on design choices to understand behavior of model in a better way.

Limitations

There are several limitations of this study that were already briefly mentioned in the Section 6. In particular, LSTM networks can be potentially better adjusted to improve their accuracy values. One of the ways of doing it is to try to apply different types of regularization. It is important to rigorously examine architecture choice for all three neural models, and experiment with other suitable alternatives. In addition, the quality of trained word embeddings can be compared to the quality of retrained ones with techniques such as similarity or analogy benchmarks. Simple yet important criteria of saving computational resources when using pretrained embeddings should not be omitted and can be illustrated with the help of different test runs.

Supplements

All the code for the models implemented in this study is available under the following repository: https://github.com/faridmusayev/Text_Mining_Project.

References

F. Chollet. 2017. *Deep Learning with Python*. Manning.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9:1735–80.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page III–1310–III–1318. JMLR.org.

Li Wang, Ji Zhu, and Hui Zou. 2006. The doubly regularized support vector machine. *Statistica Sinica*, 16(2):589–615.

Tong Zhang. 2004. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 116, New York, NY, USA. Association for Computing Machinery.