Product Categorization with LSTMs and Balanced Pooling Views

Michael Skinner*
Seattle, Washington

ABSTRACT

Recurrent Neural Networks (RNNs), and LSTMs in particular, have proven competitive in a wide variety of language modeling and classification tasks. We explore the application of such models to the Rakuten Data Challenge, a large-scale product classification task. We show that a straightforward network architecture and ensembling strategy can achieve state of the art results when trained effectively. We also demonstrate the positive impact of tightening the connections between recurrent and output layers through the use of pooling layers, and introduce the Balanced Pooling View architecture to take advantage of this. Our final solution, produced with a bidirectional ensemble of 6 such models, achieved a weighted F1 score of 0.8513 and won the challenge by a wide margin.

KEYWORDS

Rakuten Data Challenge; text classification; neural networks; deep learning

CEUR-WS.org/Vol-2319/ecom18DC_paper_9.pdf

1 INTRODUCTION

The Rakuten Data Challenge. The goal of the challenge is to classify each product into one of 3008 categories given the title of the product. As an additional challenge, the categories are highly unbalanced and intrinsically noisy as sellers take responsibility for categorizing their own products.

The dataset contains 1 million categorized products, of which 80% were available for training and the remaining 20% were reserved for a test set. The test set categories were not available during the challenge, except through a provisional leaderboard to score solutions on a subset of them.

Solutions were scored using the category-weighted average of F1 scores, with the leaderboard also showing weighted precision and recall. That is to say, if there are some labeled documents D, with a set of documents D_c per category c in C, and a corresponding set of documents $\hat{D}_c$ per predicted category, then the weighted F1-score is:

$$
\begin{aligned}
recall_c &= \frac{|D_c \cap \hat{D}_c|}{|D_c|} \\
precision_c &= \frac{|D_c \cap \hat{D}_c|}{|\hat{D}_c|} \\
F1_c &= 2 \cdot \frac{precision_c \cdot recall_c}{precision_c + recall_c} \\
weighted\text{-}F1 &= \frac{1}{|D|} \sum_{c \in C} |D_c| \cdot F1_c
\end{aligned} \tag{1}
$$

It is worth noting that the weighted recall used for this challenge is equal to accuracy. With some small abuse of notation in the last line:[1]

$$
\begin{aligned}
weighted\text{-}recall &= \frac{1}{|D|} \sum_{c \in C} |D_c| \cdot recall_c \\
&= \frac{1}{|D|} \sum_{c \in C} |D_c| \cdot \frac{|D_c \cap \hat{D}_c|}{|D_c|} \\
&= \frac{1}{|D|} \sum_{c \in C} |D_c \cap \hat{D}_c| \\
&= \frac{|D \cap \hat{D}|}{|D|}
\end{aligned} \tag{2}
$$

Our contributions. Our contributions are as follows: 1) We achieve state-of-the-art results for this dataset, applying the methodology of [6, 9] by focusing on regularizing and training straightforward LSTM architectures effectively. 2) We extend the above methodology to the problem of sizing a configurable model architecture and evaluating potential design improvements. We note that the resulting model architectures are robust to changes to the training schedule, and vice versa. 3) We introduce the Balanced Pooling View architecture, an extension of concat pooling from [6], to extract signal from variable-length recurrent outputs, and show that this improves performance compared to existing architectures.

Overall challenge results. Our approach won the challenge by a wide margin. Our final submission achieved an F1 score of 0.8513 on the test set, maintaining the recall of the next best solution while improving precision from 0.8425 to 0.8697, decreasing the false positive rate by 17.5% from 15.75% to 13%.[2] Our position at or near the top of the leaderboard was held for much of the contest, an advantage of the simple but methodical approach to model design, training, and iteration. In addition, our models do not take long to train. It takes less than 24 hours on a commodity GPU to train our final 6-network ensemble.

2 RELATED WORK

Several recent papers have shown the broad applicability of straightforward LSTM architectures. [6] were able to use transfer learning on top of the same pre-trained LSTM architecture to achieve state-of-the-art results on several text classification tasks. That work builds on the work of [9], who used a single AWD-LSTM architecture to achieve state-of-the-art results on both character- and word-level language modeling benchmarks.

[1] The numerator $D_c \cap \hat{D}_c$ is the number of true positives i.e. correctly classified documents for category c. The sum of this over all categories C is the number of correctly classified examples in the entire dataset.

[2] While we can't technically avoid false positives, we can effectively "not guess" by picking a very rare category where they will not have a noteworthy impact on the weighted scores.

*Correspondence to: research@mcskinner.com

Copyright © 2018 by the paper's authors. Copying permitted for private and academic purposes.
In: J. Degenhardt, G. Di Fabbrizio, S. Kallumadi, M. Kumar, Y.-C. Lin, A. Trotman, H. Zhao (eds.): Proceedings of the SIGIR 2018 eCom workshop, 12 July, 2018, Ann Arbor, Michigan, USA, published at http://ceur-ws.org

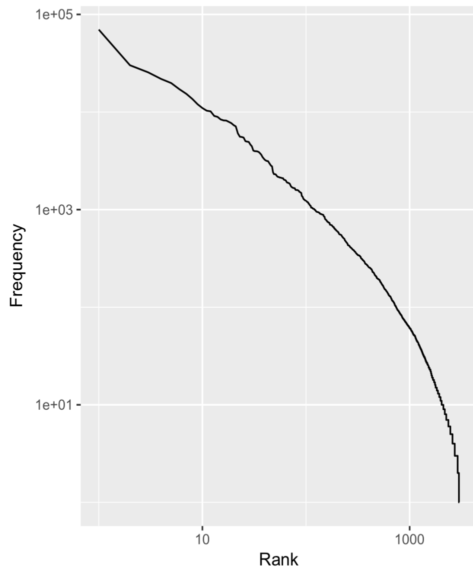

Figure 1: The category frequencies are Zipfian for the most common categories, but without the long tail for rarer categories.

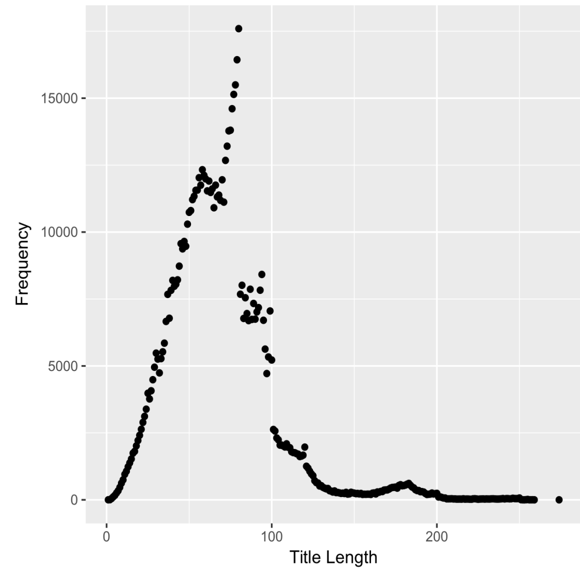

Figure 2: Product titles range in length from 1 character up to 274. The bulk of the data appears to be normally distributed around a mean just below 60, with a long tail. However there is an obvious artifact which peaks at 80 characters. This is presumed to be some systemic behavior, like a display limitation.

There is also increasing interest in the topic of robust training schedules, often with aggressively high learning rates. [8] introduced the cosine annealing schedule, which follows the descending shape of a cosine wave. Training starts with a high learning rate, then drops to lower learning rates, before plateauing to finish the cycle. Results obtained after one or more iterations were found to have better generalization error than those from other learning rate schedules. A symmetrical low-high-low triangular learning rate schedule from [14], dubbed Cyclical Learning Rates (CLR) was shown to have similar performance and generalization properties. It was promptly refined into the `1cycle` policy in [15], which added an additional annealing phase and suggested a single policy cycle rather than several as in [8, 14]. The results from [6] were obtained with a Slanted Triangular Learning Rate (STLR), an asymmetrical variant of CLR in which learning rate is increased over less time and then decreased for more.

3 DATA PREPARATION

3.1 Exploratory Analysis

With 800k examples available for training, this constitutes a fairly large dataset. With 3008 categories to pick from, it also constitutes a large-scale classification problem. For comparison, the text classification datasets in [6] ranged in size from 5.5k to 560k examples, but with only 2 to 14 categories. The categories are also highly unbalanced, although we see in Figure 1 that the tail is not quite as long as if it were Zipfian.

Due to time constraints, the product titles were not examined closely. A brief overview is given in Figure 2, but only a few things were relevant to the modeling: 1) The maximum sequence length of 274 still allows for reasonably large batches to fit into GPU memory. 2) The wide distribution of lengths implies some potential for inefficiency if long titles are placed into batches next to short titles, which results in a large number of meaningless pad tokens to process.

3.2 Tokenization

Due to the high incidence of symbols and alphanumeric product codes, we opted for character-level tokenization. Word-level tokenizers are not well suited for this task, and poor tokenization leads to an unnecessarily large and sparse parameter space to optimize over. To further alleviate sparsity concerns, any characters that occurred fewer than 10 times were replaced with an "unknown" token. Adding in typical tokens for padding, beginning, and end of string, the final vocabulary size was 128 tokens.

Categories were treated as an opaque classification label. No attempts were made to leverage the existing taxonomy rather than letting the networks learn their own. Representation learning is quite powerful, and there is some risk of adversely biasing the results by imposing a hierarchy in advance.

The int-encoding of both categories and characters was frequency ordered, such that tokens with higher frequencies had lower indices. This is a common practice, and allows for simple cutoff-based implementations for techniques like adaptive softmax from [3].

3.3 Training/Validation Split

We set aside 25% of the training data as a validation set, using stratified sampling to ensure good representation across categories. We chose this set to have 200k examples, like the test set, in order to

ensure that all model changes could be accurately evaluated during the challenge. This reduced reliance on the public leaderboard and enabled a methodical process for new ideas from introduction to deployment.

At the end of the challenge, with the model architecture finalized and training parameters dialed in, the validation data was used to train one last batch of models for the final submission. The 33% increase in training data improved the F1 score on the leaderboard by nearly 0.01, which was enough to change the final standings from a tossup to decisive.

4 MODEL ARCHITECTURE

4.1 Baseline Model

Following the lead of [6, 9], the baseline model uses a straightforward LSTM architecture. Character tokens are looked up in an embedding of size n_e, which feeds into a multi-layer LSTM of width n_w and depth n_d, which feeds into a linear layer of width $n_c = |C|$ to produce a score for each category. In addition, there are dropout layers after the embedding, between the recurrent LSTM layers, and after the LSTM output. Those have respective dropout probabilities d_e, d_r, and d_o.

The model sizing process began with a deliberately small architecture. Following a rule of thumb, $n_e = \max(|V|/2, 50) = 50$ for our 128-wide vocabulary. The values $n_w = 64$ and $n_d = 2$ were chosen to be deliberately small, and because the latter is the PyTorch default. Dropouts were chosen based on an existing example, with $d_e = 0.15$, $d_r = 0.25$, and $d_o = 0.35$.

This initial model was quickly tuned and tested as per the methodology in 5.2. After initial results were shown to quickly plateau, n_w was doubled and the model rerun until this phenomenon stopped at a width of 512.

4.2 Concat Pooling

The idea of concat pooling, as proposed in [6], is to augment the last LSTM output with average- and max-pooled aggregates over the entire sequence. The idea is that the network can become excited about certain categories at any point during the sequence, and the average and maximum are able to capture this information more easily than the final output. In our experiments, this was found to be true, increasing validation precision, recall, and F-score by 0.01, a substantial increment on the leaderboard. We note a few likely advantages from the pooling layers:

(1) Short-circuiting the outputs from all time steps directly to the loss function means direct feedback for the earlier time steps, leading to faster and more reliable convergence.
(2) The same direct connection introduces a sort of data augmentation for the RNN output layer. Instead of only solving the translation problem for the last hidden state, it benefits from successfully understanding the intermediate states as well.
(3) The average pooling is similar in spirit to the ResNet from [5], in which subsequent computations only need to make additive adjustments, i.e. error corrections, on the output from the previous layer. It is possible that this idea of sharing responsibility for the overall prediction has nice generalization properties that are shared across domains.

4.3 Balanced Pooling View

Since average- and max-pooling aggregations were able to help the output layer extract more information from the recurrent layer, it stands to reason that additional aggregates might also help. Although not commonly mentioned, min-pooling is trivial to implement as $minpool(x) = -maxpool(-x)$, and therefore easy to test. In addition, we believe that the use of both min- and max-pooling might encourage the network to use the full range of possible activations, and therefore become more expressive, than when using max-pooling alone to prefer large activations. We call this a Balanced Pooling View (BPV) architecture, and find that it increased the F-score by 0.006 for a single network, trained for 40 epochs, as compared to concat pooling without the min-pool addition. This architecture and its precursors are outlined in Figure 6.[3]

It is worth noting that the widening of the pooling options also expands the size of the final output weight matrix, and therefore the capacity of the model. In addition, independent dropout across the views means that most of the 512-wide recurrent outputs will be at least partially observable most of the time.

To understand the impact of pooling alone, we trained a separate network with the same capacity as the concat pooled networks but multiple copies of the final RNN output rather than the different pooled views. This network did not surpass the performance of the non-pooled baseline model,[4] so we conclude that the performance improvement comes from increasing the ability of the network to extract signal from the recurrent outputs. See Figure 6 for a visualization of the final BPV architecture.

4.4 Ensembling

Previous work has shown the benefits of bidirectional training on sequence tagging ([12]) and text classification ([6]). Reversing the input sequence acts as a sort of data augmentation, and training a second independent model adds the known advantages of ensembling as in [11].

Bidirectional training is more powerful than a simple 2 network ensemble. Various ensembles of 2 networks in the same direction, forward or backward, were never as good as bidirectional ensembles. The effect of directionality is explored a bit more in Section 5.7.5. This technique improved our model performance as well, increasing the F1 score by 0.013 with pooling and 0.011 with BPV, and is another significant increment on the leaderboard.

As many additional models and training parameters were tested, it was found that it was not uncommon for solutions to converge to the current best single model result. Despite the lack of single model gains, building an ensemble from several models was found to improve overall performance, continuing to fall in line with results from [11]. An 8 network ensemble, 4 each forward and backward, increased the F-score by another increment of nearly 0.01 on the validation set.

[3] This has been placed on the last page of this paper to avoid disruption to the layout.
[4] To make things worse, it also had trouble converging.

5 EXPERIMENTS

5.1 Implementation Details

All models were implemented in PyTorch ([10]) and trained on a single Paperspace P5000 instance with a 16GB NVIDIA Quadro P5000 with 2560 CUDA cores. We used the cross-entropy loss function, which implies a softmax over the final activations.

5.2 Training Methodology

Following the advice in [13], the batch size was increased until GPU memory struggled to keep up, which it did at a batch size of 256. Due to large variances in the text lengths leading to an unnecessarily large and inefficient number of pad tokens per batch, it was desirable to organize the batches to have similar lengths. In order to avoid the same batch ordering every time, the sorting is done in chunks of several batches at a time. Sorting 50 batches of examples at a time cut training times by approximately half, and it also appeared to have a regularizing effect. This is perhaps due to an observed correlation between title length and categorization difficulty, in which shorter titles are more difficult. Experiments with a fully randomized training set not only lost the 2x speed gain, but also resulted in poorer performance. Changing the width of the sorting window to 10 or 250 batches did not appear to impact the results.

Models were optimized using SGD with momentum, using a cross-entropy loss function. Adam was attempted a few times, but it required learning rates that were two orders of magnitude smaller, and still exhibited difficulty in converging. This is in line with research from [16] on the weakness of adaptive gradient methods. [4] provides further reason to believe that SGD leads to faster training and better generalization error.

In general we rejected training or architectural changes that substantially reduced the learning rate. This follows the insight from [13] that high learning rates have a regularizing effect, improving generalization error while also reducing training time. Our rule of thumb is a sort of corollary: if high learning rates are desirable, then it is undesirable when modeling variants are surprisingly intolerant to those same learning rates.

5.3 Hyperparameter Tuning

When training each new model architecture, we followed the disciplined approach from [13]. First the peak learning rate was chosen using a learning rate sweep as described in [14]. The sweep starts with a low learning rate, at which nothing much happens. As the learning rate continues to increase, the training and validation loss eventually drop, level out, and then diverge. An example of this is shown in Figure 3.

Good peak rates were found to exist just short of the basin in which loss has leveled out. Our methodology was to choose the first learning rates to be on the high end of plausible, sometimes in the basin, and then decrease the estimate until a short training run of 5 epochs trains without exhibiting signs of divergence or oscillation at the peak rate. Peak learning rates found this way were near 1, and sometimes even higher. For the highest performing models, the learning rates were further tuned by hand within a small range to squeeze out remaining performance gains. The majority of our models, including those used for our final submission, were trained with a learning rate of 0.8.

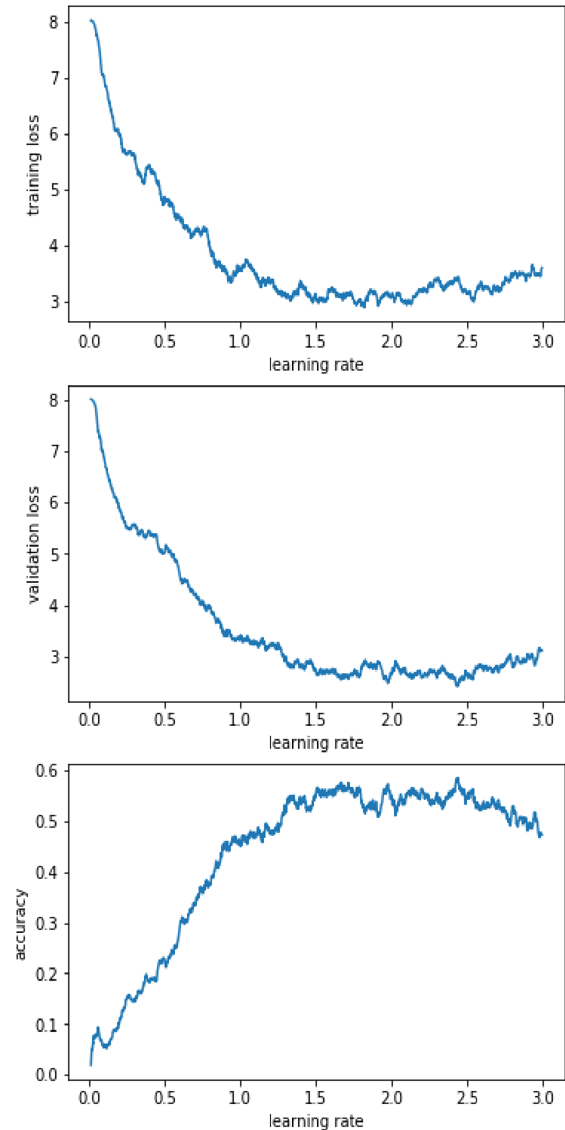

Figure 3: A typical learning rate sweep, run for 1 epoch with a learning rate that increases linearly to 3. The learning rate levels off substantially around 0.8, with one last drop around 1.3. Using a 40 epoch `1cycle` policy, peak learning rates of 0.7, 0.8, and 1.0 were all found to perform well.

Momentum was also chosen according to the approach in [13], running a learning rate sweep for several options and comparing the loss curves. Good momentums have nice convergence properties like a quick descent, a low minimum loss, and a longer basin before divergence. The best momentums for this challenge were around 0.95, which is fairly high given the observation in [9] that momentum is not usually preferred for language modeling. While our experiments with low momentum SGD were able to achieve

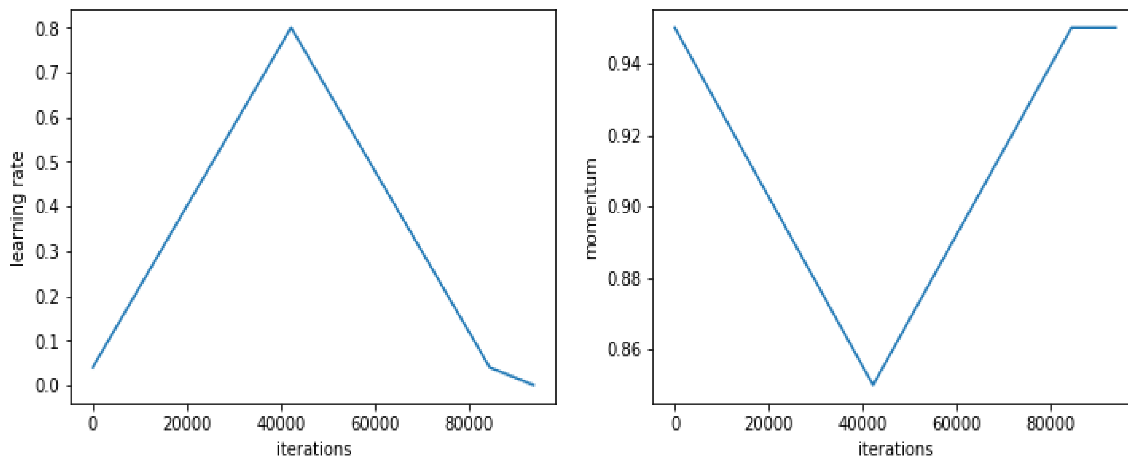

Figure 4: A typical `1cycle` policy with a peak learning rate of 0.8, an initial learning rate 20x smaller, momentum ranging from 0.95-0.85, and with 10% of the iterations reserved for final annealing at very low learning rates. This example was used on a training run of 40 epochs with 2344 batches per epoch.

very high learning rates, as high as 10, the models were not able to converge to low losses.

Weight decay was chosen similarly, with early experiments quickly showing that weight decay above 0 led to difficulty with convergence. Due to time considerations, this parameter was left at 0 for the remainder of the SGD runs rather than trying to tune it further.

5.4 Training Schedule

We used the `1cycle` learning rate policy from [15]. Learning rates follow a low-high-low triangular schedule as in [14], followed by a few iterations of increasingly very low learning rates. The typical low rate used was 10-20 times smaller than the peak, and the very low rate is a few orders of magnitude smaller still. See Figure 4 for an example, with the parameters selected using the methodology in Section 5.3. A few variations like the Slanted Triangular Learning Rate (STLR) from [6] were tested and not found to perform any better or worse than the `1cycle` policy. The results were found similarly insensitive to the ratio of the peak learning rate to the lower starting value.

In the early phases of the competition, networks were trained for 20 epochs using a `1cycle` policy with a peak learning rate of 1,[5] with the remaining parameters as specified in Figure 4. In the later phases of the competition networks were trained for 40 epochs, with the peak learning rate reduced to 0.8. This is exactly the parameterization shown in Figure 4.

When testing new model architectures or training parameters, we used short training runs of 5 epochs to filter for clear improvements. This was enough to distinguish ideas, while keeping the training time within 30 minutes. Each doubling of the schedule length to 10, 20, and then 40 epochs was shown to improve results, and so the full training schedules were again chosen based on timing considerations. In the early phases of the challenge we used a schedule of 20 epochs so that a competitive 2 network ensemble could be trained in under 4 hours, enough to test a few quality ideas per day. The final schedule of 40 epochs was chosen so our tuned model, a bidirectional ensemble of 6 BPV networks, could be trained in under 24 hours. There were no apparent gains from a 60 epoch training cycle, though we also did not take time to carefully re-tune the hyperparameters.

[5]For the most common network architecture. When dropout was increased, learning rate was decreased to balance the increased regularization from higher dropout.

5.5 Model Variants

This section outlines the most interesting of the various extensions that we tried to add on to the basic model. None led to a performance improvement, but several maintained the same performance.

5.5.1 Larger Models. While decreasing the LSTM width by a modest amount did decrease performance, no increases to the model width or depth were able to increase model performance. Adding additional layers before or after the LSTM led to overfitting or otherwise poor convergence. Increasing LSTM depth n_d to 3, increasing the LSTM width n_w by 50%, and increasing the embedding size n_e to 64 all led to the same performance as the baseline model.

5.5.2 Regularization Tuning. We tested a variety of changes to the dropout magnitude and mix, and none were found to improve F1 score. Lowering the dropout in any of the layers was prone to problems with overfitting. In particular, $d_o = 0.1$ is a common choice, but did not add enough regularization and led to overfitting in our experiments. This includes experiments in which we increased to $d_e = 0.4$ and $d_r = 0.4$ to match the tuning from [6].

Increasing the dropout by a consistent factor across all layers resulted in better cross-entropy loss, but did nothing to improve discriminative accuracy. We also tested the introduction of a batch normalization layer right after the LSTM but before dropout and

decoding. The resulting networks were sometimes competitive, but prone to diverging during the fitting process.

5.5.3 RNN Variants. Swapping out the LSTM for simpler models like GRU led to degraded performance. The QRNN model from [2] was found to be much faster to train as advertised, but also suffered from slightly degraded performance compared to LSTM.

5.5.4 Adaptive Softmax. Given the unbalanced nature of the categories, it seems worthwhile to focus network bandwidth on high-frequency categories rather than the rare ones. Adaptive softmax, as proposed in [3], does this by partitioning the output categories and then using increasingly small bottleneck layers in front of each partition to limit the number of weights that need to be learned. No variation of this technique managed to improve upon the concat-pooled network, which apparently was not having trouble with this number of categories as compared to the $O(million)$ weights to which adaptive softmax intends to scale.

5.5.5 Training Variants. Noting the positive results that [1] had with an auxiliary loss function, we experimented with accuracy as a weighted component of the loss function. Even at high weightings this did not appear to have a substantial effect on performance.

5.6 Prediction Generation

The model is trained according to a cross-entropy loss function, but the challenge is scored according to a weighted F1-score. A simple highest probability wins strategy is not optimal in this case, and it is better to select a discriminative cutoff for each category to maximize an estimate of the F1-score. This can be done directly from the probabilities output by the network. Assuming well-calibrated probability estimates...

...the estimated number of true examples n_{true} for each category is the sum of probabilities for that category.

...the estimated number of true positives n_{tp} given a certain discriminative cutoff is the sum of probabilities that make the cut.

...the number of guessed positives n_{guess} given a cutoff is the number of probabilities that make the cut.

Given those estimates, it is possible to estimate the precision, recall, and F1-score for each potential cutoff. Then the cutoff is chosen by taking that for which the F1-score is maximized. It is straightforward to compute this efficiently by sorting the probabilities and using cumulative computations. See Algorithm 1 for details. This strategy tends to increase precision at the expense of recall, boosting the validation precision of bidirectional BPVs from 0.828 to 0.854 and an F1-score of 0.827 to 0.836, but at the cost of dropping recall from 0.833 to 0.826.

5.6.1 Probability Calibration. Since the final probability estimates are not guaranteed to be well calibrated, we apply a simple piecewise linear model to ensure that the probability estimates are suited for the F1 tuning. To do so, we first computed the actual accuracy for each 1% increment of predicted probability. A prediction of 20% should be correct 20% of the time, and if it is only correct 10% of the time then the raw estimate needs to be halved to get a true probability.

Figure 5 shows these raw calibration factors, which are a bit noisy. Rather than using these directly, we applied a simple moving average to visualize the broader trend. Since this trend appeared piecewise linear, we applied such a fit to the data. Figure 5 shows these smoothed probability factors as well. Applying these adjustments to true up the raw network probabilities had a small but positive effect on the validation performance, indicating an improved fit. However, to be conservative, these estimates were not applied to our final submission.

ALGORITHM 1: F1-Score Optimization

Data: P, probability estimates for each observation, for a single category.

Result: $\hat{p}_{cut}$, an F1-optimizing cutoff below which values in P should be rejected as predictions.

$n_{true} \leftarrow sum(P)$
sort P in descending order

$n_{tp,0} \leftarrow 0$
$\hat{p}_{cut} \leftarrow 0$
$score_{best} \leftarrow 0$
for $i \leftarrow 1$ **to** $|P|$ **do**
  $n_{guess,i} \leftarrow i$
  $n_{tp,i} \leftarrow n_{tp,i-1} + p_i$
  $rec_i \leftarrow n_{tp,i}/n_{true}$
  $prec_i \leftarrow n_{tp,i}/n_{guess,i}$
  $score_i \leftarrow 2 \cdot \frac{rec_i \cdot prec_i}{rec_i + prec_i}$
  if $score_i > score_{best}$ **then**
    $\hat{p}_{cut} \leftarrow p_i$
    $score_{best} \leftarrow score_i$
end

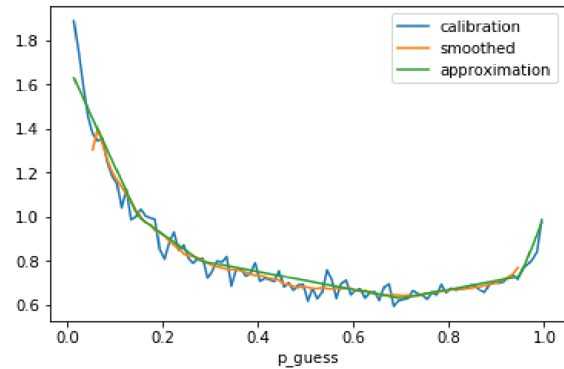

Figure 5: Raw probability estimates from the ensemble tend to be overconfident. Low probability estimates need a bit of a boost, while mediocre probabilities need to be deflated. A piecewise estimate is very close to the smoothed calibrations.

5.7 Results

5.7.1 Model Architecture. Table 1 shows the validation set performance for each iterative improvement of the network architecture, using the preliminary training schedule of 20 epochs and a simple best-wins strategy for selecting the prediction. We leave accuracy out of the tables, since it is the same as recall as shown in Equation 2. The introduction of concat pooling increases performance by 0.01 or so on all evaluation metrics, and BPV adds an addition 0.003 on top of that. Using pairs of bidirectional networks led to another boost of 0.01, but that also requires twice as long to train. Section 5.7.5 provides a more apples-to-apples comparison.

5.7.2 Training Schedule. In Table 2 we show the benefit of increasing the training schedule from 20 to 40 epochs. The concat pooled networks benefit from a modest scoring boost of 0.001 to 0.002 for a single network and 0.003 to 0.004 for a bidirectional pair. The BPV architecture took better advantage of the increased time, with scores growing 0.004 to 0.005 for a single network and 0.006 to 0.007 for the bidirectional network. In taking advantage of the extra epochs, BPV was able to grow the performance gap to alternatives and highlight the increased modeling capacity of the new architecture.

5.7.3 F1 Optimization. Table 3 shows the results after the F1 optimization procedure. The use of F1 optimization has a clear positive impact on both precision and F-score. Precision is increased by nearly 0.03 for all networks, while F-scores are all higher by 0.01 or so. This comes at a smaller cost to recall, which falls off by up nearly 0.01 for some model architectures. This approximately 3 to 1 tradeoff of recall for precision is a good one to make in terms of F1 score, and a result of choosing to break any near-ties away from large and precise categories.

5.7.4 Ensembling. In Table 4 we show the performance improvement for increasing ensemble sizes, using later stage networks trained for 40 epochs. The 1-pair ensemble here is the same as the bidirectional BPV results in Table 2. There is a big step when increasing from 1 to 2 pairs, but results quickly level off after an ensemble of even 4 pairs. Our final submission was generated by 3 pairs of networks trained with validation data included. A solution with 4 pairs decreased the test scores slightly, and so we reverted to a previous submission.

5.7.5 Bidirectionality. Finally we examine the impact of directionality on validation set results. Table 5 shows the best-wins F1 scores for a few ensembles, while Table 6 shows the optimized F1 scores. Interestingly, we found that reverse networks perform slightly better than forward networks. Balanced bidirectional ensembles outperformed equivalently sized one-way ensembles in either direction.

Compared to the results in Table 2, it is clear that the bulk of the improvement comes from ensembling. With only forward networks a single model reaches a best-wins F1 of 0.814, adding a second network improves to 0.825, and doubling the ensemble to 4 networks improves again to 0.830. By comparison, an F1 increase of 0.002 for reversing every other network is more modest. It is nonetheless a noticeable improvement and does not add to the training time, so it is worth including as a part of the ensembling strategy.

Table 1: Model Performance (Best-Wins, 20 Epochs)

Model	Precision	Recall	F-Score
Baseline	0.795	0.804	0.796
Concat Pooling	0.806	0.813	0.806
Balanced Pooling View	0.809	0.816	0.809
Bidirectional (CP)	0.820	0.826	0.819
Bidirectional (BPV)	0.821	0.827	0.820

Table 2: Model Performance (Best-Wins, 40 Epochs)

Model	Precision	Recall	F-Score
Concat Pooling	0.807	0.814	0.808
Balanced Pooling View	0.814	0.820	0.814
Bidirectional (CP)	0.823	0.829	0.823
Bidirectional (BPV)	0.828	0.833	0.827

Table 3: Model Performance (F1-Optimized, 40 Epochs)

Model	Precision	Recall	F-Score
Concat Pooling	0.838	0.805	0.819
Balanced Pooling View	0.842	0.812	0.824
Bidirectional (CP)	0.852	0.821	0.833
Bidirectional (BPV)	0.854	0.826	0.836

Table 4: Ensemble Performance (Best-Wins, 40 Epochs)

Ensemble Size (in pairs)	Precision	Recall	F-Score
1	0.828	0.833	0.827
2	0.833	0.838	0.832
3	0.835	0.840	0.834
4	0.836	0.841	0.835
5	0.837	0.841	0.835

Table 5: Ensemble F1 Scores (Best-Wins, 40 Epochs)

Model	2 Networks	4 Networks
Forward	0.825	0.830
Reverse	0.827	0.831
Bidirectional	0.827	0.832

Table 6: Ensemble F1 Scores (F1-Optimized, 40 Epochs)

Model	2 Networks	4 Networks
Forward	0.835	0.839
Reverse	0.835	0.840
Bidirectional	0.836	0.841

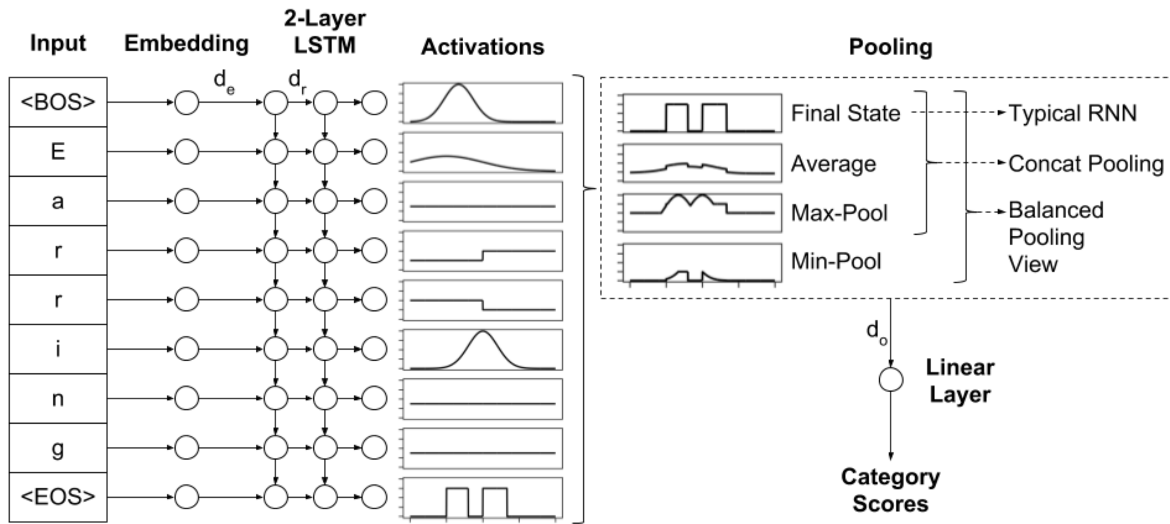

Figure 6: An illustration of the core RNN architecture, as well as Concat Pooling and BPV enhancements.

6 FUTURE WORK

Recalling the model refinements in section 5.5, we note that the training process achieved remarkably similar results across a wide variety of mutations to the model architecture. This was true of both model changes as well as modifications to the learning rate schedule, and is an encouraging result since it implies saturation of the current model architecture and training methodology. It also implies that further improvement is more likely to come from some enhancement to either the input representation, or from additional improvements to the extraction of signal from the recurrent layer as in sections 4.2 and 4.3.

The input representation at this point is quite simple, and has not yet been well explored. The cutoff of 10 occurrences for character inclusion has not been challenged, and it is possible that more tokens could increase the model power. If that conjecture is true, then it might also be worthwhile to try a hybrid word level tokenization in which rare words are split into characters. Another alternative is to try a hierarchical model as proposed in [7], who were working on a short-phrase language identification task which bears some similarity to this one. The idea of avoiding sparsity problems by computing the word embeddings as a convolution over character embeddings seems like it might apply well to this problem.

It is also possible that there are additional pooling or aggregation techniques which could bolster the connection between recurrent and output layers. An attention mechanism is one option, which could use convolutions as in [1] to weight the most interesting time steps. Multiple such attention vectors could also be generated and viewed together, and network capacity could be tuned by increasing both the complexity and number of such signal extractors.

ACKNOWLEDGMENTS

The author would like to his colleagues at Duetto for supporting and embracing this research, even though it was done off the clock. Thank you as well to the fast.ai community for developing and supporting an amazing MOOC and deep learning library. Finally, thank you to the challenge organizers for fostering a friendly and collaborative environment around this new dataset, providing valuable feedback, and for answering the author's numerous questions throughout the process.

REFERENCES

[1] Mikolaj Binkowski, Gautier Marti, and Philippe Donnat. 2017. Autoregressive Convolutional Neural Networks for Asynchronous Time Series. *CoRR* abs/1703.04122 (2017).

[2] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. 2016. Quasi-Recurrent Neural Networks. *CoRR* abs/1611.01576 (2016).

[3] Edouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou. 2017. Efficient softmax approximation for GPUs. In *ICML*.

[4] Moritz Hardt, Benjamin Recht, and Yoram Singer. 2016. Train faster, generalize better: Stability of stochastic gradient descent. In *ICML*.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), 770–778.

[6] Jeremy Howard and Sebastian Ruder. 2018. Universal Language Model Fine-tuning for Text Classification. arXiv:arXiv:1801.06146

[7] Aaron Jaech, George Mulcaire, Shobhit Hathi, Mari Ostendorf, and Noah A. Smith. 2016. Hierarchical Character-Word Models for Language Identification. In *SocialNLP@EMNLP*.

[8] Ilya Loshchilov and Frank Hutter. 2016. SGDR: Stochastic Gradient Descent with Restarts. *CoRR* abs/1608.03983 (2016).

[9] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2017. Regularizing and Optimizing LSTM Language Models. arXiv:arXiv:1708.02182

[10] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[11] Michael P. Perrone. 1993. When Networks Disagree: Ensemble Methods for Hybrid Neural Networks.

[12] Matthew E. Peters, Waleed Ammar, Chandra Bhagavatula, and Russell Power. 2017. Semi-supervised sequence tagging with bidirectional language models. In *ACL*.

[13] Leslie N. Smith. 2018. A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay. arXiv:arXiv:1803.09820

[14] Leslie N. Smith and Nicholay Topin. 2017. Exploring loss function topology with cyclical learning rates. arXiv:arXiv:1702.04283

[15] Leslie N. Smith and Nicholay Topin. 2017. Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates. arXiv:arXiv:1708.07120

[16] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. 2017. The Marginal Value of Adaptive Gradient Methods in Machine Learning. In *NIPS*.