Translating Chemical Skeletal Formula Images into InChI Annotations

Lainey Wang
lyw0015@stanford.edu

1. Introduction

In our technology-forward world, sometimes the best and easiest tools are still pen and paper. The skeletal formula, a two dimensional representation of a molecule based on its bonding and geometry, has become ubiquitous in organic chemistry, because of its simplicity and ease to use. One major drawback of the skeletal formula, however, is that while incredibly efficient on pen and paper, it does not translate well to online search. The IUPAC International Chemical Identifier, or InChI, is a textual identifier for chemical substances, designed to provide a standard way to encode molecular information and facilitate easy search for such information in databases and on the web. Recent publications of skeletal formulae are annotated with InChI, but there are decades of scanned documents that can't be automatically searched for specific chemical depictions.

Our goal is to utilize machine learning to translate skeletal formulae images to their InChI counterparts. The input to our algorithm is a skeletal formula image. We then use various LSTM models to output a predicted InChI string annotation.

This project is taken from the Kaggle competition: Bristol-Myers Squibb – Molecular Translation [1]. I shared this project with CS231N. For 229, I focused on implementing a barebones LSTM, and added mechanisms like image transforms, and attention on top of that to analyze their effect. For 231N, I chose to start with the end product of the 229 project (LSTM with attention) and implement a transformer to get the best score I could.

2. Related Work

By far the most common approach to image captioning is the encoder-decoder architecture introduced by Xu et al. [2]. Encoders, which are usually pretrained CNNs like ResNet (He et al. [3]), are used as a mechanism to extract features from images, and the decoders predict words according to the features extracted by the encoder networks. The addition of attention to the LSTM decoder in Xu et al. allowed it to focus on specific parts of the image to generate more accurate captions, reaching a new state-of-the-art performance. In our experiments, we have a baseline LSTM, and gradually build up to a model much like the one described there. Vaswani et al. [4], following the success of transformers in the field of NLP, used transformers in the decoder instead of LSTM, achieving a new state-of-the-art.

Specifically in the context of molecular translation, however, the literature is sparse. The first models by Park et al. [5] relied heavily upon hand-written rules to extract features and identify chemical groups. Such methods required significant domain expertise, were complicated, difficult to improve upon, and inefficient. In 2017, Goh et al. [6] showed that the deep learning approach, which required minimal chemistry knowledge, outperformed the rule-based expert developed models. Deep learning with attention models were also proposed Staker et al. [7] and Rajan et al. [8], making use of a CNN and RNN with attention with moderate success.

3. Dataset and Features

From Kaggle [1], we are given 2,424,186 training images and 1,616,107 test images, with each of the training images having their own InChI label. However, given our compute limitations, the design decision to have more than 1 epoch of training, and the desire to not have each round of training take 10 days, we instead chose to random sample 200,000 images from the training set and use that as our training set instead. We split our training set into 5 folds, with each fold having 40000 images. Every fifth image from each fold is withheld as the validation set, so we have 160000 training images and 60000 validation images.

Each image has a different height and width, and 3 channels for RGB, producing an input matrix of shape $(H, W, 3)$ for each image. Because our dataset has such varying resolutions (Figure 1), we resized all images to a resolution of 224x224 pixels before inputting them to our model. Since some of the data is also vertically oriented, we chose to transpose vertically oriented images for later models, but not the baseline, to see what impacts they have on the performance of the model (Figures 2 and 3). Likewise, later models also have normalized, zero-centered images. For inference, Kaggle requires that our models run on all 1,616,107 test

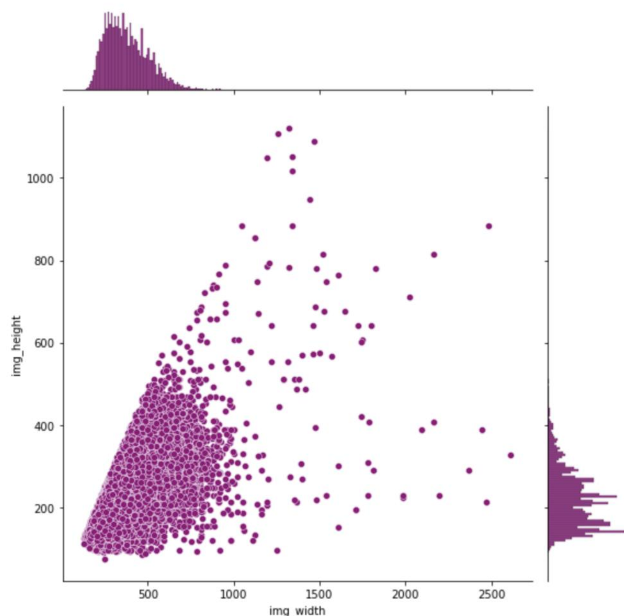


Figure 1. Distribution of image pixel heights and widths for the training set.

images. For the first baseline inference, we do not do any transforms on the test data (except the resizing). However for each subsequent model, we transpose vertically oriented images, normalize, and zero-center.

4. Methods

4.1. Baseline Encoder - LSTM Decoder

Our first baseline uses an encoder-decoder model with LSTM to predict captions.

4.1.1. ENCODER: CONVOLUTIONAL FEATURES

The role of the encoder is to take a raw input image of size (H, W, C) and to extract a set of smaller feature vectors that represent everything useful in the original image. There is no need to train an encoder from scratch, because there are already a plethora of convolutional neural networks trained to represent images (ResNet, VGG, etc.) that we can adapt for our use. For our models we chose to use ResNet-34 based on its smaller memory requirements and relative effectiveness. The final encoding produced by our ResNet-34 encoder is of size 7x7 with 512 channels.

4.1.2. DECODER: LSTM

The decoder's job is to look at the encoded image and generate a caption word by word. For our models, we use LSTM (Long Short-Term Memory). At each timestep, the LSTM receives inputs $x_t \in R^D$, the previous hidden state $h_{t-1} \in R^H$, and the previous LSTM cell state $c_{t-1} \in R^H$. The learnable parameters of the LSTM are an input-to-hidden matrix $W_x \in R^{4H \times D}$, a hidden-to-hidden matrix $W_h \in R^{4H \times H}$ and a bias vector $b \in R^{4H}$. At each timestep we first compute an activation vector $a \in R^{4H}$ as:

$$a = W_x x_t + W_h h_{t-1} + b. \tag{1}$$

We then divide this into four vectors $a_i, a_f, a_o, a_g \in R^H$ where a_i consists of the first H elements of a, a_f is the next H elements of a, etc. We then compute:

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

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

$$\begin{aligned} c_t &= f \odot c_{t-1} + i \odot g \\ h_t &= o \odot \tanh(c_t) \end{aligned} \tag{2}$$

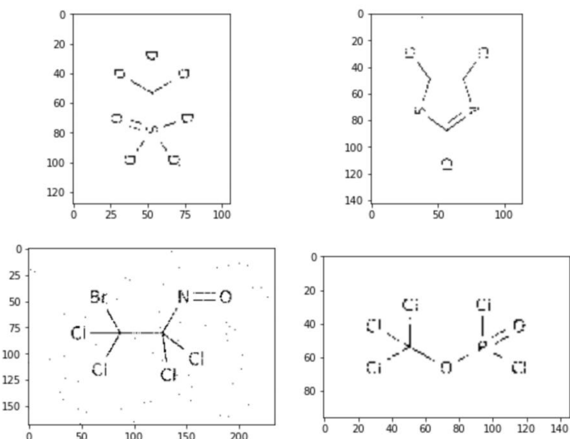


Figure 2. Training data of chemical formulas with no pre-processing. Some images are vertically oriented, which may not play well with the model.

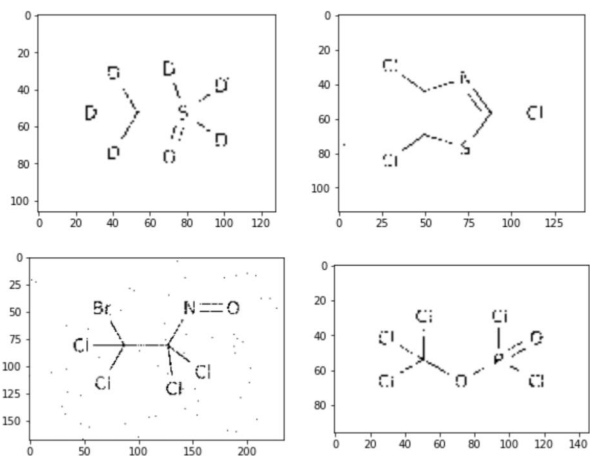


Figure 3. Training data of chemical formulas after pre-processing. Notice the vertically orientation present previously is gone, and the data is more standardized.

Figure 4 depicts this process. Intuitively, the forget gate f determines what information to forget from the old cell state c_{t-1} to the new cell state c_t (0 means forget completely, 1 means remember completely). The input gate i along with the $\tanh$ g gate decide what new information to add back to c_t. Finally, the output gate o determines which sections of the new hidden state output h_t should be affected by the the new cell state. For each timestep, where one "timestep" in our case refers to a decoded character in the sequence (such that for the InChI string "C17H28N2O", timestep 1 would decode to 'C', timestep 2 to '17', timestep 3 to 'H', etc.), our decoder runs the LSTM cell, followed by a dropout layer, and a final linear layer to build up its prediction one character at a time. The dropout layer randomly zeros some entries in the LSTM hidden output states with probability p, which prevents overfitting and forces the model to be redundant in its representation. The linear layer:

$$s = Wh_t + b \tag{3}$$

transforms this new hidden state into scores for each word in the vocabulary, the largest of which becomes our prediction.

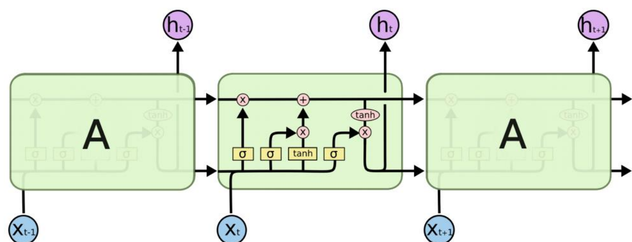


Figure 4. A series of 3 LSTM cells which learn what information to keep and erase in its memory cell. Figure from [9].

4.2. Baseline + Data Preprocessing and Transforms

On top of our baseline encoder-decoder model, we experiment with some transforms on the train and test set. In the original dataset, there are some figures that are vertically oriented (see Figure 2). In this experiment, we transpose and then vertically flip such images, resulting in all horizontal images. We also normalize and zero center our images, which should result in our model being less sensitive to changes in weights and easier optimization.

4.3. Baseline + Transforms + Attention

For our final model, we add attention to our LSTM decoder. Attention is a mechanism that allows our decoder to focus on the part of the image most relevant to the character it will next predict. In a simple encoder-decoder architecture, the decoder is supposed to start making predictions by looking only at the final output of the encoder step which has condensed information. On the other hand, attention based architecture attends every hidden state from each encoder node at every time step and then makes predictions after deciding which one is more informative.

We use soft attention as defined by Xu et al. [1]

$$E_{p(s_t|a)}[\hat{z}_t] = \sum_{i=1}^{L} \alpha_{t,i} a_i \tag{4}$$

where a is the encoder output / input hidden states, α represents the attention scores, L is the length of the encoding, and s_t represents the location where the model decides to focus attention when generating the t-th word. To calculate attention scores α, we have two separate linear layers which transform both the encoded image and the hidden state output from the decoder to the same dimension (the attention dimension). They are then added and ReLU activated. A third linear layer flattens this result and applies softmax, giving us an attention score for each pixel.

$$\text{ReLU}(x) = \max(x, 0) \tag{5}$$

$$\text{Softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{D} e^{x_j}} \tag{6}$$

$$\alpha = \text{Softmax}(\text{ReLU}((W_1 x + b_1) + (W_2 h + b_2))) \tag{7}$$

We compute the weights and attention-weighted encoding at each timestep with the Attention network. In section 4.2.1 of the paper, Xu et al. [1] recommend scaling the attention-weighted encoding by a gating constant β, which is produced by putting the previous hidden state through a linear layer and passing that through a sigmoid function. The authors state that this helps the Attention network put more emphasis on the objects in the image, so we include this step in our implementation.

$$\beta = \sigma(f_\beta(h_{t-1})) \tag{8}$$

5. Results

5.1. Quantitative Metrics

We have two primary metrics, Levenshtein Distance and cross entropy loss.

5.1.1. Levenshtein Distance

Levenshtein distance or score (LD) is the official Kaggle competition metric, and the primary measure we use to evaluate our model. The LD between two strings is the number of deletions, insertions, or substitutions required to transform the source string into target string. Thus, the higher the Levenshtein distance between a model's prediction and the label, the worse the model has performed. For example, the Levenshtein distance between "kitten" and "sitting" is 3 since at a minimum, 3 edits are required to change one into the other. More concretely,

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

where a, b are strings, and i, j are their respective character indices.

5.1.2. Cross Entropy Loss

To train our model, we use cross-entropy loss.

$$L = -\log\left(\frac{\exp(x_i)}{\sum_j \exp(x_i)}\right) \tag{9}$$

We do not use Levenshtein distance, because Levenshtein distance is undifferentiable.

5.2. Design Decisions

To choose our batch size and learning rate, we conducted experiments with 3 sets of batch sizes: 64, 32, 16, and scaled the learning rates appropriately, because the smaller our batch size the more gradient updates we have. Table 1 summarizes the hyperparameters we tried and their Levenshtein scores on the validation set. The batch size 64 model

Table 1. Hyperparameters tested on the Baseline LSTM include batch size, learning rates for the encoder and decoder, and their resulting Levenshtein Distance on the validation set.

Batch Size	Learning Rate	LD
64	4e-4, 1.6e-3	45.7463.
32	2e-4, 8e-4	47.0123.
16	1e-4, 4e-4	55.4766.

performed the best, with a LD of 45.7463, and narrowly beat the batch 32 model. Surprisingly, we found that the batch size 16 model performed the worst, despite having the most gradient updates and taking more than twice as long to train as the batch size 64 model. With such a small training set (relative to the whole set), however, these results are far from conclusive. We determined that batch size 64 and 32 models seem to be equivalent, and 16 is not ideal, which we attribute to the stochasticity of the estimated gradient as shown by the high back-and-forth oscillations of the training loss in Figure 5. We chose to not use cross validation, because the compute demand would have been too high. We did, however, save the best performing model on the validation fold across each epoch, so even if we begin to overfit in training, we will still only keep the best performing validation model. For our remaining models, we continued with the batch size of 64 and learning rates of 4e-4, 1.6e-3.

5.3. Performance Comparison

For each model, we trained for 5 epochs on a total training set size of 200,000 (160,000 training, 40,000 validation)

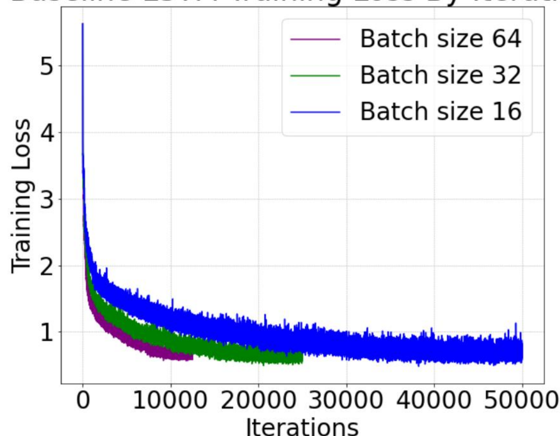


Figure 5. Cross Entropy Loss of Baseline LSTM models with different hyperparameters

and evaluated their Levenshtein score on the full test set of 1,616,107 images. For the LSTM + Attention model we also trained for 1 epoch on the whole training set (1,939,349 training, 484,837 validation), and evaluated that performance as well (Table 2).

Table 2. Levenshtein distance (LD) of each model, with different epoch and training set sizes (N) on the test set of 1,616,107 images.

Model	Epochs	N	LD
LSTM Baseline	5	200,000	63.70.
LSTM + Transforms	5	200,000	60.12.
LSTM + Attention	5	200,000	31.26.
LSTM + Attention	1	2,424,186	21.44.

5.4. Discussion

As expected, our third model, LSTM with attention, performed the best with a test score of 31.26, even with limited training. The attention aspect proved to be critical to the prediction of longer InChI sequences, as shown by Figure 6, where the Attention model was able to more effectively predict InChI strings when their length was 100-250 characters long. This finding is expected, because non-attention models should have more trouble with longer sequences. LSTMs must encode the information from all previous characters in a single vector of fixed dimension, so as the number of characters grows, the encoding becomes more lossy. Attention allows the model to attend to all hidden states, rather than just the immediate output. The visualized effect of attention can be seen in Figure 7. Furthermore, training the same attention model on the whole dataset for one epoch was able to drop our score further 10 points. We hypothesize that training this model for even more epochs would have dropped the score even further, because the model at present still struggles with long sequences (Figure 8). Surprisingly,

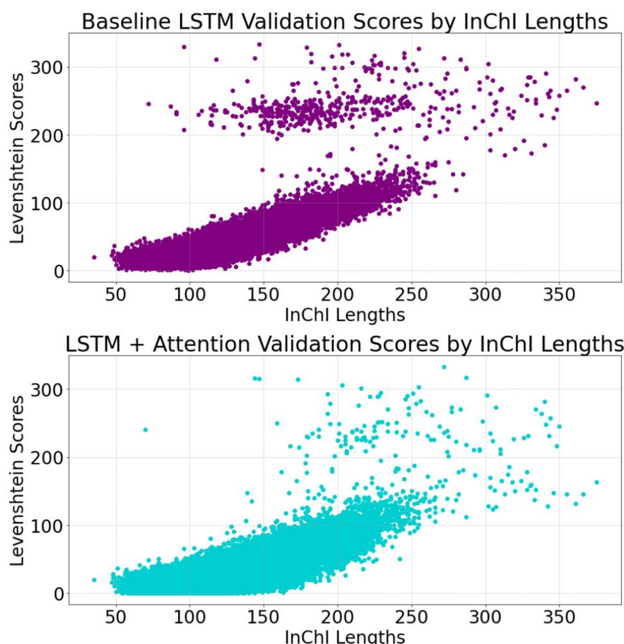


Figure 6. Levenshtein scores sorted by the length of the correct InChI string. Non-attention models have trouble with longer input sequences, but our attention model does better, removing that "cloud" of high Levenshtein scores with lengths of 100-250.

our second model, where we transformed and normalized both the test and training sets, only had a very slight improvement from the baseline (63.70 to 60.12). We suspect this is for two reasons. One, there are few vertically oriented images relative to the rest of the test set. So because the Levenshtein score is calculated as the mean LD of all test set images, the small benefits gained with the transforms wouldn't translate to a large difference in scores. Two, although normalization makes our models easier to optimize, without the attention module, the expressive power of our baseline model itself is low, so the ease of optimization doesn't make a difference.

For the models trained with $N = 200000$ images, it is likely that we overfit the dataset. From the training and validation to the much larger test set, our Levenshtein score increased from 45.7463 to 63.70. However, for the Attention model with $N = 2,424,186$, it is highly unlikely that we have overfit the training set. In fact, we have probably underfit, and would benefit significantly from increased training.

6. Conclusion

In this report, we experimented with 3 different models, LSTM, LSTM + Transforms, and LSTM with Attention in the under-explored task of captioning chemical images. Our baseline LSTM encoder-decoder model had mediocre performance, with a Levenshtein score of 63.70. Surprisingly, our second model, with the addition of training and test set transforms and normalization did not perform as well as we hypothesized. We assume this is because there are few vertically oriented images relative to the rest of the set, and because normalization only makes our models easier to optimize, but does not increase the low expressive power of our baseline. As expected, our third and final model, LSTM with attention, performed the best. Attention was crucial for more accurate predictions of longer InChI sequences, and was able to improve the Levenshtein score to 31.26, even with limited training. The same model trained for one epoch on the full training set achieved a score of 21.44.

One huge limitation of our work is the lack of compute power to adequately train all models. If we were to train all models for 5 epochs on the whole training set of 2,424,186 images, we imagine our Levenshtein scores, at least for the attention model, would drop.

For future work, we would want to experiment with longer training times and more epochs. Running for many more epochs and using the entire training set is needed to determine what the true performance capabilities of these models are, especially for the Attention model. It could also be interesting to try larger, or different CNN models, like Resnet-50 and 101, or VGG instead as the encoder.

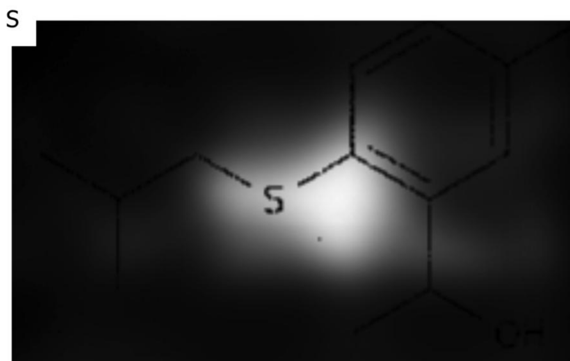
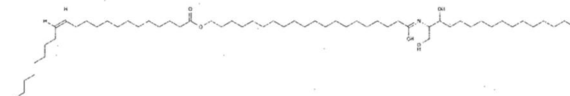


Figure 7. Visualization of attention scores. Notice, when we are captioning the character 'S', the model's attention is focused on that area of the image. Code from [10].

label: 'InChI=1S/C59H115NO5/c1-3-5-7-9-11-13-15-17-18-19-20-23-26-29-33-37-41-45-49-53-59(64)65-54-50-46-42-38-34-30-27-24-21-22-25-28-32-36-40-44-48-52-58(63)60-56(55-61)57(62)51-47-43-39-35-31-16-14-12-10-8-6-4-2/h17-18,56-57,61-62H,3-16,19-55H2,1-2H3,(H,60,63)/b18-17-'



prediction: 'InChI=1S/C64H123NO5/c1-3-5-7-9-11-13-15-17-19-21-23-25-28-31-35-39-43-47-51-55-59-61-66-69(77)67-63-59-55-51-47-43-39-35-31-27-25-23-23-25-27-29-31-33-35-37-39-43-47-51-55-61-61-65-66-69(76)67-61-60-56-52-48-44-38-16-14-12-10-8-6-4-2/h18,20,60,61,67-66,68-69H,3-18,20,23-25,27-29,30-32,34-53H2,1-2H3,(H,73,75)/b13-11-,18-17-,22-21-,58-56+'

Figure 8. A example of an incorrect caption from the Attention model (LD = 155). Our model still struggles with captions that are too long.

7. Contributions

Lainey looked at the existing Kaggle projects that performed well on the task and tried to figure out how they worked, and did background research relevant to those topics. Lainey wrote all of the LSTM model, and trained them on the Kaggle dataset. Lainey wrote this report and consolidated figures and plots to analyze the results of the model.

8. References

[1] *Bristol-Myers Squibb – Molecular Translation*. Kaggle. (n.d.). https://www.kaggle.com/c/bms-molecular-translation/data.

[2] Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., ... & Bengio, Y. (2015, June). Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning* (pp. 2048-2057). PMLR.

[3] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

[4] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. *arXiv preprint arXiv:1706.03762*.

[5] Park, J., Saitou, K., & Rosania, G. (2010, August). Image-based automated chemical database annotation with ensemble of machine-vision classifiers. In *2010 IEEE International Conference on Automation Science and Engineering* (pp. 168-173). IEEE.

[6] Goh, G. B., Siegel, C., Vishnu, A., Hodas, N. O., & Baker, N. (2017). Chemception: a deep neural network with minimal chemistry knowledge matches the performance of expert-developed QSAR/QSPR models. *arXiv preprint arXiv:1706.06689*.

[7] Staker, J., Marshall, K., Abel, R., & McQuaw, C. M. (2019). Molecular structure extraction from documents using deep learning. *Journal of chemical information and modeling, 59*(3), 1017-1029.

[8] Rajan, K., Zielesny, A., & Steinbeck, C. (2020). DECIMER: towards deep learning for chemical image recognition. *Journal of Cheminformatics, 12*(1), 1-9.

[9] Chauhan, M. (2021, February 25). *A Simple Overview of RNN, LSTM and Attention Mechanism*. Medium. https://medium.com/swlh/a-simple-overview-of-rnn-lstm-and-attention-mechanism-9e844763d07b.

[10] Sgrvinod. (n.d.). *sgrvinod/a-PyTorch-Tutorial-to-Image-Captioning*. GitHub. https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Image-Captioning.

[11] kaushal2896. (2021, April 27). *BMS MT: Show, Attend, and Tell PyTorch Baseline*. Kaggle. https://www.kaggle.com/kaushal2896/bms-mt-show-attend-and-tell-pytorch-baseline.

[12] Pasewark. (2021, March 6). *Pytorch ResNet+LSTM with attention*. Kaggle. https://www.kaggle.com/pasewark/pytorch-resnet-lstm-with-attention.

[13] Yasufuminakama. (2021, March 12). *InChI / Resnet + LSTM with attention / starter*. Kaggle. https://www.kaggle.com/yasufuminakama/inchi-resnet-lstm-with-attention-starter.