Recognizing Image-Recipe Pairs using Cross-Modal Learning

Geoffrey Horowitz
Georgia Institute of Technology
ghorowitz3@gatech.edu

Justin Havas
Georgia Institute of Technology
jhavas3@gatech.edu

Abstract

This paper investigates the use of deep learning models for recipe generation from image inputs. We investigated the Recipe1M+ model architecture presented in [6], looking at performance over a smaller dataset, investigated the effects of various introduced parameters (training mismatch rate), and identified methods of potential architecture improvement (transformers, modified LSTMs). Ultimately, we found that insufficient data size and training time resulted in poorer performance relative to the benchmark, however the investigated alternative architectures appear promising avenues for further research.

1. Introduction/Background/Motivation

Food is fundamental for human life and quality. Technology has enabled us to share food around the world for everyone's benefit. In order to recreate their favorite meals, humans rely on manually curated recipes and step-by-step instructions online. However, what if we wanted to know the possible ingredients or recipe for a food we ate at a restaurant? Knowing exactly what we are eating at any moment could be valuable for maintaining a healthy lifestyle. Therefore, this paper investigates and reproduces a deep learning system that helps solve this problem by automatically returning what the most likely recipe and ingredients are for an image of a dish or drink.

The current state-of-the-art [6] uses a dual CNN and LSTM architecture to learn cross-modal data (both image and textual info). We aim to reproduce this architecture on a limited dataset and investigate the ability of our model to generate similar results. The CNN is pre-trained on a ResNet50 [3] architecture with a primary approach of freezing the model and training the last fully-connected layer. Two LSTMs are used; a bidirectional LSTM to learn the ingredient list (used since order doesn't matter) and a unidirectional LSTM to learn the recipe instructions. Lastly, the output from the CNN and LSTMs are then combined into a joint embedding space for loss function development. This approach is common in much of the literature [2] [1] [5] [7]. Additional details are discussed throughout this paper for comparison.

One of the main reasons this architecture became the state-of-the-art is because it was trained on an immense dataset curated by querying recipe websites as well as image search engines. In total, the dataset consists of over 13 million images and 1 million recipes, which results in an average of 13 images per recipe.

A limitation of the state-of-the-art system is that it relies on so much data that it requires long training sessions of over 700 epochs. This requires many resources to run and train in order to produce outstanding results. We were interested if we could produce similar state-of-the-art results without devoting as many resources to training. Using slightly different architectures to improve the resource bottleneck will increase the ability to extend the cross-modal network. These reductions will decrease our carbon footprint as a research community and help promote less complex models for solving deep learning problems.

We used a specific subset of the Recipe 1M+ dataset [6] for this project. This subset contained approximately over 51,000 image/recipe pairs. We randomly selected 60% of this data (roughly 30,000 image/recipe pairs) for training, 20% for validation (roughly 10,000 image/recipe pairs), and the last 20% for testing. Each image is stored as a jpeg and pre-processed through transforms for consistency during training and validation. Recipes are made up of both ingredients and instructions which are stored as texts and encoded for training and validation.

2. Approach

2.1. Our Approach

The first task was to approximate the architecture presented in [6] in order to test how well it would perform with less resources. As discussed above and visualized in Figure 1, we utilized a ResNet50 CNN [3] pre-trained on the ImageNet dataset. Using pre-trained weights from ImageNet was important so that our system already contained good feature maps that would be useful for transfer learning. Unlike the original architecture which started with a

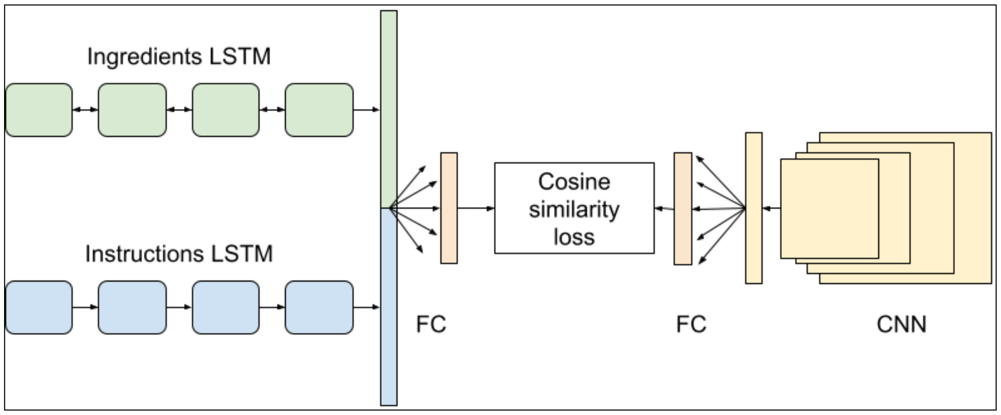


Figure 1: Architecture of our baseline cross-modal deep learning network. The convolutional neural network (CNN) output is fed into a fully-connected layer (FC) that outputs an image embedding. The output of the bi-directional ingredients LSTM and unidirectional instructions LSTM is concatenated and passed through a fully-connected layer to produce the recipe embedding. Cosine similarity loss is then computed between image and recipe embeddings.

frozen CNN and eventually unfroze it, we decided to freeze the parameters of this CNN throughout the entire training period in order to focus our limited resource usage on optimizing the recipe pipeline. The last fully-connected layer in this CNN was replaced with a new fully-connected layer that would help output a low dimensional embedding for the image.

The second half of our architecture consisted of two LSTMs, one bidirectional LSTM for the instructions and the other standard unidirectional LSTM for the ingredients. Together these instructions and ingredients are concatenated sent through a final fully-connected layer in order to create our recipe embedding.

After creating the architecture in Figure 1, we experimented with several different approaches discussed further in Section 3 that we expected could increase performance in the low resource setting.

On the LSTM side, we looked at a basic pytorch LSTM implementation, pytorch LSTMs with sequence packing (a commonly used technique (here and here) to help train the model faster using fewer computations), and tried to replace the LSTMs entirely with transformer encoders [8]. Of note, the transformer used was a custom built transformer used for CS7643 Assignment 4, modified for purposes in this project. We also looked at the effects of changing the loss metric to include additional feedback terms, modifying the mismatched data rate used in training, and investigated the ultimate effects of similarity outputs from our model.

To compare results to the original paper, we reused a common similarity metric called median rank (the ranking of model results compared to a random sample of 1000 other potential results), although we also present training and validation model loss metrics as we believe those give insight into how well the model was able to learn the underlying data (relative to compared models in the experiments).

Experiments were run using a Google Cloud Platform (GCP) virtual machine (VM) that was configured with at least 4-8 CPU and 25-50 GiB of RAM. The VM also used one NVIDIA Tesla K80 GPU at runtime.

2.2. Problems

We anticipated that downloading and setting up the dataset within our Google Cloud Platform VMs would take a significant amount of time. Because of this, we dedicated multiple days to ensure our experimentation setup was optimal before beginning training.

One unforeseen problem that occurred was that the data loading mechanism provided by the Recipe1M+ dataset was slow. This was because loading the data was reliant on the CPU used for the VM. Relying on a single CPU caused our training to be so slow that it took hours to run a single epoch. We came to the realization that we needed to upgrade our VMs to have more CPU and RAM in order to load in data faster. After doing this, we were able to improve our average epoch time to around 6 minutes, which greatly improved our ability to train and test the model.

2.3. Code Repository and Changes Made

Our code repository for this project can be found on github. While implementing, we referenced the Recipe1M+ code repository. We reused the data_loader.py file in order to load in their dataset with slight modifications for passing in a custom mismatch ratio, as well as which indexes to select random mismatches from. This was necessary since the original data_loader.py file assumed that training, validation, and testing data came from separate files, but our code

Method	Training Loss	Validation Loss	Training MedR	Validation MedR
random ranking	N/A	N/A	500	500
[6] baseline	N/A	N/A	N/A	7.2
our baseline	0.2545	0.2763	227.1	102.45
our baseline with 30 epochs	0.2262	0.2947	191.7	83.45
80% mismatch	0.1782	0.8218	395.45	93.95
20% mismatch	0.1826	0.0002	409.05	383.85
Base LSTM	.2984	.2465	295.6	159.5
Custom LSTM	.4486	.1644	529.1	499.2
Transformer	.4563	.4010	499.6	491.4
.98/.01/.01 similarity/CNN/LSTM weighting	.3195	.3438	228.4	91.35
.8/.1/.1 similarity/CNN/LSTM weighting	.7365	.994	159.9	58.8

Table 1: Results across all experiments performed. The training and validation loss correspond to the final cosine similarity loss at the last epoch. MedR is the median rank of the results across a random subset of 1000 samples. N/A is added in locations where data was not provided in the original paper.

used the original testing data and subsampled out the training, validation, and testing data from it. Additionally, we used the LSTM in the Recipe1M+ code repository in order to obtain a baseline parameter for our model (ie, given the data and epoch limitations) and in non-LSTM/transformer experiments in Section 3.3 to isolate the effects of the investigated parameter and maintain comparability to the underlying model.

3. Experiments, Results, and Analysis

3.1. Baseline

We wanted to compare our results to those obtained in [6]. As such, throughout our experiments we calculated the median rank (MedR) results using the same process as [6][4][9], which looks at the (average) median rank of a subset of randomly selected 1,000 recipe-image pairs compared to the results from the trained model. We aimed to compare our MedR results to those in the paper. We also used training loss as a metric to analyze training and validation results across experiments and to provide a secondary metric to observe how well the model was learning.

3.2. Hyperparameters

For all experiments, we fixed non-investigated parameters in order to ascertain the effects of only the varying parameters. Specifically, our model used a batch size of 256, learning rate of 1e-4, 10 epochs for experiments, training data subset of 60%, validation subset of 20%, and test subset of 20%, as well as a 50% mismatch ratio for training. Our Deep Learning framework was PyTorch.

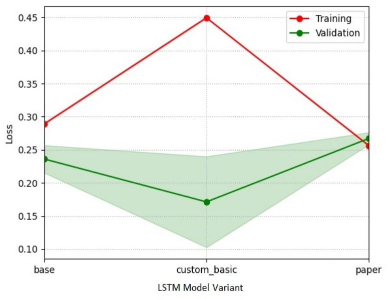
We chose these hyper-parameters through mostly trial-and-error. 10 epochs was chosen despite it being relatively low compared to the epochs used in [6] (700+) because it was necessary in order to conduct these experiments in the allotted time, and appeared to show some initial behavior even if the models would have benefited from higher epoch values.

3.3. LSTM Variants

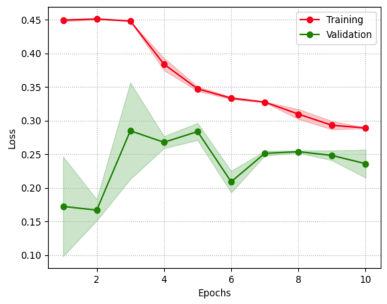
One experiment revolved around the implementation of the LSTMs in the architecture. Based on the description in [6], LSTMs were used to obtain the feature representations for the recipe and ingredients list. Note that while a bidirectional LSTM was used for the ingredients and a unidirectional LSTM was used for the recipe, for simplicity in experiments we changed both LSTMs in unison as discussed presently, though maintained the bidirectional/unidirectional underlying nature. 3 runs of each experiment were used in order to account for randomness in the data, with the mean and standard deviation of the results calculated accordingly.

First we implemented a “raw” LSTM (“base LSTM”), only using pytorch’s torch.nn.LSTM module. Secondly, based on research of common LSTM implementations (here and here), packing the embedded inputs is a technique that is used to help the model learn faster with fewer overall computations. Thus, we implemented a base LSTM but wrapped it by packing the inputs and unpacking the outputs (“custom LSTM”). Finally, in order to fully compare the results of these two architectural changes, we compared the results to the LSTM configuration implemented in [6] (“paper LSTM”).

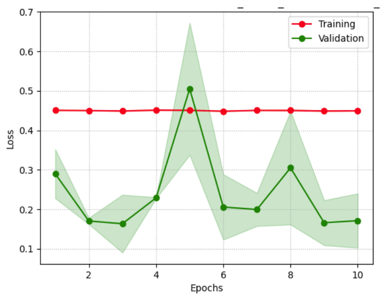
As seen in Table 1 and Figure 2a, the base LSTM actually performed admirably compared to the paper’s LSTM (”our baseline” results). As expected, it performed slightly worse, but this is likely the result of the additional sorting/packing used in the paper’s model. Confusingly, the



(a) Complexity curve showing the losses for the LSTM variants studied; base LSTM, custom LSTM, and Recipe1M+ LSTM.



(b) Learning curve for base LSTM variant, showing the mean loss (and standard deviation) for the training and validation models over the epoch range.



(c) Learning curve for custom LSTM variant, showing the mean loss (and standard deviation) for the training and validation models over the epoch range.

Figure 2: LSTM Experimental Graphs

base LSTM's validation loss and MedR are both lower than their training counterparts - unexpected behavior from any model. One possible reason for this could be that the validation set is significantly smaller than the training set, and so more randomness is contained in the validation set. As such, the cosine similarity of the validation set samples might generally be closer for all samples in the underlying set. Given more training time, we might expect this difference to become less pronounced; a fact supported by Figure 2b, where the standard deviation in the validation curve is much higher, and the two curves are getting closer together as training proceeds.

However, the custom LSTM performed very poorly. As seen in Table 1 and Figure 2c, the custom LSTM wasn't even able to learn the underlying data, with no observed change in the training loss and a median score roughly in the middle of the data range (equivalent to random ranking). Ultimately, it's vexing why the custom LSTM performed as poorly as it did, given that it had an input packing component, which should otherwise help the model learn faster. It's possible that our custom LSTM was not treating the bidirectional LSTM unpacking correctly. If this were the case, the ingredients model would have a difficult time learning and may produce random results as those seen here. Given that these additions should only add to the model's performance, it would be interesting to look into possible architectural flaws further to see if the paper LSTM results could be matched.

3.4. Transformer

Another experiment looked at removing the LSTM altogether and replacing it with a transformer [8]. As discussed prior, we wrote this transformer ourselves (though for a different assignment in this course). Of note, our analysis indicated that we only needed the encoder portion of the transformer, not the decoder, since this was being used to encode

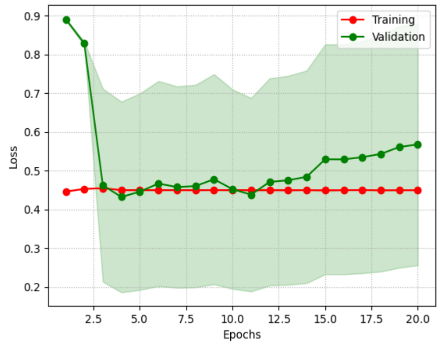
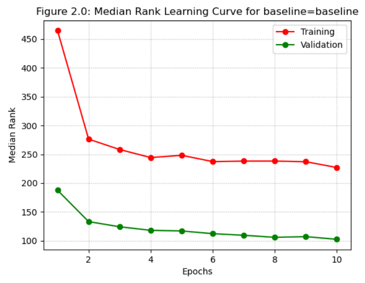


Figure 3: Learning curve for the Transformer, showing the mean loss (and standard deviation) for the training and validation models over the epoch range

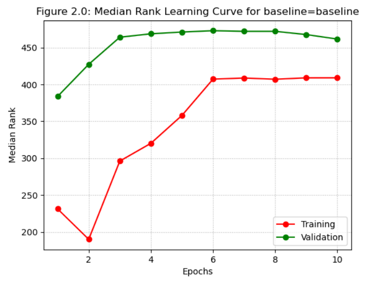
the recipe & ingredients into the joint feature space and the outputs were generated from that joint embedding.

Additionally, we had to consider the implications of the bidirectional LSTM initially implemented. The purpose of the bidirectional LSTM was to encode the ingredient information without an assumption of linearity (the ingredient list is the same in any order). However, the recipe list is order dependent. As such, we appended a positional encoding to the recipe list, however since the ingredient information was inherently unordered, we did not apply a positional embedding and we believe this properly accounted for the goals of the bidirectional LSTM. Again, 3 runs of each experiment were used in order to account for randomness in the data, with the mean and standard deviation of the results calculated accordingly.

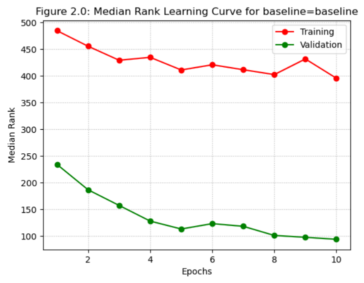
As can be seen in Figure 3, the transformer model was unable to learn in the allotted training time. The learning curve shows that the training loss didn't improve and the wide range for validation likely indicates noise from an untrained model (essentially producing random outputs); facts supported by the high loss and MedR values in Table 1. Since the model was able to learn with the correct LSTM configurations, these observations are likely due to underlying structural error in the transformer. It's possible that



(a) Median rank curve for baseline 50% mismatch experiment for the training and validation models over the epoch range.



(b) Median rank curve for 20% mismatch experiment for the training and validation models over the epoch range.



(c) Median rank curve for 80% mismatch experiment for the training and validation models over the epoch range.

Figure 4: Mismatch Experimental Graphs

one of the aforementioned assumptions are incorrect. For example, we assumed that only the encoder was needed to produce the embedded space representations (discussed above), however it may be that the output of the encoder is not sufficient to represent the underlying features in this space.

Further, given the time constraints, we were unable to tune the parameters for the transformer, which could have significantly impacted performance. It is possible that the Transformer was over- or under- parameterized for our case, resulting in an inability to learn due to insufficient time/data (over-parameterized) or inability to properly discritize the outputs (under-parameterized). Finally, while we took on the architectural challenge of modifying a custom Transformer we developed for the current use case, it may have an internal logic error, which may have been avoided using a more tested package such as the pytorch implementation.

Still, given the general benefits of transformers seen in literature [8], it would be interesting to look further into these issues and try to achieve better results.

3.5. Loss Metric

We also experimented with using an alternate loss metric to that presented in [6]. Originally, cosine similarity was used as the loss metric. This is a reasonable metric since in helps encapsulate and minimize the difference between the target image-recipe pair in the high-dimensional, joint embedding space. Inspired by the use of semantic regularization in [6], we were interested in how we might help the individual components (CNN, LSTM) learn independently of the joint embedding. As such, we implemented an alternative loss measure that still relies on cosine similarity, but also includes a weighting (α) that fuses the cross-entropy (CE) loss for the CNN and LSTM outputs, as seen in Equation 1:

$$L = \alpha_{similarity} L_{cos} + \alpha_{cnn} L_{ce}^{cnn} + \alpha_{lstm} L_{ce}^{lstm} \quad (1)$$

Comparing the results for various α weights in Table 1, its clear that increasing the CNN and LSTM weights do help provide lower MedR values. We believe this observation makes sense since it allows the CNN and LSTMs to learn independently of the joint space and thus reduces (somewhat) the long range dependency feedback from the joint embedding space, focusing on the individual results of each sub-model. This in turn likely helps the models learn faster; an important result given the limited data and training time. It is noted that while the loss values are higher than the baseline, increasing the CNN and LSTM weights produced higher starting and ending losses overall, which makes sense since instead of a continuous similarity metric, the CE losses are providing larger overall values.

While this investigation was limited due to time and compute resources, we believe a further investigation into optimal tuning of these weighting parameters could help the model perform better, even under the limited data and epoch constraints we placed on the experiment.

3.6. Training Mismatch Rate

In the original architecture presented in [6], the researchers felt that it was most useful to train with 80% mismatch rate. This meant that during training, an image that did not match a recipe was chosen with 80% probability with the goal of driving the cosine similarity to disassociate the image and recipe. While training our model, we found that this rate greatly affected the outcome of the validation loss and median rank. Therefore, we chose to experiment with this value to see if we could improve learning.

Results of our baseline model with different mismatch values can be found in Table 1.

Ultimately, we found that 80% mismatch rate performed well with respect to median rank. This can be seen in Figure 4c where both validation and training median rank continue to decrease after 10 epochs. However, the validation loss in Table 1 for 80% mismatch rate was much higher than the

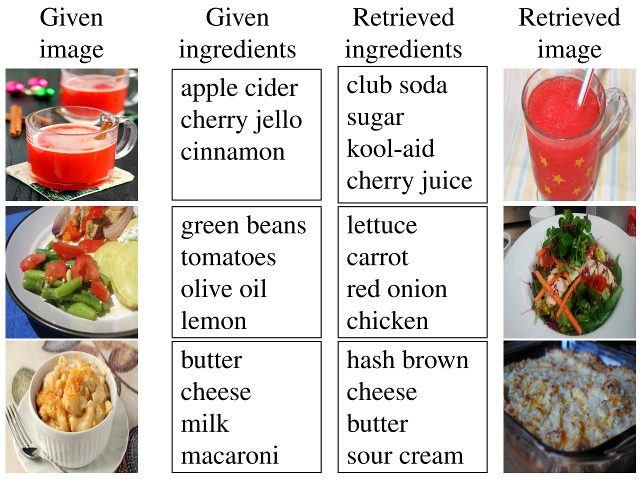


Figure 5: Examples of the most similar image/recipe pair retrieved for a given image/recipe pair. From left to right: the given image, its associated ingredients list, the retrieved ingredients, and its associated image.

training loss. One reason for this could be because that the model was observing more non-matching pairs than matching ones, making it difficult to learn the correct matches. We found that 50% mismatch rate was the best choice for overall performance in Figure 4a and Table 1 as both training/validation loss and median rank decreased overtime together. Lastly, a mismatch rate of 20% was detrimental to learning because it caused the model to overfit. This should be evident in Figure 4b as training median rank was lower than the validation median rank.

3.7. Testing Image/Recipe Pairs

Something that was not entirely clear in [6] was how they calculated the retrieved image/recipe pair from the given image/recipe pair. Thus, we developed logic that would help us visualize and test how well the system was able to match image/recipe pairs. This was calculated by taking the inner product between each image embedding and all the recipe embeddings in the validation/test dataset. The product with the highest value and its corresponding recipe would then be matched as the most likely pair.

Given this setup, in theory the most likely recipe for the given image should be the exact recipe corresponding to that image. However, in practice, this does not seem to occur since the system is trained with so many images and recipes that it usually finds one that is approximately similar to the original. This can be seen in Figure 5 where several examples of similar looking foods with also similar ingredients are displayed.

One interesting observation was that, in general, the pairings that had the highest matches across the validation/test dataset were consistently drinks. A possible explanation for this is that foods come in different shapes and sizes, however the shape of a cup is much more distinct and recognizable. This suggests that the CNN might have a greater impact on similarity matching than the LSTMs. In the end, these matches are not the exact results we would be hoping for, but it is interesting that it was able to match many types of similar looking foods and drinks.

4. Conclusion & Overall Observations

Based on the experiments conducted in Section 3, it is evident that our architecture did not perform as well as the baseline architecture used in [6]. As seen in Table 1, none of our models were able to achieve a single digit MedR. There are a few overarching factors that likely influence the inability to compete with the original paper and have been discussed throughout this report. Specifically, we used significantly less data, far fewer epochs, and had limited time to tune the model hyperparameters. Indeed, we can see an improvement on the model results in Table 1 when the number of epochs are increased from 10 to 30.

However most of our experimental investigations were able to compete with (and in some cases beat) our modified baseline results under the more stringent epoch and data constraints we used. The exceptions to this were seen in the Custom LSTM and Transformer models, where we believe the underlying issues were related to problems in our custom built architectures that were resulting in these models not learning the model space at all.

5. Future work

While our investigation did not lead to comparable results as initially desired, the investigation did provide some insight into the method used in [6]. Further, it allowed us to experiment with multi-modal data, apply various learning from the course, and even experiment with alternate architectures in the model.

As future work, we believe that given more resources (time, compute), these investigations on architectural modification and parameter tuning could lead to significant insights into the underlying workings of the original architecture and potentially provide improvements on that architecture. Specifically, the Deep learning community has been looking at transformers for improvements in many areas of deep learning [8], and we believe our cursory investigation in this area could be built upon to see if additional benefits could be gained. Further, the use of a fused loss metric provided significant value to the end results, and should be investigated further.

Student Name	Contributed Aspects	Details
Geoff Horowitz	LSTM, Implementation, Analysis	• Set up and populated initial github (for code base), box (for data store), and jupyterlab repository code & interfaces to enable collaboration and to utilize cloud services. • Generated and tuned the LSTM architecture for ingredient and recipe processing in accordance with [6]. • Implemented Transformer module to replace LSTM architecture in overall model. • Wrote grid search training script and plotting methods to help tune parameters and display results. • Conducted experiments and analysis for LSTM variants, Transformer, and Loss Metric investigations.
Justin Havas	CNN, Implementation, Analysis	• Developed the main.py function to run the code for necessary use in this project. • Modified data_loader to load appropriate data for both image and text analysis. • Generated and tuned the CNN architecture for image processing in accordance with [6]. • Implemented the image to recipe matching to gain understandings of model outputs and similarity results. • Conducted experiments and analysis for Training Mismatch and Testing Image/Recipe Pairs investigations.

Table 2: Contributions of team members in Alphabet Soup.

6. Work Division

The work division across team Alphabet Soup for this project table can be found in Table 2.

References

[1] Micael Carvalho, Rémi Cadène, David Picard, Laure Soulier, Nicolas Thome, and Matthieu Cord. Cross-modal retrieval in the cooking context: Learning semantic text-image embeddings. *CoRR*, abs/1804.11146, 2018. 1

[2] Jing-Jing Chen, Chong-Wah Ngo, Fu-Li Feng, and Tat-Seng Chua. Deep understanding of cooking procedure for cross-modal recipe retrieval. In *Proceedings of the 26th ACM International Conference on Multimedia*, MM '18, page 1020–1028, New York, NY, USA, 2018. Association for Computing Machinery. 1

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. 1

[4] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3128–3137, 2015. 3

[5] Madhu Kumari and Tajinder Singh. Food image to cooking instructions conversion through compressed embeddings using deep learning. In *2019 IEEE 35th International Conference on*

Data Engineering Workshops (ICDEW), pages 81–84, 2019. 1

[6] Javier Marín, Aritro Biswas, Ferda Ofli, Nicholas Hynes, Amaia Salvador, Yusuf Aytar, Ingmar Weber, and Antonio Torralba. Recipe1m+: A dataset for learning cross-modal embeddings for cooking recipes and food images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(1):187–203, 2021. 1, 3, 5, 6, 7

[7] Himanshu Rawlani, Jayesh Saita, Vignesh Zambre, and R.L. Priya. Deep learning based approach to suggest recipes. In *2018 International Conference on Smart City and Emerging Technology (ICSCET)*, pages 1–4, 2018. 1

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. 2, 4, 5, 6

[9] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164, 2015. 3