LensLingo - AI Caption Generation

Intro/Motivation:

Have you ever been searching for an image online, perhaps a specific photo you saw months ago, a particular image you desperately need for a project, or even just for fun? We (Nate, Robert, and Michael) have teamed up to implement an exciting idea that could greatly help with this desire: automatic captioning of images. When brainstorming for this project, we wanted to design a model that could tell you what brand of car you're looking at, but we thought, why stop there? We decided to design a model that could generate captions for any type of image, making tasks like searching online easier. Additionally, this model could be used for accessibility means, such as a "read aloud" function, which helps listen to a text message with images or for those with vision impairments. We also thought it would be pretty cool to give a computer an image and have it describe what's going on - something that, if you told us at the beginning of our computer science journey was possible, would set us into disbelief.

Data

We got the data for our project from the Flickr 8k dataset on Kaggle. This dataset contains 8,092 images, each paired with five different captions, each with varying details about the photo, describing what is happening in the image.

```
A little girl covered in paint sits before a painted rainbow with her hands in
a bowl .
A little girl is sitting in front of a large painted rainbow .
A small girl in the grass plays with fingerpaints in front of a white canvas
with a rainbow on it .
There is a girl with pigtails sitting in front of a rainbow painting .
Young girl with pigtails painting outside in the grass .
```

The images were chosen from six different groups on Flickr and don't contain well-known people or locations. They encompass a wide range of arbitrary scenarios. We will use this dataset to train our model to predict captions for future inputted images, training each combination of photo and caption to yield the most accurate results. We chose this dataset because it had multiple different captions for each photo, was not as massive as something like ImageNet (our laptops don't have the processing power for that, unfortunately), and was rated highly by the community.

Preprocessing

Before we can use our data to train our model, we have to process it. To extract the necessary features from our images, we used the VGG16 model. We load our pictures, convert them to numpy arrays, reshape them for the VGG16 model, pass them into the model, extract the features, and finally store the image in a dictionary, with the key being the image name.

```
# extract features from image
features = {}
directory = os.path.join(BASE_DIR, 'Images')

for img_name in tqdm(os.listdir(directory)):
    # load the image from file
    img_path = directory + '/' + img_name
    image = load_img(img_path, target_size=(224, 224))
    # convert image pixels to numpy array
    image = img_to_array(image)
    # reshape data for model
    image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
    # preprocess image for vgg
    image = preprocess_input(image)
    # extract features
    feature = model.predict(image, verbose=0)
    # get image ID
    image_id = img_name.split('.')[0]
    # store feature
    features[image_id] = feature
```

We stored our features in a pickle file for easier access later on.

Next, we need to map our images to the five captions for each image. To do this, we create a dictionary and iterate through the list of captions, each containing the name of the associated image. For each image, we store the list of related captions in the dictionary using the image name as the key. But we're not done there - we need to "clean" our captions so they are standardized, all lowercase, and don't contain extra spaces, punctuation, or digits. We also add a "startseq" and "endseq" tag to the start and end of each of our captions to allow the model to know precisely where each sequence starts and begins.

Next, we use a tokenizer to get every unique word by creating a list of all the captions and fitting the tokenizer to the list. From this, we get our vocabulary size, which is the length of the list of unique words plus one (an index for words not in the vocabulary). This allows us to map each word to a unique index in the vocabulary to give the model the required numerical input.

We need to split the data into train and test data. We use a 90% split, 90% of the data for training and 10% for testing. Since our dataset is highly variable with many different types of images, we need to use as much data as possible to ensure model accuracy. Our dataset is relatively large, so 10% of the data will be sufficient for testing. Additionally, we can manually verify our model by generating images and seeing if they are correct.

```python
# generates batches of data for training
def data_generator(data_keys, mapping, features, tokenizer, max_length, vocab_size, batch_size):
    # loop over images
    # X1 is image features, X2 is input sequence, y is output sequence
    X1, X2, y = list(), list(), list()
    n = 0
    while 1:
        for key in data_keys:
            n += 1
            captions = mapping[key]
            # process each caption
            for caption in captions:
                # encode the sequence
                seq = tokenizer.texts_to_sequences([caption])[0]
                # split the sequence into X, y pairs
                for i in range(1, len(seq)):
                    # split into input and output pairs
                    in_seq, out_seq = seq[:i], seq[i]
                    # pad input sequence
                    # in_seq is all the tokens up the current token, out_seq is the next token in the sequence
                    in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
                    # encode output sequence
                    out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]

                    # store the sequences
                    X1.append(features[key][0])
                    X2.append(in_seq)
                    y.append(out_seq)

            if n == batch_size:
                X1, X2, y = np.array(X1), np.array(X2), np.array(y)
                yield [X1, X2], y
                X1, X2, y = list(), list(), list()
                n = 0
```

A data generator is a tool that efficiently handles data loading and preprocessing, ensuring that only small batches of data are loaded into memory at any time. It is beneficial for our image captioning since each training instance involves both image and sequential caption data, resulting in a significant amount of data to generate. Let us explain what the above code is doing.

Processing and Encoding: The generator reads all associated captions for each image (denoted by key). The generator encodes the captions into sequences of integers with a tokenizer, padding them to a fixed length and converting the output words into one-hot encoded formats.

Creating training pairs: For each caption, the generator iterates through each word to generate training pairs of input sequences (`X2`) and the following word (`y`), where `X2` is the sequence

caption: "Man riding a red bike"

	in_seq	out_seq
i = 1	in_seq: "Man"	out_seq: "riding"
i = 2	in_seq: "Man riding"	out_seq: "a"
i = 3	in_seq: "Man riding a"	out_seq: "red"
i = 4	in_seq: "Man riding a red"	out_seq: "bike"

leading up to the current word, and y is the one-hot encoded next word.

Feature Preparation: The image features (`X1`) corresponding to each input sequence are also prepared. These features are retrieved from our dictionary (`features`).

Batch Yielding: Once enough data pairs (determined by `batch_size`) are prepared, the generator puts them into arrays and yields these arrays for training. After yielding, it resets the lists and counter, ready to prepare the next batch.

Now that our data is processed and split into training and test data, we can train our model!

Method

```
# Image feature extractor
inputs1 = Input(shape=(4096,))
fe1 = Dropout(0.4)(inputs1)
fe2 = Dense(256, activation='relu')(fe1)
fe2_repeated = RepeatVector(max_length)(fe2)

# Sequence processor
inputs2 = Input(shape=(max_length,))
se1 = Embedding(vocab_size, 256, mask_zero=True)(inputs2)
se2 = Dropout(0.4)(se1)

combined_input = Concatenate()([fe2_repeated, se2])
se3 = Bidirectional(LSTM(256, return_sequences=False))(combined_input)

# Output layer for predicting the next word directly
outputs = Dense(vocab_size, activation='softmax')(se3)
model = Model(inputs=[inputs1, inputs2], outputs=outputs)
```

We gained inspiration for our model from a code example on the Kaggle page for the Flickr 8k dataset. While the Kaggle architecture performed well, we wanted to make it better. We knew immediately that we wanted to do an encoder-decoder model, as we found they were the best architecture for NLP tasks through our research. We experimented with different types of layers, trying LSTMs, bidirectional LSTMs, and various combinations of dropout, dense, add,

etc. After our experimentation, the architecture above had the highest performance out of all our models, we used Weights and Biases to fine-tune the hyperparameters. By running “sweeps,” where different combinations of parameters are tested, we optimized our model for the best results. We tested changing dropout, learning rate, epochs, batch size, and the optimizer (Adam vs SGD). Our final parameters were as follows:

batch_size	dropout	epochs	learning_rate	optimizer
32	0.4	20	0.001	Adam

Image Feature Extractor Layers

Input layer: We start with the image input layer containing the preprocessed image features. We take in our pre-processed images, represented as a flat array of 4096 elements - the output of the VGG16 model we used to extract features from the images.

Dropout Layer: We set the dropout rate to 40% of the data points. This is used to prevent overfitting during training by randomly removing subsets of features during different iterations. By experimenting with the hyperparameters, 40% was chosen as a reasonable dropout rate. In our model, 40% yielded the best results.

Dense Layer: After the dropout, we use a dense layer with 256 units and a ReLU activation function. This layer transforms the linearized images into a non-linear higher-dimensional representation so patterns and relationships can be better captured.

RepeatVector Layer: We use the repeat vector to match the output of the dense layer max_length times. This is done to match the sequence length of the inputted captions.

Sequence Processor Layers

Input Layer: This layer accepts a sequence of integers, where each integer represents a word in our vocabulary. The vector length is capped to the max length of the captions.

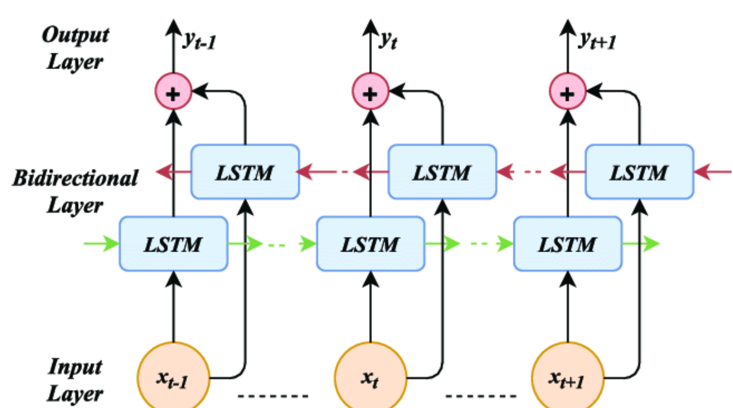
Embedded Layer: The integers are then mapped to a dense vector of size 256 using an embedding layer. This layer learns a dense representation for each word in the vocabulary. This

layer is like a lookup table that transforms word indices into continuous vectors. The embedding ignores zeros, which are used for padding the sequences.

Dropout Layer: Similar to the image features, the dropout layer is used here to help mitigate overfitting by randomly setting inputs to zero during training (again using 40%).

Concatenate Layer: This layer concatenates the RepeatVector layer's output with the dropout layer's output. It combines the image features with the textual features before further processing.

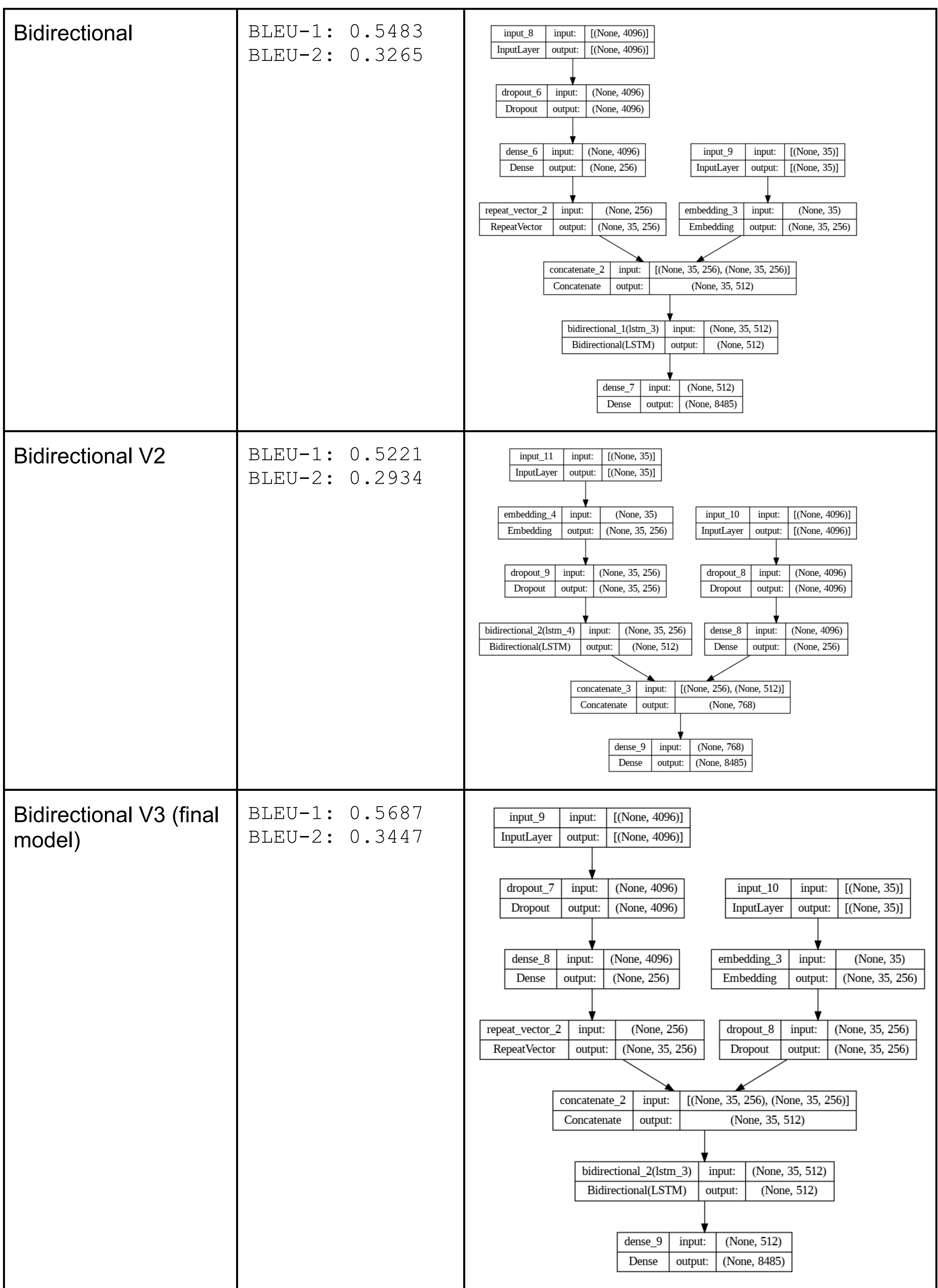
Bidirectional LSTM Layer: This layer has a Bidirectional Long Short-Term Memory unit with 256 units. This is used to process the combined input sequence and output a representation of its understanding of the sequence. The bidirectional aspect allows the model to look at the context from both directions of the input sequence. When researching our model, we found that Bidirectional LSTMs are used extensively in NLP projects. This makes a lot of sense, as they can learn and remember over long sequences, making them highly suitable for the sequential data that NLP problems contain.



Output Layer: Finally, our output layer consists of a dense layer with a softmax activation function. This predicts the probability distribution over the vocabulary for the next word in the sequence.

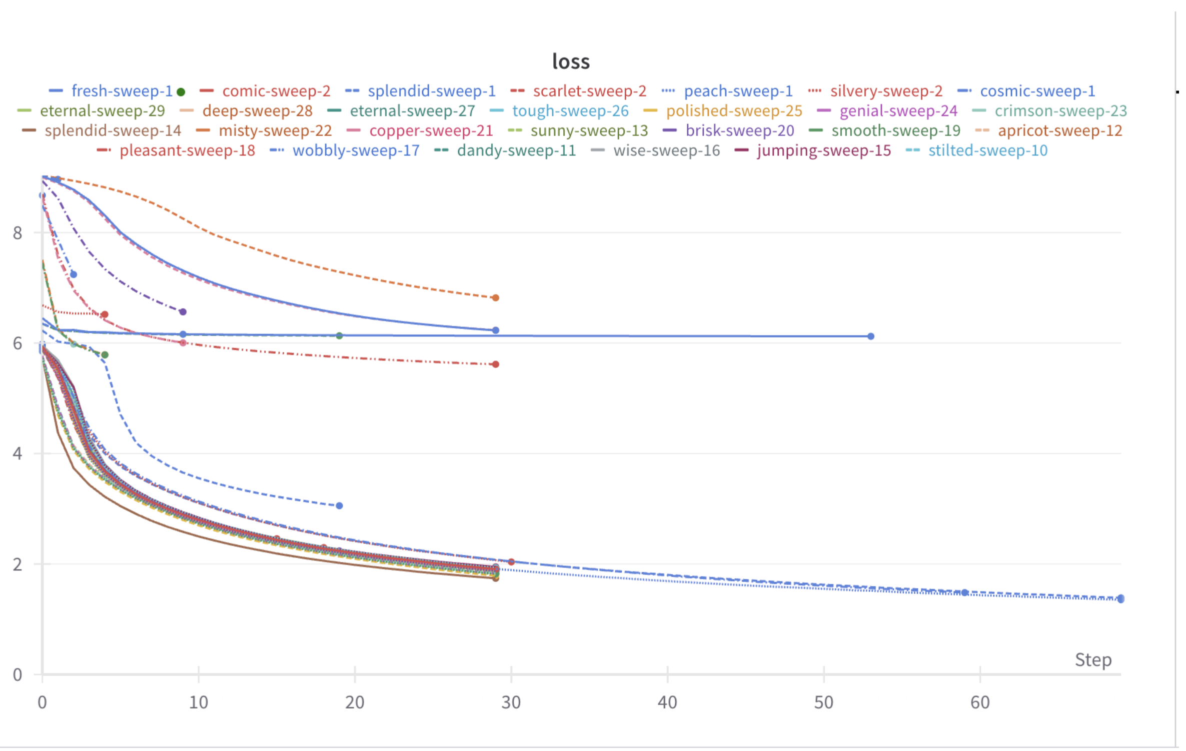
Results

Model Arch	Corpus Scores	Photo
Kaggle Inspiration	`BLEU-1: 0.5497` `BLEU-2: 0.3210`	input_3 input: [(None, 35)] InputLayer output: [(None, 35)] embedding input: (None, 35) Embedding output: (None, 35, 256) input_2 input: [(None, 4096)] InputLayer output: [(None, 4096)] dropout_1 input: (None, 35, 256) Dropout output: (None, 35, 256) dropout input: (None, 4096) Dropout output: (None, 4096) lstm input: (None, 35, 256) LSTM output: (None, 256) dense input: (None, 4096) Dense output: (None, 256) add input: [(None, 256), (None, 256)] Add output: (None, 256) dense_1 input: (None, 256) Dense output: (None, 256) dense_2 input: (None, 256) Dense output: (None, 8485)
Concat	`BLEU-1: 0.5290` `BLEU-2: 0.3010`	input_5 input: [(None, 35)] InputLayer output: [(None, 35)] embedding_1 input: (None, 35) Embedding output: (None, 35, 256) input_4 input: [(None, 4096)] InputLayer output: [(None, 4096)] dropout_3 input: (None, 35, 256) Dropout output: (None, 35, 256) dropout_2 input: (None, 4096) Dropout output: (None, 4096) lstm_1 input: (None, 35, 256) LSTM output: (None, 256) dense_3 input: (None, 4096) Dense output: (None, 256) concatenate input: [(None, 256), (None, 256)] Concatenate output: (None, 512) dense_4 input: (None, 512) Dense output: (None, 8485)

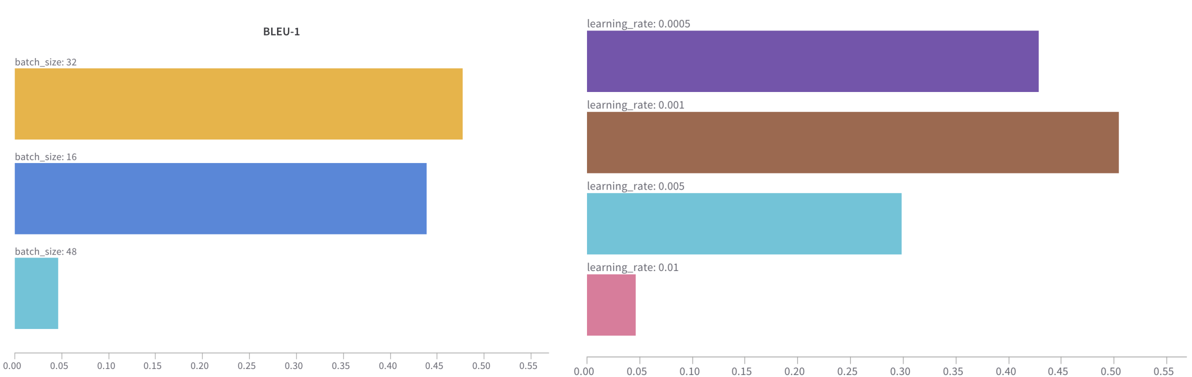


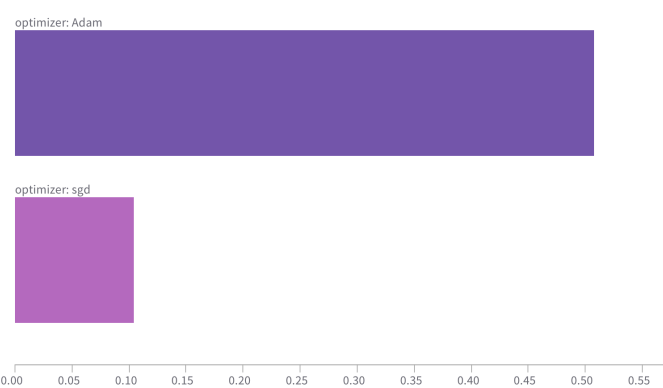
Model	Scores	Architecture
Bidirectional	BLEU-1: 0.5483 BLEU-2: 0.3265	
Bidirectional V2	BLEU-1: 0.5221 BLEU-2: 0.2934	
Bidirectional V3 (final model)	BLEU-1: 0.5687 BLEU-2: 0.3447	

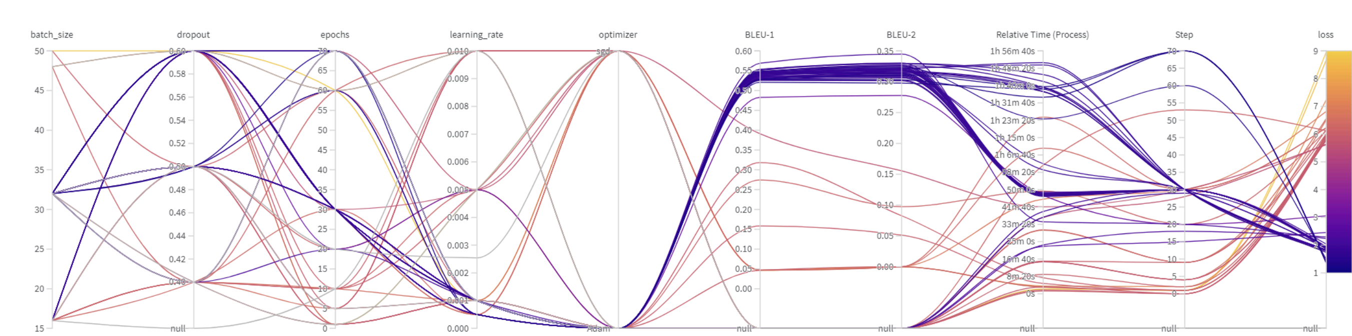
Pictured above are some of the models we tested. While we did test other architectures, mostly just tweaking some of the layers, we only included the successful models. After experimentation, we settled on the Bidirectional V3 model, as it rendered the best results.



From our step vs loss graph, we see that our loss drops dramatically between 0 and 10 steps and then slowly decreases. Although most of our sweeps were capped at 30 epochs, we ran a few sweeps for many more steps (60, 70), which resulted in a marginal decrease in loss.



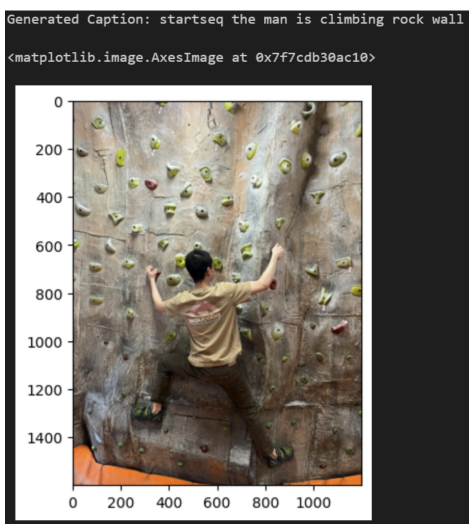


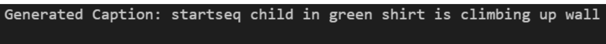


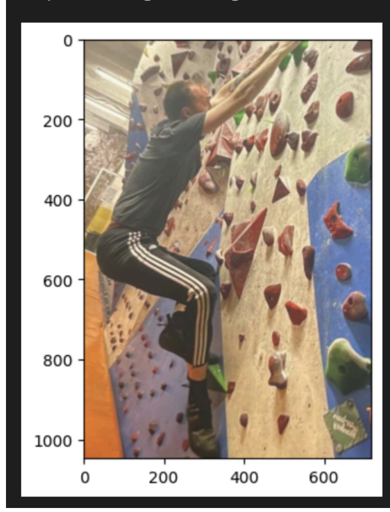
We use the BLEU score, 0 to 1, for our accuracy metric to measure how similar a given set of text is to reference text. BLEU-1 is a unigram precision score, which will compare each word token with a respective reference word token. BLEU-2 is a bigram precision score comparing sets of two-word tokens with respective sets of two reference word tokens. After researching the BLEU score, a score between 0.6 and 0.7 is the best BLEU-1 score that can reasonably be achieved. When looking at our BLEU-1 bar charts, the Adam optimizer drastically outperforms the sgd optimizer. We suspect this is because Adam has an adaptive learning rate, whereas sgd has a fixed one. This means our learning rate choices may not have been ideal for sgd to perform well. The sweeps with batch sizes of 32 and 16 outperformed the sweeps with a batch size of 48. This can likely be explained by the fact that larger batch sizes will lead to lower variance and speed up training; however, they may lead to lower accuracy. From our BLEU score vs learning rate chart, we found that a learning rate of 0.001 performed the best.

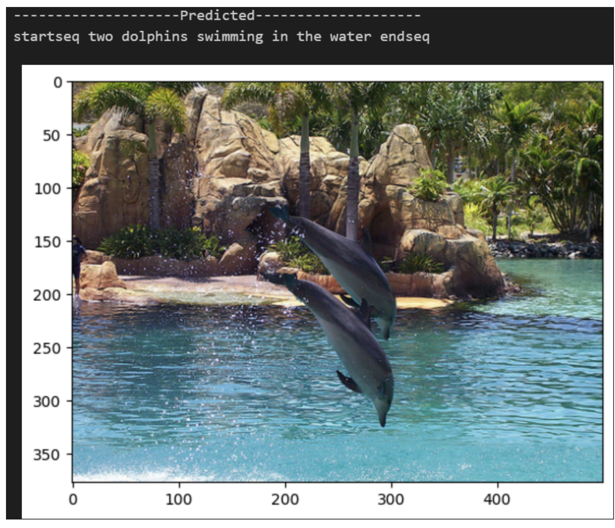
Examples

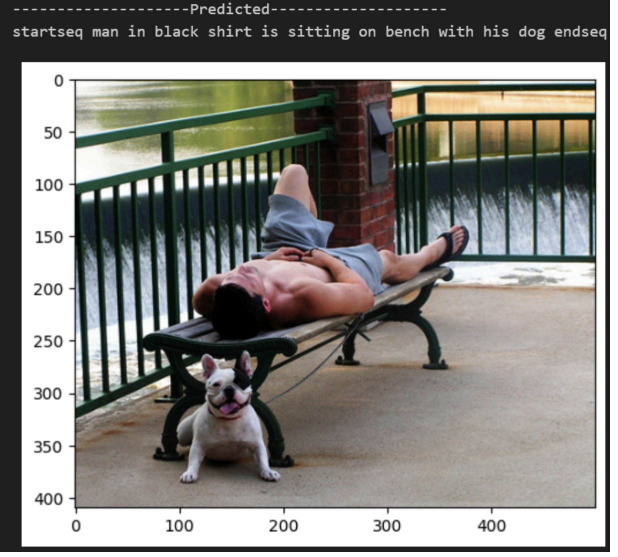
Generated Caption: startseq group of girls in red outfits perform on stage
<matplotlib.image.AxesImage at 0x7f7cdb4f4050>


Generated Caption: startseq the man is climbing rock wall
<matplotlib.image.AxesImage at 0x7f7cdb30ac10>


Generated Caption: startseq child in green shirt is climbing up wall


<matplotlib.image.AxesImage at 0x7f7cc0408850>


-------------------Predicted-------------------
startseq two dolphins swimming in the water endseq


-------------------Predicted-------------------
startseq man in black shirt is sitting on bench with his dog endseq

Conclusion

We had a great time building, testing, and improving this model. Using Weights and Biases to optimize our model was a game changer, as it let us set the computer up overnight and test many different combinations of hyperparameters. It also allowed us to visualize our data and see the best hyperparameters easily. Using an encoder-decoder model was a great choice, as they work very well for NLP problems, and our model rendered results that we were satisfied with. If we had more computational power, we would have used a larger dataset, like ImageNet, to recognize a broader range of images more accurately. Flickr8k, as the name suggests, only contains 8,000 photos, which is nothing compared to the fourteen million something in ImageNet. Another issue was needing more time and processing power to run the model with all hyperparameters using random and grid search on Weights and Biases. Our solution was to explore hyperparameter values using the Bayes optimization algorithm and to implement early stopping to terminate the sweep without exploring all hyperparameter combinations. Another issue was that saving models becomes challenging when using Weights and Biases because the models are only scoped to the Weights and Biases function. Our solution was to save the models for each Weights and Biases iteration. The project taught us a lot about optimizing our model and designing an architecture to solve the task, and we are happy with the results. Perhaps with more computational power and a larger dataset, we could create a far more accurate image captioning AI.