In the Name of God

University of Tehran

Faculty of Electrical and Computer Engineering

Neural Networks and Deep Learning

Bonus Assignment

Full Name	Mohammad Taha Majlesi
Student Number	810101504
Submission Deadline	2025/06/22

Contents

List of Figures

List of Tables

1 Question 2 - Generating Textual Descriptions for Images

1.1 Introduction

Automatic generation of textual descriptions for an image, or *image captioning*, is a task closely aligned with the core goals of computer vision. An image captioning model must not only be powerful enough to solve vision challenges like object detection but also understand the relationships between objects in the image and express them in natural language.

For this reason, this problem has always been considered a difficult one. This challenge is also significant in machine learning because the model attempts to mimic the remarkable human ability to compress salient visual information into descriptive language.

With advancements in neural network training and the emergence of large datasets, recent works have significantly improved the quality of generated captions. In this project, we will first extract image features using a convolutional neural network (CNN) and then decode these feature vectors with a recurrent neural network (RNN) to produce natural language sentences.

One of the interesting abilities of humans is to focus on important parts of an image. This is particularly important in cluttered scenes. In deep networks, interpretations are usually attempted for regions the model attends to by receiving feature vectors from the final layer of a CNN. However, this approach overlooks important information present in the early layers. Therefore, in the reference paper, a type of attention mechanism called **Additive Attention** is used to investigate which parts of the image the model focuses on more.

1.2 Data Preparation

1.2.1 Data Acquisition

The COCO (Common Objects in Context) dataset is a well-known dataset used for tasks such as object detection, image segmentation, and image captioning. This dataset was created by Microsoft and includes over 300,000 images in various formats. We use a translated version of this dataset containing 40,000 image-caption pairs. The images in the dataset have various dimensions, with widths ranging from 72 to 673 pixels and heights from 51 to 664 pixels.

1.2.2 Caption Preprocessing

We preprocess the textual descriptions using the ‘hazm‘ library. With the ‘Normalizer‘ class from this library, we normalize the texts. The normalizer function corrects spacing, replaces Arabic numerals and symbols with their Persian equivalents, removes diacritics, redundant letters, and other special characters that are not useful for text processing. We chose not to remove numbers and punctuation marks, as they are not expected to be frequent in captions, their presence is meaningful, and the large amount of training data allows the model to learn their usage.

1.2.3 Data Splitting for Training

The COCO dataset does not have a separate test set, so we must split the data ourselves. After preprocessing the captions, we divide them into three parts: training, validation, and test. Given the large number of data points, we randomly select 10,000 for training, 500 for validation, and 500 for testing, ensuring no overlap.

1.2.4 Vocabulary Creation

Tokenizer Analysis In natural language learning, a tokenizer is a module that converts a given text into a list of numbers, which represent indices in an embedding layer for the model to understand. To do this, the text must be broken down into tokens. There are several methods for this.

One approach is to consider tokens as characters or sets of bytes forming characters. This results in a small dictionary, but the semantic load of characters is low, making it difficult for the model to understand the text. Also, for each input, the number of tokens becomes very large.

Another approach is to consider a few words or a whole sentence as a single token. Due to the variety and numerous forms, the number of dictionary elements becomes very large. Since each token represents a larger portion of the input, the context window is consumed less, and longer inputs can be processed. However, the model needs to see more data to learn all these different tokens.

Modern large language models use a combination of the above methods. For example, a tokenizer might break words into prefixes and suffixes, but if it encounters a new word, it breaks it down into smaller components like characters or bytes to correctly tokenize most inputs.

Special Tokens Tokenizers usually include special tokens that are inserted into the input text for specific purposes. The model learns these tokens just like any other. Since these tokens appear in various texts, they can store meaningful information from all texts.

In this project, we use four special tokens. Two tokens, ‘¡sos¿‘ and ‘¡eos¿‘, mark the beginning and end of a sentence, respectively. This helps the model learn that texts have a start and an end, and the sentences it generates should also have a beginning and an end.

To leverage GPU parallelization, we assume a fixed length for all token sequences. Considering the addition of ‘¡sos¿‘ and ‘¡eos¿‘ to all captions, we set the maximum number of tokens based on the longest caption in the training set. If a caption in the evaluation data has more tokens, we truncate the end. If the training data is a good representation of the whole dataset, the maximum token count we derive should be close to the maximum for the entire dataset.

If a caption has fewer tokens than the maximum, we need to pad it. For this, we use the ‘¡pad¿‘ token. We add ‘¡pad¿‘ tokens after the ‘¡eos¿‘ token until the sequence reaches the desired length. The presence of padding is essential in modern NLP, as most models, especially from the Transformer family, can only process a fixed number of tokens.

The last special token is ‘¡unk¿‘, which represents unknown tokens. It’s possible that during evaluation, a token appears that was not in the vocabulary built from the training data. To enable the model to understand the meaning of these tokens to some extent, we use the ‘¡unk¿‘ token. We replace tokens in the inference time that are not in the vocabulary with ‘¡unk¿‘. To make this token meaningful, we must use it in the training

data so the model can learn it. We can do this by replacing a certain number or percentage of the least frequent tokens in the training data with ‘¡unk¿‘.

Creating the Tokenizer Class The ‘Tokenizer‘ class receives the training captions and, by analyzing the frequency of tokens, builds a vocabulary for tokens with a certain minimum frequency. Each token in the vocabulary is assigned a unique integer from 0 up to the vocabulary size. This class uses the ‘word$_t$*okenize*‘*function from the*‘*hazm*‘*library to identify toke*

```
def _create_dictionary(self, captions):
    dictionary = dict({token: i for i, token in enumerate(
    special_tokens.list_of_tokens)})

    for caption in captions:
        tokens = self.tokenize(caption)
        for token in tokens:
            self.frequencies[token] += 1
            self.total_tokens += 1

    self.frequencies = sorted(list(self.frequencies.items()), key=
    lambda x: x[1])
    freq_threshold = self.frequencies[int(self.unknown_percent * self.
    total_tokens)][1]

    for token, frequency in self.frequencies:
        if frequency <= freq_threshold:
            self.unknown_words.append((token, frequency))
            self.total_unknown_tokens += frequency
        else:
            dictionary[token] = len(dictionary)

    return dictionary
```

Listing 1: Code for creating the vocabulary dictionary.

1.2.5 Data Exploration

To display Persian text in images, we use the ‘arabic$_r$*eshaper*‘*and*‘*bidi*‘*libraries*.

The longest caption in the training data had 38 tokens. With the addition of ‘¡sos¿‘ and ‘¡eos¿‘, we set the token length to 40. In total, there were 5166 unique tokens in the training data, of which 2971 (about 42.5%) were considered ‘¡unk¿‘ due to low frequency and were removed. This might seem like a large number of tokens, but all these tokens appeared only once in the captions. Out of all tokens (including repetitions), which numbered 110,881, the number of ‘¡unk¿‘ tokens was less than 2%. The ten most frequent ‘¡unk¿‘ tokens are: ‘pink-clad‘, ‘laughing‘, ‘concept‘, ‘tow‘, ‘more‘, ‘sends‘, ‘merchant‘, ‘metro‘, ‘protein‘, ‘start‘.

Among these ten words, some, while not necessarily rare in Persian literature, are logically infrequent in image descriptions, like ‘concept‘. Others are generally rare in Persian, like ‘pink-clad‘ and ‘tow‘. For some other words, we expect them to be more frequent in images, like ‘laughing‘ and ‘sends‘, but perhaps other forms of these words are more common, such as ‘laughed‘, ‘laughs‘, ‘laughter‘, etc.

We could have prevented the removal of some words by replacing all forms with a common root, but we avoided this for a few reasons. First, the model would then only

be able to generate the root for all these words, which would reduce the quality of the generated captions. Also, as mentioned, only two percent of the tokens were marked as ‘¡unk¿‘, and the number of existing tokens is sufficient. By including some words with moderate frequency as ‘¡unk¿‘, we can help train a suitable embedding for this token.

The ten most frequent tokens, in ascending order of frequency, are: ‘man: 1254‘, ‘state: 1520‘, ‘on: 2086‘, ‘from: 2207‘, ‘that: 3114‘, ‘and: 3141‘, ‘with: 3267‘, ‘a: 6089‘, ‘in: 7524‘, ‘.: 7826‘. As can be seen, most of these tokens are stop words and are very common in Persian literature. Although these words are usually removed in tasks like sentiment analysis and classification, they are essential for text generation. Also, as we noted, we did not remove punctuation, and the period (‘.‘) is the most frequent token.

Now let’s examine a sample sentence: ”A woman in pink is laughing at a cooking station.” After adding special tokens by the tokenizer, the sentence becomes: ‘¡sos¿ a woman ¡unk¿ at a cooking station ¡unk¿ . ¡eos¿ ¡pad¿ ¡pad¿ ...‘ We can see that the words ‘pink-clad‘ and ‘laughing‘ have been converted to ‘¡unk¿‘. Of course, there are more ‘¡pad¿‘ tokens in the actual output to bring the total number of tokens to 40.

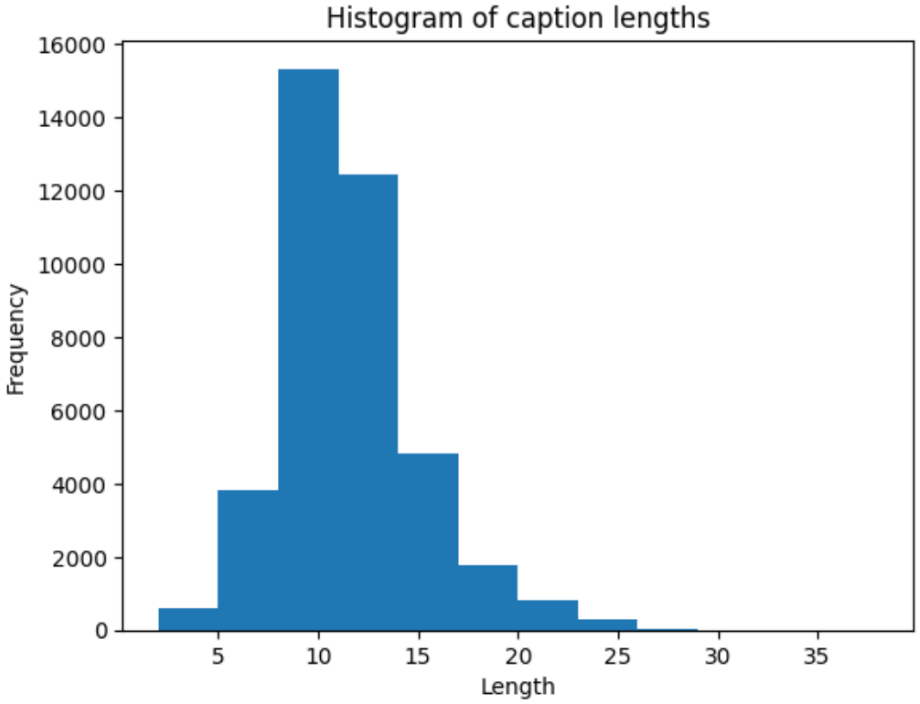

Figure 2: Frequency histogram of captions with different lengths.

From the figure above, we can see that the distribution of caption lengths is somewhat Gaussian, with most captions, excluding special tokens, having around 10 tokens. This is logical for describing an image, as we usually describe an image in a single sentence.

1.2.6 Image Preprocessing

Now we create a dataset class that, upon receiving an image index, loads the image and, after applying transformations, returns it along with the image and the list of caption numbers obtained from the tokenizer. Since the pre-trained encoder model we use was trained on ImageNet, we use the same transformations for preprocessing the images. This preprocessing resizes the images to 528 pixels using the BICUBIC method and

then, after scaling the values to the range [0, 1], normalizes them with the mean and standard deviation of the ImageNet images. Thus, the image corresponding to the caption discussed in the previous section looks as follows after preprocessing.

Figure 3: A sample image after preprocessing.

```
class COCODataset(Dataset):
    def __init__(self, image_dir, captions, tokenizer, transform=None):
        self.images_dir = image_dir
        self.transform = transform
        self.captions = captions
        self.ids = defaultdict(list)
        for img, caption in captions.items():
            self.ids[img] = tokenizer.text_to_ids(caption)
        self.tokenizer = tokenizer
        self.image_files = list(captions.keys())

    def __getitem__(self, index):
        image_file = self.image_files[index]
        image_path = os.path.join(self.images_dir, image_file)
        image = Image.open(image_path).convert('RGB')
        if self.transform:
            image = self.transform(image)
        ids = self.ids[image_file]
        return index, image, ids

    def __len__(self):
        return len(self.image_files)
```

Listing 2: PyTorch Dataset Class for COCO Captions.

1.3 Implementation of CNN+RNN Architecture with Attention Mechanism

The model used consists of an encoder, which is a convolutional network, and a decoder, which is a recurrent network with an attention mechanism. This model takes an image as input and generates a caption y, which is a sequence of length C words, where the words are chosen from a dictionary of K words.

$$y = \{y_1, \ldots, y_C\}, y_i \in \mathbb{R}^K$$

1.3.1 Encoder Implementation

We initially intended to use the EfficientNet-B7 model, but this model has about 66 million parameters and accepts images with dimensions of 600 pixels, which leads to GPU memory issues and slow training. Therefore, we used the EfficientNet-B4 model, which has about 17.5 million parameters and accepts images with dimensions of 380 pixels, but its performance is not significantly different from the previous model.

EfficientNet is a convolutional neural network that aims to create a model with high performance and computational efficiency by systematically scaling depth, width, and input resolution.

Thus, by removing the final pooling and fully connected classification layers and freezing the model's weights, the encoder part is completed. This model, upon receiving an image, produces L feature vectors, each of dimension D, pointing to a part of the image.

$$a = \{a_1, \ldots, a_L\}, a_i \in \mathbb{R}^D$$

1.3.2 Attention Mechanism Implementation

The paper describes two attention mechanisms for the model: random attention and deterministic attention. In hard random attention, the model produces an output $s_{t,i}$, which is a one-hot vector. It is one if, out of the L available locations in the features, the model's attention is drawn to the i-th part to generate the t-th word. The context vector $\hat{z}_t$ is obtained as follows:

$$p(s_{t,i} = 1 | s_{j<t}, a) = \alpha_{t,i}$$

$$\hat{z}_t = \sum_i s_{t,i} a_i$$

The paper tries to reduce a loss function defined as a lower bound for the above expression. The BLEU scores for this type of attention mechanism are slightly better than the other, but due to its slight increase in performance, decrease in performance in terms of the METEOR metric, lower interpretability, and lack of common use today, we use the other attention mechanism.

The soft deterministic attention mechanism directly obtains the mean of the context vector $\hat{z}_t$ from the following relation:

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

In the above relation, α represents the relative importance that the model assigns to the i-th location to generate the t-th word. Also, this attention mechanism, unlike the

previous one, is continuous and differentiable, so the error can be reduced with the usual backpropagation algorithm. This attention mechanism was introduced in the paper by Bahdanau et al. (2014), and the way to obtain alpha is as follows:

$$e_{t,i} = f_{att}(a_i, h_{t-1}) = v_a^T \tanh(W_a a_i + U_a h_{t-1})$$

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^{L} \exp(e_{t,k})}$$

Thus, we form the 'Attention' class:

```
class Attention(nn.Module):
    def __init__(self, encoder_dim, decoder_dim, attention_dim):
        super().__init__()
        self.w_a = nn.Linear(encoder_dim, attention_dim)
        self.w_h = nn.Linear(decoder_dim, attention_dim)
        self.w_v = nn.Linear(attention_dim, 1)
        self.tanh = nn.Tanh()
        self.softmax = nn.Softmax(dim=1)
        self.beta_layer = nn.Linear(decoder_dim, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, a, h):
        att_a = self.w_a(a)
        att_h = self.w_h(h).unsqueeze(1)
        att = self.w_v(self.tanh(att_a + att_h)).squeeze(2)
        alpha = self.softmax(att)
        weighted_sum = (a * alpha.unsqueeze(2)).sum(dim=1)
        beta = self.sigmoid(self.beta_layer(h))
        context = beta * weighted_sum
        return context, alpha
```

Listing 3: Attention Module Implementation.

The paper states that the attention weights are further multiplied by a value called beta, which is calculated by the model upon receiving the hidden state.

$$\phi(\{a_i\}, \{\alpha_i\}) = \beta \sum_{i=1}^{L} \alpha_{t,i} a_i$$

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

1.3.3 LSTM Implementation

The LSTM model is from the family of RNN models, introduced to solve the vanishing gradient problem of RNN models. It consists of three gates called the forget gate, input gate, and output gate. The 'cell state' refers to the transient information of the model. The gates modify the information present in the 'cell state'.

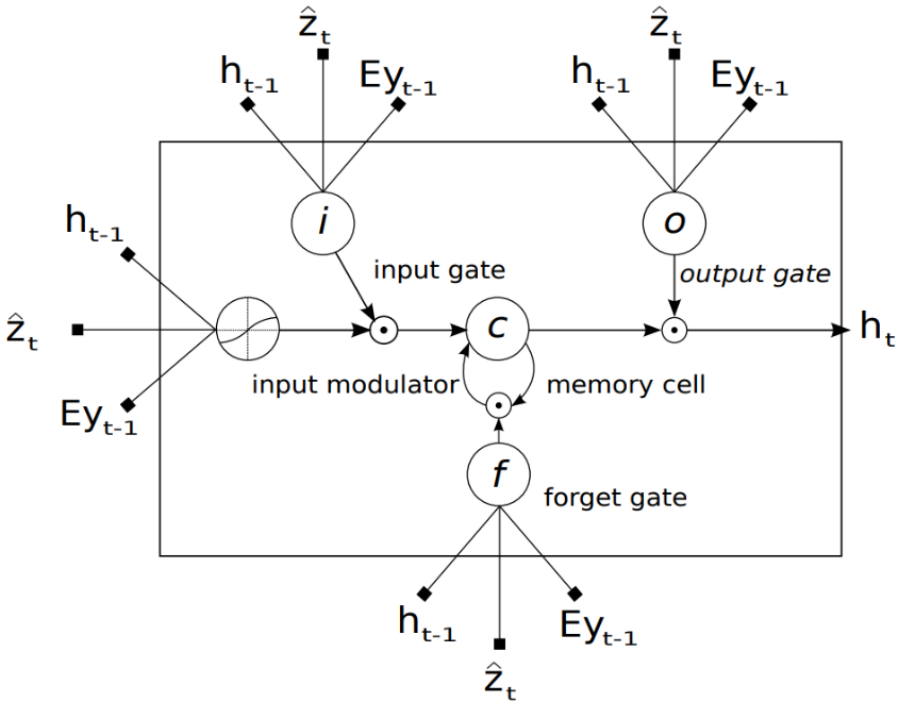

Figure 4: Structure of the LSTM model used, from the reference paper.

As can be seen, the structure of this LSTM is different from the standard LSTMs available in most frameworks. In this structure, the model, like other LSTMs, receives the previous hidden state and cell state, but in addition to the embedding of the previous output, the model also takes the context vector of the current moment. For this reason, we must write the LSTM class ourselves to receive four inputs. The relationship of this model in the paper is as follows:

$$\begin{pmatrix} i_t \\ f_t \\ o_t \\ g_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} T_{D+m+n,n} \begin{pmatrix} Ey_{t-1} \\ h_{t-1} \\ \hat{z}_t \end{pmatrix}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$h_t = o_t \odot \tanh(c_t)$$

In the above relations, i_t, f_t, o_t, c_t, h_t are the input gate, forget gate, output gate, cell state, and hidden state of the LSTM, respectively. The vector $\hat{z} \in \mathbb{R}^D$ is the context vector containing visual information for a specific input, weighted by the attention weights. $E \in \mathbb{R}^{m \times k}$ is the embedding matrix, and m and n represent the dimensions of the embedding and LSTM, respectively. σ is the sigmoid function and $\odot$ denotes element-wise multiplication. Thus, we implement the 'LSTM' class.

1.3.4 Decoder Implementation

After creating the token vectors, we can one-hot encode their indices and feed them to the model for learning. This method has several drawbacks. First, due to the use of one-hot representation, a vector with all zero elements is created, with only the house

corresponding to the token being one. This requires a vector with a length equal to the vocabulary size, while the only information stored in it is the token index. Also, in this way, the entire burden of learning the meaning of these words is placed on the LSTM model, which makes learning almost impossible.

For this reason, in language model construction, another method is used to convert the input to numbers, called **embedding**. In this way, for each word, we place a vector, and the vectors corresponding to all words together form the embedding layer. Then, the model, with the backpropagation algorithm, changes the weights corresponding to the embedding vector of each word in such a way that it can store the different meanings of each word within its vector.

We can use a pre-trained embedding model like fastText, but to achieve better performance, we add an embedding layer to the decoder to learn the meaning of the tokens from the beginning. The decoder model, after obtaining the embedding vector of the true captions' tokens, obtains the initial hidden state and cell state. For this, according to the paper, we use this relation:

$$c_0 = f_{\text{init},c}\left(\frac{1}{L}\sum_i a_i\right)$$

$$h_0 = f_{\text{init},h}\left(\frac{1}{L}\sum_i a_i\right)$$

where $f_{\text{init},c}$ and $f_{\text{init},h}$ are two fully connected layers that are trainable. For the maximum length of the tokens, for each token, first, by giving the feature vector and the hidden state to the attention module, the attention weights and the context vector are obtained. Then, with the embedding data of the last word along with the context vector and the hidden and cell states to the LSTM, the new hidden and cell states are obtained. Then, with the following relations according to the paper, the next token is predicted:

$$p(y_t|a, y_{t-1}) \propto \exp(L_o(Ey_{t-1} + L_h h_t + L_z \hat{z}_t))$$

where $L_o \in \mathbb{R}^{K\times m}$, $L_h \in \mathbb{R}^{m\times n}$, $L_z \in \mathbb{R}^{m\times D}$ and embedding E are learnable and are initialized randomly.

1.3.5 Encoder and Decoder Integration

The decoder module has an embedding length of 300, an LSTM size of 512, an attention layer size of 512, and about 13 million trainable parameters. We tried to make these hyperparameters as similar as possible to the paper. We also use dropout layers in this model according to the paper. These layers work with a rate of 50% and one is placed after the embedding and the other after the first fully connected layer.

Now we form the 'ImageCaptioningModel' class, which includes two parts, the encoder and the decoder. First, we pass the images through the encoder to get the features. Then, we give the feature vector along with the true captions to the decoder to learn.

```
1 class ImageCaptioningModel(nn.Module):
2     def __init__(self, vocab_size, embed_dim, decoder_dim, encoder_dim,
      attention_dim):
3         super().__init__()
4         self.encoder = Encoder()
5         self.decoder = Decoder(vocab_size, embed_dim, decoder_dim,
    encoder_dim, attention_dim)
```

```
    def get_encoder(self):
        return self.encoder

    def get_decoder(self):
        return self.decoder

    def generate_greedy(self, images, max_new_tokens):
        features = self.encoder(images)
        return self.decoder.generate_greedy(features, max_new_tokens)

    def generate_beam_search(self, images, beam_width, max_new_tokens):
        features = self.encoder(images)
        return self.decoder.generate_beam_search(features, beam_width,
    max_new_tokens)

    def forward(self, images, captions):
        features = self.encoder(images)
        return self.decoder(features, captions)
```

Listing 4: Main Image Captioning Model.

This model has a total of about 28.38 million parameters, of which only the parameters related to the decoder, i.e., about 10.83 million parameters, are learnable.

1.4 Training and Evaluation

1.4.1 Model Training

We use the Adam optimizer, which was also used in the paper, with a 'weight$_decay$'$of1e-5andaninitiallearningrateof0.01.Aftereachepoch,thelearningrateismultipliedby0.98.Similartothepa$

The paper states that early stopping was based on the BLEU metric. They observed that the correlation between the BLEU error decreases in the final epochs, and since the main evaluation metric is BLEU, they used this metric for early stopping, and we also use this same metric.

Loss Function In classification models like this, the use of the cross-entropy loss function is common. However, in the paper, a regularizer term is added to the loss function, and it is stated that with it, they have achieved better results, and we also use it. By passing the attention weights through softmax, their sum for all features becomes one. In the paper, the regularizer term is added with the aim that the sum of these weights for all tokens also becomes approximately one. That is, if we consider a specific feature, the sum of the weights given to that feature for each token becomes one. This can be interpreted as the model paying equal attention to all parts of the image throughout the generation process. The paper states that this term increases BLEU and produces better captions. Thus, the loss function is equal to:

$$L_d = -\log(P(y|x)) + \lambda \sum_{i}^{L} \left(1 - \sum_{t}^{C} \alpha_{ti}\right)^2$$

where the first term is the cross-entropy loss and the second term is the regularizer. We consider the weight of the regularizer to be one.

Teacher Forcing Many text generation models use the teacher forcing method, and we also use this method in this part of the training. In this method, we give the model the true token at time t and, with the help of the model, produce the token for the next moment. But without using teacher forcing, at each moment, the token predicted by the model in the previous moment is given to it to predict the next token with its help.

Using teacher forcing makes convergence faster. Also, because the model's predictions have a high error in the initial moments of training, the model's weights are updated by incorrect predictions, and with the accumulation of errors, training becomes difficult. Of course, at inference time, the model does not have the true tokens and teacher forcing is not used. For this reason, there is a difference between the training and evaluation time, and this may cause a drop in quality and instability.

Figure 5: Caption generated in epoch 1: "A man skiing is playing a game of baseball."

Review of a Sample Image and Caption Generated During Training It can be seen that after only one epoch, the model was able to detect the presence of a man engaged in common activities on the beach. However, it could not identify the woman or the kite. The training data error is 5.98 and BLEU is 0.0023. The validation data error is 4.67 and BLEU is 0.0058.

Figure 6: Caption generated in epoch 14: "A person is flying a kite on the beach."

Finally, epoch 14 has the best validation BLEU score, after which overfitting occurs. After 10 epochs, with the occurrence of early stopping, the weights of this epoch are loaded. In this epoch, the generated sentence has no grammatical errors and correctly identifies one of the two people flying a kite. The training data error is 3.45 and BLEU is 0.041, and the validation data error is 3.93 and BLEU is 0.091.

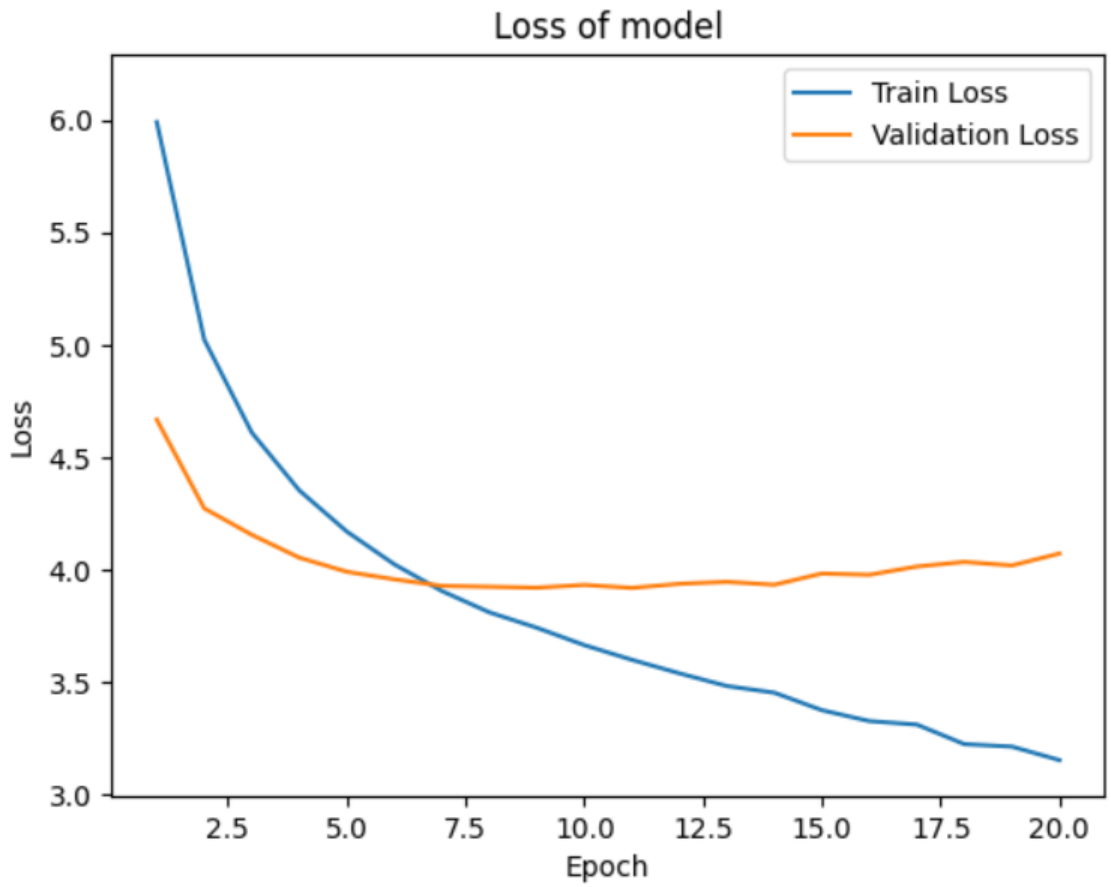

Figure 7: Changes in model loss during training.

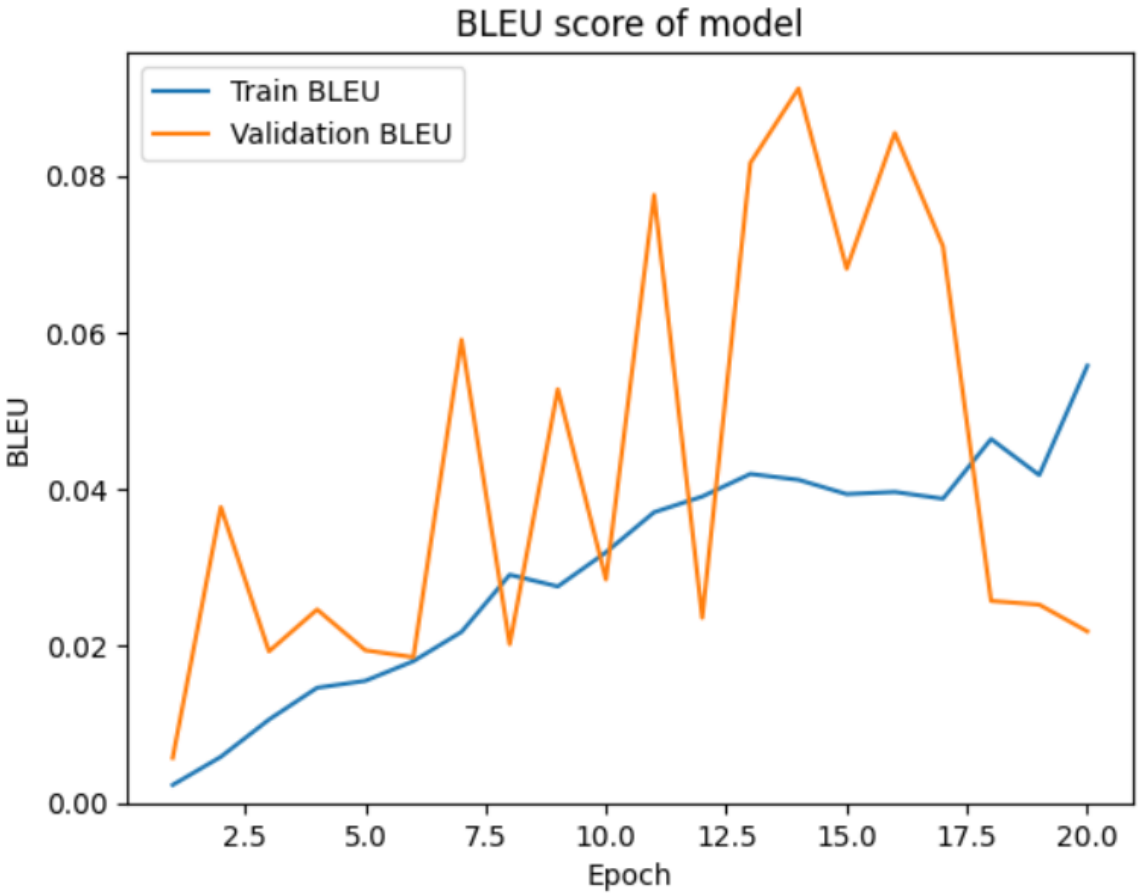

Figure 8: Changes in BLEU score during training.

Training and Validation Error Plots As is clear from the charts, the model overfits, but it is stopped before it occurs due to early stopping. Also, the BLEU chart fluctuates a lot during training, but its overall trend is upward.

1.4.2 Model Evaluation

Text Generation Methods For model evaluation, besides checking the error on the validation data over epochs, we also use other methods. But before examining the evaluation methods, we must determine the text generation method. We implement and review two common methods.

The first algorithm for text generation is the **greedy** method. In this method, at each step, we select the most probable token. This implementation is very simple and justifiable, but it is possible that by choosing a token with high probability, and choosing the next tokens in the same way, the total probability of the generated text is low. Thus, this method does not produce the most probable text, but it produces a reasonably good suboptimal text. The details of this algorithm are as follows:

1. Create an empty array to store the output.
2. Add the start-of-sentence token ‘¡sos¿‘ to the array as the first token.
3. Generate tokens until either the end-of-sentence token ‘¡eos¿‘ is produced or the maximum number of tokens is reached.

4. Use a decoder to get the probability of the next token based on the previous tokens.
5. Select the most probable token. This is the best local choice at each step.
6. Consider the selected token as the current input to generate the next tokens.
7. Add the selected token to the list of generated tokens.
8. Stop the process after generating ‘¡eos¿‘ or generating enough tokens.

The next common method is called **beam search**, which usually produces more accurate and diverse captions. The beam refers to a set of n consecutive tokens, and if the number of beams is considered one, we arrive at the greedy algorithm. The more beams there are, the higher the probability of finding the most probable phrase. This method solves the problems of the greedy algorithm by selecting the most probable beams at each stage and ultimately considering the most probable beam as the output. The details of this algorithm are as follows:

1. Create an initial beam containing the start-of-sentence token.
2. For each beam, form the set of probabilities for the next token. This is done with the help of the model and by giving the existing tokens in each beam.
3. For each next token in each beam, assign a score. This score is equal to the sum of the log probabilities of each token in that beam plus the log probability of the new token.
4. Consider the K most probable beams to use as the beam in the next stage.
5. Repeat steps 2 to 4 until a sufficient number is reached or the ‘¡eos¿‘ token is produced.
6. Finally, consider the most probable beam as the generated text.

Qualitative Evaluation Now we examine the text generated for each of the five images.

True: مردی با عینک و پیراهن و کراوات .

Greedy: با کت و شلوار و کراوات در حال عکس گرفتن مردی .

Beam (K=3): مردی با کت و شلوار و کراوات .

True: نمای نزدیک از یک پیتزای بسیار خوشمزه .

Greedy: یک پیتزا با پنیر و گوجه‌فرنگی روی آن

Beam (K=3): تازه پخته بالای یک میز نشسته_است یک پیتزای .

True: بورد در حال امتحان ترفندهای هستند که یاد گرفته‌اند دو اسکیت .

Greedy: حال اسکیت بورد در هوا در حال انجام یک ترفند مردی در .

Beam (K=3): در حال انجام یک ترفند اسکیت بورد است یک اسکیت بورد .

True: حمام با توالت ، سینک و کابینت آینه‌ای .

Greedy: یک حمام کوچک با یک سینک ، توالت و وان .

Beam (K=3): حمام با سینک ، توالت و وان .

True: سه در حالی که را در دست گرفته‌اند ، لبخند می‌زنند .

Greedy: مردی که یک جفت قیچی در دست دارد .

Beam (K=3): یک مرد و یک مرد در حال اجرا هستند .

Figure 9: Several images with their generated captions.

For the first image, the greedy text is generally correct except for mentioning a suit and tie, whereas the man is only wearing a shirt with a tie. Similar mistakes exist in the beam search text. In the second image, both generated texts have identified the pizza, but the greedy text refers to cheese and tomato, which are not very visible in the image. In the third image, the generated texts are similar, and in each, the presence of a person performing a trick on a skateboard is correctly stated, although in the original text, both individuals are described, the generated texts are also completely correct. In the fourth image, the sink, toilet, and bathroom are present in both texts, but neither

text mentions the mirror, and both mistakenly mention the presence of a bathtub. In the last image, neither of the two texts has enough detail, and in neither is the presence of three individuals mentioned, and in the greedy text, the skateboard they are holding is mistakenly described as scissors. Overall, the model has been able to describe the images reasonably well and has correctly identified the general content, but its ability to express some details, such as the number of people or avoiding mentioning details that are not in the image, is limited.

Common Evaluation Metrics for Caption Generation Models Like other machine learning domains, using a numerical metric can help us compare models more accurately based on fixed mathematical criteria. In general, numerical evaluation methods have advantages over human evaluation methods, including the fact that reviewing a large number of captions by a human is a very slow process, and usually, after reviewing a number of captions, the quality of comparison drops due to fatigue. Also, human comparisons can be subjective, and each person's opinion about comparing captions can differ from others, whereas numerical metrics have a fixed and specific value.

Of course, any numerical metric can only evaluate the caption from a specific aspect, and there is no metric that is without problems and correctly compares all aspects of the text. Thus, usually, to check the metrics, they are compared with human evaluations. Also, in generating captions for images, captions can be generated that, although not present in the reference captions, are logical with respect to the image. For example, the model might generate a caption about objects in the background, and we expect this caption to get a better score than a caption that is completely unrelated to the image, but due to not considering the images, both get a low score.

Usually, the metrics used in natural language processing and machine translation in image captioning are used. The first metric, which we also use, is the **BLEU** or Bilingual Evaluation Understudy, which measures the precision of n-grams between the target text and the generated text. BLEU-1 only uses 1-grams, and 1-gram also shows the number of common words between two captions without considering their position. Thus, BLEU-1 measures the number of words in the generated text that are in the target text. BLEU-2, in addition to 1-grams, also uses 2-grams. 2-grams consider all two consecutive tokens and return the number of pairs that are identical in both captions. Thus, BLEU-2 calculates the average precision obtained from 1-grams and 2-grams. BLEU-3 and BLEU-4 work similarly.

Table 1: Evaluation metrics of the model on the test images.

Method	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Greedy	0.315400	0.161123	0.090481	0.050306
Beam K=3	0.300971	0.154804	0.092486	0.053113

Quantitative Evaluation It can be seen that the results produced by both methods have similar performance, but in the Beam search method, the results are slightly better. In the paper, the BLEU-4 score is reported to be around 24, which is much better than the 5 we achieved. However, the paper used a much larger amount of data for training, and there are also many differences in terms of implementation that can cause this difference.

As we saw, the qualitative results were understandable, and in general, the BLEU metric is not convergent with human evaluation, but many flaws can also be observed in the generated captions, which is probably why the BLEU is low.

1.4.3 Model Improvement and Error Analysis

Error Analysis Model interpretability is an important issue in many fields. In interpretability, we try to investigate the reason and why of the model's behavior. Many methods have been proposed for model interpretation in various tasks, especially computer vision. Some methods are dependent on a specific architecture, and others respond to all models. Also, while some methods assume the models are white-box, others only try to interpret the model by having the inputs and outputs. Also, many methods can only interpret the model for a specific input, but fewer methods exist that interpret the model in general and for all possible inputs.

Attention mechanism is one of the most used mechanisms in modern models, and fortunately, models based on attention are somewhat interpretable. Various methods have been proposed for the interpretability of these modules, but in this exercise, we act similar to the paper. We output alpha, which are the weights of each of the feature pixels, from the attention module, and by resizing it to the dimensions of the original image, the parts where alpha is larger are considered as the areas where the model has paid more attention, and we color them. We write a function that, by receiving a specific sample along with the model and tokenizer, produces the model's output and, by receiving the alpha weights and resizing them, draws the image along with the attention heatmap.

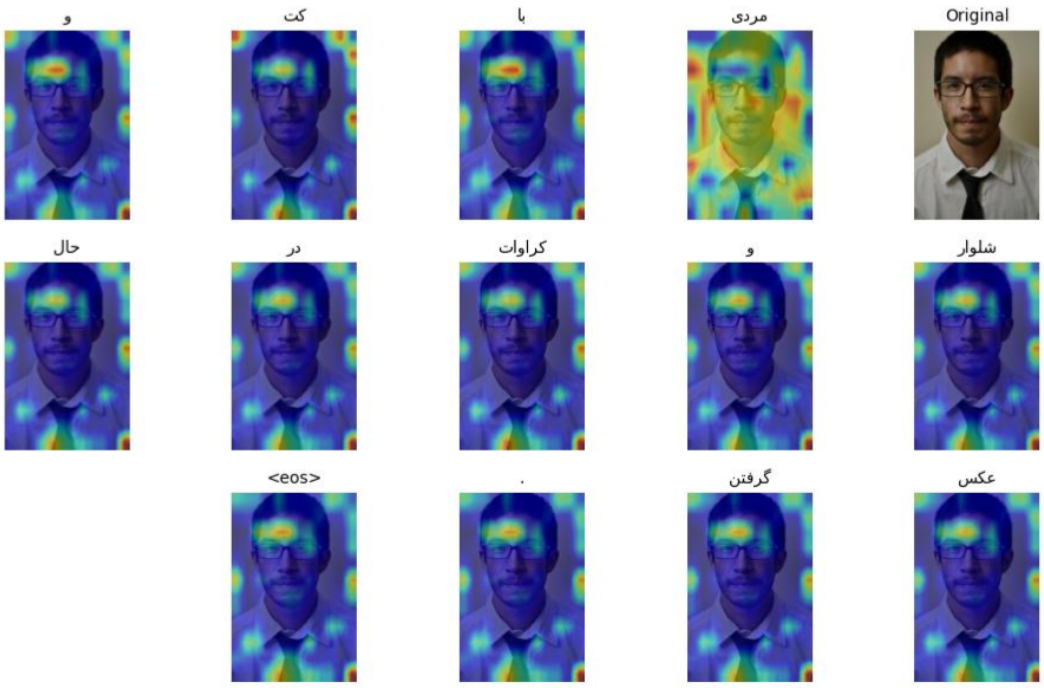

Figure 10: Error Analysis Sample 1: Attention Heatmaps.

In the image above, it can be seen that the model has paid attention to most of the image to produce the first word, and for the next words, it has only paid attention to scattered and small parts, and its attention is almost the same for all subsequent words.

It is not very clear which part's attention caused the production of the words "suit" and "tie", but from the model's attention to small parts of the clothes, it can be guessed that it could not determine the exact type of clothing.

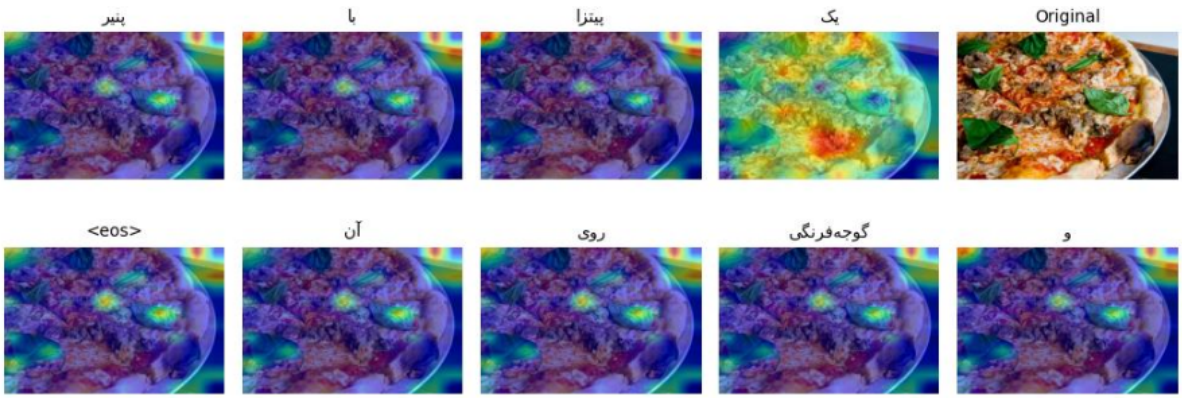

Figure 11: Error Analysis Sample 2: Attention Heatmaps.

Similarly, in this image, the model initially pays attention to the whole image, and then its attention becomes scattered and almost constant. The model has specifically paid attention to the table visible in the corner of the image and the leaves on the pizza, and it is likely that it has identified the presence of cheese and tomato from them.

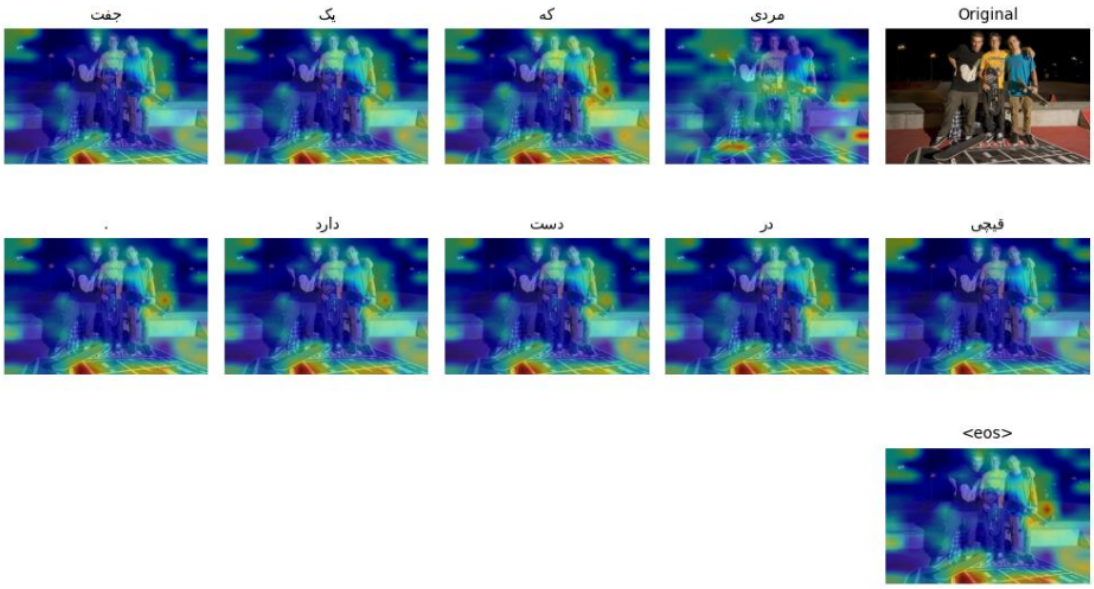

Figure 12: Error Analysis Sample 3: Attention Heatmaps.

In this image, too, it can be seen that the model has paid the most attention to the ground and the skateboard of one of the three people, and from them, it has mistakenly produced the token "scissors".

Implementing Scheduled Sampling In section 2.4.1.2, we explained about Teacher Forcing and its advantages and disadvantages. In these recurrent models, training is usually done by maximizing the probability of producing the correct sequence of tokens. In practice, this is done by maximizing the probability of the next token by receiving the current state of the model and the previous true token. During evaluation, where we do not have access to the true tokens, instead of the true tokens, the tokens produced by the model are given as input. This creates a difference in how the model is run during training and evaluation.

Using other text generation methods other than greedy can partially solve this problem, but it is not completely solved. The main problem is that at evaluation time, unlike training time, the model receives its own output as input at runtime and can create states that are outside of what it was trained on.

Of course, on the other hand, if we do not use Teacher Forcing, in the initial epochs, the model has not yet learned the language and produces wrong tokens, and these mistakes accumulate in the training and cause slow training and a drop in model performance.

In this paper, a method is proposed to solve the problems of Teacher Forcing while preserving its advantages. In the Scheduled Sampling method, the model, which has not yet learned the language at the beginning and can learn better with the help of real tokens, uses the Teacher Forcing method with a probability of 1, but according to a function, after each epoch, the probability decreases, and for each sample, the probability of using the previous token produced by the model increases.

The paper has experimentally shown that this method works better than other methods. In the paper, three different functions for changing the probability are introduced:

- **Linear Decay**: $\epsilon_i = \max(\epsilon, k - ci)$ where $0 \leq \epsilon_i \leq 1$ is the probability of using Teacher Forcing. k and c are two hyperparameters that determine the speed of probability change.
- **Exponential Decay**: $\epsilon_i = k^i$ where $k < 1$ is a hyperparameter.
- **Inverse Sigmoid Decay**: $\epsilon_i = k/(k + \exp(i/k))$ where $k \geq 1$ is a hyperparameter.

Training the Model with Scheduled Sampling In this exercise, we use the first method, i.e., linear change of probability, so that initially the probability is 1, but after each epoch, the probability decreases by 0.02. We train the model with the previous hyperparameters.

Figure 13: Caption generated for the model trained with Scheduled Sampling at epoch 12.

After 12 epochs, the model is stopped due to early stopping. In this epoch, the training error is 3.96 and the validation error is 4.38, and the BLEU for the training images is 0.0246 and for the validation images is 0.0356. In this epoch, the probability of using Teacher Forcing has reached 0.78. It can be seen in the figure above that the generated caption is not correct and lacks the necessary quality, but we will investigate more closely later.

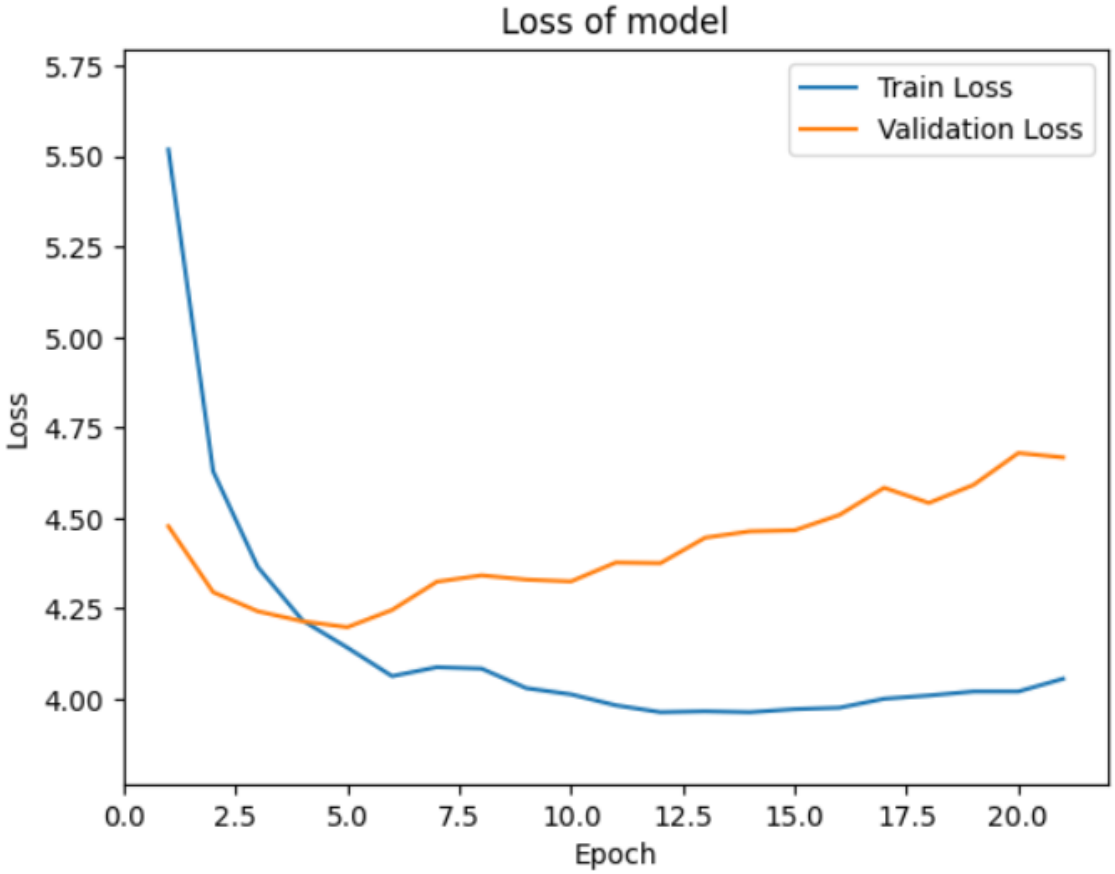

Figure 14: Loss changes of the model trained with Scheduled Sampling during training.

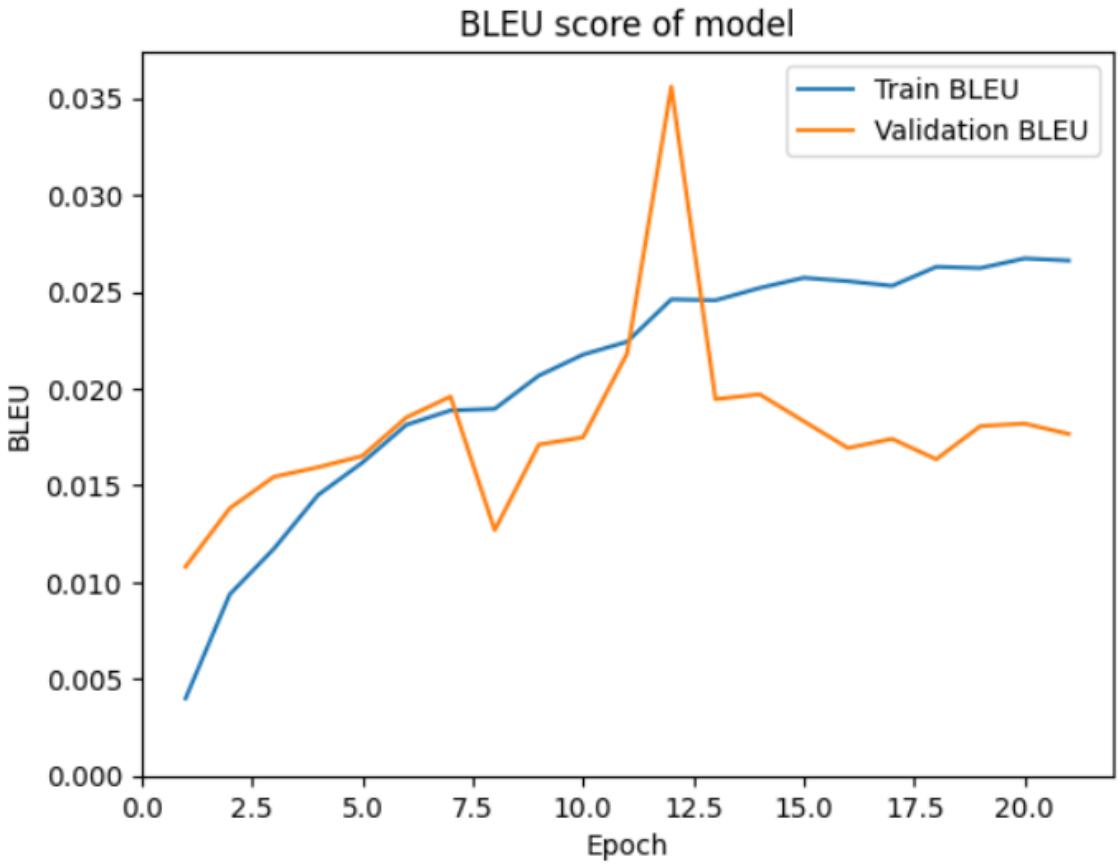

Figure 15: BLEU changes of the model trained with Scheduled Sampling during training.

From the error change chart, it is clear that the model is severely overfitting. We believe that the occurrence of overfitting after using Scheduled Sampling and with the decrease in probability may be due to the slowness of the model's training and the higher speed of the probability decrease compared to the model's training speed, which causes the model to try to learn its own noisy and wrong outputs at the beginning when it has not yet learned the language well, and thus overfitting occurs.

Table 2: Evaluation metrics of the model trained with Scheduled Sampling on the test images.

Method	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Greedy	0.304765	0.153921	0.087267	0.049145
Beam K=3	0.246133	0.127750	0.075998	0.045121

Model Evaluation with Scheduled Sampling It can be seen that unlike the previous model, in this model, the greedy method has produced better results in terms of numerical metrics, which is contrary to the result we got from observing the samples above, which could be due to the distance of BLEU from human metrics or more likely due to the small number of samples reviewed qualitatively. Also, in general, it can be

seen that the performance of this model is weaker than the previous model, the main reason for which we believe is the occurrence of overfitting. Although we stopped it with early stopping, due to overfitting, the model could not learn well for more epochs.

Error Analysis with Scheduled Sampling Similar to the previous model, for this model, we also draw the attention areas.

Figure 16: Error Analysis Sample 1 (Scheduled Sampling).

From the images above and in the rest of the images, it is clear that in this model, the attention happens completely opposite to the previous model. That is, the model does not pay attention to a specific place to produce the first word. Of course, even for the first word, the image features reach the LSTM through the initial hidden state, and the model can produce the first word. To produce the rest of the words, unlike the previous model, in this model, attention is paid to most parts of the image. Of course, with the beta scalar gate, it is possible that the model does not use this attention information to the whole image to produce the next words. Thus, we cannot understand the reason for the wrong color detection.

1.5 Implementing Scaled Dot-Product Attention Mechanism

Today, the attention mechanism is the foundation of most deep networks. So far, we have implemented a specific type of this mechanism called **Additive Attention**, in which the matrices Q and K are added together, and after passing through a non-linear tanh layer and passing through a neuron layer and passing through softmax, the attention weights are calculated. That is, the weights are obtained according to this relation:

$$\alpha = \sigma(V^T \tanh(Q + K))$$

In 2017, another type of attention mechanism called **Scaled Dot-Product Attention** was introduced, which is essentially the same mechanism that is known as the attention mechanism today and is present in many networks. In this method, the matrices Q and K are multiplied together, and after normalization and passing through softmax, the

attention weights are obtained. In this type of attention, the weights are calculated as follows:

$$\alpha = \sigma\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

The main advantage of the second attention mechanism is computational. The increase in speed and improvement in computational cost are clear from the relations of the two mechanisms. In the second attention mechanism, the result is obtained only from the multiplication of two matrices and passing through softmax, for which we know there are fast methods for multiplying two matrices, and for this reason, this mechanism has a computational advantage over the previous attention mechanism, which also needed to pass through tanh and then multiply by another matrix.

Usually, in modern models, a large number of attention mechanisms are placed in parallel next to each other, and each one learns to pay attention to a specific part from a specific aspect, and finally, their results are combined. But in this exercise, we only use one module of attention so that we can correctly compare the results with the other attention mechanism.

```python
class Attention(nn.Module):
    def __init__(self, encoder_dim, decoder_dim, attention_dim):
        super().__init__()
        self.scale = attention_dim ** 0.5
        self.ln_q = nn.LayerNorm(decoder_dim)
        self.ln_kv = nn.LayerNorm(encoder_dim)
        self.q_proj = nn.Linear(decoder_dim, attention_dim)
        self.k_proj = nn.Linear(encoder_dim, attention_dim)
        self.softmax = nn.Softmax(dim=-1)
        self.beta_layer = nn.Linear(decoder_dim, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, features, h):
        h = self.ln_q(h)
        features = self.ln_kv(features)
        Q = self.q_proj(h).unsqueeze(1)
        K = self.k_proj(features)
        V = features
        scores = torch.bmm(Q, K.transpose(1, 2)) / self.scale
        alpha = self.softmax(scores)
        context = torch.bmm(alpha, V).squeeze(1)
        beta = self.sigmoid(self.beta_layer(h))
        context = beta * context
        return context, alpha.squeeze(1)
```

Listing 5: Scaled Dot-Product Attention Module.

Training the Model with Dot-Product Attention Thus, we train the model with all the same hyperparameters as before, except for 'lambda$_r$*egularization* $= 0$', *whichcausestheregulari*

مردی در حال پرواز بادبادک در حال پرواز بادبادک است .

Figure 17: Caption generated for the model with Dot-Product attention at epoch 11.

It can be seen that the model's performance in this epoch is not very suitable, and in the text produced for the image, the phrase "flying a kite" is repeated twice.

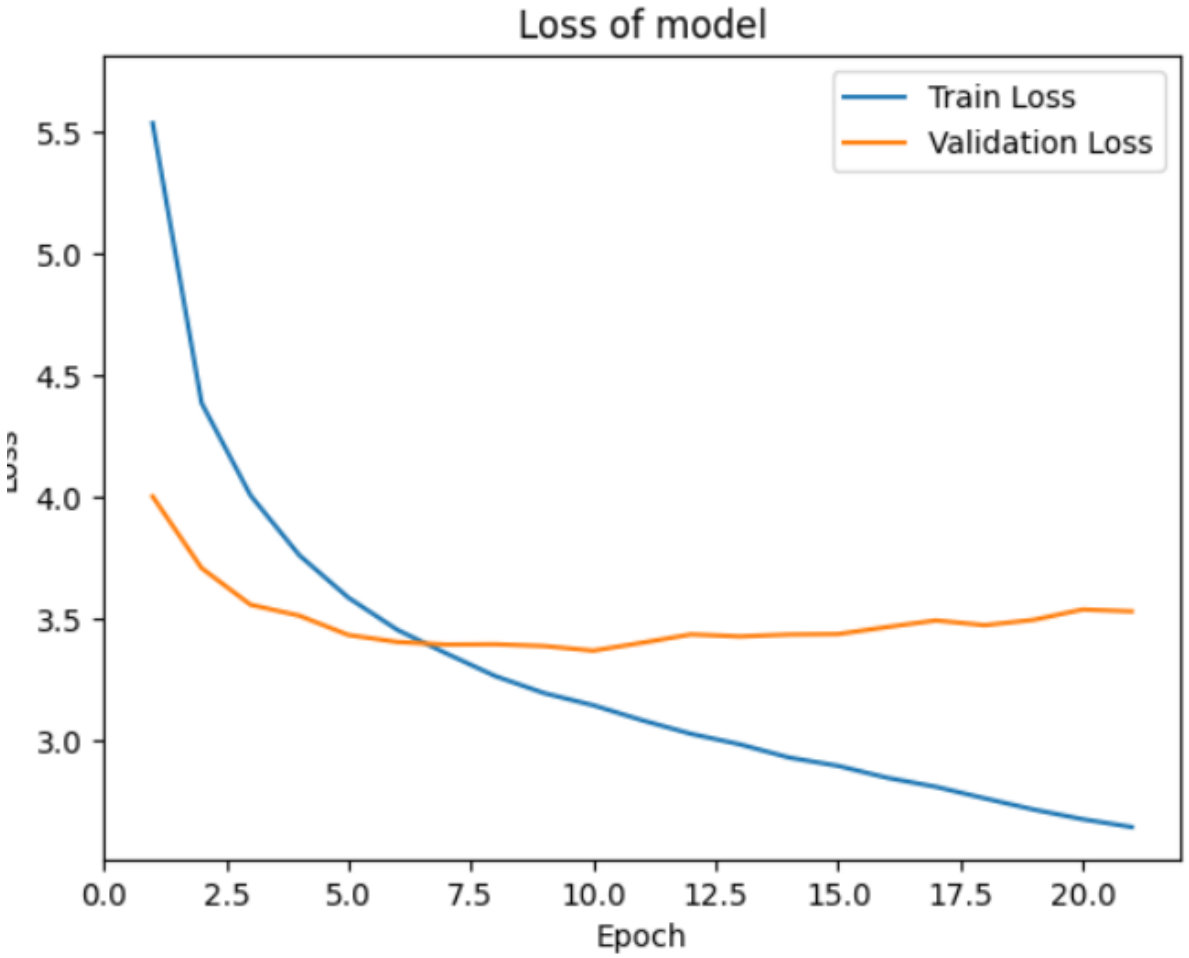

Figure 18: Loss changes of the model with Dot-Product attention during training.

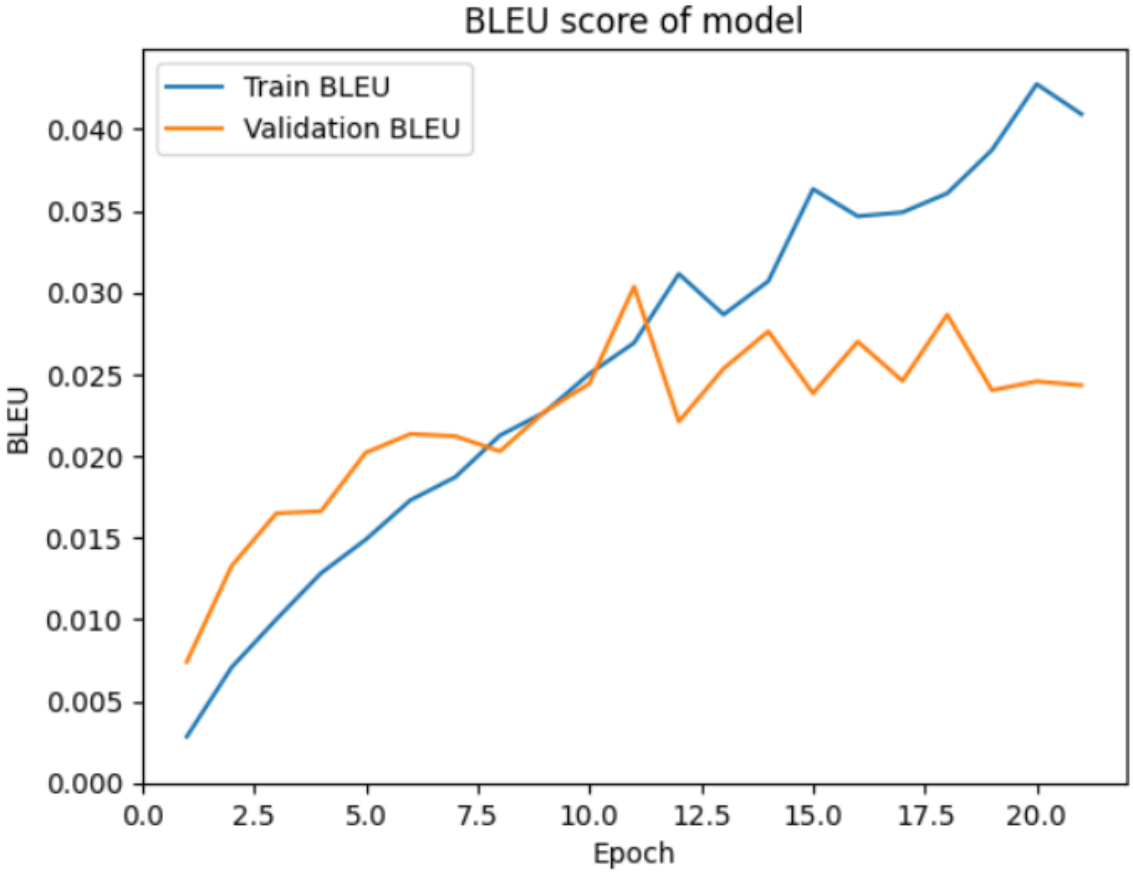

Figure 19: BLEU changes of the model with Dot-Product attention during training.

From the images above, it can be seen that the model overfits if the training is not stopped, and it does not seem to be able to achieve better performance with the current architecture and hyperparameters.

Table 3: Evaluation metrics of the model with Dot-Product attention on the test images.

Method	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Greedy	0.255505	0.128975	0.073439	0.043177
Beam K=3	0.246227	0.125683	0.072289	0.043669

Model Evaluation with Dot-Product Attention From the table above, it is clear that, as we also noticed visually, there is not much difference between the two text generation methods for this model. Also, it can be seen that, as we guessed, the performance of this model is weaker than the previous two models. Although this type of attention mechanism is much more computationally and cost-effective than the previous one, the previous attention mechanism, by using neural and non-linear layers, can be more flexible and perform better with less data, which we also observed here. With an increase in the amount of data and hardware power, this mechanism can be used to learn a large number of patterns from images with less cost by using the ability to parallelize several modules.

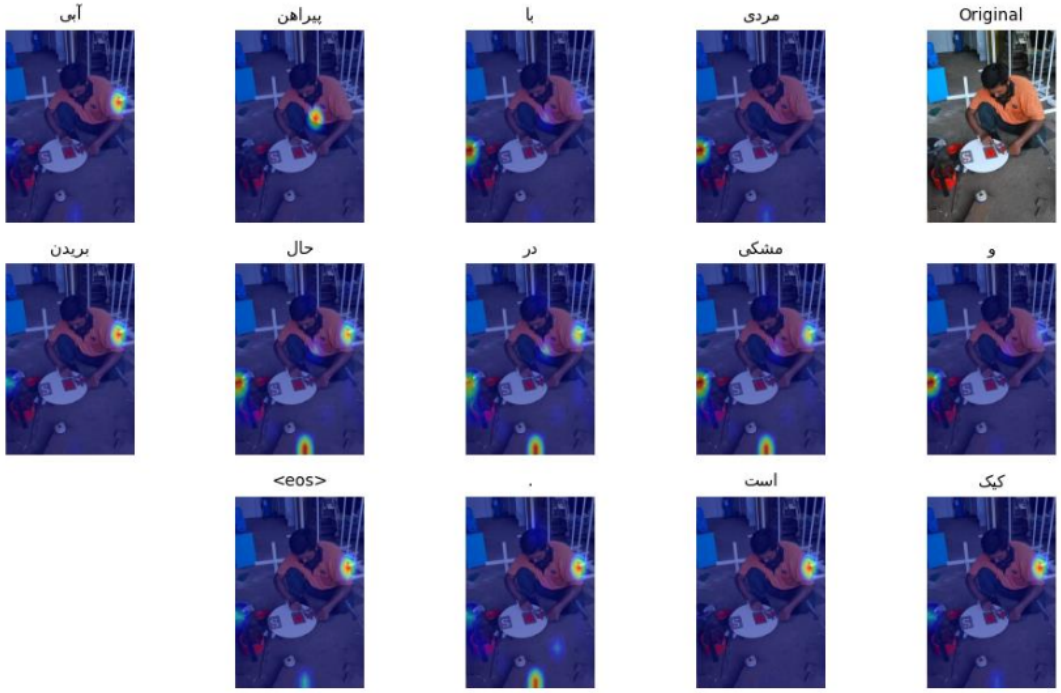

Figure 20: Error Analysis Sample 1 (Dot-Product Attention).

Error Analysis with Dot-Product Attention Again, in the new model, completely new patterns compared to the other models are observed in the attention heatmap. In this model, it seems that unlike the previous two models, where for most words, the models paid attention to the whole image, which one of its possible reasons could be the use of the regularizer term in the loss function, in this model, without using that term, the model only pays attention to small areas of the image and for different words, these areas change slightly. In the image above, the model, without paying attention to the man, has produced the word "man", but it can be seen that to produce the word "shirt", it has paid attention to the man's shirt. For the rest of the words, it has paid attention to the man's sleeve, and it is possible that the production of "cutting a cake" is also a wrong diagnosis of the sleeve as a cake.

1.6 Conclusion

In this exercise, we intended to present a model with the aim of producing descriptive text by receiving an image, which can be used in various fields such as helping blind people, searching for images, and analyzing content, and train it. First, we became familiar with the methods of preprocessing captions and then with the tokenizer and special tokens and how to produce a tokenizer. Then, according to the paper, we implemented an architecture containing an encoder, which was a convolutional network, and a decoder, which was a network consisting of LSTM and an attention mechanism. Also, with the help of validation images, we tried to train the model many times with various hyperparameters to achieve suitable results. We reviewed various numerical evaluation methods for models and explained the advantages and disadvantages of each. For model evaluation in the decoder part, we implemented two common text generation methods, which are the greedy method and the Beam search, and evaluated them qualitatively and numerically with

BLEU metrics. We also tried to investigate by drawing the heatmap of the attention weights, with what logic the texts that are incorrect are produced by the model, and in a way, we provided an interpretation of the model's behavior. Then, by examining the drawbacks of the evaluation and heatmaps, we first replaced the Scheduled Sampling method with the Teacher Forcing method to reduce the distance between how the model is run during training and inference and achieve better performance. We also evaluated this method similarly to before. Finally, we replaced the more modern and common Dot-Product Attention mechanism with the Additive Attention mechanism that was used in the paper to measure the impact of different attention mechanisms on the model's performance. One of the main challenges of this exercise was the costly training of the models in terms of time and data consumption due to the large number of images, which we partially reduced this challenge by reducing the number of training images, and we tried to test a large number of models despite the challenges to find suitable hyperparameters and structure.