Q1
The paper "CAM-RNN: Co-Attention Model Based RNN for Video Captioning" introduces a novel approach for generating captions for videos using a combination of recurrent neural networks (RNNs) and co-attention mechanisms.
The proposed CAM-RNN model uses the concept of co-attention to capture the temporal and spatial information in videos. Co-attention allows the model to attend to relevant video frames and image regions while genarating captions. They use a pre-trained convolutional neural network (CNN) to extract visual features from video frames and employ a bidirectional RNN to model the temporal dependencies in the captions.
The co-attention mechanism in CAM-RNN uses two attention mechanisms: spatial attention and temporal attention. Spatial attention focuses on different regions of each video frame. temporal attention attendss to different video frames. These attention mechanisms enable the model to capture important visual and temporal cues, enhancing the quality of generated captions

to train the CAM-RNN model, the authors use a large-scale video captioning dataset and employ a sequence-to-sequence learning framework. They introduce a new loss function that combines standard captioning loss with a ranking loss, which encourages the model to generate captions that are not only accurate but also rank higher than incorrect captions.

Experimental results show that the CAM-RNN model outperforms several state-of-the-art methods on various evaluation metrics, including BLEU, METEOR, and CIDEr. The model's superior performance indicates the effectiveness of the co-attention mechanism in capturing both spatial and temporal information in videos, leading to more accurate and descriptive captions.

The idea presented at this paper uses two types of attention: region-level attention and frame-level attention, although the region-level attention might produce good results for image captioning the frame-level is not appropriate: because we do not have any frames, we only have one image and as the paper mentioned the image captioning task can be regarded as one frame video captioning.

Q2 – A

In a standard GRU (Gated Recurrent Unit) architecture, there are two gates: the reset gate and the update gate. The reset gate determines how much of the previous hidden state should be forgotten, while the update gate controls how much of the previous hidden state should be combined with the current candidate activation.

If you remove the reset gate and only keep the update gate, the update gate alone is responsible for controlling the information flow and update of the hidden state. By removing the reset gate, the gru loses the control over resetting the hidden state, which can impact the model's ability to capture long-term dependencies in sequential data.

Q2 – B

Making $u_t = 0$ causes the gru to not rely on x_t and it only uses h_t.
Making $u_t = 1$ and $r_t = 0$ causes gru to rely solely on x_t.

Q3-A
softmax operations:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

We want to apply softmax over a vector with 20 units (n = 20):
- (n) exp ops (and keep the result for summation)
- (n-1) sum ops (and keep the sum to use it for other divisions too)
- (n) div ops

sigmoid operations:

$$S(x) = \frac{1}{1 + e^{-x}}$$

We want to apply sigmoid over a vector with 20 units (n = 20):
- (n) exp ops
- (n) sum ops
- (n) div ops
- (n) minus ops (0-x)

so if we are going to use sigmoid, we end up with (1) more summation and (n) more subtractions.
But if we do not consider the negative op as an op, we may say they have roughly the same number of operations.

Q3 – B
As it discussed in class, a linear layer can act as embedding layer since it is a map from input to other domain and it has the same number of parameters to be learned (except some few more biases). So in this case both model 1 and model 2 has roughly the same learning capacity. Although with embeddings we can use pretrained vectors. In this case we are prune to overfitting and we may have more generalization.

4-A: Code explain:

This class is responsible for basic NLP operations.

```
class Vocab():
    def __init__(self):
        self.i2s = ['<PAD>', '<SOS>', '<EOS>', '<UNK>']
        self.s2i = {item:index for index, item in enumerate(self.i2s)}
            these two datastructures are keeping track of indices and strings

    def build_vocabulary(self, caps):
            this function used to generate tokens and store them with their indices
        for cap in caps:
            for tok in self.tokenize(cap):
                if tok not in self.s2i:
                    self.i2s.append(tok)
                    self.s2i[tok] = len(self.i2s) - 1

    def tokenize(self, cap): using the spacy lib tokenizer, we tokenize (and clean) each caption.
        return [tok for tok in spacy_eng(cap.replace("'","").replace('"',"").lower())]

    def vectorize(self, cap):
            with this function we are going to create a vector for the input caption, this vector contains
token ids
        tokenized_cap = self.tokenize(cap)
        return [self.s2i[tok] if tok in self.s2i else self.s2i["<UNK>"] for tok in tokenized_cap]

    def __len__(self):
        return len(self.i2s)
```

```
class MyDataset(Dataset): ## this is basic dataset class, it uses vocab class to tokenize and do nlp stuffs.
    def __init__(self, dir_img, dir_ann):
        super(MyDataset, self).__init__()
        self.dir_img, self.dir_ann = dir_img, dir_ann
        self.df = pd.read_csv(dir_ann)

        self.caps = self.df['caption'].tolist()
        self.imgs = self.df['image'].tolist()

        self.vocab = Vocab()
        self.vocab.build_vocabulary(self.caps)
        self.vec_caps = [torch.tensor([self.vocab.s2i['<SOS>']] + self.vocab.vectorize(cap) + [self.vocab.s2i['<EOS>']]) for cap in self.caps] ## vectorizing each caption

        self.transform = transforms.Compose(
            [ ## transform to be applied to images
                transforms.ToTensor(),
                transforms.Resize((224, 224)),
                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
            ]
        )

    def __len__(self):
        return len(self.df)

    def __getitem__(self, index): ## returning cleaned, tokenized caption with image
        cap = self.vec_caps[index]
        img = self.transform(cv2.imread(os.path.join(self.dir_img, self.imgs[index])))
        return (img, cap)
```

4-B:
10 epoch, 200 hidden, 200 embedding, adam op, bce
BLEU 1 = 0.4411764705882353
BLEU 2 = 0.20097113025182892

4-C:
10 epoch, 200 hidden, 200 embedding, adam op, bce
bleu1 = 0.5071764789787113
bleu2 = 0.3062017326554036

4-D:
Using the glove embeddings resulted in better results, because those embeddings are pretrained and they can converge to better trained embeddings. But when we train an embedding from scratch we need lots of data for it to work well. So we can conclude that our dataset was not big enough.

4-E:
10 epoch, 200 hidden, 200 embedding, adam op, bce
bleu1 = 0.4524544524053224
bleu2 = 0.2464745214996422

well this result is still better than trainng embeddings from scratch but is worse than freezing cnn, I think the reason is the same as before, since we do not have a lot of data we could not reach a good point by training cnn weights. But since we used the glove embeddings, it is still better than normal situation.

4-F:
No I don’t think so, because all the three models converged (loss get smaller) and if the gradient vanishing was happening, the models could not learn the train data and loss would not reduce.

4-g:
Token-level models and character-level models (RNN) differ in the level of granularity at which they process text. Token-level models operate on individual words or subword units. They are good at generalizing to unseen words and handling long-range dependencies efficiently. In contrast, character-level models process text at the character level, allowing them to handle any combination of characters and effectively **handle out-of-vocabulary** terms. While they can capture character-level patterns and handle sequences of variable length, they may require more training data and computational resources.