Vision and Language

Sanghyuk Chu
Hyewon Yoo
Department of Electrical and Computer Engineering

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

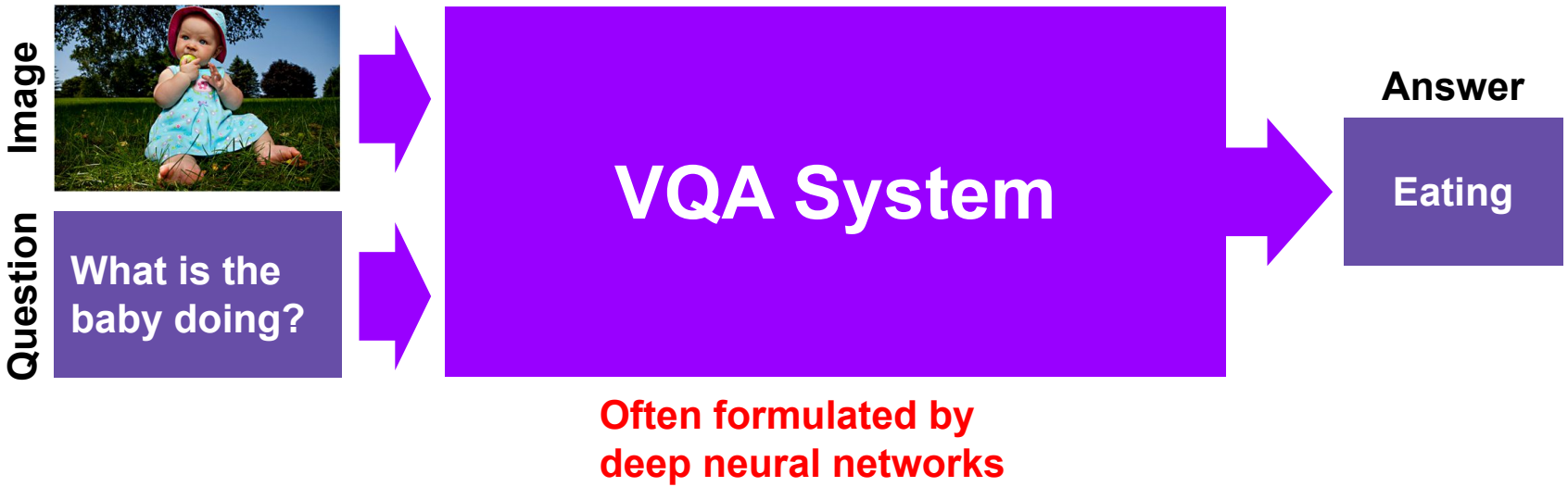
Visual Question Answering

Visual Question Answering

- Characteristics
  - Providing an image and a question as inputs
  - Involving significant amount of learning from data
- Motivation
  - General purpose image understanding: from low to high level
  - Unified framework to solve the tasks defined on demand by questions

Visual Question Answering

- Technique to answer questions about an image
  - Input: image and question (sequence of words)
  - Output: answer (a word or sequence of words)



Problem Formulation

- Technique to answer questions about an image
  - Input: image and question (sequence of words)
  - Output: answer (a word or sequence of words)

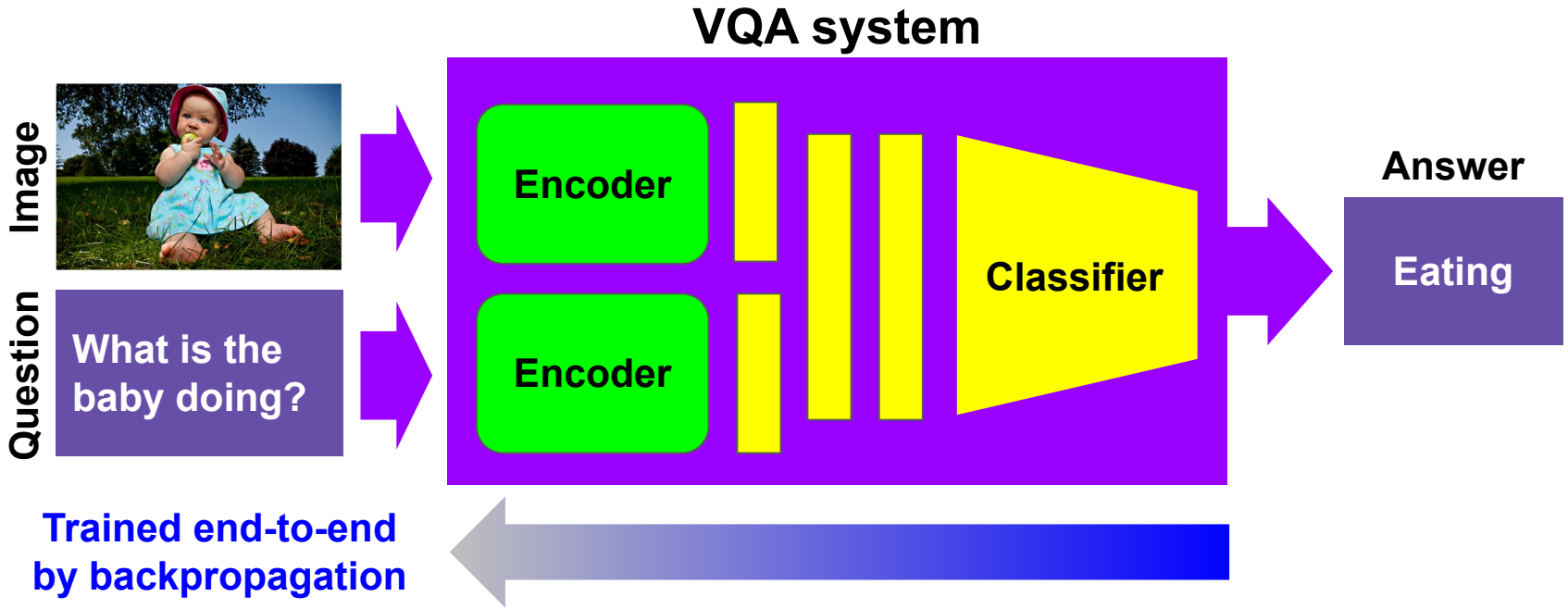
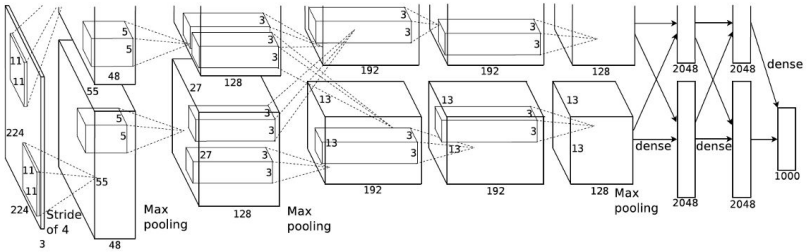
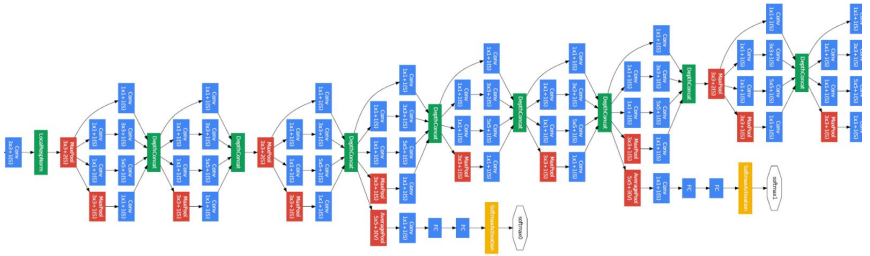


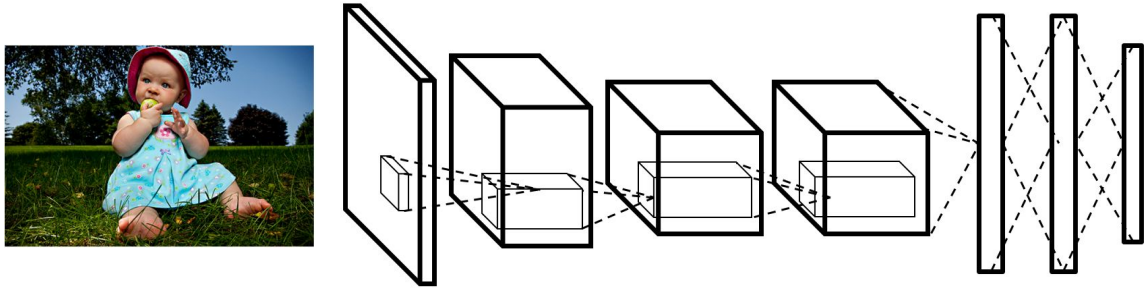
Image Encoder

- Convolutional Neural Networks (CNNs)
  - AlexNet / VGG / GoogLeNet / ResNet



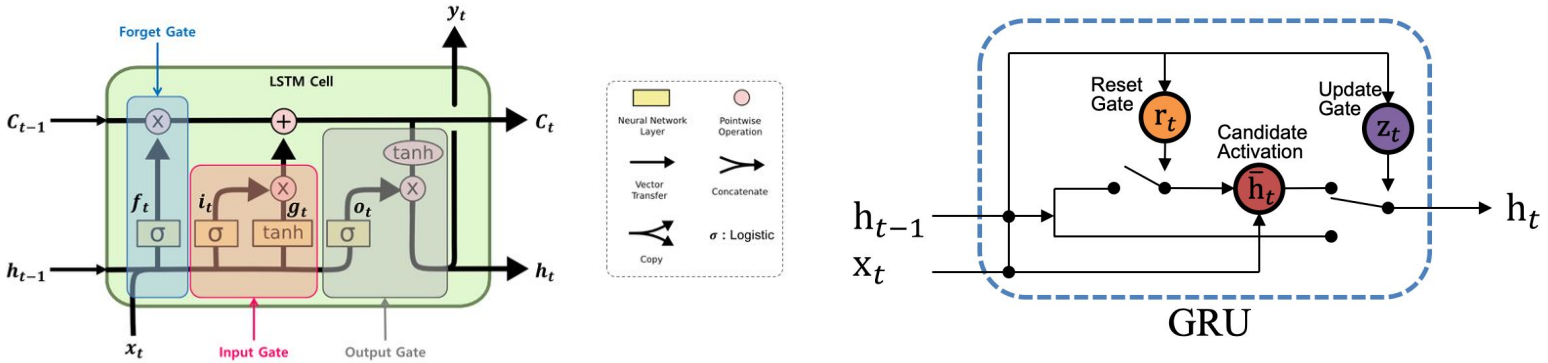


- Representing images with fixed dimensional feature vectors

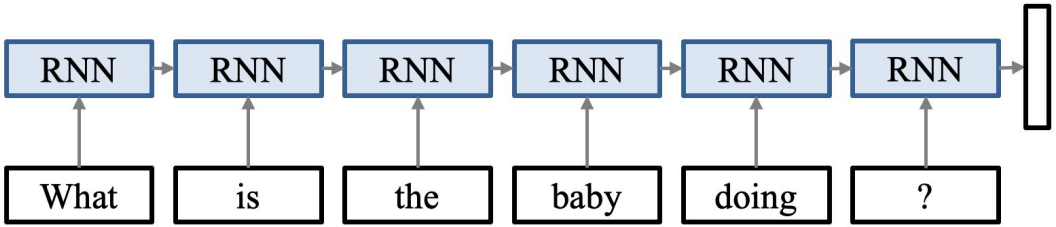


Sentence Encoder

- Recurrent Neural Networks (RNNs)
  - LSTM / GRU

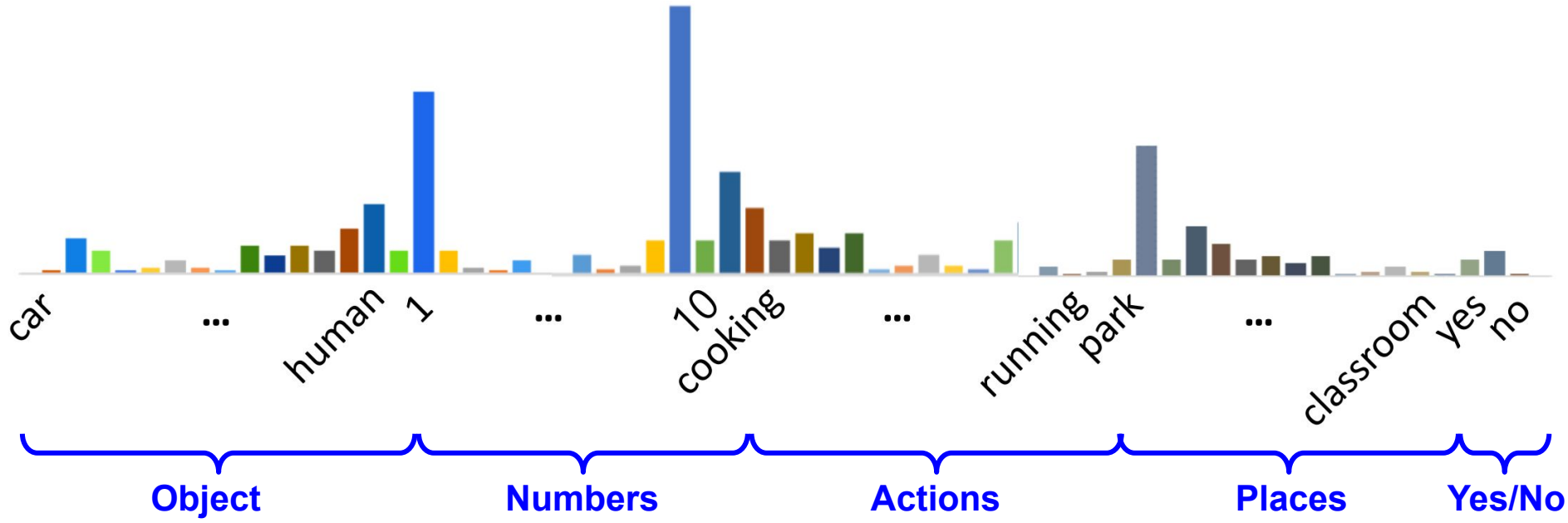


- Representing questions with fixed dimensional feature vectors



Answer Generation

- Flat classification



  - All classes are handled equivalently.
  - However, some classes may not be comparable, exclusive, and compatible.

VQA Dataset

Q: What animal is this?

Q: Is this vegetarian pizza?

Q: How many dogs are seen?

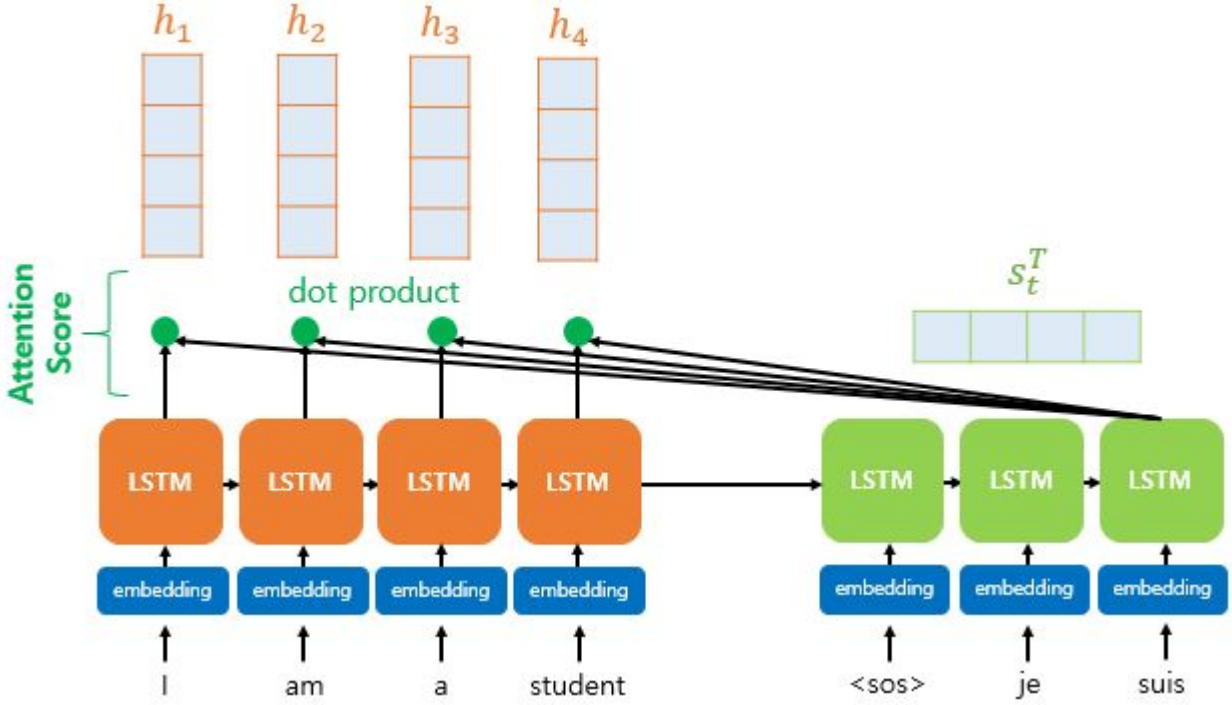
Q: What color is the car?

Q: What is the mustache made of?

[Antol15] S. Antol et al.: **VQA: Visual Question Answering**. ICCV 2015

Attention

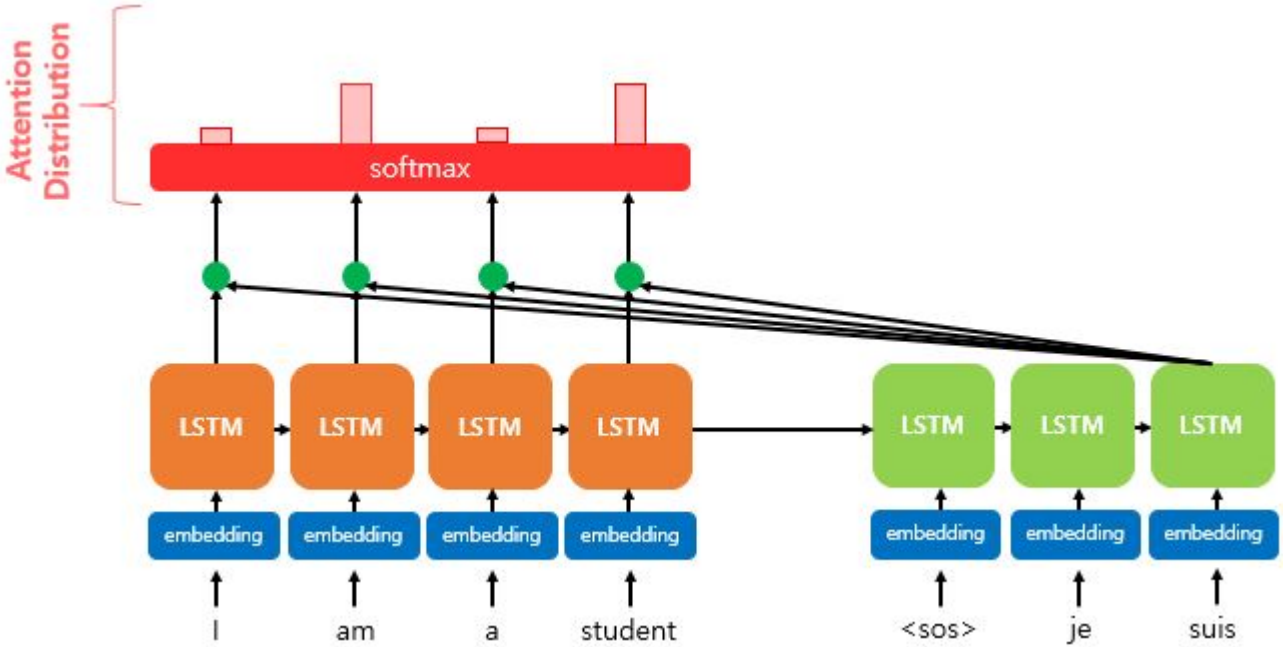
0. Hidden state = word embedding



https://wikidocs.net/22893

Attention

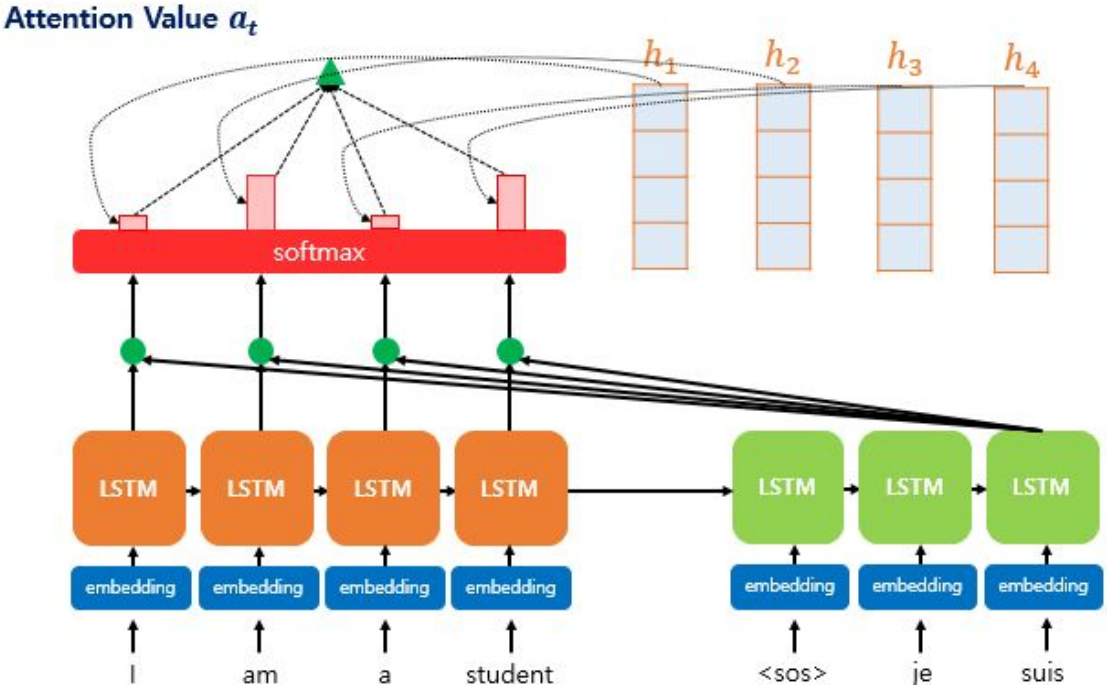
1. Compute score for each hidden state



https://wikidocs.net/22893

Attention

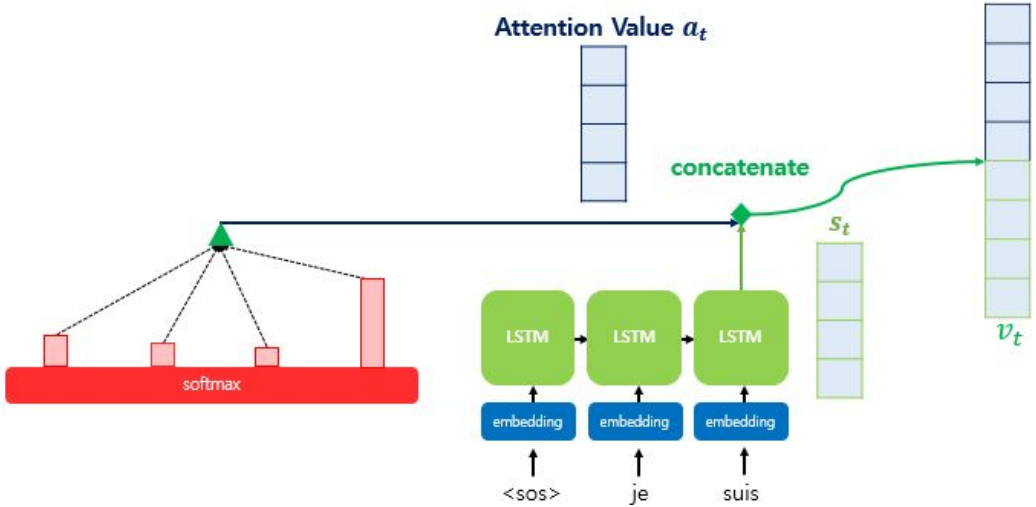
2. Multiply each hidden state by its softmax score



https://wikidocs.net/22893

Attention

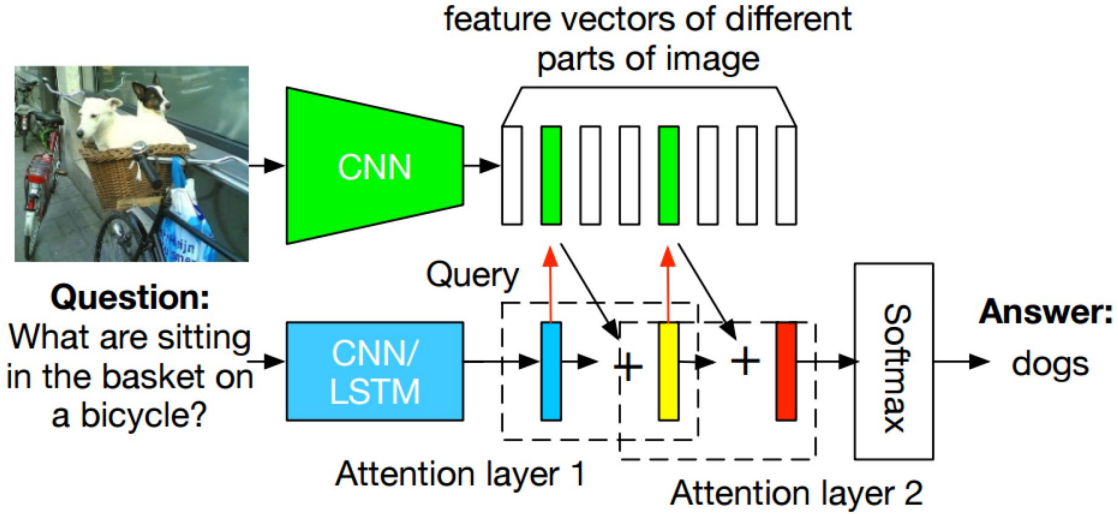
3. Sum alignment vectors & feed the context vector into the decoder



https://wikidocs.net/22893

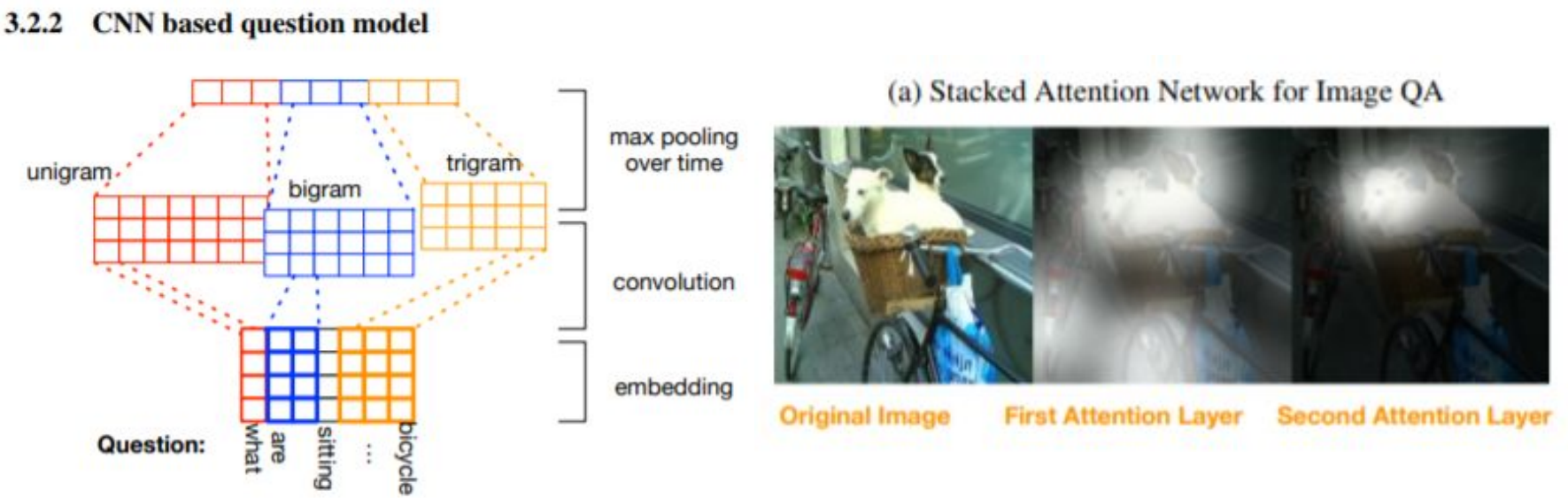
Stacked Attention Network

- Multi-step reasoning for image QA
  - Image model: using CNN to extract high level image representations
  - Question model: using CNN or LSTM to extract a semantic vector
  - Attention model: locating, via multi-step reasoning, the image regions that are relevant to the question for answer prediction



[Yang15] Z. Yang, X. He, J. Gao, L. Deng, A. Smola: **Stacked Attention Networks for Image Question Answering**. CVPR 2016

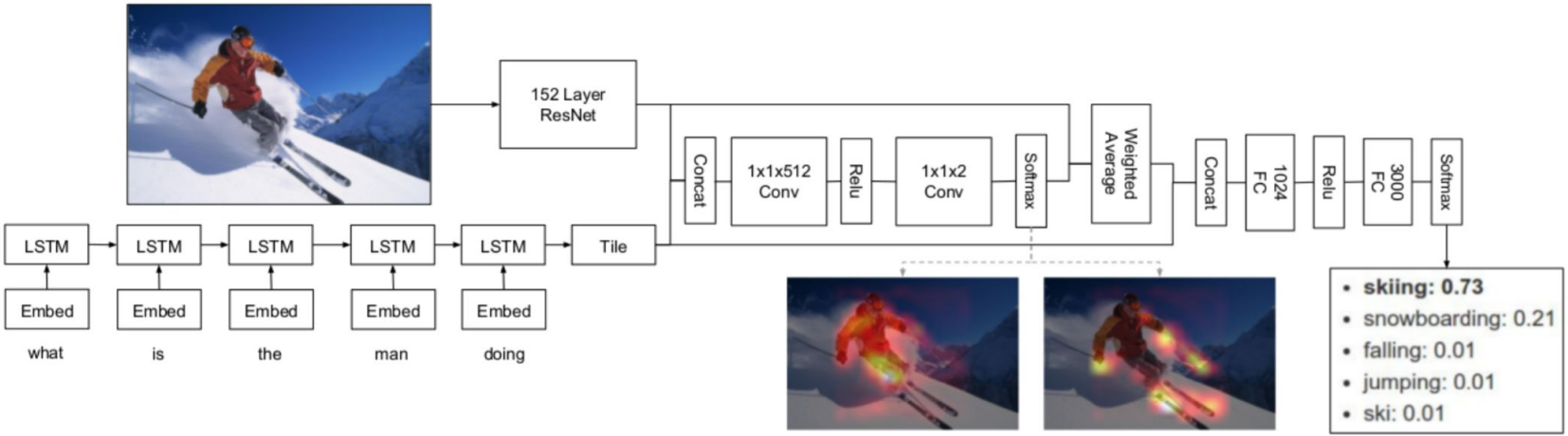
Stacked Attention Network



[Yang15] Z. Yang, X. He, J. Gao, L. Deng, A. Smola: **Stacked Attention Networks for Image Question Answering**. CVPR 2016

Show, Attend and Answer: Strong baseline for VQA

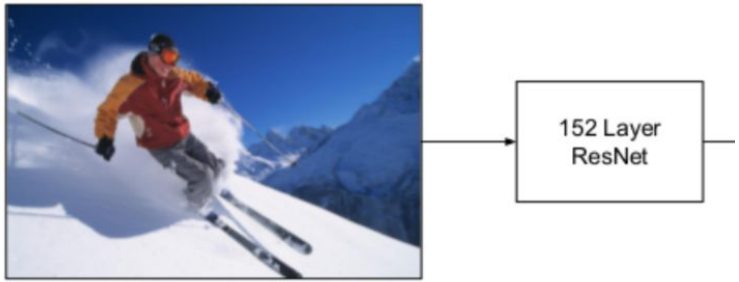
- Simple version of Stacked Attention Network
  - Image model: using CNN to extract high level image representations
  - Question model: using LSTM to extract a semantic vector
  - Attention model: locating, via multi-step reasoning, the image regions that are relevant to the question for answer prediction



[Yang15] Vahid Kazemi, Ali Elqursh: **Show, Ask, Attend, and Answer: A Strong Baseline For Visual Question Answering**.

Image Encoder

- ResNet 152 as a feature encoder
- Image preprocessing:
  - Resize
  - Center crop
  - Normalize
- Extract feature after 4th residual block




```
def get_transform(target_size, central_fraction=1.0):
    return transforms.Compose([
        transforms.Resize(int(target_size / central_fraction)),
        transforms.CenterCrop(target_size),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225]),
    ])
```

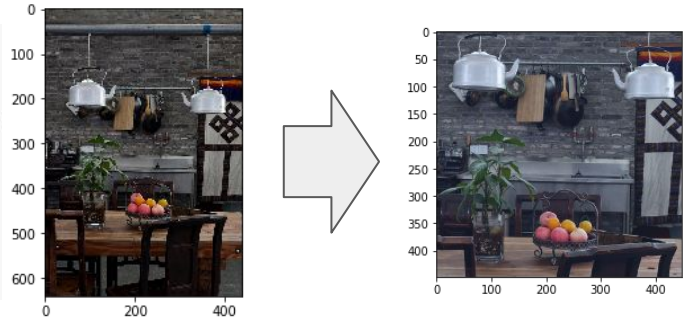
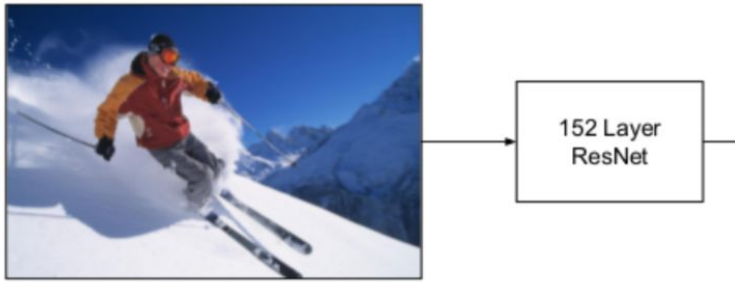
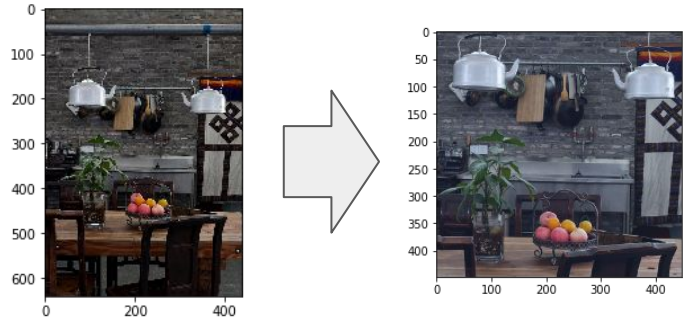


Image Encoder

- Extract feature after 4th residual block
- Useful tip for extracting particular feature




```
class FeatureNet(nn.Module):
    def __init__(self):
        super(FeatureNet, self).__init__()
        self.model = resnet152()
        self.model.load_state_dict(torch.load('pretrained/resnet152.pth'))

        def save_output(module, input, output):
            self.buffer = output
        self.model.layer4.register_forward_hook(save_output)

    def forward(self, x):
        self.model(x)
        return self.buffer
```



Question Preprocessing

- Preprocessing
  - Remove question mark→
  - Tokenization based on blanks between words→
  - Convert words to vocabulary index→
  - We need length of the sequence too→

- Input: What is the table made of?
  - What is the table made of
  - ['what', 'is', 'the', 'table', 'made', 'of']
  - tensor([3, 2, 1, 68, 59, 9]
  - tensor(6)

- Important: words that is not in vocabulary
  - Words that are not in vocabulary is usually encoded as a special token <UNK>
  - In our implementation this corresponds to vocabulary index ‘0’.

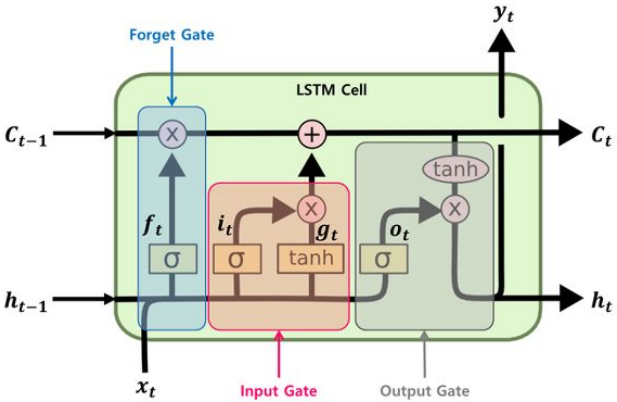
Question Preprocessing

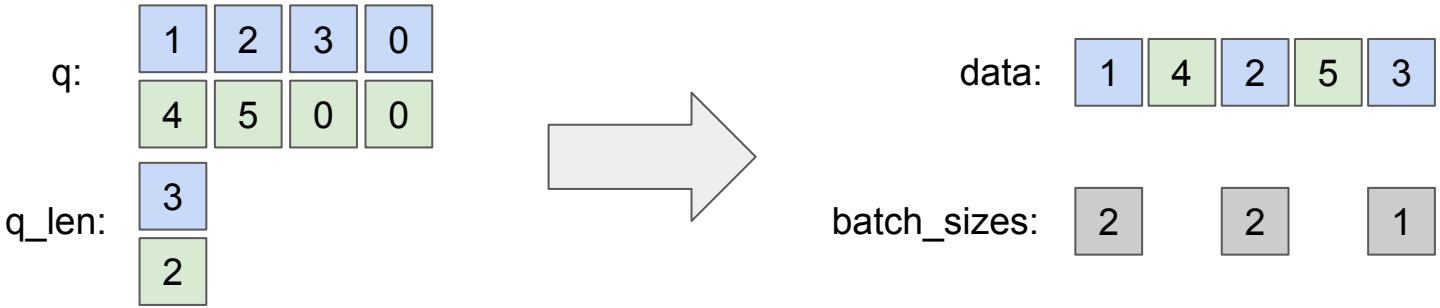
- Exercise 1
  - Write two functions for preprocessing question for visual question answering
  - Use vocabulary in dictionary: token_to_index
  - There is no <unk> token, whose index is ‘0’, in the token_to_index dict.

```
def process_question(question):
    """
    Implement this function
    """
    return tokenized_question

def encode_question(question, sequence_length=None):
    if sequence_length is None:
        sequence_length = len(question)
    """
    Implement this function
    """
    return q_tensor, q_length_tensor
```

Question Encoder

- Using LSTM for encoding question
  - (h_n, c_n) = LSTM(x[0...n], (h_0, c_0))
- Handling variable length questions
  - Pytorch LSTM implementation supports variable length sequence based on special data format: PackedSequence
- Pack_padded_sequence
  - Question should be sorted in descending order of length before applying the function

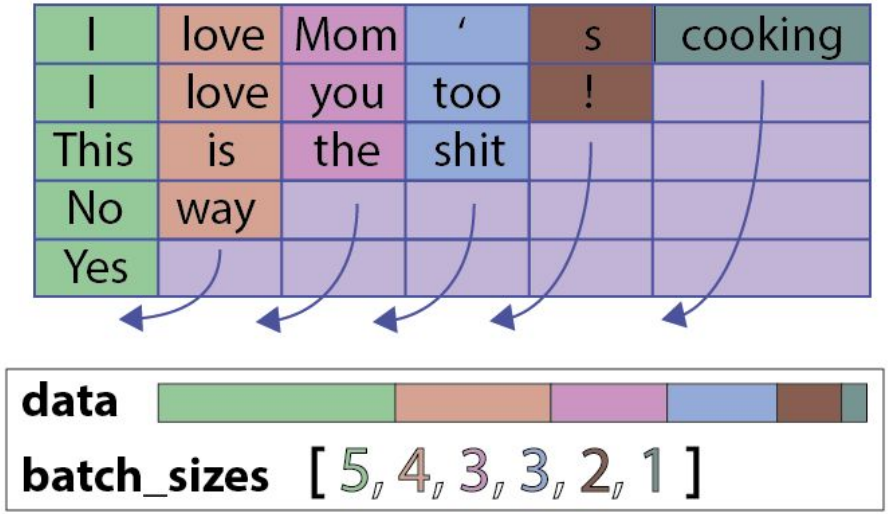




Question Encoder

- Pack_padded_sequence

I	love	Mom	'	s	cooking
I	love	you	too	!	
No	way				
This	is	the	shit		
Yes					

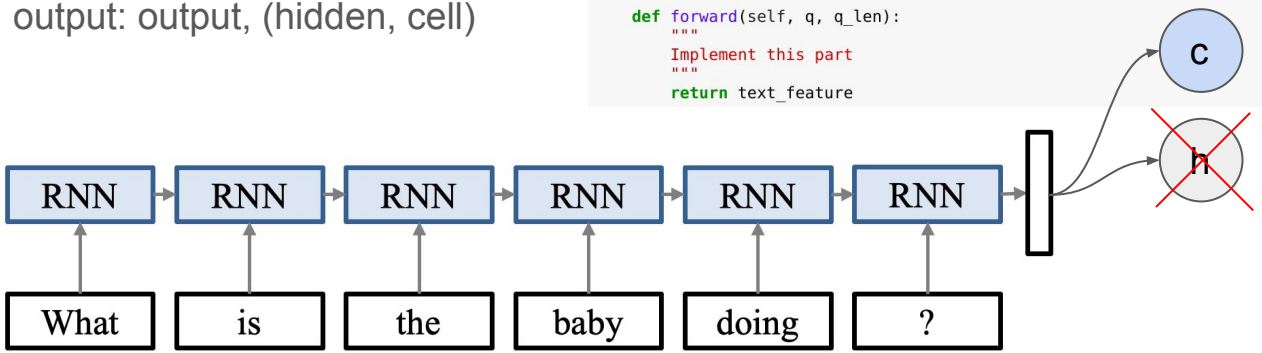


https://medium.com/huggingface/understanding-emotions-from-keras-to-pytorch-3ccb61d5a983

Question Encoder

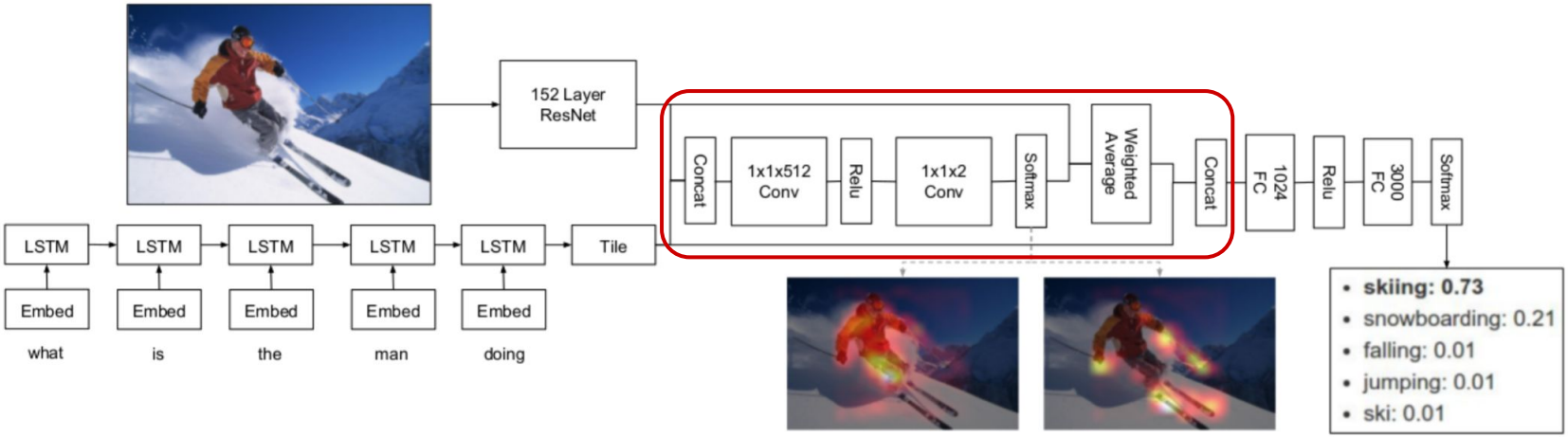
- Using last memory cell as a feature
- Exercise 2
  - q -> embedding -> dropout -> tanh -> LSTM -> cell as feature
  - packed_padded_sequence(
    input, length, batch_first=True
)
  - LSTM output: output, (hidden, cell)

```
class TextProcessor(nn.Module):
    def __init__(self, embedding_tokens, embedding_features, lstm_features, drop=0.0):
        super(TextProcessor, self).__init__()
        self.embedding = nn.Embedding(embedding_tokens, embedding_features, padding_idx=0)
        self.drop = nn.Dropout(drop)
        self.tanh = nn.Tanh()
        self.lstm = nn.LSTM(input_size=embedding_features,
                            hidden_size=lstm_features,
                            num_layers=1)
        self.features = lstm_features

        self._init_lstm(self.lstm.weight_ih_l0)
        self._init_lstm(self.lstm.weight_hh_l0)
        self.lstm.bias_ih_l0.data.zero_()
        self.lstm.bias_hh_l0.data.zero_()

        init.xavier_uniform_(self.embedding.weight)

    def _init_lstm(self, weight):
        for w in weight.chunk(4, 0):
            init.xavier_uniform_(w)

    def forward(self, q, q_len):
        """
        Implement this part
        """
        return text_feature
```



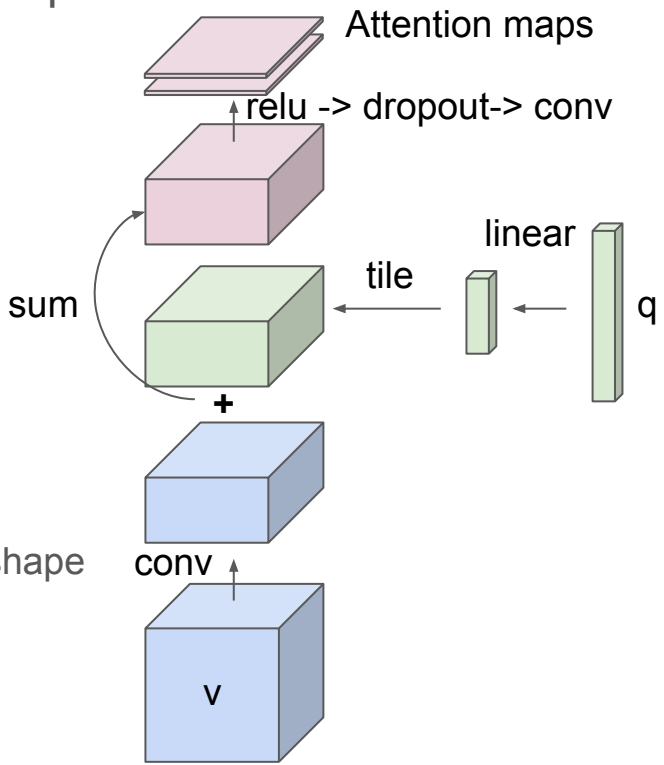
https://pytorch.org/docs/1.9.1/generated/torch.nn.LSTM.html

Fusion - Multi-head Attention



Multi-head Attention

- Compute multiple attention map with a single step
  - glimpse: number of attention maps
  - Attention module
    - Input: visual feature map, question feature
    - Output: Attention maps (feature map)
  - apply_attention(input, attention)
    - attention: [b, g, h, w, 1] tensor (g is glimpse)
    - Input: [b, h, w, c] tensor
    - Output: [b, g * c] tensor
      - Attention -> reshape -> softmax
      - Input - reshape
      - (Attention, input) -> * (broadcasting) -> reshape



Multi-head Attention

- Exercise 3
  - Implement attention module
  - Use ‘tile_2d_over_nd’ function

```
class Attention(nn.Module):
    def __init__(self, v_features, q_features, mid_features, glimpses, drop=0.0):
        super(Attention, self).__init__()
        self.v_conv = nn.Conv2d(v_features, mid_features, 1, bias=False)  # let self.lin take care of bias
        self.q_lin = nn.Linear(q_features, mid_features)
        self.x_conv = nn.Conv2d(mid_features, glimpses, 1)

        self.drop = nn.Dropout(drop)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, v, q):
        """
        Implement this part
        """
        return attention
```

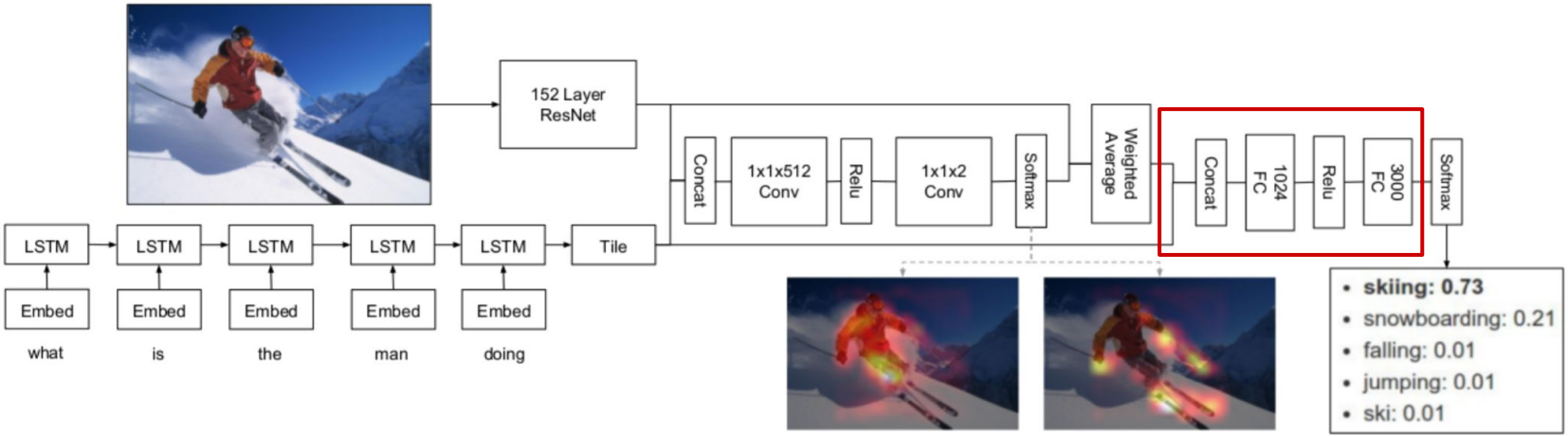
Multi-head Attention

- Exercise 4
  - Implement function for applying attention to pool feature

```
def apply_attention(input, attention):
    """ Apply any number of attention maps over the input. """
    n, c = input.size()[:2]
    glimpses = attention.size(1)

    # flatten the spatial dims into the third dim, since we don't need to care about how they are arranged
    """
    Implement this part
    """
    return pooled_feature
```

Classifier

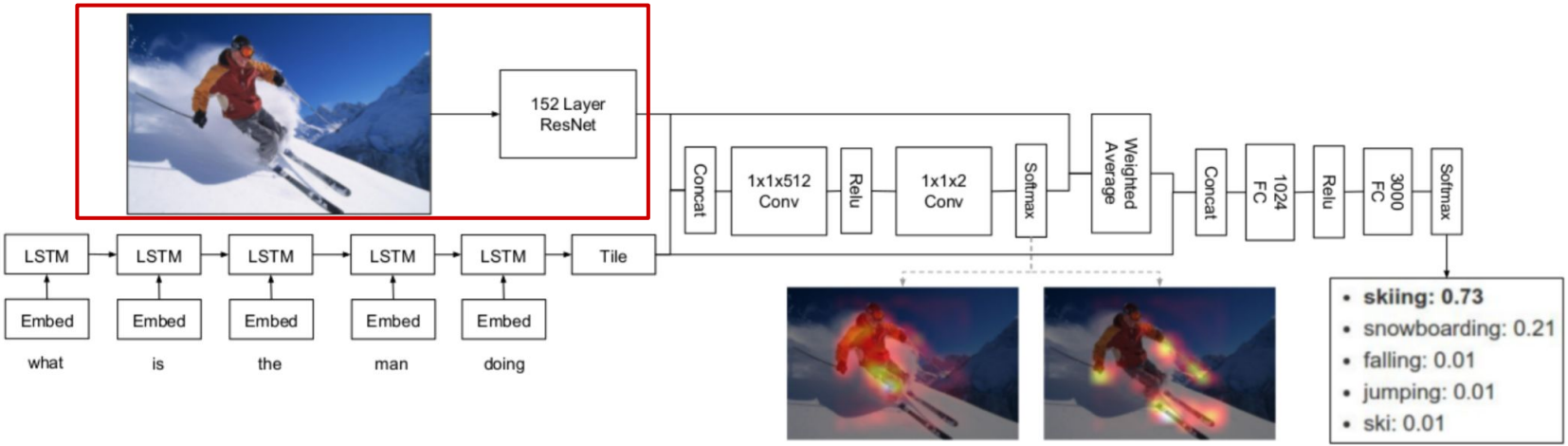
- Multi-layer perceptron
  - Input: attention feature, question feature
  - Output: logit, whose dimension is 3000 (The number of potential answers)



Putting everything together

- V = FeatureNet(image)
- Logit = VQA(v, q, q_len)
- Let's play with the VQA moel

FeatureNet

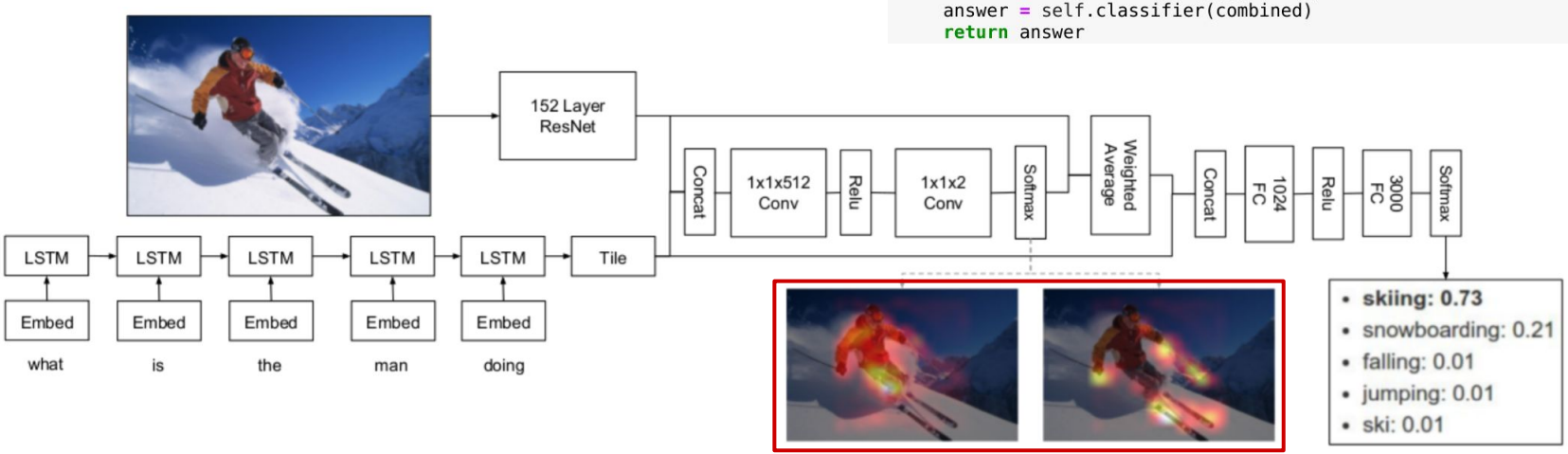


Attention Visualization

- Visualizing attention is useful to understand model's behaviour
  - Visualize attention image
  - Useful tip for keeping attention during inference

VQANet

```
def forward(self, v, q, q_len):
    q = self.text(q, list(q_len.data))

    v = v / (v.norm(p=2, dim=1, keepdim=True).expand_as(v) + 1e-8)
    a = self.attention(v, q)
    self.attention_score = a
    v = apply_attention(v, a)

    combined = torch.cat([v, q], dim=1)
    answer = self.classifier(combined)
    return answer
```



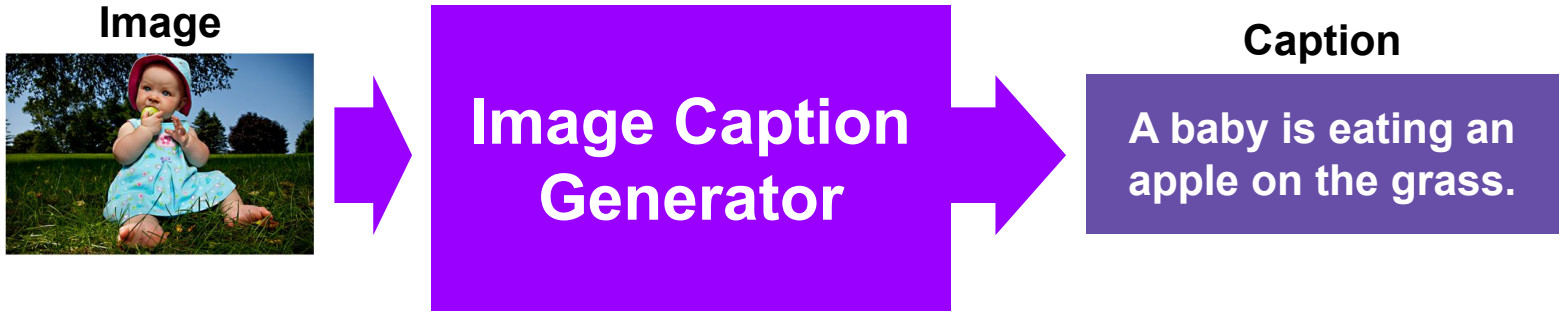
Attention Visualization

- Exercise 5
  - Implement attention_score_map function
  - Input: attention ([1, 2, 14, 14]) tensor, Output: attention_score ([2, 14, 14]) numpy array

Image Captioning

Image Caption Generation

- Goals
  - Automatically describing the content of an input image using text
  - Generating a sentence relevant to an input image



- Problem formulation
  - Finding the sentence (a sequence of words) with maximum likelihood

$$S^* = \underset{S}{\operatorname{argmax}}\, p(S|I), \quad \text{where} \quad S = \{S_1, S_2, \ldots, S_N\}$$

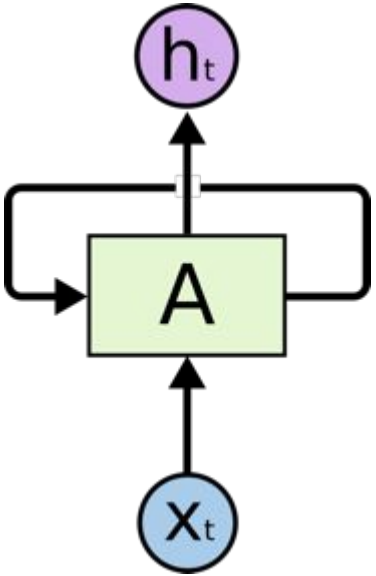
Prerequisite

Recurrent Neural Networks (RNNs)

- A family of neural networks for processing sequential data
  - Have loops in their architecture
  - Perform the same task for every element of a sequence, with the output being depended on the previous computations
  - Have a “memory” which captures information about what has been calculated so far

$$\boldsymbol{h}^{(t)} = f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right)$$

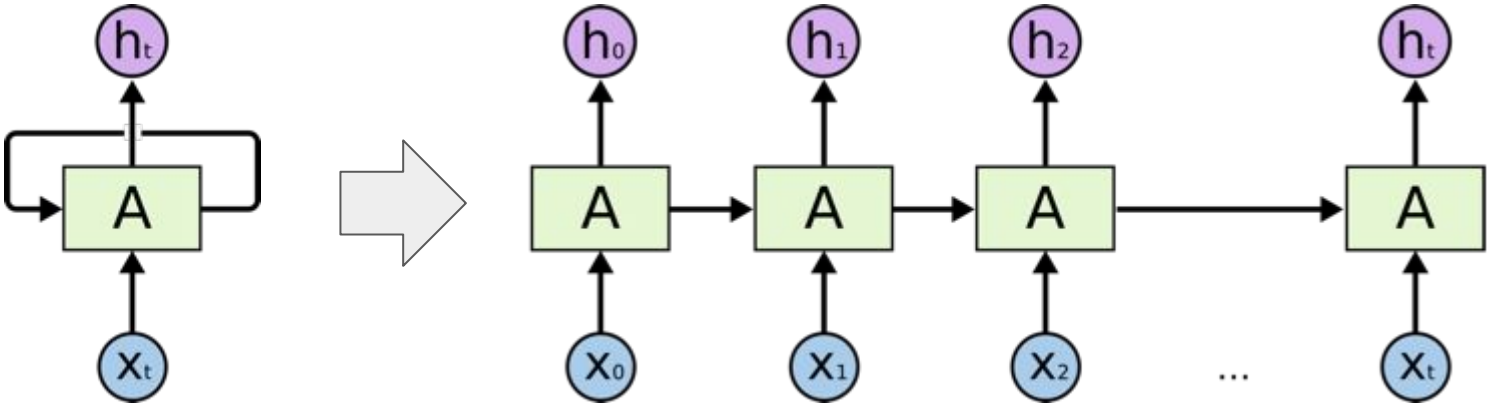
- Applications
  - Natural language processing
  - Speech recognition
  - Video processing



http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Recurrent Neural Networks (RNNs)

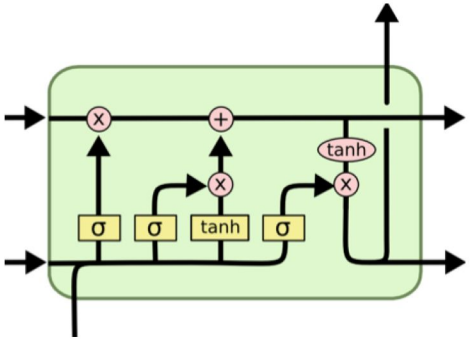
- Parameter sharing
  - Extends the model to examples of different forms such as different lengths
  - Generalizes across such heterogeneous examples
- Network architecture
  - Neural network with loops
  - Unfolding the network: useful to understand and train the network



http://colah.github.io/posts/2015-08-Understanding-LSTMs/

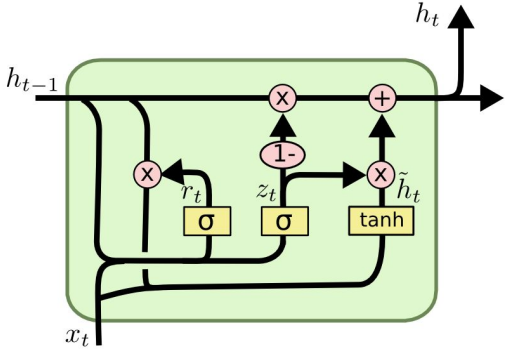
Recurrent Neural Networks (RNNs)

- Long Short Term Memory (LSTM)



$$
\begin{aligned}
f_t &= \sigma(W_f[h_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i[h_{t-1}, x_t] + b_i) \\
\tilde{C}_t &= \tanh(W_C[h_{t-1}, x_t] + b_C \\
C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t \\
o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o \\
h_t &= o_t * \tanh(C_t)
\end{aligned}
$$

- Gated Recurrent Unit (GRU)



$$
\begin{aligned}
z_t &= \sigma(W_z[h_{t-1}, x_t]) \\
r_t &= \sigma(W_r[h_{t-1}, x_t]) \\
\tilde{h}_t &= \tanh(W[r_t * h_{t-1}, x_t]) \\
h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t
\end{aligned}
$$

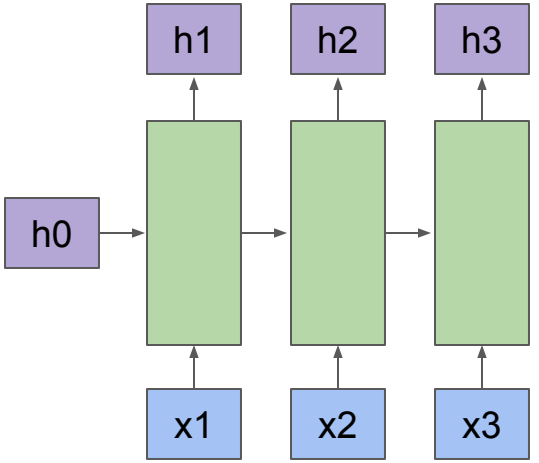
RNN in Pytorch

- 2 types of implementation
  - GRU examples
  - RNNCell
    - https://pytorch.org/docs/master/generated/torch.nn.GRUCell.html
  - RNN
    - https://pytorch.org/docs/master/generated/torch.nn.GRU.html

RNN vs RNNCell in Pytorch

- RNNCell
  - Operation for one time step

```
>>> rnn = nn.GRUCell(10, 20)          (D_I, D_h)
>>> input = torch.randn(6, 3, 10)     (L, B, D_I)
>>> hx = torch.randn(3, 20)           (B, D_h)
>>> output = []
>>> for i in range(6):
        hx = rnn(input[i], hx)
        output.append(hx)
```



RNN vs RNNCell in Pytorch

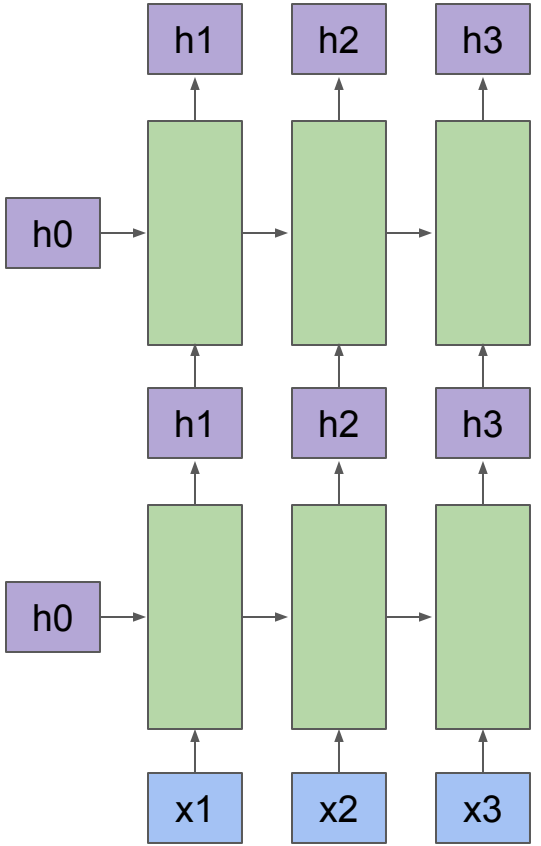
- RNN
  - Operation for L time step

```
>>> rnn = nn.GRU(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```

(D_l, D_h, num_layer)

(L, B, D_l)

(num_layer, B, D_h)



RNN vs RNNCell in Pytorch

- RNN
  - Operation for L time step

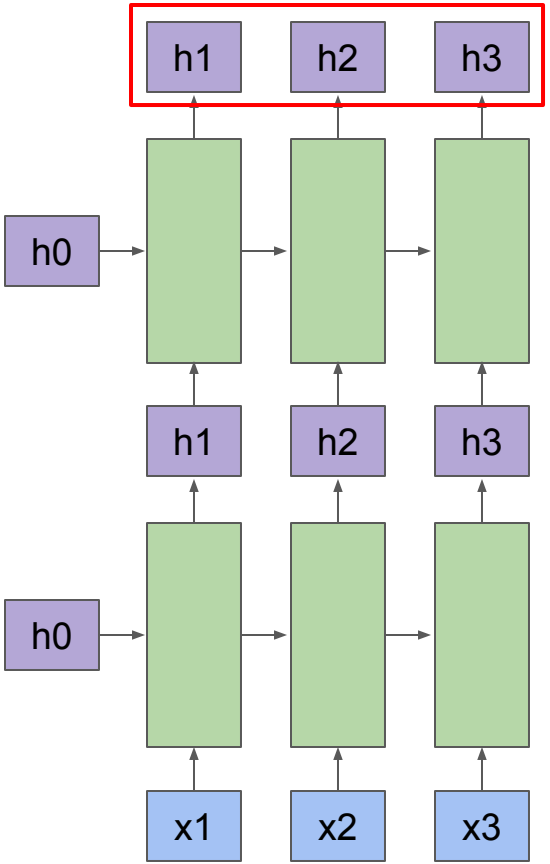
```
>>> rnn = nn.GRU(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```

(D_l, D_h, num_layer)

(L, B, D_l)

(num_layer, B, D_h)

(L, B, D_h * num_direction)



RNN vs RNNCell in Pytorch

- RNN
  - Operation for L time step

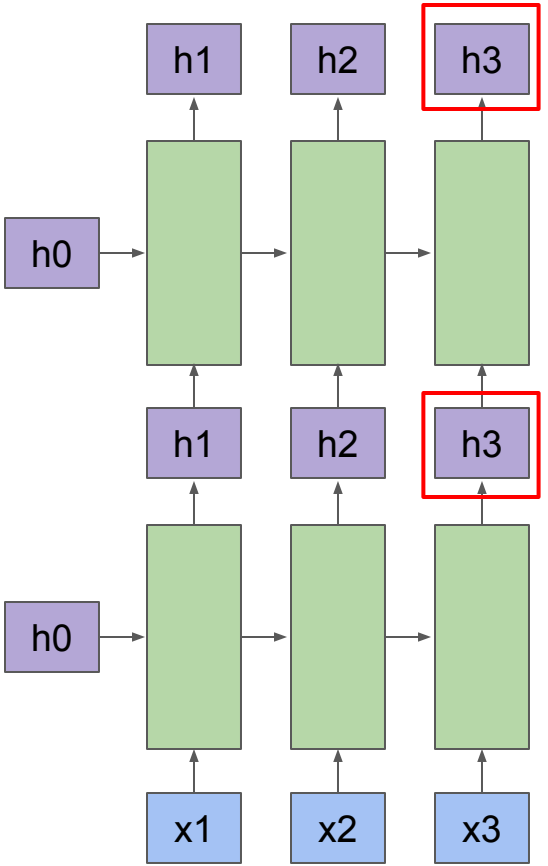
```
>>> rnn = nn.GRU(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```

(D_l, D_h, num_layer)

(L, B, D_l)

(num_layer, B, D_h)

(num_layers * num_direction, B, D_h)

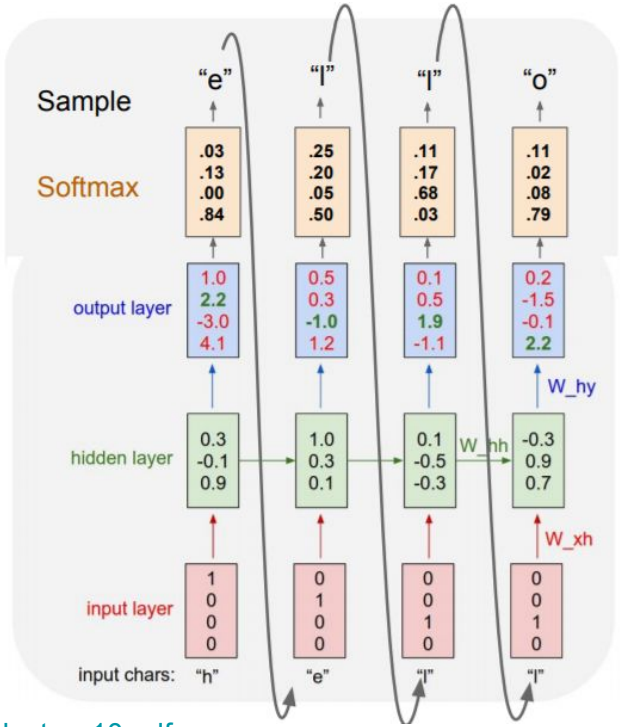


Language Model

Example:
Character-level
Language Model
Sampling

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model



Reference: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

Image Captioning Model Structure

Image Captioning

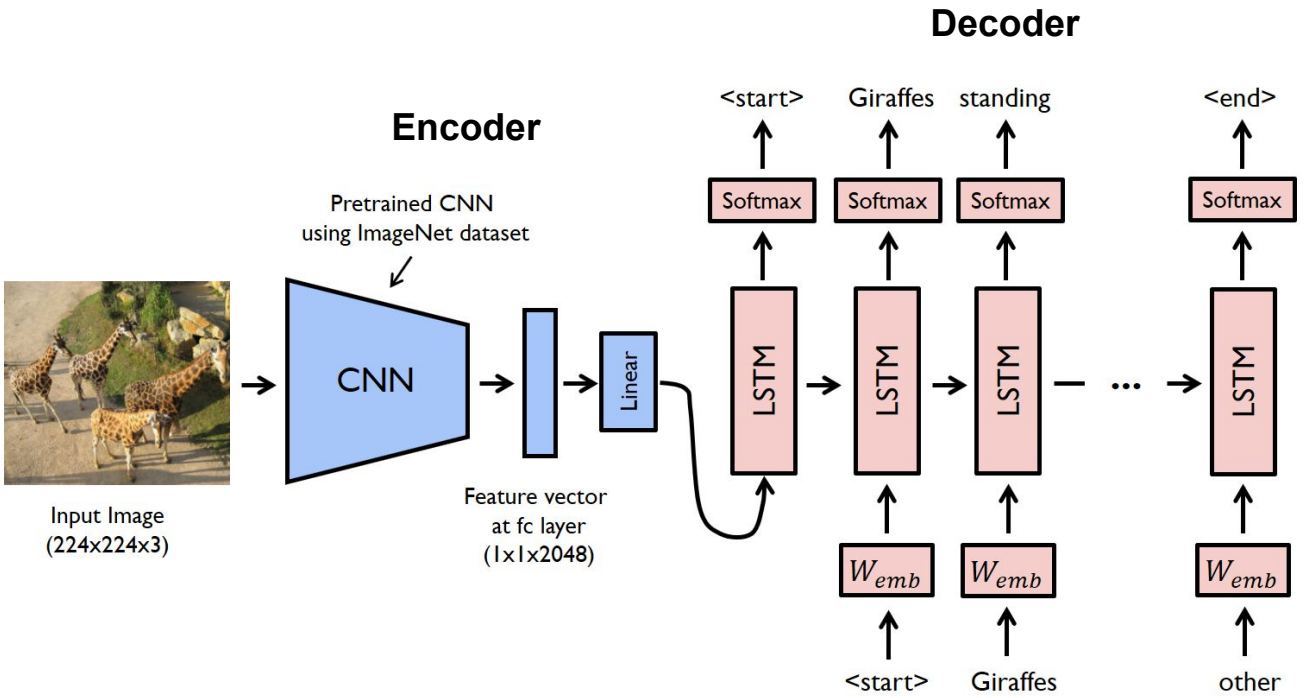
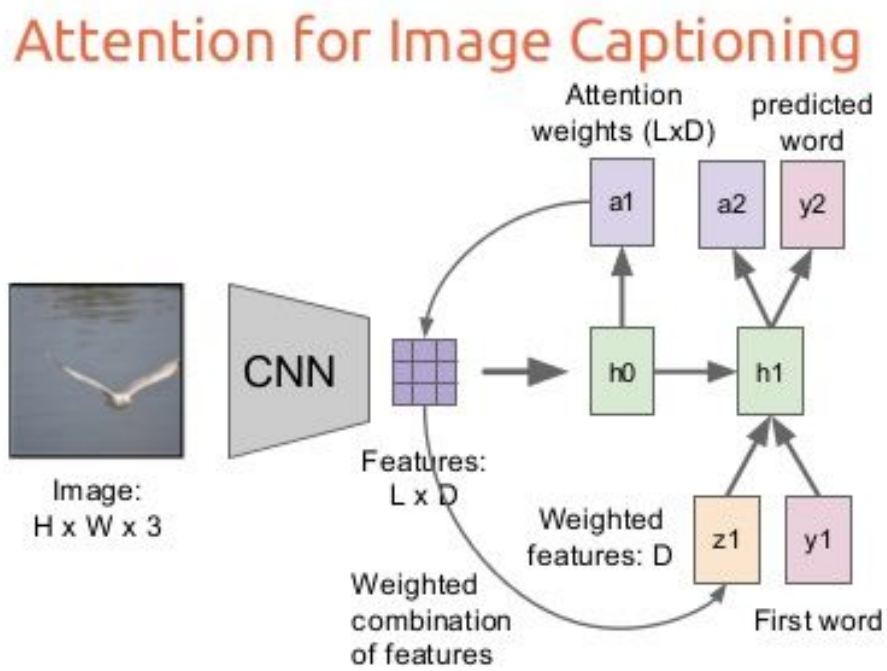


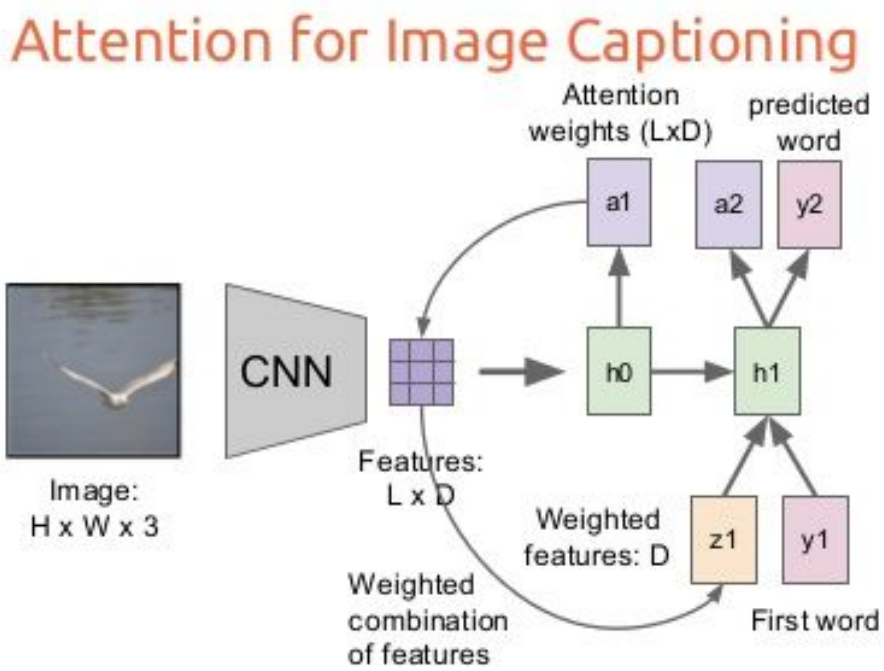
Image Captioning with Attention



Reference: https://www.slideshare.net/xavigiro/deep-learning-for-computer-vision-attention-models-upc-2016

Slide Credit: CS231n

Image Captioning with Attention



$$e_{ti} = f_{\text{att}}(\mathbf{a}_i, \mathbf{h}_{t-1})$$

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{k=1}^{L} \exp(e_{tk})}.$$

Reference: https://www.slideshare.net/xavigiro/deep-learning-for-computer-vision-attention-models-upc-2016

Slide Credit: CS231n

Attention module

```
class Attention(nn.Module):
    """
    Attention Network.
    """


    def __init__(self, encoder_dim, decoder_dim, attention_dim):
        """
        :param encoder_dim: feature size of encoded images
        :param decoder_dim: size of decoder's RNN
        :param attention_dim: size of the attention network
        """
        super(Attention, self).__init__()
        self.encoder_att = nn.Linear(encoder_dim, attention_dim)  # linear layer to transform encoded image
        self.decoder_att = nn.Linear(decoder_dim, attention_dim)  # linear layer to transform decoder's output
        self.full_att = nn.Linear(attention_dim, 1)  # linear layer to calculate values to be softmax-ed
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)  # softmax layer to calculate weights
```

$$e_{ti} = f_{\text{att}}(\mathbf{a}_i, \mathbf{h}_{t-1}) \tag{4}$$

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{k=1}^{L} \exp(e_{tk})}. \tag{5}$$

Exercise 1

- Complete blank lines in forward() of Encoder
- There are the output shape of the output Tensor in comment

```
def forward(self, images):
    """
    Forward propagation.

    :param images: images, a tensor of dimensions (batch_size, 3, image_size, image_size)
    :return: encoded images
    """

    ############################## Fill in blank lines ###################################

    # Fill in the empty line to complete following 'TODO'
    # TODO: complete fowarding path of Encoder network and calculate final 'output' feature
    # 1. calculate 'out' using 'images' & 'self.resnet'
    # 2. calculate 'out' using 'out' & 'self.adaptive_pool'
    # 3. modify shape of 'out' by using tensor.permute() function


    out =   # (batch_size, 2048, image_size/32, image_size/32)
    out =   # (batch_size, 2048, encoded_image_size, encoded_image_size)
    out =   # (batch_size, encoded_image_size, encoded_image_size, 2048)
    return out
```

Exercise 2

- Complete blank lines in forward() of Attention
- There are the output shape of the output Tensor in comment

```
def forward(self, encoder_out, decoder_hidden):
    """
    Forward propagation.

    :param encoder_out: encoded images, a tensor of dimension (batch_size, num_pixels, encoder_dim)
    :param decoder_hidden: previous decoder output, a tensor of dimension (batch_size, decoder_dim)
    :return: attention weighted encoding, weights
    """

    ############################## Fill in blank lines ###################################

    # Fill in the empty line to complete following 'TODO'
    # TODO: calculate attention_weighted_encoding (context vector) and alpha (weight of each pixel)
    # 1. calculate 'embedding1' using self.encoder_att & encoder_output
    # 2. calculate 'embedding2' using self.decoder_att & decoder hidden
    # 3. calculate 'alpha' using self.softmax & att
    # 4. calculate 'attention_weighted_encoding' using encoder_out & alpha
    # hint for 4: you can use tensor.unsqueeze(dim) to add new dimension



    embedding1 =                                # (batch_size, num_pixels, attention_dim)
    embedding2 =                                # (batch_size, attention_dim)
    att = self.full_att(self.relu(embedding1 + embedding2.unsqueeze(1))).squeeze(2)  # (batch_size, num_pixels)
    alpha =                                     # (batch_size, num_pixels)
    attention_weighted_encoding =               # (batch_size, encoder_dim)

    return attention_weighted_encoding, alpha
```

Decoder module

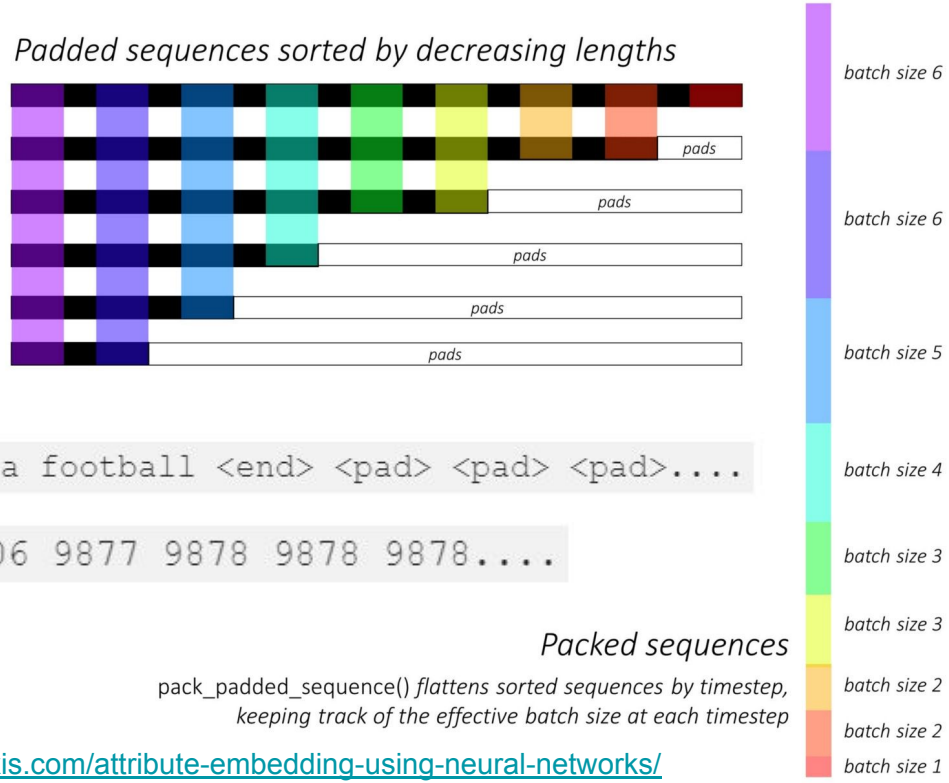
```
def __init__(self, attention_dim, embed_dim, decoder_dim, vocab_size, encoder_dim=2048, dropout=0.5):
    """
    :param attention_dim: size of attention network
    :param embed_dim: embedding size
    :param decoder_dim: size of decoder's RNN
    :param vocab_size: size of vocabulary
    :param encoder_dim: feature size of encoded images
    :param dropout: dropout
    """
    super(DecoderWithAttention, self).__init__()

    self.encoder_dim = encoder_dim
    self.attention_dim = attention_dim
    self.embed_dim = embed_dim
    self.decoder_dim = decoder_dim
    self.vocab_size = vocab_size
    self.dropout = dropout

    self.attention = Attention(encoder_dim, decoder_dim, attention_dim)  # attention network

    self.embedding = nn.Embedding(vocab_size, embed_dim)  # embedding layer
    self.dropout = nn.Dropout(p=self.dropout)
    self.decode_step = nn.LSTMCell(embed_dim + encoder_dim, decoder_dim, bias=True)  # decoding LSTMCell
    self.init_h = nn.Linear(encoder_dim, decoder_dim)  # linear layer to find initial hidden state of LSTMCell
    self.init_c = nn.Linear(encoder_dim, decoder_dim)  # linear layer to find initial cell state of LSTMCell
    self.f_beta = nn.Linear(decoder_dim, encoder_dim)  # linear layer to create a sigmoid-activated gate
    self.sigmoid = nn.Sigmoid()
    self.fc = nn.Linear(decoder_dim, vocab_size)  # linear layer to find scores over vocabulary
    self.init_weights()  # initialize some layers with the uniform distribution
```

Training


```
# Batches
for i, (imgs, caps, caplens) in enumerate(train_loader):
    data_time.update(time.time() - start)

    # Move to GPU, if available
    imgs = imgs.to(device)
    caps = caps.to(device)
    caplens = caplens.to(device)

    # Forward prop.
    imgs = encoder(imgs)
    scores, caps_sorted, decode_lengths, alphas, sort_ind = decoder(imgs, caps, caplens)

    # Since we decoded starting with <start>, the targets are all words after <start>, up to <end>
    targets = caps_sorted[:, 1:]

    # Remove timesteps that we didn't decode at, or
    # pack_padded_sequence is an easy trick to do this
    scores, _ = pack_padded_sequence(scores, decode_lengths, batch_first=True)
    targets, _ = pack_padded_sequence(targets, decode_lengths, batch_first=True)

    # Calculate loss
    loss = criterion(scores, targets)

    # Add doubly stochastic attention regularization
    loss += alpha_c * ((1. - alphas.sum(dim=1)) ** 2).mean()
```

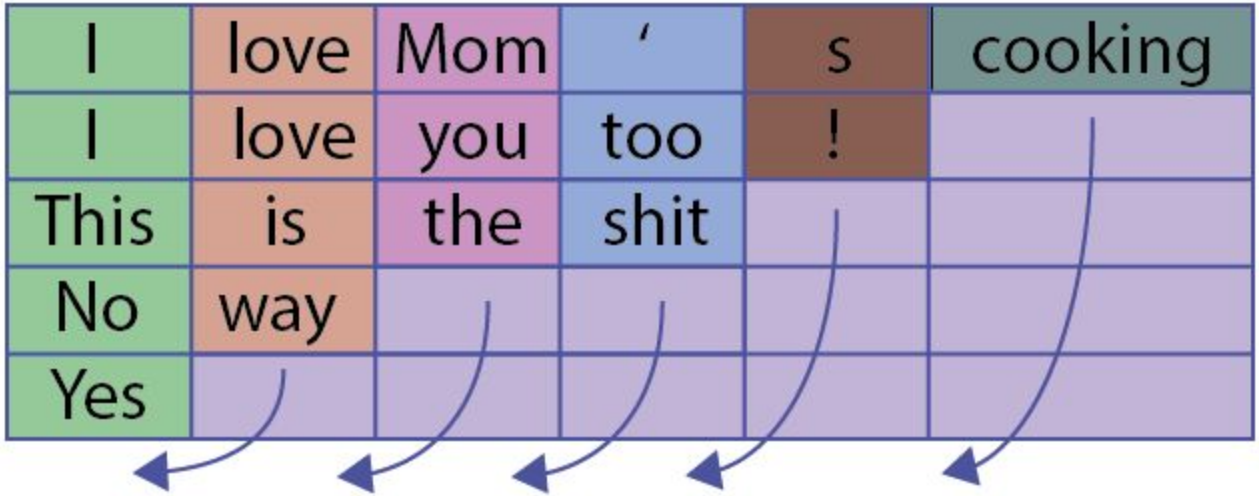
```
    # We won't decode at the <end> position, since we've finished generating as soon
    # So, decoding lengths are actual lengths - 1
    decode_lengths = (caption_lengths - 1).tolist()
```

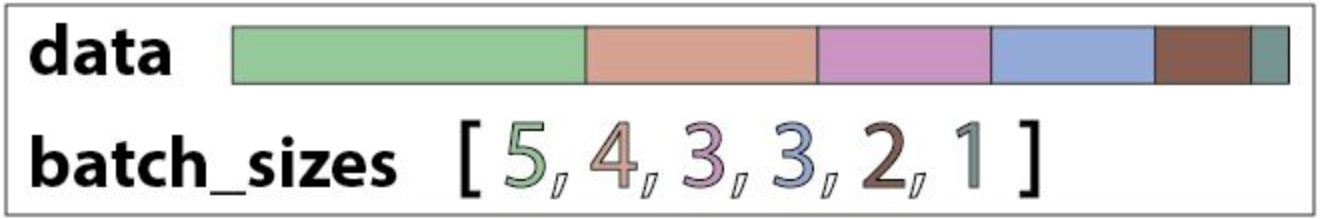
What is pack_padded_sequence in Pytorch?



Reference: http://praelexis.com/attribute-embedding-using-neural-networks/

Why pack_padded_sequence?

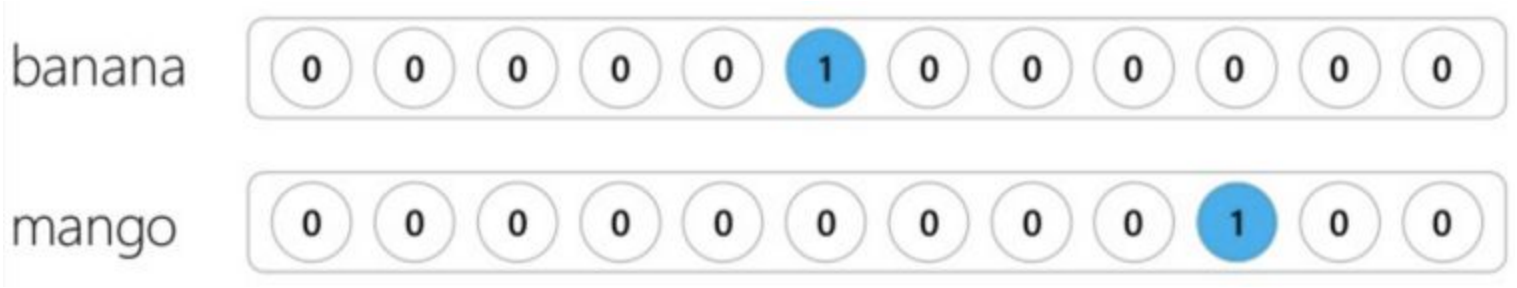




Reference: https://simonjisu.github.io/nlp/2018/07/05/packedsequence.html

Word Embedding

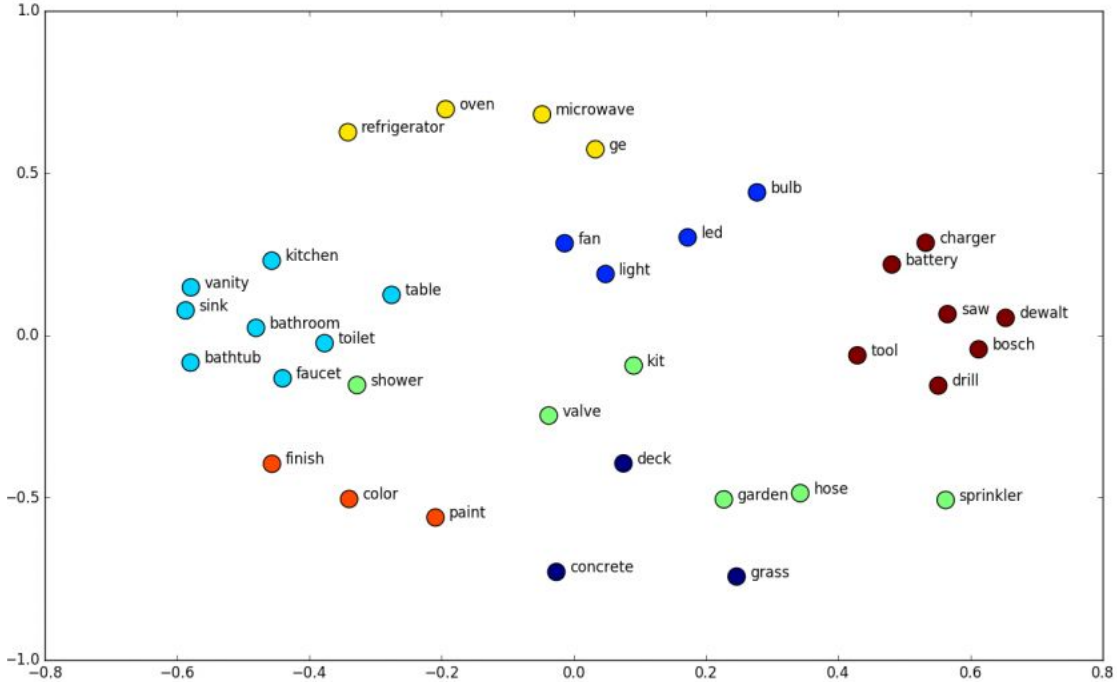
- Traditional word representation method: Bag of Words
  - Same distance between all kinds of two words



Reference: https://mc.ai/beyond-word-embeddings-part-2-word-vectors-nlp-modeling-from-bow-to-bert/

Word Embedding

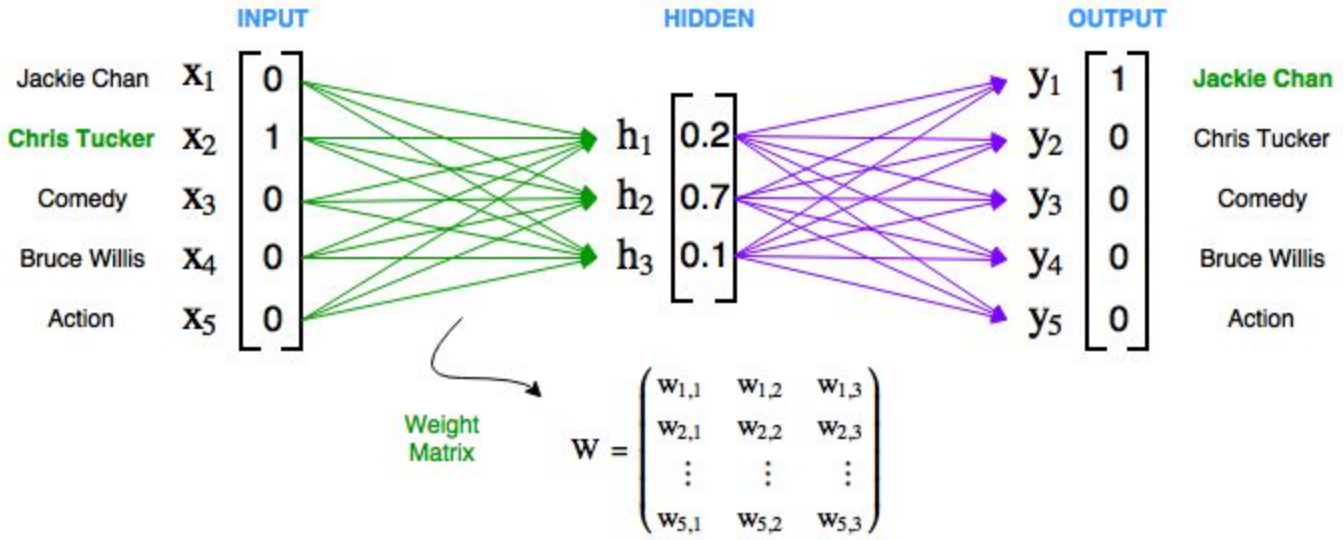
- Word embedding methods learn a **real-valued vector representation** for a predefined fixed sized vocabulary from a corpus of text.



Reference: https://www.shanelynn.ie/get-busy-with-word-embeddings-introduction/

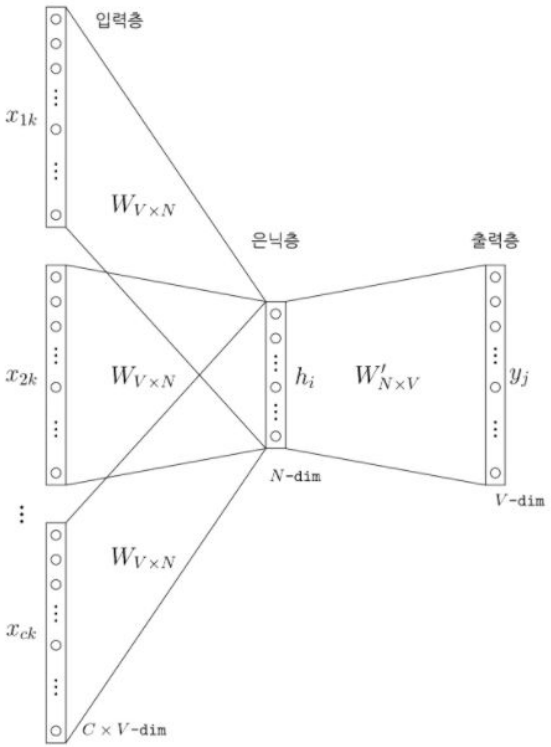
Word Embedding

- How to learn word embedding?



Reference: http://praelexis.com/attribute-embedding-using-neural-networks/

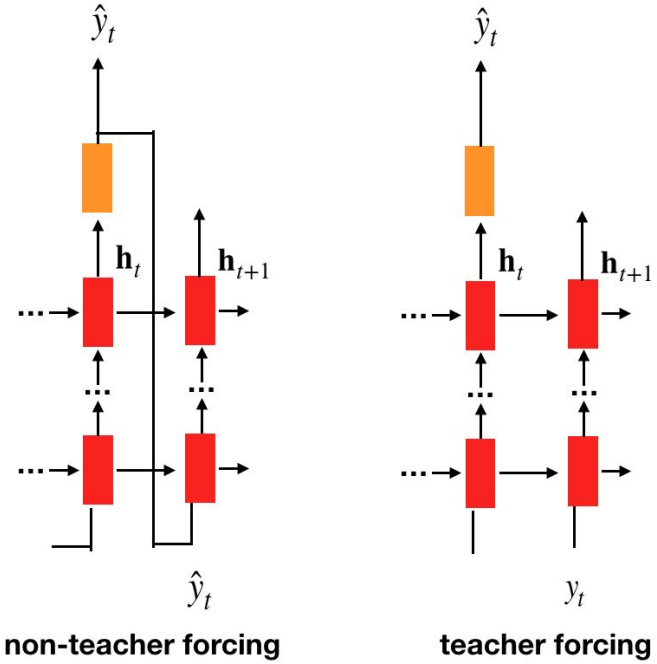
Word Embedding



Reference: https://mc.ai/beyond-word-embeddings-part-2-word-vectors-nlp-modeling-from-bow-to-bert/

```
class DecoderWithAttention(nn.Module):
    """
    Decoder.
    """

    def __init__(self, attention_dim, embed_dim, decoder_dim, vocab_size, encoder_dim=2048, dropout=0.5):
        """
        :param attention_dim: size of attention network
        :param embed_dim: embedding size
        :param decoder_dim: size of decoder's RNN
        :param vocab_size: size of vocabulary
        :param encoder_dim: feature size of encoded images
        :param dropout: dropout
        """
        super(DecoderWithAttention, self).__init__()

        self.encoder_dim = encoder_dim
        self.attention_dim = attention_dim
        self.embed_dim = embed_dim
        self.decoder_dim = decoder_dim
        self.vocab_size = vocab_size
        self.dropout = dropout

        self.attention = Attention(encoder_dim, decoder_dim, attention_dim)  # attention network

        self.embedding = nn.Embedding(vocab_size, embed_dim)  # embedding layer
        self.dropout = nn.Dropout(p=self.dropout)
        self.decode_step = nn.LSTMCell(embed_dim + encoder_dim, decoder_dim, bias=True)  # decoding LSTMCell
        self.init_h = nn.Linear(encoder_dim, decoder_dim)  # linear layer to find initial hidden state of LSTMCell
        self.init_c = nn.Linear(encoder_dim, decoder_dim)  # linear layer to find initial cell state of LSTMCell
        self.f_beta = nn.Linear(decoder_dim, encoder_dim)  # linear layer to create a sigmoid-activated gate
        self.sigmoid = nn.Sigmoid()
        self.fc = nn.Linear(decoder_dim, vocab_size)  # linear layer to find scores over vocabulary
        self.init_weights()  # initialize some layers with the uniform distribution
```

Teacher Forcing (only for training)



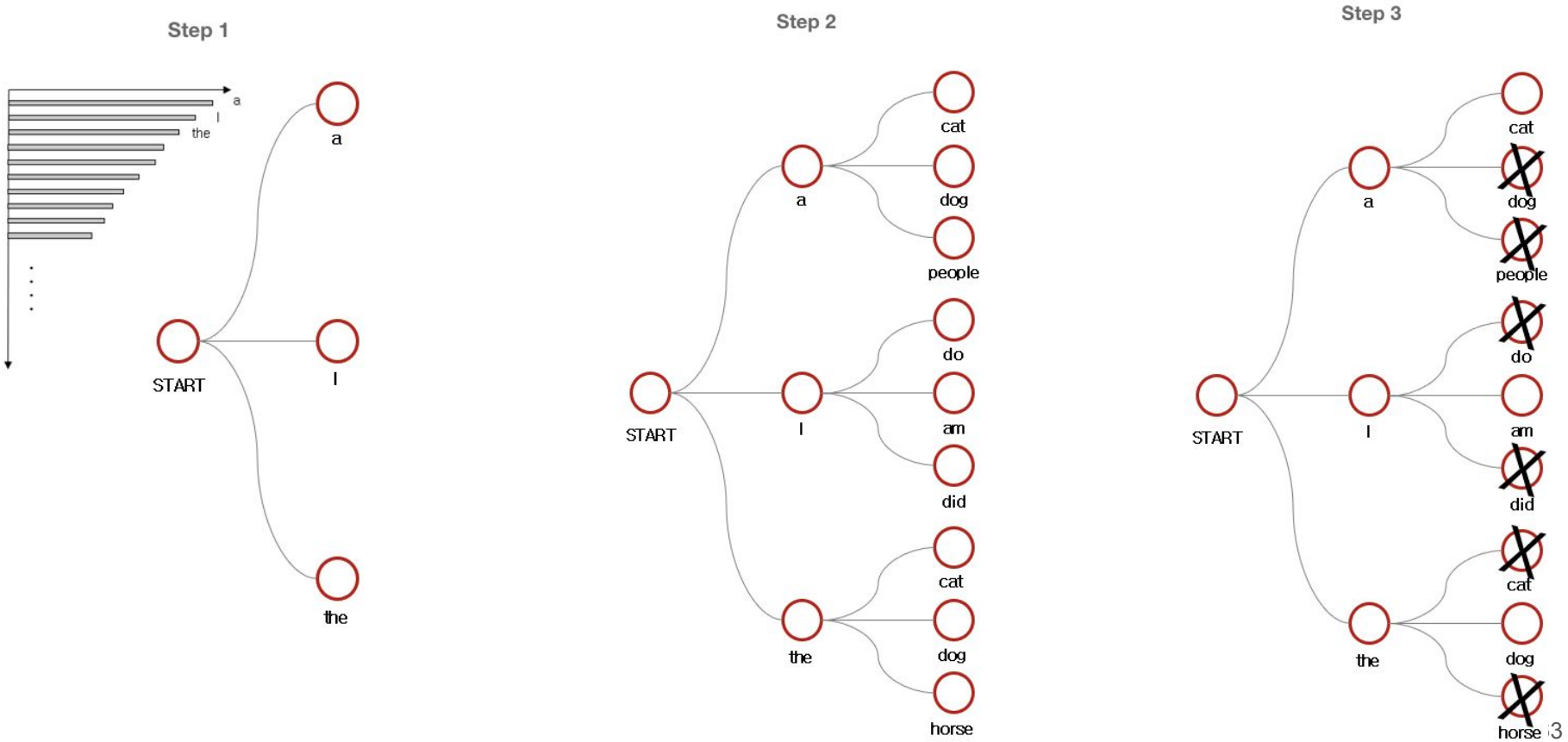
Reference: http://cnyah.com/2017/11/01/professor-forcing/

Teacher Forcing (only for training)

```
for i, (imgs, caps, caplens) in enumerate(train_loader):
    data_time.update(time.time() - start)

    # Move to GPU, if available
    imgs = imgs.to(device)
    caps = caps.to(device)
    caplens = caplens.to(device)

    # Forward prop.
    imgs = encoder(imgs)
    scores, caps_sorted, decode_lengths, alphas, sort_ind = decoder(imgs, caps, caplens)

    # Since we decoded starting with <start>, the targets are all words after <start>, up to <end>
    targets = caps_sorted[:, 1:]

    # Remove timesteps that we didn't decode at, or are pads
    # pack_padded_sequence is an easy trick to do this
    scores, _ = pack_padded_sequence(scores, decode_lengths, batch_first=True)
    targets, _ = pack_padded_sequence(targets, decode_lengths, batch_first=True)

    # Calculate loss
    loss = criterion(scores, targets)
```

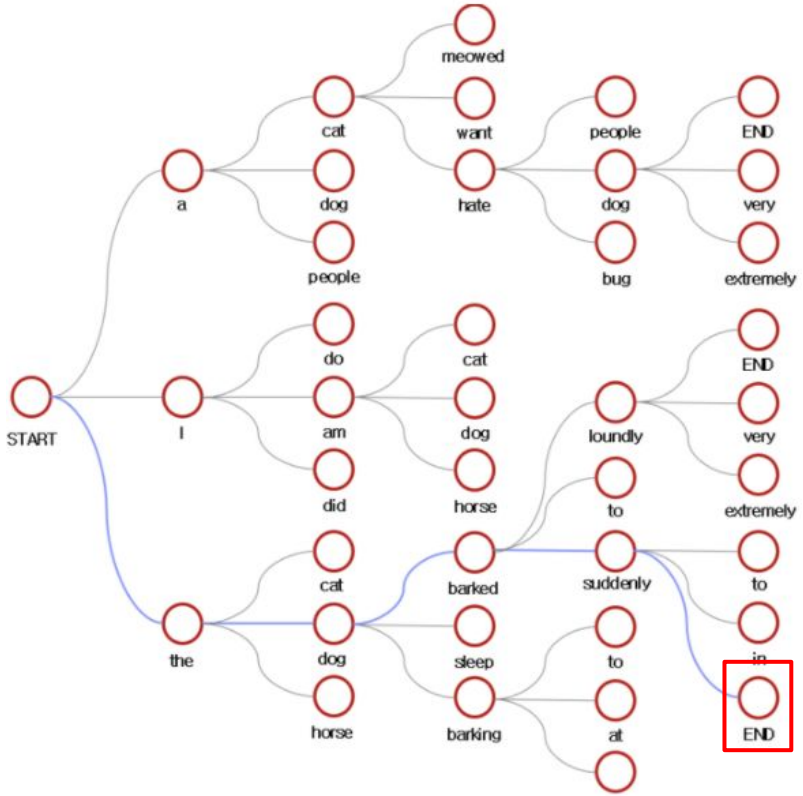
```
# Embedding
embeddings = self.embedding(encoded_captions)  # (batch_size, max_caption_length, embed_dim)

# Initialize LSTM state
h, c = self.init_hidden_state(encoder_out)  # (batch_size, decoder_dim)

# We won't decode at the <end> position, since we've finished generating as soon as we generate <end>
# So, decoding lengths are actual lengths - 1
decode_lengths = (caption_lengths - 1).tolist()

# Create tensors to hold word predicion scores and alphas
predictions = torch.zeros(batch_size, max(decode_lengths), vocab_size).to(device)
alphas = torch.zeros(batch_size, max(decode_lengths), num_pixels).to(device)

# At each time-step, decode by
# attention-weighing the encoder's output based on the decoder's previous hidden state output
# then generate a new word in the decoder with the previous word and the attention weighted encoding
for t in range(max(decode_lengths)):
    batch_size_t = sum([l > t for l in decode_lengths])
    attention_weighted_encoding, alpha = self.attention(encoder_out[:batch_size_t],
                                                        h[:batch_size_t])
    gate = self.sigmoid(self.f_beta(h[:batch_size_t]))  # gating scalar, (batch_size_t, encoder_dim)
    attention_weighted_encoding = gate * attention_weighted_encoding
    h, c = self.decode_step(
        torch.cat([embeddings[:batch_size_t, t, :], attention_weighted_encoding], dim=1),
        (h[:batch_size_t], c[:batch_size_t]))  # (batch_size_t, decoder_dim)
    preds = self.fc(self.dropout(h))  # (batch_size_t, vocab_size)
    predictions[:batch_size_t, t, :] = preds
    alphas[:batch_size_t, t, :] = alpha

return predictions, encoded_captions, decode_lengths, alphas, sort_ind
```

Beam Search (only for inference)



Reference: https://blog.naver.com/sooftware/221809101199

Beam Search (only for inference)



Reference: https://blog.naver.com/sooftware/221809101199

Exercise 3

- Complete blank lines in ‘caption_image_beam_search’ function
- There are the output shape of the output Tensor in comment

```
def caption_image_beam_search(encoder, decoder, image_path, word_map, beam_size=3):
    """
    Reads an image and captions it with beam search.

    :param encoder: encoder model
    :param decoder: decoder model
    :param image_path: path to image
    :param word_map: word map
    :param beam_size: number of sequences to consider at each decode-step
    :return: caption, weights for visualization
    """

    k = beam_size
    vocab_size = len(word_map)

    # Read image and process
    img = imread(image_path)
    if len(img.shape) == 2:
        img = img[:, :, np.newaxis]
        img = np.concatenate([img, img, img], axis=2)
    img = imresize(img, (256, 256))
    img = img.transpose(2, 0, 1)
    img = img / 255.
    img = torch.FloatTensor(img).to(device)
    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                     std=[0.229, 0.224, 0.225])
    transform = transforms.Compose([normalize])
    image = transform(img)  # (3, 256, 256)

    # Encode
    ############################## Fill in blank lines ###################################
    image =   # (1, 3, 256, 256), Use .unsqueeze(k) function to add new dimension into 'k'th dim
```

Practice

1. Fill in blank lines in forward() of Encoder
2. Fill in blank lines in forward() of Attention
3. In 1, 2 case, you can check yourself by running test code
4. Fill in blank lines in ‘caption_image_beam_search’
5. Put any images in ‘imgs/’ directory & run ipython file
6. Try with different ‘beam size’ value & compare the result (ex. 1, 5, 25, 125…)

End of Image Captioning

Reference: Xu et al, “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention”, ICML 2015