TensorFlow & AI

Donglin Jia

December 24, 2020

Contents

1 Introduction **2**
- 1.1 Background Information . . . 2
- 1.2 Linear Function . . . 3

2 Image Processing **4**
- 2.1 Image Recognition . . . 4
- 2.2 Convolutional Neural Networks . . . 6

3 Enrich Training Data **9**
- 3.1 Data Augmentation . . . 9
- 3.2 Three types of data augmentation . . . 10

4 Natural Language Processing **12**
- 4.1 Tokenization . . . 12
- 4.2 Turning sentences into data . . . 13
- 4.3 Embedding . . . 15

5 Generating Sentences **16**
- 5.1 Recurrent Neural Network . . . 16
- 5.2 Long Short-Term Memory . . . 18
  - 5.2.1 Core Idea Behind LSTMs . . . 19
  - 5.2.2 Step-by-Step LSTM Walk Through . . . 19
  - 5.2.3 Variants on LSTM . . . 21

Chapter 1

Introduction

1.1 Background Information

- Normal programming is a function that takes inputs and rules and gives out outputs
- Machine learning is a model that takes inputs and outputs and gives out rules

Important concepts:

- Activation Function:
  - Activation Functions are mathematical equations that determine the output of a neural network.
  - It is attached to each neuron and determine if it should be activated or not, based on whether each neuron's input is relevant for the model's prediction.
  - Here is a good article I found: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/
- Backpropagation:
  - After forward propagation, we get an output which is the predicted value. By using loss function
- Behavioral Cloning

1.2 Linear Function

```
1   import numpy as np
2
3   model = keras.Sequential(
4       [keras.layers.Dense(units=1, input_shape=[1]
5       )])
6   model.compile(optimizer='sgd', loss='mean_squared_error')
7
8   xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
9   ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)
10
11  model.fit(xs, ys, epochs=500)
12
13  print(model.predict([10.0]))
14
```

Listing 1.1: model example

Details Explanation:

- A model is a trained neutral network. In this example, we have simplest possible neutral network. A single layer indicated by the keras.layers.Dense code. That is line #4
- That layer has only one single neuron in it. That is **units=1**
- There is only one feed to this neural network, which is indicated by **input_shape=[1]**
- Compiling model needs two function – loss function and optimizer.
  - Model needs to make a guess about the relationship between the numbers.
  - Then loss function will be used to calculate the accuracy for that guess.
  - Lastly, using optimizer function to generate a new guess..
  - By doing this way, the result will get closer and closer
- Then we are using **fit** function to train the model by 500 times that is **epochs=500**. In other word, line #11 says fit the Xs to the Ys and try 500 times.
- Finally, model.predict can only provide a value that is very close to the result instead of the correct result.

Chapter 2

Image Processing

2.1 Image Recognition

The library we are going to use to help train model is **Fashion MNIST** where includes 70k images and 10 categories with 28x28px

```
import tensorflow as tf
from tensorflow import keras

fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels),
    (test_images, test_labels) = fashion_mnist.load_data()

model = keras.models.Sequential([
        keras.layers.Flatten(input_shape=(28,28)),
        keras.layers.Dense(1024, activation=tf.nn.relu),
        keras.layers.Dense(10, activation=tf.nn.softmax)
    ])

model.compile(optimizer = tf.optimizers.Adam(),
              loss = 'sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=10)

# load the trained model
# model = tf.keras.models.load_model('imageRecognition.h5')


# save the trained model
# model.save('imageRecognition.h5')

test_loss, test_acc = model.evaluate(test_images, test_labels)

```

Listing 2.1: Image Recognization

Details Explanation:

- Fashion MNIST dataset has already been built into TensorFlow, so it is easy to load as code in line#9
- There are 60000 images to be a train sets we can use for training model and 10000 to be a test sets for testing the performance for model.
- The label is a **number** indicating the class of that type of clothing.
  - number is easier for computer to process.
  - number can eliminate bias between different language. In other words, number sets a standard.
- The entire model is similar to a filter which takes 28x28 set of pixels and outputs 1 of 10 values.
  - The first layer has the input of shape 28x28 which is the size of images
  - The last layers has 10 neurons which indicates 10 different outputs.
  - In the hidden layer, there are 1024 neurons (i.e. 1024 functions with parameters inside of each) that is used to process the 28*28pixels
    - when the pixels of shoes get fed into them, one by one, that the combination of all of these functions will output the correct value.
    - Computer (AI) needs to figure out what the parameters inside of these functions to get the correct results.
    - Then extend the same approach to all of the other items of clothing in the dataset.
    - Once it has done that, it should be able to recognize the type of clothing.
  - Activation functions in layers:
    - activation=tf.nn.relu
      - *RELU* indicates reactified linear unit. It returns a value if it is greater than 0. If it has zero or less as an output, then the output will be filtered out.
    - activation=tf.nn.softmax
      - *softmax* has the effect of picking the biggest number in a set. It sets the largest output as 1 and rest is 0.

2.2 Convolutional Neural Networks

- A CNN is a deep learning algorithm which can take an input image, assign importance to various aspects in the image and be able to differentiate one from the other.
- The basic concept is that convolution is a linear operation that involves the multiplication of a set of weights with the input
  - Given that the technique was designed for two-dimensional input, the multiplication is performed between an array of input data and a two-dimensional array of weights, called a filter or a kernel.
  - The filter is smaller than the input data and the type of multiplication applied between a filter-sized patch of the input and the filter is a dot product.
  - In short, given an input (images of pixel value), provide a filter (a set of weights), and the filter is systematically applied to the input data to create a feature map.
- Filter the images before training the deep neutral network.
- After filter the images, features within the images come to the forefront and then spot those features to identify something
- A filter is simply a set of multipliers. It can be defined variously. By using different multipliers, images can be processed into different ways (only keep vertical/horizontal lines).
- Then we use ***Pooling*** to combine the rest of pixels. It groups up the pixels in the image and filters them down to a subset.
  - Pooling apply layers to streamline the underlying computation.
  - Pooling layer reduce the dimensions of the data by compiling the outputs of neuron clusters at one layer into a single neuron in the next layer
  - Pooling layer is a form of non-linear down-sampling
  - For example, 2x2 max pooling groups a pixels of 2x2 and pick the highest pixel. Then it combines with others highest pixel from other 2x2 pixels. The image can be reduced to a quarter size but the feature can still be maintained.
- The filter is learned through training. During the training, a number of randomly initialized filters will pass over the image. The result of these are fed into the next layer and matching is performed by NN.
- Overtime, the filters that gives the image outputs that give the best matches will be learned and the process is called feature extraction

We need to change the layers as follows:

```
1    model = keras.models.Sequential([
2            keras.layers.Conv2D(64,
3                                (3,3),
4                                activation='relu',
5                                input_shape=(28,28,1)),
6            keras.layers.MaxPooling2D(2,2),
7            keras.layers.Flatten(),
8            keras.layers.Dense(1024, activation=tf.nn.relu)
9            keras.layers.Dense(10, activation=tf.nn.softmax)
10       ])
11
```

Listing 2.2: Image Recognization

Details Explanation:

- Now the convolutional layer take input that is 28x28px images.
- We generate 64 filters on line 2. That is generating 64 filters and multiply each of them across the image. Then in each epoch, it will figure out which filters gave the best signals to help match the images to their labels (a.k.a. learned which parameters worked best in the dense layer).
- The max pooling compress the image and enhance the feature
- From line 1 to line 6, it can be repeated several times to really break down the image and try to learn from very abstract features.
- From this two layers, model can learn the feature of images instead of the raw patterns of pixels

Here is the code for defining the CNN for model

```
1
2    import tensorflow as tf
3
4    model = tf.keras.models.Sequential([
5            tf.keras.layers.Conv2D(
6                64,
7                (3,3),
8                activation='relu',
9                input_shape=(150, 150, 3)),
10           tf.keras.layers.MaxPooling2D(2,2),
11           tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
12           tf.keras.layers.MaxPooling2D(2,2),
13           tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
14           tf.keras.layers.MaxPooling2D(2,2),
15           tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
16           tf.keras.layers.MaxPooling2D(2,2),
17           tf.keras.layers.Flatten(),
18   # improve efficiency of neural network by throwing away some of neurons
```

```
            tf.keras.layers.Dropout(0.5),
            tf.keras.layers.Dense(512, activation='relu'),
            tf.keras.layers.Dense(3, activation='softmax')
    ])

```

Listing 2.3: rocks paper and sessors CNN

Chapter 3

Enrich Training Data

3.1 Data Augmentation

For this section, we will mainly focus on talking about data augmentation.
As we all know that, the model we trained is good at recognizing what is has seen before, but it is not so great at generalizing.
To avoid that, we need data augmentation to help the model do a better job on generalizing.

Data augmentation encompasses a wide range of techniques used to generate 'new' training samples from the original ones by applying random jitters and perturbations.

Our goal when applying data augmentation is to increase the generalizability of the model

In the context of computer vision, data augmentation lends itself naturally.

For example, we can obtain augmented data from the original images by applying simple geometric transformation, such as random:

- Translations
- Rotations
- Changes in scale
- Shearing
- Horizontal/Vertical flips

As we mentioned before, every type is associated to a class label for model to recognize.

Applying a small amount of the transformations to an input image will change tis appearance slightly, but it *does not* change the class label – thereby making data augmentation a

very natural, easy method to apply for computer vision tasks.

In keras ImageDataGenerator, it replaces the original batch with the new, randowmly tranformed batch. It *only* returns the new, transformed data.

3.2 Three types of data augmentation

Type 1: Dataset Generation and expanding an existing dataset

Training model needs a great amount of data. When there is only limited data, it is hard for us to make the model generalized.

The following steps need to be accomplished:

1. Load the origin input image from disk
2. Randomly transform the original image via a series of random translations, rotations, etc
3. Take the transformed image and write it back out to disk
4. Repeat steps 2 and 3 for N times

We cannot expect to train a NN on a small amount of data and then expect it to generalize to data it was never trained on and has never seen before.

Type 2: In-place/on-the-fly data augmentation

1. An input batch of images is presented to the *ImageDataGenerator*.
2. The *ImageDataGenerator* transforms each image in the batch by a series of random translations, rotation, etc.
3. The randomly transformed batch is then returned to the calling function.

There are two important points

1. The *ImageDataGenerator* is not returning both the original data and the transformed data – **the class only returns the randomly transformed data.**
2. This type of augmentation is done at **training time**.

The entire point of the data augmentation technique described above is to ensure that the network sees 'new' images that it has never 'seen' before at each and every epoch.

Type 3: Combining dataset generation and in-place augmentation

This type is the combination of above two types.
Here is the part of code for ImageDataGenerator for training datasets.

```
1
2   import keras_preprocessing
3   from keras_preprocessing import image
4   from keras_preprocessing.image import ImageDataGenerator
5
6   TRAINING_DIR = "/PATH/rps/"
7   # to avoid overfitting data, we apply image augmentation
8   training_datagen = ImageDataGenerator(
9       rescale = 1./255,
10      rotation_range=40,
11      width_shift_range=0.2,
12      height_shift_range=0.2,
13      shear_range=0.2,
14      zoom_range=0.2,
15      horizontal_flip=True,
16      fill_mode='nearest')
17
18  training_generator = training_datagen.flow_from_directory(
19      TRAINING_DIR,
20      target_size=(150, 150),
21      class_mode='categorical',
22      batch_size=126
23  )
24
25  history = model.fit(
26      training_generator,
27      epochs=25,
28      steps_per_epoch=20,
29      validation_data=validation_generator,
30      verbose = 1)
31
32
```

Listing 3.1: Image Augmentation

Detail Explanation:

- In the training dataset, we apply image augmentation to generate 'new' images for training model (line 8 - 16)

Chapter 4

Natural Language Processing

4.1 Tokenization

Firstly, each word is made up by number of letters. By using ASCII, we can map each letter to a specific number and using the combination of this number to represent a word.

However, if two words has the same types of letters but with different orders, say listen & silent, then it is hard for us to understand sentiment of a word just by the letter in it.

Therefore, if we treat each word as a subject to encode it.

I	love	my	dog.
001	002	003	004

I	love	my	cat.
001	002	003	005

Then we can find out that:

$$\text{I love my} \rightarrow 1\ 2\ 3$$

The above two sentences has already shown some patterns between them.

```
    import tensorflow as tf
    from tensorflow import keras
    from tensorflow.keras.preprocessing.text import Tokenizer

    sentences = [
        'I love my dog', 'I love my cat'
    ]

    tokenizer = Tokenizer(num_words = 100)
    tokenizer.fit_on_texts(sentences)
    word_index = tokenizer.word_index
    print(word_index)

```

Listing 4.1: Tokenization

Details Explanation:

- Firstly, we import Tokenizer from tensorflow.keras
- Need to represent sentences as a Python array, line 5-8
- Create an instance of tokenizer object, num_words parameters is the maximum number of words to keep → we can also filter the most frequent 100 words in all the existing words
- Tokenizer goes through all the sentences and fit itself to them on line 11
- Full list of words is in tokenizer.word_index property

Note that:

- Tokenizer is smart enough to catch up some exceptions, including different notations after words, say 'dog.' vs 'dog!'. Tokenizer will not create a new token for 'dog!'.

4.2 Turning sentences into data

Applying the following line to get the sequences for provided sentences:

```
1     sequences = tokenizer.texts_to_sequences(sentences)
2
3 => Output:
4     [[4, 3, 2, 5], [4, 3, 2, 6], [7, 3, 2, 5], [8, 9, 10, 11, 12, 13, 2]]
5
```

Listing 4.2: generating sequences

However, it is not sufficient enough for us to provide data to train model, since there might be some new words that tokenizer does not provide a token for it.

The tokenizer will omit the word that it does not recognize. Therefore, a large word index is needed to handle the word that is not in the training sets.

In order to keep the length of the sequence, another token property, OOV (out of vocabulary) can be added into tokenizer:

```
1         tokenizer = Tokenizer(num_words=100, oov_token="<OOV>")
2
```

Listing 4.3: Deal with OOV

By adding that, tokenizer will create $< oov >$ token for those unrecognized words so that we keep the length of sentences successfully.

In order to provide the same length of sequences for training model, we need to use RaggedTensor to set the same length. However, padding can also be used to resolve this issue.

```
    from tensorflow.keras.preprocessing.sequence import pad_sequences

    padded = pad_sequences(sequences)

=> Output:
    [[ 0  0  0  0  0  4  3  2  5]
     [ 0  0  0  0  0  4  3  2  6]
     [ 0  0  0  0  0  7  3  2  5]
     [ 8  9 10 11 12 13 14  2 15]]

```

Listing 4.4: padding

For the **arguments** in pad_sequences:

- sequences: List of lists, where each element is a sequence.
- maxlen: Int, maximum length of all sequences.
- dtype: Type of the output sequences. To pad sequences with variable length strings, you can use ‘object‘.
- padding: String, ’pre’ or ’post’: pad either before or after each sequence.
- truncating: String, ’pre’ or ’post’: remove values from sequences larger than ‘maxlen‘, either at the beginning or at the end of the sequences.
- value: Float or String, padding value.

Note that:

- OOV is 1 and padding is 0

4.3 Embedding

A method used to represent discrete variables as continuous vectors.

- By looking at the direction of the vector, we can determine the meaning of the word.
- When extend 2D into multiple dimensions, we can determine if the word is toxic or sarcastic or not.
- During the training, model can learn what the direction in these multi-dimensional spaces each word look like.
  - e.g. words have sarcastic meaning have a strong component in the sarcastic direction.
  - e.g. words have non-sarcastic have a component in the non-sarcastic direction.
- After fully trained network, it can look up the vector of each word and sum it up to predict if this sentence is sarcastic or not.

We define embedding layer as follows:

```
    model = tf.keras.Sequential([
            tf.keras.layers.Embedding(vocab_size,
                                      embedding_dim,
                                      input_length=maxlen),
            tf.keras.layers.GlobalAveragePooling1D(),
            tf.keras.layers.Dense(24, activation='relu'),
            tf.keras.layers.Dense(1, activation='sigmoid')
    ])

```

Details Explanation:

- vocab_size represents how many different types of items will be represented by vector.
- embedding_dim represents how many different dimensions will be in each vector.
- input_length represents the maximum length of each item.
- GlobalAveragePooling1D is simply summing up vector and pass the value to the Dense layer.
- In terms of this case (classifier for text), the order in which the words appear in the sentence does not really matter. What determined the sentiment was the vector that resulted in adding up all of the individual vectors for the individual word. The direction of that vector roughly gave out the sentiments.

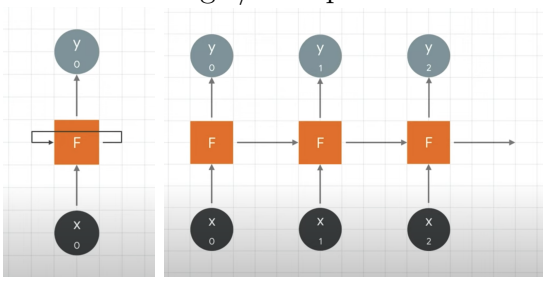
Chapter 5

Generating Sentences

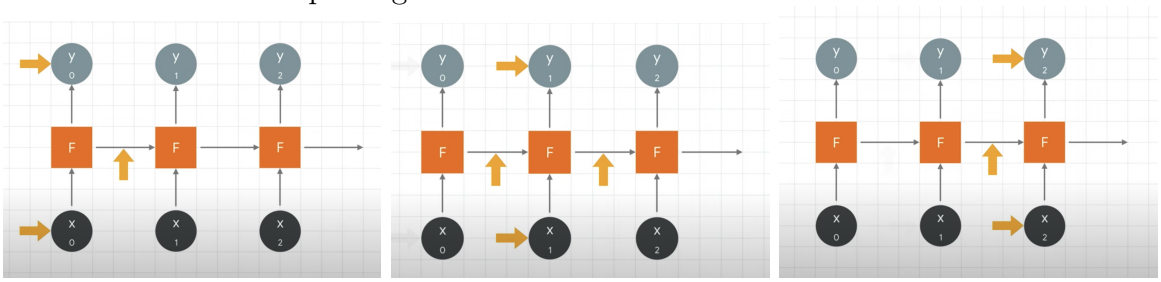
5.1 Recurrent Neural Network

- This type of Neural Network takes sequences of data into account when it is learning.
- RNN remembers the past and it's decisions are influenced by what it has learnt from the past. They remember things learnt from prior input(s) while generating output(s).
- During training, a neuron also produce another feed-forward value that gets passed to the next neuron.
- Thus, sequence is encoded into the outputs. This recurrence of data gives the name of RNN.
- The sequence can be very strong but it weakens as the context spreads. This issue is called long-term dependency problem.
  - For example, in Fibonacci sequence, the number at the position 1 has very little impact on the number at position 100.

Here is the single/multiple neuron in RNN:



Here is the details for passing feed-forward value to next neuron:



Steps:

1. Feed x_0 into first neurons and calculate the result y_0 as well as the result(a.k.a feed-forward value) get to pass the next neuron.

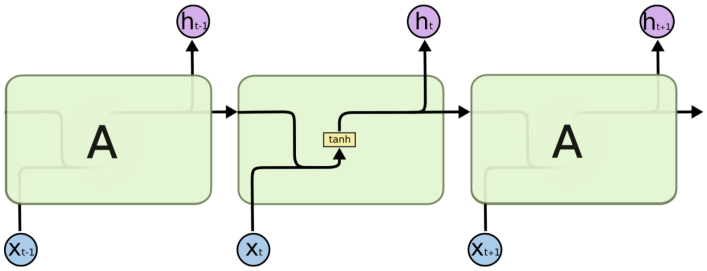
2. In terms of second neuron, it gets x_1 along with the fed-forward value from previous neuron and calculates y_1

3. In terms of third neuron, it gets x_2 along with the fed-forward value from previous neuron and calculates y_2 and so on

5.2 Long Short-Term Memory

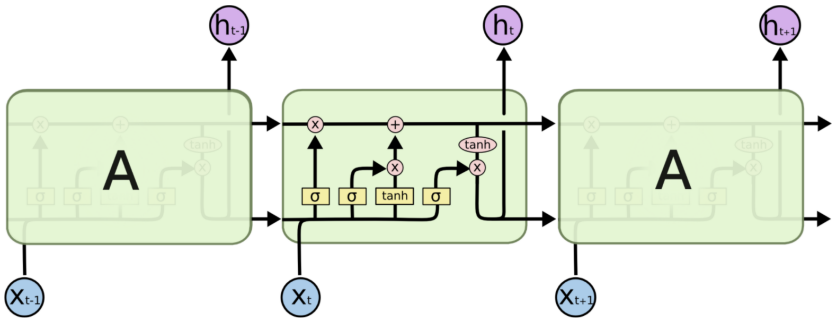
Here is a wonderful article, all notes are abstracted from here:
https://colah.github.io/posts/2015-08-Understanding-LSTMs/

- As we mentioned in the previous section, RNN has long-term dependency issue. However, LSTMs are explicitly designed to avoid it.
- Every RNNs have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

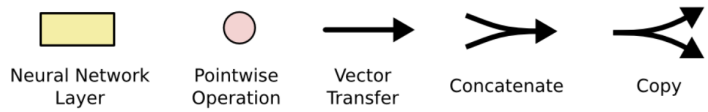


The repeating module in a standard RNN contains a single layer.

- LSTMs also have the similar chain structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



The repeating module in an LSTM contains four interacting layers.

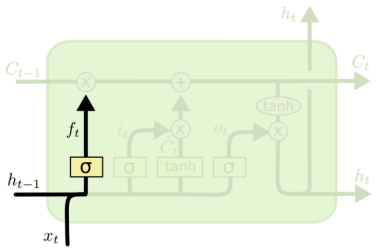


5.2.1 Core Idea Behind LSTMs

- The key to LSTMs is the cell state – horizontal line running through the top of the diagram. (like a conveyor belt with limited linear interactions.)
- The LSTM have the ability to remove or add information to the cell state, carefully regulated by structures called gates.
- Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.
- The sigmoid layer outputs numbers between 0 and 1, describing how much of each component should be let through. 0 means let nothing through, while 1 means let everything through.
- One LTSM has three of this type of gates to protect and control the cell state.

5.2.2 Step-by-Step LSTM Walk Through

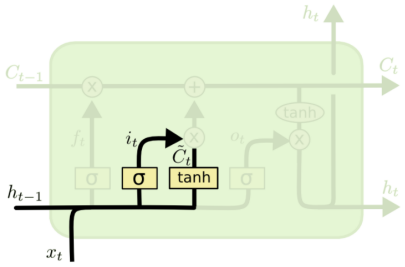
1. LSTM needs to decide what information should be thrown away from the cell state.
    - The decision is made by a sigmoid layer called the "forget gate layer".
    - It looks at h_{t-1} abd x_t, then outputs a number between 0 and 1 for each number in the cell state C_{t-1}.
    - 1 represents completely keep this and 0 means completely throw away.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

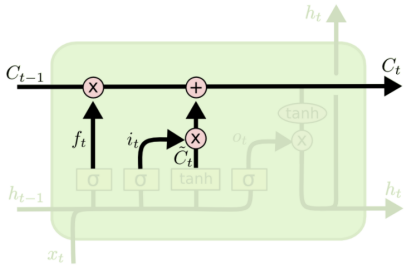
2. LSTM needs to decide what new information to be stored in the cell state.

   - A sigmoid layer called "input the gate layer" decides which values need to be updated.
   - A tanh layer creates a vector of new candidate values, $\widetilde{C}_t$, that could be added to the state.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

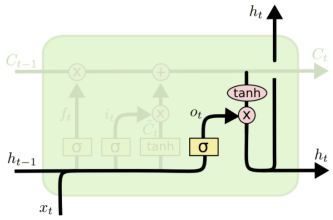
   - Then combining above two to create an update to the state.
     - Update the old cell state, C_{t-1}, into the new cell state C_t. The previous steps already decided what to do, then in current step, we need to actually do it.
     - Multiply the old state by f_t, forgetting the things that is decided to forget earlier.
     - Then add $i_t \times \widetilde{C}_t$.
     - Finally, we get the new candidate values, scaled by how much we decided to update each state value.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

3. LSTM need to decide what to output. The output is based on cell state, but will be filtered version.

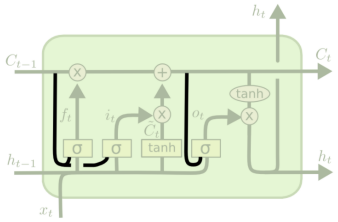
   - Run a sigmoid layer which decides what parts of the cell state to output.
   - Put cell state through tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

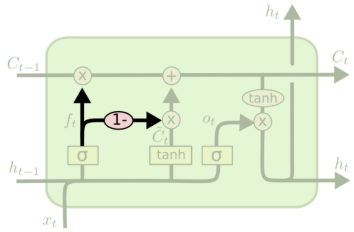
5.2.3 Variants on LSTM

- Peephole Connection: let certain number of gate layers look at the cell state.



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] + b_o\right)$$

- Applying coupled forget and input gates: instead of separately deciding what to forget and what should add new information to, we make those decision together.
  - Only forget when input something in its place
  - Input new values to the state when forget something older.



$$C_t = f_t * C_{t-1} + \boldsymbol{(1 - f_t)} * \tilde{C}_t$$