COMPUTER VISION

Kaiwalya Joshi

(20D070043)

Mentor - Hitvarth Diwanji

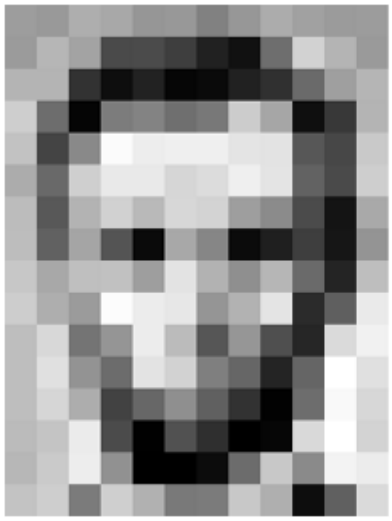
July 2021

Contents

Introduction

One of the most powerful and compelling types of AI is Computer Vision which we've almost experienced in many ways without even knowing. Computer Vision is the field of computer science that focuses on replicating parts of the complexity of the human visual system and enabling it to identify and process objects in images and videos in the same way that humans do. On a certain level, Computer Vision is all about pattern recognition. So one of the ways to train a computer how to understand visual data is to feed it lots of images that have been labeled, and then subject those to various software techniques, or algorithms, that allows the computer to hunt down patterns in all the elements that relate to those labels.

Let's get more technical. Below is a simple illustration of a grayscale image buffer that stores our image of Abraham Lincoln. Each pixel's brightness is represented by a single 8-bit number, whose range is from **0 (black) to 255 (white)**



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

This pixel data is stored in a one- dimensional array. Computers usually read colour as a series of 3 values -- **Red, Green, Blue (RGB)** on a scale of **0 - 255** for each colour.

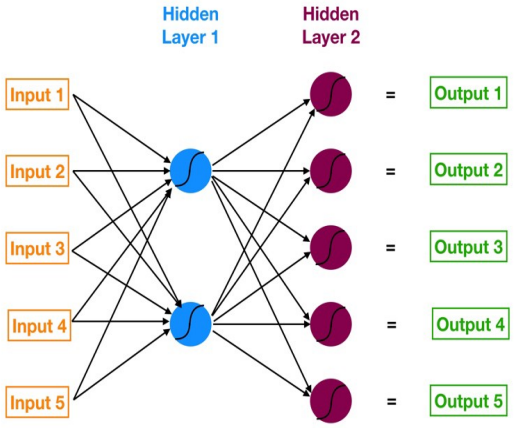
With this basic introduction of Computer Vision, we'll start learning concepts of Computer Vision with Neural Networks, BackPropagation and Convolutional Neural networks (CNN).

Neural Networks

Neural networks are multi-layer networks of neurons that we use to classify things, make predictions, etc

Structure of Neural Network

Below is a simple Neural Network diagram with 5 input neurons, 5 output neurons, and 2 hidden layers with 2 and 5 hidden neurons respectively.



The arrows that connect the dots show how all the neurons are interconnected and how data travels from the input layer all the way through to the output layer.

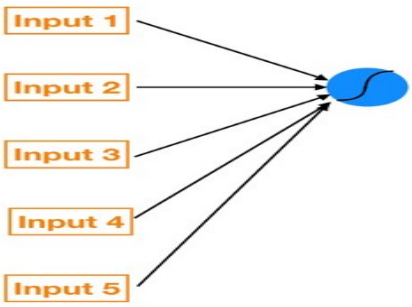
Input and Output

Let's get a basic idea of what the input and output layers are. Recall the image of Abraham Lincoln from the introduction. Recall that there we had converted that grayscale image of Abraham Lincoln into a one-dimensional array consisting of 12×16 numbers. There, a grayscale colour value of each pixel belongs from 0 to 255. But here we are going to assume that the grayscale colour value of each pixel belongs from 0 to 1 where 0 corresponds to black and 1 corresponds to white. And of course, these pixel values (0 to 1) are stored in a one-dimensional array of 192 (12×16) elements.

Now consider a Neural Network which we will use to predict the animal given in the image. It will have 192 input neurons where we will just provide the data in our array. Thus we will just provide a value in the array to an input neuron. The output will have a list of various animals (eg cat, dog, human). Based on the given 192 inputs, our neural network will predict the animal in the image by giving the output of 1 in the neuron corresponding to that particular animal and 0 in all other neurons. (Clearly, as we have

provided the inputs corresponding to humans, our neural network should predict human as answer)

Weights and Biases



Assuming that the above example must have given a better idea of input and output. Now we should try to find out hidden layer activation value. In order to do that, we'll multiply inputs by a particular set of weights and try to find out their weighted sum. Also, we'll add a bias to this weighted sum. Basically, anyhow we are trying to find out the correct set of weights and biases for any particular Neural Network.

Consider those arrows as weights and let's denote them as $\mathbf{W_{m,n}}$ where **m** indicates to neuron from input layer and **n** indicates to neuron from hidden layer. So this $\mathbf{W_{m,n}}$ indicates the weight of the arrow joining **mth** neuron from the input layer to **nth** neuron from the hidden layer.

Now let's calculate the outputs of each neuron in Hidden Layer 1 (known as the activation). We do so using the following formulas.

$$\mathbf{Z_1 = W_{1,1}*X_1 + W_{2,1}*X_2 + W_{3,1}*X_3 + W_{4,1}*X_4 + W_{5,1}*X_5 + B_1}$$
$$\mathbf{Z_2 = W_{1,2}*X_1 + W_{2,2}*X_2 + W_{3,2}*X_3 + W_{4,2}*X_4 + W_{5,2}*X_5 + B_2}$$

But this Z_1 and Z_2 can be greater than 1 also. But the activation values should lie between 0 and 1. To solve this problem, we use activation functions.

$$\begin{bmatrix} W1,1 & W2,1 & W3,1 & W4,1 & W5,1 \\ W1,2 & W2,2 & W3,2 & W4,2 & W5,2 \end{bmatrix} \times \begin{bmatrix} X1 \\ X2 \\ X3 \\ X4 \\ X5 \end{bmatrix} + \begin{bmatrix} Bias1 \\ Bias2 \end{bmatrix} = \begin{bmatrix} Z1 \\ Z2 \end{bmatrix}$$

$$\mathbf{[W] * [X] + [Bias] = [Z]}$$

Activation Function

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

2)Softmax = $\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \dots, K \text{ and } \mathbf{z} = (z_1, \dots, z_K) \in \mathbb{R}^K.$

3)ReLU(Rectified Linear Unit) = max(0,a)

Above are examples of few famous activation functions. So instead of just using [Z] as the activation value of the hidden layer, we operate a particular activation function on [Z] and use activation function(Z) as the activation value.

Eg. Neuron 1 Activation = Sigmoid(Z_1)

By repeatedly calculating [Z] and applying the activation function to it for each successive layer, we can move from input to output. This process is known as forward propagation.

Maths Behind Neural Networks

Now that we know how the outputs are calculated, it's time to start evaluating the quality of the outputs and training our neural network. **The training process of the Neural Network is to define a cost function and minimize it.** First, let's think about what levers we can pull to minimize the cost function. In traditional linear or logistic regression, we are searching for beta coefficients that minimize the cost function. For a neural network, we are doing the same thing but at a much larger and more complicated scale.

In traditional regression, we can change any particular beta in isolation without impacting the other beta coefficient and measuring its impact on the cost function, it is relatively straightforward to figure out in which direction we need to move to reduce and eventually minimize the cost function.

In Neural Network, changing the weight of any one connection(or the bias of a neuron)has a reverberating effect across all the other neurons and their activations in the subsequent layers.

Cost Function

We want to find the set of weights and biases that minimize our cost function where the cost function is an approximation of how wrong our predictions are relative to the target outcome.

For training our Neural Network, we will use Mean Square Error(MSE) as the cost function: **MSE = Sum [(Prediction - Actual)²] * (1 / num_observations)**

The MSE of a model tells us on average how wrong we are but with a twist — by squaring the errors of our predictions before averaging them, we punish predictions that are way off much more severely than ones that are just slightly off.

Optimizers

Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and learning rate in order to reduce the losses (cost function). How you should change the weights or learning rates of your neural network to reduce the losses is defined by the optimizers you use. Optimization algorithms or strategies are responsible for reducing the losses and to provide the most accurate results possible.

1) Gradient Descent:

Gradient Descent is the most basic but most used optimization algorithm. Backpropagation in neural networks also uses a gradient descent algorithm. Gradient descent is a first-order optimization algorithm that is dependent on the first-order derivative of a loss function. It calculates that which way the weights should be altered so that the Cost function can reach a minimum.
Algorithm: $\theta=\theta-\alpha \cdot \nabla J(\theta)$ Where θ is a weight and J(θ) is the cost function.

2) Stochastic Gradient Descent:

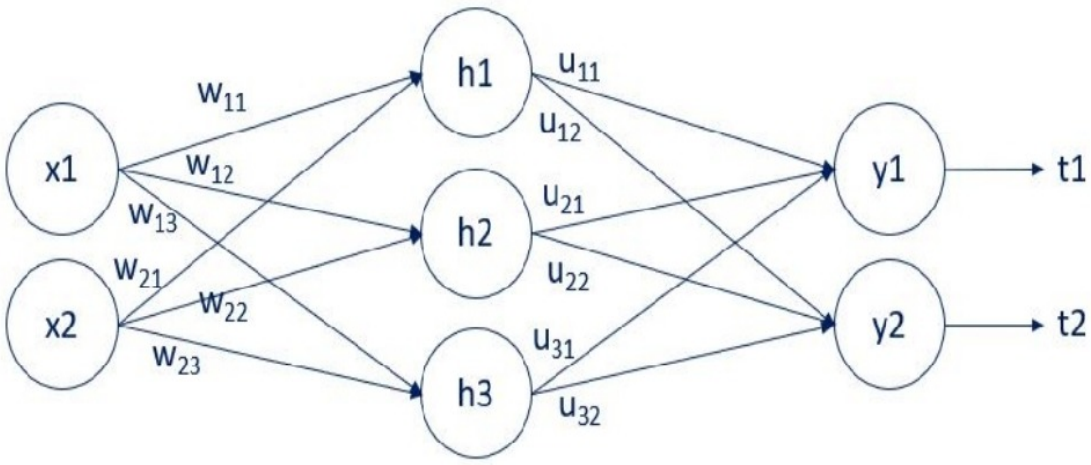
It's a variant of Gradient Descent. It tries to update the model's parameters more frequently. In this, the model parameters are altered after the computation of loss on each training example. As the model parameters are frequently updated parameters have high variance and fluctuations in loss functions at different intensities.
Algorithm: $\theta=\theta-\alpha \cdot \nabla J(\theta;x(i);y(i))$, where {x(i) ,y(i)} are the training examples.

3) Mini Batch Gradient Descent:

It's best among all the variations of gradient descent algorithms. It is an improvement on both SGD and standard gradient descent. It updates the model parameters after every batch. So, the dataset is divided into various batches and after every batch, the parameters are updated.
Algorithm:$\theta=\theta-\alpha \cdot \nabla J(\theta; B(i))$, where {B(i)} are the batches of training examples.

BackPropagation

Remember that forward propagation is the process of moving forward through the neural network (from inputs to the ultimate output or prediction). Backpropagation is the reverse. Except instead of signal, we are moving error backward through our model.



We have two inputs: x_1 and x_2 . There is a single hidden layer with3 units (nodes): $h_1, h_2, and\ 8h_3$. Finally, there are two outputs: y_1 and y_2. The arrows that connect them are the weights. There are two weights matrices: w, and u. The w weights connect the input layer and the hidden layer. u weights connect the hidden layer and the output layer. We have employed the letters w, and u, so it is easier to follow the computation to follow. You can also see that we compare the outputs y_1 and y_2 with the targets t_1 and t_2. There is one last letter we need to introduce before we can get to the computations. Let a be the linear combination prior to activation. Thus, we have: $a^{(1)} = xw + b^{(1)}$ and $a^{(2)} = hu + b^{(2)}$. Since we cannot exhaust all activation functions and all loss functions, we will focus on two of the most common. A sigmoid activation and cost function. With this new information and the new notation, the output y is equal to the activated linear combination. Therefore, for the output layer, we have $y = \sigma(a^{(2)})$, while for the hidden layer: $h = \sigma(a^{(1)})$. We will examine backpropagation for the output layer and the hidden layer separately, as the methodologies differ. **Useful Formulas**

$$1)\ Cost\ Function\ L(u) : \frac{1}{2}\sum_i (y_i - t_i)^2.$$

$$2)\ Sigmoid\ Function : \sigma(x) = \frac{1}{1+e^{-x}}$$

$$3)\ Derivative\ of\ sigmoid : \sigma'(x) = \sigma(x)(1-\sigma(x))$$

BackPropagation for the output layer

$$Algorithm : u - \eta \nabla_u L(u) \rightarrow u$$

We must calculate $\nabla_u L(u)$ 9 Let's take a single weight u_{ij}. The partial derivative of the loss w.r.t u_{ij} equals

$$\frac{\partial L}{\partial u_{ij}} = \frac{\partial L}{\partial y_j} . \frac{\partial y_j}{\partial a_j^{(2)}} . \frac{\partial a_j^{(2)}}{\partial u_{ij}}$$

where i corresponds to the previous layer (input layer for this transformation) and j corresponds to the next layer (output layer of the transformation). The partial derivatives were computed simply following the chain rule.

$$\frac{\partial L}{\partial y_j} = (y_j - t_j)$$

following the cost function derivative.

$$\frac{\partial y_j}{\partial a_j^{(2)}} = \sigma(a_j^{(2)})(1 - \sigma(a_j^{(2)})) = (y_j)(1 - y_j)$$

Finally, the third partial derivative is simply the derivative of $a^{(2)} = hu + b^{(2)}$.

$$\frac{\partial a_j^{(2)}}{\partial u_{ij}} = h_i$$

Replacing the partial derivatives in the expression above, we get:

$$\frac{\partial L}{\partial u_{ij}} = \frac{\partial L}{\partial y_j} . \frac{\partial y_j}{\partial a_j^{(2)}} . \frac{\partial a_j^(2)}{\partial u_{ij}} = (y_j - t_j).(y_j).(1 - y_j).(h_i) = \delta_i h_i$$

Therefore, the update rule for a single weight for the output layer is given by:

$$u_{ij} - \eta \delta_i h_i \rightarrow u_{ij}$$

10 Backpropagation of a hidden layer Similarly to the backpropagation of the output layer, the update rule for a single weight, w_{ij} would depend on:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial h_j} . \frac{\partial h_j}{\partial a_j^{(1)}} . \frac{\partial a_j^{(1)}}{\partial w_{ij}}$$

Taking advantage of the results we have so far for transformation using the sigmoid activation and the linear model, we get:

$$\frac{\partial h_j}{\partial a_j^{(1)}} = \sigma(a_j^{(1)})(1 - \sigma(a_j^{(1)})) = h_j(1 - h_j)$$

And

$$\frac{\partial a_j^{(1)}}{\partial w_{ij}} = x_i$$

The actual problem for backpropagation comes from the term $\frac{\partial L}{\partial h_j}$. That's due to the fact that there is no "hidden" target. You can follow the solution for weight w_{11} below. It is advisable to also check Figure , while going through the computations.

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial y_1} . \frac{\partial y_1}{\partial a_1^{(2)}} . \frac{\partial a_1^{(2)}}{\partial h_1} + \frac{\partial L}{\partial y_2} . \frac{\partial y_2}{\partial a_2^{(2)}} . \frac{\partial a_2^{(2)}}{\partial h_1} = (y_1 - t_1) y_1 (1 - y_1) u_{11} + (y_2 - t_2) y_2 (1 - y_2) u_{12}$$

From here, we can calculate $\frac{\partial L}{\partial w_{11}}$, which was what we wanted. The final expression is:

$$\frac{\partial L}{\partial w_{11}} = [(y_1 - t_1)y_1(1 - y_1)u_{11} + (y_2 - t_2)y_2(1 - y_2)u_{12}h_1(1 - h_1)x_1$$

11 The generalized form of this equation is:

$$\frac{\partial L}{\partial w_{ij}} = \sum_k (y_k - t_k)y_k(1 - y_k)u_{jk}h_j(1 - h_j)x_i$$

Backpropagation generalization Using the results for backpropagation for the output layer and the hidden layer, we can put them together in one formula, summarizing backpropagation, in the presence of cost function and sigmoid activations.

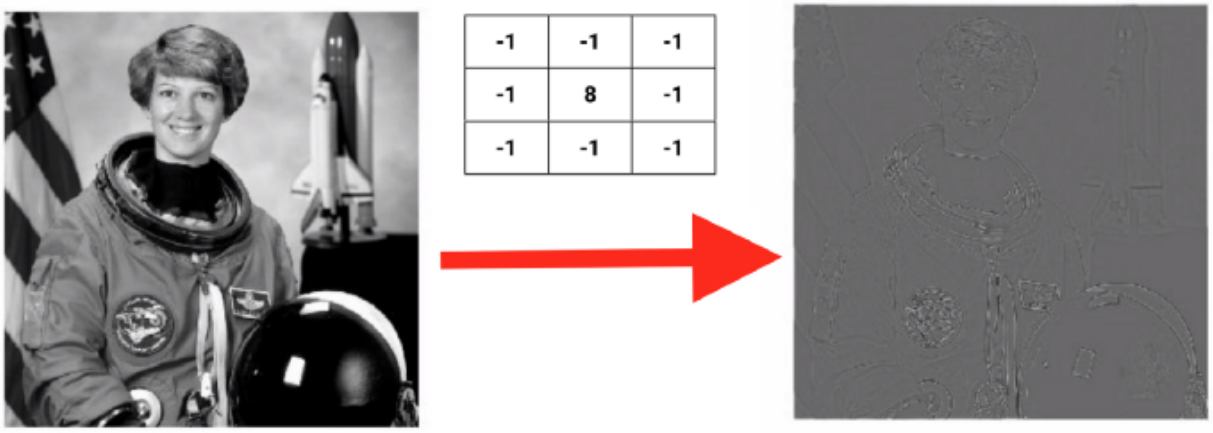
$$\frac{\partial L}{\partial w_{ij}} = \delta_j x_i.$$

where for a hidden layer

$$\delta_j = \sum_k \delta_k w_{jk} y_j (1 - y_j) x_i$$

Convolutional Neural Network

A convolution is how the input is modified by a filter. In convolutional networks, multiple filters are taken to slice through the image and map them one by one and learn different portions of an input image.



For example, there is a picture of Eileen Collins, and the matrix above the red arrow is used as convolution to detect dark edges. As a result, we see an image where only dark edges are emphasized.

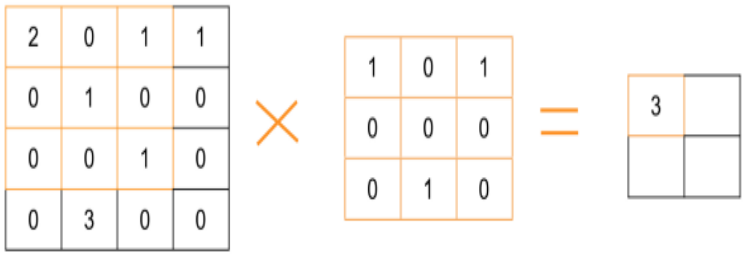
Note that an image is 2 dimensional with width and height. If the image is colored, it is considered to have one more dimension for RGB color. For that reason, 2D convolutions are usually used for black and white images, while 3D convolutions are used for colored images.

Convolution in 2D

Let's start with a (4 x 4) input image with no padding and we use a (3 x 3) convolution filter to get an output image.

2	0	1	1
0	1	0	0
0	0	1	0
0	3	0	0

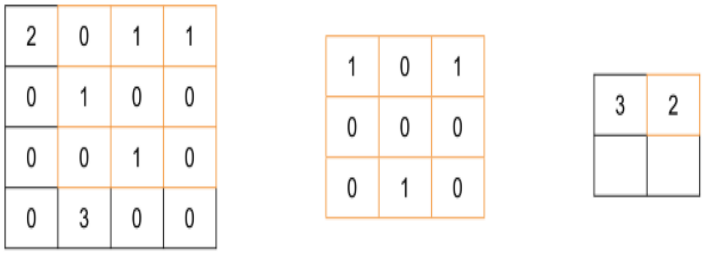
1	0	1
0	0	0
0	1	0



The first step is to multiply the yellow region in the input image with a filter. Each element is multiplied by an element in the corresponding location.

Then you sum all the results, which is one output value.

Mathematically, it's (2 * 1) + (0 * 0) + (1 * 1) + (0 * 0) + (1 * 0) + (0 * 0) + (0 * 0) + (0 * 1) + (1 * 0) = 3



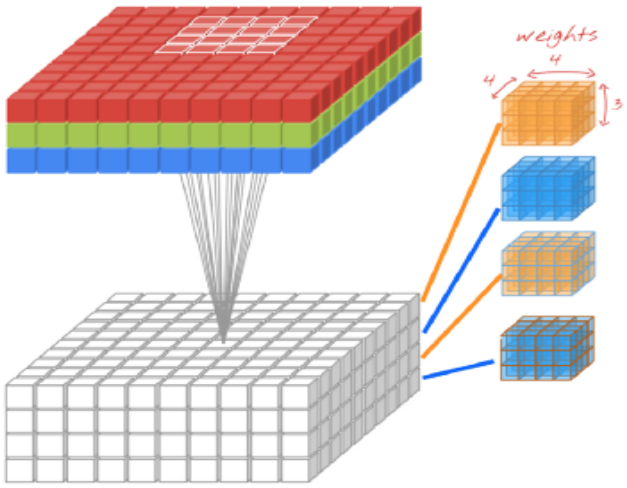
Then, you repeat the same step by moving the filter by one column. And you get the second output. Notice that you moved the filter by only one column. The step size as the filter slides across the image is called a **stride.** Here, the stride is 1.

At last, you are getting the final output. We see that the size of the output image is smaller than that of the input image. In fact, this is true in most cases.

Convolution in 3D

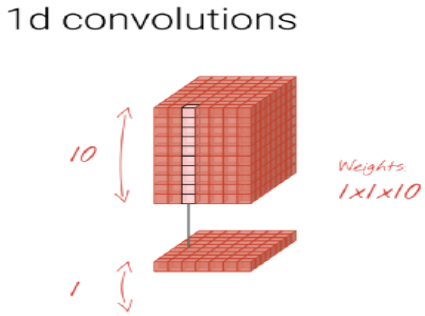
Convolution in 3D is just like 2D, except you are doing the 2d work 3 times because there are 3 color channels. Normally, the width of the output gets smaller, just like the size of the output in the 2D case.

As you add more filters, it increases the depth of the output image. If you have the depth of 4 for the output image, 4 filters were used. Each layer corresponds to one filter and learns one set of weights. It does not change between steps as it slides across the image.



An output channel of the convolutions is called a **feature map**. It encodes the presence or absence, and the degree of presence of the feature it detects. Notice that unlike the 2D filters from before, each filter connects to every input channel. This means they can compute sophisticated features. Initially, by looking at R, G, B channels, but after, by looking at combinations of learned features such as various edges, shapes, textures and semantic features.

Convolution in 1D

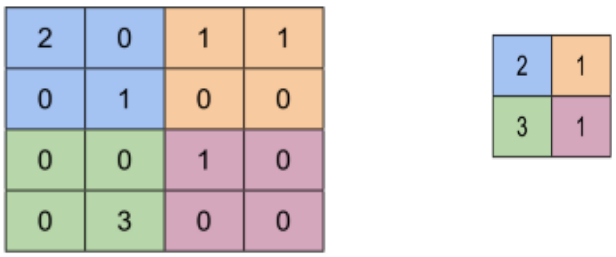


It's usually under-explained, but it has noteworthy benefits. They are used to reduce the depth (number of channels). Width and height are unchanged in this case. The 1D convolutions compute a weighted sum of input channels or features, which allow selecting certain combinations of features that are useful downstream

Pooling

Note that pooling is a separate step from convolution. Pooling is used to reduce the image size of width and height. Note that the depth is determined by the number of channels. As the name suggests, all it does is it picks the maximum value in a certain size of the window. Although it's usually applied spatially to reduce the x, y dimensions of an image.

Max-Pooling



Max pooling is used to reduce the image size by mapping the size of a given window into a single result by taking the maximum value of the elements in the window.

Average-Pooling

It's the same as max-pooling except that it averages the windows instead of picking the maximum value.

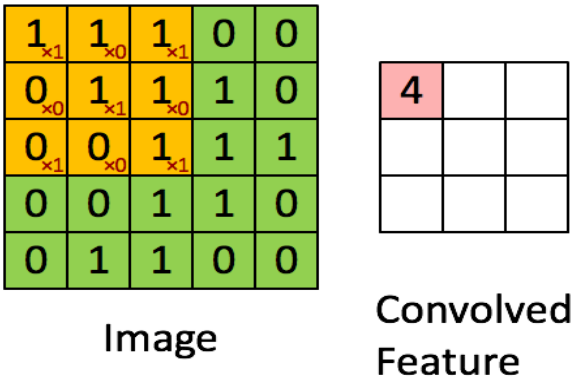
Common Setup

In order to implement CNNs, the most successful architecture uses one or more stacks of convolution + pool layers with relu activation, followed by a flatten layer then one or two dense layers. As we move through the network, feature maps become smaller spatially and increase in depth. Features become increasingly abstract and lose spatial information. For example, the network understands that the image contained an eye, but it is not sure where it was.

Basics of a larger architecture

Convolutional Neural Network:

Surely we now know a lot about CNN and thus we won't go into details of it. We will just see a summary of CNN here. Where regular vanilla neural networks were trained on layers that computed WX + b, Where W is the weight matrix that is learned through backpropagation, convolutional neural networks use weights called filters.



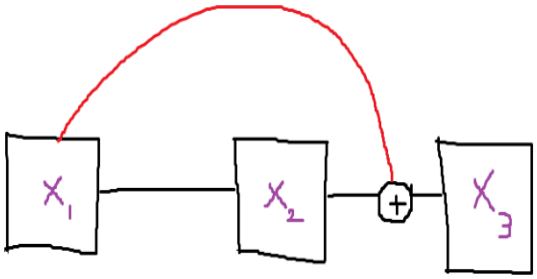
We can think of a convolutional filter, like a sliding window over the input matrix. In the image, the filter is the orange shaded matrix with red numbers. The input matrix is the green matrix with black numbers. At each stage, the filter is multiplied with the overlapped section of the input matrix element-wise, and the values are summed. This gives the first output. Then the filter is moved one step to the left, and so on. The loss can be calculated for the output and label with respect to the filter values, and with backpropagation, we can learn the values of the filter.

CNNs are extremely powerful for 2 main reasons:

1)They have significantly fewer parameters per layer, and hence can be stacked to form deeper layers.

2)They address the locality of input. The locality of pixels in an image is maintained, as the kernel acts on parts of the image at a time, and close pixels in the input create output values that are also close. This is different from traditional networks that do not take locality into consideration.

Residual Networks(ResNets):



ResNets have an extremely simple yet extremely elegant idea. The idea was to add skip connections or shortcut connections, that created a gradient highway. This enabled gradients to flow better during the backward

step, and greatly increased convergence, time of training, and reduced gradient explosion and vanishing. The subtle beauty of Resnets is that the best-case scenario is when the skip connection actively adds to the output and computes useful information, and the worst-case scenario is when it simply ignores the skipped connection and is at worst the same as a network without a skip connection. So skip connections add so much value and have no downside!

U Nets:

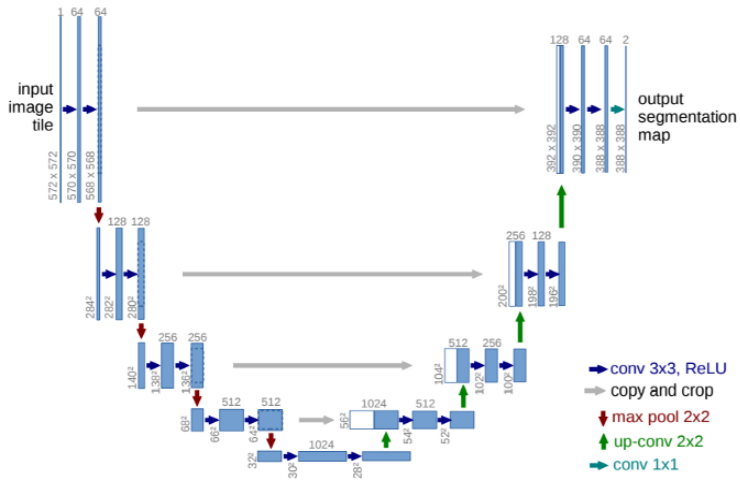
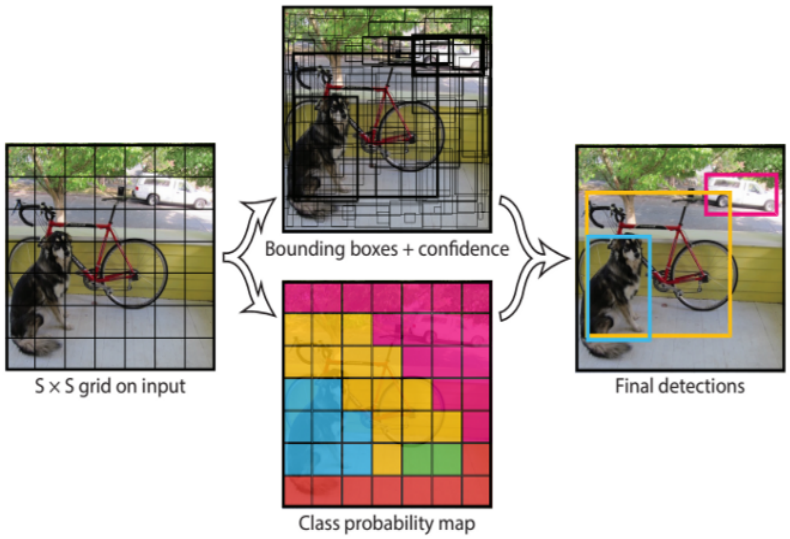


Image segmentation is the task of labeling each pixel in an image with its category. U-Nets have 2 parts, the contracting path (downsampling path) and the expansive path (upsampling path). In a traditional image classification convolutional network, the image is fed into a network that performs convolutions and pooling operations, both of which reduce the height and width of the output but increase the depth of the output. With a loss in height and width, the depth gained adds features to the classification output. However, in segmentation tasks, we want the output to be the same shape as the input image and the added features for labeling pixels. So the downsampling of a traditional Conv architecture is supplemented by an upsampling path, to add back the height and width of the image to the output, while maintaining the features. There are many upsampling methods, but the most common one used in most libraries is Transpose convolution upsampling. The transposed convolution operation forms the same connectivity as the normal convolution but in the backward direction. We can use it to conduct up-sampling. Moreover, the weights in the transposed convolution are learnable. So we do not need a predefined interpolation method. Even though it is called the transposed convolution, it does not mean that we take some existing convolution matrix and use the transposed version. The main point is that the association between the input and the output is handled in a backward fashion compared with a standard convolution matrix (one-to-many rather than many-to-one association). As such, the transposed convolution is not a convolution. But we can emulate the transposed convolution using a convolution. We up-sample the input by adding zeros between the values in the input

matrix in a way that the direct convolution produces the same effect as the transposed convolution. You may find some article explains the transposed convolution in this way. However, it is less efficient due to the need to add zeros to up-sample the input before the convolution.

YOLO(You Only Look Once):

YOLO stands for you only look once. When the paper was released, the popular method for object detection was to reuse classifiers to classify local regions of an image and use a sliding window approach to check if each region of an image has an object. YOLO shifted the paradigm by proposing object detection as a regression problem, where they only use a single network for the entire pipeline and process the whole image at once rather than in regions.



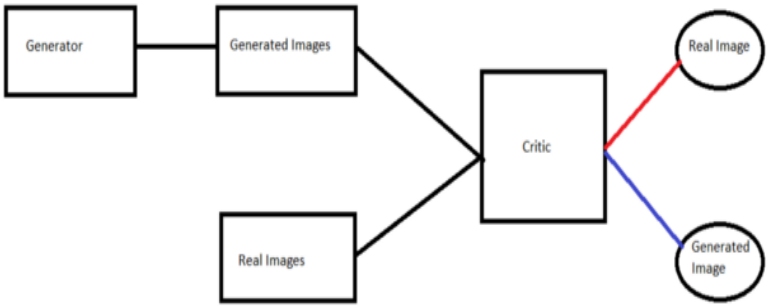
YOLO divides the input image into an SxS grid. And for each grid predicts if the center of an object is present within the grid. If the center of the object is in the grid, the grid will then predict a bounding box with 5 values, x,y,w,h,c. (x,y) are the coordinates of the center of the object relative to the grid, (w,h) is the width and height of the object relative to the whole image, and c is the class of the object.

YOLO has 3 main benefits:

- First, YOLO is extremely fast. Since the paper frames detection as a regression problem, it doesn't need a complex pipeline.
- Second, YOLO reasons globally about the image when making predictions. Unlike sliding window and region proposal-based techniques, YOLO sees the entire image during training and test time so it implicitly encodes contextual information about classes as well as their appearance.

- Third, YOLO learns generalizable representations of objects. Since YOLO is highly generalizable it is less likely to break down when applied to new domains or unexpected inputs.

Generative Adversal Networks:



GANs are neural network pairs that are trained through an adversarial process. The 2 parts of a GAN are a Generator and a Critic/Discriminator. The role of the generator is to generate high-quality data that is similar to training data, and the role of the critic is to differentiate between the generated data and the real data. The objective function of the generator is to maximize the loss of the critic, and the objective function of the critic is to minimize its loss. Think of this process as analogous to a thief and the police. The thieves want to fool the police and keep improving their tools and techniques, and the police want to catch the thieves so they improve too. The generator is like the thief and the critic is like the police. There are numerous applications for GANs and many new applications are coming out all the time. But since this article is about computer vision, two extremely interesting applications of GANs are:

- **Super Resolution:**

Super-resolution refers to taking a low-quality image and generating a high-quality image from it. There is an interesting approach called as noGAN approach for Super-resolution. This process is a sort of pretraining for GANs, where high-quality images are converted to lower quality images for training data of the Generator, and the critic is pre-trained on the generated images. This way, both the generator and critic have a head start, and this method is found to significantly improve training time for GANs.

- **Deep fakes:**

Deep fakes are also GANs where the generator is trained to perform the faking operation, and the critic is tasked with detecting the fake. The generator can be trained for long enough to fool most humans. This is a somewhat dangerous technology and something to be aware of on the internet.

Training a CNN on MNIST dataset:

We are going to write a digit recognition code which is often considered as a **“Hello World”** program in this field. The MNIST dataset is a dataset of 28 × 28 70000 images of digits along with their labels. Out of 70000 images, 60000 images are for training, and the rest of the 10000 are for testing.

So let’s import tensorflow and load our data as shown below.

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Note that each pixel value should be between 0 and 1. To do so, we must normalize our x_train, x_test.

```
x_train = tf.keras.utils.normalize(x_train, axis = 1)
x_test = tf.keras.utils.normalize(x_test, axis = 1)
```

Now resize our model to make it suitable for applying CNN.

```
import numpy as np
x_trainr = np.array(x_train).reshape(-1, 28, 28, 1)
x_testr = np.array(x_test).reshape(-1, 28, 28, 1)
```

Make a CNN model

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense, Dropout, Conv2D,
MaxPooling2D, Activation

model = Sequential()

model.add(Conv2D(64, (3,3)))
model.add(Activation("relu"))
```

```
model.add(MaxPooling2D(pool_size = (2,2)))

model.add(Conv2D(64, (3,3)))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size = (2,2)))

model.add(Flatten())
model.add(Dense(64, Activation("relu")))
model.add(Dense(32, Activation("relu")))
model.add(Dense(10, Activation("Softmax")))
```

You can see the summary of your model using

```
model.summary()
```

Now, this was for one x_train image. After compiling, we can use this for all 60000 images. So let's compile it using

```
model.compile(optimizer="adam",
              loss=tf.keras.losses.sparse_categorical_crossentropy,
              metrics=["accuracy"]
```

Now it is time to fit our model.

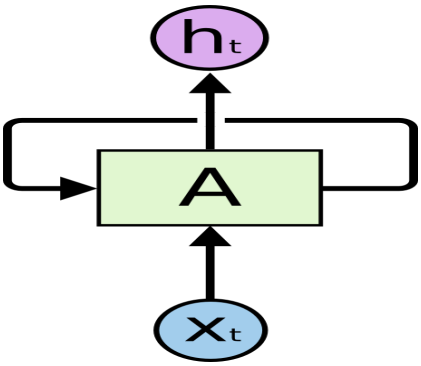
```
model.fit(x_trainr, y_train, epochs=5, validation_split= 0.3)
```

This completes training our model. Now it is time to check our model's accuracy by testing them on x_test and y_test.

```
loss, accuracy = model.evaluate(x_testr, y_test)
predictions = model.predict([x_testr])
print(np.argmax(predictions[0])) #for predicting first number.
```

Recurrent Neural Networks:

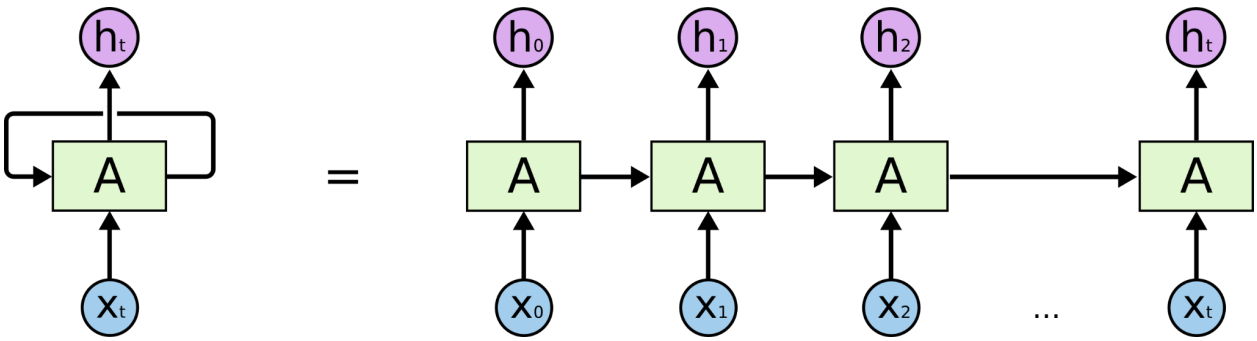
Humans don't start their thinking from scratch every second. As you read this Computer Vision Report, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence.
Traditional neural networks can't do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. It's unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.
Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.



In the side diagram, a chunk of the neural network, A looks at some input x_t and outputs a value h_t . A loop allows information to be passed from one step of the network to the next.

Recurrent Neural Networks have loops.

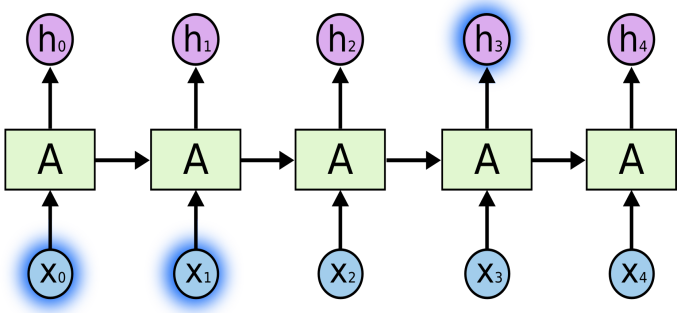
These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they aren't all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:



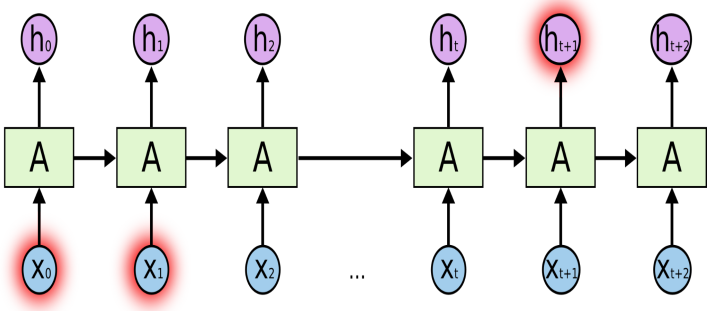
This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of the neural networks to use for such data.
And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning… The list goes on.
Essential to these successes is the use of "LSTMs," a very special kind of recurrent neural network that works, for many tasks, much much better than the standard version. Almost all exciting results based on recurrent neural networks are achieved with them. It's these LSTMs that this essay will explore.

The Problem of Long-Term Dependencies:

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If RNNs could do this, they'd be extremely useful. But can they? It depends.
Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the ***sky***," we don't need any further context – it's pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.
But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France… I speak fluent ***French***." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.



Low gap easy to predict

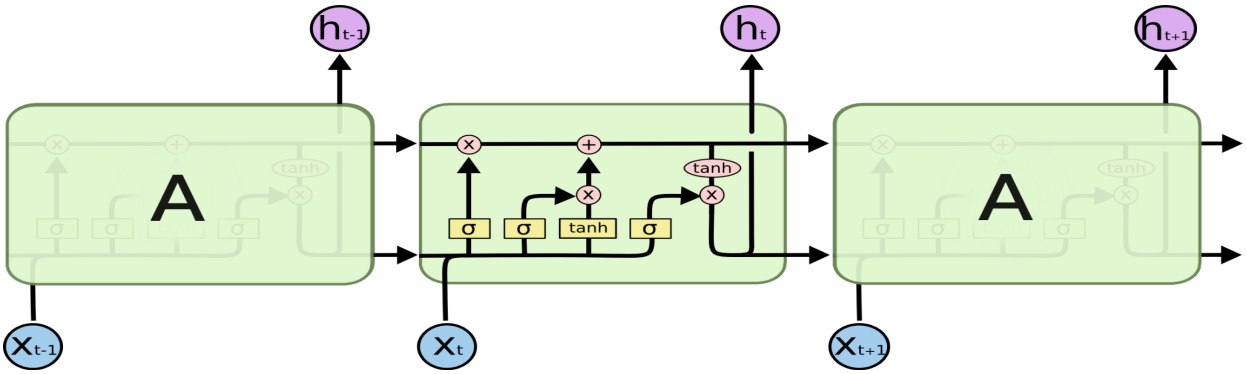


Large gap tough to predict

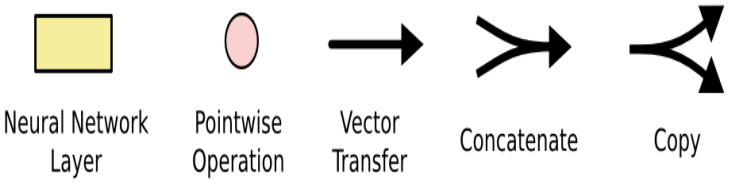
In theory, RNNs are absolutely capable of handling such “long-term dependencies.” A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don’t seem to be able to learn them.
Thankfully, LSTMs don’t have this problem!

LSTM Networks:

Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies. They work tremendously well on a large variety of problems, and are now widely used.
All recurrent neural networks 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.
LSTMs also have this chain like 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.

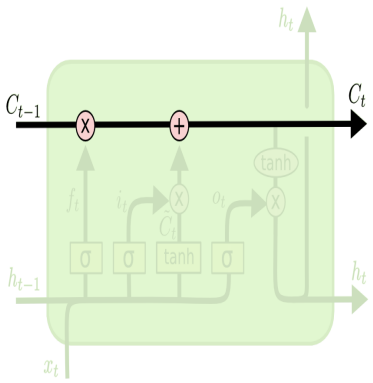


The repeating module in an LSTM contains four interacting layers.

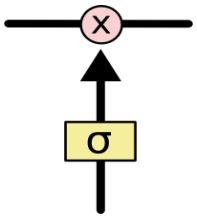


In the side diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations.

The Core Idea Behind LSTMs:



The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

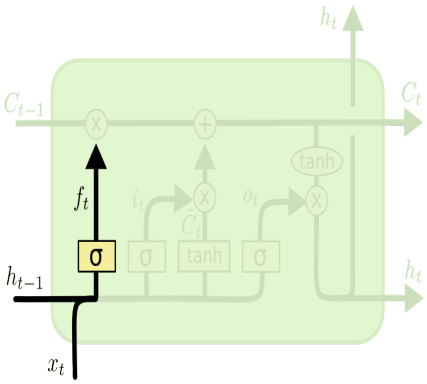


The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!"
An LSTM has three of these gates, to protect and control the cell state.

Step-by-Step LSTM Walk Through:

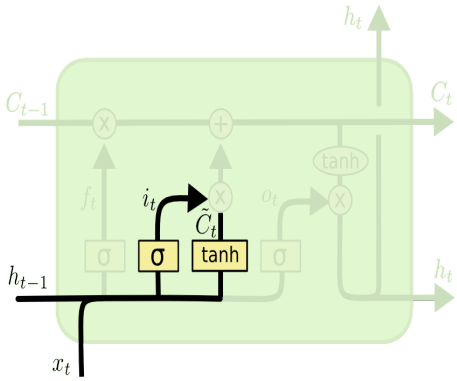
The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at h_{t-1} and x_t and output a number between 0 and 1 for each number, the cell state C_{t-1}. One represents "completely keep this" while a zero represents "completely get rid of this."



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

Let's go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell

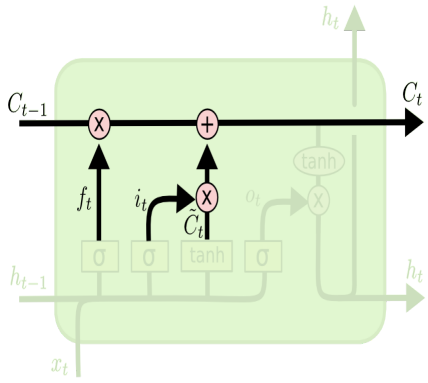
state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.
The next step is to decide what new information we're going to store in the cell state.



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

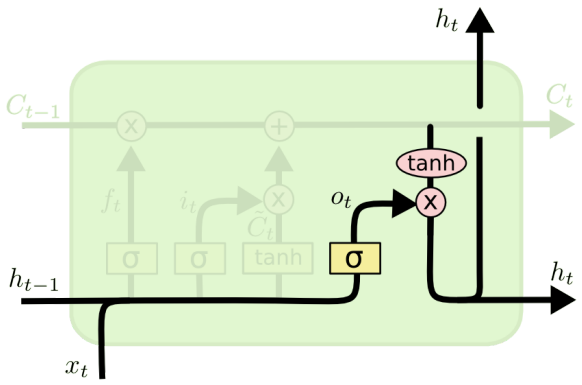
This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, $\tilde{C}_t$, that could be added to the state. In the next step, we'll combine these two to create an update to the state.
In the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting.

It's now time to update the old cell state, C_{t-1}, into the new cell state C_t . The previous steps already decided what to do, we just need to actually do it.
We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value. In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.



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

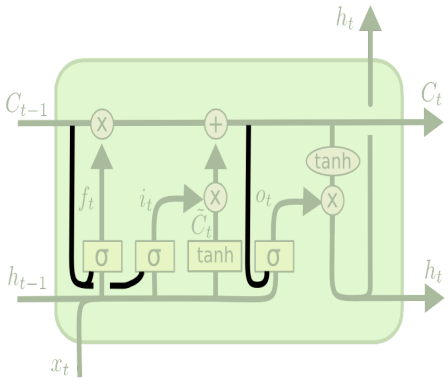
Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.



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

Variants on Long Short Term Memory:

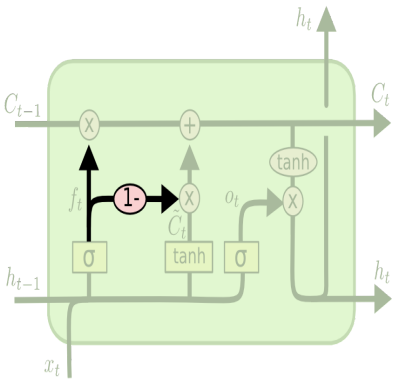
One popular LSTM variant is adding “peephole connections.” This means that we let the gate layers look at the cell state.



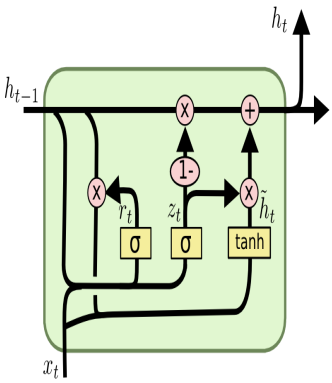
$$f_t = \sigma\left(W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [C_t, h_{t-1}, x_t] + b_o\right)$$

The side diagram adds peepholes to all the gates, but many papers will give some peepholes and not others.

Another variation is to use coupled forget and input gates. Instead of separately deciding what to forget and what we should add new information to, we make those decisions together. We only forget when we’re going to input something in its place. We only input new values to the state when we forget something older.



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



$$z_t = \sigma\left(W_z \cdot \left[h_{t-1}, x_t\right]\right)$$

$$r_t = \sigma\left(W_r \cdot \left[h_{t-1}, x_t\right]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot \left[r_t * h_{t-1}, x_t\right]\right)$$

$$h_t = \left(1 - z_t\right) * h_{t-1} + z_t * \tilde{h}_t$$

A slightly more dramatic variation on the LSTM is the Gated Recurrent Unit, or GRU. It combines the forget and input gates into a single “update gate.” It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.

Conclusion:

Earlier, I mentioned the remarkable results people are achieving with RNNs. Essentially all of these are achieved using LSTMs. They really work a lot better for most tasks! Written down as a set of equations, LSTMs look pretty intimidating. Hopefully, walking through them step by step in this essay has made them a bit more approachable. LSTMs were a big step in what we can accomplish with RNNs. It’s natural to wonder: is there another big step? A common opinion among researchers is: “Yes! There is a next step and it’s attention!” The idea is to let every step of an RNN pick information to look at from some larger collection of information. For example, if you are using an RNN to create a caption describing an image, it might pick a part of the image to look at for every word it outputs. In fact, it might be a fun starting point if you want to explore attention! There’s been a number of really exciting results using attention, and it seems like a lot more are around the corner…