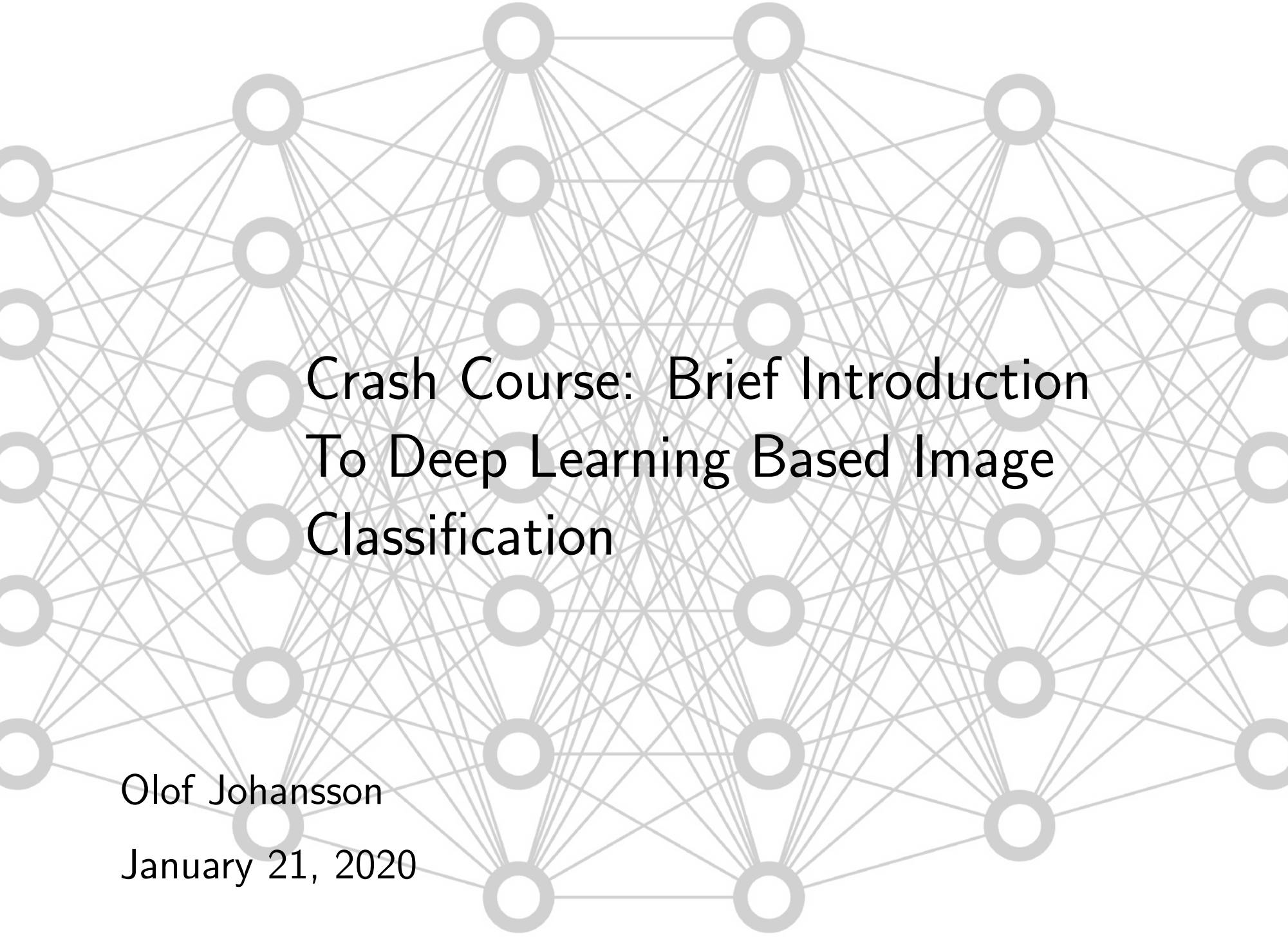


Crash Course: Brief Introduction To Deep Learning Based Image Classification

Olof Johansson

January 21, 2020

This is a brief introduction to deep learning based video classification and gives an overview of what deep learning is and the different steps required to perform video classification. Many of the topics obviously requires more reading to understand more intiutivly and the key words are enlightened like this.

Have in mind that this is just an overview to get you in the understanding of how a some processes work in order for you to more easily build upon it. It's therefore not a guide to start using and building deep learning models and neural networks.

What is Deep Learning?

Neural Networks and Deep Learning

Deep learning is basically just a more complex subbranch of machine learning - where computers learn to perform certain tasks based on, like human intelligence, prior experience. It is made up by layers stacked upon each other to form a deep neural network. The word "deep" learning comes from the required amount of layers needed to perform the more complex machine learning tasks.

What is Deep Learning?

Neural Network and Deep Learning

Essentially, a layer is a multidimensional matrix (or tensor) where each element is called a artificial neuron (or node). These neurons consists of functions called activation functions. When input data is being shoved into the network, the activation functions takes the input data, perform its computations and then pass it to the next layer of neurons until it reaches the final layer called the output layer. This whole process from input to output layer is called forward propagation since the data is being forwardly propagated through the network.

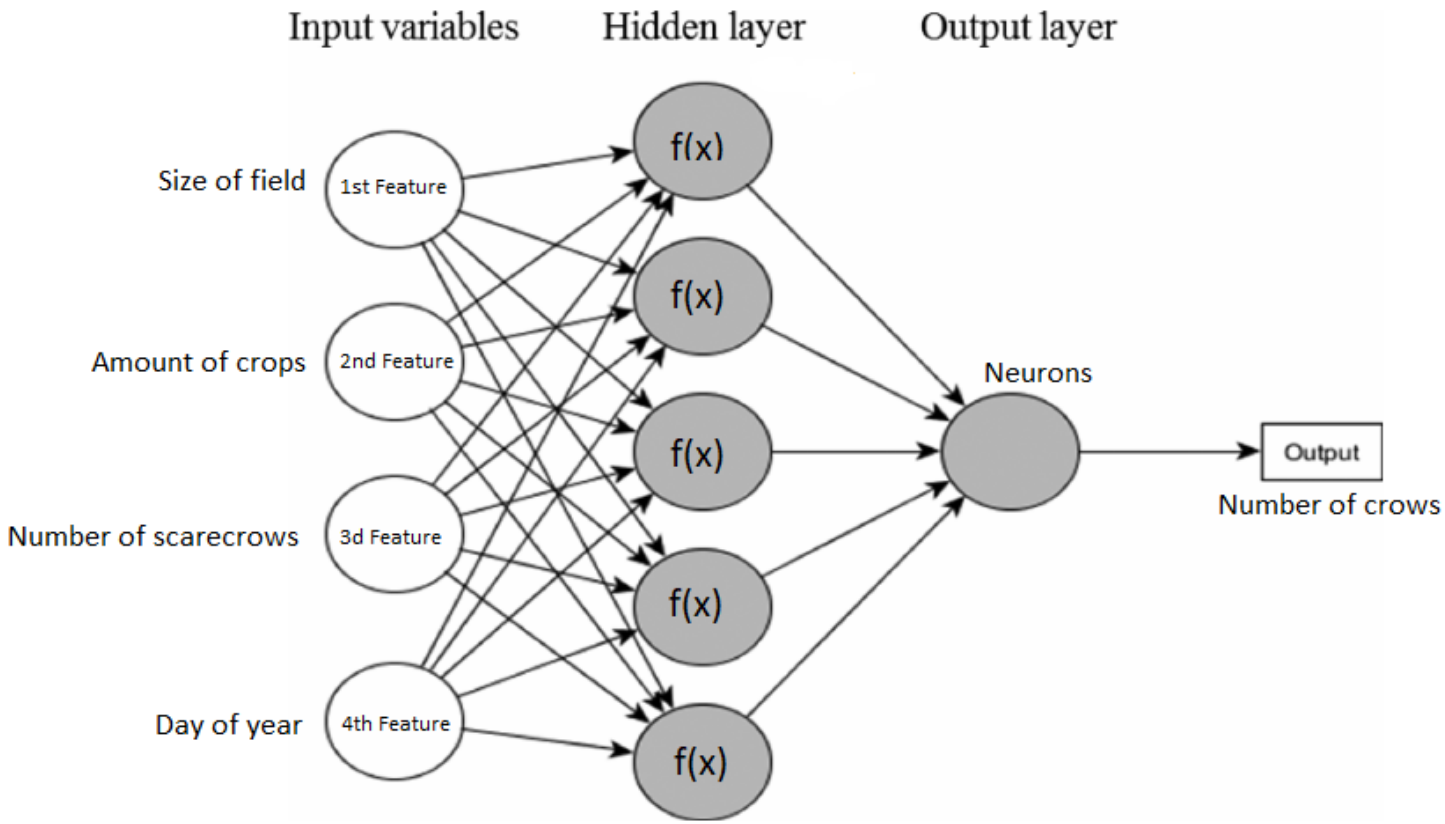
What is Deep Learning?

Neural Networks and Deep Learning

Consider you want to predict the number of crows on a field from the amount of acres. Then, the "feature" you want to train on is the amount of acres. This data will be the input of a neuron consisting of an activation function which will output a number of the predicted number of crows. If the number of crows depends on more than just the field size, for example the amount of crops, number of scarecrows and the day of the year, the number of features that predicts the amount of crows increase to four. These are collected in a vector called feature vector which will all be the input to the neuron. When having multiple features it's convenient to also increase the amount of neurons in order to get better predictions. As the neurons increase, it might also be suited to use more layers of neurons such that the output of the first set of neurons will be input to the next. These layers are called hidden layers.

What is Deep Learning?

Neural Networks and Deep Learning



Supervised Deep Learning

Neural Network and Deep Learning

So how do they learn? Essentially there are three different types of learning; supervised learning, where the network learns to predict data (example above), unsupervised learning, where the network doesn't care about what the data actually is, but rather distinctions in the data, and reinforcement learning, where the network is given a reward based on the performance. This crash course will focus on the initial, supervised learning.

Supervised Deep Learning

Neural Network and Deep Learning

During training, a supervised learning model is given a set of data and its corresponding true value (if numerical data) or label (if categorical data). After the data is forwardly passed through the network, the output is then compared by the correct value by a loss (or error) function that computes how much the predicted output differ from the true value. After the loss is computed, a process called backpropagation updates the parameters in all the neurons based on the loss starting from the last layer and going backwards through the network. This is done using an optimization function. The most well-known optimization function is the gradient descent where the parameters are updated by taking the negative gradient of the loss function times a learning rate. The aim of gradient descent, and the optimization function in general, is to move in the direction of the global minimum of the loss function to reach the minimal difference of the predicted values and the correct ones, hence the negative gradient. The learning rate is basically just the step size of this movement.

Supervised Deep Learning

Neural Network and Deep Learning

Essentially, backpropagation using gradient descent is process where each parameter in the neural network is being updated proportional to the partial derivative of the loss function with respect to the current parameter. Starting from the last one and going backwards and is done for each iteration of training.

Supervised Deep Learning

Neural Network and Deep Learning

Summarized, the learning process of a very simple neural network consists of

- Forward propagation
- Loss computation
- Backpropagation
- Repeat until loss is close to global minimum

This is the basic idea of the learning process for a supervised learning model (note - basic, but alot of problems can be solved by just this). Before moving into the next layer, normalizing the data is crucial for the learning performance, since the values of the different features is usually of different sizes and types.

Convolutional Neural Networks

Neural Networks and Deep Learning

Consider now that you have a security camera at your front door which snapshots everytime the door bell goes off and you want to know whether there's a person there or just a kid being playful. This is called a binary classification task since the task is to classify if there is a person in the snapshot (output value 1) or not (output value 0). Assume the snapshot is an 224x224 RGB image, how would the input data to the neural network look like?

Convolutional Neural Networks

If to use network as previously, to get a reasonable input data, the image of size (pixel width x pixel height) is extracted into one big feature vector where each element is a pixel value. So, if the image is of size 224x224 and is an RGB image, the feature vector will be of size 224x224x3 = 150528. This means that for ONE image, EACH neuron will have 150528 parameters to train. Having a network with multiple layers, each with multiple neurons, the amount of parameters required to train will be fucking massive and the process would take ages and the computational cost would be absurd. Luckily, there is a solution to this - convolution.

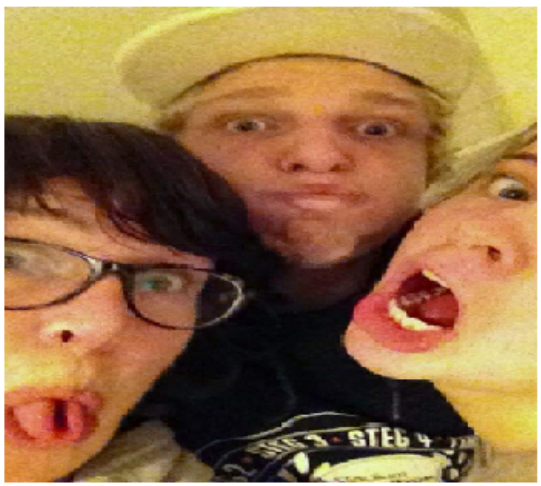
Convolutional Neural Networks

Convolution is a mathematical operation where the shape of one input can be modified by another. In deep learning, it is done matrix wise where a convolutional filter (which is a matrix consisting of fixed integers), called a kernel, slides over an input matrix and modify/extracts specific properties in that image (eg edges, colors etc). Using a kernel can be thought of as a flashlight sliding across the image one row at a time, extracting/modifying the image in a way determined by the integers in the kernel.

Convolutional Neural Networks

Neural Networks and Deep Learning

Original image



3x3 Kernel for vertical edges

1	1	1
0	0	0
-1	-1	-1

Convolved image

Convolutional Neural Networks

Neural Networks and Deep Learning

The output of a kernel is a matrix called a feature map (matrix version of feature vector) and the pure motivation behind it is parameter reduction since it breaks down the image into basic horizontal and vertical edges. The deeper the network, the more details can be extracted and examined by it. After each convolutional layer, the output normalizes and often regularizes to increase generality and enhance performance. Pooling is also performed after each convolutional layer which is basically just taking the most interesting pixel value from each part of the feature map of size determined by the pooling size. Either average pooling which is taking the averages of all the pixel values of each part or, more used, max pooling which instead takes the maximum pixel value. By the use of pooling, the dimensions of the feature map is reduced. A neural network with layers consisting of convolutional blocks is called a convolutional neural network (CNNs or ConvNets) and is always used when dealing with graphical data (and some other shit).

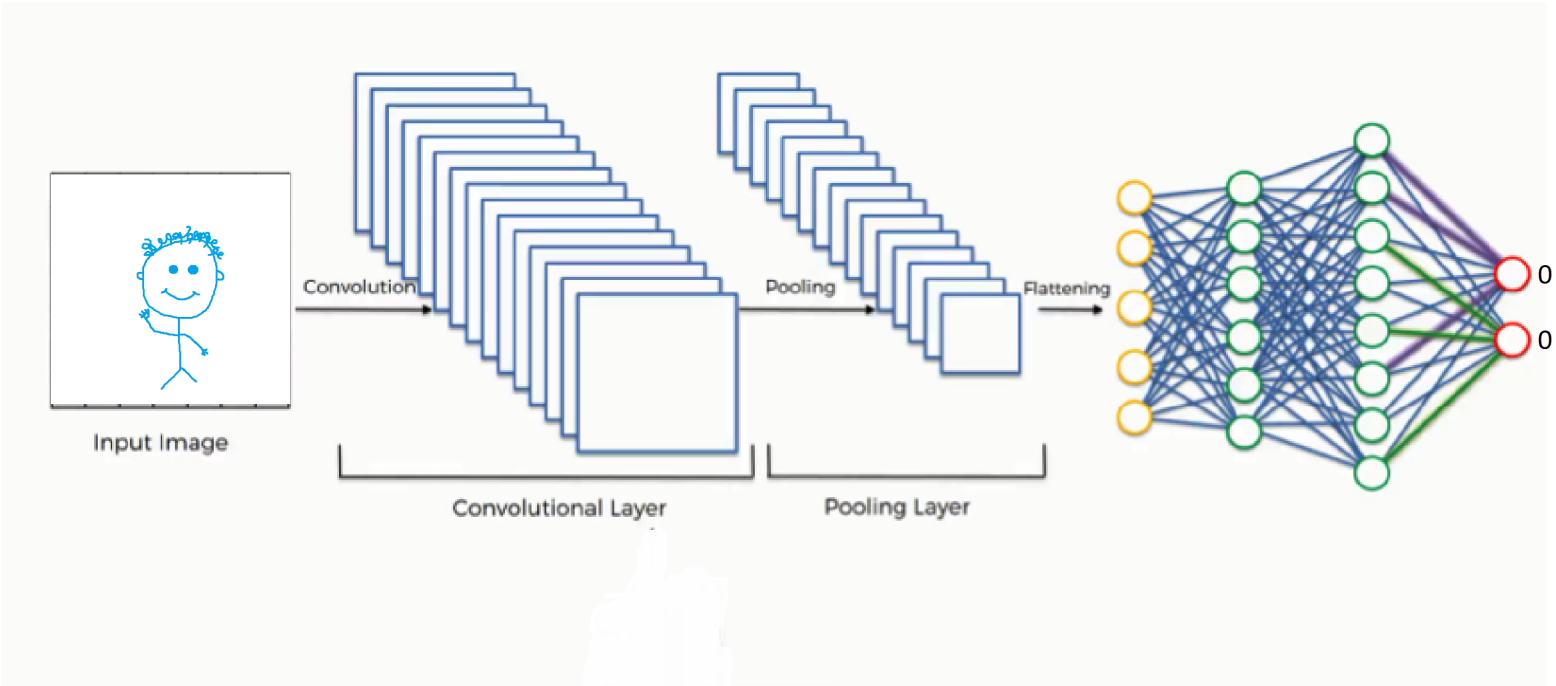
Image Classification

Convolutional Neural Networks

So, back to the security camera. We want the output to be either 0 or 1 (or somewhere in between), but the feature maps (output of the convolutional blocks) are matrices. In order to get probability values, the output of the last convolutional blocks flattens out into a feature vector which then can be treated as before. The feature vector being outputted by the network is then being compressed down into an output layer consisting of the probabilites of the output being either a person or not.

Image Classification

Convolutional Neural Networks



One-shot Learning

Neural Network and Deep Learning

Now, if you wish to extend the models complexity so that it not only tell you if there is a person there or not, but also tell you if the person is a stranger or a friend. For this task, a different type of classification learning called one-shot learning (or N-shot learning) is prefered. One-shot learning is used when having little training data and also for tasks where the number of objects may increase. As for the security camera, the number of friends may (or may not :/) increase. If used a standard classification network, the whole model needs to be retrained everytime you wish to add a person to be recognized (which will take days or weeks). One-shot learning address this problem in a simple and clever way.

One-shot Learning

Neural Network and Deep Learning

Instead of learning the network to recognize the object in the image, a one-shot learning model learns to differ distinctions between reference data and new data by comparing the similarities between the two. For images, a network called siamese network is used.

Siamese Network

One-shot Learning

A siamese network uses two convolutional networks, called legs, one for the reference images (positive sample containing images for each class) and one for the input image (called anchor image). This is trained by replacing the loss function with a distance function which computes the euclidian distance between the output feature vectors from both legs (called contrastive loss). This loss is defined by minimizing the distance if the anchor image is the same class as the reference image and maximizing the distance if it isn't. This works fine but it can be enhanced by adding one more leg and extend the loss into a triplet loss function. Then after training, more classes can easily be added by just extend the positive sample set.

Recurrent Neural Networks

Sequence models

Okey, let's expand the security camera even further. Let's now suggest that your street is packed with kids playing football and hide and seek right, but it doesn't contain any speed bumps so cars often just blast by. You wish to track cars that's going way to fast right, how would you do this by using deep learning? (This is an examlpe of a problem where using deep learning is an expensive exaggeration, but for the sake of point let's do it anyway). First of all you want to know when a car has entered the vision of the camera which is done by an object detection model. Then, you must have a network that makes prediction out of time series examines dynamic behaviour. This is recurrent neural networks (RNNs). Here we'll cover the most used one, the Long Short Term Memory network (LSTM).

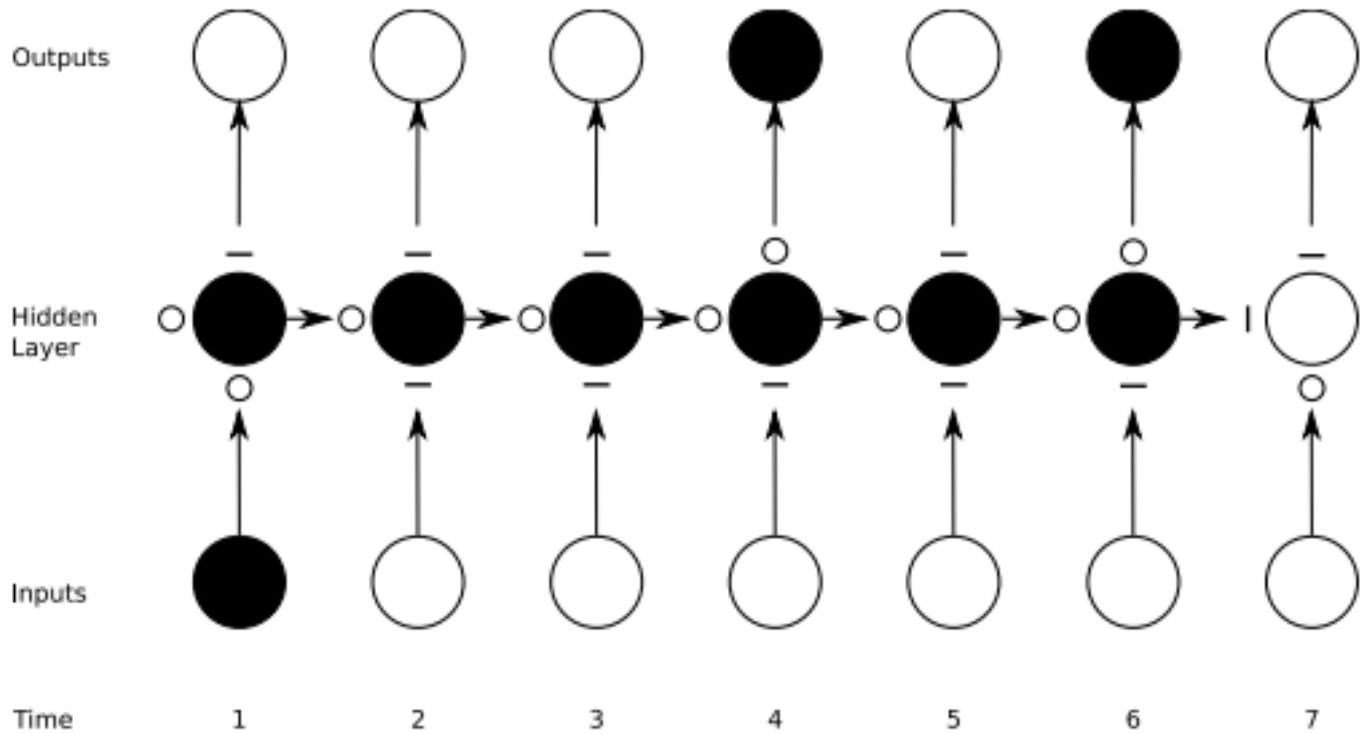
Long Short Term Memory (LSTM)

Recurrent Neural Networks

Without getting into too much detail, the architecture of a LSTM network consists of a cell containing information from a CNN and then three gates (input gate, forget gate and output gate) controlling the flow of information. Several units are bind together, passing information to the next unit. This is easier explained through visualisation with an example below where the minus sign represents a closed gate and circle represents an open one. LSTM solves the problem earlier experienced in RNNs with vanishing gradients (problem when the gradient becomes too small for training) and also enabled long time dependencies (longer time series). Don't contemplate to much about them tho, get to know what they are used for instead of how they do shit.

Long Short Term Memory (LSTM)

Recurrent Neural Networks



Putting It All Together

Okey, so let's put it all together with another example. Suggest that you wish to create a model that recognizes different moving hand gestures. Since the data consists of video seqences of the desired hand gestures, a RNN seems appropriate. A LSTM is therefore prefered. Since a video consists of multiple stacked images, a CNN will be attached to the LSTM. The number of units of the LSTM will be determined by the video length of the gestures. Loads of training data for hand gestures can be hard to obtain so one-shot learning will also be of preference. One key aspect not yet discussed is the training data, which is crucial for any machine learning application.

Training Data

Finally there is training data (the examples to be trained on) that needs to be discussed. Every model is as good as the data it get trained on, no matter how good the network is. The quality of the data, and the annotation of it (labeling and featurizing) is therefore the most important part of any deep learning application. Collecting and preprocessing data (making it workfriendly) can be very time consuming and there's different tools to faciliate this process.

Transfer Learning

Training Data

As previously mentioned, training a model requires thousands and thousands of images and could take weeks. Luckily there is a way to overcome this issue called transfer learning. There are several pre-trained open source models that one can utilize that has been trained on different datasets containing millions of images of thousands of different objects. One can then use these models, replacing the last layer with one fitting ones training data and retrain the model on this. Since it has already learned to classify objects, the transition from classifying the original objects to the new ones doesn't then require that much data and can be done rather quick.

CIFAR10 Example

Basic CNN Implementation Example

Lets now finish of with an actual implementation example from the CIFAR10 dataset which consists of 10 classes of various objects. First of all the data needs to be loaded and assigned.

```
# Importing som packages
import tensorflow as tf # Machine Learning Framework

from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt # For plotting

```

```
# Pre-fixed dataset so just need to load it
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images/255.0, test_images/255.0
```

CIFAR10 Example

Basic CNN Implementation Example

Lets plot a few images (note that the images are very small (32x32) hence the blurriness).

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(12,12))
for i in range(4):
    plt.subplot(1,4,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```

CIFAR10 Example

Basic CNN Implementation Example

```python
model = models.Sequential() # Initializing the model

# Convolutional layers:
# 32 = number of output filters (dimensionality of output space)
# (3,3) = Kernel size
# Activation function for each neuron = Rectified Linear Unit (relu)
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.Conv2D(32, (3, 3), activation='relu'))

# Max pooling - reducing and compressing the image, (2,2) = pooling window size
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(32, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

# Flattens the feature map - extracts it into a long vector
model.add(layers.Flatten())

# Compressing into smaller vector of size 64
model.add(layers.Dense(64, activation='relu'))

# Reducing into smaller vector with same size as number of classes to predict
# Activation function = softmax - multiclass classification function
model.add(layers.Dense(10, activation='softmax'))

# Printing the network with all the layers
model.summary()
```

CIFAR10 Example

Basic CNN Implementation Example

Result of the created network with all its layers and number of parameters.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_74 (Conv2D)           (None, 30, 30, 32)        896
_________________________________________________________________
conv2d_75 (Conv2D)           (None, 28, 28, 32)        9248
_________________________________________________________________
max_pooling2d_46 (MaxPooling (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_76 (Conv2D)           (None, 12, 12, 32)        9248
_________________________________________________________________
max_pooling2d_47 (MaxPooling (None, 6, 6, 32)          0
_________________________________________________________________
conv2d_77 (Conv2D)           (None, 4, 4, 64)          18496
_________________________________________________________________
conv2d_78 (Conv2D)           (None, 2, 2, 64)          36928
_________________________________________________________________
max_pooling2d_48 (MaxPooling (None, 1, 1, 64)          0
_________________________________________________________________
flatten_9 (Flatten)          (None, 64)                0
_________________________________________________________________
dense_18 (Dense)             (None, 64)                4160
_________________________________________________________________
dense_19 (Dense)             (None, 10)                650
=================================================================
Total params: 79,626
Trainable params: 79,626
```

CIFAR10 Example

Basic CNN Implementation Example

Activating the network by choosing optimization function (using default configuration) and training it with certain hyperparameters (parameters that is tuned to increase accuracy - i.e epochs and batch size)

```
# Compiling the model = "activating it" - using Adam optimization function
# loss - used for multiclass prediction
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])


# "Fitting" the model = training
# train_images = images (x-data), train_labels = correct label for each training image (y-data)
# epochs = number of iterations, batch_size = number of images at a time
# validation data = x and y data to be used for evaluation after each epoch, unused during training
history = model.fit(train_images, train_labels, epochs=20, batch_size = 32,
                    validation_data=(test_images, test_labels))
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/20
50000/50000 [==============================] - 35s 703us/sample - loss: 1.5921 - acc: 0.4104 - val_loss: 1.2902 - val_acc: 0.53
49
Epoch 2/20
16032/50000 [========>.....................] - ETA: 21s - loss: 1.2243 - acc: 0.5588 ETA: 22s - loss:
```

CIFAR10 Example

Basic CNN Implementation Example

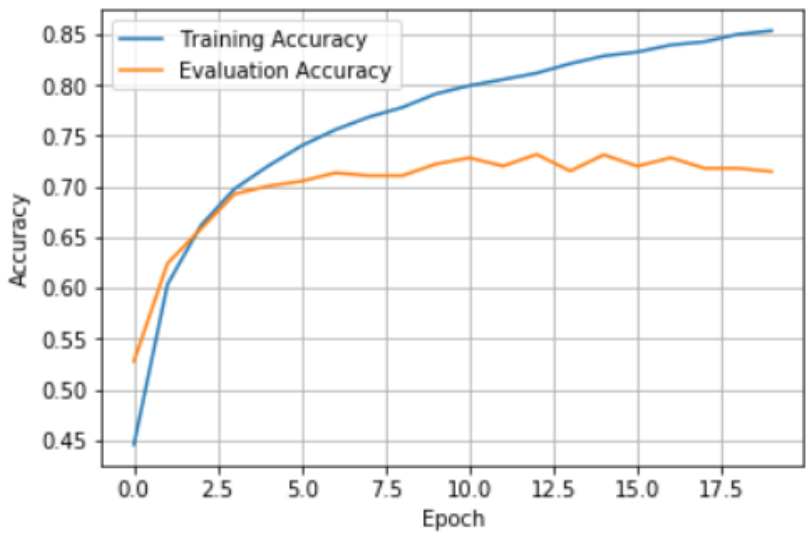
Plotting the result of the training process and testing the model on unseen data.

```
# Plotting the training accuracy and validation accuracy for each epoch

plt.plot(history.history['acc'], label='Training Accuracy')
plt.plot(history.history['val_acc'], label = 'Evaluation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='upper left')
plt.grid('on')
plt.show()


# Testing the fully trained model on unseen images
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=1)
print('Accuracy on unseen test images: {0:.2f}%'.format(100*test_acc))
```

CIFAR10 Example

Basic CNN Implementation Example

As seen in the plot, this newtork performes quite poorly with an evaluation accuracy on 71%. This is a perfect example of overfitting, where the model fits the training data too well and has therefore difficulties to perform on new unseen data (orange line has reached a plateau while the blue is still increases). This issue is resolved by tuning the hyperparameters or changing the architecture. An okey accuracy should be $> 90\%$.




```
10000/10000 [==============================] - 3s 293us/sample - loss: 1.0088 - acc: 0.7147
Accuracy on unseen test images: 71.47%
```

Hopefully, this makes you understand the mechanisms behind image classification and some basic Computer Vision tasks. Note however that this is just an introduction and it requires alot more understanding (particularly the preprocessing part when working with raw data), but this will be enough to start playing around and perform simple CNN tasks.