Applied Deep Learning

Florian Muellerklein
fmuellerklein@minerkasch.com

MINER & KASCH
DATA SCIENCE

Contents

- Introduction
  - Why is deep learning useful?
  - What is a neuron / neural network?
- Theory
  - How to make a neural network
  - How to train
  - Representation learning
- Convolutional Neural Networks
- Code Introduction
  - Keras
  - TF-Learn
- Practice
  - Tips and tricks for better training
- Applications of ConvNets
  - Images
  - Text
  - Audio
- Recurrent Neural Networks
  - Example Code
- Word Embeddings
  - Example Code

What is deep learning?

Conventional machine learning algorithms have some problems with feature engineering:

- Time consuming
- Error prone
- Difficult for complex data types (images, text)
- Requires domain expertise

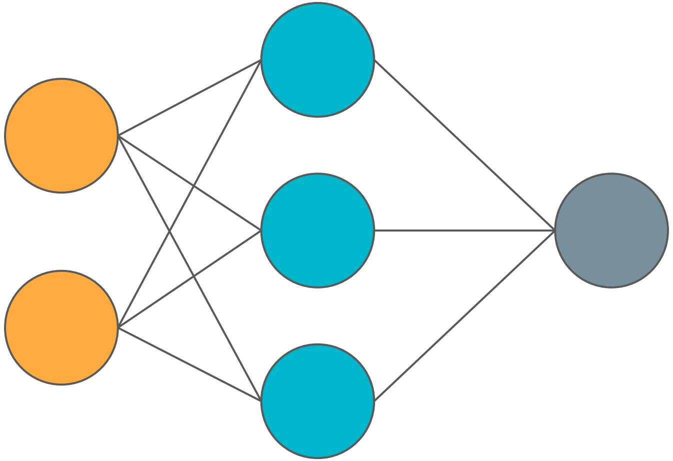
With Deep Learning:

- Model learns its own way to represent the features
- Able to work on data in a more raw form

http://www.cs.toronto.edu/~hinton/absps/NatureDeepReview.pdf

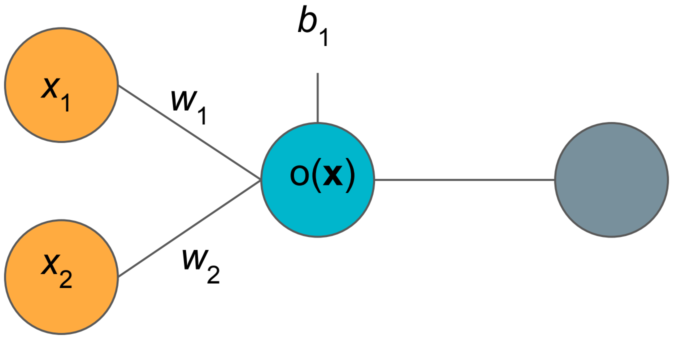
What is deep learning?

Basically, it’s a rebranding of neural networks.



Starting simple

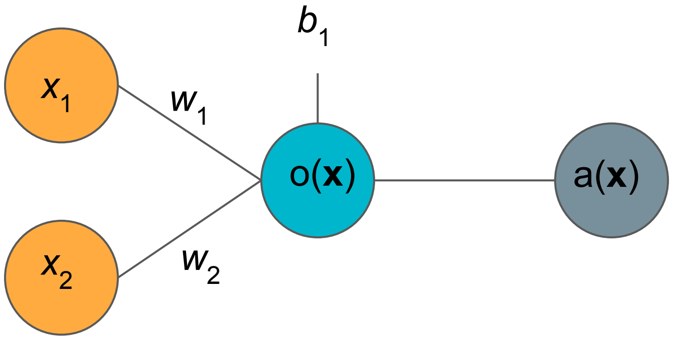
The first neural network was basically a single mathematical function that was inspired by biological neurons.



$$o(x) = b + \sum_{i}^{n} x_i w_i$$

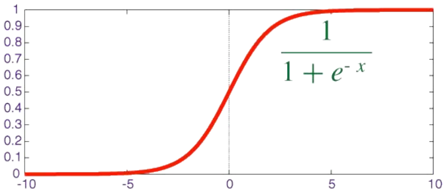
Starting simple

We typically apply a function to the output of that weighted sum

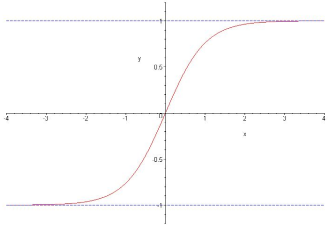


$$a(x) = a(o(x)) = a(b + \sum_{i}^{n} x_i w_i)$$

Activation Functions

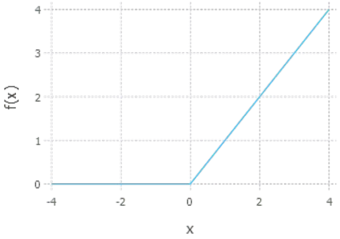


Sigmoid Function - popular in the past but has fallen out of favor



$$\frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Tanh Function - similar to sigmoid except has the benefit of being centered on 0, still common in RNN



Rectified Linear Unit (ReLU) - Most common activation function now. It’s simply max(x, 0)

Is it useful?

Let’s say that we have three functions $f^{(1)}$,$f^{(2)}$, and $f^{(3)}$ all chained together.

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$

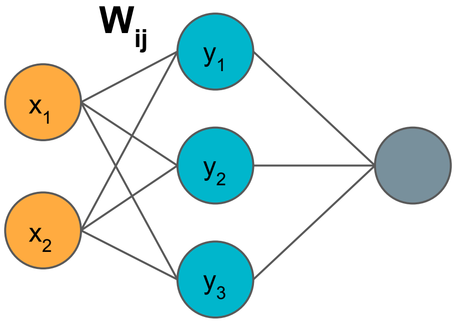
What is deep learning?

The chaining of these linear combinations turns out to be a very powerful idea. It allows the network to create meaningful abstractions and representations.

The functions for each node are the same as the ones from earlier.

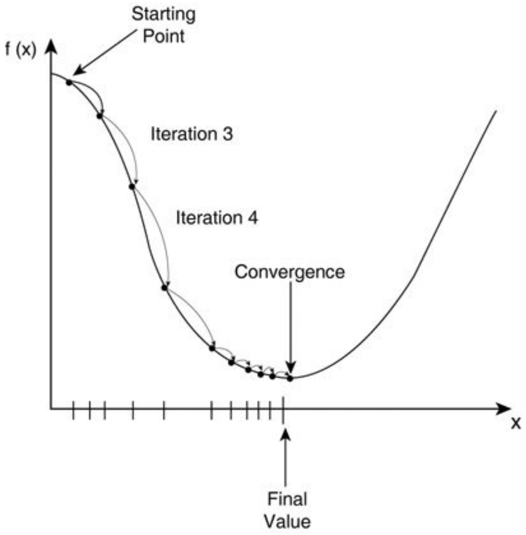
Except with different weights and the input to each successive node is the output from the previous node.

$$a(x) = a(o(x)) = a(b + \sum_{i}^{n} x_i w_i)$$



How to train a network?

Almost every deep learning model is trained by an algorithm called Stochastic Gradient Descent.



$$F(k) = (t(k) - a(k))^2$$

Backpropagation

The process to updating the weights with each stochastic gradient calculation is called backpropagation.

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$

When we make predictions or outputs from the network we feed the data forward through these functions.

To update the weights we just go backwards through each functions and take the derivatives along the way. It's basically an exercise in the chain rule of calculus.

Backpropagation

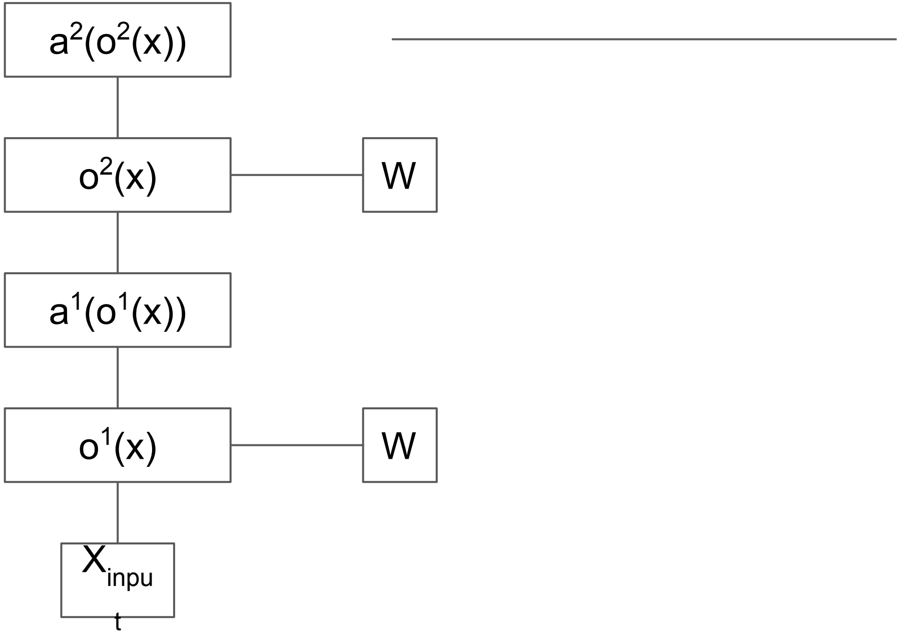
Let's say we have have functions *y* = *g*(x) and *z* = *f*(*g*(x)) = *f*(*y*). To find the derivative of *z* w.r.t. to *x* we have to move through the intermediate function *y*.

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

If you are familiar with the chain rule for finding derivatives, then you can think of backpropagation as an application of the chain rule.

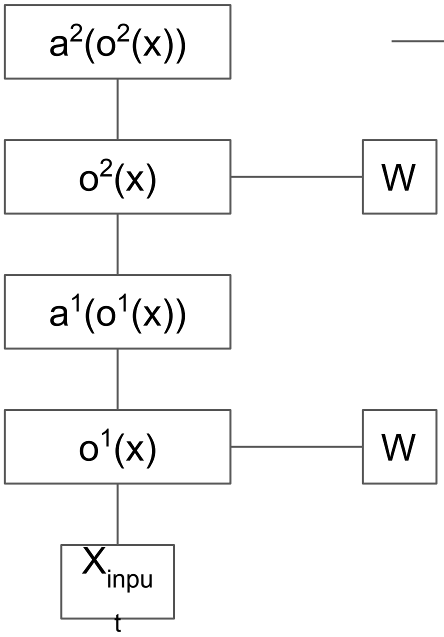
Backpropagation

$$F(x) = (t(x) - a^2(x))^2$$



$$a^2(x) = a^2(o^2(x))$$

Backpropagation



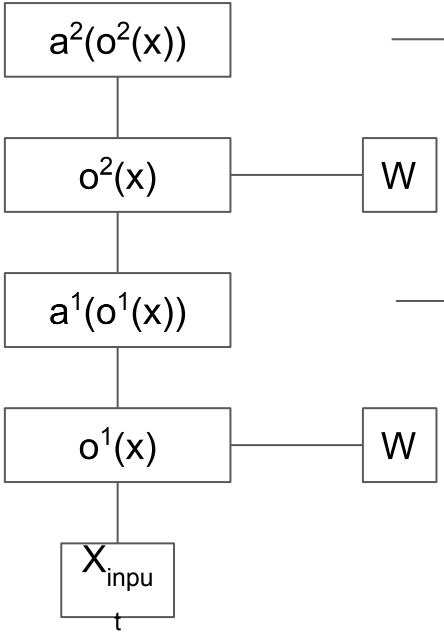
$$F(x) = (t(x) - a^2(x))^2$$

$$a^2(x) = a^2(o^2(x))$$

$$o^2(x) = b + \sum_i^n x_i w_i \qquad ; x_i = a^1(o^1(x))$$

Backpropagation

$$F(x) = (t(x) - a^2(x))^2$$



$a^2(o^2(x))$

$o^2(x)$

W

$a^1(o^1(x))$

$o^1(x)$

W

X_{input}

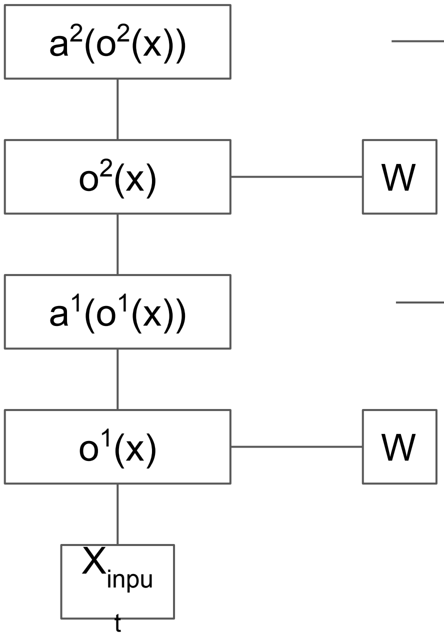
$$a^2(x) = a^2(o^2(x))$$

$$o^2(x) = b + \sum_{i}^{n} x_i w_i \qquad ; x_i = a^1(o^1(x))$$

$$x = a^1(o^1(x))$$

Backpropagation

$$F(x) = (t(x) - a^2(x))^2$$



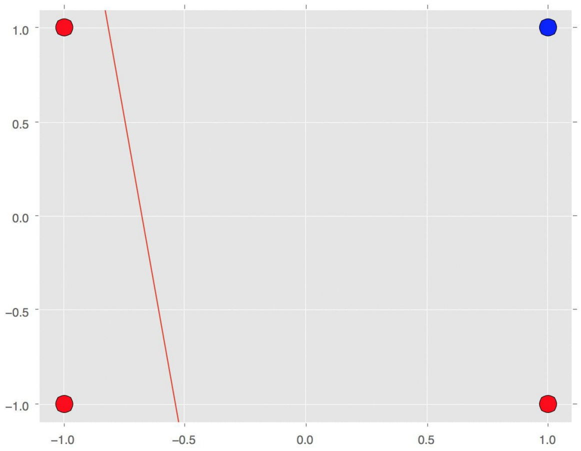
$$a^2(x) = a^2(o^2(x))$$

$$o^2(x) = b + \sum_{i}^{n} x_i w_i \qquad ; x_i = a^1(o^1(x))$$

$$x = a^1(o^1(x))$$

$$o^1(x) = b + \sum_{i}^{n} x_i w_i \qquad ; x_i = x_{input}$$

Iterative training



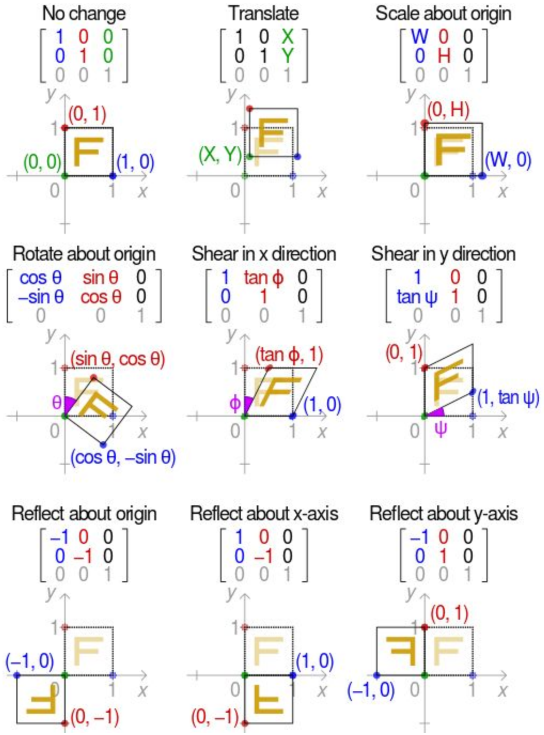
Here is an animation showing a single linear unit finding a linear boundary to separate two classes.

How does the representation happen?

We can think of each of those linear combinations as doing a simple affine transformation.

If we take our input vector **x** and apply our weight matrix **W** to it we can map our input data into new places in space.

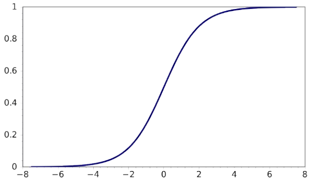
$$x \mapsto Wx + b$$



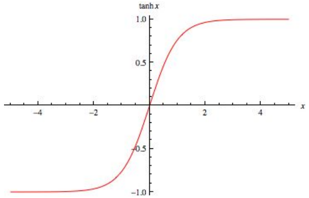
Note: in affine transformations the lines stay parallel

Source: https://en.wikipedia.org/wiki/Affine_transformation

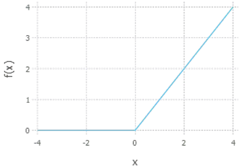
How does the representation happen?



http://tikalon.com/blog/blog.php?article=2011/sigmoid



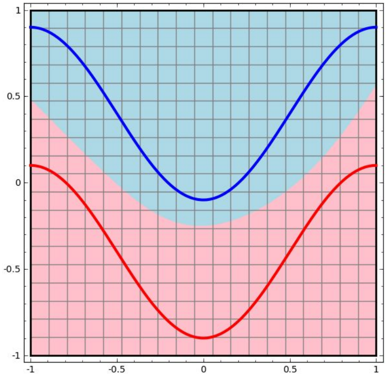
http://mathworld.wolfram.com/HyperbolicTangent.html

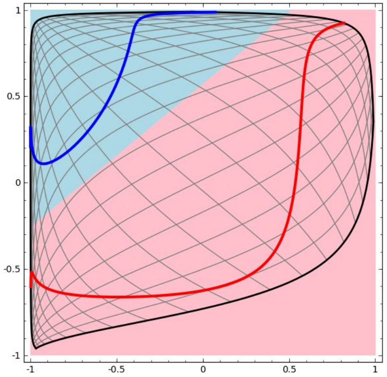


http://int8.io/neural-networks-in-julia-hyperbolic-tangent-and-relu/

What is deep learning?

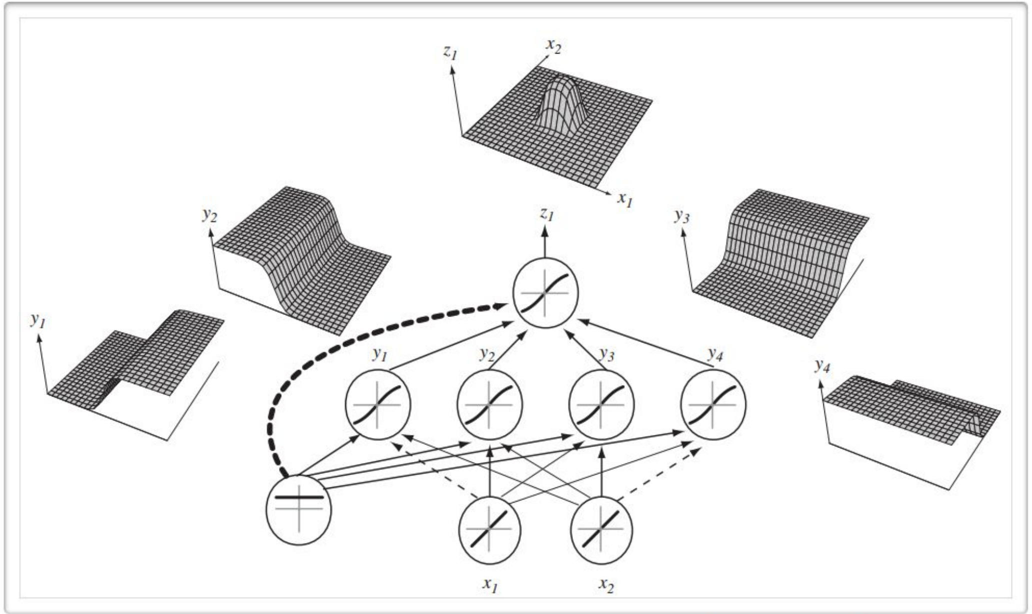
A series of equations in the form $y_1 = x_1w_{11} + x_2w_{12} + b_1$ can only really perform affine transformations on the input data. Meaning that the grid lines on the plot on the left will always 'look' parallel to each other through each transformation. Introducing the 'nonlinearity' function *f* allows for the warping.





Source: Christopher Olah (Google Brain) http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

How does the representation happen?

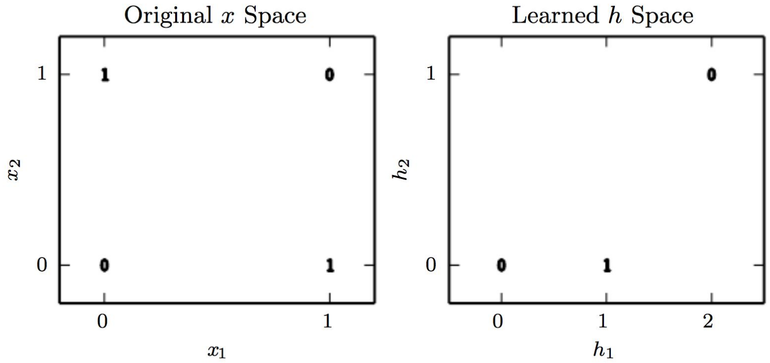


(from Pascal Vincent's slides)

http://www.cs.cmu.edu/~rsalakhu/10807_2016/Lectures/Lecture_NN_Part1.pdf
Who in turn took this slide from another presentation (Pascal Vincent)

What do these representations look like?

One of the simplest examples of what a neural network can solve that a linear model can't is the XOR problem.

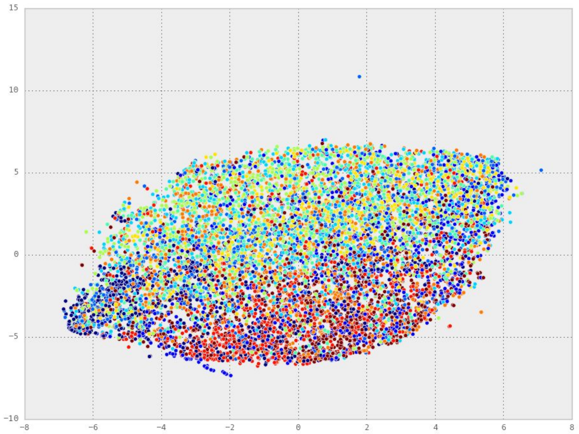


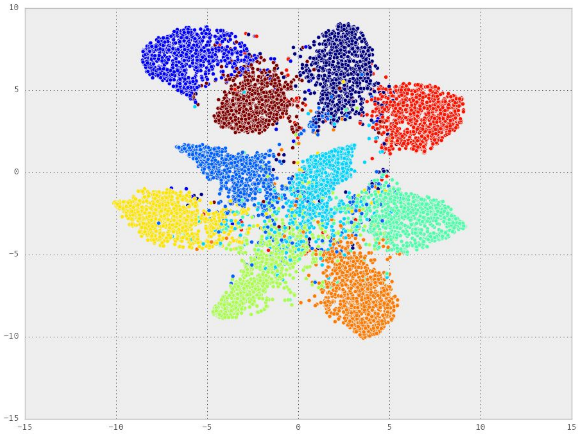
On the left we have the original dataset, on the right we have the learned feature representations that a multi-neuron network can learn.

Source deep learning book: http://www.deeplearningbook.org/

What do these representations look like?

Here is a similar test that I did with some natural images. I did some unsupervised clustering on the raw images and then again on the features extracted from the final layer of a deep neural network.





What do these representations look like?

That network that allowed for the images to be so easily categorized was 16 layers deep and had 133 million weight parameters.

a(x) = ¯_(ツ)_/¯(x)

Representation learning

Learning the best way to represent the input data is the real strength of deep learning.

Typical machine learning applications involve feature engineering and all kinds of clever thinking.

Deep learning promises to learn the features for you and to do it in a way that also trains the final classifier at the same time. All it takes is some time and calculus.

References on vectorized representation of data

Decoding the Thought Vector

http://gabgoh.github.io/ThoughtVectors/

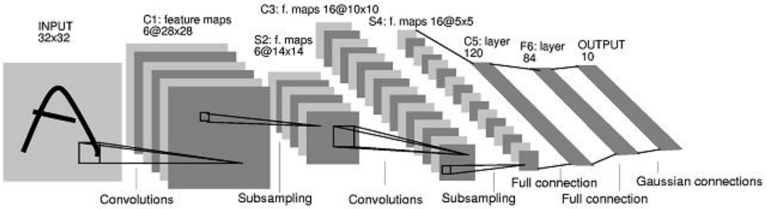
Christopher Olah: Neural Network Manifolds

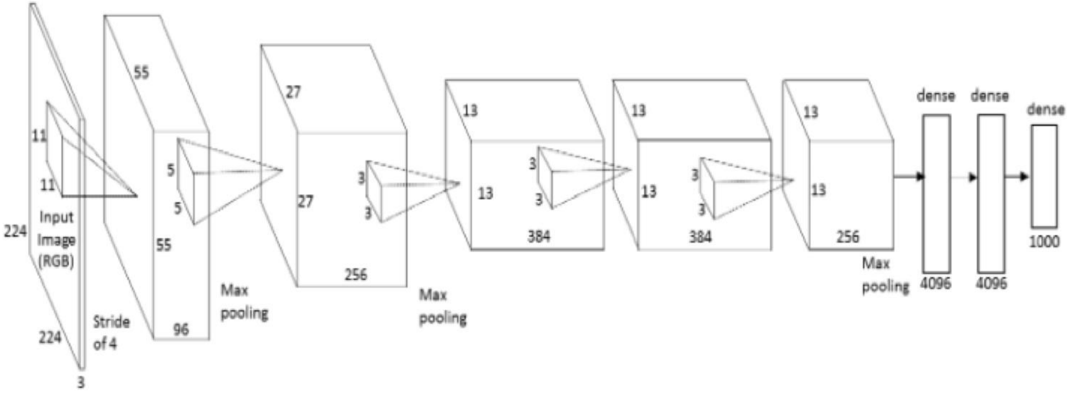
http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Convolutional Neural Networks

ConvNets

Convolutional Neural Networks (ConvNets) are a special type of feedforward neural network that are particularly well suited for data which contain some kind of spatial structure.



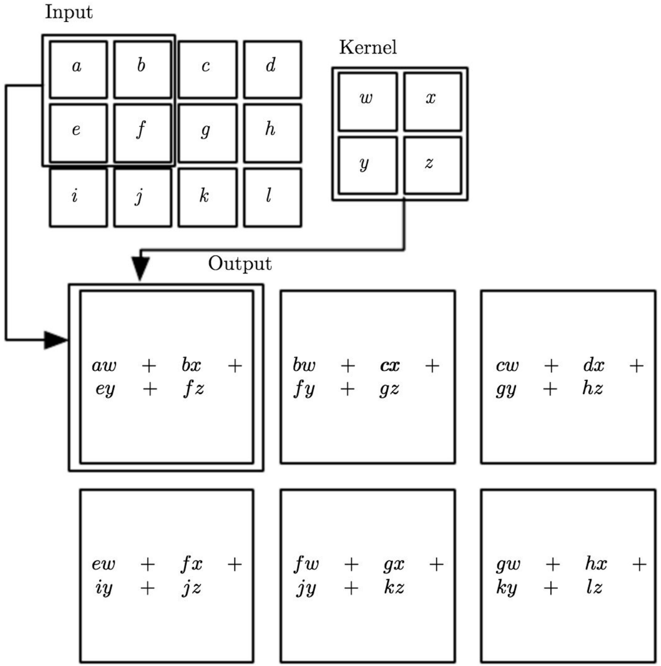


ConvNets

Instead of using a simple linear model for each node a ConvNet will use kernels that are applied to data.

Each kernel has their own weights and biases just like the normal types of layers that we looked at earlier.

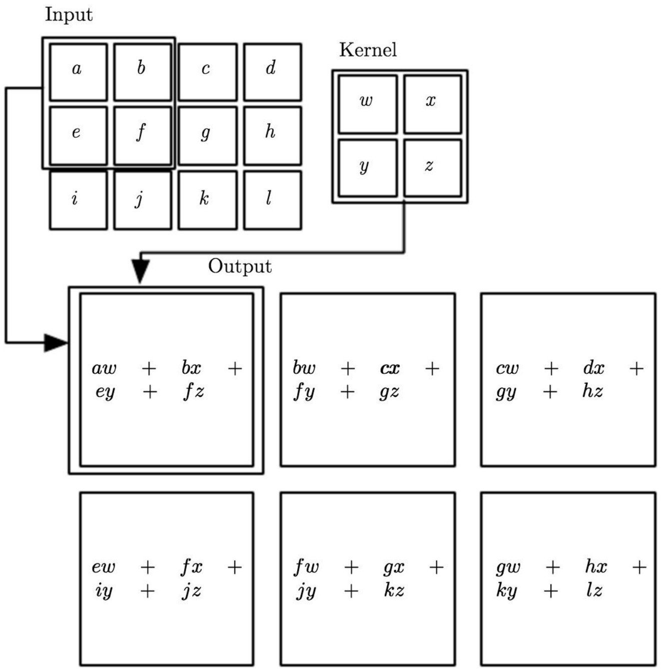
ConvNets



The kernels are applied like a sliding window along the input data.

The output of that operation will be the sum of each kernel weight multiplied by each corresponding input data point.

ConvNets

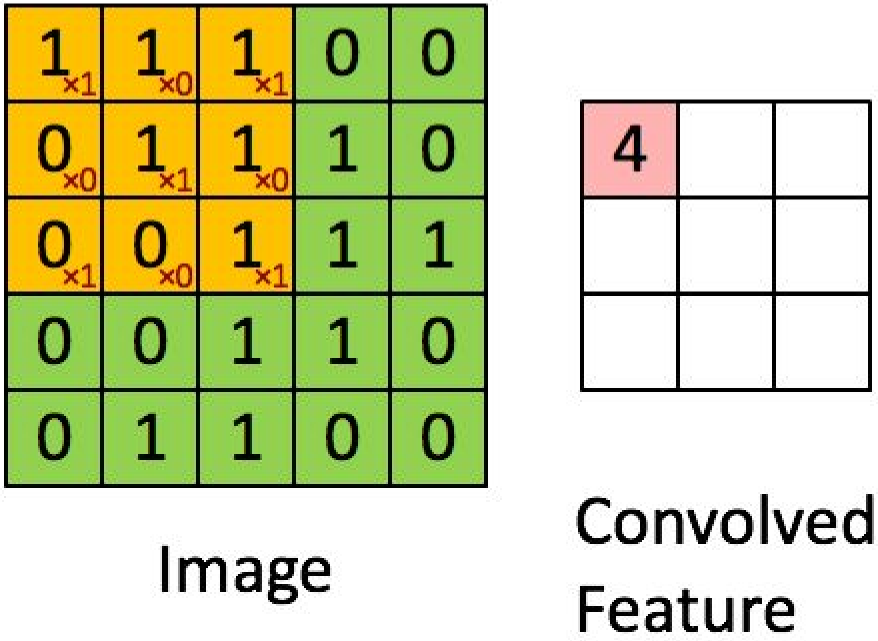


A single number is the output for each unique position that the kernel can occupy.

In this example, sliding a 2x2 kernel over a 3x4 matrix results in a new 2x3 matrix.

We call that new 2x3 matrix a feature map and it can then become the input to a following kernel.

ConvNets



1 ×1	1 ×0	1 ×1	0	0
0 ×0	1 ×1	1 ×0	1	0
0 ×1	0 ×0	1 ×1	1	1
0	0	1	1	0
0	1	1	0	0

Image

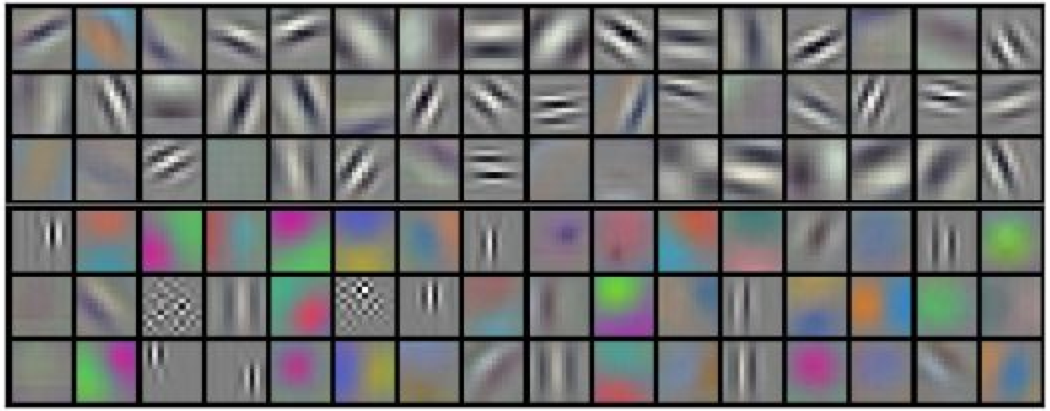
4		

Convolved Feature

http://deeplearning.stanford.edu/tutorial/

ConvNets

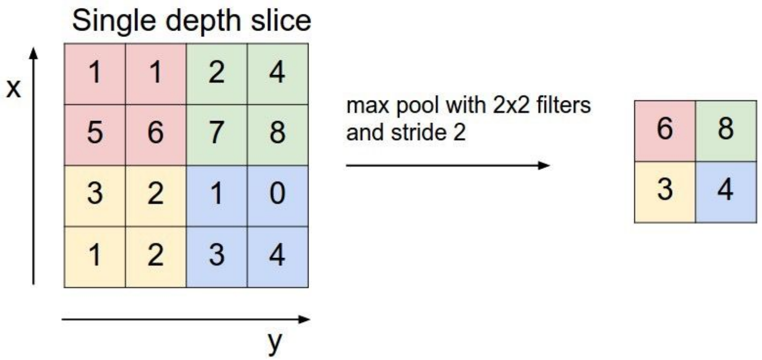
Looking at the first layer of a well trained ConvNet will often yield kernels that look like this.



Pooling

Another useful tool for ConvNets are pooling layers.

The most common type of pooling is max pooling.



Single depth slice

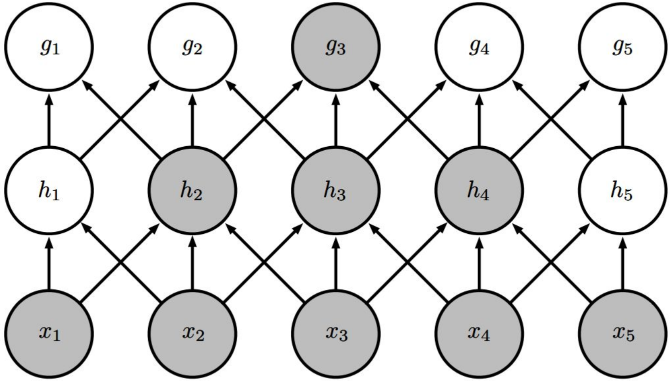
1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pool with 2x2 filters and stride 2

6	8
3	4

http://cs231n.github.io/convolutional-networks/

Pooling



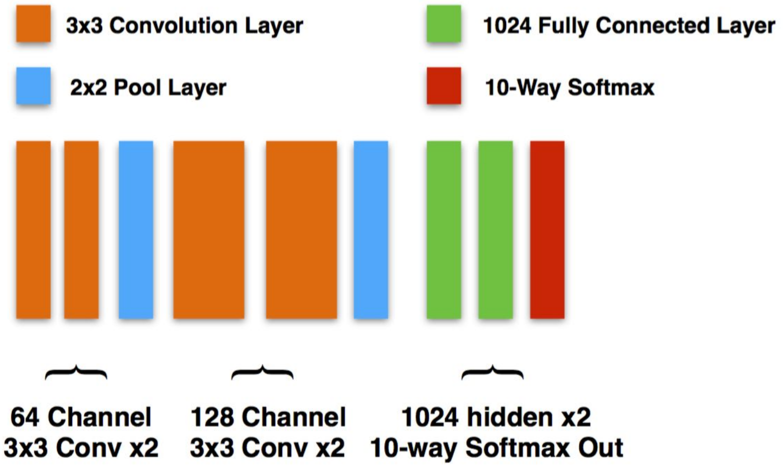
We can see that g_3 takes three neurons as it's input, but each of those three also take three. So g_3 receives information from a much wider area than just its immediate connections.

Putting the layers together

Typically people will stack a few convolution layers together then insert a pooling layer.

Since pooling layers down-sample your image size, we can go as far as possible then insert a fully-connected layer or two. Those fully-connected layers are the same as an MLP.

Finally we tack on a classifier or regression layer depending on the problem.



Learning

SGD!

ConvNet References

Stanford: Convolutional Neural Networks for Visual Recognition

http://cs231n.github.io/convolutional-networks/

Deep Learning Book: ConvNets Chapter

http://www.deeplearningbook.org/contents/convnets.html

Michael Nielsen’s Neural Networks and Deep Learning ConvNet Chapter

http://neuralnetworksanddeeplearning.com/chap6.html

Convolutional Neural Networks Use Cases

What can we do with a ConvNet?

1) **Image Classification**: are these pictures of dogs, cars, people, …?
2) **Image Similarity**: Clustering, reverse image search
3) **Object Localization**: where is a specific object, where are certain landmarks, find every object of a certain type.
4) **Self Driving Car Lane Assist**: Predict steering angle.
5) **Audio Classification**: music genre classification, transcribing music, transcribing speech
6) **Text**: classification, sentiment analysis, translation

Image Classification

Applying convolutional neural networks to a novel image classification task typically follows a standard workflow.

1) We obtain a labeled dataset with examples for our problem.
    a) Can be things like a folder with a lot of cat pictures and another with a lot of dog pictures.
2) Find an appropriate pretrained model.
    a) The groups that are pushing deep learning performance are also very generous with their models. They typically release their highest performing models which turn out to be very good general feature extractors. All we have to do is slightly tweak them to be useful for our problems. This allows us to have state of the art performance with much less data.

Image Classification

3. Either fine tune the network on the new dataset or just use the pretrained network as a feature extractor to vectorize the images that we have.

a. The problems that researchers tend to solve are very general. You don’t want to evaluate a state of the art computer vision algorithm on only images of one type. You could never know if your algorithm was actually improving the state of the art in general or if it’s only good at one thing, like classifying cars or dogs.

b. This benefits us, as people trying to apply machine learning to real problems we are the ones that are looking to only do well at a specific task. So starting with a very general model we only need to do some small tweaks to get there.

c. We can take a state of the art model and repurpose it in just a few minutes

Image Clustering

Applying convolutional neural networks to find similar images is a great exercise in exploiting the central philosophy of deep learning.

As we saw earlier today, deep learning is really about finding good representations of our data.

We can exploit this property by extracting the internal representation of each of our images.

Image Clustering

This image from earlier is a perfect example of how successful clustering can be with the internal representations of ConvNets.

We just return the activation from the layer before the final classifier layer to get these representations.

Using a state of the art pretrained model the only thing that we have to do is pass each image through that model. No training needed!

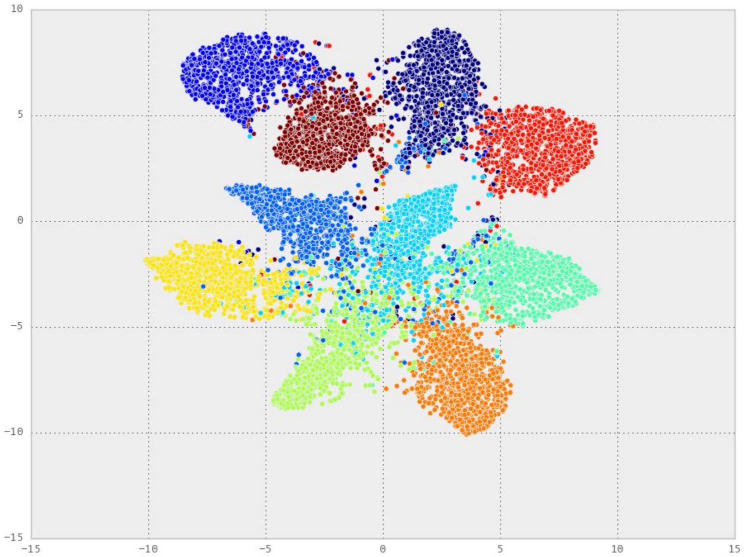
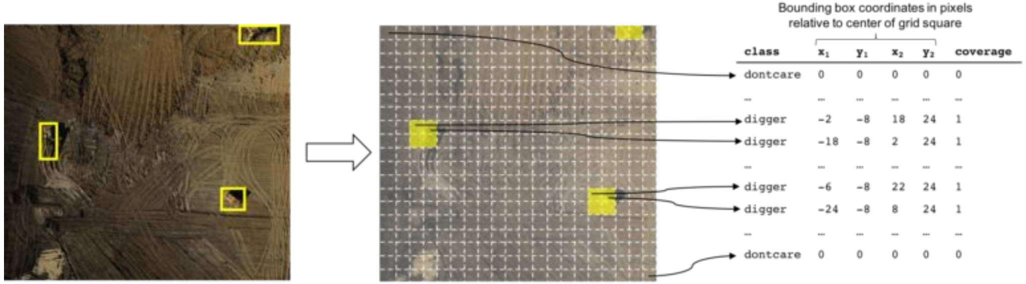


Image Localization

Convolutional neural networks are also great for localization.



Training image with bounding box annotations

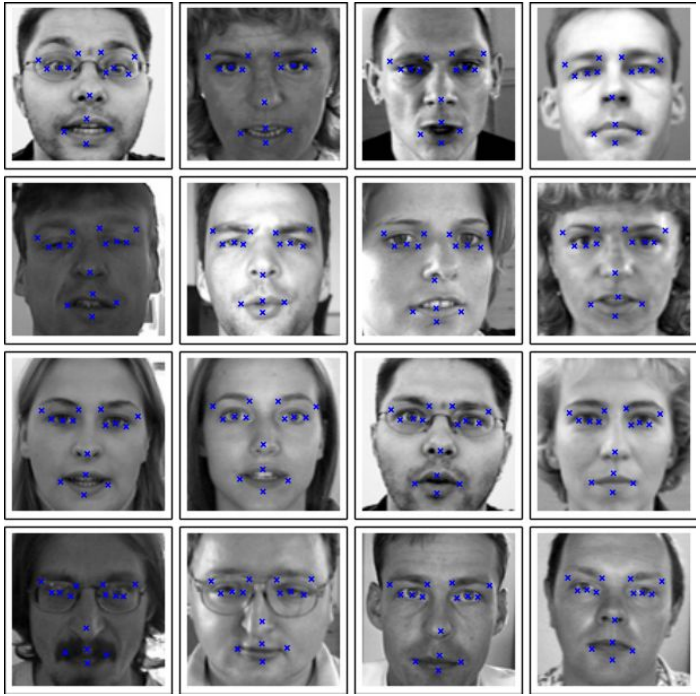
Bounding boxes mapped to grid squares

Bounding box coordinates in pixels relative to center of grid square

class	x_1	y_1	x_2	y_2	coverage
dontcare	0	0	0	0	0
...	...	...	...	...	...
digger	-2	-8	18	24	1
digger	-18	-8	2	24	1
...	...	...	...	...	...
digger	-6	-8	22	24	1
digger	-24	-8	8	24	1
...	...	...	...	...	...
dontcare	0	0	0	0	0

DetectNet input data representation

https://devblogs.nvidia.com/parallelforall/detectnet-deep-neural-network-object-detection-digits/



http://danielnouri.org/notes/2014/12/17/using-convolutional-neural-nets-to-detect-facial-keypoints-tutorial/

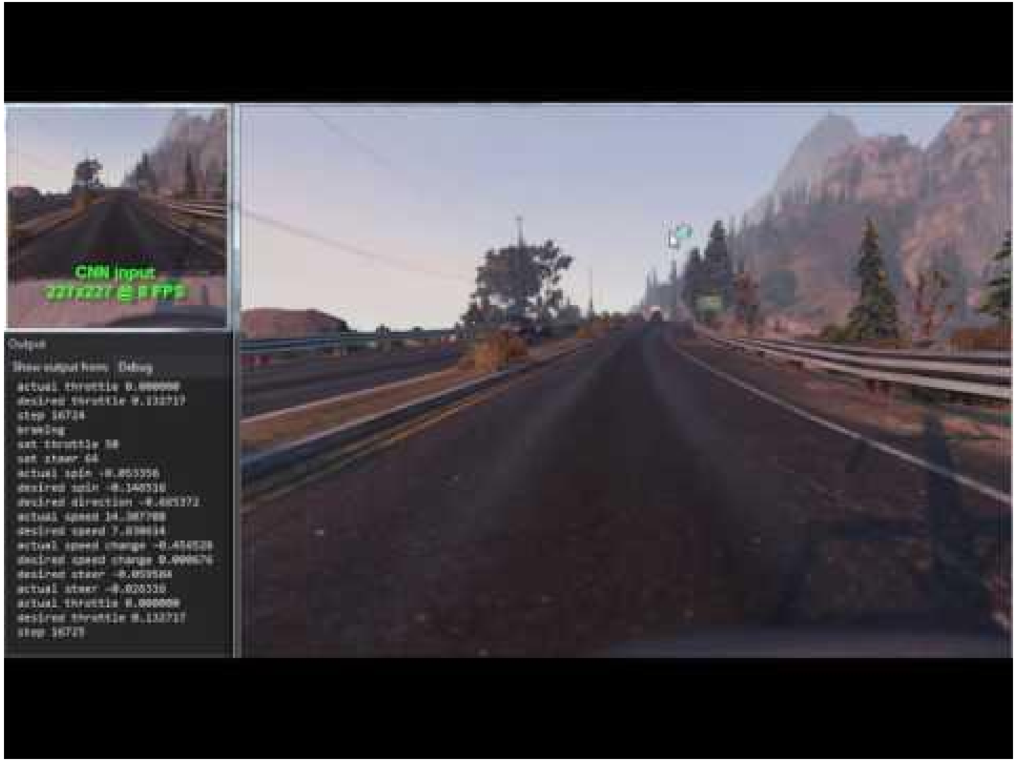
Image Localization

Instead of predicting labels for each image we are instead predicting points or regions.

We need to have annotated images to train the network with, could be coordinates of the key points or coordinates of the bounding boxes.

If we are doing regions, we convert the coordinates into those discrete objects and then train the network as if it were a classification problem. Predicting a yes/no for each region.

Self Driving Car

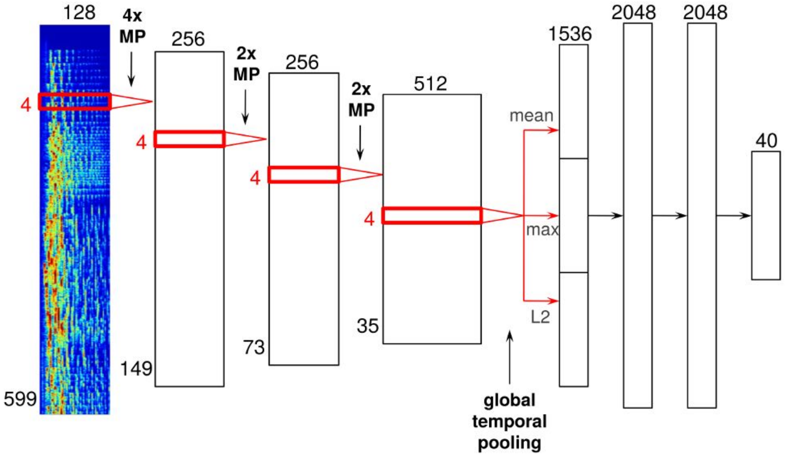


```
actual throttle 0.000000
desired throttle 0.132717
step 16724
braking
set throttle 50
set steer 64
actual spin -0.053356
desired spin -0.140316
desired direction -0.485372
actual speed 14.387708
desired speed 7.030614
actual speed change -0.456528
desired speed change 0.000676
desired steer -0.055584
actual steer -0.028338
actual throttle 0.000000
desired throttle 0.132717
step 16725
```

http://deepdrive.io/

Audio Classification

ConvNets are also great at dealing with audio classification.



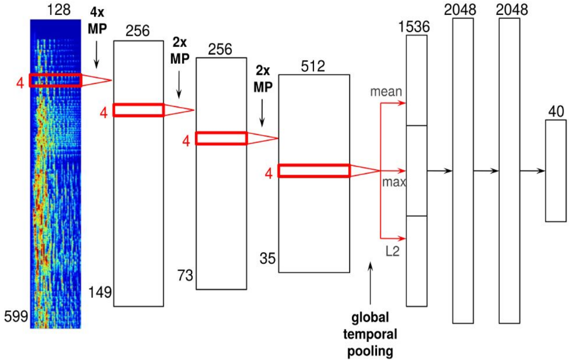
http://benanne.github.io/2014/08/05/spotify-cnns.html

Audio Classification

This ends up becoming a very similar workflow to image classification.

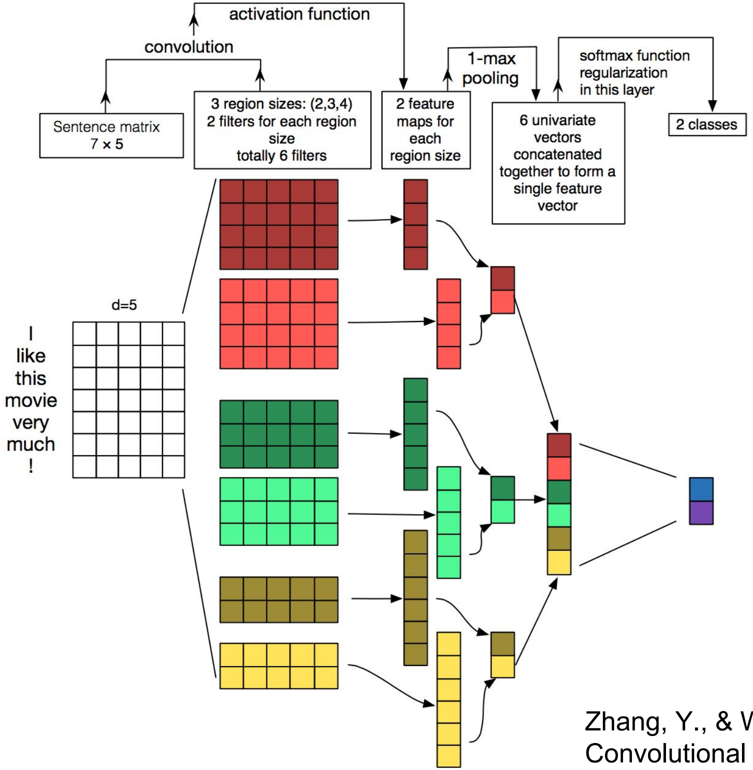
By working with the audio spectrogram the neural network will process it in almost exactly the same way that it does with an image.

The differences being that it's only a 2D matrix instead of 3D.



http://benanne.github.io/2014/08/05/spotify-cnns.html

Text Classification


activation function
convolution
1-max pooling
softmax function regularization in this layer
Sentence matrix 7 × 5
3 region sizes: (2,3,4) 2 filters for each region size totally 6 filters
2 feature maps for each region size
6 univariate vectors concatenated together to form a single feature vector
2 classes
d=5
I
like
this
movie
very
much
!

Although it may seem a little counter intuitive, convolutional neural networks can be very successfully applied to natural language problems.

The intuition here is that a sentence can be represented as a structured matrix.

Zhang, Y., & Wallace, B. (2015). A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification.

Code Introduction

Theano and Tensorflow

Theano and TensorFlow are the two main frameworks in python for creating deep learning systems.

Theano came first, from U.Montreal, and it should be thought of as a symbolic math library with some properties that make it very useful for deep learning (automatic differentiation and gpu support for matrix algebra).

TensorFlow, from Google, is much newer but is very similar to Theano. It’s still a symbolic math library but learned a bit from some of Theano’s problems.

Both libraries are very low level, need to define every function yourself. Makes work tedious.

Keras

“Keras is a minimalist, highly modular neural networks library, written in Python and capable of running on top of either TensorFlow or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.

Use Keras if you need a deep learning library that:

Allows for easy and fast prototyping (through total modularity, minimalism, and extensibility).

Supports both convolutional networks and recurrent networks, as well as combinations of the two.

Supports arbitrary connectivity schemes (including multi-input and multi-output training).

Runs seamlessly on CPU and GPU.”

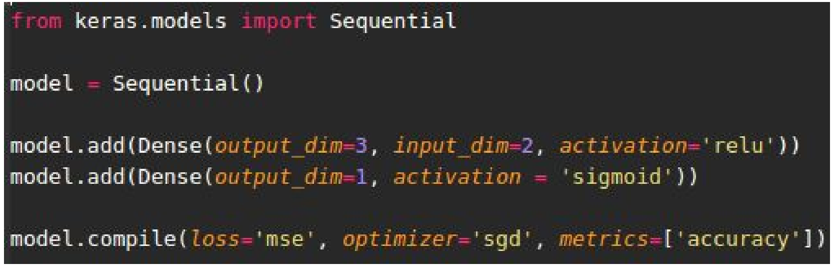
Source: https://keras.io/

Keras

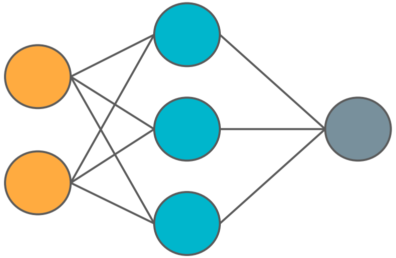
Sequential api?

Functional api?

Sequential Models



```
from keras.models import Sequential

model = Sequential()

model.add(Dense(output_dim=3, input_dim=2, activation='relu'))
model.add(Dense(output_dim=1, activation = 'sigmoid'))

model.compile(loss='mse', optimizer='sgd', metrics=['accuracy'])
```

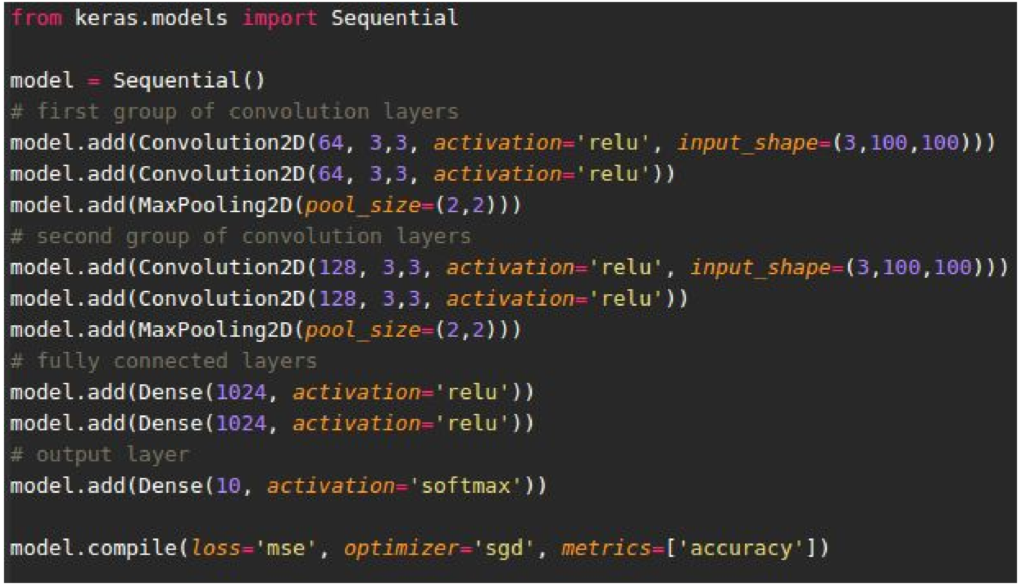


With the sequential model you start by declaring your model as a sequential. That gives you a blank canvas to start working.

From there you can add anything you want using model.add(). The data will flow through the model in the order that you declare each layer.

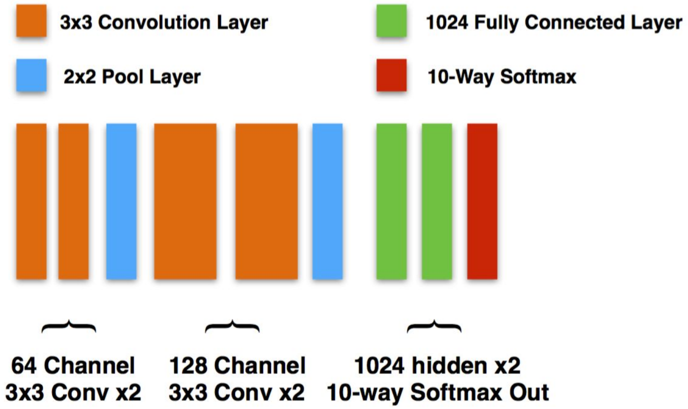
The example on the left builds the small neural network that we looked at the in the beginning.

Sequential Models

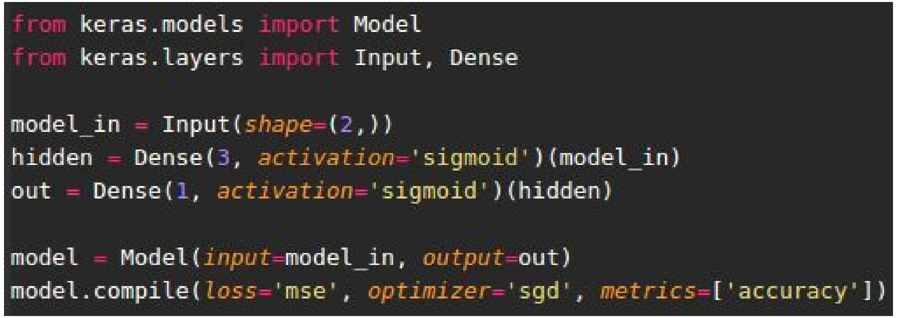


```
from keras.models import Sequential

model = Sequential()
# first group of convolution layers
model.add(Convolution2D(64, 3,3, activation='relu', input_shape=(3,100,100)))
model.add(Convolution2D(64, 3,3, activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
# second group of convolution layers
model.add(Convolution2D(128, 3,3, activation='relu', input_shape=(3,100,100)))
model.add(Convolution2D(128, 3,3, activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
# fully connected layers
model.add(Dense(1024, activation='relu'))
model.add(Dense(1024, activation='relu'))
# output layer
model.add(Dense(10, activation='softmax'))

model.compile(loss='mse', optimizer='sgd', metrics=['accuracy'])
```

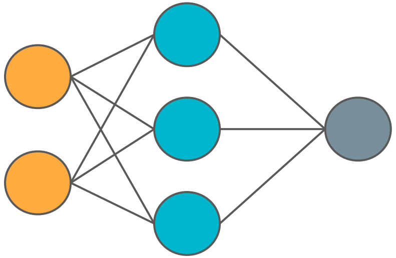
A slightly more complicated and modern looking ConvNet.



Functional Models



```
from keras.models import Model
from keras.layers import Input, Dense

model_in = Input(shape=(2,))
hidden = Dense(3, activation='sigmoid')(model_in)
out = Dense(1, activation='sigmoid')(hidden)

model = Model(input=model_in, output=out)
model.compile(loss='mse', optimizer='sgd', metrics=['accuracy'])
```

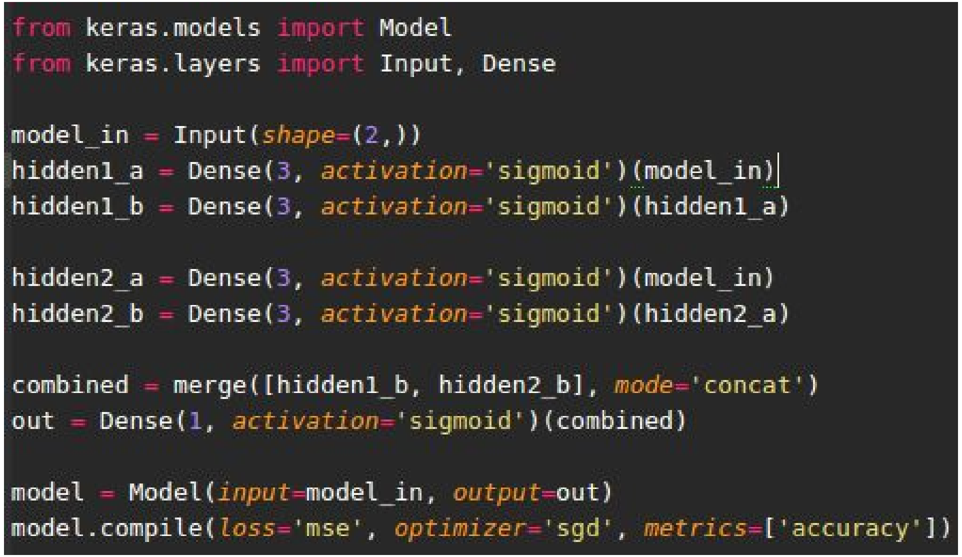


With the functional model you give each computational layer a name instead of just stacking them up with model.add().

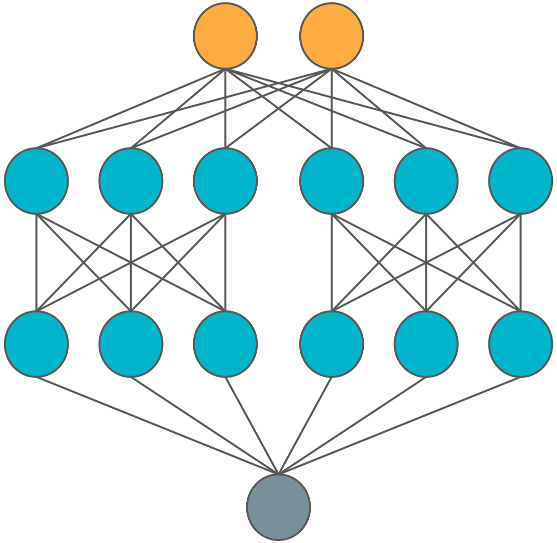
This is key, because you also explicitly provide the input to each layer. This allows you more freedom to try some different architectures with no extra effort to hack the sequential models.

For example some architectures could have multiple independent branches of neurons coming off of the input. This method makes setting that up a breeze.

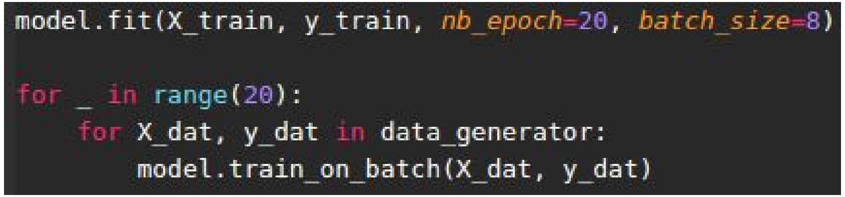
Functional Models



```
from keras.models import Model
from keras.layers import Input, Dense

model_in = Input(shape=(2,))
hidden1_a = Dense(3, activation='sigmoid')(model_in)
hidden1_b = Dense(3, activation='sigmoid')(hidden1_a)

hidden2_a = Dense(3, activation='sigmoid')(model_in)
hidden2_b = Dense(3, activation='sigmoid')(hidden2_a)

combined = merge([hidden1_b, hidden2_b], mode='concat')
out = Dense(1, activation='sigmoid')(combined)

model = Model(input=model_in, output=out)
model.compile(loss='mse', optimizer='sgd', metrics=['accuracy'])
```



Using Models

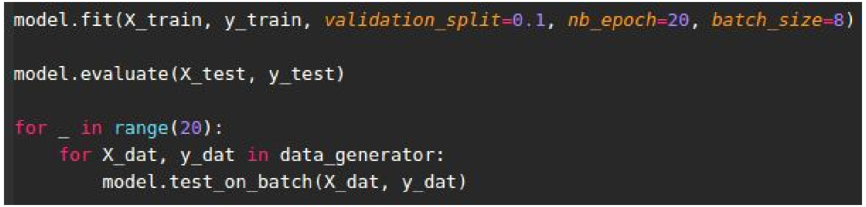


```
model.fit(X_train, y_train, nb_epoch=20, batch_size=8)

for _ in range(20):
    for X_dat, y_dat in data_generator:
        model.train_on_batch(X_dat, y_dat)
```

Training a network is an iterative process, so basically we’ll be creating loops that stream the data into each model.

We can use Keras built in methods or create our own. The ‘data_generator’ would just be some function that would loop through the data, dividing it up into batches to stream into the model.

Using Models

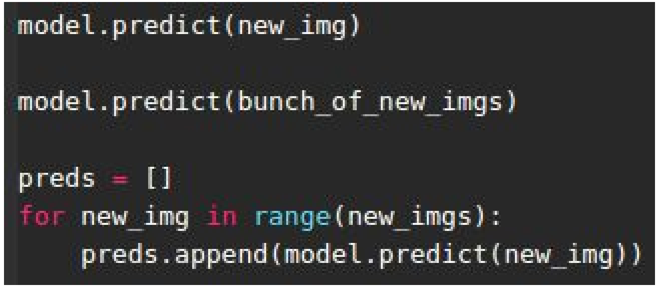


```
model.fit(X_train, y_train, validation_split=0.1, nb_epoch=20, batch_size=8)

model.evaluate(X_test, y_test)

for _ in range(20):
    for X_dat, y_dat in data_generator:
        model.test_on_batch(X_dat, y_dat)
```

With .fit() we don’t have to split the data into training and testing before using it. Just tell Keras what percentage of data we want to use to evaluate the model with.

With .evaluate() we split the data before training and use it the same way that we use .fit(). Except this time it won’t update parameters it will just return evaluation metrics.

Using Models



```
model.predict(new_img)

model.predict(bunch_of_new_imgs)

preds = []
for new_img in range(new_imgs):
    preds.append(model.predict(new_img))
```

Each model has a .predict(), it can be used in a variety of ways. Similar to how training and evaluating had a variety of methods, except each one of these will be called with .predict().

You can pass a single new point of data, a numpy array of new data.

You could also loop through the data and make a prediction on each data point.

Other Frameworks

Torch (Lua)

http://torch.ch

MXNet (everything?)

https://github.com/dmlc/mxnet

Theano (Python)

http://deeplearning.net/software/theano/

TensorFlow (Python, C++)

https://www.tensorflow.org

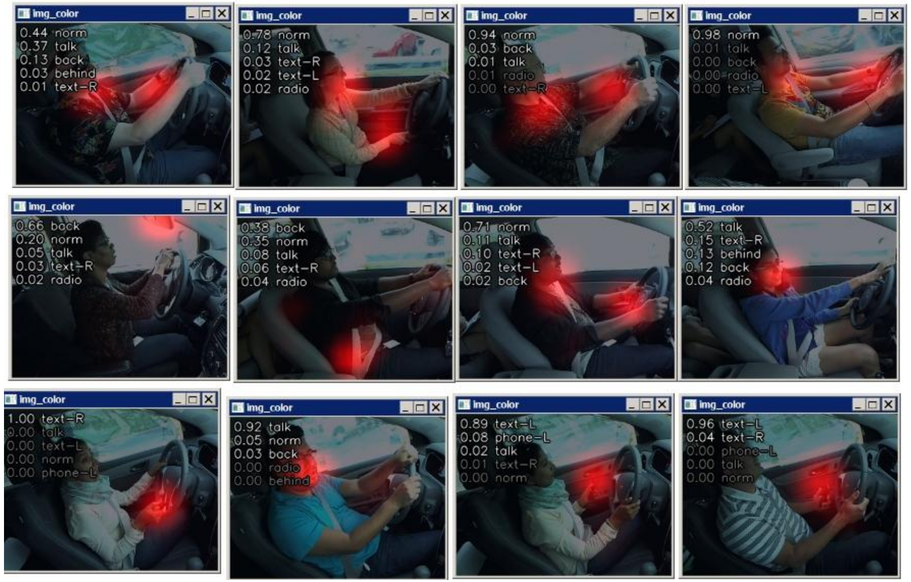
Hands on Practice

Classification Heatmaps

One technique that can be incredibly helpful when training a ConvNet for an image classification problem is to make heatmaps.

The basic idea here is to occlude parts of the image and record the effect on the output probabilities.

If you cover a part of an image and the confidence in a certain classification goes down then you can reasonably conclude that you have covered an important feature for classification.



From Kaggle user: Heng CherKeng
https://www.kaggle.com/c/state-farm-distracted-driver-detection/forums/t/21994/heat-map-of-cnn-output

Classification Heatmaps Examples

Visualizing and Understanding Convolutional Neural Networks, Matthew D Zeiler, Rob Fergus
https://arxiv.org/abs/1311.2901

Example Code from Daniel Nouri:
https://github.com/dnouri/nolearn/blob/master/nolearn/lasagne/visualize.py#L105

Example for Keras from Jacob Gildenblat: https://github.com/jacobgil/keras-cam

Word Embeddings

Word Embeddings

Word embeddings represent a major breakthrough in the way that we represent words to computers.

The biggest change is that we no longer represent words as discrete objects. Instead we represent words in a continuous semantic space. This allows us to easily handle things like; synonyms, categories (names, vehicles, …), antonyms, and others.

The main idea is that we create a shallow neural network that will learn a mapping between the words and their new embeddings.

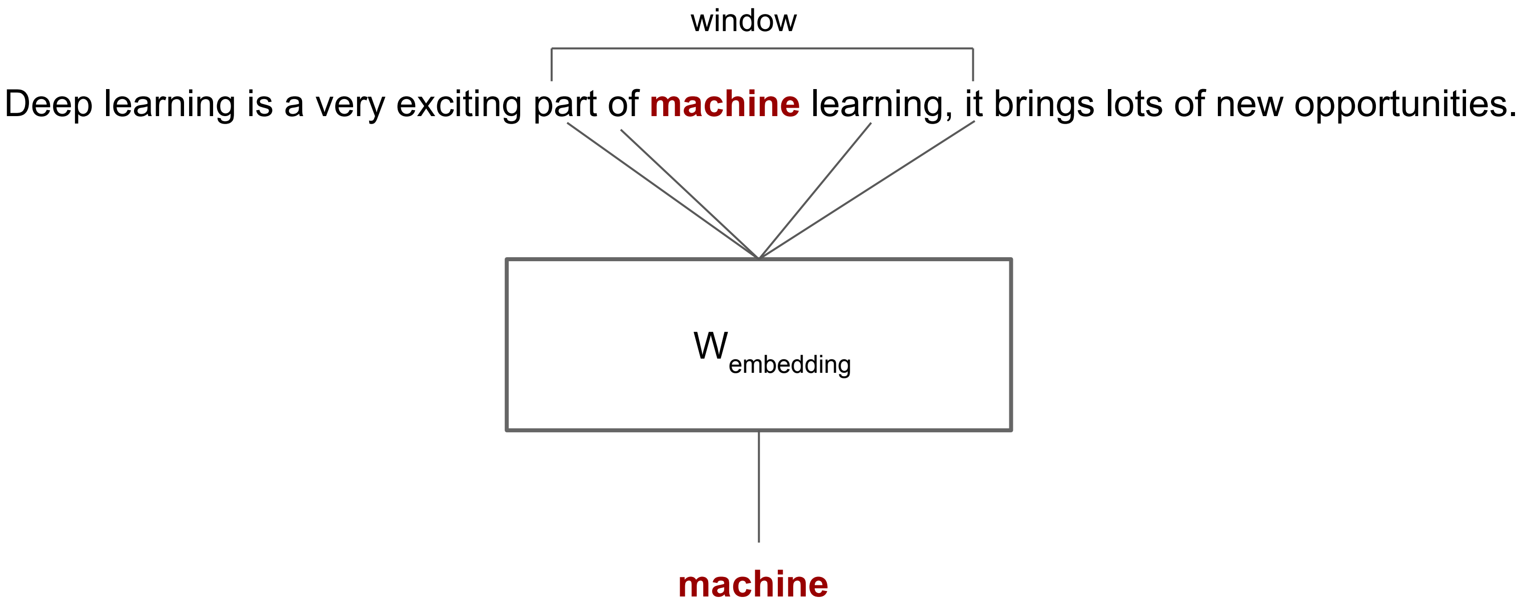
Word Embeddings

The insight that made this possible that was to treat words that are close to each other as being related.

We try to predict each word by looking at it’s surrounding context.

“You will know a word by the company that it keeps” - John Rupert Firth

Continuous Bag of Words



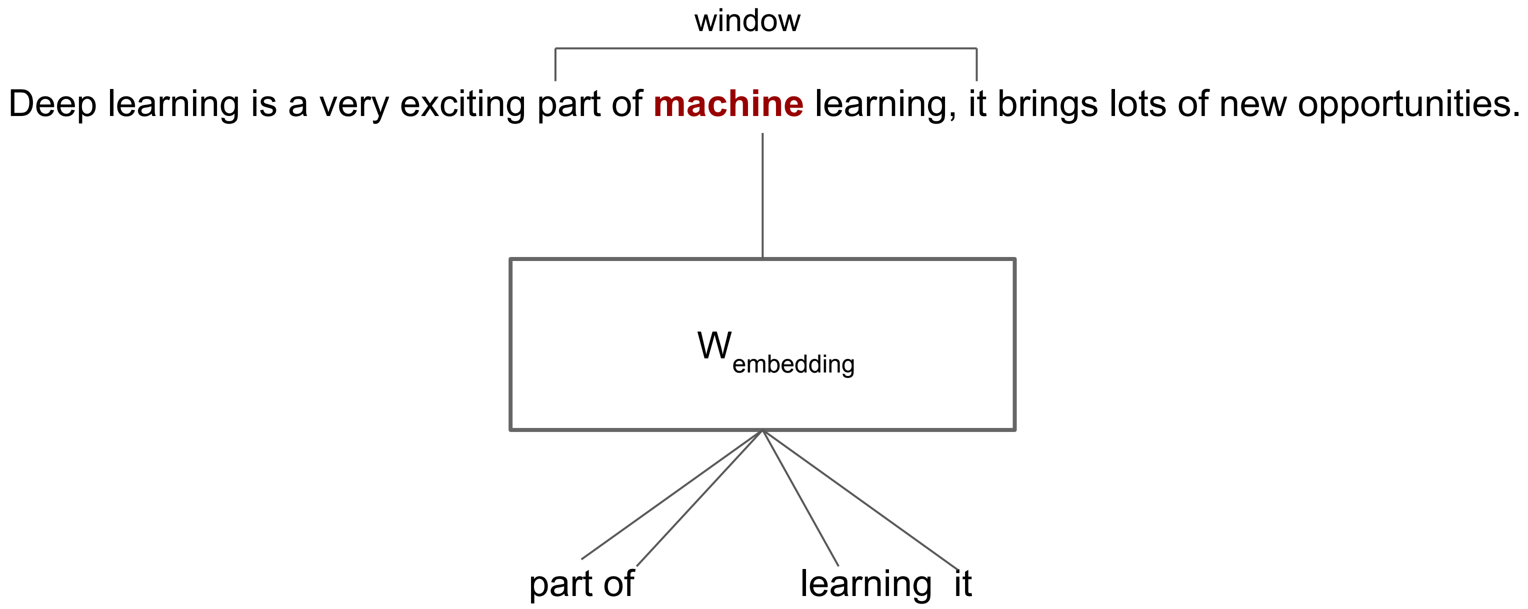
window

Deep learning is a very exciting part of **machine** learning, it brings lots of new opportunities.

$W_{\text{embedding}}$

machine

Skip-Gram



window

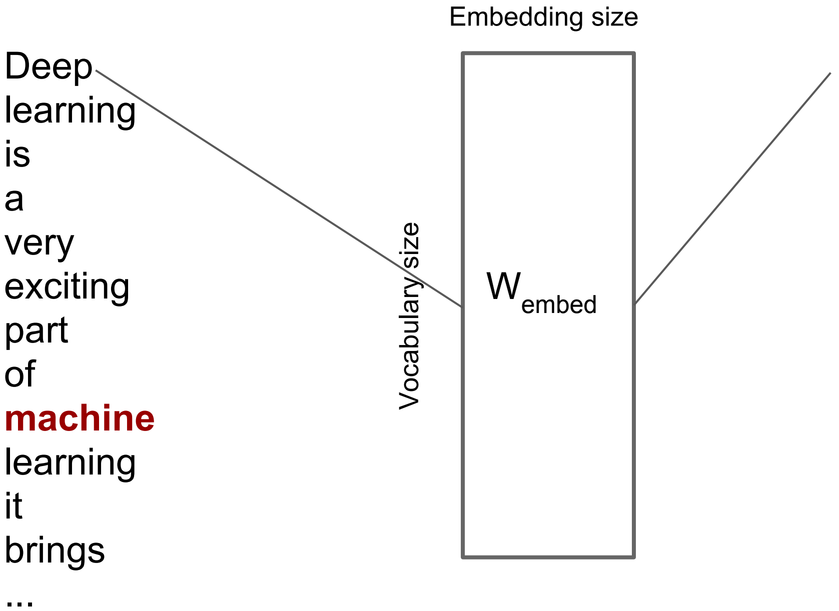
Deep learning is a very exciting part of **machine** learning, it brings lots of new opportunities.

$W_{\text{embedding}}$

part of

learning it

Retrieving the Embeddings



Deep
learning
is
a
very
exciting
part
of
machine
learning
it
brings
...

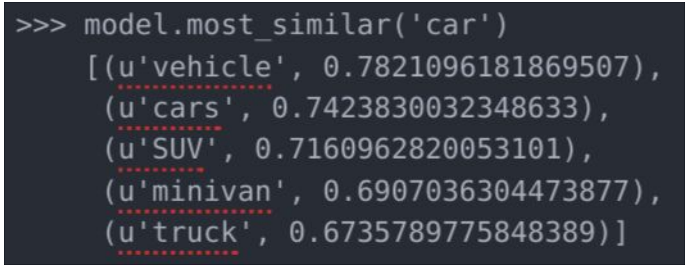
Deep = [0.7 0.2 0.8 0.4 ...]

Once the model is trained, the $W_{embedding}$ matrix turns into a lookup table.

To retrieve the embeddings we just take each row vector that corresponds to each word in the embedding matrix.

The shape of the embedding matrix is [n_vocab, embed_size].

Finding Similar Words



```
>>> model.most_similar('car')
    [(u'vehicle', 0.7821096181869507),
     (u'cars', 0.7423830032348633),
     (u'SUV', 0.7160962820053101),
     (u'minivan', 0.6907036304473877),
     (u'truck', 0.6735789775848389)]
```

$$\frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

By using the cosine similarity between the vectors representing each word we can query the model for the most similar words.

If we search for the words that are most similar to car we get; vehicle, cars, SUV, minivan and truck.

This type of thing is impossible if we represent words only as discrete objects.

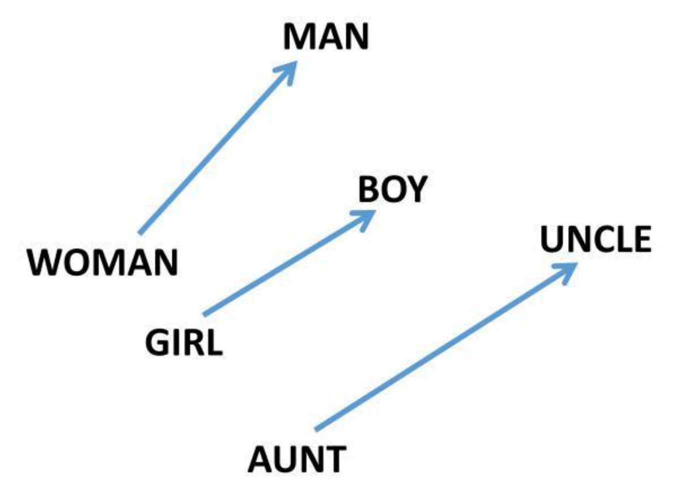
Useful Properties

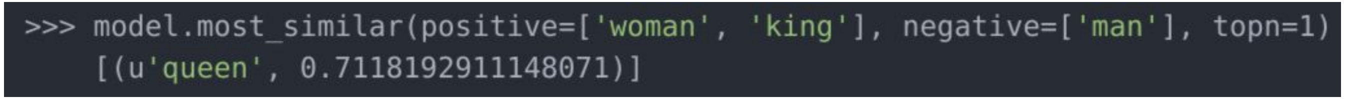
These embeddings end up encoding some really useful information.

The difference between ‘woman’ - ‘man’, ‘girl’ - ‘boy’, and ‘aunt’ - ‘uncle’ are all vectors with roughly the same direction.

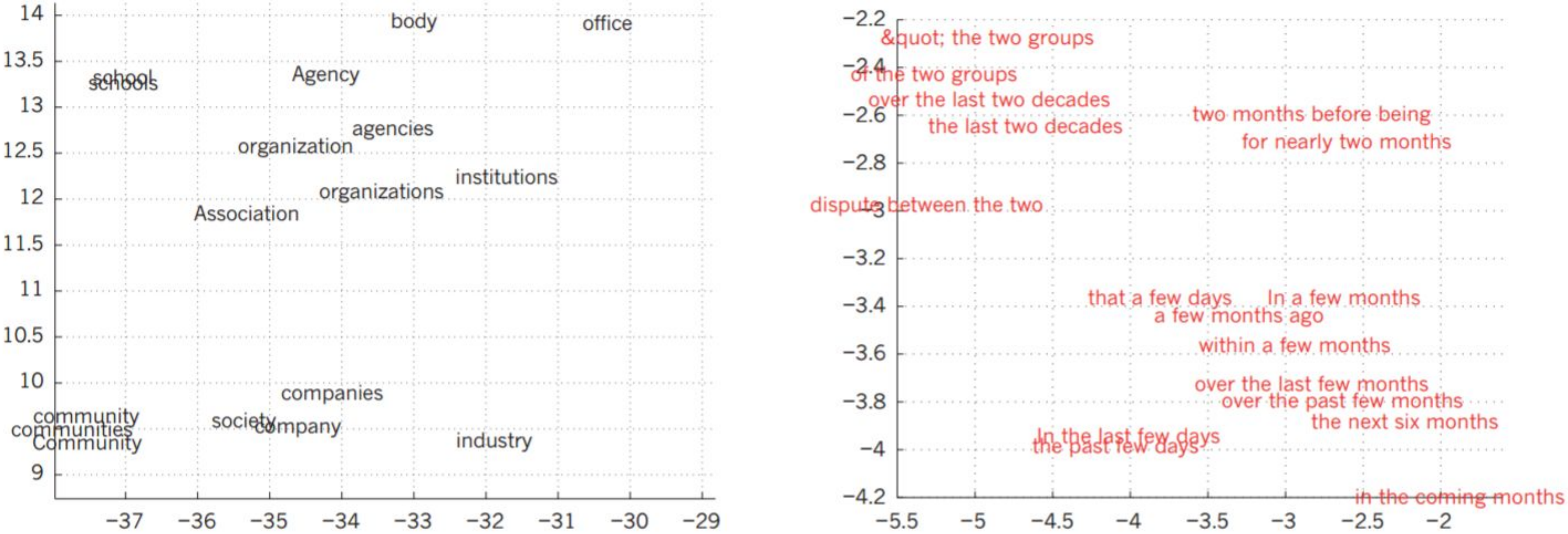
This tells us that the gender information is implicitly encoded in the word embeddings.

The same is true for many other comparisons that we can make between words.





```
>>> model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
    [(u'queen', 0.7118192911148071)]
```



http://www.cs.toronto.edu/~hinton/absps/NatureDeepReview.pdf

Hands on Practice

Recurrent Neural Networks

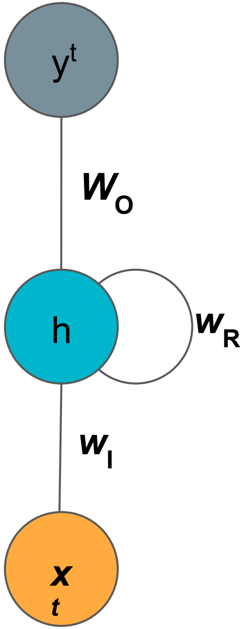
Recurrent Neural Networks

Useful when trying to apply deep learning to sequential types of data. Text, speech, and audio.

They also let us get around some problems where the input has to be a fixed size. For example with the images that we just worked with they always had to be scaled to the same exact size. But, if we wanted to train a deep learning model on sentences we can’t guarantee that they are always the same size.

Not only do they let us vary the length of the input, but they are also sensitive to temporal dependencies. Back to the sentence example, different words in different positions could be dependent each other.

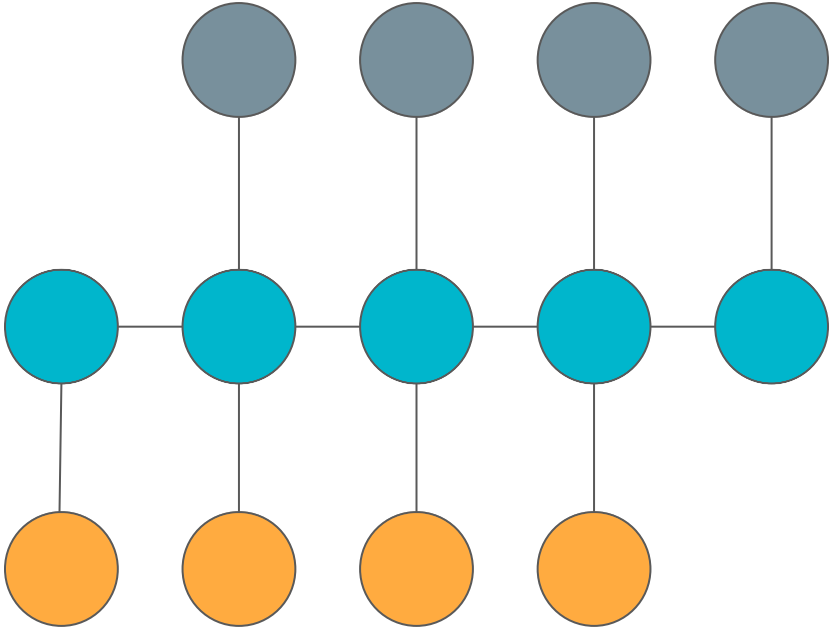
Recurrent Neural Networks



$$h^t = a_h(W_I x^t + W_R h^{t-1} + b_h)$$

$$y^t = a_y(W_o h^t + b_y)$$

Recurrent Neural Networks

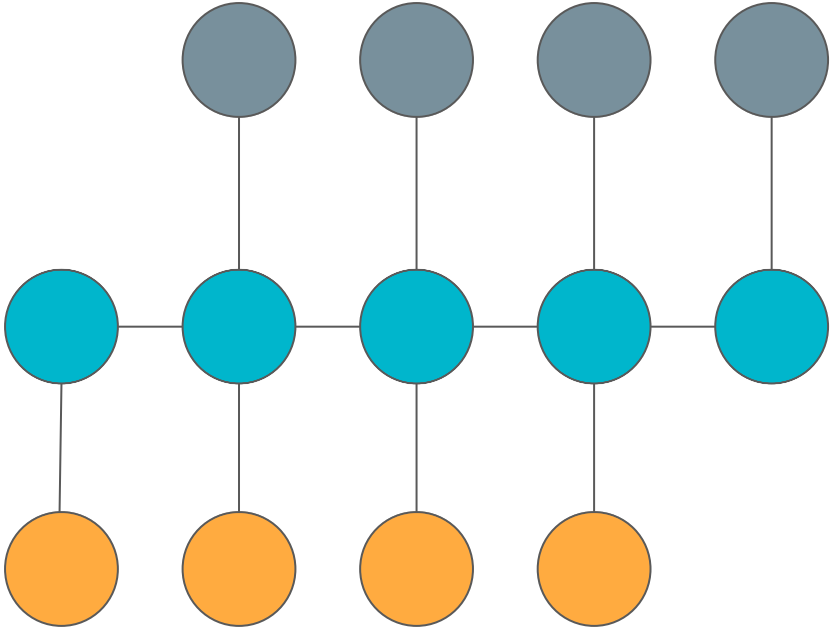


Recurrent Neural Networks

Unrolling the network like this is also helpful to gain some intuition about how they are trained.

For each output, we backpropagate the error all the way back to the first input.

It works exactly like the feed forward networks that we looked at before with the added headache of keeping track of all the connections.

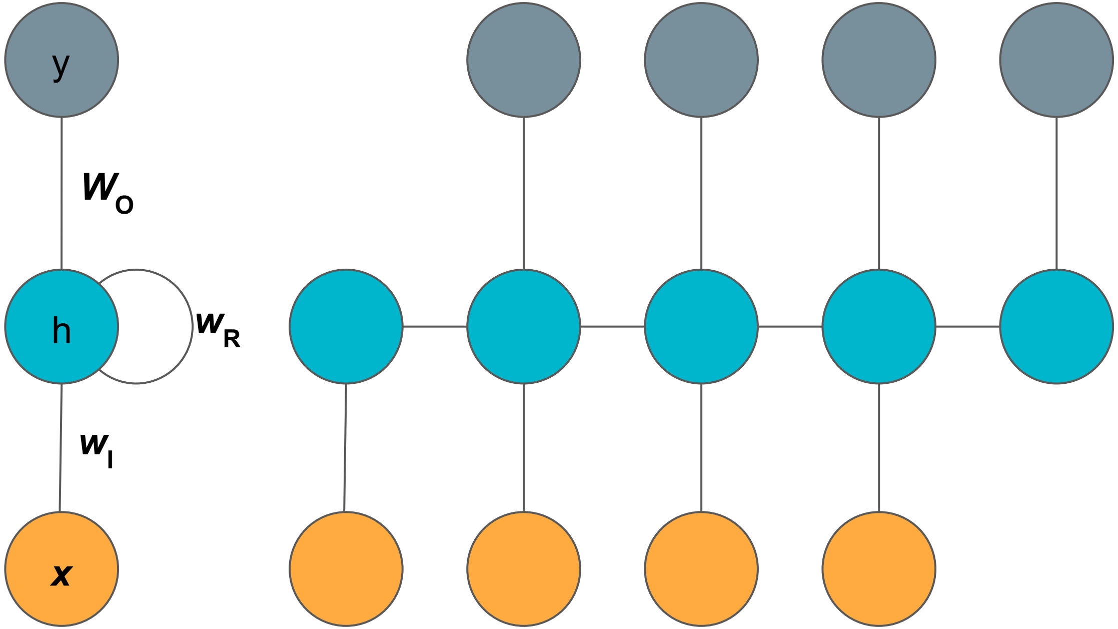


Vanishing and Exploding Gradients

One caveat here is that the weights that exist between all of the connects are shared in each time step.

So W_I, W_R, and W_O are used multiple times. So if you calculate the error at many time steps the magnitude of the update is scaled up.

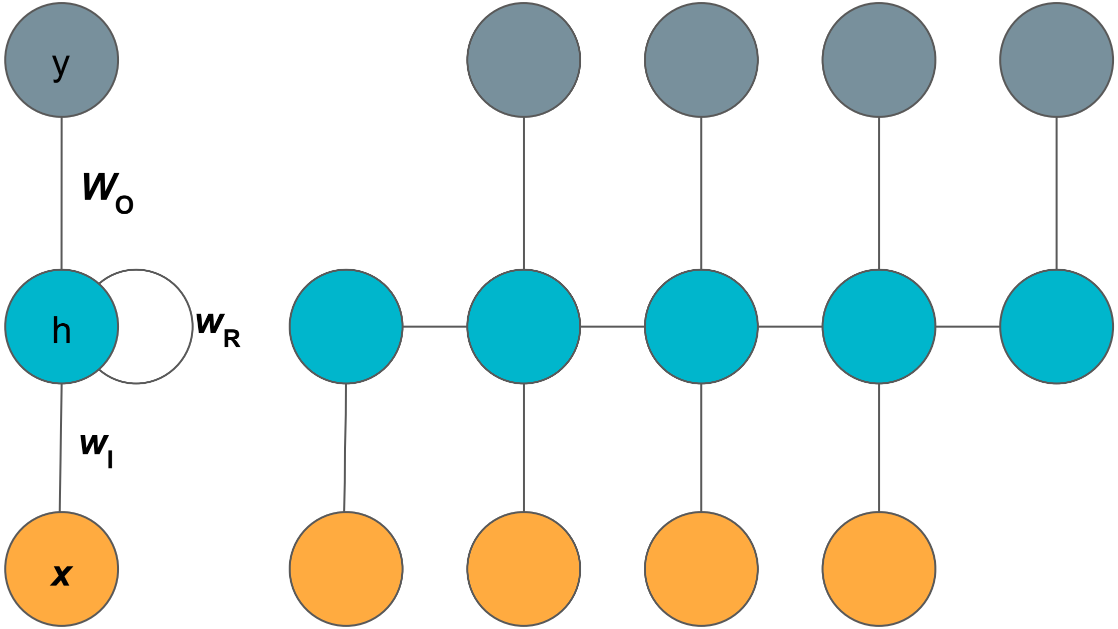
We could find ourselves in situations where the gradient signal either becomes 0 or incredibly large.



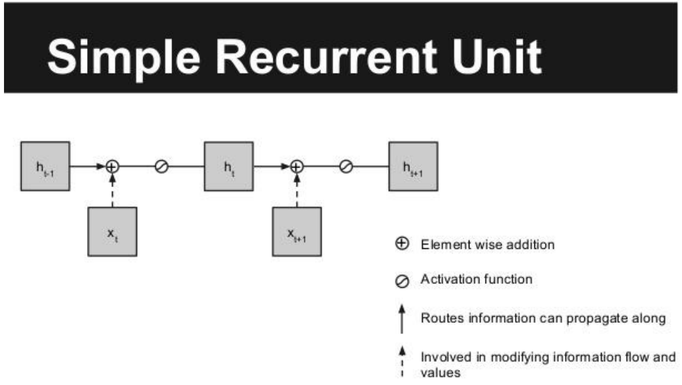
Vanishing and Exploding Gradients

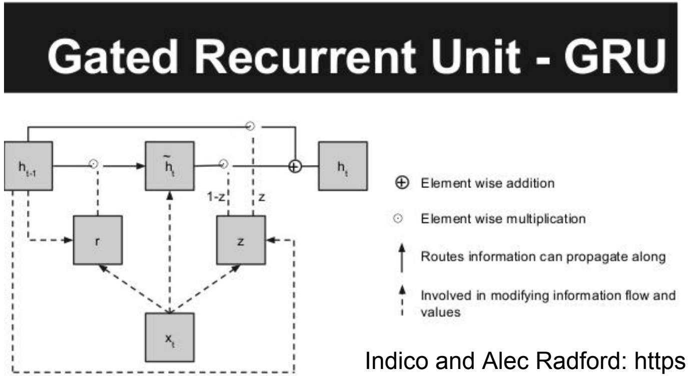
The most effective way that people combat this is to clip the gradient so that they can only take on some maximum value.

This seems kind of simple and hacky but in reality it works out pretty well.



Gated Recurrent Units



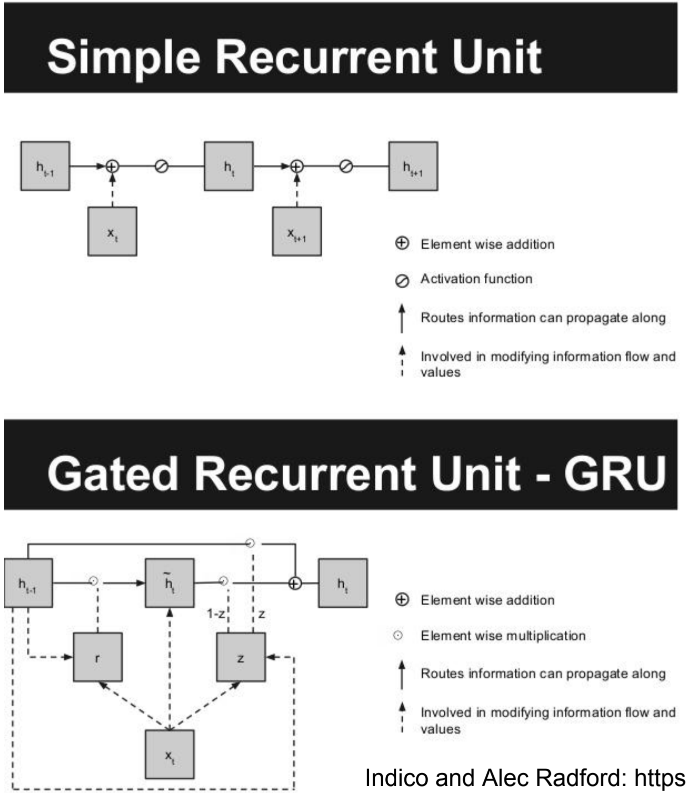

Gated Recurrent Unit - GRU
h_{t-1}
$\tilde{h}_t$
h_t
1-z
z
r
z
x_t
⊕ Element wise addition
⊙ Element wise multiplication
Routes information can propagate along
Involved in modifying information flow and values

These recurrent neural networks that we just looked at turn out not to function as well in practice. They have a significant draw-back in that the recurrent weight matrix is continuously updated. This make it difficult for information to persist through many time steps.

If you wanted to train a recurrent network on a very large body of text the network will have trouble relating words at the end of a document to words at the beginning.

Indico and Alec Radford: https://www.youtube.com/watch?v=VINCQghQRuM

Gated Recurrent Units


Simple Recurrent Unit
h_{t-1}
h_t
h_{t+1}
x_t
x_{t+1}
⊕ Element wise addition
⊘ Activation function
Routes information can propagate along
Involved in modifying information flow and values
Gated Recurrent Unit - GRU
h_{t-1}
$\tilde{h}_t$
h_t
r
z
1-z
z
x_t
⊕ Element wise addition
⊙ Element wise multiplication
Routes information can propagate along
Involved in modifying information flow and values

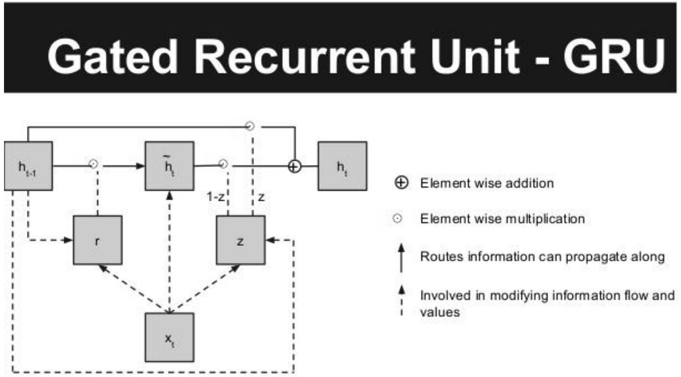
Indico and Alec Radford: https://www.youtube.com/watch?v=VINCQghQRuM

The GRU is like a simple RNN except that it has two matrices that function like gates. We typically call these gates *r* and *z.*

The reset gate, *r*, determines how much to combine the previous hidden state with the new input at time *t*. The update gate, *z*, determines how much the hidden state of the network should update with each example.

This is powerful because the network will have the ability to keep information within its hidden state for much longer.

Gated Recurrent Units


Gated Recurrent Unit - GRU
h_{t-1}
$\tilde{h}_t$
h_t
1-z
z
r
z
x_t
⊕ Element wise addition
⊙ Element wise multiplication
Routes information can propagate along
Involved in modifying information flow and values

$$z_t = \sigma(W_z h_{t-1})$$

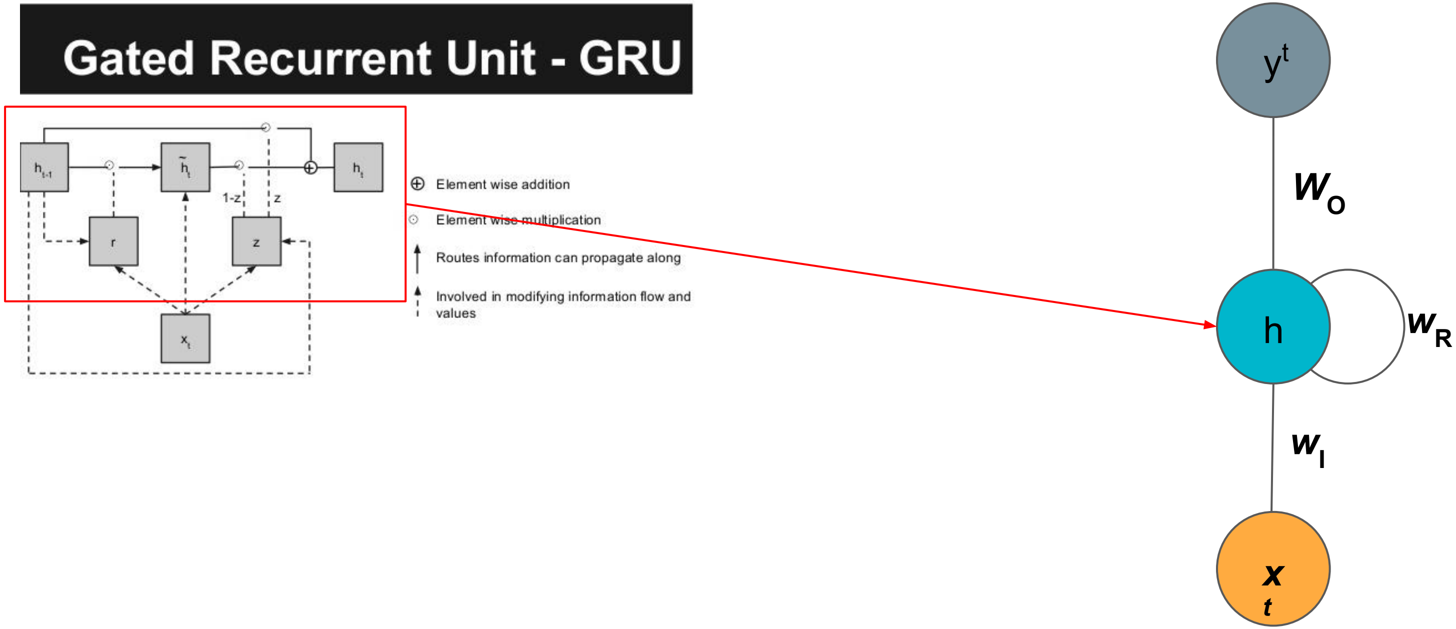
$$r_t = \sigma(W_r h_{t-1})$$

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

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

Indico and Alec Radford: https://www.youtube.com/watch?v=VINCQghQRuM

Gated Recurrent Units



Gated Recurrent Unit - GRU

⊕ Element wise addition

⊙ Element wise multiplication

↑ Routes information can propagate along

⇡ Involved in modifying information flow and values

Indico and Alec Radford: https://www.youtube.com/watch?v=VINCQghQRuM

What about LSTM?

GRU cells are a recent tweak on LSTM cells.

Their function is similar, allowing an RNN to have longer term dependencies.

However LSTM cells are a bit more complicated and in practice the two perform about the same.

“We compared the GRU to the LSTM and its variants, and found that the GRU outperformed the LSTM on nearly all tasks except language modelling with the naive initialization, but also that the LSTM nearly matched the GRU’s performance once its forget gate bias was initialized to 1.”

“An Empirical Exploration of Recurrent Network Architectures” - http://jmlr.org/proceedings/papers/v37/jozefowicz15.pdf

Rafal Jozefowicz, Wojciech Zaremba, Ilya Sutskever

Resources for LSTM?

Andrej Karpathy: Unreasonable Effectiveness of RNNs

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

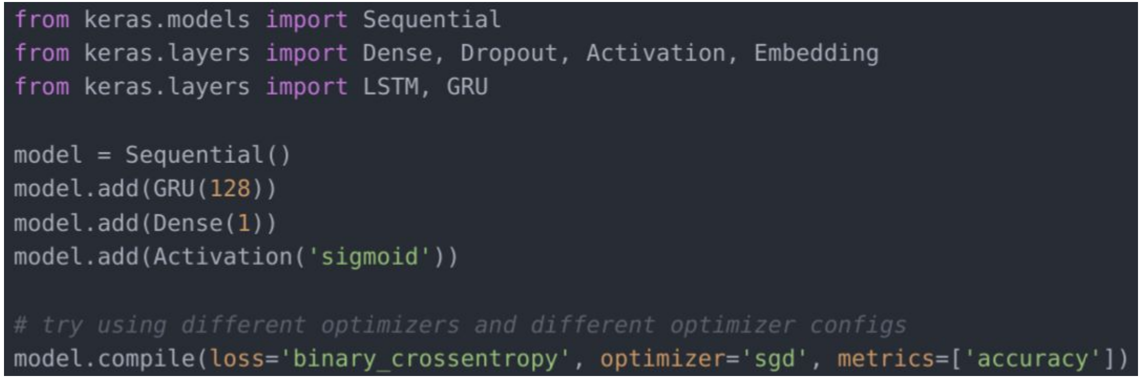
Christopher Olah: Understanding LSTM Networks

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

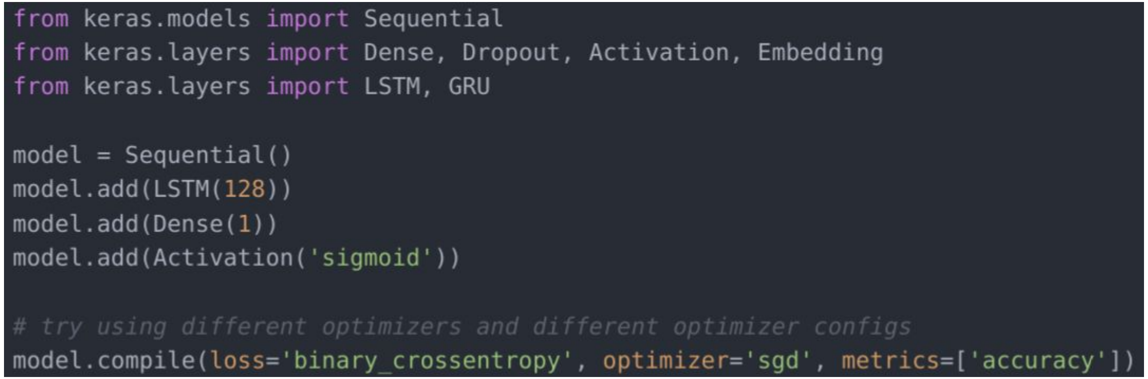
A Beginners Guide to Recurrent Neural Networks and LSTM

https://deeplearning4j.org/lstm

Coding GRU and LSTM



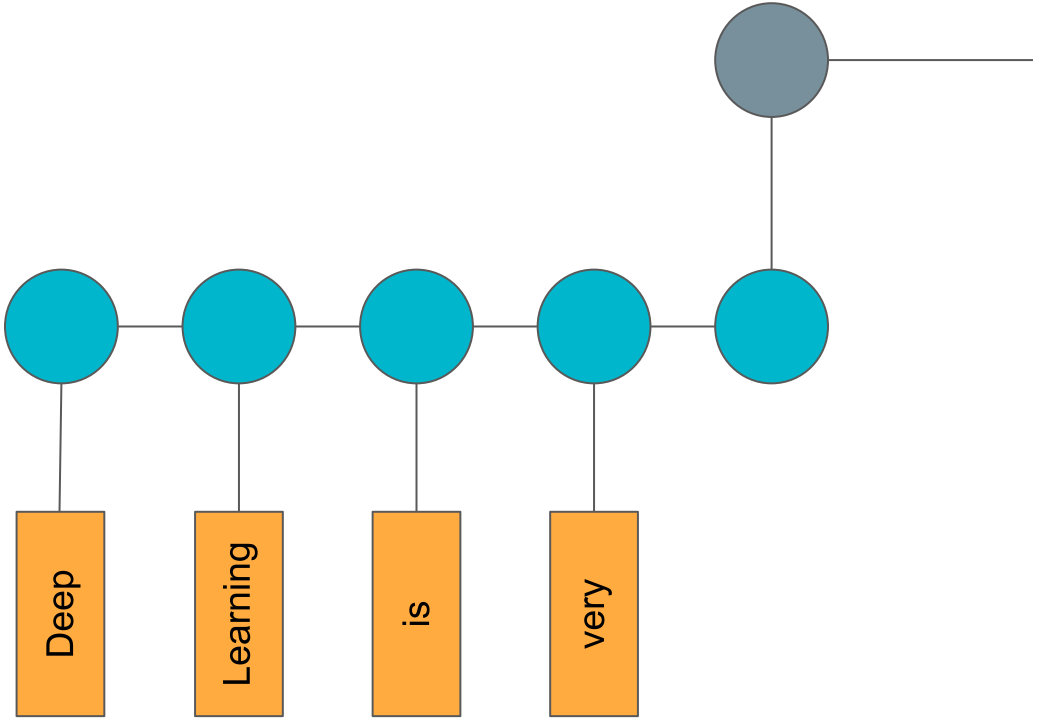
```python
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Embedding
from keras.layers import LSTM, GRU

model = Sequential()
model.add(GRU(128))
model.add(Dense(1))
model.add(Activation('sigmoid'))

# try using different optimizers and different optimizer configs
model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['accuracy'])
```



```python
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Embedding
from keras.layers import LSTM, GRU

model = Sequential()
model.add(LSTM(128))
model.add(Dense(1))
model.add(Activation('sigmoid'))

# try using different optimizers and different optimizer configs
model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['accuracy'])
```

Hands on Practice

A Match Made in Heaven



Some kind of positive sentiment!

In Summary

Deep learning is a popular machine learning method for tasks that attempt to mimic human perception.

1) Images
2) Text
3) Audio
4) Speech

In Summary

We typically define neural networks by their computational graphs and most computations are done as matrix multiplies.

The ‘dogma’ of deep learning is that we are using these graphs to learn ways of representing the input data in a form that makes it easiest for the given task.

The final classifier layer is simple, the real work is done above it.

We can also extract and use these internal representations for other tasks.

See our image classification example, and the word embeddings slides.

In Summary

Neural networks are usually trained with Stochastic Gradient Descent and it's variations.

A typical workflow for images would be taking a network pretrained on ImageNet and fine tuning it for your needs. Like what we did in our examples.

RNNs are still a little bit more tricky to work with. Usually takes some time to nail down the best hyperparameters.

Word embeddings are an incredibly powerful way of representing words by extracting the internal representations of the neural network.

Useful Links

Deep Learning Book - amazing resource for everything from theory to application

http://www.deeplearningbook.org/

Tensorflow - Tutorials and code examples if you want something more low level than Keras.

https://www.tensorflow.org/

Keras - High level wrapper for tensorflow or theano. Most useful for those looking to do applied work instead of research.

https://keras.io/

Useful Links

Theano - primarily developed as a research tool for deep learning. Functions a lot like TensorFlow. Some people call TensorFlow theano 2.0.

http://deeplearning.net/software/theano/

Torch - another popular framework for deep learning. Uses Lua instead of python.

http://torch.ch/

Stanfords DL tutorial - quick low level introduction to deep learning.

http://deeplearning.stanford.edu/tutorial/

Useful Links

Hugo Larochelle's neural network class - Great video series of neural networks

https://www.youtube.com/playlist?list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH

Deep Learning for Natural Language Processing

http://cs224d.stanford.edu/

Convolutional Neural Networks for Visual Recognition

http://cs231n.stanford.edu/