TDT265 Notes

Jon Eirik Andresen

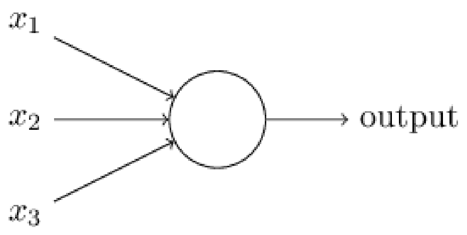
May 2018

Contents

1 Shallow Nets

1.1 Perceptrons and Network Basics

A perception is a simple neuron, that can make yes/no decisions based in weighted, inputs, that takes in a vector of binary inputs **x** and outputs a single binary output y. The neuron has weights w and a threshold. The mathematical model and figure is given below.



$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

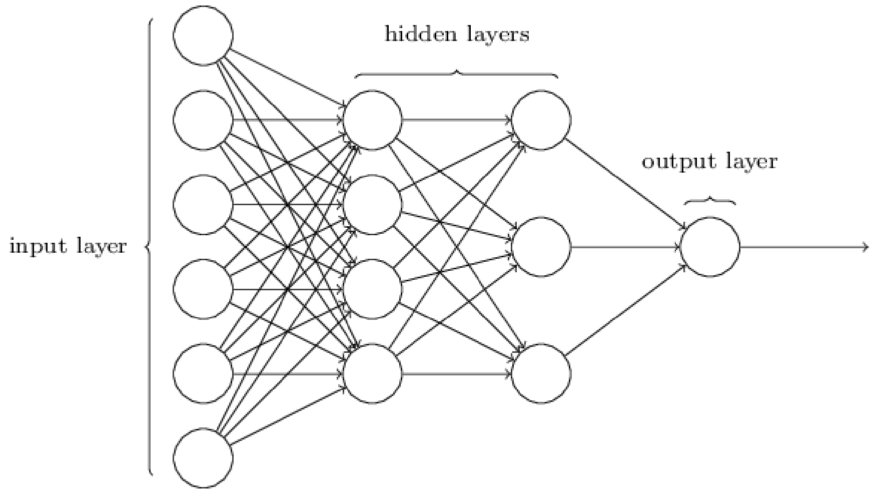
Of course, these can be combined into a network to make even more complex decisions.

If you're a little clever about it, you use a bias term instead of a threshold, and write things on vector form. Then a perception becomes:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

If you want to be really clever, you can use a perception to implement any logical function.

When drawing networks, it is normal to add a **input layer** as a layer that simply outputs the input, and has the input written on it. The layers between the input and output is called **hidden layers** and the **output layer** is called just that.



1.2 Sigmoid Neurons

You can't really train a network of perceptrons, as flipping a single neuron will cause the network to behave very differently. Sigmoid neurons has a smooth **activation function**, the function that determines the output of the neuron.

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$

z is the dot product of $\mathbf{x}$ and $\mathbf{w}$. Note that the 0'th element of $\mathbf{x}$ is 1 and of $\mathbf{w}$ is b so as to simplify the calculations.

1.3 Tanh

Another neuron is a tanh neuron. Looking very sinilar to a sigmoid, except it will gi values from -1 to 1.

1.4 ReLu

Stands for **rectified linear unit**. Can be expressed as $\max(0, w \cdot x + b)$. These are often better at image based applications, but no-one knows why.

2 Learning

2.1 Learning and Gradient Descent

To learn, one must define a **cost function** to determine how well the output of the network approximates the ground truth. One such function is mean square error:

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

a is the output and $y(\mathbf{x})$ is the ground truth for input $\mathbf{x}$. Note that this function has only one minimum, i.e. where $y(\mathbf{x}) = a$.

By taking the gradient of C and using that to update w one will move in the direction of the minimum.

With:

$$\nabla C = \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_n}\right)^{\mathrm{T}}$$

$$w_{new} = w_{old} - \eta \nabla_w C$$

and

$$b_{new} = b_{old} - \eta \nabla_b C$$

η is called the **learning rate**. This update is done until convergence is reached.

Training on the whole dataset x at a time is to expensive, so choosing a random sample, a **batch**, of the data and training on that instead will approximate the gradient of the whole set.

2.2 Back Propagation

Back propagation is an effective algorithm to update weights a biases for each neuron in a net. At its centre is the calculation of the derivatives $\frac{\partial C}{\partial w}$ for each neuron/layer.

2.2.1 Some Notation

w^l_{jk} means the connection from the k^{th} neuron in the $(l-1)$ layer to the j^{th} neuron in the l^{th} layer.

b^l_j is the bias of the j^{th} neuron in the l^{th} layer.

a^l_j is the activation of the j^{th} neuron in the l^{th} layer.

Using an activation function σ the output/activation becomes:

$$a^l_j = \sum_k^K w^l_{jk} a^{l-1}_k + b^l_j$$

Summing over all neurons$K$ in the previous layer.

To be clever about it, one will write the weights as a matrix w^l where the elements w^l_{jk} are in places jk in the corresponding matrix. We also create a bias vector b^l and an activation vector a^l. Now we also write the activation function as a vector, where each element of the vector is the function applied to that element in the input vector. Then stuff boils down to:

$$a^l = \sigma(w^l a^{l-1} + b^l := \sigma(z^l)$$

z^l is the **weighted input** and will be used later. Maybe...

$\mathbf{A} \circ \mathbf{B}$ is the element-wise multiplication of the two matrices of vectors.

2.2.2 Equations of Backpropagation

This is the result:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

To adjust the weights and biases, we need the derivative of the cost relative to those values. Insead of computing it all at once, we introduce δ^l_j which is the error in the j^{th} neuron of the l^{th} layer. Backprop gives us *delta* and then ∇C.

We now say $\delta^l \equiv \frac{\partial C}{\partial z^l}$ for layer l.

For the output layer, and on vector form, the error

$$\delta^L = \nabla_a C \circ \sigma^{'}(z^L) = (a^L - y) \circ \sigma^{'}(z^L)$$

.

To calculate the error δ^l of any layer in the network, we use recursion, basing it on the error in the layer before it δ^{l+1}.

$$\delta^l = ((w^{l+1})^{\intercal} \delta^{l+1}) \circ \sigma^{'}(z^l)$$

An so, starting from L, one can calculate all the δs.

As it turns out: $\frac{\partial C}{\partial b^l_j} = \delta^l_j$. Which is nice, since we already have δ from the above calculation.

And lastly, the rate of change related to the weights in the net:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_k$$

2.2.3 ZE ALGORITHM

1. **Input a set of training examples**

2. **For each training example x:** Set the corresponding input activation $a^{x,1}$, and perform the following steps:

   - **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$.

   - **Output error $\delta^{x,L}$:** Compute the vector $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$.

   - **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$.

3. **Gradient descent:** For each $l = L, L-1, \ldots, 2$ update the weights according to the rule $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$, and the biases according to the rule $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$.

2.3 Improvements to Learning

2.3.1 Cost Entropy Cost Function

The quadratic cost function, with a sigmoid neuron has very slow learning rates "to the left" and "to the right" of the sigmoid function. To adress this, we indtroduce the **cross entropy cost function**

$$C_{CE} = -1/n \sum_x [y \ln(a) + (1-y) \ln(1-a)]$$

Note that C_{CE} is non negative, which is nice. When the error is low, the cost is low. A miracle of the cross entropy is that it is independent of $\sigma^{'}$ ensuring that there is no slowdown.

For a whole network, with L output neurons the function looks like this:

$$C = -\frac{1}{n}\sum_x \sum_j \left[y_j \ln a_j^L + (1 - y_j)\ln(1 - a_j^L) \right].$$

Whenever the output neurons are sigmoid neurons, one should always prefer cross entropy over quadratic cost.

2.3.2 Softmax

Another way of addressing learning slow-down is to add another output layer to the network, a **softmax** layer. Instead of outputting the sigmoid function of the weighted input, one does this to output nr. j, and summing over all neurons K:

$$a_j^L = \frac{e^{z_j^L}}{\sum_k^K e^{z_k^L}}$$

Doing this causes:

- All the outputs to sum to 1
- The neuron with the highest input will have a **way higher** output than the others.
- With the above, we can pretend the output is a probability distribution, where the network says that it is so an so sure that the correct output is j is neuron j has the highest output.

2.4 Over fitting and Regularization

Over fitting a model is to have the parameters not learn the underlying features, but rather only the data we provide it. To avoid this, and to also smartly choose other hyper-parameters we use a **validation set** , a subset of the data provided to training. This ensures that the net is not over-fitted to either the training data or the test data.

Regularization is another trick to avoid over-fitting. We will discuss several types; L1, L2 dropout and expanding the training data.

L2 is all about adding a term to the cost function such that it becomes:

$$C = C_{original} + \frac{\lambda}{2n}\sum_w w^2$$

Where λ is the **regularization parameter**. Adding this term penalizes large weights, forcing the learning to prefer having smaller weights. This helps the network to not learn much from random noise in input data

L1 is similar to L2.

$$C = C_{original} + \frac{\lambda}{n}\sum_w |w|$$

Dropout is about modifying the net, not the cost function. When training, at each forward or back pass a random set of neurons are "commented out" of the net, so the net trains with out them and updates the still active neurons. This is similar to training a lot of different nets and then combining them.

2.4.1 Algorithmically Create More Data

To reduce over-fitting without having more data to train on, one can expand the training data by changing it somehow. This can be a simple moving of the picture by a set amount of pixels or a warping or rotation of the image.

2.4.2 Having Many Networks

When initializing networks with random weights and biases the trained nets will be ever so slightly different. Thus, having several nets running the same classification and then voting on which result is the correct one will, at increased computational cost, increase the accuracy and reduce the effects of over-fitting.

2.5 How to choose hyper parameters

Learning rate find one that doesn't make things oscillate or increase and that will be your absolute threshold. A good learning rate will be smaller than that, e.g half that. An other approach is to decrease the learning rate as time goes on to ensure that one reaches

Epochs. When the validation accuracy stops improving, that is a good limit for the amount of epochs to limit over-fitting.

2.6 Vanishing Gradient Problem

When training deep neural networks, the gradient will drop the further back in the net you go. This is bad for training, as it slows it way down. This happens because when the derivative of the activation function comes in as a term at each step in the back propagation and has a value < 1. There is a possibility of **exploding gradient** which is when the gradients increase. Then the further back into the net you go, the higher the gradients will be.

3 Convolutional Neural Networks (CNNs)

CNNs are deeper networks than discussed earlier, and have a different architecture than a fully connected one. They also use some other tricks.

3.1 Some Basic Ideas in CNNs

3.1.1 Local Receptive Field

As with normal networks, there is still an input layer. Here, however, we will keep it "looking like an image" e.g. $m \times m$. Instead of having every neuron in the input layer be connected to every neuron in the first hidden layer (fully connected), every neuron in the first hidden layer only takes input from a set amount of neurons in the input layer, say $n \times n$. This filed of neurons the next layer looks at is called its **local receptive field**. The hidden neurons inputs will all have their own weights, and the neuron will have a single bias.

The next neuron in the first hidden layer looks at another field which is shifted to either the side or down, depending on which next it is, by what is called **stride length**. For a stride length of 5, there will be no overlap between receptive fields.

3.1.2 Shared Weights and Biases

Another difference is that each neuron in the first layer has the same weight and bias as all the others. By doing this, each neuron is activated by the same feature in the input image. The output of neuron j, with a stride length of n and activation function σ is:

$$\sigma\left(b + \sum_{l=0}^{n-1}\sum_{m=0}^{n-1} w_{l,m}a_{j+l,k+m}\right)$$

The mapping from the input layer to the first hidden layer a **feature map**, and the shared weights and bias a **kernel** or **filter**.

To complete a convolutional layer, more than one such feature maps are put in the same place, and thus looks for different features in the same spot in the network.

A benefit of CNNs is that there are WAY less trainable variables, while still getting pretty good performance, compared to a normal fully connected one.

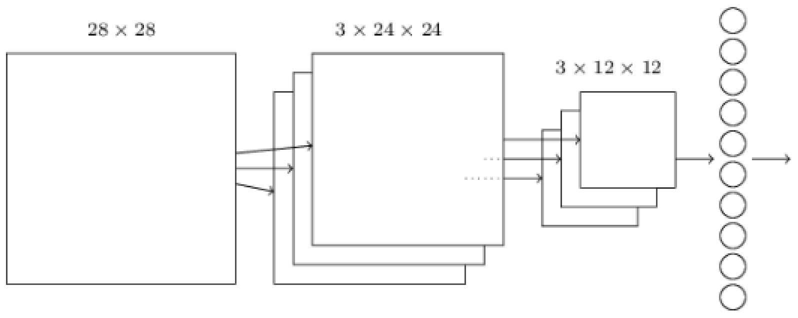
Having to look at the whole image and sharing weights across all neurons means that the layer is way less sensitive to noise and thus dropout is not as useful in convolutional layers.

3.1.3 Pooling Layers

A pooling layer takes a section of the previous layers output and condenses it in some way. This is very often done after a convolutional layer. A normal pooling layer is **max pooling** where only the maximum value of the section the pooling layer looks at is sent out.

The consequence of a pooling layer is a reduction in data sent along in the net. It also tries to ensure that only the feature ”most present” in a set of receptive fields is sent along. We also disregard the exact position of the feature and only care about it approximate position relative to other features.

A net with a 28×28 input-layer, a $3 \times 24 \times 24$ convolutional layer and a $3 \times 12 \times 12$ layer followed by an output layer will look like this:



Note that convolutional layers should use a ReLu activation function.

3.2 Visualizing

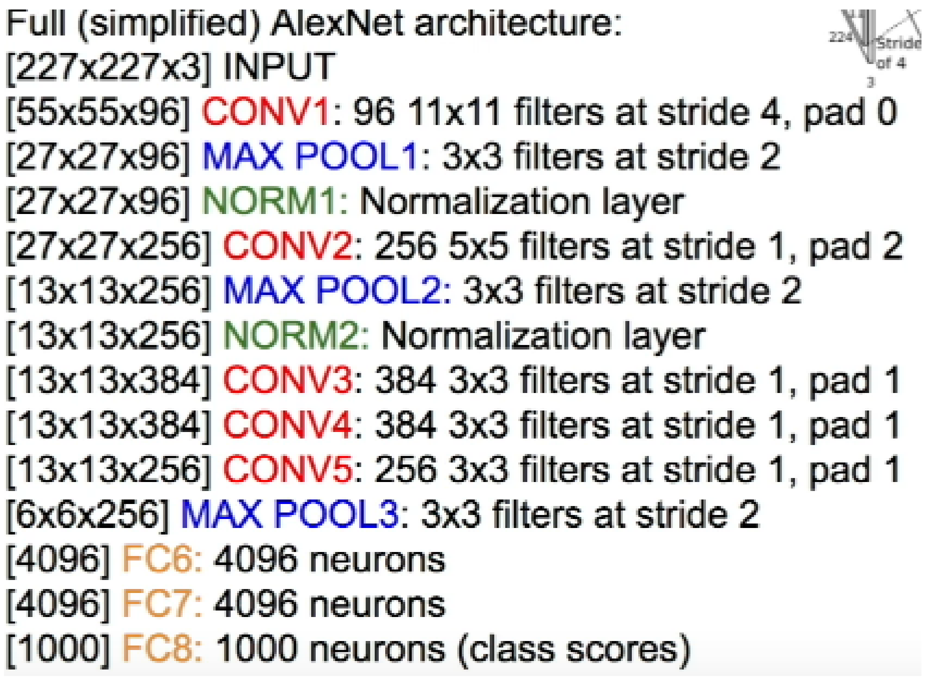
To look at what a filter or layer is learning to look at, we look at the weights of the filter. In the beginning of a CNN we can see features like contrasting colours or edges. However, the deeper into the network you go, the harder it is to simply look at the filter weights since the layer no longer looks at an image, but a features extracted from an image.

For intermediate layers, instead of looking at weights, one can look at activations. What we look at is what parts of the image causes the layer to activate. This, by then comparing the picture of the activation level to the input image we can deduce what that layer is looking for, i.e a face or headlights on a car. You can also study which patch causes the maximal activation of a layer/neuron and then take a look at that patch.

Another way is **occlusion experiments**. This tells you what part of an image is important for the classification. You block out a block of the image, i.e replacing it with a block of pixels with the average colour, and then try to classify the occluded image. You do this for every possible position of the block. If the score of the image changes drastically, then that part of the image likely was important for the classification. Then you create a heat-map of probabilities.

3.3 Some Architectures

3.3.1 AlexNet


224
Stride of 4
3

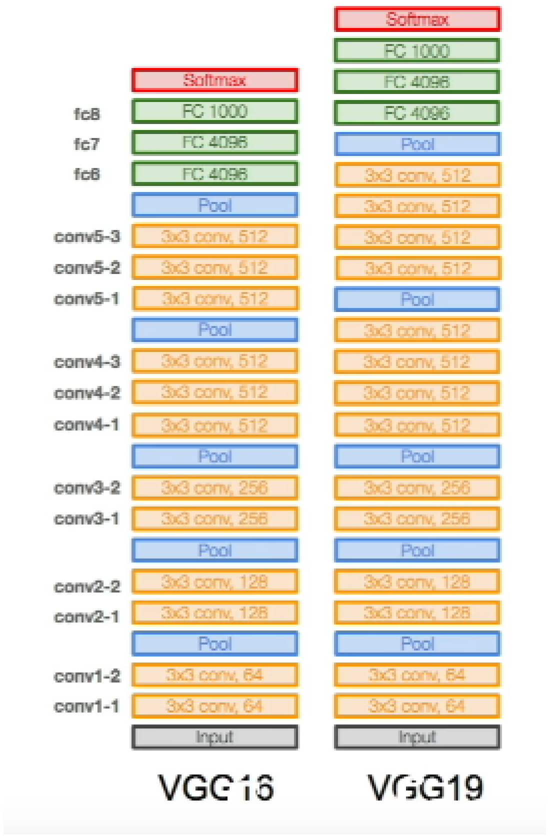
Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

First to use ReLU. To train, a lot of data augmentation was used. Used dropout to train.

3.3.2 VGG

Much deeper than AlexNet, but smaller filter. 3x3 conv filters at each layer. Being deeper, but shallower allows for the same effective receptive field, but with more non-linearities and fewer parameters.

Demands ish 100 MB of VRAM pr image for a forward pass.



3.3.3 GoogLeNet

Also deep. No fully connected layers. Uses what is called an **inception module**. Only 5 million parameters.

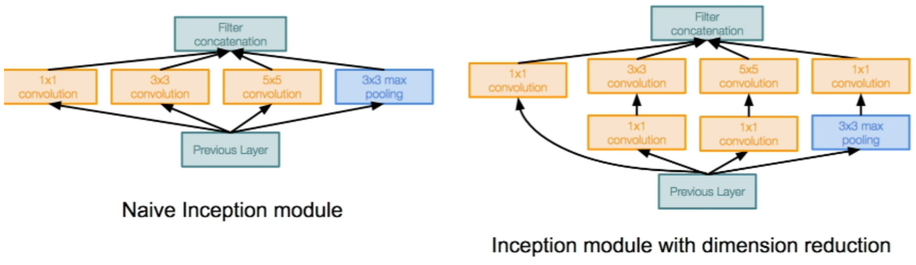
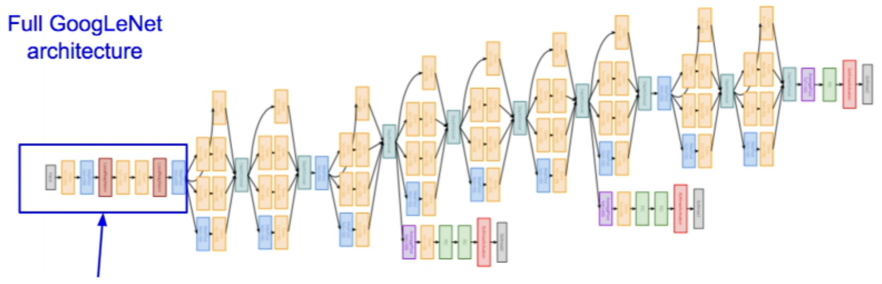


Figure 1: Naive Inception Module

The inception module is a part of the network, and several of these are stacked on top of each-other. The three rightmost 1x1 conv layers are called **bottlenecks** and reduce the depth of the feature maps. Reduces the amount of work done by the module by a factor of more than 2.

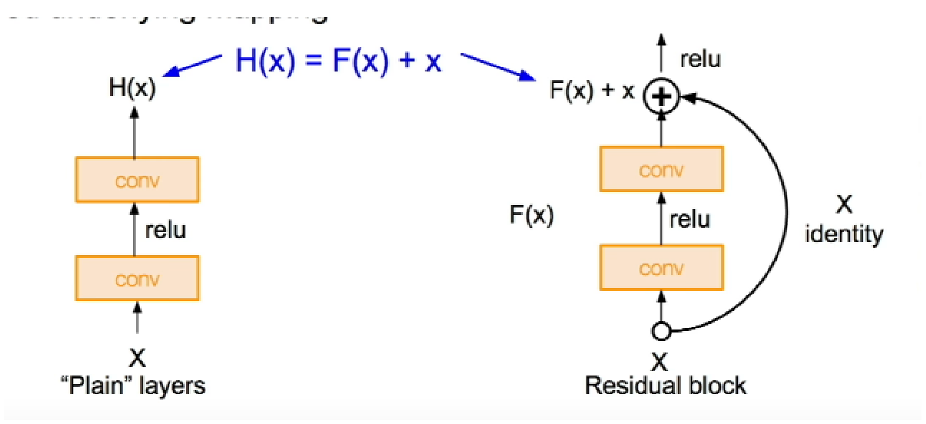
The whole network is a stack of these:



Starts with some normal conv layers and pooling. The two extra output "arms", called **auxiliary outputs**, are there to inject the gradient in more places to beat the vanishing gradient problem.

3.3.4 ResNet

Stupid deep. 152 layer. A deeper net is hard to train so use **residuals** of intermediate input further up in the net. Somewhere in the net, the intermediate input x is added further up in the net, and so, instead of learning what is to be done with $x = H(x)$ we learn $F(x) = H(x) - x$.



Each res thing has 2 3x3 conv layers between them. 1 fully connected in the end. Periodically double filter size and down-sample. For ResNet's that are deeper than 50 there is also the bottleneck trick from GoogleNet.

3.4 Object Detection

The task is to find objects, maybe many, in an image, say what it is and put a box around it.

Can do it in these, and maybe more ways:

- **Sliding window** Run a classifier over a lot of places in the image. Slow and super expensive and just plain crap.
- **Region Based Methods** Somehow find regions of interest in the image and run a classifier at those regions, then return the box with the best fit using a bounding box regressor.
- **Singe shot detection** Split image into regions and run classifications one these and use some set threshold for a positive classification.

Accuracy is measured by **intersection over union (IoU)** $\in [0-1]$ of proposed bounding boxes and the ground truth. Typically an IoU of $> .50$ is a hit. Then calculate the **mean Average Precision** which is

$$mAP = \frac{1}{|classes|} \sum_{c \in classes} = \frac{TP(c)}{TP(c) + FP(c)}$$

Where TP and FP is number of true and false positives at the chosen IoU.

3.4.1 Region Based Detection

R-CNN, Fast R-CNN and Faster R-CNN all use some way of creating region proposals and running classification on these. The two first use something called **selective search** to generate the region proposals, while Faster R-CNN uses a CNN. Fast and Faster R-CNN uses a CNN for the bounding box regressor, and R-CNN uses a SVM.

4 Orientation

4.1 RNN

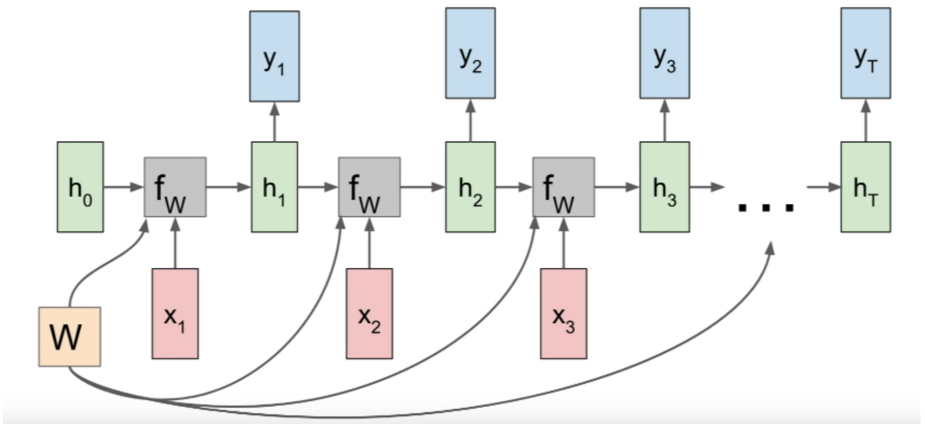
Recurrent Neural Networks. Instead of just feeding everything forward and having a fixed size, we can have variable input length or size. E.g reading a video or creating a photo caption. The RNN will have a hidden state that is, in combination with the current input, influences the current output. The hidden state will also update with new input. You can have mulitple levels of hidden states.

A simple configuration of hidden state h_t, x_t, W (weight matrix) and y_t might be:

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

Looking at the process of feeding the old state into the new one as not recursion, but a series of time-steps might look a little something like this:



When calculating the loss for training, it will flow from all the time-steps to the shared weight matrix. If you have a lot of steps, calculating the gradient for all steps is not feasible. What one does is called **truncated backprop**. Only back-propagate a set amount of steps.

Combining a CNN with a RNN will allow you to take in images and produce natural language captions for images. You take away the final softmax layers of the CNN and connect the 4096 neuron last layer into the prior state of the RNN. One can measure the **attention** of the the net to see where it looks to generate the language.

When training an RNN, the gradient will often either explode or vanish. For exploding, you can do hack, but for vanishing, there is little do to other than move to other architecture, e.g **LSTM** (long short term memory) that has two hidden states h_t, hidden state, and c_t, cell state.

Vanilla RNN

$$\boxed{h_t = \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)}$$

LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Figure 2: Caption

4.2 Generative Models

5 Traditional Computer Vision

5.1 Interest Point Detection

Instead of learning features, we **engineer** them. A common point is to find good and key features in an image, e.g. an edge or a corner. An edge is somewhere the gradient of some value is very large. A corner is somewhere two, or more, such edges meet.

Keypoint matching is what we do when we have some points in an image and want to check if they are the same. You pick some point, extract and normalize some data around it, and compute the **local descriptor** i.e a function that measures "equalness". Have to make trade offs between **repeat-ability** and **no. of points**, and **descriptive patches** and **flexible patches**.

5.1.1 Harris Corner Detector

Corners are amazing, as their location can be specified exactly.

Shifting a window around a corner should give you a large change in intensity. The change in intensity E for a shift u, v, intensity I and window function w:

$$E(u,v) = \sum_{xy} w(x,y)\left[I(x+u, y+v) - I(x,y)\right]^2$$

Taylor expanding the first I E becomes:

$$E(u,v) = \sum_{xy} w(x,y)\left[I(x,y) + uI_x + vI_y - I(x,y)\right]^2$$

$$E(u,v) = \sum_{xy} w(x,y)\left[uI_x + vI_y\right]^2$$

Some matrix tricks later involving expanding and multiplying: $E(u,v) = (uv)M(uv)^{\mathrm{T}}$

Where:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_y I_x & I_y^2 \end{bmatrix}$$

The eigenvalues of M will determine which direction the gradient is largest. Where both eigenvalues λ_1 and λ_2 is a corner.

The measure of **cornerness** in terms of the eigenvalues is:

$$R = \det(M) - k \cdot (trace)(M)^2$$

R large means corner. Take local maxima's of R as corners.

5.1.2 SIFT

Scale-invariant Feature Transform. Doesn't care about change in 3D-viewpoint, noise, scale and rotation. Looks for local features in an image.

Use the Laplacian of Gaussian (wat dis?). Find the extrema values for different values of σ. These points provide points of interest for the SIFT-filter.

5.1.3 RANSAC

The idea is to sample random s pairs of points and call it a sample.Propose a model and compute a **consensus set** C_i, which are the points within a threshold of the model. Do this a lot of times, and chose the model that works best.

Parameters to be set:

- Minimal set s
- Threshold t. Set it by looking at the noise model.
- Number of iterations N. Chose it such that at least one of the minimal sets contain as many points from the actual model as possible.

5.2 Interest region descriptors

5.2.1 Viola Jones

Is used to detect faces. Trained used faces and non faces.

Uses four or five types of features called **Haar features**. They are blocks of either white or black. When used on an image it results in a single value where the sum of the pixel values under the white part is subtracted from the sum pixels under. Note that the features can vary in size. Doing this at each position is not feasible. Getting rid of crap features is done using **adabost** which is magic.

For every "dot-product" there is a lot of calculation. Therefor one uses something called **integral images**. The idea is for each pixel (x,y), sum up all pixel values above and to the left of that pixel. The resulting image is the integral image. Doing this lets you use only the pixel values in the corners of the Haar filter, and thus for each application of different sizes of filters you need only four inputs, given that you have already calculated the integral image.

To not have to do the whole classification for the whole image, a set of smaller classifications is done in **cascad(ing)e**. Each sub-classifier determines if it is definitely not a face, or maybe a face. If maybe, it passes it onto the next classifier. Thus the more face like an object is, the further up the chain it goes.

5.2.2 HOG

Histogram of Oriented Gradients. Divide an image into block who overlap each-other 50 %. Each block consists of 2x2 cells. Then quantize the gradient into 9 bins where each bin is representing some direction $\in (0^{\circ}, 180^{\circ})$, and a negative value represent going the opposite way. If the value is between some bin value, then divide it into two bins. I.e bins are do not represent an interval in this case.

5.2.3 DPM

Deformable Part Models. Great for detecting people at different scales, positions and deformations i.e different poses and such. Use HOG on the whole image several times, but at each time, the image is to be at different scales and then apply non-max suppression. Learns features sort of like the ones in Viola Jones, but more general and less "blocky".

5.3 Supervised Classification

5.3.1 kNN

k Nearest Neighbours.

Given n training vectors the algorithm identifies the k nearest neighbours of a new vector c, regardless of labels given. The neighbours then vote, and c is then given label or the majority vote.

Have to choose an odd k for 2 class problems. k cannot be multiple of number of classes.

5.3.2 SVM

Support Vector Machine.

Learn to separate some sets with the widest possible division between the sets. If we say that we have some line that separates the two sets, we then have a vector w, which is perpendicular to that line. Then, a new data element u is in category 2 if:

$$\mathbf{w} \cdot \mathbf{u} + b \geq 0$$

Else in category 1. So we need w and b. This is what to learn.

To be clever, and to ensure that the seperation is wider than nothing we do

$$\mathbf{w} \cdot \mathbf{u}_{+} + b \geq 1$$

$$\mathbf{w} \cdot \mathbf{u}_{-} + b \leq -1$$

Introduce some y_i such that $y_i = 1$ for positive samples, and $y_i = -1$ for negative samples.

Now for both positive and negative x:

$$y_i \cdot (\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \geq 0$$

If $\mathbf{x}$ in the "gutter" then:

$$y_i \cdot (\mathbf{w} \cdot \mathbf{x}_i + b) - 1 = 0$$

Now, since we want to maixmise the distances between the two gutters we need to express it.

If we have the closest positive and negative samples $\mathbf{x}_{+}$ and $\mathbf{x}_{-}$ and take the difference and project that along a unit vector of same direction as $\mathbf{w}$, which is normal to the gutters. Now the widt is:

$$\text{WIDTH} = (\mathbf{x}_{+} - \mathbf{x}_{-}) \cdot \frac{\mathbf{w}}{||\mathbf{w}||} = \frac{2}{||\mathbf{w}||}$$

The last step comes from the values of y_1 in the gutter.

Now, we want to maximize $\frac{2}{||\mathbf{w}||}$

Or, to be clever, we minimize:

$$\frac{1}{2}||w||^2$$

To do this, with the constraints, we use Lagrange:

$$L = \frac{1}{2}||\mathbf{w}||^2 - \sum \alpha_i \left[y_i \left(\mathbf{w} \cdot \mathbf{x}_i + b\right) - 1\right]$$

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum \alpha_i y_i x_i = 0$$

Then

$$\mathbf{w} = \sum \alpha_i y_i x_i$$

For b:

$$\frac{\partial L}{\partial b} = -\sum \alpha_i y_i = 0 \implies \sum \alpha_i y_i = 0$$

Now, L becomes

$$L = \sum \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i \cdot x_j$$

The decision rule now becomes: $\sum alpha_i y_i \mathbf{x}_i \cdot \mathbf{u} + b \geq 0 \implies +$
Solve everything by optimization.
If the set is not **linearly separable**, then make it so by using a **kernel**.