Deep learning notes

Erik Pärlstrand

February 2018

1 Hyper-parameter tuning

When to use deep learning

The following heuristic can be used to determine if deep learning is a appropriate:

1. Little noise, complex structure: use deep learning
2. Lots of noise, little structure: don't use deep learning

In order to determine which type of neural network to use, the following heuristic can be used:

1. No structure: feed forward network
2. Spatial structure: convolutional networks
3. Sequential structure: recurrent network

Bias and Variance in deep learning

Assume the following performance for a classifier (presume that Bayes error-rate is 0%):

Error	Example 1	Example 2
Train	1%	15%
Validation	11%	16%

Example 1 has low bias, high variance. Example 2 has high bias, low variance. Below is a "basic recipe"/heuristic for training a deep neural network:

1. Has the model high bias? Try: train a larger network, train longer/use better optimization algorithm, try different network architecture/hyper-parameter-search. If the model has low bias, go to step 2.
2. Has the model high variance? Try collect more data, add regularization (L_2, dropout etc), try different network architecture/hyper-parameter-search. If the model has low bias and low variance, go to step 3.
3. Test if the model works well in practice. See orthogonalization section for that.

Shallow versus deep networks

Some function can be estimated with shallow networks. However, shallower networks requires exponentially more hidden units to approximate a function compared to deeper ones.

As a rule of thumb, start with one hidden layer. Then increase if needed (i.e. to reduce bias).

Weight regularization

Two common weight regularizer are L_1 and L_2, i.e. add W^2 or $|W|$ to the cost function. L_1 creates a sparse weight matrix (in theory, however usually not the case in practice). L_2 (sometimes called weight decay) is the most commonly used in practice.

Dropout

Randomly ”drop” neurons during training. Intuition about dropout: The network can't rely on certain feature, so it has to spread the weights which leads to shrinking weights. It has similar effect as L_2-norm, however they are not the same.

One can use lower probability of keeping neurons for larger layers (i.e. more hidden units). Rarely used in input layer. One downside of dropout is that the cost function becomes stochastic.

Data augmentations

Commonly used in computer vision. For example, flip the pictures horizontally or add random distortion. Three common methods are:

1. Mirroring
2. Flip horizontally
3. Random cropping (choose a random subset of the image)

Additional methods are (not often used in practice often):

1. Rotation
2. Sheering
3. Local wrapping
4. Color shifting (add distortion to the different channels)

Early stopping

Stop the training when the validation error increases. However, this method complicates orthogonalization (see below).

Normalizing the features

One method is to ensure that all features have zero mean and unit variance. Makes gradient descent converge faster/better. It is only needed when features are on dramatically different scales.

Vanishing/exploding gradients

The gradient becomes very small/big. Problem for deep networks. Caused since the weight matrices are multiplied.

Weight initialization

Initialize the weights ”smart”. Can help with vanishing/exploding gradient. However, don’t initialize the weights to zero. This can be done for the bias terms though.

Recommended initialization for layer l with Relu activation (note that here we use variance and not standard deviation):

$$W \sim N\big(0, \frac{2}{n^{[l-1]}}\big)$$

where $n^{[l-1]}$ is the number of hidden units in the layer before l. This is called He-initialization.

For tanh layers it’s recommended to use:

$$W \sim N\big(0, \frac{1}{n^{[l-1]}}\big)$$

This is called Xavier-initialization.

Mini-batch gradient descent

In one epoch, we go through the entire training set once. During training, one can split the data into different batches (mini batches).

Guidelines: if the training set has less then 2000 examples, one can use batch-gradient descent. Otherwise, one should use mini-batch gradient descent.

Typical sizes of the mini-batches are 64, 128, 256, ... (powers of two). One should make sure that the mini-batch data fits in memory for CPU/GPU, otherwise it will drastically decrease the training speed.

Exponential moving average

$$v_t = \beta + (1 - \beta)\theta_t$$

v_t approximately average over the last $\frac{1}{1-\beta}$ values.

If one uses the initialization $v_0 = 0$, it will be a bias estimate. It can be fixed by $\tilde{v}_t = \frac{v_t}{1-\beta^t}$.

Gradient descent with momentum

Adds a moment term to the gradient descent. It's defined as:

$$v_t = \beta v_{t-1} + \alpha \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

where θ is the parameters of interest, α is the learning rate and β is the momentum term. Typically, one uses $\beta = 0.9$. The momentum optimizer usually works better than regular gradient descent.

RMSprop

RMSprop is an adaptive learning rate method proposed by Geoff Hinton. It divides the learning rate by an exponentially decaying average of squared gradients. Now let:

$$g_t = \nabla_\theta J(\theta_t)$$

The update equation for RMSProp follows:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Hinton suggests γ to be set to 0.9, while a good default value for the learning rate α is 0.001, however this depends on the application.

Adam

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum. We compute the decaying averages by:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

One usually bias correct them by:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

The update equation for Adam follows:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

Typical value are $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The default values usually works well in practice (tune if needed). A good rule of thumb is to use Adam as default optimizer.

Learning rate decay

One can decay the learning rate a bit after each epoch:

$$\eta = \frac{\eta_0}{1 + DecayRate \times EpochNumber}$$

However, the most common approach is to decay in steps, i.e. decay the learning rate by 10% every 5:th epoch, for example.

Hyper-parameter tuning

To achieve low bias, learning rate η is usually the most important hyper-parameter. Then the number of hidden units in the layers. Then the number of layers and learning rate decay. Last, one can try to change Adam's hyper-parameters. However, this order depends on the problem.

To achieve low variance, regularization is the most important together with the ones above.

One should use random search instead of grid search. It is a good idea to start with large ranges of the hyper-parameters to see which values makes "sense" (coarse search) and then create smaller ranges from these results (fine search).

If the hyper-parameters are on the same scale, one can sample uniformly. However, if they are not one should sample on log-scale. Assume that we want to investigate the range $[10^{-9}, 10^{-6}]$. Sample $r \in [-9, -6]$ uniformly. Then the hyper-parameters values are set to 10^r.

Panda versus caviar approach

One can baby sit one model (called the panda approach). Good when one doesn't have a lot of computational resources. Check the performance and update the model/hyper-parameters accordingly.

One can also train many models in parallel (called caviar approach). Good if one has a lot of computational resources.

Batch-normalization

Normalize the mean and variance before each activation step (some advocate to normalize after the activation step), i.e.

$$z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sigma}$$

$$\hat{z}^{(i)} = \gamma z_{norm}^{(i)} + \Gamma$$

where γ and Γ are learnable parameters through back-propagation. Use $\hat{z}^{(i)}$ in forward and back-propagation. Useful for covariance shift, i.e. reduce distribution shift for different layers.

It has some regularizing effect, however it should not be used for that.

During training, keep track of μ and σ with an EMA from the mini-batches. Use the mean and standard deviation to normalize z at test time.

2 Practical aspect of deep learning

Orthogonalization

Split a problem into orthogonal parts:

1. Fit the training set well given a cost function. This can be improved by larger networks, better optimizer (Adam etc). If satisfied, move to (2).
2. Fit the validation set well given a cost function. This can be improved by regularization, more data etc. If satisfied, move to (3).
3. Make sure that the model works well on the test set. This can be improved by getting a larger validation set. If satisfied, move to (4).
4. Make sure that the model works well in practice. This can be improved by changing the validation set or cost function.

Single number evaluation metric

Before creating a model, define and use a single number evaluation metric (accuracy, F_1 etc). Use your validation set and evaluation metric to choose model/direction. This leads to faster iterations of the modelling.

Satisfying and optimizing metric

Use the evaluation metric to (globally) optimize the model. However, one can have satisfying metric that the model needs to accomplish, for example run-time. Choose the model that has the highest/lowest optimizing metric s.t. the model accomplish the satisfying metrics.

For example, maximizing accuracy s.t. the run-time of the model is less then 1000 ms. If you have N metrics of interest, optimize one and have the $N-1$ as satisfying metrics.

train/validation/test distribution

Validation and test set should come from the same distribution. However, the training set can come from another one. Choose the validation and test set to reflect the data you expect to get in the future and consider important to do well on.

Rule of thumb the size of the different sets

If data size $< 10,000$: Use 60% for training, 20% for validation and 20% for test.

If data size $> 1,000,000$: Use 98% for training, 1% for validation and 1% for test.

The size of the test set should be big enough to give a high confidence in the overall performance of the model. It's recommended to have a train, validation and test set (to get an unbiased estimate of the performance).

If the current metric doesn't reflect your "goal", redefine it. For example, add weights to the different classes when computing the accuracy. Can also add weights to the loss function. Moreover, if doing well on your metric + validation/test set doesn't correspond to doing well on your application, change your metric and/or validation/test set.

Furthermore, if the performance of the validation and test set differ a lot, you have over-fitted the validation set and should get more validation data.

Comparing to human-level performance

Humans are quite good at a lot of tasks, for example image/speech recognition etc. So as long as the model is worse than humans, you can:

1. Get labeled data from humans
2. Gain insight from manual error analysis
3. Better analysis of bias/variance

Human-level error can be viewed as a proxy for the Bayes error-rate (i.e. lowest possible error).

Below is an example of a cat classifier:

Error	Example 1	Example 2
Human	1%	7.5%
Train	8%	8%
Validation	10%	10%

In example 1, one should focus on bias reduction. In example 2, one should focus on reducing variance.

Methods for reducing bias: train bigger model, train longer/use better optimization algorithm (RMSprop, Adam etc) or change network architecture/hyper-parameter search.

Methods for reducing variance: get more data, regularization (L^2, dropout, data augmentation) or change network architecture/hyper-parameter search.

Error analysis

Manually inspect the errors (on the validation data). Example: should you try to make your cat classifier do better on dogs? Below is an example of a workflow for error analysis (assume that the validation error is 10%):

1. Get $\sim$ 100 mislabeled validation data points.
2. Manually go through the errors, count how many are dogs?
3. If 5/100 are dogs, probably not worth investigating the cause of the error.
4. If 50/100 are dogs, probably worth investigating.

Ideas to improve the cat classifier:

1. "Fix" images of dogs being recognized as cats
2. "Fix" great cats (lions etc) being mis-classified
3. Improve performance on blurry images
4. etc

Document the validation error (in a spread-sheet for example):

Images	Dogs	Great cats	incorrect labeled	...	comments
1	Yes	No	...	...	Pitbull
2	No	Yes	...	...	Lion
...	...	...	...	...	...
% of total errors	8%	43%	10%	...	...

Cleaning up incorrect labeled data

Deep learning algorithms are quite robust to random errors in the training set. However, less robust to systematic errors. In error analysis, add columns for incorrect labels in the table above. If it makes a significant different, one can try to fit it.

However, one should apply the same process to your validation and test set to make sure they come from the same distribution.

Build your first system quickly, then iterate

1. Set up your validation and test set and choose a metric
2. Build your first system quickly, then iterate
3. Use bias/variance analysis and error analysis to prioritize the next steps

Less applicable if you have prior knowledge within the field or if there is a large academic literature on the subject.

Training and testing on different distributions

Example: You have 200,000 pictures of cats from the web (web-crawling) and 10,000 from your app (i.e. the performance you care about). Use the web pictures for training (maybe some data from the app). Use all/most data from the app for the validation and test set.

Bias variance with mismatching data distributions

Introduce another set, training-validation set. Comes from the same distribution as the training set, but not used for training. Example for the cat classifier (assume Bayes error-rate is 0%):

Error	Example 1	Example 2	Example 3
Train	1%	1%	10%
Train-validation	9%	1.5%	11%
Validation	10%	10%	12%

In example 1, we have a variance problem. In example 2, we have a data-mismatch problem. In example 3, we have a bias problem.

Addressing data mismatch

Potential routes:

1. Carry out manual error analysis to try to understand the difference between train and validation/test set (i.e. if they have different distributions).

2. Make training data more similar to validation/test data, or collect more data similar to the validation/test set.

Can also try artificial data synthesis. For example, if there is a lot of car noise in the validation/test set, try to add original voice + car noise to generate the synthetic data. However, one should be careful to not overfit the car noise in this example if we use the same noise multiple times.

Transfer learning

Use pre-trained networks and tune the last/multiple layers for your application. The transfer is from A to B. Make sense when:

1. Task A and B have similiar type of input (images etc)

2. You have a lot more data for task A than for task B

3. Low level features from A could be helpful for task B

Rule of thumb regarding how many layers should be re-trained:

If you have a small amount of data, replace the last few layers (i.e. the fully connected layers and output layer). The earlier layers are ”freezed”, i.e. not updated in gradient descent. Called fine-tuning.

If you have a lot of data, you can re-trained the entire network. The earlier layers becomes initialization. However, the fully connected layers and output layer needs to be re-trained from scratch. Called pre-training.

End-to-End deep learning

Let the network come up with the features, i.e. use ”raw” data.

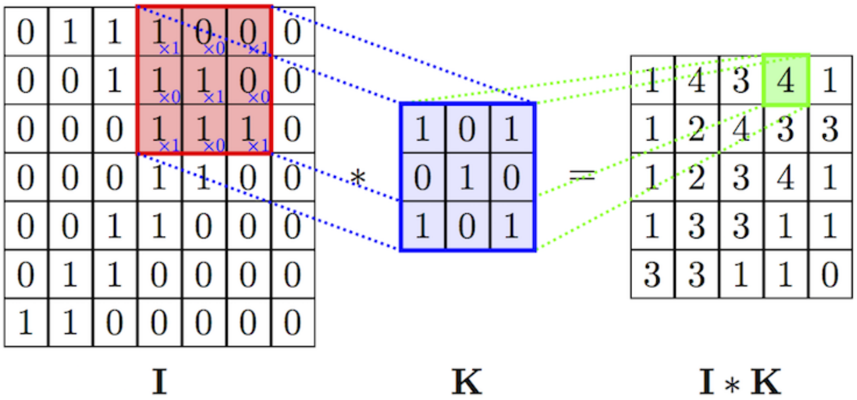
Pros:
- Let the data speak
- Less hand-designing of components

Cons:
- May need a lot of data
- Exclude potentially useful hand-designed components.

3 Convolutional networks

Convolutional operation

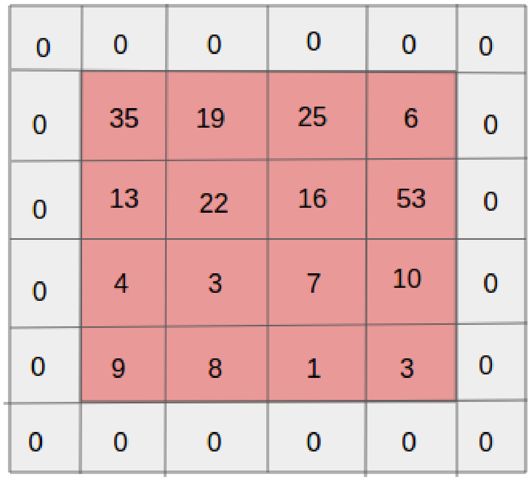
Convolution occurs between two matrices, an image (I below) of size $n \times n$ and a filter (K below) of size $f \times f$, where f is usually odd. Below is an example of a convolution operation between I and K (stride is one):



Traditionally, K was constructed by hand. But with deep learning, they are estimated with back propagation.

Padding

When doing convolution the edges of the picture gets included in fewer convolutional operations then the central pieces. Therefore, one can add zeros around the edges to address this problem, i.e. zero padding. The picture below is an example when we have added padding of size 1 (p=1):



0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Also, one problem for convolutional networks is that the size of the picture decreases after each convolution. Therefore, zero padding helps to keep the size of the image constant if so needed. If we want the input size to be equal to the output size, set $p = \frac{f-1}{2}$. There are two common terms for padding:

- Valid convolution - use no padding
- Same convolution - keep the output size the same as input size (see above).

Strided convolution

In the convolutional operation section, we used a stride of size 1 (s=1), i.e. we moved the convolutional filter one step after each operation. However, one can use a larger number, for example $s = 2$. Then we will jump two steps after each operation. The output size with a given stride (s), image size (n), padding size (p) and filter size (f):

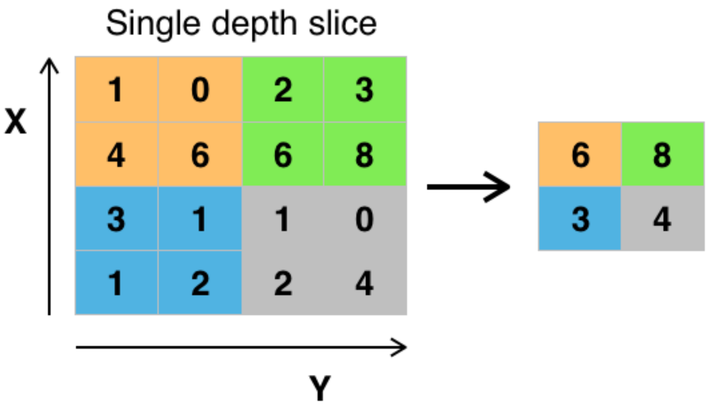
$$\text{output} = \frac{n + 2p - f}{s} + 1$$

Convolution over volume

Previously, we did convolution over an 2D-area (e.g. grey-scaled image). However, we could also do it over a volume, for example $n \times n \times 3 * f \times f \times 3$, i.e. we have stacked 3 filters/channels.

Pooling

The max-pooling operation is illustrated below (with filter size 2 and stride 2):



1	0	2	3
4	6	6	8
3	1	1	0
1	2	2	4

6	8
3	4

If a useful feature is detected, keep it. You can look at it a little like focusing the network's attention on what's most characteristic for the image at hand.

One can also use average pooling, i.e. take the average within the region. Average pooling is sometimes used in deep networks in the end. However, but max-pooling is used more often for most layers.

Convolution networks advice and intuition

Earlier layers detects lower level features (edges etc) and later layers detects higher level features (faces etc).

In ConvNets, there are three types of layers; convolutional, pooling and fully connected layer. For network architecture, look in the literature for advice. One typical layer sequence is Conv → Pooling → Conv → .. → Fully connected. Also, typical value of the filters and strides are:

$$f = 2, s = 2$$

$$f = 3, s = 2$$

Typically, for ConvNets n decreases along the layers and the number of channels increases. However, we don't want the size (n) to drop too fast.

Why convolution?

Below are some advantageous properties of convolutional layers:

Number of parameters

Assume that we have a picture of size $1000 \times 1000 \times 3$ and that we will use a fully connected layer of size 1000. That would becomes 3 billion parameters in the first layer, which is intractable. With a convolution layer we would only get 27 (assuming 3x3x3 filter).

Parameter sharing

A feature detector that's is useful in one part of the image is probably useful in another part of the image.

Sparsity of connections

In each layer, each output value depends only on a smaller number of local inputs.

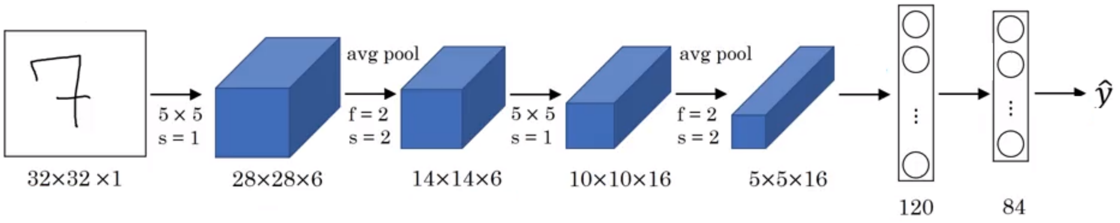
Translation invariance

It doesn't matter how the object is placed. For example, it classify a cat even if it's rotated.

Common architectures

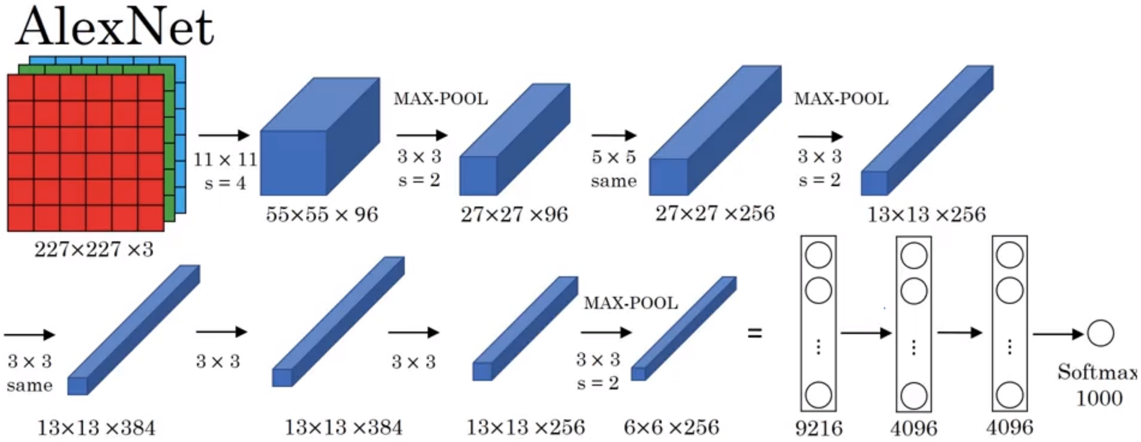
LeNet

The first convolutional networks. Has around 60k parameters. Deeper in the model the height/width decreases and the number of channels increases. Below is the architecture of it:



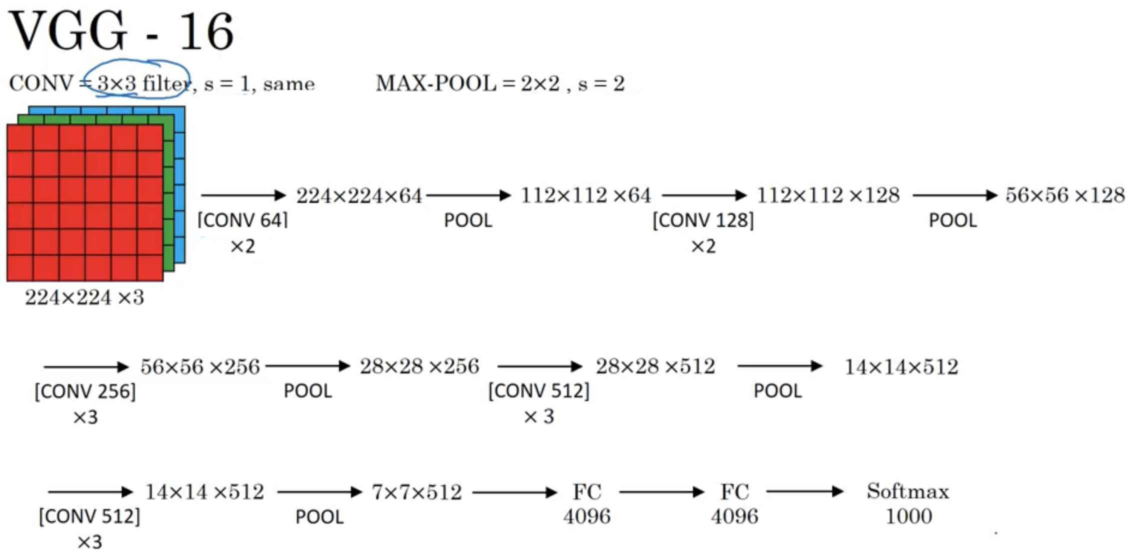
AlexNet

One of the first ”new” convolutional networks. Has around 60M parameters. Deeper in the model the height/width decreases and the number of channels increases. Below is the architecture of it:



VGG-16

Has around 138M parameters. Deeper in the model the height/width decreases and the number of channels increases. The height/width decreases with a factor of two with each ”block” and the number of channels increases by a factor of two with each ”block”. Below is the architecture of it:

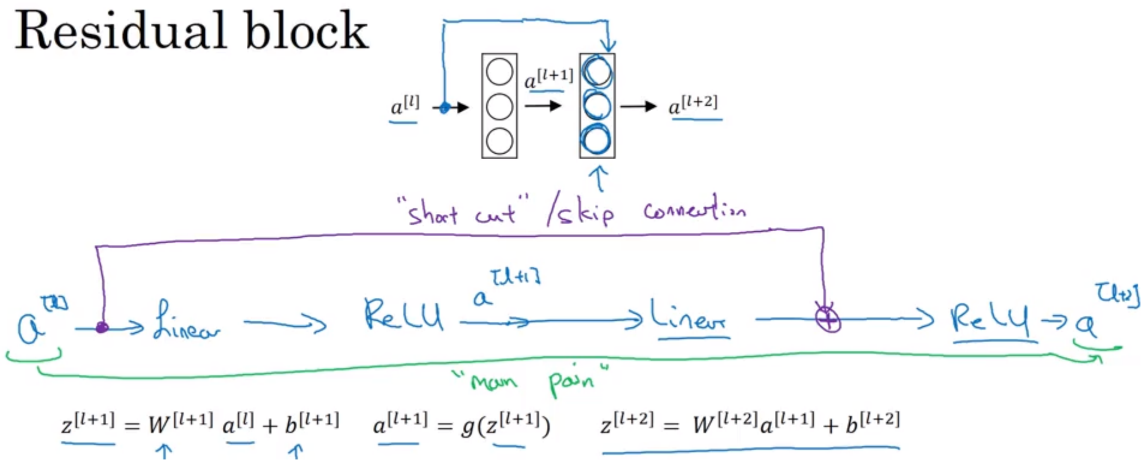


VGG - 16

CONV = 3×3 filter, s = 1, same   MAX-POOL = 2×2 , s = 2

ResNet

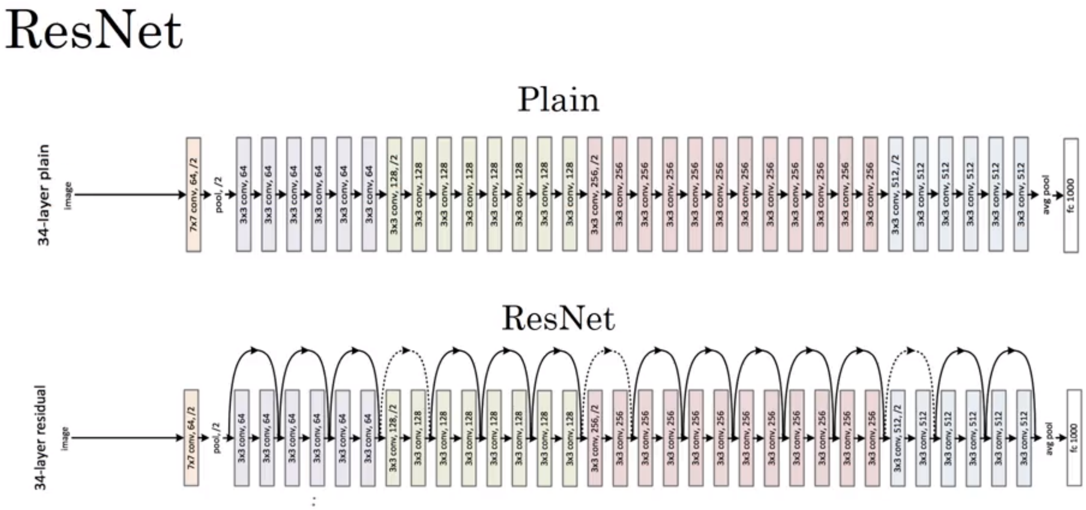
Use skip connections which allows for very deep networks. Below is residual block:



Residual block

$z^{[l+1]} = W^{[l+1]} a^{[l]} + b^{[l+1]}$ $a^{[l+1]} = g(z^{[l+1]})$ $z^{[l+2]} = W^{[l+2]} a^{[l+1]} + b^{[l+2]}$

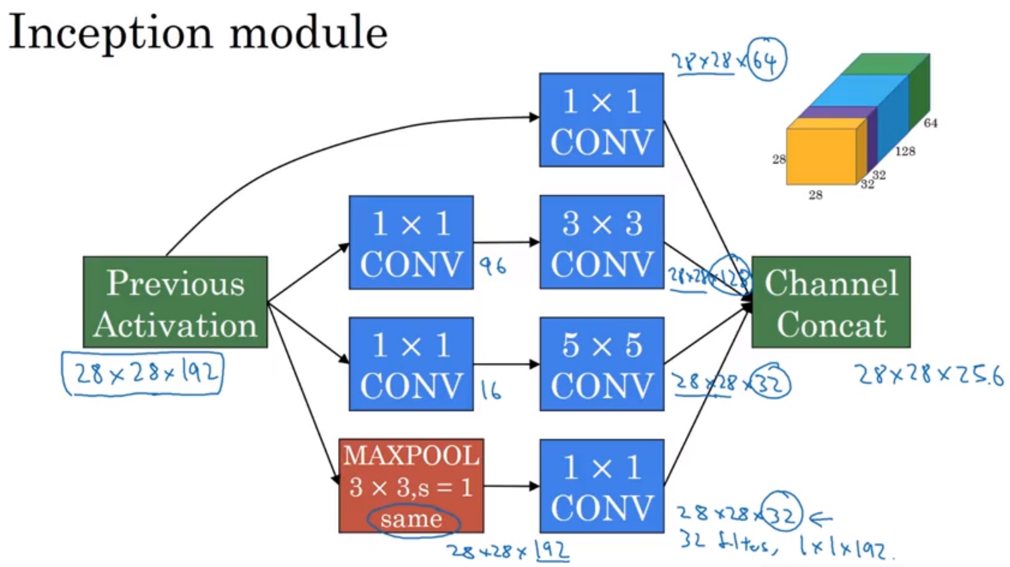
ResNet architecture:



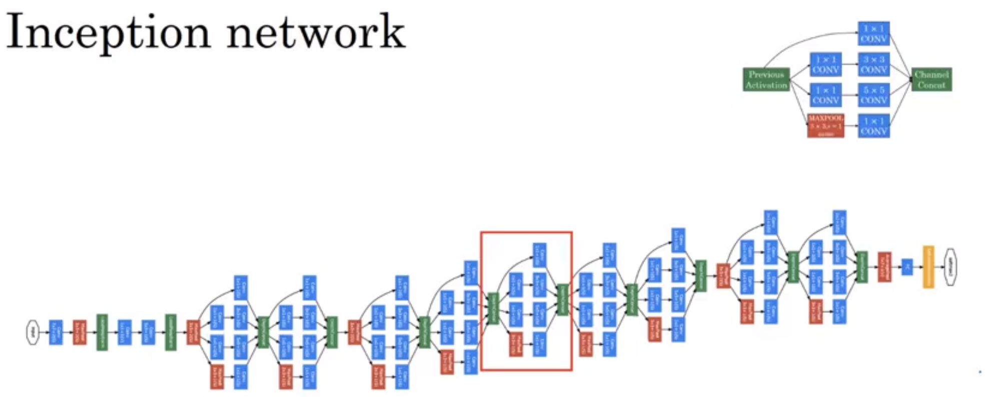
ResNet

Inception network

”Network in Network”. Uses 1 × 1 filter to decrease the number of channels. One inception module is:



The inception network is:



Object detection

In object detection, one tries to identify and localize multiple objects, i.e. set bounding box around the objects.

One simple method is to train a conv-net on small (cropped) images. Then use a sliding windows on the "big" image. One typically start with a small window and increase it sequentially. However this is not efficient (in terms of computational time).

One improvement is to share the computation between windows (i.e. convolutional implementation of the sliding window). The output becomes a tensor. However the bounding boxes are usually not good. One improvement to this is the YOLO-algorithm.

Intersection over union

Measures overlap between two bounding boxes. Defined as:

$$\text{IoU} = \frac{\text{Area of intersection}}{\text{Area of union}}$$

One usually claims that there is overlap if IoU > 0.5.

None-max suppression

Find highest probability within each box. Suppresses other boxes that have high IoU.

YOLO-algorithm

YOLO reframe object detection as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities.

A single convolutional network simultaneously predicts multiple bounding boxes and class probabilities for those boxes. YOLO trains on full images and directly optimizes detection performance. This unified model has several benefits over traditional methods of object detection, such:

- Speed
- Reasons globally about the image when making predictions
- Learns generalizable representations of objects

The system divides the input image into an $S \times S$ grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object.

Each grid cell predicts B bounding boxes and confidence scores for those boxes. Formally we define confidence as $P(\text{Object}) * \text{IoU}$. If no object exists in that cell, the confidence scores should be zero. Otherwise we want the confidence score to equal the intersection over union (IoU) between the predicted box and the ground truth.

Each bounding box consists of 5 predictions: x, y, w, h, and confidence. The (x, y) coordinates represent the center of the box relative to the bounds of the grid cell. The width and height are predicted relative to the whole image.

Finally the confidence prediction represents the IoU between the predicted box and any ground truth box. Each grid cell also predicts C conditional class probabilities, $\mathrm{P}(\mathrm{Class}_i|\mathrm{Object})$. These probabilities are conditioned on the grid cell containing an object. We only predict one set of class probabilities per grid cell, regardless of the number of boxes B.

At test time we multiply the conditional class probabilities and the individual box confidence predictions,

$$P(\text{Class}_i|\text{Object}) * P(\text{Object}) * \text{IOU} = Pr(\text{Class}_i) * \text{IOU}$$

which gives us class-specific confidence scores for each box. These scores encode both the probability of that class appearing in the box and how well the predicted box fits the object.

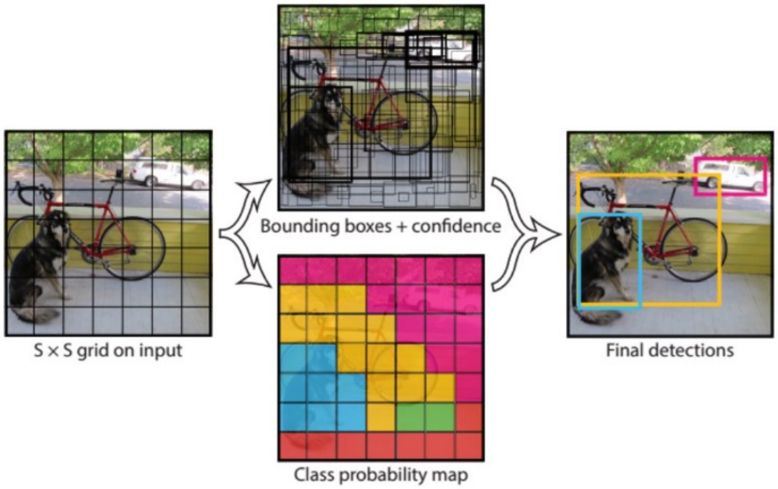


Figure 2: The Model. Our system models detection as a regression problem. It divides the image into an $S \times S$ grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor.

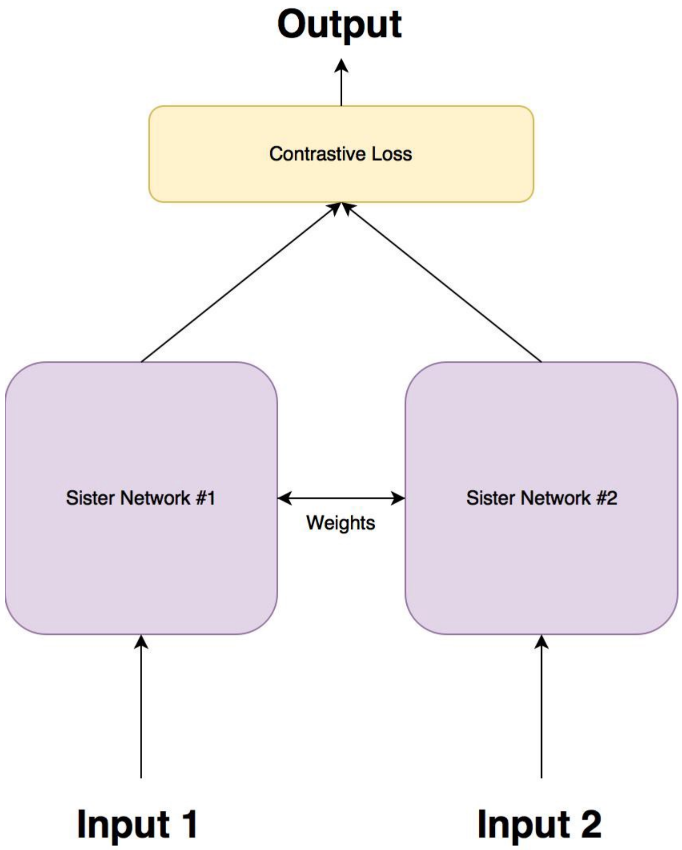
Face recognition/verification

For face verification, given an input image verify whether the person is the claim one (person).

For face recognition, given a database of K persons and an input image, check if the image/person is any of these K persons.

Siamese network

Siamese neural network is a class of neural network architectures that contain two (or more) identical sub-networks. identical here means they have the same configuration with the same parameters and weights. Parameter updating is mirrored across both sub-networks. Below is an example of the network:



The objective of the network is not to classify input images, but to differentiate between them. So, a classification loss function (such as cross entropy) would not be a good fit. Instead, this network is better suited to use a contrastive function. Intuitively, this function just evaluates how well the network is distinguishing a given pair of images. It's defined as:

$$\frac{(1-y)}{2}D^2 + \frac{y}{2}(\max(0, m - D))^2$$

where D is defined as the euclidean distance between the outputs of the siamese networks, $D = \sqrt{(h(X_1) - h(X_2))^2}$. h is the output of one of the networks, X_1 and X_2 is the input data-pair.

Here, y is either 1 or 0. If the inputs are from the same class, then the its value is 0, otherwise 1.

m is a margin value which is greater than 0. Having a margin indicates that dissimilar pairs that are beyond this margin will not contribute to the loss. This makes sense, because you would only want to optimise the network based on pairs that are actually dissimilar, but the network thinks are fairly similar.

4 Sequential models

Recurrent neural networks

The idea behind RNNs is to make use of sequential information. In a traditional neural network we assume that all inputs are independent of each other. But for many tasks that's a bad assumption.

If you want to predict the next word in a sentence knowing which words came before it is important. RNNs are called recurrent because they perform the same task for every element of a sequence. Another way to think about RNNs is that they have a "memory" which captures information about what has been calculated so far.

There are many applications where we have sequences. The figure below shows some applications (from left to right):

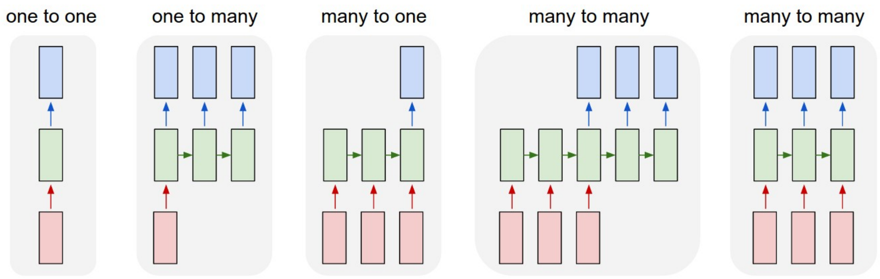
1. Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification).

2. Sequence output (e.g. image captioning takes an image and outputs a sentence of words).

3. Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment).

4. Sequence input and sequence output (e.g. machine translation).

5. Synced sequence input and output (e.g. video classification where we wish to label each frame of the video).



Now let x_t be the input at time t, h_t be the hidden state at time t and $\hat{y}_t$ be the predicted value. RNN's forward equation is given by:

$$h_t = f(Wx_t + Uh_{t-1} + b_h)$$

$$\hat{y}_t = g(Vh_t + b_y)$$

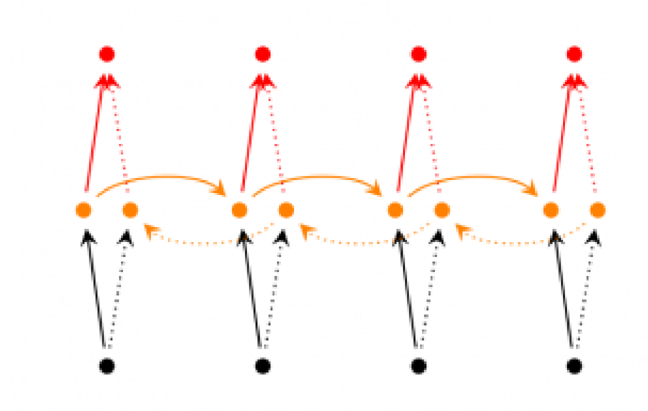
for $t = 0, ..., T$, where f is the activation function (typically tanh) and g is the output activation (typically sigmoid/softmax). h_0 is typically initialized to zero. The output $\hat{y}$ depends on the application:

If we have the many to one structure we will only have one output $\hat{y}_T$ and the loss function will be defined as $L(\hat{y}_T, y_T)$.

If we have the one/many to many structure we will have T outputs $\hat{y}_1, \hat{y}_2, ..., \hat{y}_T$ and the loss function will be defined as $\sum_t L(\hat{y}_t, y_t)$.

Bidirectional RNN

Bidirectional RNNs are based on the idea that the output at time t may not only depend on the previous elements in the sequence, but also future elements. For example, to predict a missing word in a sequence you want to look at both the left and the right context. Bidirectional RNNs are quite simple. They are just two RNNs stacked on top of each other. The output is then computed based on the hidden state of both RNNs. Below is a figure describing them:



Let $h_{t,f}$ be the hidden state at time t in the forward direction and $h_{t,b}$ be the hidden state at time t in the backward direction. The update equation follows:

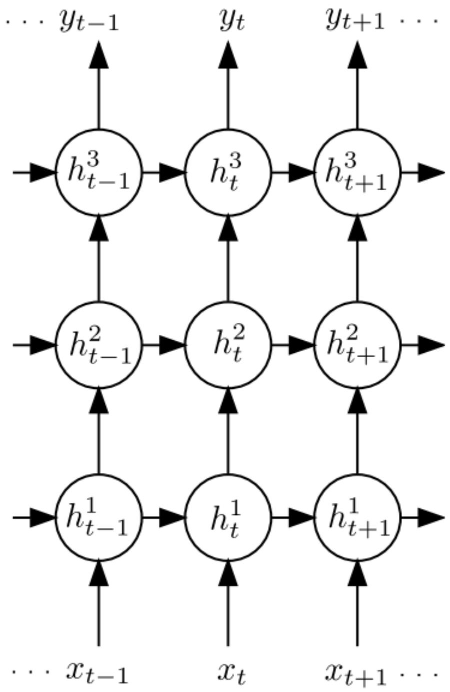
$$h_{t,f} = f(W_f x_t + U_f h_{t-1,f} + b_{h,f})$$

$$h_{t,b} = f(W_b x_t + U_b h_{t+1,b} + b_{h,b})$$

$$\hat{y}_t = g(V[h_{t,f}, h_{t,b}] + b_y)$$

Deep RNN

We could stack multiple RNN on top of each other, creating a deep RNN. This gives us a higher learning capacity. The figure below describes the process:



Vanishing and exploding gradients

RNNs often suffers from vanishing/exploding gradients. RNNs is trained by backpropagation through time and therefore its unfolded into a very deep feed forward net. When the gradient is passed back through many time steps, it tends to grow or vanish, similar to deep feed forward nets.

Exploding gradients are fairly easy to detect and combat, e.g. gradient clipping. However, vanishing gradients are harder to overcome.

Another related issue is that RNNs have local dependencies which makes it hard to learn long-term dependencies.

Fortunately, there are other recurrent neural networks that addresses these problems, namely Long Short Term Memory networks (LSTM) and Gated Recurrent Units (GRU).

LSTM

The key to LSTMs is the cell state. The cell state is kind of like a conveyor belt; it runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged. The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through", while a value of one means "let everything through". An LSTM has three of these gates:

The forget gate decides what information we will through away from the cell state. It's defined as:

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

The input gate decides what information we will add from a new candidate $\tilde{C}_t$:

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_c[h_{t-1}, x_t] + b_c)$$

The new cell state C_t is updated according to:

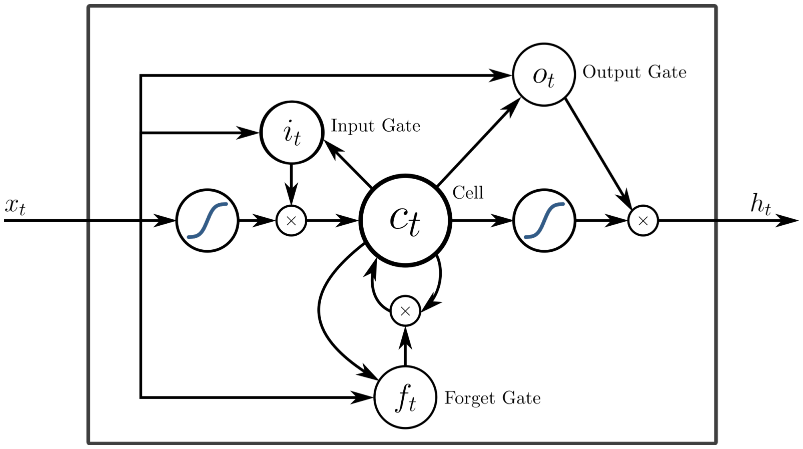
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

where $*$ described a point-wise multiplication. Finally, we have the output gate which decides what information we will output and what information will be added to the hidden state:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

Note that there are many different versions of LSTMs, this is the "standard" one. Below is a figure of the LSTM process:



GRU

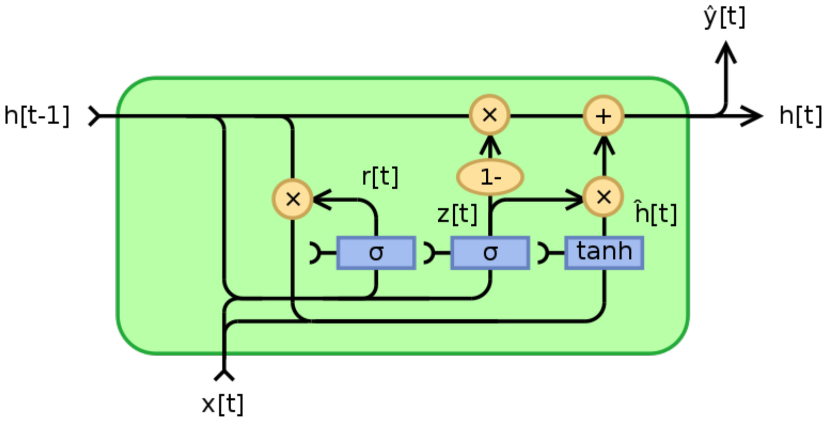
GRU is an alternative to LSTM. It has fewer parameters than LSTM, but are in most cases less flexible. Similar to LSTM, GRU has gates, namely the update gate z and the reset gate r:

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z)$$
$$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r)$$

The hidden state h is defined as:

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tanh(W_h x_t + U_h(r_t * h_{t-1}) + b_h)$$

The output of the network is the same as the hidden state, i.e. $\hat{y}_t = h_t$.



Word representation

We would like to represent words in a numeric form in order to use them as features. There are two common ways of doing this:

The first is to use a one-hot encoding representation for each word. However this approach is bad at generalizing between words.

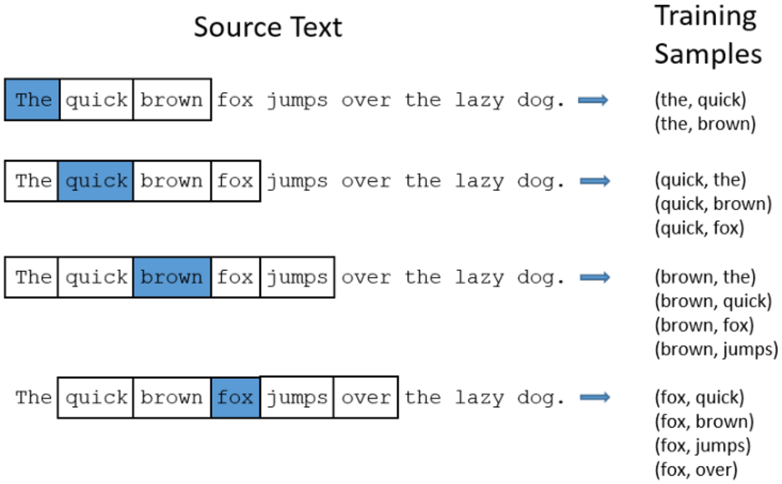
The other approach is with word embeddings, i.e. create a representation for words that generalize better. There are two common ways of learning these word embeddings; word2vec and Glove. However, we will only cover word2vec.

One can view this approach as a form of transfer learning, i.e. learn the word embeddings from a large text corpus (not uncommon to use ~ 1 billion words) or download pre-trained embeddings online. Then use these embeddings for another task, e.g. sentiment analysis. This is especially good when the task does not have a lot of training data.

Word2Vec

Word2Vec is an algorithm for learning word embeddings. There are two versions; CBOW and skip-gram. Given a set of sentences the model goes through the words of each sentence and either tries to use the current word to predict its context, in which case the method is called skip-gram, or it uses each of these contexts to predict the current word, in which case the method is called CBOW (”Continuous Bag Of Words”).

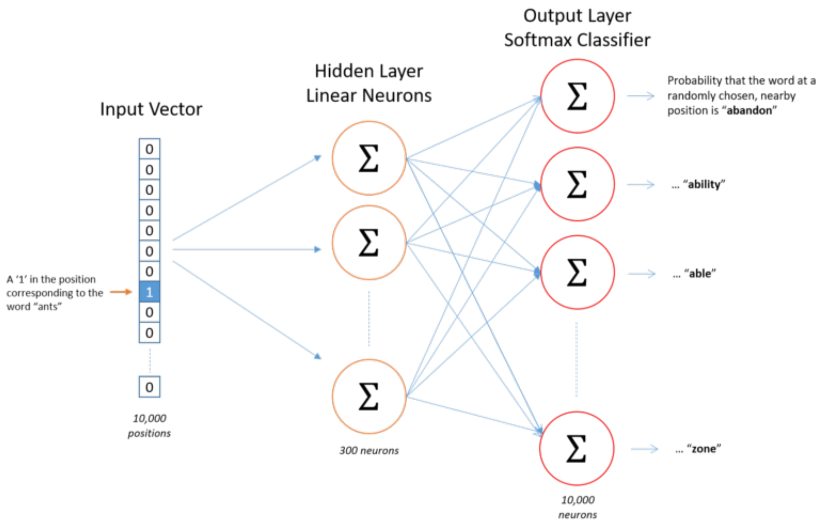
The number of words in each context is determined by the window size parameter. A rule of thumb is to use a window size of $5 - 20$ for smaller data sets and $2 - 5$ for larger ones. Below is an example of skip-gram from the sentence ”The quick brown fox jumps over the lazy dog” with a window size of 2:



The model will be a one-layer neural network. The network is trained by doing the following:

Given a specific word in a sentence, look at the words nearby and pick one at random (or by a specified distribution). The network output will tell us the probability for every word in the vocabulary of being the ”nearby word”. For example, if you gave the trained network the input word ”Soviet”, the output probabilities are going to be much higher for words like ”Union” than for ”watermelon”.

The words (for example ”ants”) will be encoded as a one-hot vector (both input and output) with a size equal to the vocabulary size. Below is an example of the setup:



For Word2Vec we are interested in the hidden representation (from the figure above the 300 dimensional vector produced by the network). We can measure similarities between word embeddings by:

$$\text{sim}(u,v) = \frac{u^T v}{||u||_2||v||_2}$$

The word embeddings have semantical meaning, e.g. $v_{\text{man}} - v_{\text{women}} \approx v_{\text{king}} - v_{\text{queen}}$.

Subsampling

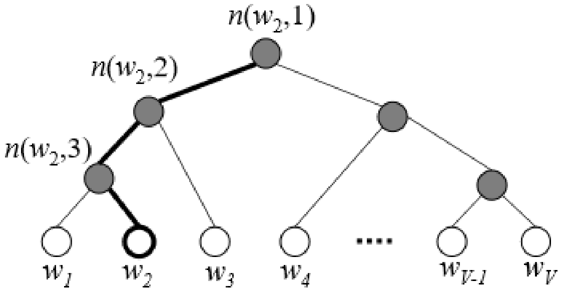
With word2vec there is a problem with common words like ”the” since it doesn’t tell us much about the meaning of the surrounding words (like ”fox” in the example above). Therefore word2vec uses subsampling schemes to address this. For each word we encounter in our training text, there is a chance that we will effectively delete it from the text. The probability that we cut the word is related to the word’s frequency.

Hierarchical softmax

The training speed for models with softmax output layers quickly decreases as the vocabulary size grows. This is due to the linear increase in parameter size and computation cost with respect to the vocabulary. A computationally efficient approximation of the softmax is the hierarchical softmax.

In hierarchical softmax the output layer is represented as a binary tree with the V words in vocabulary as its leaf nodes and the intermediate nodes as internal parameters. Each intermediate node explicitly represents the relative probabilities of its child nodes.

The idea behind decomposing the output layer to a binary tree was to reduce the complexity to obtain probability distribution from $O(V)$ to $O(log(V))$. For each leaf node there exists a distinct path from root node and this path is used to estimate the probability of the word represented by the leaf node.



Above is an example hierarchical binary tree, the leaf nodes represent the words and the inner nodes represent probability mass. The highlighted nodes and edges make a path from the root to an example leaf node w_2 with a length of the path $L(w_2) = 4$. Furthermore, $n(w, j)$ means the j-th node on the path from the root to a leaf node w.

In hierarchical softmax model, each of the $V-1$ intermediate node has an output vector $v'_{n(w,j)}$. And the probability of a word being the output class follows:

$$p(w = w_o) = \prod_{j=1}^{L(w)-1} \sigma\big(\llbracket n(w, j+1) = ch(n(w, j))\rrbracket v^T_{n(w,j)} h\big)$$

where $\llbracket\rrbracket$ is 1 if true else -1, $ch(n)$ is the left child of n and h is the output of the hidden layer.

Language modelling

A statistical language model is a probability distribution over sequences of words. Given such a sequence of length T_y, it assigns a probability $p(y_1, \ldots, y_{T_y})$ to the whole sequence.

Having a way to estimate the relative likelihood of different phrases is useful in many natural language processing applications, especially ones that generate text as an output. Language modeling is used in speech recognition, machine translation and more.

There are two common ways of represent the words:

The first is to use a one-hot encoding for the vocabulary. For example, choose the $10,000$ most common words in English for the vocabulary. If a word is not in the vocabulary, use the tag <UKN>. It's not uncommon to have a vocabulary size of $30,000$ for commercial applications.

The other approach is to use each character as a one-hot encoding. For example, the word "pear" would become a sequence (p, e, a, r) where each letter is a one-hot encoding. This allows for a smaller one-hot vectors (since there are fewer characters than words). However, it typically does not work well for longer sequences.

Sample sequences

$p(y_1, \ldots, y_{T_y})$ can be defined by conditional probabilities, i.e.:

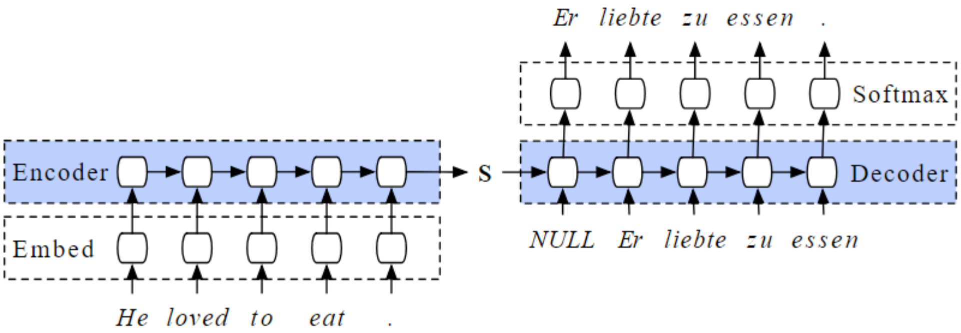
$$p(y_1, \ldots, y_{T_y}) = \prod_{i=1}^{T_y} p(y_i | y_1, \ldots, y_{i-1})$$

For a recurrent neural network with a sequence output, we can sample from the softmax probabilities $\hat{y}_t$ at each time step. For example, assume that we sample the word "the" for $t = 1$, "cat|the" for t=2 and so on. The probability of the sentence is given by the product of the probability of each words.

Machine translation

Assume that we would like to translate English sentences into French sentences. Given a sequence of words in English $x_1, \ldots, x_{T_x}$ we would like to translate it into a sequence of words in French $y_1, \ldots, y_{T_y}$. Note that in general $T_x \neq T_y$.

In traditional machine translation models one uses an encoder-decoder framework, i.e. given the sentence in English encoding it into a state s. Then decode the state s it into a French sentence. Below is an example of the process:



We are interested in the probability of the French sentence given the English sentence, i.e.:

$$p(y_1, \dots, y_{T_y} | x_1, \dots, x_{T_x})$$

This is usually modelled with a recurrent neural network with a sequence to sequence structure.

Beam search

We would like to receive the sentence in French that has the highest probability, i.e. $\max_{y_1, \dots, y_{T_y}} p(y_1, \dots, y_{T_y} | x_1, \dots, x_{T_x})$. However the y's has a non-trivial impact on each other, i.e. maximizing $p(y_i | x_1, \dots, x_{T_x}, y_1, \dots, y_{i-1})$ greedily for every i does not necessarily yields the highest joint probability.

Beam search is a search method which tries to find the maximal probability by expanding all possible next steps and keeps the B most likely, where B is the Beam width.

Beam search follows the following step:

1. Choose the beam width B.
2. Repeat step 3-4 for $i = 1, \dots, T_y$:
3. Compute the probability for each word in the vocabulary given the previous B words at time $i - 1$.
4. Keep the B highest probability words for time i.

Note that if $B = 1$ we will do a greedy search. Unlike exact search algorithms (e.g. breadth first search), beam search runs faster but is not guaranteed to find exact maximum. Larger B gives better results but requires more computation. Smaller B yields worse results but requires less computation. Typically, one uses $B \approx 10$.

Remember that beam search tries find $\hat{y}_1, \dots, \hat{y}_{T_y}$ such that:

$$\text{argmax}_{\hat{y}_1, \dots, \hat{y}_{T_y}} \prod_{t=1}^{T_y} p(\hat{y}_t | x_1, \dots, x_{T_x}, y_1, \dots, y_{t-1})$$

In order to avoid numerical underflow one usually does a log-transformation, i.e.

$$\text{argmax}_{\hat{y}_1, \dots, \hat{y}_{T_y}} \sum_{t=1}^{T_y} log\big(p(\hat{y}_t | x_1, \dots, x_{T_x}, y_1, \dots, y_{t-1})\big)$$

However since $p(\ldots) \leq 1$, beam search will prefer shorter sentences. Therefore, one usually normalize the expression by the length, i.e.

$$\text{argmax}_{\hat{y}_1,\ldots,\hat{y}_{T_y}} \frac{1}{T_y^\alpha} \sum_{t=1}^{T_y} log\big(p(\hat{y}_t|x_1,\ldots,x_{T_x},y_1,\ldots,y_{t-1})\big)$$

where α is a hyper-parameter to tune. In practice, $\alpha = 0.7$ usually works well.

Error analysis in beam search

Assume that the translation does not work as expected. We then have two components that could be at fault: the RNN model or the beam search (i.e. the beam width is too small). Remember that the output from the RNN is $p(y|x)$. Let y^* be the human translation and $p(\hat{y}|x)$ be beam search's output.

If $p(y^*|x) > p(\hat{y}|x)$: Since beam search chooses $\hat{y}$, but y^* attains higher probability beam search is at fault.

If $p(y^*|x) < p(\hat{y}|x)$: y^* is a better translation than $\hat{y}$ but RNN predicts the reverse. Therefore the RNN is at fault.

If beam search is the problem, increase B. If RNN is the problem, try to decrease bias/variance in the model.

Bleu score

One of the challenges of machine translation is that given a sentence in a language there are one or more possible good translation in another language. So how do we evaluate our results? The way we do this is by using BLEU score. BLEU stands for bilingual evaluation understudy.

BLEU uses a modified form of precision to compare a candidate translation against multiple reference translations. Below is an example of poor machine translation output with high precision:

Candidate: the the the the the the the

Reference 1: the cat is on the mat

Reference 2: there is a cat on the mat

Of the seven words in the candidate translation, all of them appear in the reference translations. Thus the candidate text is given a unigram precision of:

$$\frac{m}{w_t} = \frac{7}{7} = 1$$

where m is number of words from the candidate that are found in the reference, and w_t is the total number of words in the candidate. This is a perfect score, despite the fact that the candidate translation above retains little of the content of either of the references.

The modification that BLEU makes is fairly straightforward. For each word in the candidate translation, the algorithm takes its maximum total count, m_{max}, in any of the reference translations. In the example above, the word "the" appears twice in reference 1, and once in reference 2. Thus $m_{max} = 2$.

For the candidate translation, the count m_w of each word is clipped to a maximum of m_{max} for that word. In this case, "the" has $m_w = 7$ and $m_{max} = 2$, thus m_w is clipped to 2. These clipped counts m_w are then summed over all distinct words in the candidate. This sum is then divided by the total number of words in the candidate translation. In the above example, the modified unigram precision score would be 2/7.

In practice, using individual words as the unit of comparison is not optimal. Instead, BLEU computes the same modified precision metric using n-grams, usually $n = 4$. The unigram scores are found to account for the adequacy of the translation, how much information is retained. The longer n-gram scores account for the fluency of the translation, or to what extent it reads like "good English".

To produce a score for the whole corpus the modified precision scores for the segments are combined using the geometric mean multiplied by a brevity penalty to prevent very short candidates from receiving too high a score.

Let r be the total length of the reference corpus, and c the total length of the translation corpus. If $c \leq r$, the brevity penalty applies, defined to be $e^{(1-r/c)}$. In the case of multiple reference sentences, r is taken to be the sum of the lengths of the sentences whose lengths are closest to the lengths of the candidate sentences.

Attention model

Attention is a solution to the limitation of the encoder-decoder model; encoding the input sequence to a fixed length vector from which to decode each output time step. The encoder-decoder model has problems when trying to decode long sequences; one usually notice that the BLEU score drops substantially for longer sequences.

Instead of decoding the input sequence into a single fixed vector, the attention model develops a context vector c that is filtered specifically for each output time step. The attention model assigns attention weights $\alpha_{t,t'}$ to the input t' and the output t. The attention weights are used to specify which words are needed when to generate a word. The sum of the attention weights for each element in the sequence should be 1:

$$\sum_{t'} \alpha_{t,t'} = 1$$

The context vector c_t is defined by:

$$c_t = \sum_{t'} \alpha_{t,t'} h_{t'}$$

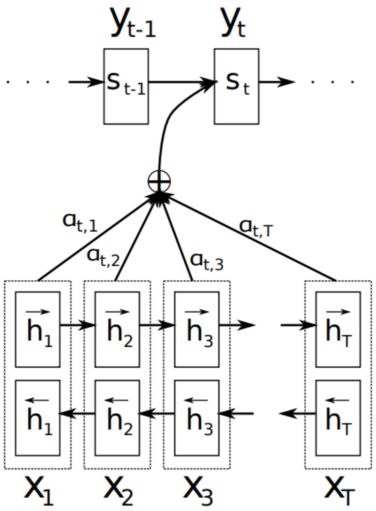
Where c_t is used as the input for the output recurrent network. The attention weights are defined by:

$$\alpha_{t,t'} = \frac{\exp(e_{t,t'})}{\sum_{t=1}^{T_x} \exp(e_{t,t'})}$$

$e_{t,t'}$ is computed with a small neural network using the hidden state from the input at time t' and the hidden state from the output at time $t-1$:

$$e_{t,t'} = f(W[h_{t'}, s_{t-1}] + b)$$

Below is an illustration of the attention model:



y_{t-1}
y_t
s_{t-1}
s_t
$a_{t,1}$
$a_{t,2}$
$a_{t,3}$
$a_{t,T}$
$\overrightarrow{h}_1$
$\overrightarrow{h}_2$
$\overrightarrow{h}_3$
$\overrightarrow{h}_T$
$\overleftarrow{h}_1$
$\overleftarrow{h}_2$
$\overleftarrow{h}_3$
$\overleftarrow{h}_T$
x_1
x_2
x_3
x_T