

Coursera Deep Learning Specialization Notes: A Review of Deep Learning Fundamentals

[Amir Masoud Sefidian](#)

Version 1.0, February 2023

Contents

1	Neural Networks and Deep Learning	5
1.1	Logistic regression as a neural network	5
1.2	Neural networks	5
1.3	Activation functions	6
1.4	Random initialization	8
2	Improving Deep Neural Networks	9
2.1	Training/Dev(Cross Validation (CV))/Test	9
2.2	Bias and Variance	9
2.3	Regularization	10
2.4	Dropout	10
2.5	Other regularization methods	12
2.6	Normalizing training sets	12
2.7	Vanishing / Exploding gradients	13
2.8	Weight initialization for deep networks	14
2.9	Numerical approximation of gradients	14
2.10	Gradient checking (Grad check)	14
2.11	Mini-batch gradient descent	15
2.12	Optimization algorithms	15
2.13	Gradient descent with momentum	16
2.14	RMSprop (Root Mean Square prop)	17
2.15	Adam (adaptive moment estimation) optimization	18
2.16	Learning rate decay	18
2.17	Local optima and saddle points	18
2.18	Hyperparameter tuning process	19
2.19	Batch normalization	20
2.20	Multi-class classification	22
3	Structuring Machine Learning Projects	24
3.1	Orthogonalization	24
3.2	Single number evaluation metric	24
3.3	Satisfying and Optimization metric	24
3.4	Training/dev/test sets Distributions	25
3.5	Comparing to human-level performance	25
3.6	Error analysis	27
3.7	Build your first system quickly, then iterate	28
3.8	Training and testing on different distributions	28
3.9	Transfer learning	31
3.10	Multi-task learning	31
3.11	End-to-end learning	32
4	Convolutional Neural Networks	34
4.1	Fundamentals	34
4.2	One layer of a convolutional network	36
4.3	Convolutional network	37
4.4	Pooling layers	38
4.5	Fully connected layers	38
4.6	Why convolutions?	39

4.7	Historical/Classic architectures	39
4.8	ResNet	41
4.9	Networks in Networks / 1x1 Convolution	42
4.10	Inception network (aka GoogleNet)	42
4.11	Using Transfer Learning	43
4.12	Data Augmentation	44
4.13	State of computer vision	45
4.14	Tips for doing well on benchmarks/winning competitions	45
4.15	Classification with localization	46
4.16	Landmark detection	47
4.17	Object detection	47
4.18	Convolutional implementation of sliding windows	48
4.19	YOLO algorithm	50
4.20	Face recognition	54
4.21	Face Verification as a binary classification problem	55
4.22	Neural style transfer	56
5	Sequence Models	59
5.1	Applications	59
5.2	Notation	59
5.3	Recurrent neural networks (RNNs)	60
5.4	Language Model and Sequence generation	62
5.5	Gated Recurrent Unit (GRU)	65
5.6	Long Short-Term Memory (LSTM)	67
5.7	Bidirectional RNNs (BRNN)	69
5.8	Deep RNNs	69
5.9	NLP and Word embeddings	70
5.10	Word2Vec (CBOW and Skip-Gram)	74
5.11	Negative Sampling (Skip-Gram revised)	76
5.12	GloVe (global vectors for word representation)	78
5.13	Interpretability of word embeddings	78
5.14	Sentiment classification	79
5.15	Debiasing word embeddings	80
5.16	Basic Sequence Models	81
5.17	Beam search algorithm	83
5.18	Bleu Score	85
5.19	Attention Model	86
5.20	Speech recognition	89

Preface

A couple of years ago, I completed [Deep Learning Specialization](#) taught by AI pioneer Andrew Ng. I found this series of courses immensely helpful in my learning journey of deep learning. After years, I decided to prepare this document to share some of the notes which highlight the key concepts I learned in this specialization.

The first course, Neural Networks and Deep Learning, teaches you the theory behind deep learning and applies this technology in the real world, such as creating and training a simple neural network and understanding the key parameters of deep learning.

The second course, Improving Deep Neural Networks, imparts the knowledge of how to optimize your model performance by applying many algorithms and techniques. For instance, how to tune the learning rate, the number of layers, and the number of neurons in each layer. Then regularization techniques like dropout and Batch Normalization are covered, to end with an optimization section that discusses stochastic gradient descent, momentum, RMS Prop, and Adam optimization algorithms.

The third course, Structuring Machine Learning Projects, is all about how to build ML projects, get results quickly and iterate to improve these results. It gives delightful insights into how to diagnose the outcomes of our models so that we can see where the performance problems such as small training sets, different distributions of train and test sets, and over-fitting, along with their solutions.

In the fourth course, Convolutional Neural Networks (CNNs), you will learn how to build convolutional neural networks and apply them to image data. When it comes to computer vision, CNNs are the bee knees. CNNs are what have given rise to incredible improvements in facial recognition, classifying X-ray reports, and self-driving car systems.

In the fifth course, you will become familiar with Sequence Models and their exciting applications such as speech recognition, music synthesis, chatbots, machine translation, Natural Language Processing (NLP), and more. You will be empowered to build and train Recurrent Neural Networks (RNNs) and commonly-used variants such as GRUs and LSTMs. Moreover, you will apply RNNs to Character-level Language Modeling, gain experience with NLP and Word Embedding, and use HuggingFace tokenizers and Transformer Models to solve different NLP tasks such as Named Entity Recognition and Question Answering.

Notes are based on lecture videos and supplementary material provided and my own understanding of the topics. The content of this document is mainly adapted from this [GitHub](#) repository. I have included a multitude of explanations, illustrations, and visualizations to make some complex concepts easier to grasp for readers. This document could be a good reference for Machine Learning Engineers, Deep Learning Engineers, and Data Scientists to refresh their knowledge on the fundamentals of deep learning. Please don't hesitate to contact me via my website ([Sefidian Academy](#)) if you have any questions.

Happy Learning!
Amir Masoud Sefidian

1 Neural Networks and Deep Learning

1.1 Logistic regression as a neural network

- A small-scale example of a neural network with a single neuron (and still useful as a classifier).
- n : number of features, m : number of samples
- Inputs: $X = [x^{(1)} \ x^{(2)} \ \dots \ x^{(m)}]_{n \times m}$, $Y = [y^{(1)} \ y^{(2)} \ \dots \ y^{(m)}]_{1 \times m}$
- Logistic Loss / Binary Cross Entropy Loss:
 $L(y, a) = -[y \log a + (1 - y) \log(1 - a)]$

- Vectorized Optimization:
 $Z = W^T X + b$

$$A = \sigma(Z) = \frac{1}{1 + e^{-Z}}$$

$$dZ = A - Y$$

$$dW = \frac{1}{m} X dZ^T$$

$$db = \frac{1}{m} \sum dZ$$

$$W \leftarrow W - \alpha \cdot dW, b \leftarrow b - \alpha \cdot db$$

- **Note:** d stands for “derivative” where there is only a single variable. ∂ stands for “partial derivative” where there are multiple variables.

1.2 Neural networks

- Typically the input layer is not counted as a layer when counting the layers of a neural network, only the hidden layers and the output layer are considered. The output layer is NOT a hidden layer.
- $A^{[l]}$ is the activation for layer l , $W^{[l]}$ is the weights matrix, and $b^{[l]}$ is the bias term. $g(z)$ is the chosen activation function.
- The learning rate α scales the size of the gradient descent steps and therefore determines how fast it converges (though values too high can actually make it not converge).
- Regarding notation $a_j^{[l](i)\{k\}}$ means the activation (could be any other parameter) on the j^{th} unit/neuron of the l^{th} network layer, for the i^{th} example/data point of the k^{th} minibatch.

- Notations:

L : # of layers

$n^{[l]}$: # of units in layer l

$b^{[l]}$: biases in layer l (size: $(n^{[l]}, m)$)

$Z^{[l]}$: outputs in layer l (size: $(n^{[l]}, m)$)

$A^{[l]}$: activations in layer l (size: $(n^{[l]}, m)$)

$W^{[l]}$: Weights connecting layer $l - 1$ to l (size: $(n^{[l]}, n^{[l-1]})$)

$n^{[0]} = n_X$: # of input features

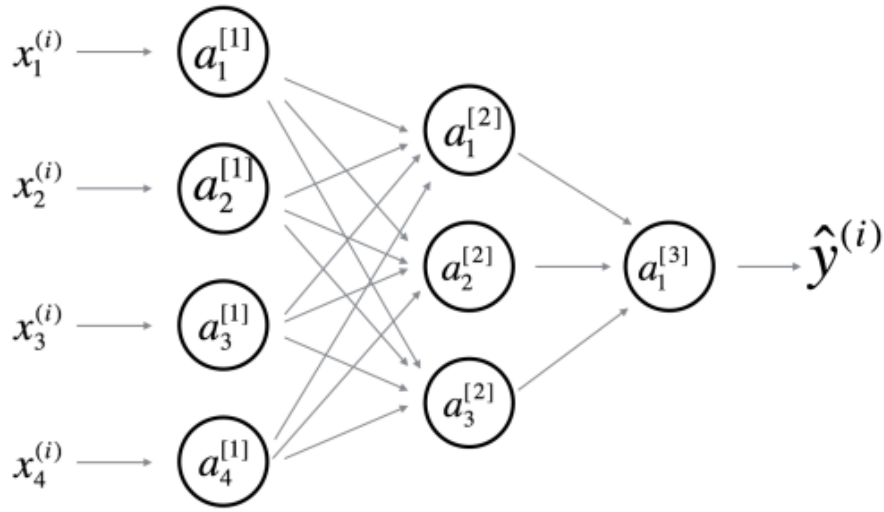


Figure 1

Forward Propagation for layer l (vectorized):

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$

$$A^{[l]} = g^{[l]}(Z^{[l]})$$

Backward Propagation for layer l (vectorized):

$$dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]})$$

$$dW^{[l]} = \frac{1}{m} dZ^{[l]} (A^{[l-1]})^\top$$

$$db^{[l]} = \frac{1}{m} \text{np.sum}(dZ^{[l]}, \text{axis} = 1, \text{keepdims} = \text{True})$$

$$dA^{[l-1]} = (W^{[l]})^\top dZ^{[l]}$$

Note: $*$ is the element-wise multiplication

Parameter Update for layer l (vectorized):

$$W^{[l]} \leftarrow W^{[l]} - \alpha \cdot dW^{[l]}, \text{ where } \alpha \text{ is the learning rate}$$

$$b^{[l]} \leftarrow b^{[l]} - \alpha \cdot db^{[l]}, \text{ where } \alpha \text{ is the learning rate}$$

1.3 Activation functions

- Sigmoid (or logistic function) - usually used for the output layer only (because it outputs either 1 or 0), and not for the hidden layers because its derivative can be close to 1 for values of z further away from the origin.

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \sigma'(z) = \sigma(z)(1 - \sigma(z))$$

- \tanh is better than sigmoid for hidden layers (though it is an offset sigmoid) only because it is defined between -1 and 1, which means that for the mean of zero, the \tanh will be close to 0 as well, which can be useful for computation.

$$\tanh(z) = \frac{2}{1 + e^{-2z}} - 1, \tanh'(z) = 1 - \tanh(z)^2$$

- Rectified linear unit (ReLU) - $\text{ReLU}(z) = \max(0, z)$ - is the de facto standard for linear units nowadays, its derivative is easy to calculate (not defined for $z = 0$, but we can work around that with a convention on what 0 means), and does not suffer from slow convergence of gradient descent due to the derivative being close to 0.

$$\text{ReLU}(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}, \text{ReLU}'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

- Leaky ReLU - $\max(0.01 * z, z)$ - Similar to the ReLU, but instead of being zero for values of z lower than 0, it actually has a small positive slope in that section, so its derivative is not 0, making it easier for the optimizer (e.g. gradient descent) to converge. In reality normal ReLU are still the standard though.

$$\text{LReLU}(z) = \begin{cases} 0.01z & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}, \text{LReLU}'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

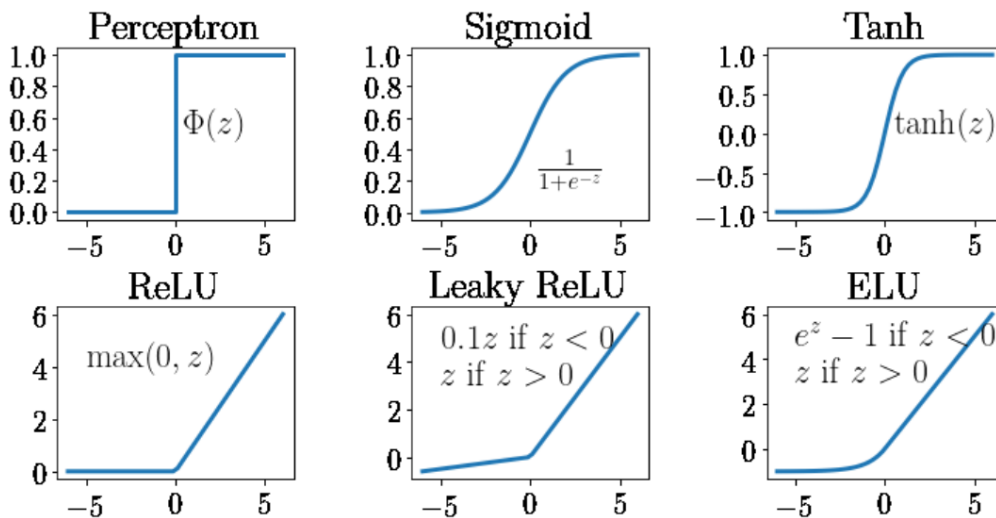


Figure 2: Activation Functions

Why do we need some sort of activation function anyway?

- If we had none, the activation would be linear, and all layers would have linear activation functions, resulting in the NN activation being itself a linear activation function, which negates the usefulness of the hidden layers. There must always be a hidden layer.

1.4 Random initialization

- Two neurons are considered symmetric if they are doing the exact same computation.
- We avoid that by using random initialization rather than zero initialization, otherwise all neurons/units will keep on being symmetric throughout all backprop iterations.
- b (bias) doesn't require random initialization like the weights W .
- W should be initialized to small random values (multiply by 0.01 for example), to allow faster convergence of gradient descent with sigmoid or *tanh* activation functions.

2 Improving Deep Neural Networks

2.1 Training/Dev(Cross Validation (CV))/Test

- Training set is used to train the model's parameters.
- Dev(cross-validation) is used to train the model's hyperparameters and check the model's performance while in development.
- Test set is an unbiased set of data that was never seen by the model. Some teams may not use this and only a dev set instead.
- Traditionally with small data the split between these two sets would be either 70/30% (train/test(or dev)) or 60/20/20% (train/dev/test). With big data that can be something like 98/1/1%.
- Ensure that dev and test sets are from the same distribution.
- If the training data is from a different distribution than the test set, then it is recommended that the dev set should belong to the same distribution as the test set.

2.2 Bias and Variance

- High bias generally means underfitting - high error on the training set.
- High variance generally means overfitting - high error on the test set.
- Base error is the reference (e.g. human) error for the same task, and should be compared to the model to determine high variance or high bias.
- There used to be a tradeoff between these two but that is not so discussed in the scope of deep learning, because we can always increase the network or/and add more data.
- High bias and high variance can occur at the same time if the model underfits some parts of the model and overfits other parts.

Solutions for high bias:

- Bigger network (KEY) - does not cause high variance (with good regularization)
- Train longer
- Change NN architecture
- Hyperparameter search
- Increase the number of useful features

Solutions for high variance:

- More training data (KEY) - does not cause high bias
- Regularization
- Reduce model complexity
- Dropout
- Early Stopping
- Data Augmentation
- Batch Normalization

2.3 Regularization

- **Regularization** is also called **weight decay** because it causes weights to be smaller for higher values of lambda (the regularization parameter). It reduces **high variance**
- There is L2 and L1 regularization, L2 uses the squared of the weights, L1 uses only the norm and has the “advantage” of making the weights matrix sparse, though L2 is the most used in practice.
- There is usually no regularization of the bias term because it is just a constant.

Regularization in Logistic Regression:

L1 norm:

$$J(W, b) = \frac{1}{m} \sum_{i=1}^m L(y^{(i)}, \hat{y}^{(i)}) + \frac{\lambda}{2m} \|W\|_2^2$$

L2 norm:

$$J(W, b) = \frac{1}{m} \sum_{i=1}^m L(y^{(i)}, \hat{y}^{(i)}) + \frac{\lambda}{2m} \|W\|_1$$

Regularization in Neural Networks:

$$J(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^m L(y^{(i)}, \hat{y}^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^L \|W^{[l]}\|_F^2,$$

where $\|W^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (W_{ij}^{[l]})^2$ (Frobenius norm)

Gradient Descent update:

$$W^{[l]} \leftarrow (1 - \alpha \frac{\lambda}{m}) W^{[l]} - \alpha \cdot (\text{Backprop Term})$$

- **Intuitions:** Why does it help with reducing variance problems?
 - When $\lambda \rightarrow \infty$, it set the weight matrices $W^{[l]}$ to be reasonably close to zero. As a result, the neural network becomes a much smaller neural network. See Figure (3).

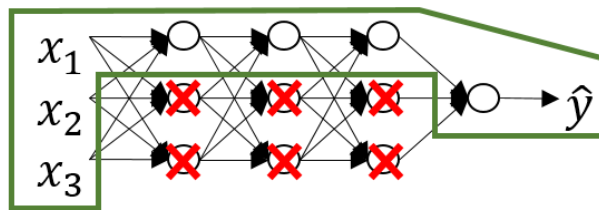


Figure 3: Regularization intuition.

- If the regularization becomes very large, the parameters $W^{[l]} \approx 0$, so Z will be relatively small. Thus, the activation function if is \tanh , say, will be relatively linear when $Z \rightarrow 0$. Thus, the whole neural network will be computing something not too far from a big linear function which is therefore a pretty simple function rather than a very complex highly non-linear function. See Figure (4).

2.4 Dropout

- **Dropout** regularization consists of training the NN with a number of neurons “switched off” at every training iteration (though not during testing). It has a similar effect to regularization,

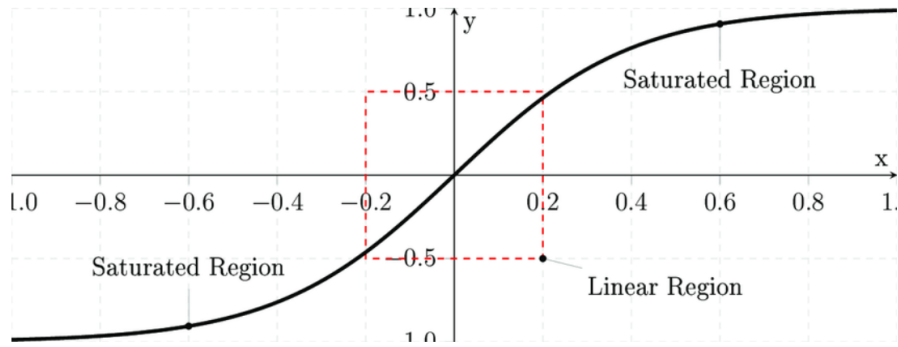


Figure 4: Regularization intuition.

and it is possible to have different percentages of dropped units/neurons for each layer, making it more flexible. The same units/neurons are dropped in both forward and backward steps. The cost function with dropout does not necessarily decrease continuously as we usually see for gradient descent.

- **Inverted dropout** is the most common type of dropout, and it consists of scaling activations by dividing with the activation matrix with **keep_prob** (the probability of keeping units), for each layer.
- Steps:

Listing 1: Inverted Dropout

```
keep_prob = 0.8
d3 = np.random.rand(a3.shape[0] a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3)
#ensures that the expected value of a3 remains the same
a3 /= keep_prob
```

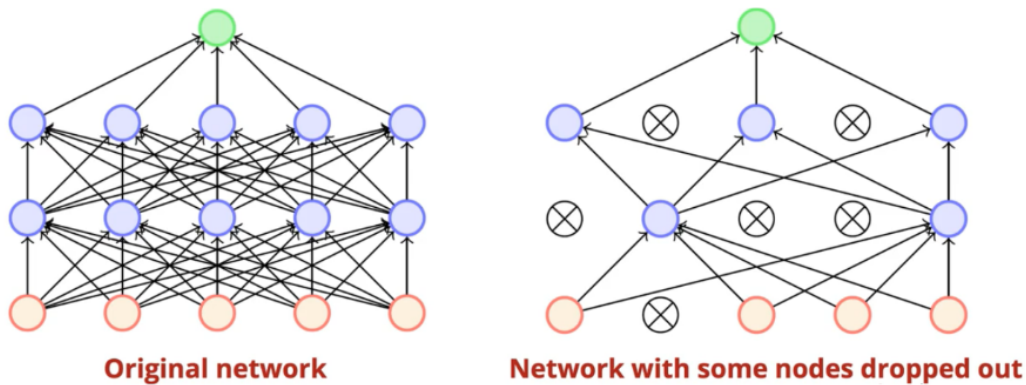


Figure 5: Dropout

- **Intuitions:**

- Dropout randomly knocks out units in the network. Hence, it is as if on every iteration we are working with a smaller neural network, and so using a smaller neural network seems like it should have a regularizing effect.

- Let's take a look from the perspective of a single unit. This unit takes some inputs and generates some meaningful output. Now with dropout, the inputs can get randomly eliminated. Therefore, it can't rely on any one feature because any one feature could go away at random or any one of its own inputs could go away at random. The weights, we are reluctant to put too much weight on any one input because it can go away. Thus, this unit will be more motivated to *spread out* this way and give a little bit of weight to each of inputs to this unit. And spreading all the weights will have the effect of shrinking the squared norm of the weights.
- One big downside of dropout is that the cost function J is no longer well-defined.

2.5 Other regularization methods

- **Early stopping** consists of stopping training when the error of the network is the lowest for the dev(cross-validation) dataset, even if it can still be decreased for the training set.

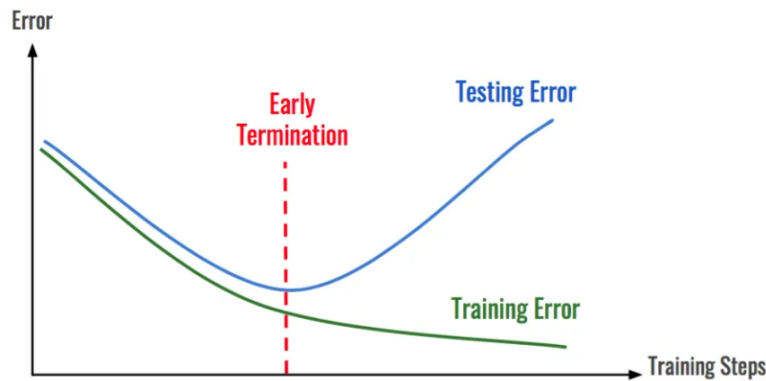


Figure 6: Early stopping

- **Data augmentation** is a technique of artificially increasing the amount of data by generating new data points from existing data. This is helpful when we are given a dataset with very few data samples. In the case of Deep Learning, this situation is bad as the model tends to overfit when we train it on a limited number of data samples. This includes adding minor alterations to data or using machine learning models to generate new data points in the latent space of original data to amplify the dataset.
- Note: **Orthogonalization** is the separation of the cost optimization step (e.g. gradient descent) from steps taken for not overfitting the model (e.g. regularization), in other words, optimizing model's parameters vs. optimizing model hyperparameters.

2.6 Normalizing training sets

- $\mu = \frac{1}{m} \sum_{i=1}^m x^{(i)}$
 $x \leftarrow x - \mu$
 $\sigma^2 = \frac{1}{m} \sum_{i=1}^m (x^{(i)})^2$
 $x \leftarrow x / \sigma^2$

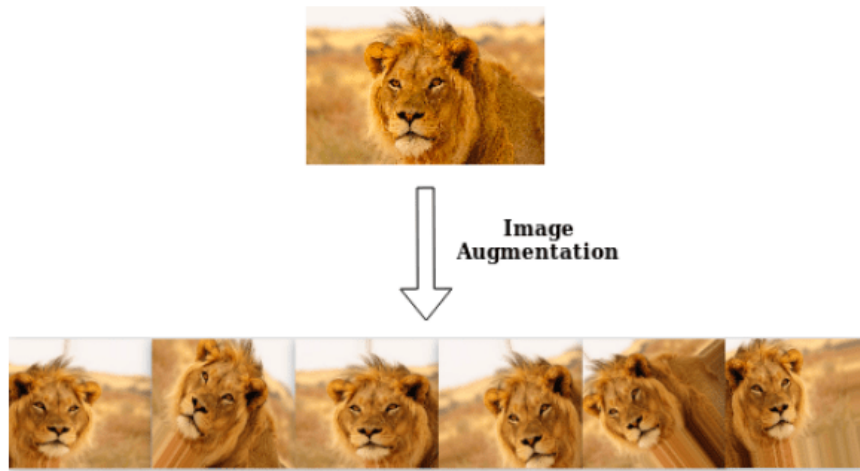


Figure 7: Data augmentation

- The mean and variance obtained in the training set should be used to scale the test set as well, (we don't want to scale the training set differently).
- Allows using higher learning rates and faster convergence for gradient descent.
- If features are on very different scales, say the feature x_1 ranges from 1 to 1000, and the feature x_2 ranges from 0 to 1, then the ratio or the range of values for the parameters w_1 and w_2 will end up taking on very different values. Then the cost function can be very elongated. When normalizing the features, the cost function will be more symmetric. When contours are spherical, we can take much larger steps with gradient descent rather than needing to oscillate. See Figure (8).

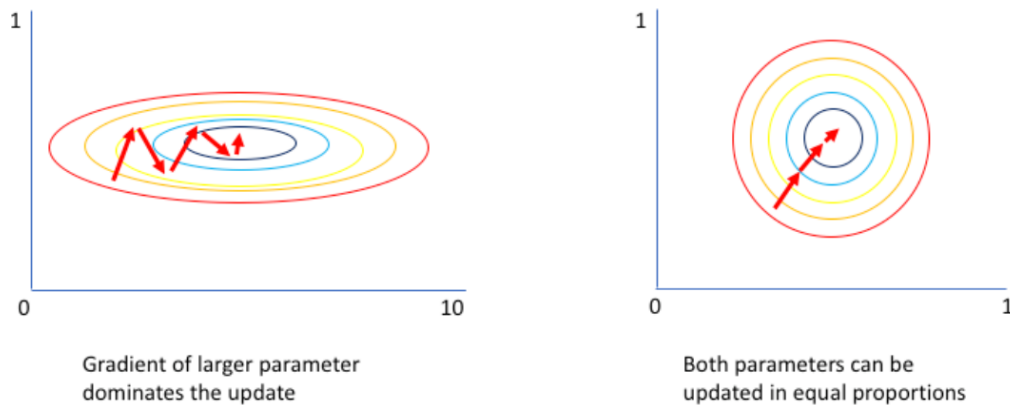


Figure 8: Normalization effect

2.7 Vanishing / Exploding gradients

- In very deep networks (depending on the activation function) weights greater than 1 can make activations exponentially larger depending on the number of layers with such weights, whereas weights smaller than 0 can make activations exponentially smaller, depending on the number of

layers with such small weights (think in terms of a very deep network with linear activations as intuitive example).

- The above is also applicable for the gradients (not just the activation/output), on the opposite direction (backward propagation), thus gradients can either explode (causing numerical instability) or become very small (with the consequence of lower layers not being updated as well as numerical instability).

2.8 Weight initialization for deep networks

- Partial solution to Vanishing/Exploding gradients
- Make the randomly initialized weights to have a variance of $\frac{1}{n^{[l-1]}}$ for *tanh* and $\frac{2}{n^{[l-1]}}$ for ReLU. $n^{[l-1]}$ is the number of inputs of layer l .

- For *tanh*:

$\text{np.random.randn()} * \sqrt{\frac{1}{n^{[l-1]}}}$ (Xavier initialization) or $\text{np.random.randn()} * \sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

For ReLU:

$\text{np.random.randn()} * \sqrt{\frac{2}{n^{[l-1]}}}$

2.9 Numerical approximation of gradients

- Two-sided difference approximates the derivative of a function with $O(\epsilon^2)$ error therefore much better than the one-sided difference that is $O(\epsilon)$ - and for ϵ smaller than 0 that that means that the two-sided difference has a much smaller error.

2.10 Gradient checking (Grad check)

- Used to check the correctness of the implementation (bugs). Only to be used during debugging, not during training (it's slow).
- Take $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}$ and concatenate and reshape them into a vector θ .
- Take $dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}$ and concatenate and reshape them into a vector $d\theta$.
- With Θ being a vector of parameters θ_i , and $d\theta[i] = \frac{\partial J}{\partial \theta_i}$, compare the “approx” derivative with the real $d\theta$ with the check:

$$\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$$

, where

$$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \dots, \theta_i + \epsilon, \dots) - J(\theta_1, \theta_2, \dots, \theta_i - \epsilon, \dots)}{2\epsilon}$$

- Note that $\|\cdot\|_2$ denotes the squared root of the sum of the squared differences (that is, the norm of the vector).
- If the result is near 10^{-7} it is great, if it is 10^{-5} , then suspect something in the formula, if 10^{-3} , something is really wrong.
- Look for what $d\theta[i]$ (what component) has the highest difference, to pinpoint the cause of the bug.
- Include the regularization term in the cost function when performing the **grad check**.

- Doesn't work with dropout (turn it off during grad check).
- Run at initialization and then again after some training.

2.11 Mini-batch gradient descent

- Applicable for large datasets (single batch).
- Running each iteration of gradient descent on smaller batches of the full dataset. May take too long per iteration.
- The cost function trends downward but not monolithically in this case.
- If batch size = m then it is just batch gradient descent (run for all the examples at once). (use this for $m \leq 2000$).
- If batch size = 1 then it is **Stochastic Gradient Descent** (every example is a mini-batch). Gradient descent doesn't completely converge. Loses speedup from vectorization.
- Ideal scenario is in between the two above (may not exactly converge, but we can reduce the learning rate).
- Typical minibatch sizes are powers of two (64, 128, 256, 512) - to ensure they fit in CPU/GPU memory.

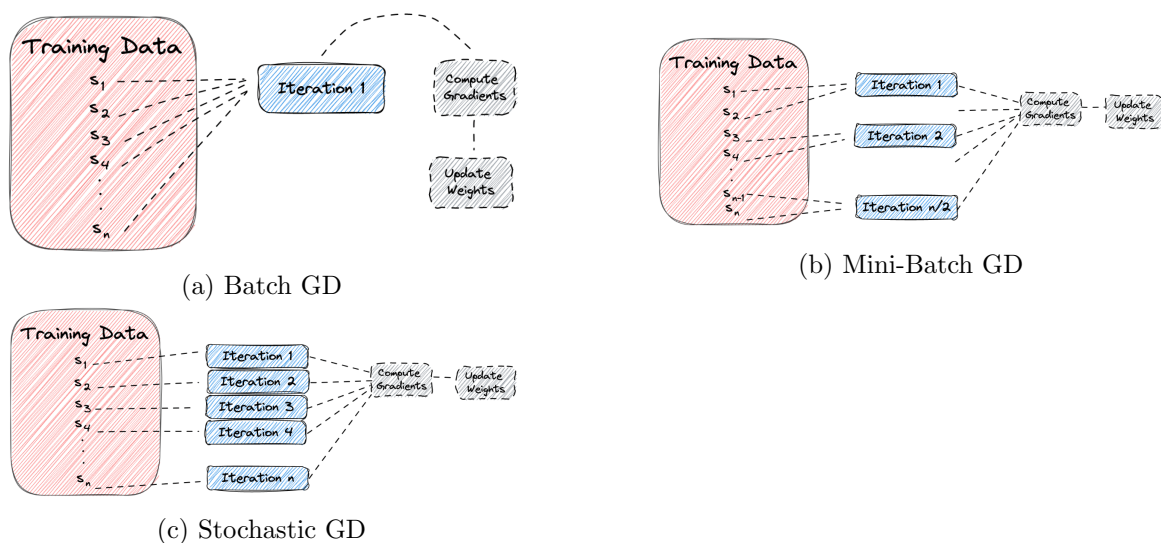


Figure 9: Batch Gradient Descent vs. Mini-Batch Gradient Descent vs. Stochastic Gradient Descent

2.12 Optimization algorithms

Exponentially Weighted Moving Average (EWMA)

- $V_t = \beta V_{t-1} + (1 - \beta)\theta_t$. V_t is average over past $1/(1 - \beta)$ data points θ s.
- All the coefficients add up to 1.

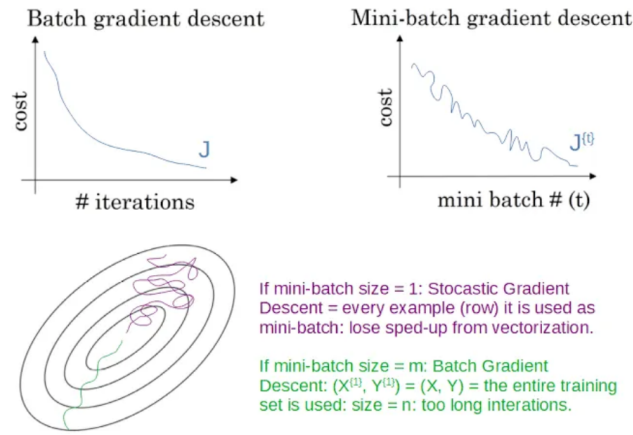


Figure 10: Batch GD cost curve

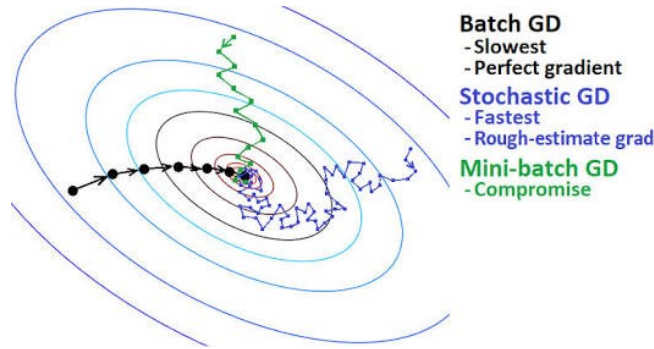


Figure 11: Convergence of different GD methods

- When $\beta = 0.9$ it takes a delay of approx 10 (think 10 days for daily time-series data) for the contribution of a point to reduce to $1/3$. The general rule (in which ϵ is 0.1 and $\beta = 1 - \epsilon$ to meet this example) is:

$$(1 - \epsilon)^{\frac{1}{\epsilon}} = \frac{1}{e}$$

- To correct the bias of the first few terms (compared to 0 initialization), the following formula can be used:

$$V_t^{\text{corrected}} = \frac{V_t}{1 - \beta^t}$$

2.13 Gradient descent with momentum

- Converges faster than the standard gradient descent algorithm.
- The basic idea is to compute an exponentially weighted average of gradients, and then use that gradient to update the weights.
- Uses exponential weighted moving averages to smooth out the derivatives dW and db , when updating W and b in each iteration. For example for dW (and similarly for db):

$$V_{dW} = \beta V_{dW} + (1 - \beta) dW$$

$$W = W - \alpha V_{dW}$$

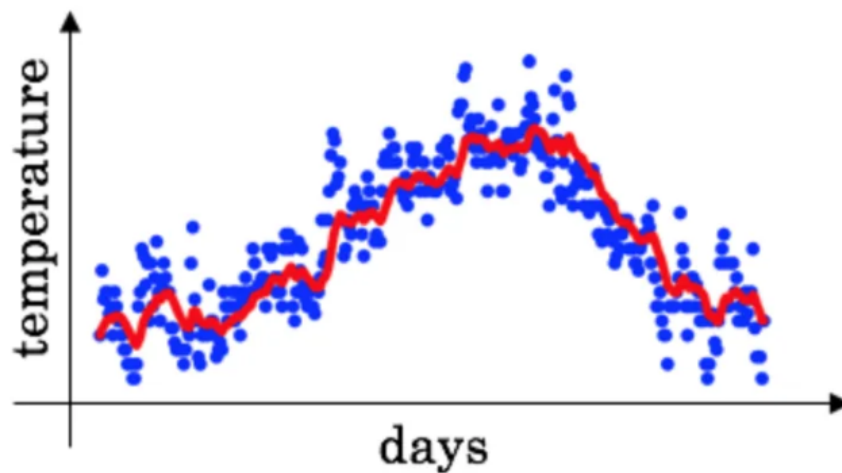


Figure 12: Exponentially Weighted Moving Average

- Sometimes a simplified version is used that factors the $(1 - \beta)$ term into the learning rate (that must be adjusted) instead of it being explicit, therefore:

$$V_{dW} = \beta V_{dW} + dW$$

$$\alpha_{adjusted} = \alpha(1 - \beta)$$

- β is most commonly 0.9 (pretty robust value)
- Bias correction is not usually used for gradient descent.

2.14 RMSprop (Root Mean Square prop)

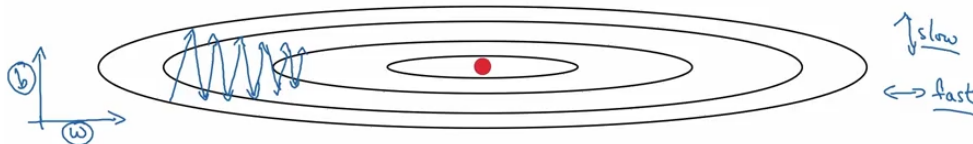


Figure 13: RMSprop.

- Update W and b on each iteration with dW or db divided by the root mean square of the exponential moving average of dW or db (the square is element-wise):

$$S_{dW} = \beta S_{dW} + (1 - \beta) dW^2$$

$$W = W - \alpha \frac{dW}{\sqrt{S_{dW} + \epsilon}}$$

- Implementations add a small ϵ to the denominator to avoid divisions by 0.
- The intuition is to have smaller/slower updates of db (the derivative of the bias term or vertical direction) and higher/faster updates of dW (the derivative of the weights or horizontal direction), to improve convergence speed. dW is a large matrix, therefore RMS of dW is much larger than RMS of db .
- Allows using a higher learning rate and faster convergence.

2.15 Adam (adaptive moment estimation) optimization

- Combine intuitions of Momentum + RMSprop together, both with bias correction!

$$V_{dW} = \beta_1 V_{dW} + (1 - \beta_1) dW, \quad V_{db} = \beta_1 V_{dbW} + (1 - \beta_1) db$$

$$S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) dW^2, \quad S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2$$

$$V_{dW}^{corrected} = \frac{V_{dW}}{1 - \beta_1^t}, \quad V_{db}^{corrected} = \frac{V_{db}}{1 - \beta_1^t}$$

$$S_{dW}^{corrected} = \frac{S_{dW}}{1 - \beta_2^t}, \quad S_{db}^{corrected} = \frac{S_{db}}{1 - \beta_2^t}$$

$$W = W - \alpha \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected} + \epsilon}}$$

$$b = b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected} + \epsilon}}$$

- There are two β parameters:
 - β_1 is the momentum parameter and is usually 0.9
 - β_2 is the RMSprop parameter and is usually 0.999
 - ϵ is usually 10^{-8}

2.16 Learning rate decay

- Have a slower learning rate as gradient descent approaches convergence.

$$\alpha = \frac{1}{1 + \text{decay_rate} \times \text{epoch_num}} \times \alpha_0$$

- Alternatives:
 - Exponential decay: $\alpha = 0.95^{\text{epoch_num}} \times \alpha_0$
 - Or: $\alpha = \frac{k}{\sqrt{\text{epoch_num}}} \times \alpha_0$
 - Discrete staircase, manual decay, etc.

2.17 Local optima and saddle points

- Most points with zero gradients are saddle points, not local optima!
 - Plateau's in saddle points slow down learning.
 - Local optima are pretty rare in comparison/unlikely to get stuck in them.

2.18 Hyperparameter tuning process

- Order of importance of hyperparameters for tuning:
 1. learning rate (α)
 2. momentum term ($\beta : 0.9$)
 3. mini-batch size
 4. number of hidden units
 5. number of layers
 6. learning rate decay
 7. $\beta_1, \beta_2, \epsilon$
- Choose the hyperparameter value combinations at random, (don't use a grid) because of the high number of hyperparameters nowadays (cube/hyperdimensional space), doesn't compensate test all values/combinations.

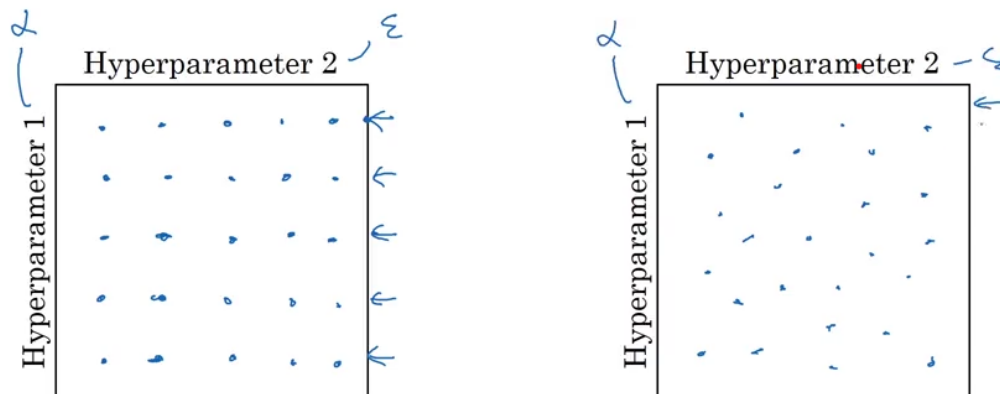


Figure 14: Grid vs. Random search.

- Coarse to fine-tuning - first coarse changes of the hyperparameters, then fine-tune them.
- Using an appropriate scale for the hyperparameters.
 - One possibility is to sample values at random within an intended range.
 - Using log scales to sample parameter values to try (for example, applicable for the learning rate).
For α (sample between $10^a \dots 10^b$), uniformly sample from $r \in [a, b]$ ($[-4, 0]$ for example) and set $\alpha = 10^r$.
 - For exponentially weighted average hyperparameters $(\beta, \beta_1, \beta_2)$, uniformly sample from $r \in [a, b]$ ($[-3, -1]$ for example) and set $\beta = 1 - 10^r$.
- Two possible approaches for hyperparameter search:
 - Panda approach: Watch only one model and change the parameters gradually and check improvements. Requires less hardware which might not be the most efficient method.
 - Caviar approach: Run multiple models with different parameters in parallel, if you have the computing power for it.

Coarse to fine

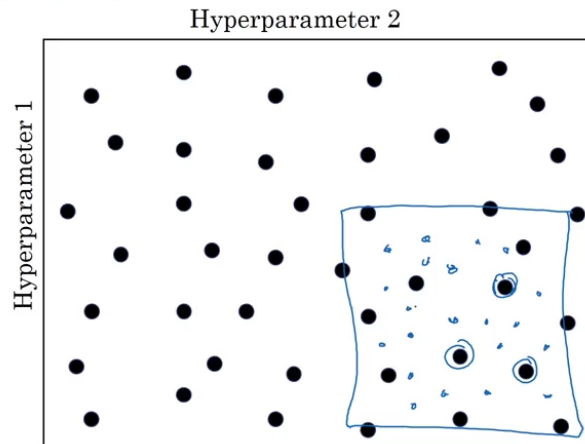


Figure 15: Coarse to fine search.

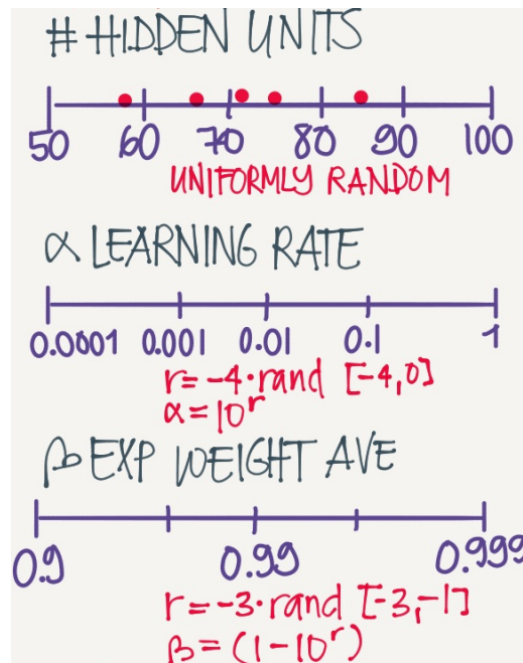


Figure 16: Hyperparameter Search

2.19 Batch normalization

- Normalize not just the inputs but the activation inputs to the next layer, subtracting the mean and dividing by the variance (mean 0, variance 1).
- Normally Z is normalized, before the activation function, though some literature suggests normalizing after the activation function.
- New parameters γ (multiplied by Z) and β (added to Z after the multiplication) are introduced and learned in the forward propagation and backward propagation. This is to prevent all neurons from having activations with mean 0 and variance 1 which is not desirable. Given some

intermediate values $z^{(1)}, z^{(2)}, \dots, z^{(m)}$ for a layer in the network:

$$Z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}, \text{ where } \mu = \frac{1}{m} \sum_i z^{(i)}, \text{ and } \sigma^2 = \frac{1}{m} \sum_i (z^{(i)} - \mu)^2$$

$$\tilde{Z}^{(i)} = \gamma Z_{norm}^{(i)} + \beta$$

- The bias parameter “b” in the calculation of Z is no longer needed because the mean of Z is being subtracted, canceling out any effect from adding “b”. The new parameter β effectively becomes the new bias term.
- At test time there is no μ and σ^2 , so these are computed based on an exponentially weighted average of these two parameters obtained on different batches during training. **Why does**

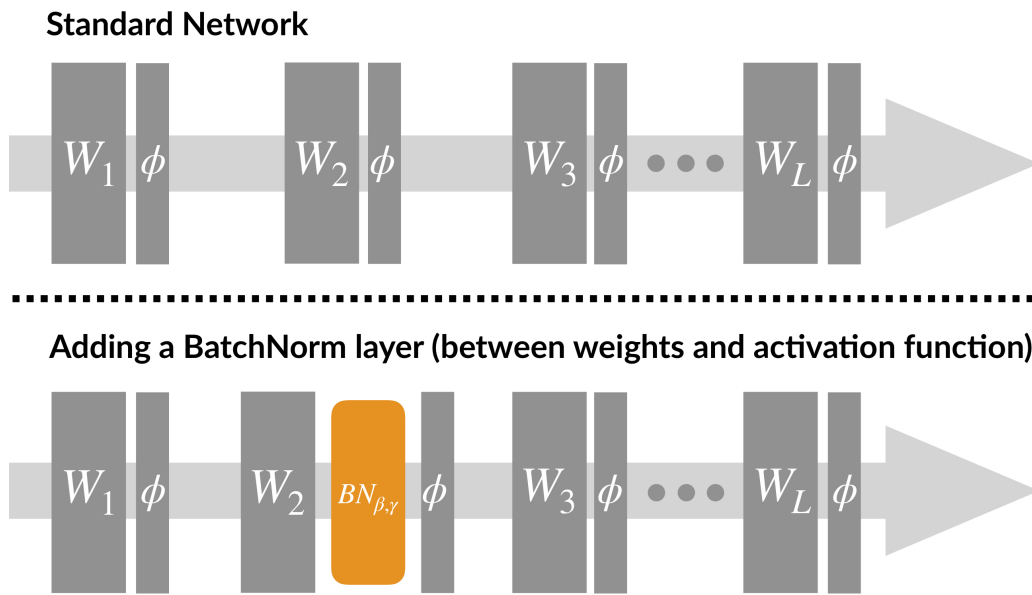


Figure 17: Batch Normalization.

Batch Normalization work?

- It makes weights of deeper layers more robust to changes in weights in earlier layers of network.
- **Covariate shift**: Data distribution changes with inputs (e.g. over time, with new batches, etc), you need to retrain your network normally.
- Batch normalization makes the process of learning easier by reducing the variability of the inputs presented to each layer (which now have similar variance and mean), therefore reducing the **covariate shift**, and that is especially important for deeper layers, where inputs could change significantly as a net effect of all the other changes in the network.
- It reduces the amount that the distribution of hidden unit values shifts around. And if it were to plot the distribution of hidden unit values, maybe this is technically we normalize Z, Batch norm ensures that no matter how the output of the previous layer changes, the mean and variance of a layer will remain the same. Therefore, the batch norm reduces the problem of the input values changing, it really causes these values to become more stable, so that the later layers of the neural network have more firm ground to stand on.

- It also has a slight regularization effect with mini-batch, due to the “noise” introduced by the calculations of mean and variance only for that mini-batch only rather than the entire dataset, which has an effect similar to that of dropout.

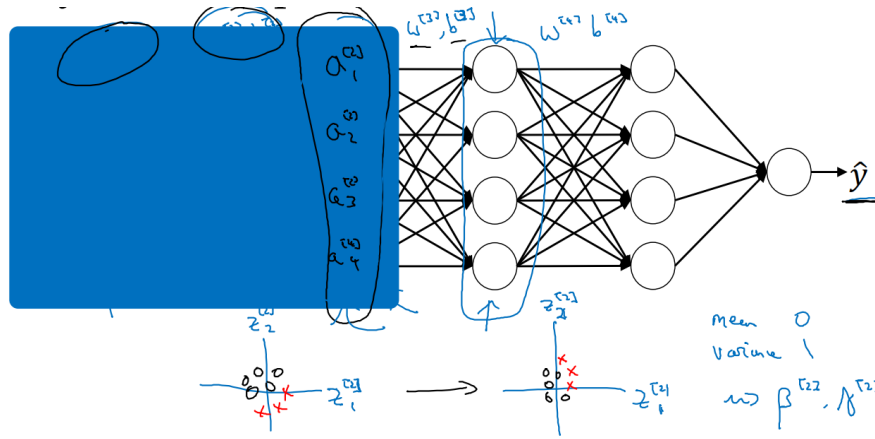


Figure 18: Covariate Shift

2.20 Multi-class classification

- # of neurons in the output layer = # of classes. The sum of all outputs must be 1, since these are probabilities of X being classified in any of these classes(likelihood).
- **Softmax** activation function is used as the output activation function:

$$t = e^{Z^{[L]}}$$

$$a_i^{[L]} = \frac{t_i}{\sum_{j=1}^K t_j}, \text{ where } K \text{ is the number of classes and output units}$$

SOFTMAX TRANSFORMS A VECTOR OF NUMBERS
INTO A VECTOR OF RELATIVE "PROBABILITIES"

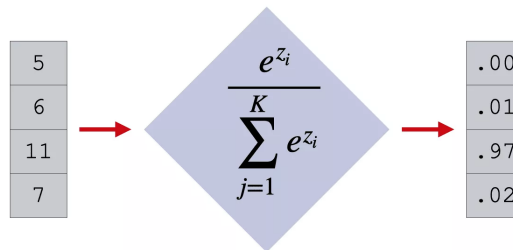


Figure 19: Softmax

- **Softmax** is in contrast with **Hardmax**, where the network’s output will be a binary vector with all 0s except for the position corresponding to the max value of $Z^{[L]}$.

- **Softmax** is the generalization of logistic regression to more than two classes. For two classes, it can be simplified/reduced to logistic regression.

Softmax Loss function:

$$\mathcal{L}(\hat{y}, y) = - \sum_{j=1}^C y_j \log(\hat{y}_j), \text{ assuming a binary vector } y$$

Softmax Cost function:

$$J = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)}), \text{ backprop: } \frac{\partial J}{\partial z} = \hat{y} - y$$

3 Structuring Machine Learning Projects

3.1 Orthogonalization

- Separating the hyperparameter tuning into different “dimensions”. “Using different knobs” for each objective/tuning/assumption in ML:
 - Fit training set well on the cost function. Knobs to improve: Try out bigger NN, Try out different NN architecture, Try out a different optimizer (Momentum, RMSprop, Adam, ...), and train longer (more iterations).
 - Fit dev set well on the cost function. Knobs to improve: Regularization, Bigger training set.
 - Fit test set well on the cost function. Knobs to improve: Bigger dev set.
 - Perform well on real-world data: Knobs to improve: Change dev set (different probability distribution), Change cost function.
- Orthogonalization states that a specific action should only affect one of the above steps, not several at the same point.
- A bad example of a “knob” is early stopping because it simultaneously affects the fit of the training set and the dev set, therefore is not a targeted “knob”.

3.2 Single number evaluation metric

There is usually a trade-off between:

- Precision: the percentage of images identified as cats that actually are cats:

$$P = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

- Recall: the percentage of cats correctly classified out of all cat images:

$$R = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

Combine the 2 metrics into a single number with the F_1 Score:

- Average of precision P and recall R :

$$F_1 = \frac{2}{\frac{1}{P} + \frac{1}{R}}, \text{ this is the harmonic mean of } P \text{ and } R$$

3.3 Satisfying and Optimization metric

- Satisfying metric: one that only needs to be *acceptable* value (e.g. running time must be $\leq 100\text{ms}$).
- Optimizing metric: one that we want to optimize (e.g. accuracy).

3.4 Training/dev/test sets Distributions

- Mismatched training and dev/test sets are often rooted in the nature of Deep Learning, which usually requires a lot of labeled data.
- Dev and Test sets must (or at least should) come from the same distribution. Dev and test set should reflect the data that you expect to get in the future and consider it important to do well on.
 - The traditional splits (70/30 or 60/20/20) are no longer useful if you have large datasets (e.g. 1 million records). In the case of deep learning with huge datasets, use a 98/1/1 split.
 - Not having a test set might be OK (sometimes there is even only a training set, though that is not recommended).
- Metrics that evaluate classifiers should reflect what is expected from a classifier, for example, measuring an algorithm just by classification error does not take into account the fact that some of the false positives that it classifies are actually porn images instead of cats, in which case, a revised (e.g. weighted version) the metric should be used instead.
 - Use orthogonalization, first place the target (the right metric), and only then try to improve the performance according to that metric.
- If doing well on your metric + dev/test set does not correspond to doing well on your application, change your metric and/or dev/test set.
- Do NOT run for too long with an appropriate dev set or applicable metric.

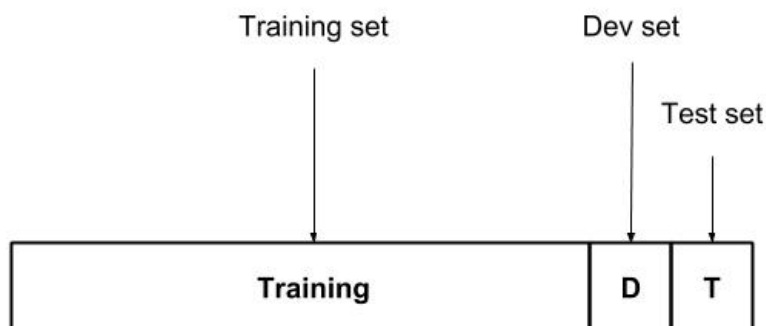


Figure 20: Train/Dev/Test Sets

3.5 Comparing to human-level performance

- Performance of an algorithm can surpass **human-level performance**, but even then it will be bound by the **Bayes optimal error** (or **Bayes error**), which is the best possible error value that can be achieved.
- There are two reasons why performance slows down when human-level performance is surpassed because humans are already close to “Bayes optimal error” in some cases.
- So long as ML is worse than humans, you can:

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

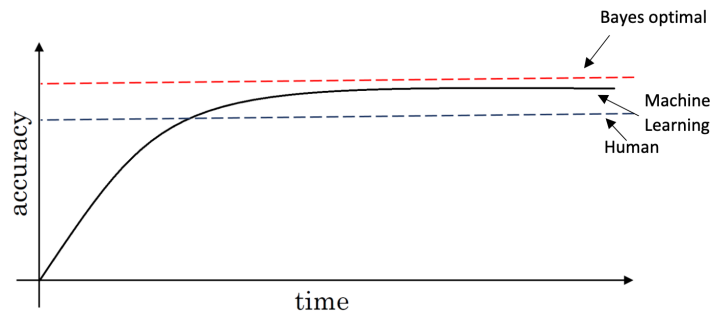


Figure 21: Human Performance

Avoidable bias

- Focus on improving bias: if human performance is much better than performance on the training set.
- Focus on improving variance: if human performance is close to performance on the training set, but performance on the dev set is significantly lower than training performance.
- The difference between training error and human error is **avoidable bias**.
- See Figure (23).

Scenario A: There is a 7% gap between the performance of the training set and the human-level error. It means that the algorithm isn't fitting well with the training set since the target is around 1%. To resolve the issue, we use bias reduction techniques such as training a bigger neural network or running the training set longer.

Scenario B: The training set is doing good since there is only a 0.5% difference with the human-level error. The difference between the training set and the human-level error is called avoidable bias. The focus here is to reduce the variance since the difference between the training error and the development error is 2%. To resolve the issue, we use variance reduction techniques such as regularization or collecting a bigger training set.

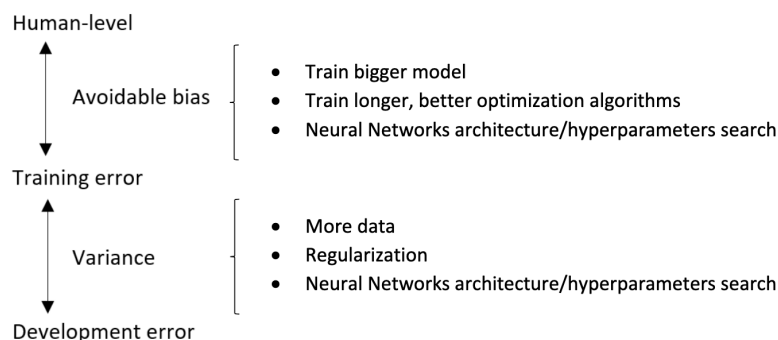


Figure 22: Avoidable Bias and Variance

	Classification error (%)	
	Scenario A	Scenario B
Humans	1	7.5
Training error	8	8
Development error	10	10

Figure 23

Understanding human-level performance

- **Human-level** error as a proxy for **Bayes error**: we use the best possible human-level error when trying to approximate Bayes error. This matters when the avoidable bias and the variance are already low.

Exceeding human-level performance

- If an algorithm surpasses human-level performance it is not clear what the avoidable bias is, and therefore we don't know if we should focus on improving bias or variance. Examples are:
 - Online advertising, Product recommendations, Logistics, and Loan approval: all structured data problems, NOT natural perception problems.
 - Speech recognition, some image recognition, medical ECG and cancer analysis, etc.

Improving your model performance

- Improve **avoidable bias**:
 - Train bigger model
 - Train longer/better optimization algorithms (momentum, RMSProp, Adam)
 - NN architecture/hyperparameters search (RNN, CNN, etc.)
- Improve variance:
 - More data
 - Regularization (L2, dropout, data augmentation)
 - NN architecture/hyperparameters search (RNN, CNN, etc.)

3.6 Error analysis

- Finding the most promising way to reduce error, by manually going through examples and finding out what the misclassified examples refer to (e.g. dogs, when trying to classify cats).
 - Depending on the percentage of these cases, choose the most prominent cases to focus on. If there are multiple examples, each team can focus on a different case. The conclusion of this process gives you an estimate of how worthwhile it might be to work on each of these different categories of errors. For example, clearly in Figure (24), a lot of the mistakes we made on blurry images, and quite a lot on were made on great cat images.
- Incorrectly labeled examples:

Error analysis

Image	Dog	Great Cat	Blurry	Incorrectly labeled	Comments
...					
98				✓	Labeler missed cat in background
99		✓			
100				✓	Drawing of a cat; Not a real cat.
% of total	8%	43%	61%	6%	

Figure 24: Error Analysis.

- Deep Learning is robust to random errors in the training set - not worth fixing manually. But we should be cautious about systematic errors.
- The frequency of mislabeled examples due to incorrect labeling should be calculated similarly to other mislabeled example analyses mentioned above.
- Correcting incorrect dev/test set examples
 - Whatever is done to the dev set we should also do to the test set so that they continue to come from the same distribution. The training set might come from a slightly different distribution.
 - Consider examining the (mislabeled) examples that the algorithm got right, not just those that it misclassified.

3.7 Build your first system quickly, then iterate

- Includes setting up dev/test set and metric.
- Allows you to identify areas where the algorithm is not performing well.
- Use Bias/Variance analysis and error analysis to prioritize the next steps.

3.8 Training and testing on different distributions

Suppose we have 200k examples from distribution A (high-resolution cat images from web pages) and only 10k images from another distribution B (lower-resolution cat images from a mobile app). It also happens that the images that we will need to classify are from distribution B.

- **OPTION 1 (BAD):** Mix A and B, randomized the images, and use a training/dev/test split of 205k/ 2.5k/2.5k images. This is bad because distribution B is the target, and there will be only a low number of the dev/test set in this case.
- **OPTION 2 (GOOD):** Use 200k from A as training examples and split the 10k from B into dev and test sets. This is much better because dev and test are used to set the “target” distribution (B in this case).

Variance (difference between error in the training and dev sets) might be higher (assuming OPTION 2) due to the different distributions of the two sets. A **training-dev** set with the same distribution as that of the *training set* (A) but *it is not used for training*. It can be created to determine if:

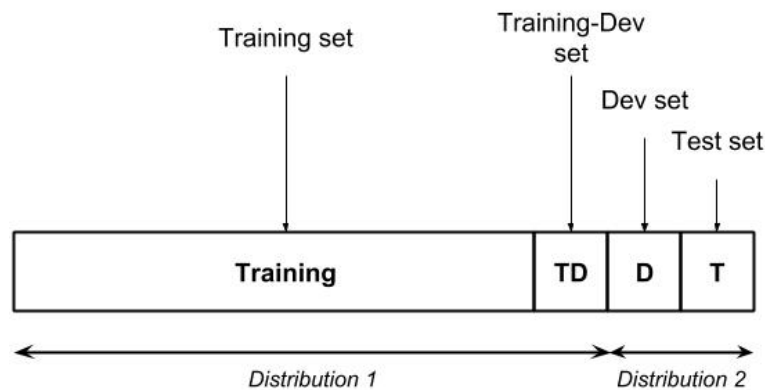


Figure 25: Training-Dev Set

- The error is due to overfitting the training set (variance) when the training-dev error is close to the dev error,
- or if it is due to the model not generalizing well to distribution B (**data mismatch** problem) if the training-dev error is close to the training error.

Note the 4 types of errors (Figure (26)):

- avoidable bias: $\text{human_level} - \text{training_set_error}$
- variance: $\text{training_set_error} - \text{training_dev_set_error}$
- data mismatch: $\text{training_dev_set_error} - \text{dev_error}$
- degree of overfitting to dev: $\text{dev_error} - \text{test_error}$

Bias and Variance Analysis Example (Figure (27)):

Scenario A:

If the development data and training set come from the *same distribution*, then there is a large variance problem and the algorithm is not generalizing well from the training set. However, when the training data and the development data come from *different distributions*, this conclusion cannot be drawn. There isn't necessarily a variance problem. The problem might be that the development set contains images that are more difficult to classify accurately. When the training set, development, and test sets distributions are different, two things change at the same time. 1) the algorithm trained in the training set but not in the development set. 2) The distribution of data in the development set is different. It's difficult to know which of these two changes produces this 9% increase in error between the training set and the development set.

Scenario B:

The error between the training set and the train-dev set is 8%. In this case, since the training set and train-dev set come from the same distribution, the only difference between them is the model sorted the data in the training and not in the training development. The model is not generalizing well to data from the same distribution that it hadn't seen before. Therefore, we have really a variance problem.

Scenario C: In this case, we have a mismatch data problem since the two datasets come from different distributions.

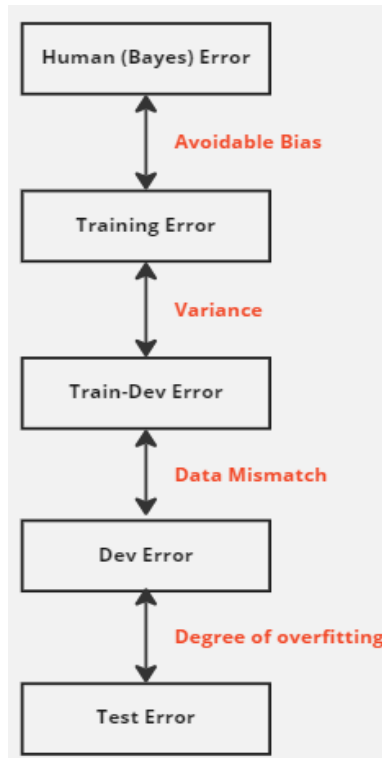


Figure 26: Types of Error.

	Classification error (%)					
	Scenario A	Scenario B	Scenario C	Scenario D	Scenario E	Scenario F
Human (proxy for Bayes error)	0	0	0	0	0	4
Training error	1	1	1	10	10	7
Training-development error	-	9	1.5	11	11	10
Development error	10	10	10	12	20	6
Test error	-	-	-	-	-	6

Figure 27: Bias and Variance Analysis Example

Scenario D:

In this case, the avoidable bias is high since the difference between Bayes error and training error is 10%.

Scenario E:

In this case, there are two problems. The first one is that the avoidable bias is high since the difference between Bayes error and training error is 10% and the second one is a data mismatched problem.

Scenario F:

Development should never be done on the test set. However, the difference between the development set and the test set gives the degree of overfitting to the development set.

Addressing data mismatch

- Carry out manual error analysis and try to understand the differences between the training and dev/test sets.
- Make training data more similar (**artificial data synthesis**), or collect more data similar to

dev/test sets. For example, if you find that car noise in the background is a major source of error, one thing you could do is simulate noisy in-car data. Make sure that artificial data synthesis is varied enough to avoid overfitting to one dimension or a sub-set domain of the synthesized data.

3.9 Transfer learning

Includes re-using a previously trained model (e.g. in general image recognition if your task is to make radiology diagnosis). This usually requires replacing the last layer of the classifier with a new layer (or layers) with weights and biases randomly initialized. Then there are two options:

- Retrain only the last layer if we don't have enough data in the new (e.g. radiology) dataset.
- Or, if there is sufficient data, retrain the entire model, in which case:
 - The initial phase of training (e.g. on image recognition) is called **pre-training**.
 - Training on the new dataset (e.g. radiology) is called **fine-tuning**.
- Transfer learning (from models A to B) makes sense if:
 - Task A and B have the same input X.
 - You have a lot more data for Task A (e.g. image recognition) than Task B (e.g. radiology).
 - Low-level features from A could be helpful for learning B.

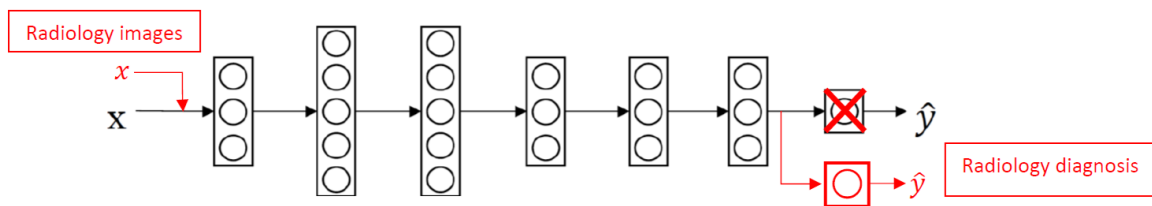


Figure 28: Transfer Learning

You can find more detailed guidelines about transfer learning on my website:
[Guidelines to use Transfer Learning in Convolutional Neural Networks](#)

3.10 Multi-task learning

Learning to perform several tasks simultaneously. For instance, learning to identify multiple objects at the same time (e.g. automatic car driving). The loss function must be a vector $\hat{y}^{(i)}$ where each entry corresponds to one object. One cost function for this can be:

$$J = \frac{1}{M} \sum_{i=1}^M \sum_{j=1}^c \mathcal{L}(\hat{y}_j^{(i)}, y_j^{(i)})$$

Where c is the number of object classes/types and \mathcal{L} is the logistic loss function.

- Unlike Softmax, one example (e.g. image) can have multiple labels.
- \hat{y} can “unknown” values in certain positions. To calculate \mathcal{L} in that case, we can simply exclude those terms from the inner sum of the cost function.

Multi-task learning makes sense when:

- Training on a set of tasks that could benefit from having shared lower-level features.
- Usually, the amount of data you have for each task is quite similar. If we focus on a single target task, then we need to make sure that the other tasks have a sufficiently large number of examples.
- Can train a big enough neural network to do well on all tasks with better performance (e.g. computer vision or object detection).

Transfer learning is much more used currently in practice.

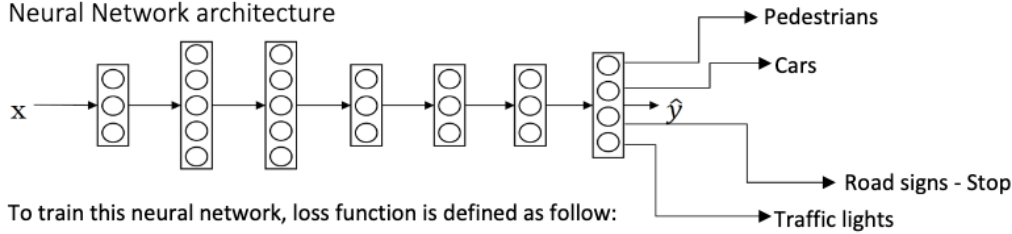
The input $x^{(i)}$ is the image with multiple labels
The output $y^{(i)}$ has 4 labels which are represents:

$$y^{(i)} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \begin{array}{l} \text{Pedestrians} \\ \text{Cars} \\ \text{Road signs - Stop} \\ \text{Traffic lights} \end{array}$$

$$Y = \begin{bmatrix} | & | & | & | \\ y^{(1)} & y^{(2)} & y^{(3)} & y^{(4)} \\ | & | & | & | \end{bmatrix} \quad \begin{array}{l} Y = (4, m) \\ Y = (4, 1) \end{array}$$



Neural Network architecture



To train this neural network, loss function is defined as follow:

$$-\frac{1}{m} \sum_{i=1}^m \left[\sum_{j=1}^4 \left(y_j^{(i)} \log(\hat{y}_j^{(i)}) + (1 - y_j^{(i)}) \log(1 - \hat{y}_j^{(i)}) \right) \right]$$

Figure 29: Multi-Task Learning

3.11 End-to-end learning

- A single classifier can replace multiple stages of processing in previous approaches (e.g. speech processing).
- For small datasets, the traditional pipeline approach works better and for large amounts of data neural networks work better.

Pros of end-to-end learning:

- Let the data speak (features are learned, not forced).
- Less hand-designing of components is needed.

Cons of end-to-end learning:

- May need a large amount of data.
- Excludes potentially useful hand-designed components.

Key question: Do you have sufficient data to learn a function of the complexity needed to map x to y ?

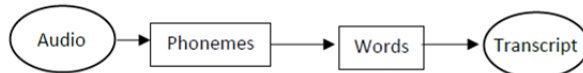
If not: It is also possible to use Deep Learning (DL) to learn individual components. Carefully chose $X \rightarrow Y$ to learn depending on what tasks you can get data for.

Example - Speech recognition model

The traditional way - small data set



The hybrid way - medium data set



The End-to-End deep learning way – large data set



Figure 30: End-to-End Learning

4 Convolutional Neural Networks

4.1 Fundamentals

Convolution: Applying a **kernel** or **filter** to a matrix. Denoted by operator $*$.

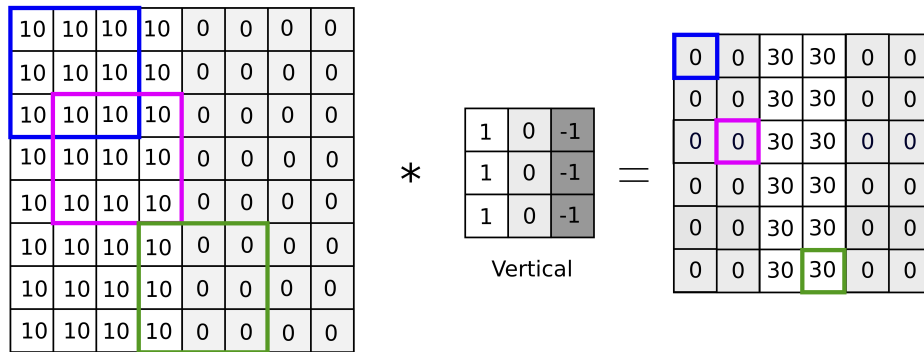


Figure 31: Vertical Edge Detection Example.

- Typical example is edge detection in images (e.g. the Sobel or Scharr algorithms/filters)

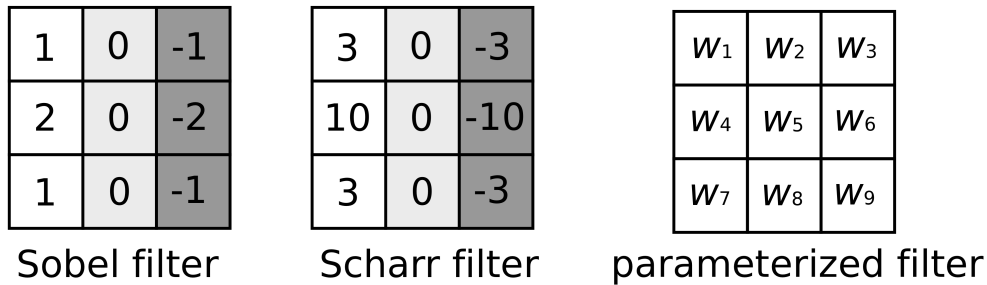


Figure 32: Filters.

- Size of the result of convolving an image of size $n \times n$ with a filter of size $f \times f$ is $(n - f + 1) \times (n - f + 1)$.
- **Padding:** adding an extra border of pixels set to 0 to avoid losing information. If $p = 1$ corresponding to a 1-pixel padding border, then the resulting convolved image is $(n + 2p - f + 1) \times (n + 2p - f + 1)$.
 - **Valid Padding** means that no padding is used.
 - **Same Padding** means a padding such that the size of the output image is the same as the input image, so $p = \frac{f-1}{2}$ (from solving $\frac{n+2p-f}{s} + 1 = n$ and assuming a stride of 1).
- **Strided convolution:** use a different step than 1 (e.g. 2) when convolving. Size of the output (for each dimension) becomes $\lfloor \frac{n+2p-f}{s} + 1 \rfloor$ where s is the stride.

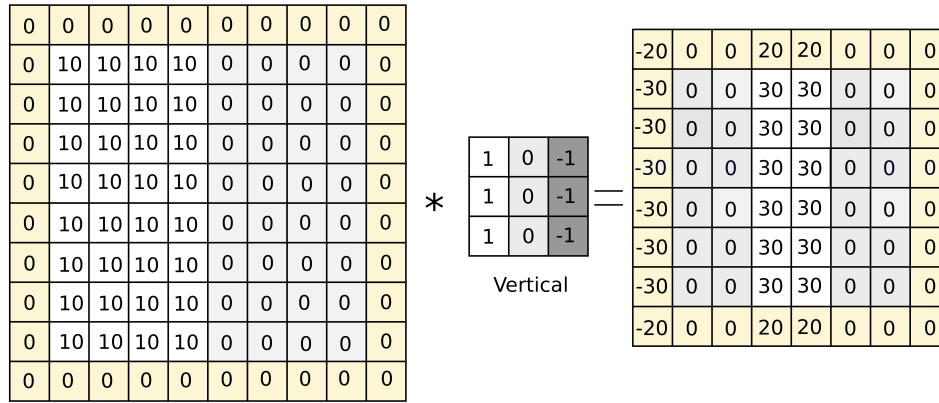


Figure 33: Padding.

- If we need the resulting image to be of a specific (or the original) size, then we need to solve for p the equation $\frac{n+2p-f}{s} + 1 = t$ where t is the target size (of each dimension).
- By convention f is always odd (so it has a central position and we can use the math above, but for no other specific reason).
- In math textbooks convolution is a mirroring step where the filter is flipped in the x and y axes. So technically the convolution used in Deep Learning is called ‘cross-correlation’, but in Deep Learning we still just call it convolution by convention. The flip is used in signal processing to ensure the associativity property of convolution $(A * B) * C = A * (B * C)$, not required in deep learning.
- On RGB images the filter is a cube $f \times f \times f$. Applying convolution is similar, but at each step, we do f^3 multiplications and then we sum all the numbers to get 1 output value. The sizing formulas above apply because the number of channels is not taken into account to determine the size of the output.
- If we have multiple filters (figure (35)), we can apply them to the same image and then stack the multiple outputs as if they were different channels n_C , into a *volume*. The term *depth* is often used to denote the *number of channels*, though that can be more confusing.

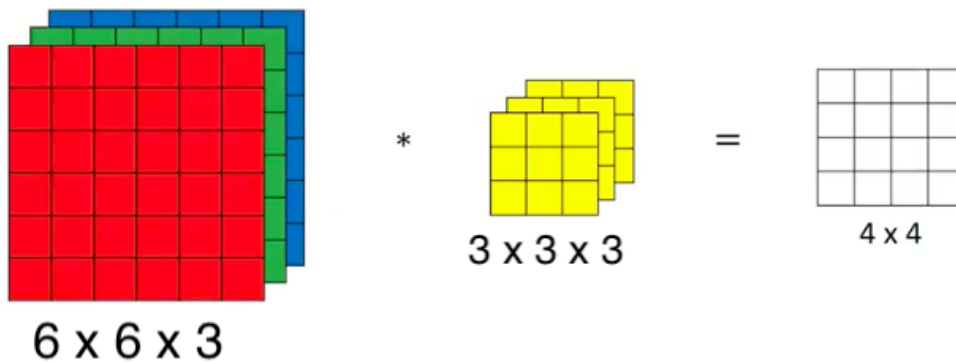


Figure 34: Convolution of a kernel on image

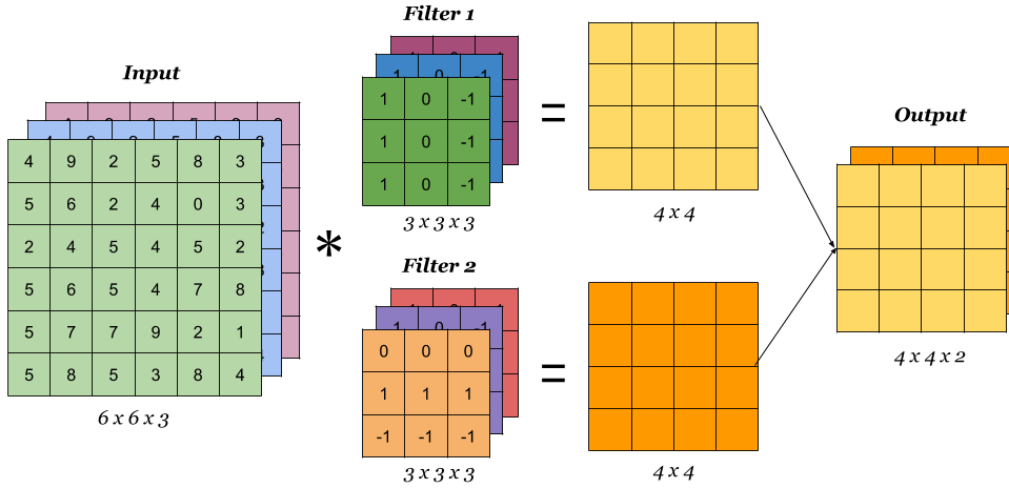


Figure 35: Convolution using multiple filters [2]

You can find an illustrated tutorial on convolution operation on my website:
[Understanding 1D, 2D, and 3D convolutional layers in deep neural networks](#)

4.2 One layer of a convolutional network

- Performs the convolution of the input image across all channels n_C , uses multiple filters, and therefore originates multiple convolution outputs.
- A bias term b is then added to the output of the convolution of each filter W and we get the equivalent of the term Z in a normal neural network.
- We then apply an activation function such as ReLU Z (on all channels), and we finally stack the channels to create a “cube” that becomes the network layer’s output. See figure (37).

Notation and volumes:

- $f^{[l]}$ = filter size of layer l
- $p^{[l]}$ = padding of layer l
- $s^{[l]}$ = stride of layer l
- $n_C^{[l]}$ = number of filters/channels of layer l
- Input size: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$
- Output size: $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$
- Output volume height: $n_H^{[l]} = \lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \rfloor$
- Output volume width: $n_W^{[l]} = \lfloor \frac{n_W^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \rfloor$
- Each filter volume: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]}$
- Activations volume: $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$

- Activations volume for M examples: $A^{[l]} \rightarrow M \times n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]} = M \times a^{[l]}$
- Weights volume: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]} \times n_C^{[l]}$ (# of filters)
- Bias size/number: $n_C^{[l]}$

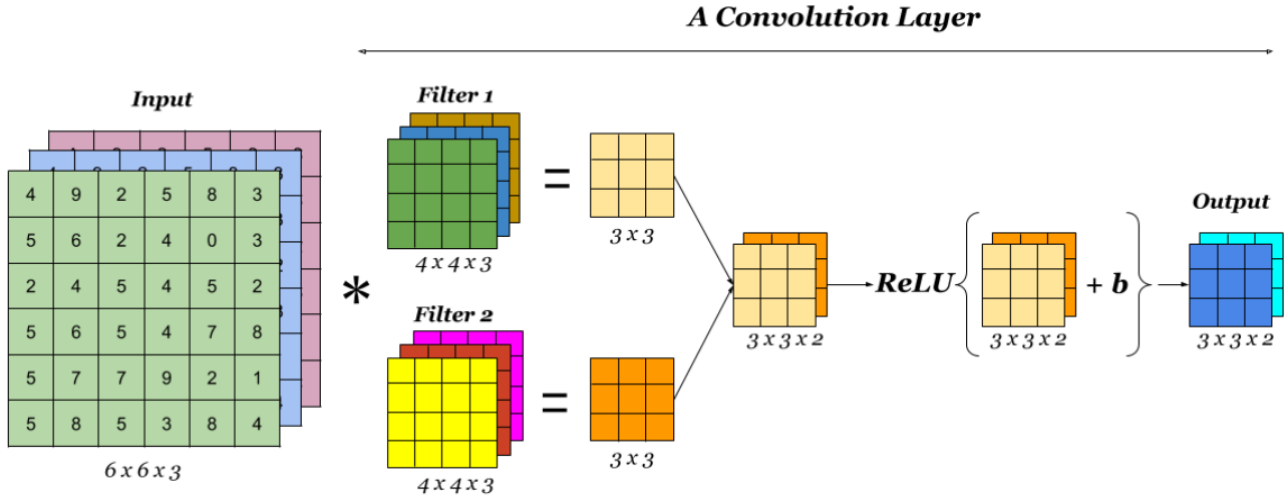


Figure 36: One Convolution Layer [2]

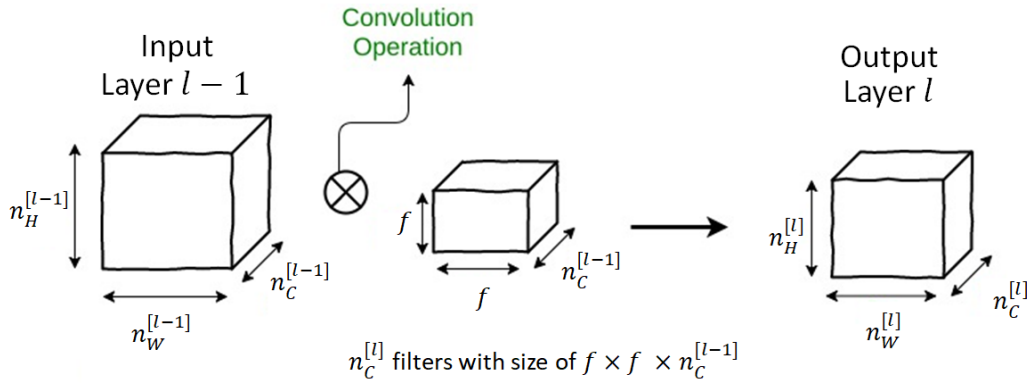


Figure 37: Convolution operator

4.3 Convolutional network

- Stack of layers as above (CONV layers)
- The last layer is a logistic or sigmoid, as required.
- There are also pooling (POOL) layers and fully connected (FC) layers.
- Sometimes POOL layers are not counted as layers since they don't have parameters to optimize. Here we will assume that one layer is a combination of CONV + POOL

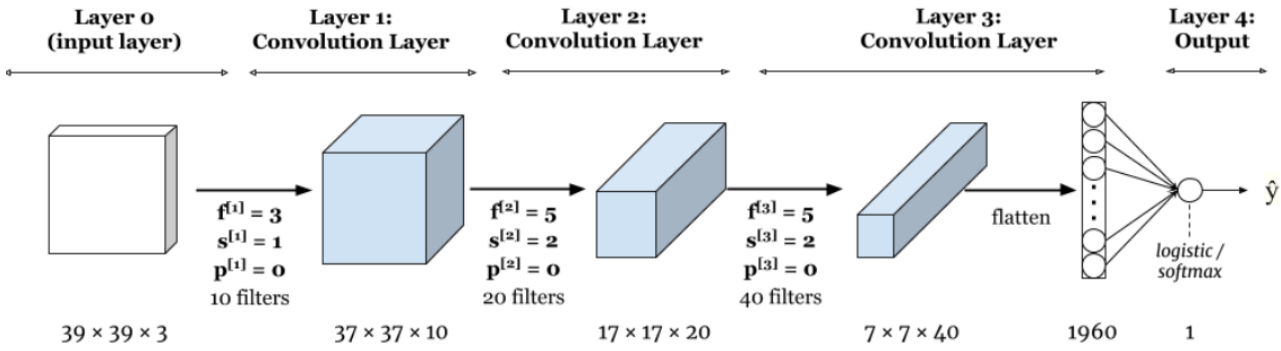


Figure 38: Sample Complete Network [2]

4.4 Pooling layers

- **Pooling** layer is used to reduce the size of the representations (width and height) and to speed up calculations, as well as to make some of the features it detects a bit more robust.
- **Max pooling** defines regions of size f (filter size) and steps through them according to stride s , and creates an output matrix containing only the max value of each region. This normally reduces the size of the representation.
- A pooling layer does not change the number of channels (depth) of the input.
- The intuition for using max pooling (except wide acceptance) is that it expresses that if a feature exists in a region, we will express it more evidently.
- Computations are made independently for each channel.
- Average pooling is also possible but less used, except deep in a neural network to collapse a representation, reducing image size.
- There are no parameters to optimize. Hyperparameters: f , s , and the type of pooling (max or average).
- Padding usually is not used for pooling layers (though there are exceptions).
- The size of the output will be calculated by *the same output volume* formulas above. Precisely, the output size of applying a pooling layer with the size of f and stride s to an input with the size of $n_H \times n_W \times n_C$ will be: $\lfloor \frac{n_H - f}{s} + 1 \rfloor \times \lfloor \frac{n_W - f}{s} + 1 \rfloor \times n_C$

4.5 Fully connected layers

- These are layers that take the output of CONV or CONV + POOL layers and take each individual pixel as an input (instead of a volume with channels/filters) by collapsing the volume into a single vector with an equivalent size.
- A fully connected layer is a normal neural network layer (e.g. with ReLU activation) that takes this collapsed output of the previous layer.
- FC layers have significantly more parameters than CONV.

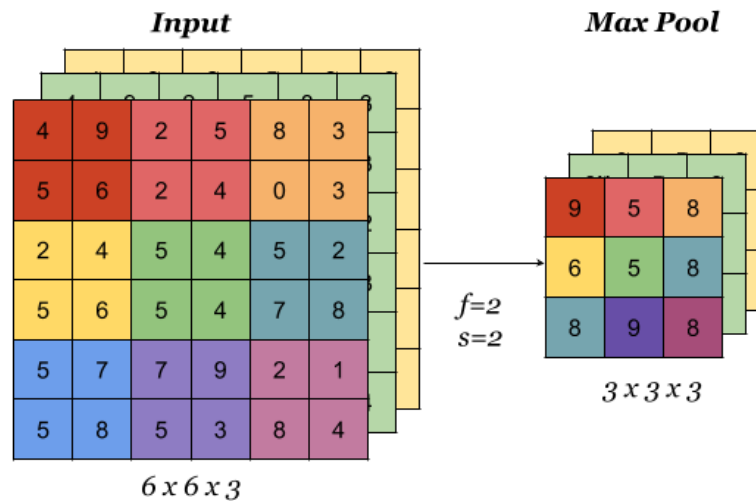


Figure 39: Pooling Layer [2]

4.6 Why convolutions?

- Even for a small image, normal NN would use a very high number of parameters per image.
- **Parameter sharing:** A feature detector (such as a vertical edge detector) that's 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 and the number of parameters depend only on the size of filters and it's independent of the size of the input of image/previous layer.

4.7 Historical/Classic architectures

- LeNet-5
 - INPUT \rightarrow CONV \rightarrow POOL \rightarrow CONV \rightarrow POOL \rightarrow FC \rightarrow FC \rightarrow OUTPUT
 - Uses Sigmoid/tanh instead of ReLU
 - Average pooling (not max pooling)
 - Includes not-linearity after the pooling
 - Different filters for each channel (not fully understood unless reading the paper)
 - Paper: “Gradient-based learning applied to document recognition”
- AlexNet
 - INPUT ($227 \times 227 \times 3$) \rightarrow CONV \rightarrow POOL \rightarrow CONV \rightarrow POOL \rightarrow CONV \rightarrow CONV \rightarrow CONV \rightarrow POOL \rightarrow FC \rightarrow FC \rightarrow Softmax \rightarrow OUTPUT
 - Similar to LeNet but much bigger
 - Uses max pooling
 - Most convolutions are “same”
 - Uses ReLU activation
 - Local Response Normalization \rightarrow normalizes each position in a volume across all channels of the volume (not very used in practice)

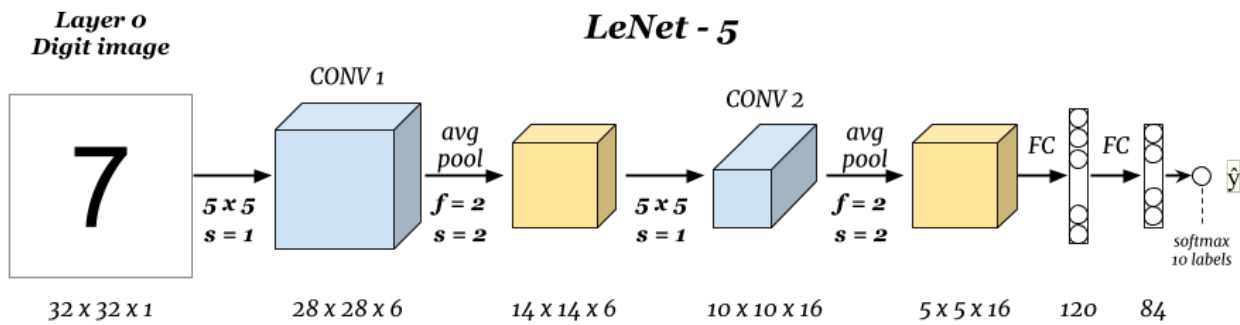


Figure 40: Lenet 5 [2]

- Paper: “ImageNet classification with deep convolutional neural networks”
- ~ 60 million parameters

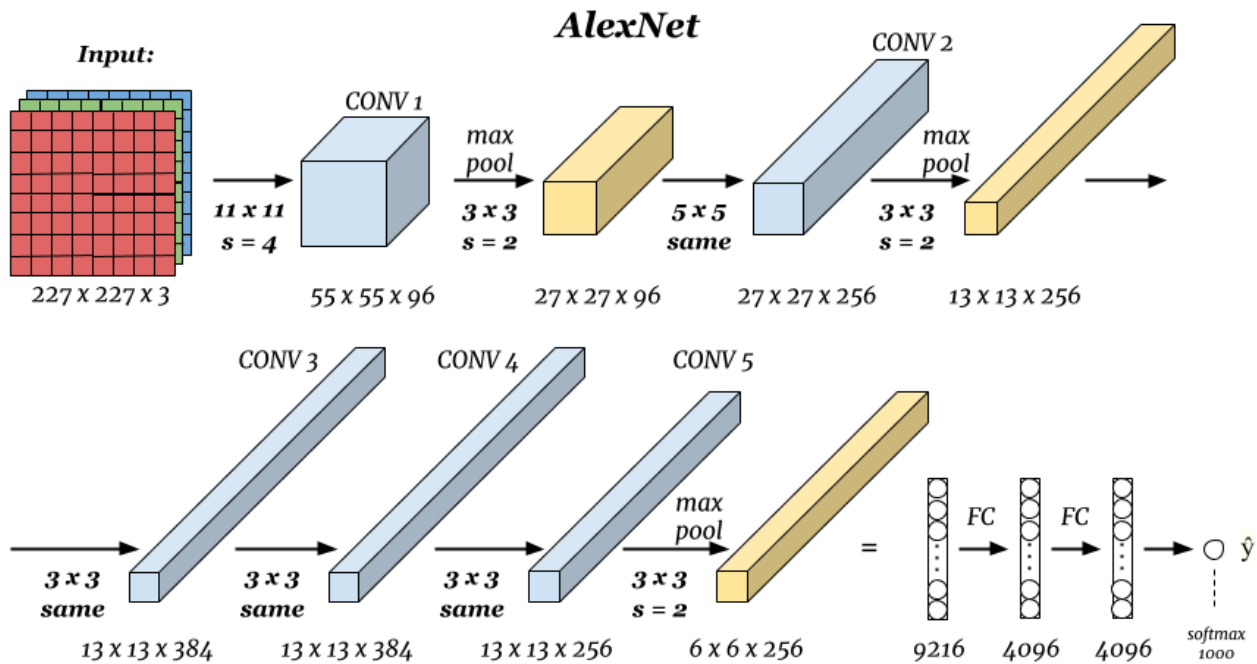


Figure 41: AlexNet [2]

- VGG-16
 - INPUT (224x224x3) -> CONV (64) -> POOL -> CONV (128) -> POOL -> CONV (256) -> POOL -> CONV (512) -> POOL -> CONV (512) -> POOL -> FC (4096) -> FC (4096) - Softmax (1000)
 - All CONV = 3x3 filter, s = 1, same
 - All MAX-POOL = 2x2, s = 2
 - ~ 128 million parameters
 - Paper: “Very deep convolutional networks for large-scale image recognition”

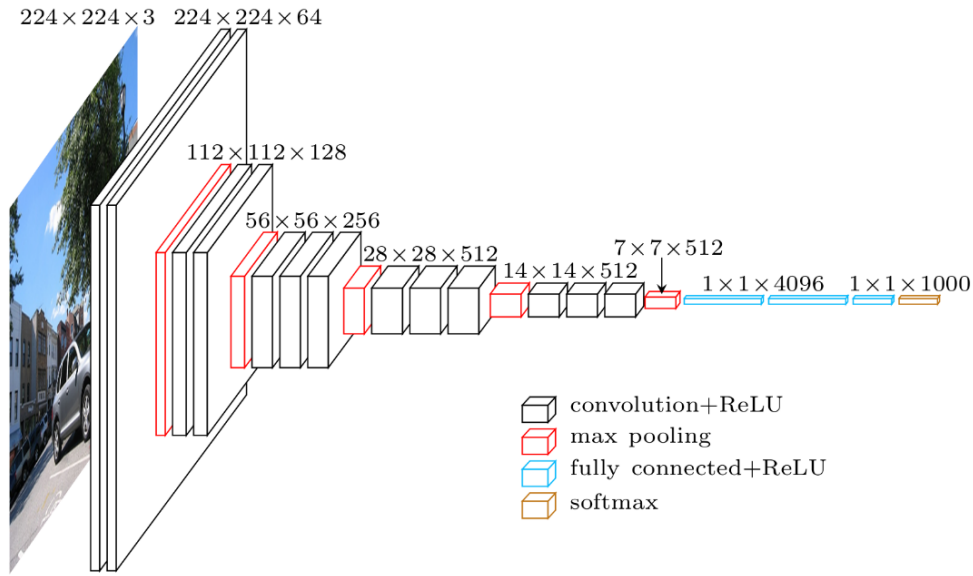


Figure 42: VGG 16

4.8 ResNet

- Paper: “Deep residual networks for image recognition”
- Add “shortcut”/“skip connection” to the network ([More details on Skip Connection](#)), where activation of past layers is included when calculating the activation of layers in the future:

$$z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]}$$

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

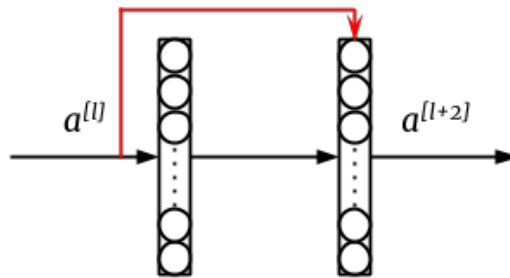


Figure 43: Skip Connection [2]

- These are residual blocks. A ResNet will have a sequence of these blocks.
- This allows training much deeper networks because the training error always goes down with an increase in the number of layers (which is not the case with traditional CONV nets):
 - “Learning the identity function is easy” - This means that it doesn’t hurt to add to layers even if they do not contribute to the activation of previous layers. When regularization is applied, then $W^{[l+2]} \approx 0$ and $b^{[l+2]} \approx 0$ and then $a^{[l+2]} = g(a^{[l]}) = a^{[l]}$
 - This means that a larger network will at least do as good as a smaller network, never worse.

- During the training of a traditional deep convolutional network we might see the magnitude (or norm) of the gradient for the earlier layers decrease to zero very rapidly as training proceeds. In ResNets the “shortcut”/“skip connection” allows the gradient to be directly backpropagated to earlier layers, thus solving the problem of **vanishing gradients**.
- For the operation $z^{[l+2]} + a^{[l]}$ to be possible, “same” convolutions must be used. Alternatively, there could be a W_s matrix such that $z^{[l+2]} + W_s a^{[l]}$ makes the dimensions match. This matrix could also add padding, or it could be just additional parameters to learn.
- Another similar (mentioned to be more “powerful”) design allows for skipping 3 layers instead of 2.

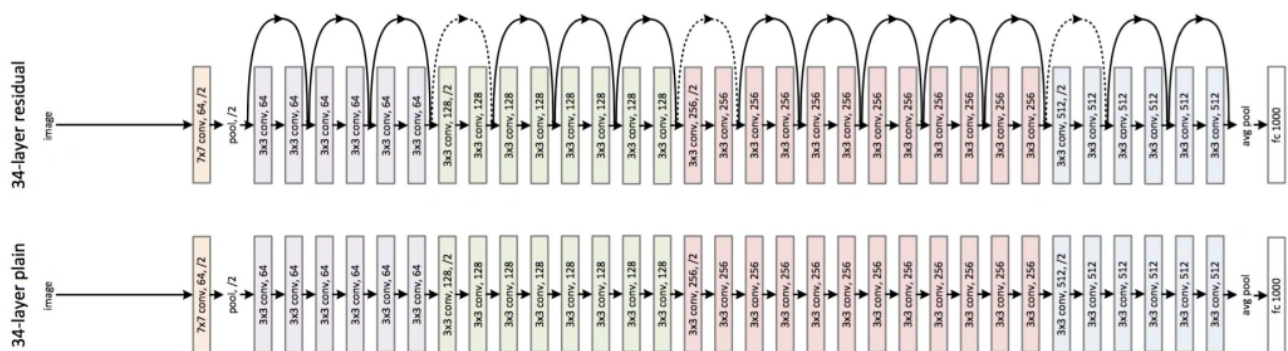


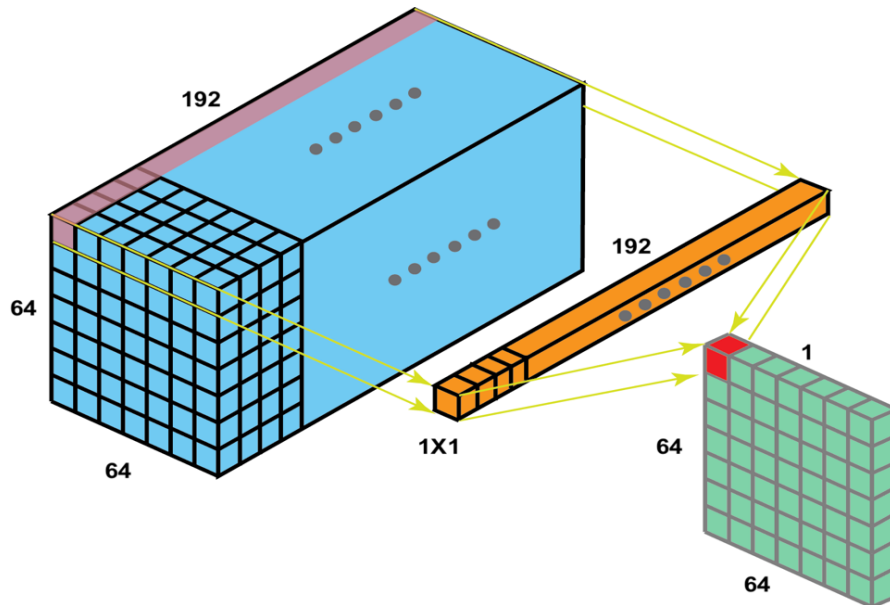
Figure 44: ResNet

4.9 Networks in Networks / 1x1 Convolution

- Useful only when the input has multiple channels.
 - Like having a fully connected layer for each pixel in the input.
- Useful to shrink the number of channels in a volume. Whereas pooling only reduces height and width.

4.10 Inception network (aka GoogleNet)

- Instead of choosing which filters to use, we used them all and the network chooses what’s important!
- In a single layer, applies several types of filters, including:
 - 1x1 convolutions
 - Several “same” convolutions with different filter sizes (and possible stride sizes)
 - MAX-POOL directly from the input, in which case **padding must be added before the MAX-POOL** to ensure that the height and width of the output are the same as the input - this is an exceptional case since padding is not usually added when doing POOL.

Figure 45: 1×1 Convolution

- To reduce the computation cost of convolutions on many channels, 1×1 convolutions with a lower number of channels are used as an intermediate step or “bottleneck” for convolutions with a larger number of channels (reducing the number of operations by a factor of 10, compared with doing the larger convolution on the input directly). See figures (47) and (48).
- Similarly, after the MAX-POOL step, a 1×1 CONV is used to reduce the number of channels in its output.
- So, 1×1 CONVs are used as intermediate (or final in the case of MAX POOL) steps in each layer either as bottlenecks to reduce the number of channels at the output of MAX-POOL.
- One layer containing these 1×1 CONV, the larger CONV, and the MAX-POOL (in parallel) is called an “Inception Module” (49) of which there are many in a network.
- There is a final FC and Softmax layer.
- Side-branches (a feature of inception networks): Take some hidden layers and use FC and then Softmax to predict a label. This ensures that even hidden layers are not too bad to predict the output cost. These have a regularizing effect on the network.
- Paper: “We need to go deeper”.

4.11 Using Transfer Learning

- Use transfer learning/pre-trained models. Example: *Imagenet* (1000 output classes).
- Put your own Softmax output layer:

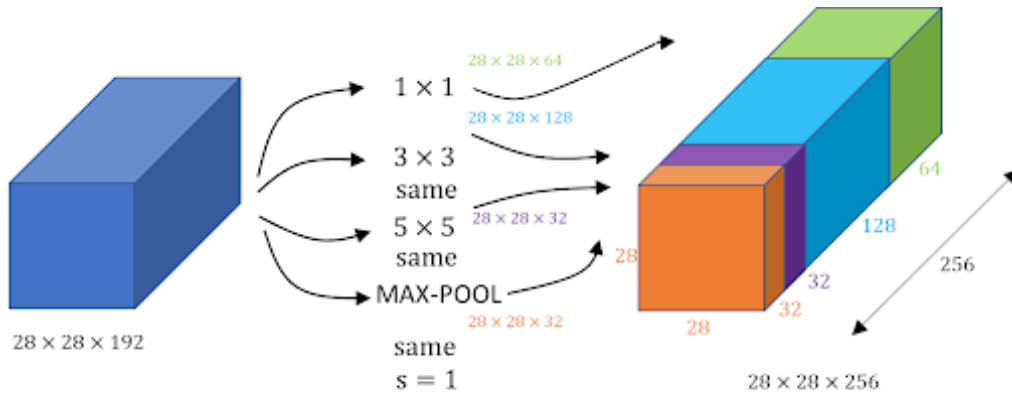


Figure 46: Inception Block

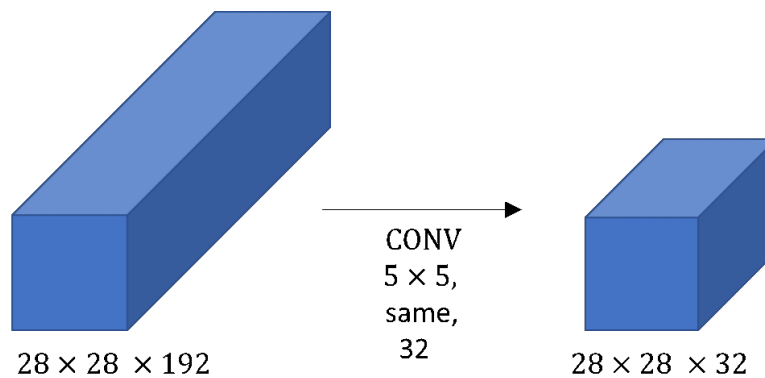
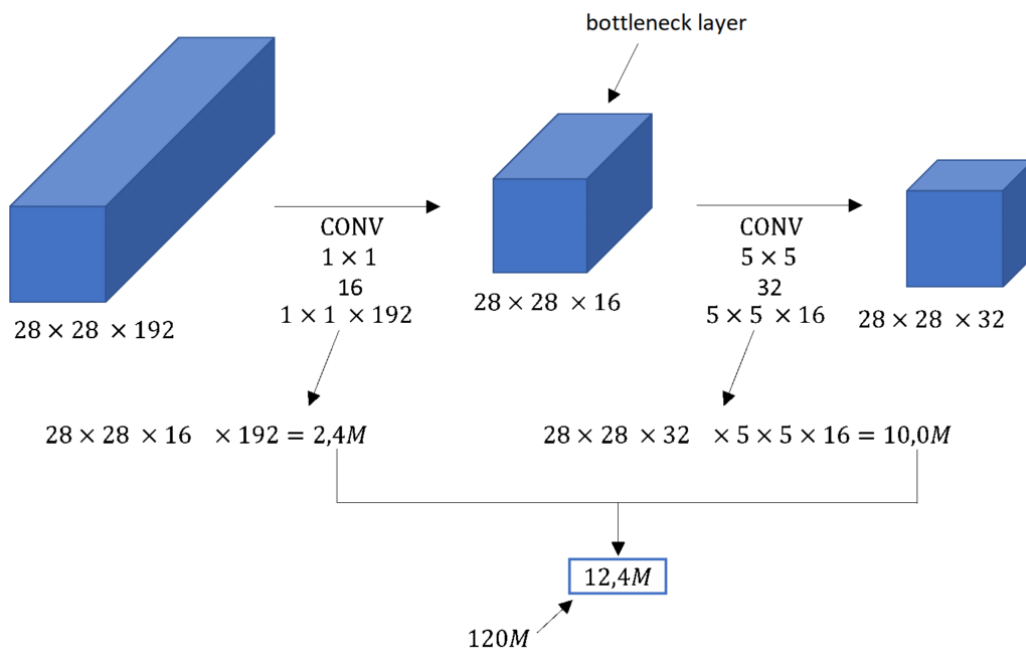


Figure 47: Huge computations in the inception block

- Freeze the parameters in the other layers (e.g. trainableParameter=0 or freeze=1, depending on framework).
- (with small data) Pre-compute the output of the network before the new Softmax layer across training examples and save them to disk (so you don't have to compute these on every epoch of the shallow training). Then train only the shallow Softmax model.
- (with more data) Include a few of the last few layers in the training, not just Softmax. We may want to retain or not the weights of some of the unfrozen layers (used them as initialization) and add just some additional extra training there. Alternatively, the last unfrozen layers could be randomly initialized, re-designed, etc. The more data you have, the fewer layers you freeze.

4.12 Data Augmentation

- Having more data helps in the majority of computer vision tasks (not the same for other areas of ML).
- Methods (from existing images):
 - Mirroring
 - Random cropping (works as long as the crops are reasonable)
 - Rotation

Figure 48: Reduce computations by 1×1 CONV

- Shearing
- Local warping
- Color shifting: change the RGB color balance according to some probability distribution. (it is possible to use PCA to choose RGB. Check AlexNet paper for “PCA color augmentation”)
- A common way of implementing augmentation is to have one (or multiple) CPU threads do the augmentation/distortion, and then run the network training in parallel in another thread or GPU.

4.13 State of computer vision

little data \leftarrow object detection - image recognition - speech recognition \rightarrow **lots of data**

Two sources of knowledge:

- Labeled data (x,y)
- Hand-engineered features/network architecture/other components

Often for computer vision, there is not as much data as needed. So hand-engineered features are more needed.

- Even though the amount of data increased dramatically in the last few years there is still a lot of hand engineering, specialized components, and hyperparameters for historical reasons.

4.14 Tips for doing well on benchmarks/winning competitions

(Not for production environments)

- **Ensembling:** train several networks independently and average their output. Typically 3-15 networks. Usually not used in production.

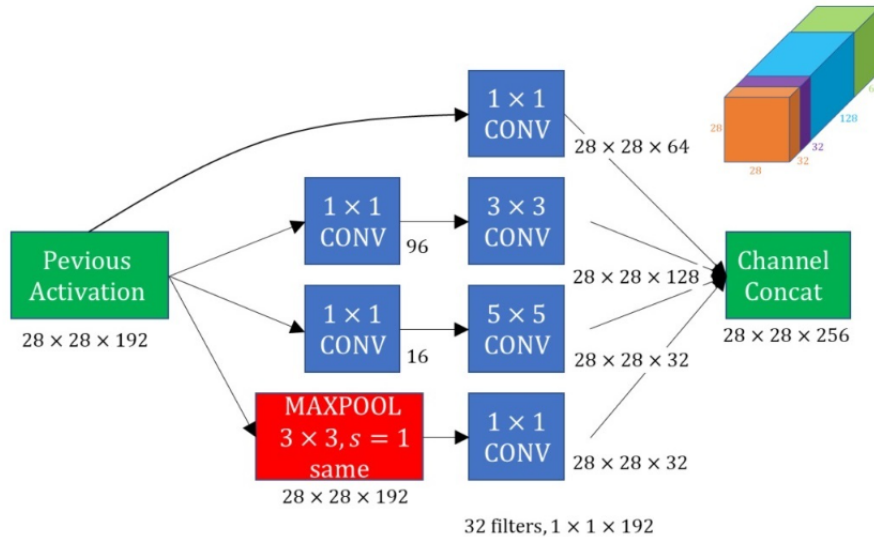


Figure 49: Inception Module

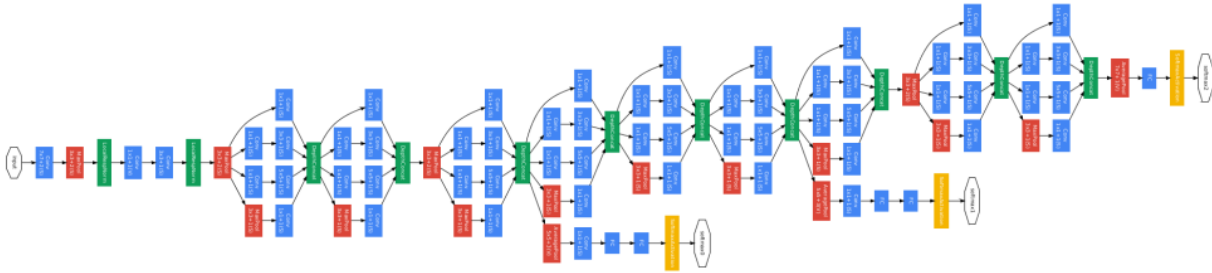


Figure 50: Inception Network

- Multi-crop at test time (data augmentation) - run the classifier on multiple versions of test images and average the results - more used in benchmarks rather than production systems.

4.15 Classification with localization

- Identify the object and localize it in the picture.
- Add 4 more outputs (in addition to the object classes) that are the bounding box coordinates b_x, b_y, b_h, b_w , that is the bounding box center coordinates and its height and width to the labels vector, respectively. Add another output P_c that indicates if there is an object in the image. For instance, in an object detection problem with three classes (c_1, c_2, c_3), the label vector for each sample would be as follows:

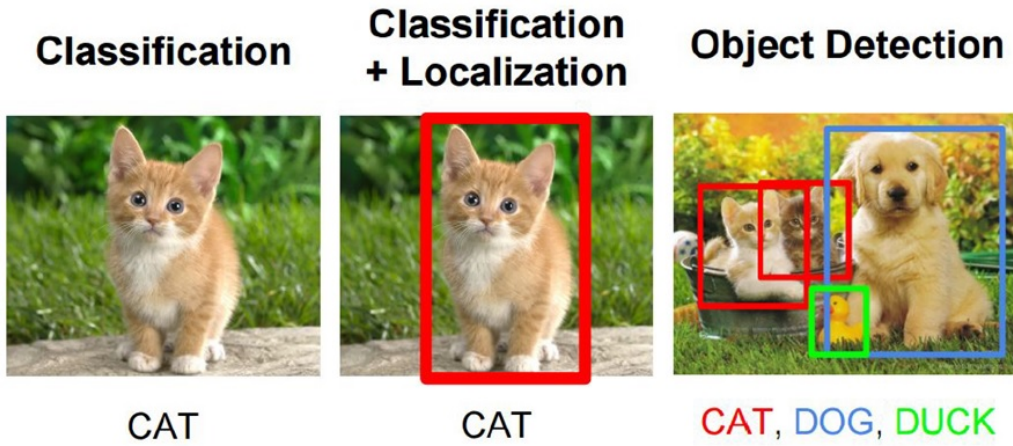


Figure 51: Object Classification, Localization, and Detection

$$y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_w \\ b_h \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

- If P_c is 0 (no object identified) then the loss function must take that into consideration, basically just calculating $(\hat{y}_1 - y_1)^2$ and ignoring the remaining components (this example uses squared error, but it could be a log-likelihood, etc.).

$$L(\hat{y}, y) = \begin{cases} (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + \dots + (\hat{y}_8 - y_8)^2, & \text{if } y_1 = P_c = 1 \\ (\hat{y}_1 - y_1)^2, & \text{if } y_1 = P_c = 0 \end{cases}$$

4.16 Landmark detection

- Basically use a conv network to output the coordinates of the points of interest (e.g. find the corners of the eyes in a person's picture). It can be a high number of points of interest though.
- Landmark labels must be consistent between training/test examples.

4.17 Object detection

- First build a conv net that identifies a certain class(es) of objects, and train it only on pictures where the positive examples include a single object (e.g. a single car).
- To identify multiple cars, use a sliding window approach (with windows of varying sizes, over multiple passes), and run each crop via the conv net. However, this is extremely inefficient because we don't know the stride or the size of the crops, and we need to evaluate a high number of crops.

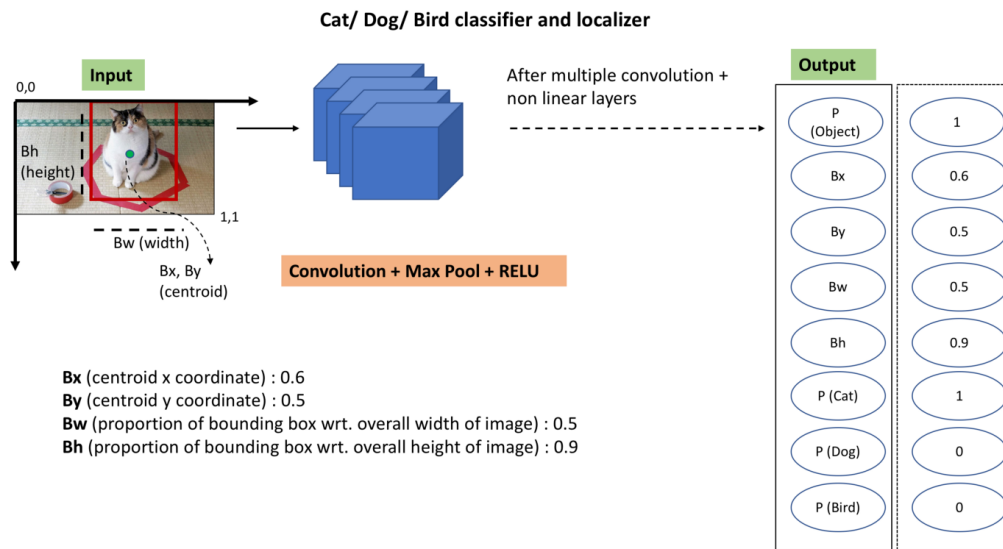


Figure 52: Label representation in Object Detection [1]

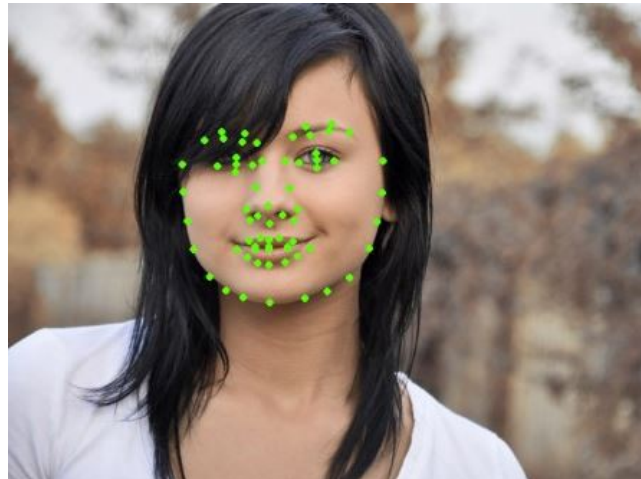


Figure 53: Landmark Detection

4.18 Convolutional implementation of sliding windows

First a useful tool: Turning FC layers into convolutional layers (55):

- Basically just use a filter with the same height and width as the input volume (no padding and or stride used), e.g. 5x5 in this example. The number of channels/filters can then correspond the number of units that we would have in an FC layer (e.g. 400 in this example). The output will be a 1x1x400 volume, which is equivalent to having a fully connected layer with 400 units!

And now the convolutional implementation for sliding windows:

- The objective is to implement a convolutional network that **in a single pass** calculates the equivalent of the convolution of a number of sliding windows, of a certain size and with a certain stride. If our test image is of dimension 16x16x3 and we had to perform the “regular” sliding

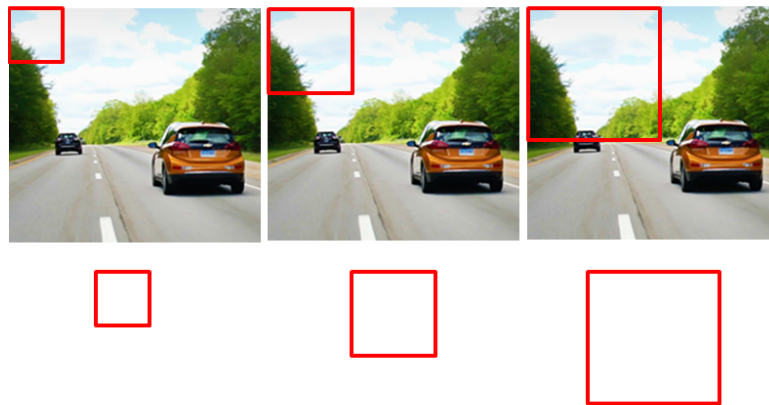


Figure 54: Sliding Window

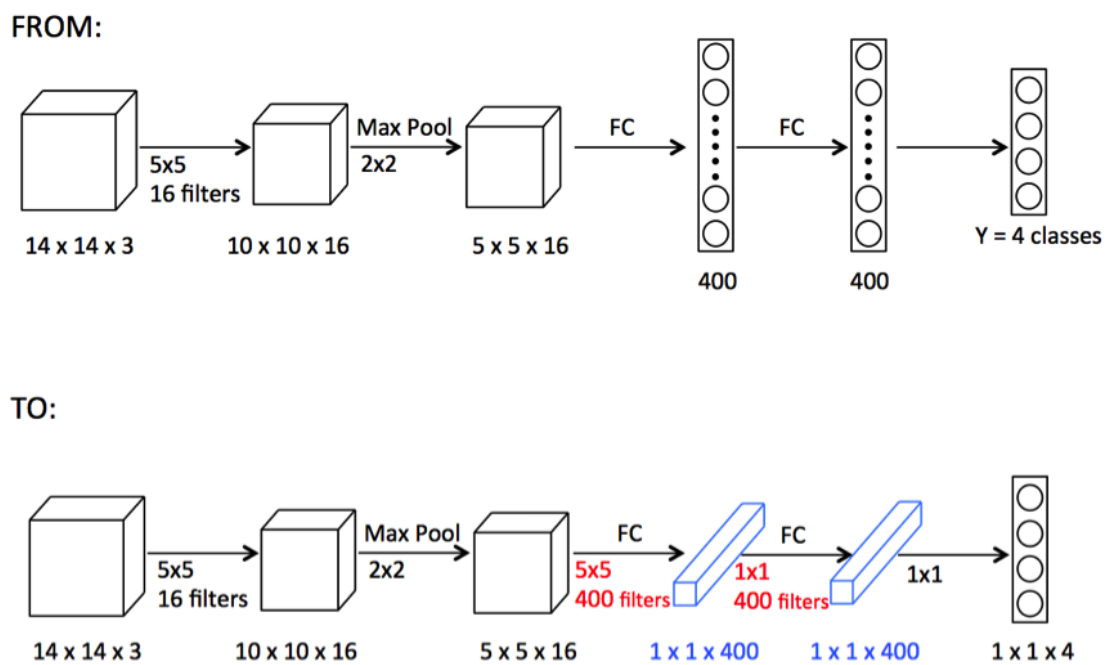


Figure 55: Converting Fully Connected layers to Convolutional Layers

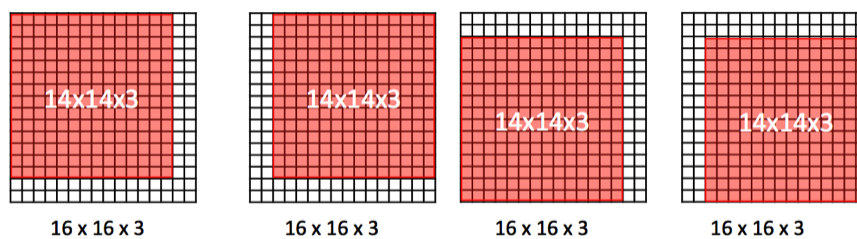


Figure 56: Sliding windows using Conv Net

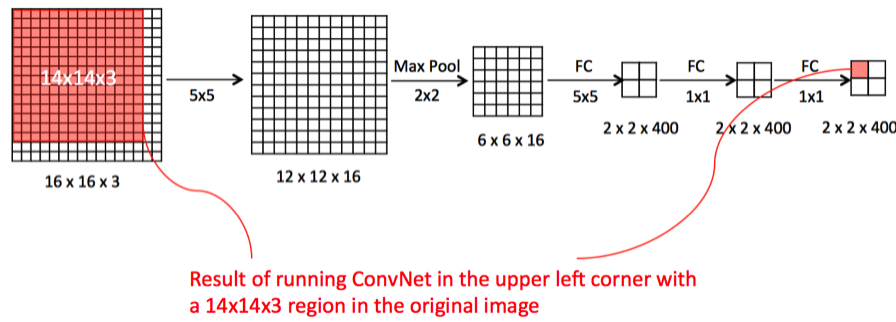


Figure 57: Computing all windows with one feed-forward

window we would have to create 4 different windows of size $14 \times 14 \times 3$ out of the original test image and run each one through ConvNet.

- This is computationally expensive and a lot of this computation is duplicated. We would like, instead, to have these four passes to share computation. So, with the convolutional implementation of sliding windows, we run the ConvNet with the same parameters and same filters on the test image and this is what we get in figure (57).
- Each of the 4 subsets of the output unit is essentially the result of running the ConvNet with a $14 \times 14 \times 3$ region in the four positions on the initial $16 \times 16 \times 3$ image.
- Think about an input image of $28 \times 28 \times 3$. Going through the network, we arrive at the final output of $8 \times 8 \times 4$. In this one, each of the 8 subsets corresponds to running the $14 \times 14 \times 3$ region 8 times with a slide of 2 in the original image.
- One of the weaknesses of this implementation is that the position of the bounding box we get around the detected object is not overly accurate.

4.19 YOLO algorithm

- Efficient algorithm, convolutional implementation, runs very fast. YOLO paper: “You Only Look Once: Unified real-time object detection”.
- A single CNN simultaneously predicts multiple bounding boxes and class probabilities for those boxes.
- First divide the input image into a grid (say 19×19). We want to use a conv net for each volume in a grid, or better do one single conv net pass that performs the equivalent of that. Say that for each grid division we want to have an output vector $y = [P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]$ (assuming 3 classes), we want to obtain an output volume of dimensions $19 \times 19 \times 8$ (8 is the dimension of the vector for each grid cell)
 - This can be achieved in a single convolution by setting the appropriate filter and max pool sizes/strides.
- YOLO considers that the grid cells that have an object are those that contain the center coordinates of the bounding box for the object, b_x and b_y .
- How to encode bounding boxes (that may go beyond the grid cell)? See figure (58).

- b_x and b_y are between 0 and 1 and are specified relative to the bounding box start point in the upper left corner (0,0).
- b_h and b_w are specified as a proportion in relation to the **grid cell size**, and therefore can be greater than 1.

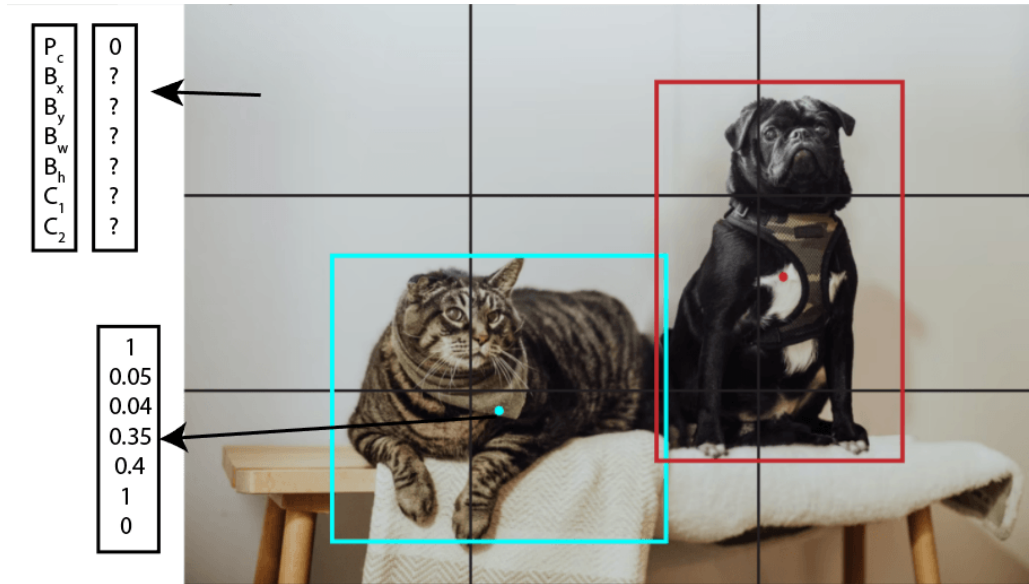


Figure 58: Bounding Boxes

Intersection over Union (IoU) - to evaluate object localization

- Given 2 overlapping bounding boxes, we define:

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

- We say that a bounding box is “correct” if its IoU with the ground truth bounding box is higher than 0.5 ($IoU \geq 0.5$). This is just a convention. 0.5 may be too low if we are more demanding (but never lower than 0.5).

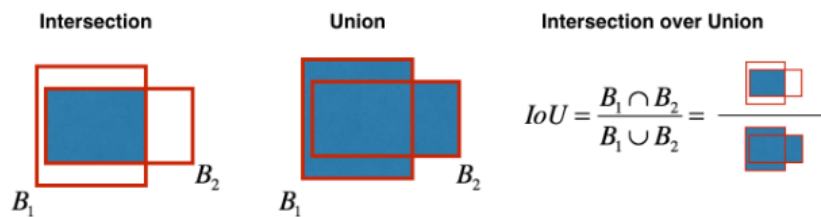


Figure 59: IoU

Non-max suppression

- One issue: the algorithm may predict several bounding boxes for the same object. Non-max suppression helps to select only one box per class, that has the highest probability.
- First of all we discard any bounding boxes with $P_c \leq 0.6$.
- Then we choose the highest probability bounding box first, and then iteratively eliminate the overlapping bounding boxes with $IoU \geq 0.5$, then select the next highest P_c , eliminate overlapping boxes with $IoU \geq 0.5$, and so on, while there are still boxes remaining.

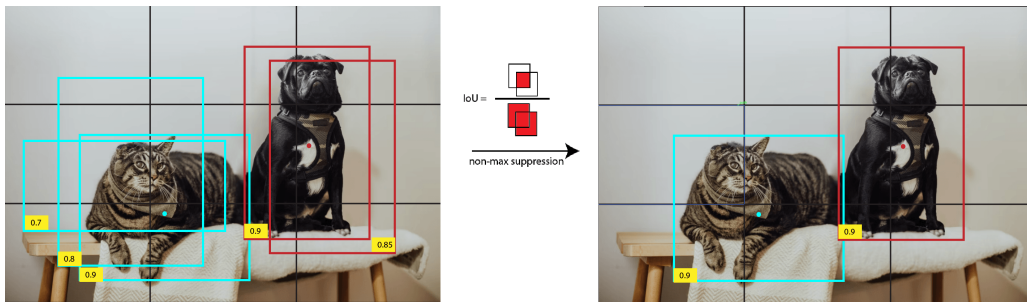


Figure 60: Non-max Suppression

Anchor boxes

- Goal: Detect several objects within the same grid. Anchor boxes are a set of predefined bounding boxes of a certain height and width. These boxes are defined to capture the scale and aspect ratio of specific object classes you want to detect.
- Anchor boxes increase the number of output parameters to encode multiple possible object detection. E.g. with two anchor boxes (and for 3 classes as before) the output vector is $y = [P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3, P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]$, notice that all parameters are doubled, one of each anchor box.
- Previously: Each object in the training image is assigned to the grid cell that contains that object's midpoint.
- With anchor boxes: Each object in the training image is assigned to the grid cell that contains the object's midpoint and anchor box for the grid cell with the highest IoU.
- What if you have 2 anchor boxes but 3 objects? The algorithm doesn't work well in this case.
- If there are 2 objects but both have the same anchor box shape the algorithm won't work well either.
- One way to choose anchor box shapes is by doing K-Means clustering over the anchor boxes of the training set.

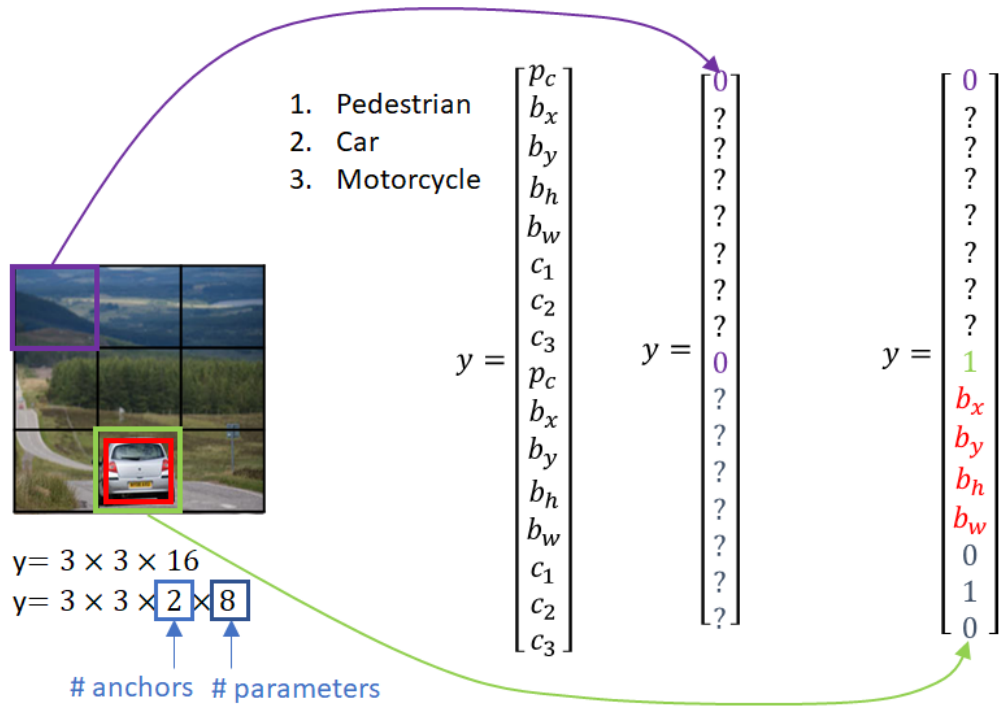


Figure 61: Anchor Boxes

YOLO algorithm (second take)

- If we have a grid of 3×3 , 3 classes and 2 anchor boxes, our output volume will be $3 \times 3 \times (5+3) \times 2$, where 5 comes from the components P_c, b_x, b_y, b_h, b_w .
- With 2 bounding boxes for each grid cell, get 2 predicted bounding boxes.
- Get rid of low-probability predictions.
- For each class, use non-max suppression to generate final predictions.
- Find out more details on my website:

[Understanding Object Detection using YOLO with Python implementation.](#)

Region proposals

- **R-CNN** (R as in regions) Select only a few regions/windows that seem relevant (contain objects) instead of sliding window through the entire image:
 - To find the regions, a **segmentation algorithm** is used, to find “blobs” that are candidates to be enclosed by bounding boxes.
 - Classify proposed regions one at a time. Output label + bounding box.
 - Fast R-CNN: Propose regions. Use convolution implementation of sliding windows to classify all the proposed windows.
 - Faster R-CNN: Use a convolutional network to propose regions. Still slower than YOLO.
 - Find out more details on my website: [RCNN, Fast RCNN, and faster RCNN algorithms for Object Detection Explained.](#)

4.20 Face recognition

- **Validation** (1:1 problem): Is he James? much easier problem than:
- **Recognition** 1:K persons. Who is that person? requires a database with recognized individuals.
- **One-shot learning**: Learning from 1 example to recognize the person again.
- Solution: Learning a **similarity function** for face verification instead of training a conv classifier on a small dataset:
 - $d(img1, img2)$ = degree of difference between images
 - if $d(img1, img2) \leq \tau$ same, otherwise different.

Siamese network

- Paper: “DeepFace closing the gap to human-level performance”.
- Idea: run each photo through a convolutional neural network and use the output as an “encoding” of the picture $x^{(1)}, \dots, x^{(m)}$ as $f(x^{(1)}), \dots, f(x^{(m)})$.
- Then calculate the distance between two encodings that is the norm between the difference of the two encodings:

$$d(x^{(1)}, x^{(2)}) = \|f(x^{(1)}) - f(x^{(2)})\|_2^2$$

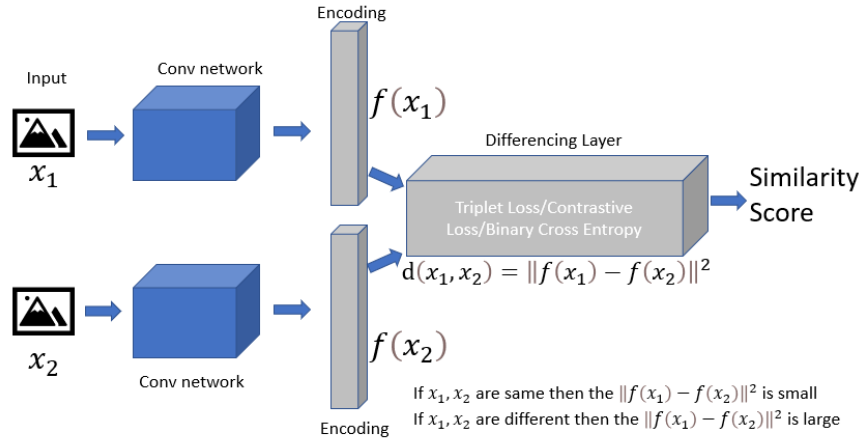


Figure 62: Siamese Network

The Triplet Loss function to learn parameters of the conv network

- Learning objective - Given an image A (*Anchor*), we want to find a function d such that d is small for another image of the same person P (*Positive*) and at the same time higher by an α **margin** when compared with an image of a different person N (*Negative*). In other words:

$$d(A, P) - d(A, N) + \alpha \leq 0$$

- Loss function - Given 3 images A, P, N (*triplet*) we define the loss as:

$$\mathcal{L}(A, P, N) = \max(d(A, P) - d(A, N) + \alpha, 0)$$

- We do need a dataset where we have multiple pictures of the same person (suggested 10 pictures per person on average).
 - Choosing the triplets randomly, then it makes it too easy for the training.
 - We should instead choose pictures where the negative is as close as possible to the positive, as these will be the best triplets to train on.
 - Paper: “A unified embedding for face recognition and clustering”.

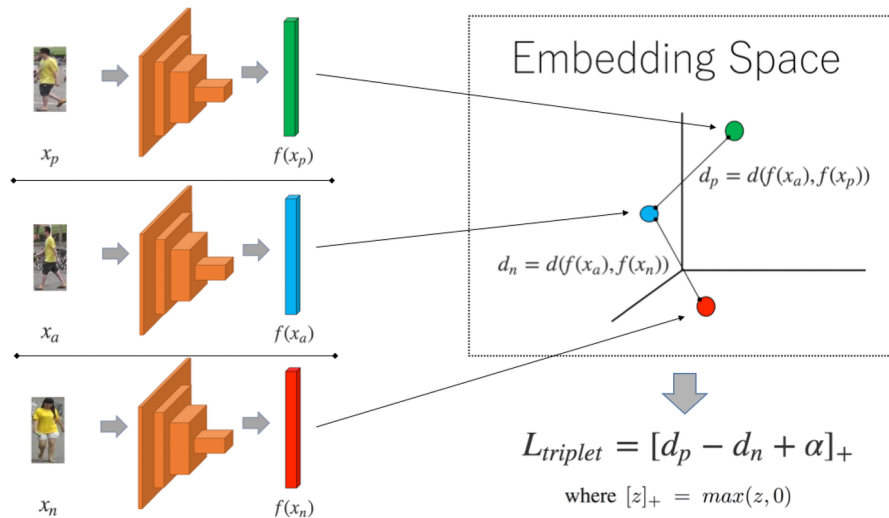


Figure 63: Triplet Loss

4.21 Face Verification as a binary classification problem

- Another way to learn weights of conv net.
- Example of **Siamese networks**, with a logistic regression unit for the output. That unit implements:

$$\hat{y} = \sigma\left(\sum_{k=1}^{N_f} w_i |f(x^{(i)})_k - f(x^{(j)})_k| + b\right)$$

- In this formula, $x^{(j)}$ and $x^{(i)}$ are training examples, w_i and b are the parameters of the logistic unit, N_f is the number of input features in each vector input to the logistic unit, k is one single feature (pair of nodes from each of the Siamese networks), and $f()$ is the function calculated by each of the Siamese networks.
- Take the difference of embeddings as features of a logistic regression model.
- In some variations the χ^2 formula is used instead of the absolute value of the difference:

$$\chi^2 = \frac{(f(x^{(i)})_k - f(x^{(j)})_k)^2}{f(x^{(i)})_k + f(x^{(j)})_k}$$

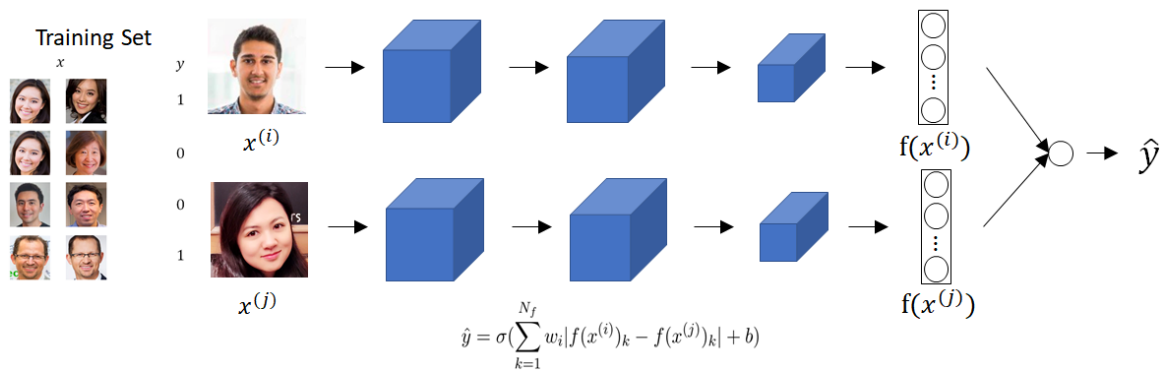


Figure 64: Face verification as a supervised learning problem

- Pre-compute encodings as a computational trick. We don't need to store the original images in a production verification system. Only the encodings of the images (e.g. for each individual).
- The Siamese network is trained using matching and mismatching pairs of pictures.

4.22 Neural style transfer

- Given a Content image C (a photo) and a style image S (e.g. a painting) generate a new version of the original content reflecting the style in the painting, the "Generated image" G .

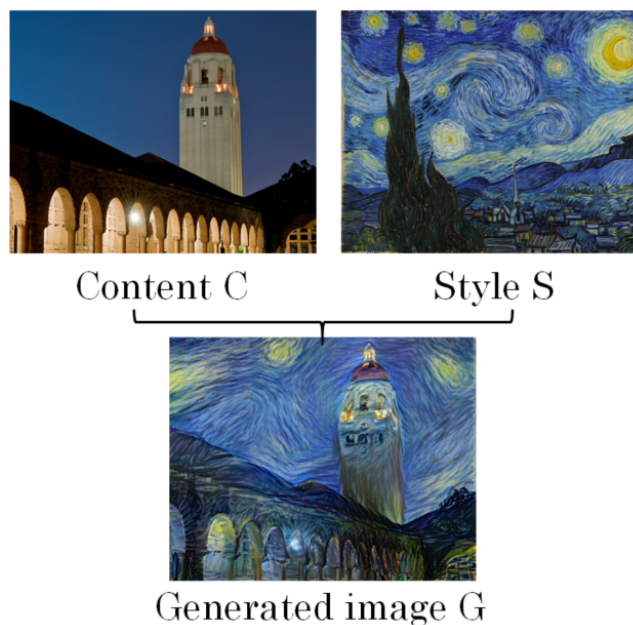


Figure 65: Neural Style Transfer

What are deep Conv Nets learning?

- Book/paper: "Visualizing and understanding convolutional networks".

- Pick a unit in layer 1, and find the nine image patches that maximize the unit's activation. Repeat for other units.
- Deeper units learn increasingly more complex/high-level features and shapes.

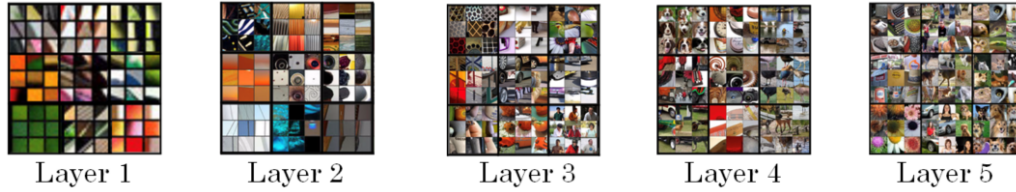


Figure 66: Visualizing deep layers

The cost function for neural style transfer

- Paper: “A neural algorithm of artistic style”.

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G)$$

- Two components measure the difference between the content image to the generated and the style image to the generated image, where α and β are hyperparameters.
- Algorithm:
 1. Initialize G randomly (e.g. G : 100x100x3 randomly initiated)
 2. Use gradient descent to minimize $J(G)$ and update G (where α is the learning rate):

$$G = G - \frac{\alpha}{2G} J(G)$$

Content Cost Function

- Choose a layer l somewhere in the middle of the layers of the network (neither too shallow nor too deep).
- Use pre-trained ConvNet. (E.g., VGG networks) to measure how similar the content and generated images are.
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images. If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content.
- Then calculate the L2 norm (element-wise sum of squared differences) of the activation vectors (activations are unrolled into a vector), with:

$$J_{content} = \frac{1}{2} \|a^{[l](C)} - a^{[l](G)}\|^2$$

- Finally, when we are generalizing in a $n_H \times n_W \times n_C$ volume we can use a more appropriate normalization constant:

$$J_{content}(C, G) = \frac{1}{4 \times n_H \times n_W \times n_C} \sum_{\text{all entries}} (a^{[l](C)} - a^{[l](G)})^2$$

Style cost function

- Using l 's activation to measure style, we define style as the **correlation between activations across channels**.
- Let $a_{i,j,k}^{[l]}$ = activation at (i, j, k) .
 - The **Style Matrix** (“Gram Matrix” in algebra) is $G^{[l]}$ and is $n_c^{[l]} \times n_c^{[l]}$. k and k' are coordinates of the style matrix corresponding to the correlation between channels k and k' .

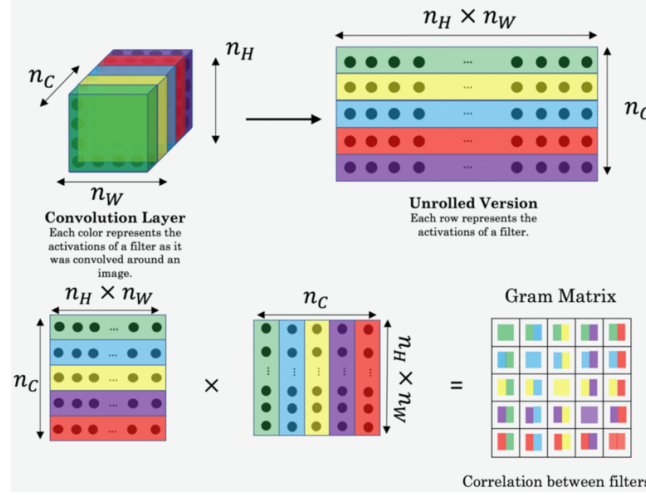


Figure 67: Style/Gram Matrix

- We calculate style matrices for the style image and the generated image:

$$G_{kk'}^{[l](S)} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{ijk}^{[l](S)} \times a_{ijk'}^{[l](S)}$$

$$G_{kk'}^{[l](G)} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{ijk}^{[l](G)} \times a_{ijk'}^{[l](G)}$$

- Finally the style cost function can now be defined by the Frobenius norm (sum of squares of the element-wise differences) multiplied by a normalization constant (the fraction portion):

$$J_{style}^{[l]}(S, G) = \frac{1}{(2n_H^{[l]}n_W^{[l]}n_C^{[l]})^2} \|G^{[l](S)} - G^{[l](G)}\|_F^2$$

$$J_{style}^{[l]}(S, G) = \frac{1}{(2n_H^{[l]}n_W^{[l]}n_C^{[l]})^2} \sum_k \sum_{k'} (G_{kk'}^{[l](S)} - G_{kk'}^{[l](G)})^2$$

- Finally style transfer is better if all the layers are used. So we get the final style cost function (with additional hyperparameter vector λ):

$$J_{style}(S, G) = \sum_l \lambda^{[l]} J_{style}^{[l]}(S, G)$$

5 Sequence Models

5.1 Applications

- Speech recognition
- Music generation
- Sentiment classification
- DNA sequence analysis
- Machine translation
- Video activity recognition
- Name entity recognition

5.2 Notation

Motivating example (Name entity recognition): Which words in the sentence are names?

x: Harry Potter and Hermione Granger invented a new spell.

y: 1 1 0 1 1 0 0 0 0

Elements in each position (9 positions in this case). t stands for timestamp:

$$x^{<1>}, x^{<2>}, \dots, x^{<t>}, \dots, x^{<9>}$$

$$y^{<1>}, y^{<2>}, \dots, y^{<t>}, \dots, y^{<9>}$$

- $x^{(i)<t>}$ is the t^{th} sequence element of example i .
- $y^{(i)<t>}$ is the t^{th} sequence element of output i .
- $T_x^{(i)}$ is the total number of sequence elements of training example i (9 in this case).
- $T_y^{(i)}$ is the total number of sequence elements of output example i (9 in this case).

Dictionary or **Vocabulary**: list of all words that we use in the representations.

One possible word representation: **One-Hot** vectors, where each word is a binary vector of the size of the whole vocabulary.

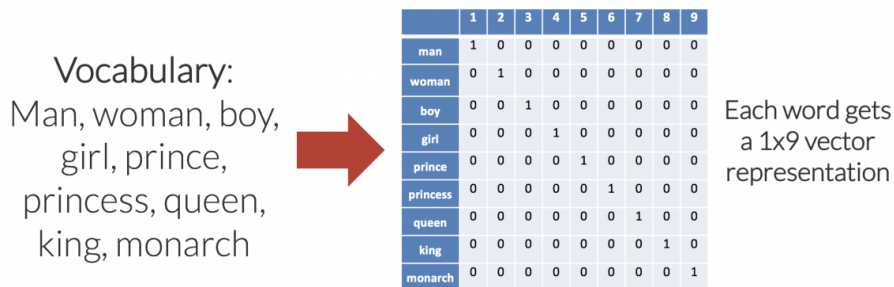


Figure 68: One Hot Representation

5.3 Recurrent neural networks (RNNs)

Why not use a standard network?

1) Inputs and outputs can be different lengths in different samples. 2) Typical networks do not share features learned across different positions of text.

Forward Propagation

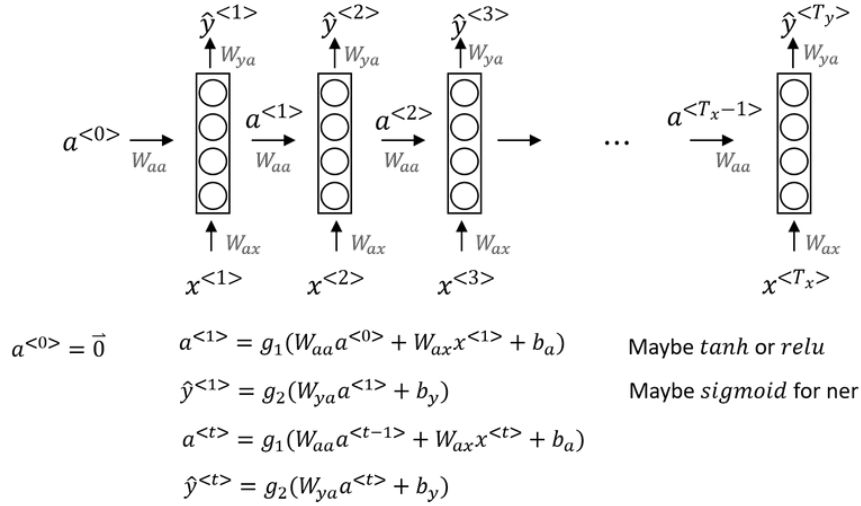


Figure 69: RNN forward propagation

RNN Forward Propagation The idea is to use the activation at $t - 1$ as well as the input at time t :

$$a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

That can be simplified by horizontally stacking W_{aa} and W_{ax} into $W_a = [W_{aa}; W_{ax}]$, and vertically stacking $a^{<t-1>}$ and $x^{<t>}$ as $[a^{<t-1>}, x^{<t>}]$:

$$a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$$

And we also have:

$$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$$

Which is usually simplified by renaming W_{ya} to W_y :

$$\hat{y}^{<t>} = g(W_y a^{<t>} + b_y)$$

Note that all weights (W_{aa} , W_{ax} , etc.) are shared among all time steps.

Dimensions of matrices assuming that the vocabulary is of the size 10000 and there are 100 hidden units:

$$x^{<t>} : (10000, 1), a^{<t>} : (100, 1)$$

$$W_{aa} : (100, 100), W_{ax} : (100, 10000), W_a : (100, 10100)$$

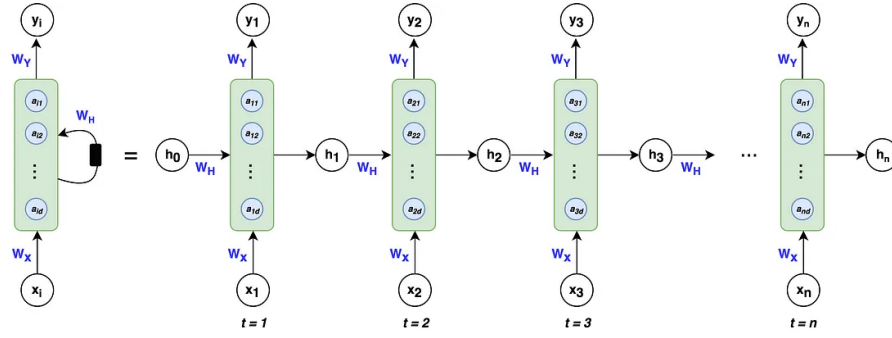


Figure 70: RNN: Folded and Unfolded

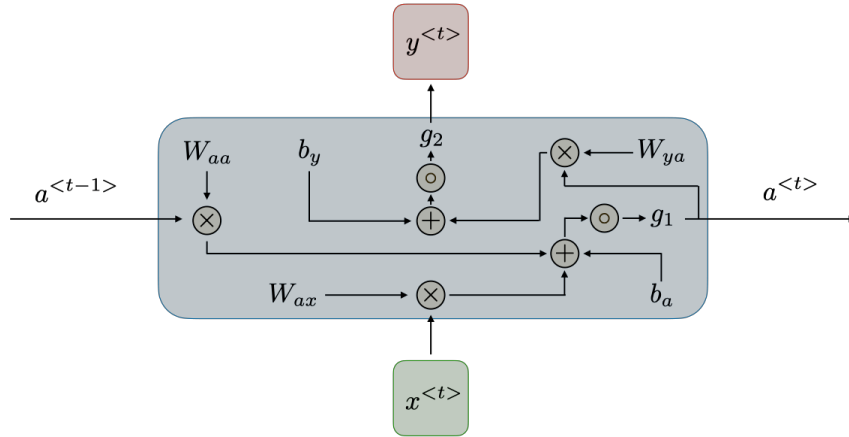


Figure 71: RNN unit

RNN backpropagation Loss is again the cross entropy loss (standard logistic regression loss). Loss at time step t :

$$\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -[y^{<t>} \log(\hat{y}^{<t>}) + (1 - y^{<t>}) \log(1 - \hat{y}^{<t>})]$$

Total loss on the entire sequence:

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

- **Backpropagation through time:** Gradient flows backward to the matrix multiplication node where we compute the gradients w.r.t. both the weight matrix and the hidden state. The gradient w.r.t. the hidden state and the gradient from the previous time step meet at the copy node where they are summed up.

Different types of RNNs Paper: “The unreasonable effectiveness of recurrent neural networks.”

- **Many-to-many:** many inputs and many outputs, when $T_x = T_y$. Name Entity Recognition.
- **Many-to-many:** many inputs and many outputs, when $T_x \neq T_y$, e.g. machine translation. Two parts, one encoder (e.g. from the source language) of size T_x , one decoder (e.g. to translate to the target language) of size T_y .

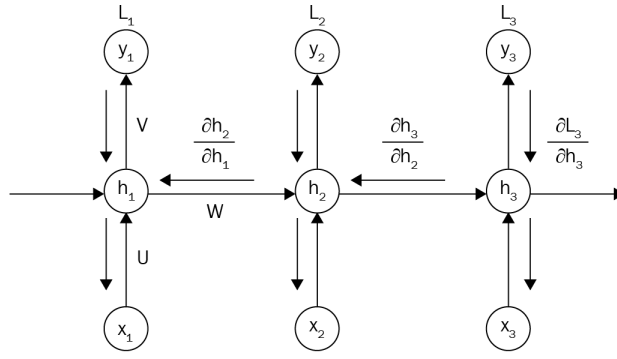


Figure 72: Backpropagation through time

- Many-to-one: many inputs and only one output (e.g. for sentiment analysis)
- One-to-one: just for the sake of completeness - really just a standard NN.
- One-to-many: one input, sequence of outputs (e.g. music generation). Note when generating sequences we usually take each output of $t - 1$ and feed it as the input of t (sometimes the single input can be 0).

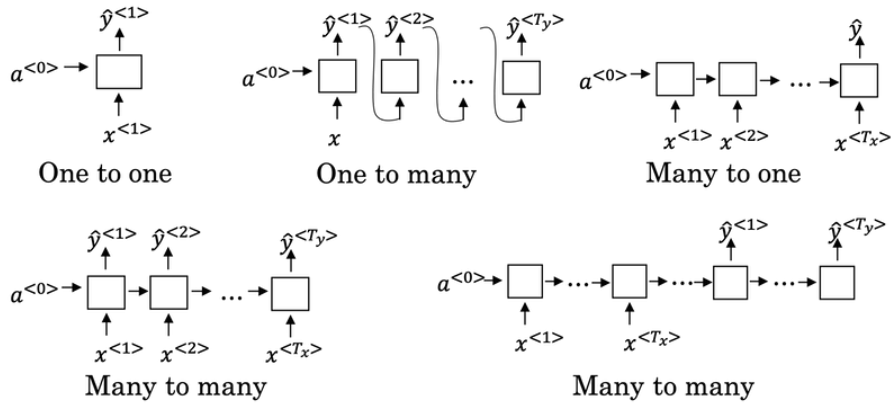


Figure 73: RNN Types

5.4 Language Model and Sequence generation

The goal of a Language Model (LM):

- 1) Calculating the probability of a given sentence, i.e. What is $P(\text{sentence}) = ?$

For example:

$$P(\text{'The apple and pear salad'}) = 10^{-15}$$

$$P(\text{'The apple and pear salad'}) = 10^{-10}$$

- 2) Given a sequence, what is the probability of the next element? Compute $P(\text{'word'} | \text{'The apple and pear ...'})$ for each 'word'.

How to learn an LM with RNNs:

- Training set: a large corpus of text from which a vocabulary is derived.

- Text is first tokenized (e.g. separated in words). It is generally useful to have an $\langle \text{EOS} \rangle$ (end of sentence) token. Words not in the vocabulary are replaced with the $\langle \text{UNK} \rangle$ token.

Dogs have an average lifespan of 12 years. $\langle \text{EOS} \rangle$
 $y^{(1)} \quad y^{(2)} \quad y^{(3)} \quad \dots \quad y^{(9)}$

The Irish Setter is a breed of dog $\langle \text{EOS} \rangle$
 $\langle \text{UNK} \rangle$

Figure 74: Tokenization [3]

- In a simplistic model, the size of the input vectors for each word is the size of the vocabulary (One-hot encoded).
- Set $x^{(t)} = y^{(t-1)}$.
- The output of each step of the RNN is a vector of the size of the vocabulary, and for each word in that vocabulary, it contains the conditional probability of that word, *given the previous words* already fed to the RNN.

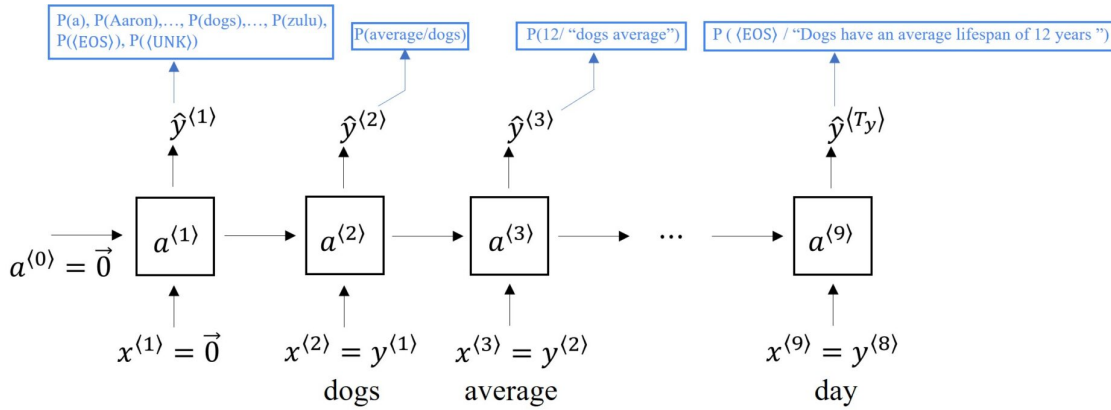


Figure 75: Train an LM using RNN [3]

- **Forward Propagation:** The forward propagation of our RNN model begins at time 0, wherein some activation $a^{(1)}$ is computed as a function of some input $x^{(1)}$, which in turn is set to all zeros or a 0-vector. The initial activation $a^{(0)}$ by convention is also an event of zeros. Now, the main role that $a^{(1)}$ plays is that it makes a Softmax prediction to try to predict the probability of the first word of our training example – “Dogs”. This is denoted by $\hat{y}^{(1)}$.

A Softmax prediction picks each word from the dictionary we have created and tries to predict the probability of that word occurring in a particular sentence. For example, computing the chance that the first word was “Zulu” or the chance that the first word is an Unknown Word, and so on. In short, Softmax helps us get to the output at each stage, that is, $\hat{y}^{(1)}$.

Continuing with the forward propagation, the RNN model uses the activation $a^{(1)}$ to predict the next word. At this time step, the network will also be given the correct value of the first word. In our case, we will tell the RNN that the first word was “Dogs”. This way the output of the previous time step also becomes the input for the next time step, that is, $y^{(1)} = x^{(2)}$. We move to the next time step and the whole process with Softmax is repeated as it predicts the second word. In the third time step, we tell the network the correct second word which is “have” and the next activation value $a^{(3)}$ is computed. In this step, the input is the first two words “Dogs have” and hence, the output of the previous step is again equal to the input of the next step, that is, $x^{(3)} = y^{(2)}$.

The process moves from left to right till it reaches the EOS token. In each step of the RNN, previous words act as input for the next word, and this way, the RNN learns to predict one word at a time propagating in the forward direction.

- **Back Propagation:** Now that we have performed forward propagation for our Language model using RNN, we shall look at how we are going to define the cost function and compute the loss using backpropagation. We are going to train our network by calculating loss at a certain time t . So, if at this time the correct word was .. and the neural network Softmax predicted the output .., we can easily compute the Softmax loss at this particular time.

$$\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) = - \sum_i y_i^{<t>} \log(\hat{y}_i^{<t>})$$

The total loss, as we understood previously, is nothing but the sum of losses computed for each time step and individual predictions.

$$\mathcal{L}(\hat{y}, y) = \sum_t \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

If we train this neural network for a larger training set, given an initial set of words such as “Dogs have an average” or “Dogs have an average lifespan”, our neural network will be able to predict the probabilities of the subsequent words using Softmax, such as $P(y^{(2)}|y^{(1)})$ and so on. Hence, for a three-word sentence or a four-word sentence, the total probability will be the product of all the previous probabilities of those three or four words.

- Calculate the probability of a new sentence:

$$P(y^{<1>}, y^{<2>}, y^{<3>}) = P(y^{<1>}) \cdot P(y^{<2>}|y^{<1>}) \cdot P(y^{<3>}|y^{<2>}y^{<1>})$$

Sampling novel sequences

- Using Softmax, our trained model predicts the chance or probability of a particular sequence of words and this can be represented as follows.

$$P(y^{(1)}, \dots, y^{(T_x)})$$

- An important aspect to be explored, once a Language Model has been trained, is how well it can generate new or novel sequences.
- The sampling begins after we have initiated the first input $x^{(1)} = 0$ and set the initial activation value $a^{(0)} = 0$. That produces a vector $\hat{y}^{<1>}$ with the probability of each word in the vocabulary (Softmax), from which we can sample a few with `np.random.choice`. Then we pick a word from

our sample (e.g. one which has a high probability of being the first word in a sentence such as “The”), and we feed it as the input $x^{<2>}$.

Then we pass on the output we sampled above, $\hat{y}^{(1)}$, and pass as input to the second time step. Softmax distribution will, then, make a prediction for the second output $\hat{y}^{(2)}$ based on this input, which is nothing but the sampled output of the previous time step.

- We obtain a vector $y^{<2>}$ corresponding to the conditional probability of each word in the vocabulary given the first chosen words. For example $P(y^{<2>} | \text{“The”})$ if the first chosen word was “The”.
- We repeat this until we have a sequence of the intended size or an $\langle \text{EOS} \rangle$ token is generated.
- Sometimes a $\langle \text{UNK} \rangle$ token can be generated, though it can be explicitly ignored.

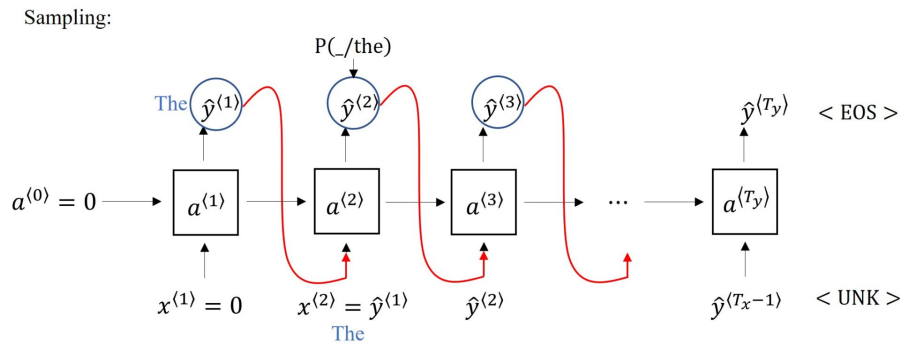


Figure 76: Sampling a sequence from an LM [3]

Note that the vocabulary can also be at the character level, which means that the sequence becomes the character sequence:

- Ends up with much longer sequences, computationally expensive, that don’t capture relationships between words.
- They have the advantage of not having to deal with $\langle \text{UNK} \rangle$ tokens.

Vanishing gradients with RNNs

- RNNs are not good at capturing long-range dependencies, for example: “The cat, which already ate . . . , was full.” The subject “cat” can be distant from the predicate “was”.
- This is due to vanishing gradients: the most common problem when training RNNs. Exploding gradients are catastrophic and usually cause NaNs in the output, so at least they are easy to spot (gradient clipping can be used but that is not ideal).

5.5 Gated Recurrent Unit (GRU)

- Paper: “On the properties of neural machine translation: Encoder-decoder approaches”
- Paper: “Empirical evaluation of Gated Recurrent Neural Networks on Sequence Modeling”

Vocabulary = [a, b, c, ..., z, ..., 0, ..., 9, A, ..., Z]

$y^{(1)}$ $y^{(2)}$ $y^{(3)}$ -individual characters

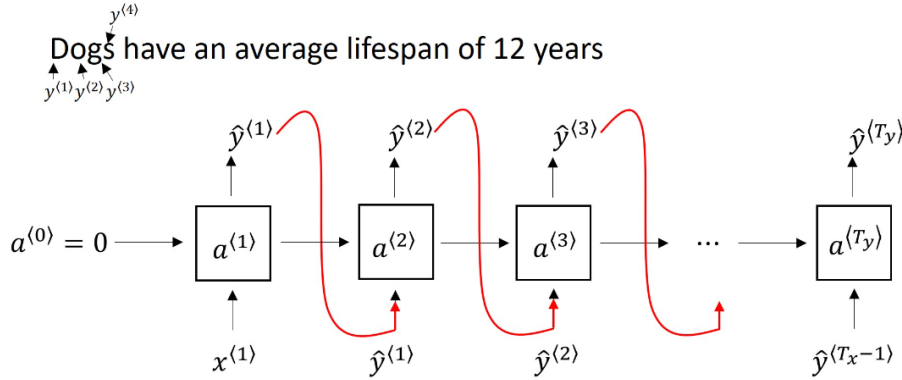
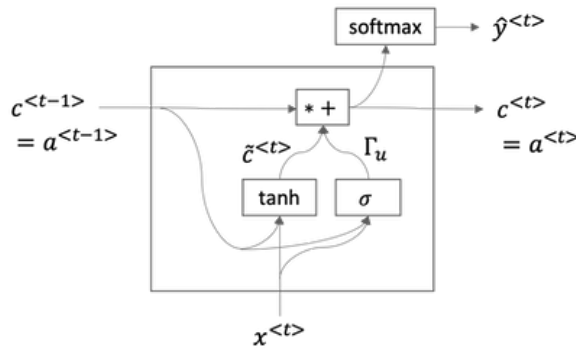


Figure 77: Character level LM [3]

GRU



$$\tilde{c}^{<t>} = \tanh(W_c[c^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$$

Full GRU:

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$$

Figure 78: GRU

- You can find a detailed tutorial about GRU on my website:

[Understanding Gated Recurrent Unit \(GRU\) with PyTorch code](#)

Notation:

c = memory cell

$$c^{<t>} = a^{<t>}$$

c and a are the same in this case, but they will be different in LSMTs, hence the distinction.

Example: c remembers if “cat” was singular or plural.

At each time instant we compute a *candidate replacement* for $c^{<t>}$ referred to as $\tilde{c}^{<t>}$.

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

The relevance gate term Γ_r ponders the relevance of $c^{<t-1>}$ for the new candidate replacement:

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$

Then we also have an update “Gate” (hence the use of the letter gamma Γ), that decides if the new candidate should replace the value of the memory gate or not.

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

This function is always very close to 0 or very close to 1, so it can be considered close to the binary output. Then the decision on whether to retain previous $c^{<t-1>}$ or update it with new candidate \tilde{c} is made with:

$$c^{<t>} = \Gamma_u * \tilde{c} + (1 - \Gamma_u) * c^{<t-1>}$$

- The $*$ operation is element-wise multiplication.
- Γ_u , $c^{<t>}$, and $\tilde{c}^{<t>}$ all have the same dimensions (equal to the number of hidden units).
- The Γ_u gate allows memorizing items like the “cat” for as long as needed, e.g. until the “was” token is found, in which case c can be replaced with \tilde{c} .
- It also solves the vanishing gradient problem for the most part.

5.6 Long Short-Term Memory (LSTM)

- Paper: “Long short-term memory” (hard to read!).
- Compared to the GRU:
 - LSTMs are much older than GRUs!
 - LSTMs are computationally more expensive, due to more gates.
 - GRU is a simplification of the more complex LSTM model.
 - No consensus over the better algorithm.
 - It does not use Γ_r .
 - $c^{<t>}$ (long-term memory) is no longer equivalent to $a^{<t>}$ (short-term memory).
 - Γ_u is replaced with two gates: Γ_u (How much to add information from the new candidate?) and Γ_f (How much to forget/remove information from the previous memory?)
 - It has an explicit output gate Γ_o
- Very good at memorizing values within long-range dependencies, such as the GRU.

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$$

$$\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$

$$a^{<t>} = \Gamma_o * \tanh(c^{<t>})$$

In some variants there is a “peephole connection”, adding $c^{<t-1>}$ to ALL gates, where the i^{th} element of $c^{<t-1>}$ affects the i^{th} element of Γ_u, Γ_f , and Γ_o .

You can find detailed tutorials about LSTM on my website:

- [A complete guide to understanding Long Short Term Memory \(LSTM\) Networks](#)
- [Implementing LSTM Networks in Python with Keras](#)
- [Understanding Long Short-Term Memory Networks \(LSTM\) with PyTorch codes](#)

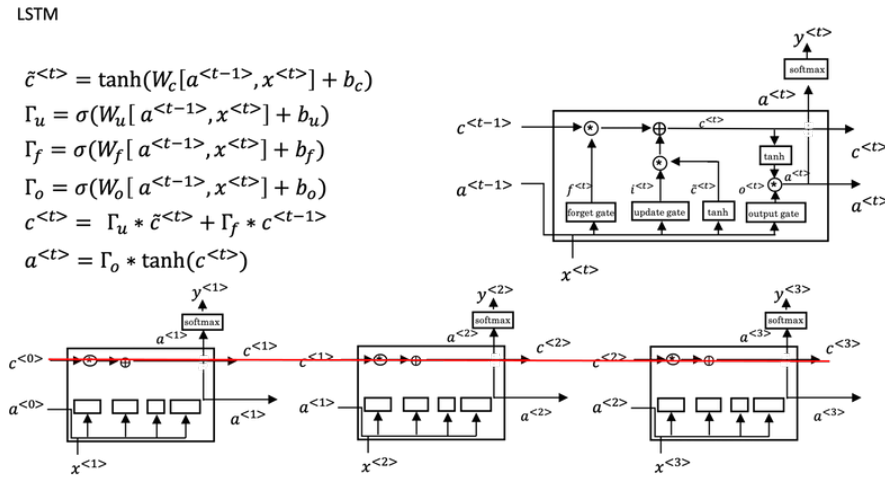


Figure 79: LSTM architecture

LSTM units

GRU

$$\begin{aligned}\tilde{c}^{<t>} &= \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c) \\ \Gamma_u &= \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u) \\ \Gamma_r &= \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r) \\ c^{<t>} &= \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>} \\ a^{<t>} &= c^{<t>}\end{aligned}$$

LSTM

$$\begin{aligned}\tilde{c}^{<t>} &= \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c) \\ \text{(update)} \quad \Gamma_u &= \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u) \\ \text{(forget)} \quad \Gamma_f &= \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f) \\ \text{(output)} \quad \Gamma_o &= \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o) \\ c^{<t>} &= \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>} \\ a^{<t>} &= \Gamma_o * \tanh(c^{<t>})\end{aligned}$$

Figure 80: LSTM vs. GRU

5.7 Bidirectional RNNs (BRNN)

- Forward propagation goes both forward and back in time, and there are separate activations for each direction for each instant t .
- BRNNs work with GRU and LSTM (LSTM seems to be more common).
- Able to make predictions in the middle of the sequence from both the future and past elements in the sequence.
- Does require the entire sequence of data before making predictions (does not work well for speech recognition in real-time, for example).

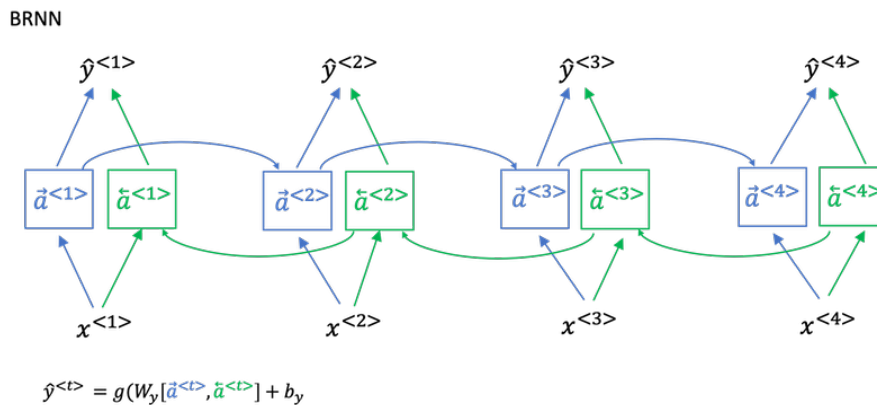


Figure 81: Bidirectional RNN

5.8 Deep RNNs

- RNNs can be stacked “vertically”, where each layer l has the same number of time units t , and the output $y^{<t>}$ is connected as the next layer’s $(l + 1)$ input.
- $a^{[l]<t>}$ represents the activation of layer l at time t .
- Usually, there aren’t many layers; 3 is already quite a lot.
- In some architectures, each output $y^{<t>}$ can be connected as the input to another neural network (e.g. densely connected).
- LSTMs and GRUs can work in this architecture as well as BRNNs.

Updated RNNs layer indexing notation

- Superscript $[l]$ denotes an object associated with the l^{th} layer.
 - Example: $a^{[4]}$ is the 4^{th} layer activation. $W^{[5]}$ and $b^{[5]}$ are the 5^{th} layer parameters.
- Superscript (i) denotes an object associated with the i^{th} example.
 - Example: $x^{(i)}$ is the i^{th} training example input.

Deep RNN example

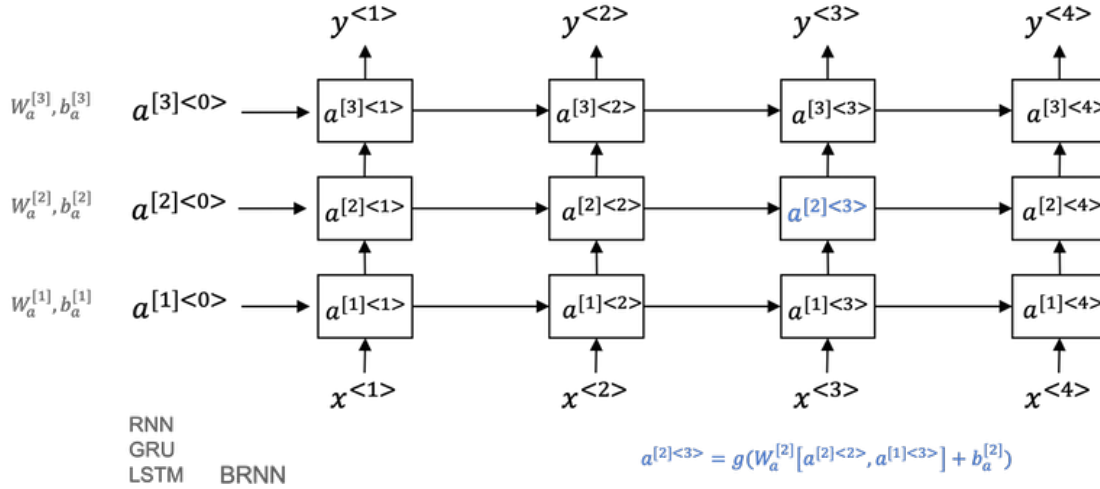


Figure 82: Deep RNNs

- Superscript $\langle t \rangle$ denotes an object at the t^{th} time-step.
 - Example: $x^{(t)}$ is the input x at the t^{th} time-step. $x^{(i)\langle t \rangle}$ is the input at the t^{th} timestep of example i .
- Subscript i denotes the i^{th} entry of a vector.
 - Example: $a_i^{[l]}$ denotes the i^{th} entry of the activations in layer l .

5.9 NLP and Word embeddings

One-hot representation: Each word is represented by a binary vector for the size of the vocabulary with a 1 at the position that corresponds to a specific word in that vocabulary. The vectors generated *do not encode semantic proximity*. For example, the meaning of apple and orange in the following sentences when trying to find the next word:

I want a glass of orange _____ I want a glass of apple _____

Word embedding (featurized representation): Each word is represented by high-dimensional feature vectors that include a measure of distance between the word represented by the vector and a series of features or other terms. Each word embedding can be named by the reference where the number is the index of the word in the vocabulary; e_{5391} or e_{9853} for example for the first and second column, respectively.

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1.00	1.00	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.7	0.69	0.03	-0.02
Food	0.04	0.01	0.02	0.01	0.95	0.97
...

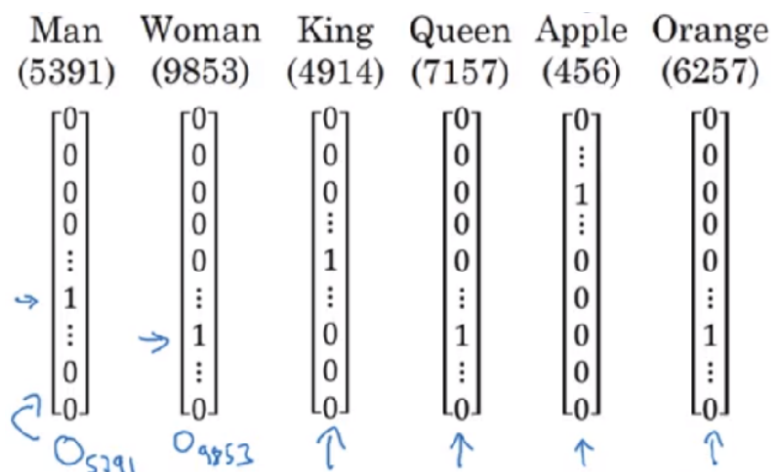


Figure 83: One-Hot representation of words

- Word vectors can be compared to find close relationships between terms and this helps generalize. In the example above, if we know that the missing word is “juice”, the vectors for “apple” and “orange” should be close enough that a learning algorithm can infer the next word “juice” in both cases.
- Words can then be visualized together with t-SNE (plot high dimensional spaces into 2d or 3d spaces), where the word proximity is visible.

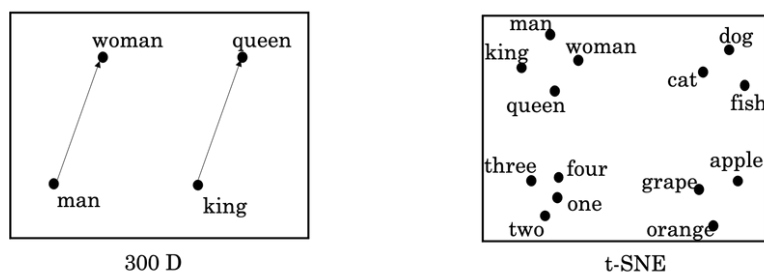


Figure 84: Word embedding

Using word embeddings

- It is possible to use “transfer learning” to learn word embeddings from a large body of text, for example, 1 billion words from the internet, and then use those embeddings with a smaller 100k word training set for the new task (e.g. Name Entity Recognition).
- This also allows representing words with smaller (compared to 1-hot) dense vectors of embeddings (e.g. size 300 instead of 10k). Word representation sizes are no longer restricted to the size of the vocabulary.

Properties of word embeddings

- Paper: “Linguistic regularities in continuous space word representations”

- Allow finding analogies: e.g. by subtracting the word embeddings of “Man” and “Woman” or “King” and “Queen” (from the table above) we find that the main difference between them is the “Gender”.
- With word embeddings we can use equations for finding the most appropriate words (**analogy tasks**):

$$e_{man} - e_{woman} \approx e_{king} - e_w$$

$$\text{Find word } w: \operatorname{argmax}_w \operatorname{sim}(e_w, e_{king} - e_{man} + e_{woman})$$

- The most common similarity (*sim*) used is the cosine similarity:

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

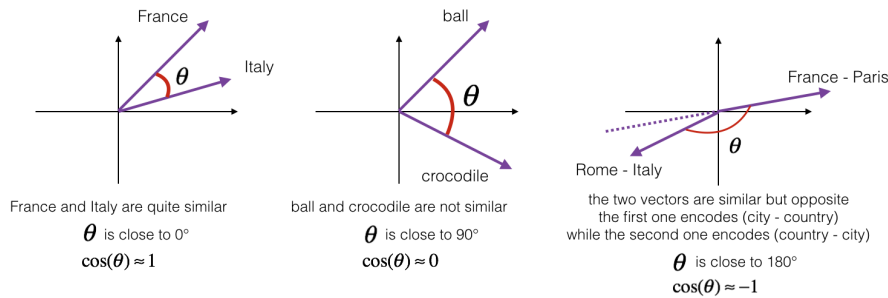


Figure 85: Cosine Similarity

- Another possible measure is the squared distance (euclidean), though it is a distance measure, not a similarity measure, so we would need to take its negative:

$$\|u - v\|^2$$

- The difference between the cosine similarity and using the euclidean distance is how they normalize for lengths of the vectors.

Embedding matrix

- An embedding matrix E , has dimensions $n_embeddings \times V$, where V is the size of the vocabulary.
- The i^{th} column in E is e_i , e.g. e_{123} .
- If we have a One-Hot vector o_i of size $V \times 1$, then:

$$e_i = E \cdot o_i$$

- In practice, use a specialized function to look up an embedding instead of multiplication.
- In Keras, there is an **Embedding** layer that allows retrieving the right embedding for the word more efficiently.

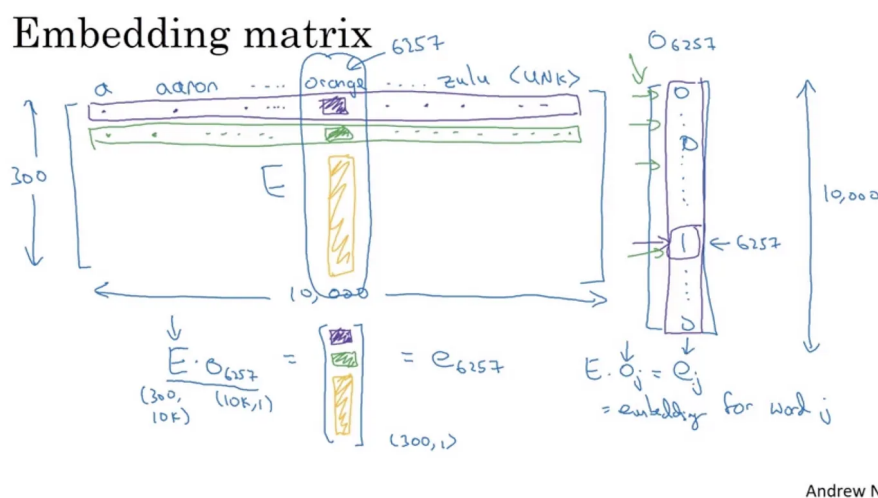


Figure 86: Embedding Matrix

How to learn word embeddings (Embedding Matrix)?

- Paper: “A neural probabilistic language model”

Neural Language Model

- The idea is to use E as a parameter matrix in a neural network.
 - The embedding from a window/**context** (a fixed number) of words is concatenated (e.g. 4 embeddings of size 300 create a vector of size 1200) and is fed to a dense layer that is subsequently fed to a Softmax layer (with the size of the vocabulary). The word embeddings are treated as parameters in that they are part of the optimization process (e.g. Gradient Descent). The output (target value) of the neural network is the *next word in a sequence*. The inputs and the output are fed as one-hot vectors.
 - Works because the algorithm tends to make words that are used together or interchangeably (in the same context) be represented closer together in the embeddings matrix since these words tend to share the same *neighborhood*.

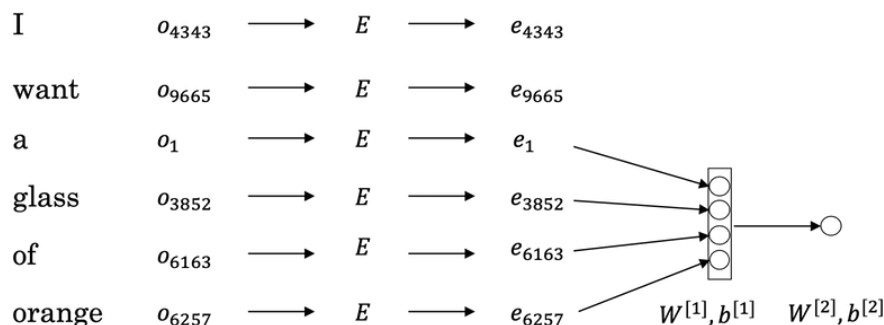


Figure 87: Neural Language Model for learning Word Embeddings

Thou shalt not make a machine in the likeness of a human mind

Sliding window across running text

thou	shalt	not	make	a	machine	in	the	...
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	

Dataset

input 1	input 2	output
thou	shalt	not
shalt	not	make
not	make	a
make	a	machine
a	machine	in

Figure 88: Sliding window technique

Different approaches for selecting the Context

- 4 words on the right
- 4 words on the right in the left (and predict the word in the middle) e.g. **CBOW** algorithm.
- The last 1 word
- Better yet, the “nearby” 1 word: **skip-grams** also works well!

5.10 Word2Vec (CBOW and Skip-Gram)

- Paper: “Efficient estimation of word representations”
- **Skip-grams**: Given a **context** word, try to estimate **target** word(s) within the neighborhood (a window) of the context word.
- **Continuous Bag-of-Words Model (CBOW)**: Predicts the middle/target word based on surrounding context words.
- Example sentence (Skip-Gram, randomly select target(s) within the neighborhood of the context (*orange*)):

I want a glass of orange juice to go along with my cereal.

Context	Target
orange	juice
orange	glass
orange	my

- Imagine a neural network to guess the target word given the context word. This is a hard problem. But the objective is not to do well on the classification problem, it is to obtain good word embeddings!
- Now we take the **context** word c (e.g. “orange”) and a **target** word (e.g. “juice”) t . (O is a one-hot representation):

$$O_c \rightarrow E \rightarrow e_c \rightarrow \text{softmax} \rightarrow \hat{y}$$

$$\text{Softmax: } p(t|c) = \frac{e^{\theta_t^\top e_c}}{\sum_{j=1}^V e^{\theta_j^\top e_c}}$$

Where θ_t is the parameters of Softmax associated with the output t , that is the chance that output t is the label. V is the vocab size.

The loss function will be:

$$\mathcal{L}(\hat{y}, y) = - \sum_{i=1}^V y_i \log(\hat{y}_i)$$

Even though there are parameters θ_t , the resulting embeddings E are still pretty good. Both context and target are fed as One-Hot encoded vectors.

- The problem is the computational cost of summation in Softmax, even for a $V=10k$ word vocabulary.
- A possible solution (in the literature) is to have a **hierarchical** Softmax classifier, that splits the words in the vocabulary into a binary tree and at each step tries to decide the branch to which the word should be long. This is actual $O(\log(V))$.
- In practice the hierarchical Softmax classifier doesn't use a balanced tree, it instead is developed so that the most common words are on top, whereas the least frequent words are deeper.

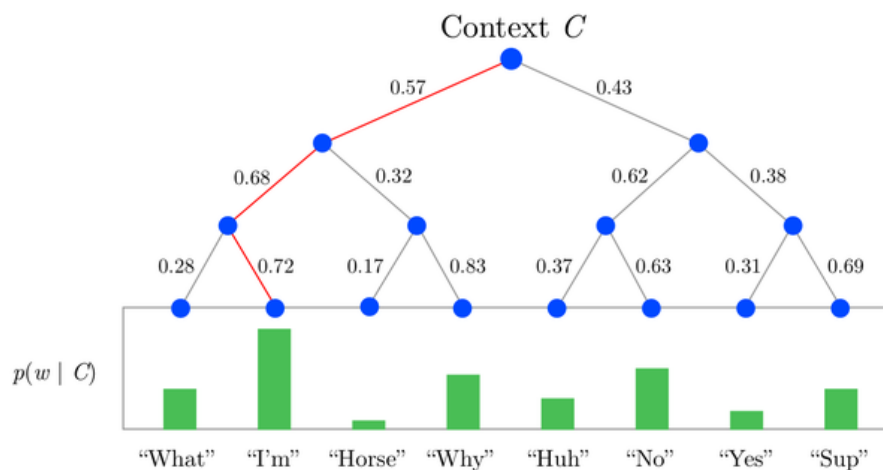


Figure 89: Hierarchical Softmax

Notes on sampling context c

- Some words are extremely frequent (the, of, a, and, to, ...) while others are less frequent (orange, apple, durian, ...)
- In practice the distribution $P(c)$ is not random, it instead must balance out between more frequent and less frequent words.

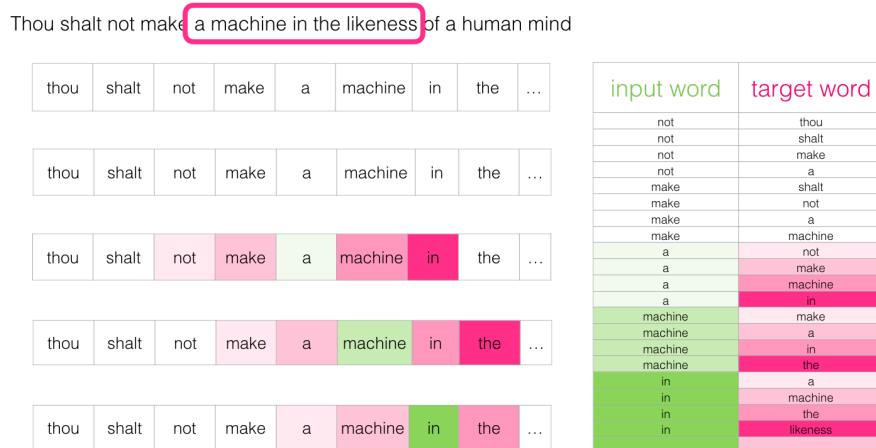


Figure 90: Sliding Window for Skip-Gram

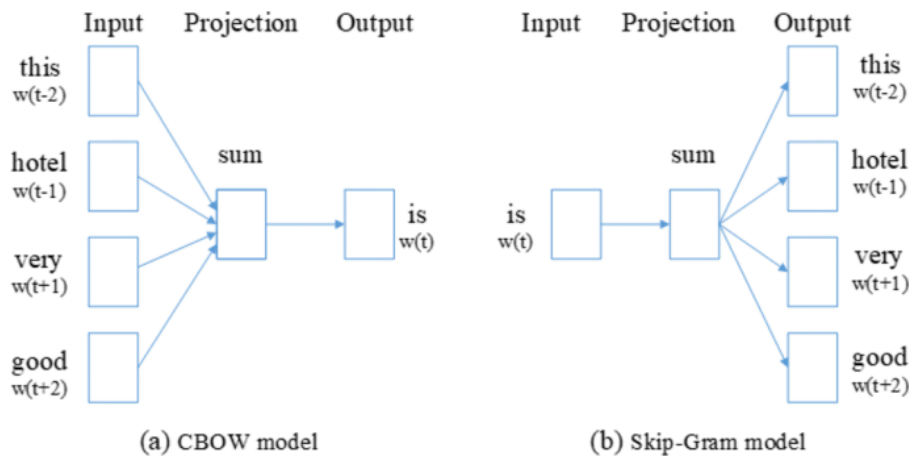


Figure 91: CBOW vs. Skip-Gram

5.11 Negative Sampling (Skip-Gram revised)

- Paper: “Distributed representation of words and phrases and their compositionality”
- Similar to skip-gram in performance but much faster.

I want a glass of orange juice to go along with my cereal.

- Switch the model’s task from predicting a neighboring word to a model that takes the input and output word and outputs a score indicating if they’re neighbors or not (0 for “not neighbors”, 1 for “neighbors”). A supervised binary classification problem.
- Similarly to skip-gram, we first choose a context word c , then choose a random target word T and set a boolean “target?” flag to one (**positive example**).
- We then choose k random words from the vocabulary and set the “target?” flag to 0 (**negative examples**). It is not a problem if by chance the random word is in the neighborhood of the context word (such as the word “of” in the table below).
 - $k \in [5, 20]$ for smaller datasets
 - $k \in [2, 5]$ for larger datasets

The result is a table as follows:

Context	T Word	Target?
orange	juice	1
orange	king	0
orange	book	0
orange	the	0
orange	of	0
...

Pick randomly from vocabulary
(random sampling)

input word	output word	target
not	thou	1
not	aaron	0
not	taco	0
not	shalt	1
not	make	1

Word	Count	Probability
aardvark		
aarhus		
aaron		
taco		
thou		
zyzzyva		

Figure 92: Negative Sampling with $k = 2$

The model uses the logistic function instead of Softmax and becomes:

$$P(y = 1|c, t) = \sigma(\theta_t^\top e_c)$$

$$O_c \rightarrow E \rightarrow e_c \rightarrow \text{logistic} \rightarrow \hat{y}$$

- This model must be thought of as training many individual logistic regression models, as many as the size of the vocabulary. V logistic regression to determine each word of the vocabulary is the target of our context or not.
- We update k (negative) + 1 (positive) logistic models in one iteration (one context word).
- Random sampling would pick up the more frequent words (of, the, and, ...) most of the time, thus under-representing less frequent words.
- Uniform sampling of each word in the vocabulary would give an even chance to each word $\frac{1}{|V|}$ (where V is the size of the vocabulary). But that would under-represent the most frequent words. To sample words as a function of their frequency ($f(w_i)$) using the special ratio parameter 3/4:

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^V f(w_j)^{3/4}}$$

5.12 GloVe (global vectors for word representation)

- Paper: “GloVe: Global vectors for word representation”
- Not as used as the Word2Vec model, but gaining momentum due to its simplicity.
- works based on the co-occurrence of words.

$$X_{ij} = \text{number of times } i \text{ appears in the context of } j$$

- Depending on how context and target word are defined it could happen that $X_{ij} = X_{ji}$

Model

$$\text{minimize } \sum_{i=1}^V \sum_{j=1}^V f(X_{ij})(\theta_i^T e_j + b_i + b_j - \log(X_{ij}))^2$$

- Where $\theta_i^T e_j$ can be seen like the $\theta_t^T e_c$ of the skip-gram model, though they are symmetrical and should be randomly initialized.
- b_i and b_j are bias terms.
- V is the size of the vocabulary.
- $f(X_{ij})$ is a weighting term, that:
 - is 0 if $X_{ij} = 0$, and we don't need to calculate $\log(X_{ij})$
 - Compensates the imbalance due to words that are more frequent (e.g. the, is, of, a,...) than others (e.g. durian).
- In this model θ_i and θ_j are symmetric, and therefore to calculate the final embedding for each word we can take the average:

$$e_w^{(final)} = \frac{e_w + \theta_w}{2}$$

5.13 Interpretability of word embeddings

- Methods used to create word an embedding matrix like Skip-Gram or GloVe do not guarantee that the embeddings are humanly interpretable. In other words, the rows of the embedding matrix will not have a humanly understandable meaning such as the rows of the motivational table that was previously used:

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1.00	1.00	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.7	0.69	0.03	-0.02
Food	0.04	0.01	0.02	0.01	0.95	0.97
...

- In algebraic terms, it is not even possible to guarantee that the rows of the embedding matrix generated by these methods are orthogonal spaces.

- For example, in the case of GloVe, it is not possible to guarantee that there is a matrix A , given GloVe's parameters θ_i and e_j , such that:

$$(A\theta_i)^\top(A^{-\top}e_j) = \theta_i^\top A^\top A^{-\top}e_j = \theta_i^\top e_j$$

Further reading on Word Embeddings:

You can find detailed tutorials about Word Embeddings on my website:

- [What are Word Embeddings and how do they work? An introduction to Word2Vec \(CBOW and Skip Gram\)](#)
- [What is Word2vec word embedding?](#)
- [Understanding Word2vec embedding with Tensorflow implementation](#)
- [Understanding GloVe embedding with Tensorflow implementation](#)

5.14 Sentiment classification

Simple sentiment classification model

$$O_c \rightarrow E \rightarrow e_c \rightarrow \text{Average} \rightarrow \text{softmax} \rightarrow \hat{y}$$

Simple sentiment classification model

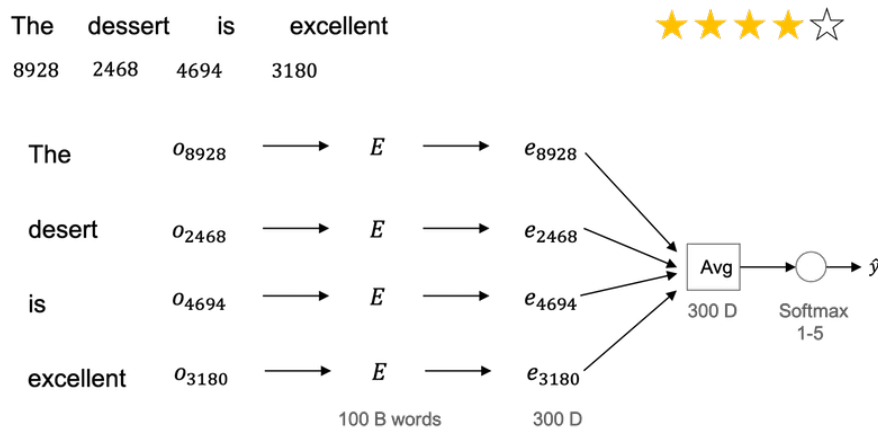


Figure 93: Simple Sentiment Classification Model

- In this model, the average is calculated for all the dimensions of the embedding vectors corresponding to a text's words, producing a single vector with the same dimensions as a single embedding vector.
- This embedding average vector is then fed to Softmax output layer with as many units as the classes that we are trying to identify (e.g. 5 for 5 star review systems).
- It completely ignores word order, thus having bad results when order matters.

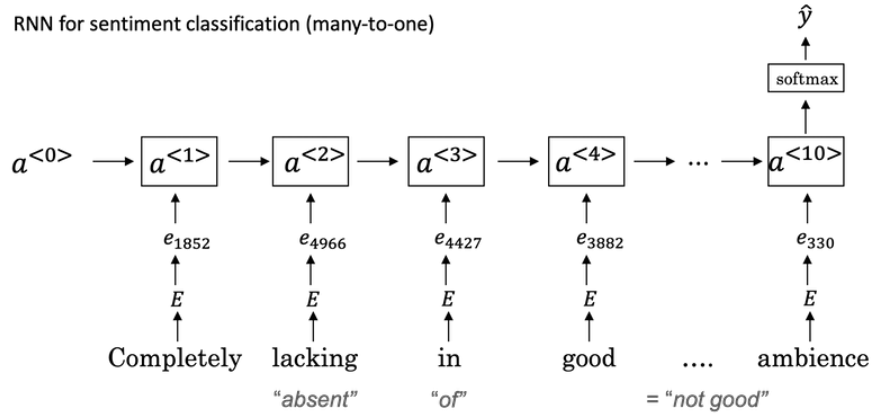


Figure 94: Sentiment Classification with RNNs

RNN for sentiment classification For each word at position t :

$$O_c^{<t>}(\text{One-Hot}) \rightarrow E^{<t>}(\text{Embedding}) \rightarrow e_c^{<t>} \rightarrow \text{RNN } a^{<t>} \rightarrow \text{softmax} \rightarrow \hat{y}$$

- We take the softmax value for the last word (the hidden state of the latest time step) as the final output.

5.15 Debiasing word embeddings

- Paper: “Man is to computer programmer as woman is to homemaker? Debiasing word embeddings”

Word embeddings can reflect gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model. e.g.:

Man:Computer_Programmer as Woman:Homemaker

Father:Doctor as Mother:Nurse

Even if the same embeddings work in the case of:

Man:Woman as King:Queen

There are some words that intrinsically capture gender: (grandmother, grandfather), (girl, boy), and (she, he) are gender intrinsic in the definition. There are other words like doctor and babysitter that we want to be gender-neutral.

Steps to address the bias problem in word embeddings

1. Identify the bias direction

- For example for gender, take a few gender embedding pairs for *definitional* words as:

$$e_{he} - e_{she}$$

$$e_{male} - e_{female}$$

$$e_{grandfather} - e_{grandmother}$$

$$e_{...} - e_{...}$$

- Take the average of such embeddings, to identify the bias direction (in practice this is done with **SVD - Single Value Decomposition**, since the bias direction can be multidimensional)

- The Non-bias direction is orthogonal to the bias direction.
2. Neutralize: For every word that is not definitional (e.g. not like grandmother and grandfather), project (in the non-bias axis) to get rid of bias. See Figure (95).

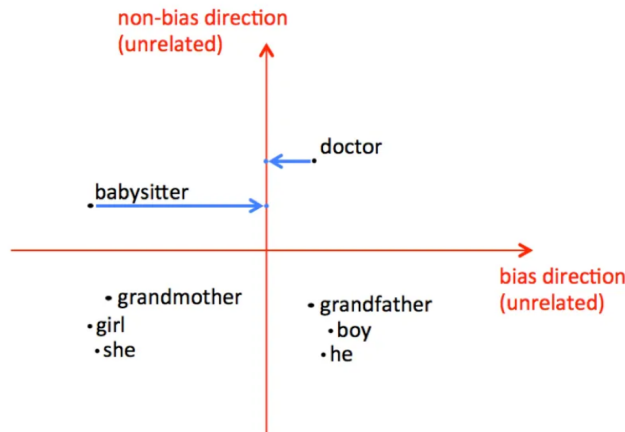


Figure 95: Neutralizing step - Addressing Bias

3. Equalize pairs - For example, after projecting the (non-definitional word) “babysitter” in the non-bias axis, move “grandmother” and “grandfather” so that they have the same distance from the non-bias and bias axis, hence having the same distance from the word “babysitter”. See Figure (96).

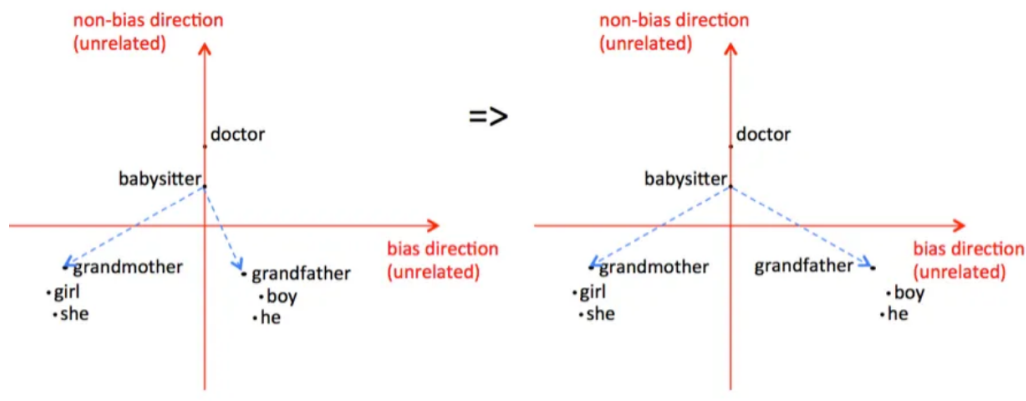


Figure 96: Equalizing pairs - Addressing Bias

Challenge: What words are definitional? Most words are not definitional for a specific bias. It is possible to train a classifier to identify words that are definitional for a specific bias (e.g. grandmother/grandfather for gender).

5.16 Basic Sequence Models

Sequence to sequence models

- Paper: “Sequence to sequence learning with neural networks”

- Paper: “Learning phrase representations using RNN encoder-decoder for statistical machine translation”

Example: Machine translation

encoder RNN (in: sentence in language A) \rightarrow decoder RNN (out: sentence in language B)

Sequence to sequence model

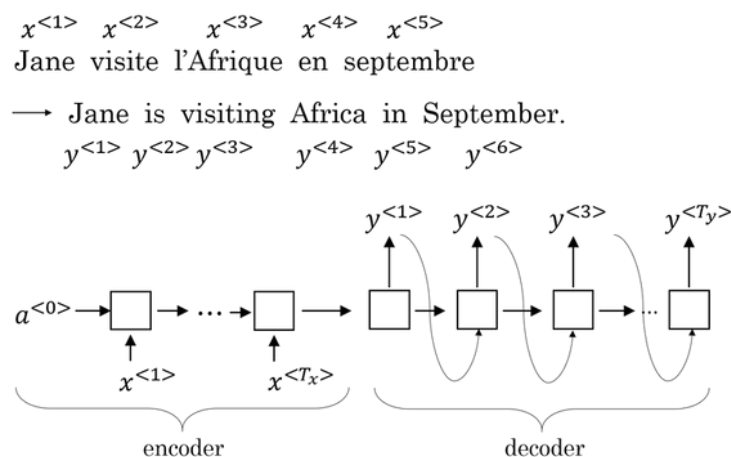


Figure 97: Machine Translation

Example: Image captioning

Conv Net (in: cat picture) \rightarrow decoder RNN (out: image description)

Image captioning

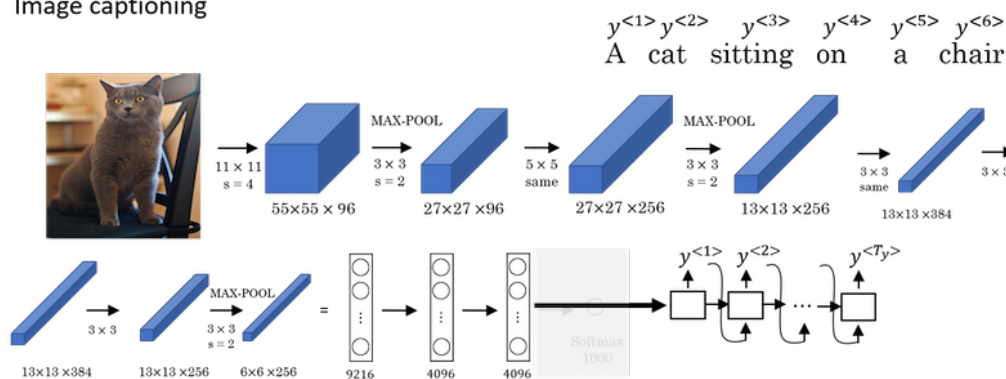


Figure 98: Image Captioning

Picking the most likely sentence

- The goals of a Language Model were: 1) To calculate the probability of a sequence $P(\hat{y}^1, \dots, \hat{y}^{<T_y>})$ and 2) Generate a novel sequence by sampling.
- **Machine Translation example:** We are trying to translate from French (A) to English (B) with the model:

encoder RNN (in language A) \rightarrow decoder RNN (out language B)

- The output of the encoder module is a vector which is the input of the decoder ($a^{<0>}$) of the English language model.
- The resulting model is called a **conditional language model**, which **maximizes** the conditional probability of an English sentence given a French sentence:

$$\operatorname{argmax}_{y^{<1>}, \dots, y^{<T_y>}} P(y^{<1>}, \dots, y^{<T_y>} | x^{<1>}, \dots, x^{<T_x>})$$

- The goal of Machine Translation is to find a sequence (a sentence in English) that maximizes the above conditional probability.

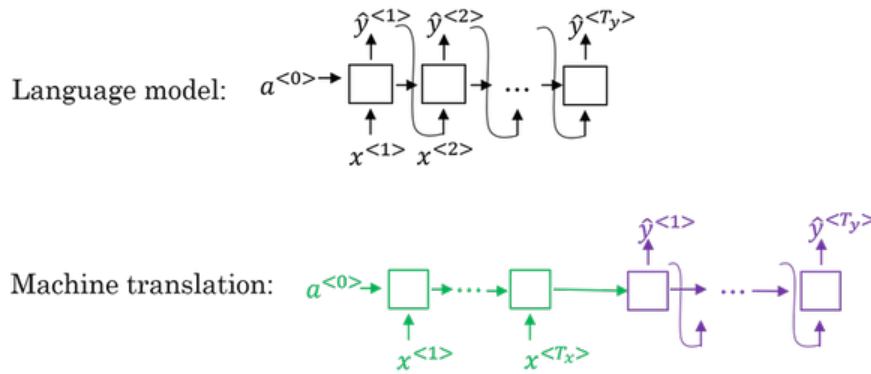


Figure 99: Language Model vs. Machine Translation

- Why not just use “greedy search”, i.e. always chose the highest probability word on the decoder stage?
 - Because the sequence of all highest probability words is not necessarily the optimal sentence to maximize $P(y^{<1>}, \dots, y^{<T_y>} | x^{<1>}, \dots, x^{<T_x>})$.
- Moreover, the total number of English words in a sentence is exponentially large. It is a huge space to check all combinations of words, which is why **approximate search** algorithms are used to try to find the sentence that maximizes P .

5.17 Beam search algorithm

1. Feed the sentence in the source language (x) to the encoder, then, on the decoder module, take the most likely first B words $P(\hat{y}^{<1>} | x)$.
2. For each of the first B words selected, feed $\hat{y}^{<1>}$ as the input for the next stage $t = 2$ in order to get $\hat{y}^{<2>}$, and calculate the top B combinations of $\hat{y}^{<1>}$ and $\hat{y}^{<2>}$ that maximize:

$$P(\hat{y}^{<1>}, \hat{y}^{<2>} | x) = P(\hat{y}^{<1>} | x) P(\hat{y}^{<2>} | x, \hat{y}^{<1>})$$

3. So now we get 3 combinations of $\hat{y}^{<1>}$ and $\hat{y}^{<2>}$, and for each of these combinations we want to find the top B combinations of $\hat{y}^{<1>}$, $\hat{y}^{<2>}$ and $\hat{y}^{<3>}$ that maximize $P(\hat{y}^{<1>}, \hat{y}^{<2>}, \hat{y}^{<3>} | x)$.
4. Repeat the step above until we get the top B combinations that maximize $P(\hat{y}^{<1>}, \dots, \hat{y}^{<t>} | x)$, and finally, select only the most likely (since we now have the entire sequence).

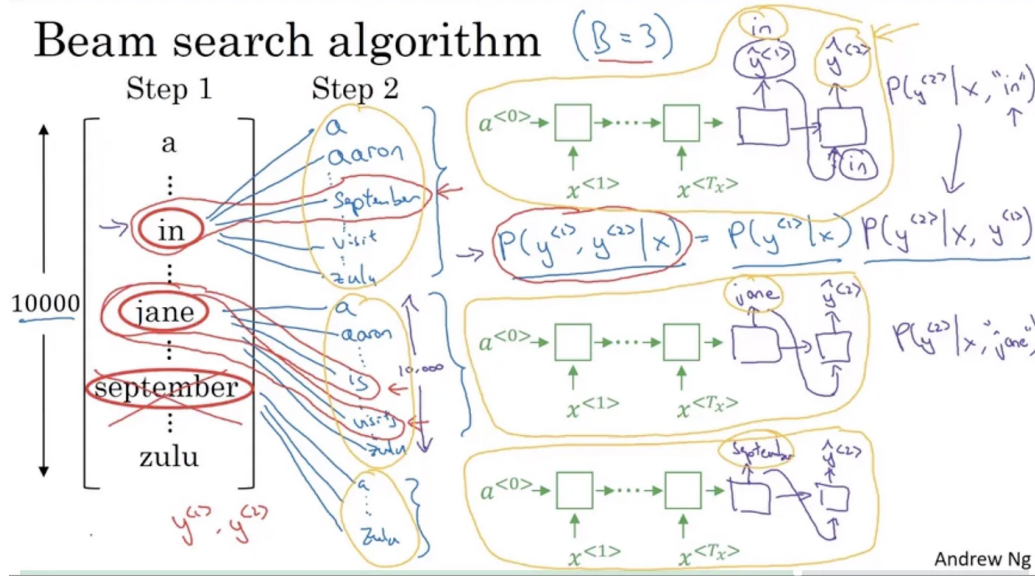


Figure 100: Beam Search

- This algorithm implies creating B copies of the network at each stage.
- If $B = 1$ then Beam search becomes a greedy algorithm, but with $B > 1$ the algorithm finds better results.
- You can find a detailed example of Beam Search at the following link:

[Example of Beam search in Sequence to Sequence models](#)

Refinements to Beam search Generalizing the conditional probability used by the Beam search algorithm:

$$\arg \max_y \prod_{t=1}^{T_y} P(y^{<t>} | x, y^{<1>}, \dots, y^{<t-1>})$$

In practice, using the sum of log we get the same result, with more numerical stability (less chance of overflow):

$$\arg \max_y \sum_{t=1}^{T_y} \log P(y^{<t>} | x, y^{<1>}, \dots, y^{<t-1>})$$

Length normalization: Since $P(.) < 1$, the $\log P(.) < 0$, then the target function has the problem that long sentences have low probabilities, so unnaturally short sentences are preferred. To solve this one thing to do is to normalize by the number of words:

$$\arg \max_y \frac{1}{T_y^\alpha} \sum_{t=1}^{T_y} \log P(y^{<t>} | x, y^{<1>}, \dots, y^{<t-1>})$$

where α is usually set to 0.7 based on the heuristic.

Choosing Beam width B

- Large B : better results, slower (high memory usage)
- Small B : worse results, faster (less memory usage)
- Typical values of B are 10 for production. 100 would be very large for production, but in research settings, usually, values of 1000 to 3000 are used.
- Increasing B pretty much never hurts the performance.

Unlike exact search algorithms like BFS (Breadth First Search) or DFS (Depth First Search), Beam Search runs faster but is not guaranteed to find the exact maximum for $\arg \max_y P(y|x)$.

Error analysis on Beam search Given an input sequence x ('Jane visite l'Afrique en septembre'), we feed both the output of translated sentence produced by the model \hat{y} ('Jane visits Africa in September') and an optimal human-translated version y^* ('Jane visited Africa last September') to the RNN and calculate $P(\hat{y}|x)$ and $P(y^*|x)$. If it is found that the machine translation is wrong (in a meaningful way), we can try to find if the cause is either the RNN or Beam Search algorithm.

- Case 1: $P(y^*|x) > P(\hat{y}|x)$
 - Conclusion: Beam search is at fault because it could not find the best sequence of words (y) that maximize $P(y|x)$. It chose \hat{y} while the y^* actually attains a much bigger value. We could try to increase B in this case.
- Case 2: $P(y^*|x) \leq P(\hat{y}|x)$
 - Conclusion: RNN model is at fault because y^* is a better translation than \hat{y} , however, according to the RNN, $P(y^*|x)$ is less than $P(\hat{y}|x)$.

We could try to perform an error analysis (looking at a batch of examples with bad translation) to try to find which of these problems is more prevalent, in order to decide what we should spend our time troubleshooting.

5.18 Bleu Score

- A metric to measure the accuracy of a translated sentence according to different ground truth translations of the same sentence.
- Paper: "Bleu: A method for automatic evaluation of machine translation"

- Bleu stands for “Bilingual Evaluation Understudy”. In the theater world “understudy” is someone who learns and can take the role of a more senior actor if necessary. In machine translation, it means that we find a machine-generated score that can replace human-generated translation references.

Given reference human translations R_1, \dots, R_n , and the machine translation MT for a given text/sentence, the Bleu score can be generally formalized as P^n , or “Precision” for “n-grams” (unigrams, bigrams, etc.) as:

$$P_n = \frac{\sum_{\text{n-grams} \in \hat{y}} \text{Count}_{\text{clip}}(\text{n-gram})}{\sum_{\text{n-grams} \in \hat{y}} \text{Count}(\text{n-gram})}$$

Where:

- $\text{Count}_{\text{clip}}$: The maximum number of times that an n-gram is presented in the machine translation MT appears in any of the reference translations R_1, \dots, R_n .
- Count : The number of times a specific n-gram is present in the machine translation MT .

Example:

French sentence: ‘Le chat est sur le tapis.’

R_1 : ‘The cat is on the mat.’

R_2 : ‘There is a cat on the mat.’

MT : ‘The cat the cat on the mat.’

Bigram	Count	Count _{clip}
the cat	2	1
cat the	1	0
cat on	1	1
on the	1	1
the mat	1	1

$$P_2 = \frac{4}{6}$$

Bleu details:

P_n = Bleu score on n-grams only

Combined Bleu score (using up to n-grams with $n = 4$):

$$\text{BP} \exp\left(\frac{1}{4} \sum_{n=1}^4 P_n\right)$$

Where BP is the **brevity penalty**, which penalizes translations that are shorter than the original:

$$\text{BP} = \begin{cases} 1 & \text{if MT_output_length} < \text{reference_output_length} \\ \exp\left(1 - \frac{\text{MT_output_length}}{\text{reference_output_length}}\right) & \text{otherwise} \end{cases}$$

5.19 Attention Model

- Paper: “Neural machine translation by jointly learning to align and translate.”

- With traditional models, the Bleu score tends to fall within the length of sequences.

Consider the French sentence:

“Jane s’est rendue en Afrique en septembre dernier, a apprécié la culture et a rencontré beaucoup de gens merveilleux; elle est revenue en parlant comment son voyage était merveilleux, et elle me tente d’y aller aussi.”

and its English translation:

“Jane went to Africa last September, and enjoyed the culture and met many wonderful people; she came back raving about how wonderful her trip was, and is tempting me to go too.”

- The way a human translator would translate this sentence is not to first read the whole French sentence and then memorize the whole thing and then regurgitate an English sentence from scratch (this is what happens in an encoder-decoder model). Instead, what the human translator would do is read the first part of it, maybe generate part of the translation, look at the second part, generate a few more words, look at a few more words, generate a few more words, and so on.
- The Encoder-Decoder architecture works quite well for short sentences, so we might achieve a relatively high Bleu score, but for very long sentences, maybe longer than 30 or 40 words, the performance comes down. By translating long sentences in small chunks (like humans tend to do) it is possible to retain a high Bleu score independently of the size of the sentence.

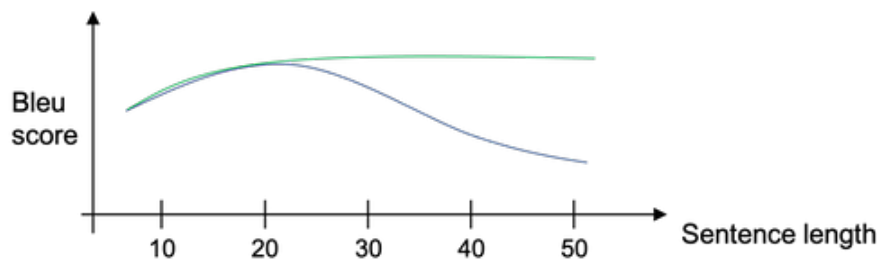


Figure 101: Bleu score drops for long sentences. Blue line: MT, Green line: Attention model

BRNNs (Bidirectional RNN) are normally used for machine translation where **Attention Models** are usually employed. With that in mind:

- Timesteps in the encoder section are referred to by t' .
- Timesteps in the decoder section are referred to by t .
- The two sets of activations of the encoder section are $a^{<t'>} = (\vec{a}^{<t'>}, \overleftarrow{a}^{<t'>})$.
- Attention weights: $\alpha^{<t,t'>} =$ the amount of “attention” to t'^{th} input of encoder when generating the t^{th} output of decoder $y^{<t>}$. For example, $\alpha^{<1,1>}$ denotes when you’re generating the first words, how much should you be paying attention to the first piece of information in the input. $\alpha^{<1,2>}$ which tells us when we are trying to compute the first word of Jane, how much attention we are paying to the second word from the input, and so on. See Figure (102).
- The input $C^{<t>}$ (or context) to each decoder unit/timestep is a weighted average of the “attention” and the outputs of the encoder units, thus allowing each decoder output to take into consideration of the closest set of words to the translated word being generated:

$$C^{<t>} = \sum_{t'} \alpha^{<t,t'>} a^{<t'>} \text{ where } \sum_{t'} \alpha^{<t,t'>} = 1$$

The $\alpha^{<t,t'>}$ parameter is in turn the application of a Softmax function (add up to one) to the exponentiation of another parameter $e^{<t,t'>}$:

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} \exp(e^{<t,t'>})}$$

$\alpha^{<t,t'>}$ and $e^{<t,t'>}$ depend on $s^{<t-1>}$ (hidden state activation from the previous time step in the decoder) and $a^{<t'>}$ (features from time step t'). But we don't know what the function is. So one thing you could do is just train a very small neural network to learn whatever this function should be. And use backpropagation and gradient descent to learn the right function. Therefore, $e^{<t,t'>}$ is a parameter learned by a simple neural network (e.g. 1 dense layer) with inputs $S^{<t-1>}$ (the previous decoder state) and $a^{<t'>}$. See Figure (104).

- The downside of this algorithm is that it runs in quadratic time, that is $T_x \times T_y$, which is asymptotically $O(n^2)$. This might be an acceptable cost, assuming that sentences are not that long.
- This technique has also been applied to other problems such as **image captioning**: Paper: “Show, attend, and tell: Neural image caption generation with visual attention”
- You can find more detailed tutorials about Attention Mechanism on my website:
 - [An illustrated guide to Attention Mechanism in Sequence Models with PyTorch code](#)
 - [Understanding Attention Mechanism in Sequence 2 Sequence Machine Translation](#)
 - [Understanding Attention Mechanism with example](#)
 - [Implementing Attention Mechanism in Python](#)
 - [Bahdanau and Luong Attention Mechanisms explained](#)

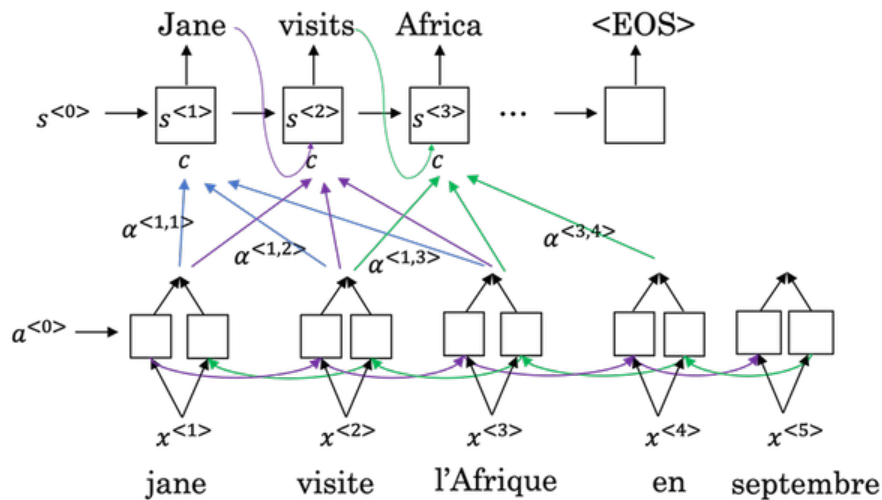


Figure 102: Attention Mechanism

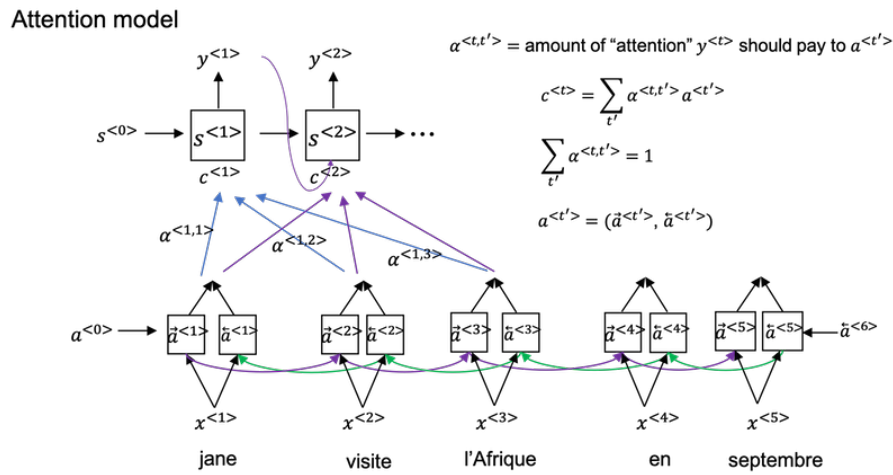


Figure 103: Attention Model

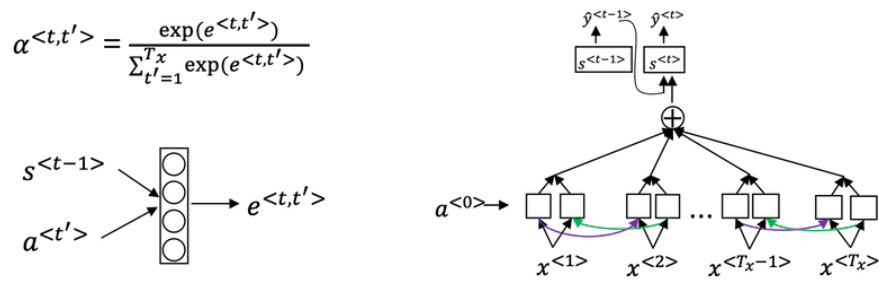


Figure 104: Attention Weights

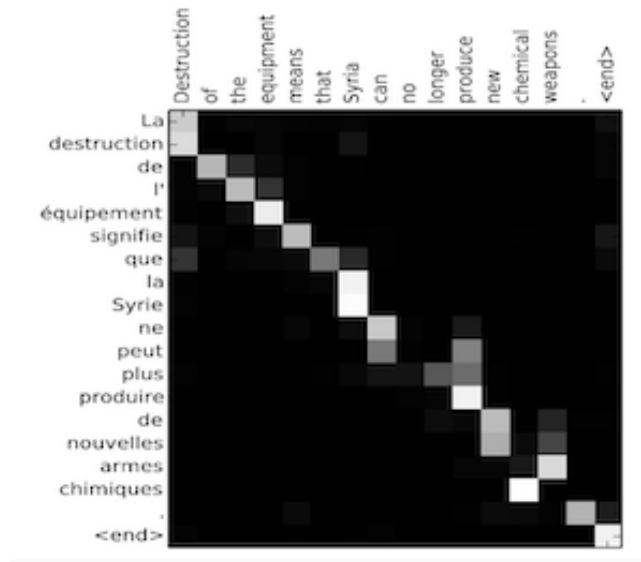


Figure 105: Visualization of attention weights

5.20 Speech recognition

- Problem: input(x): Audio clip, output(y): Transcript of the audio
- Time domain and spectral features.

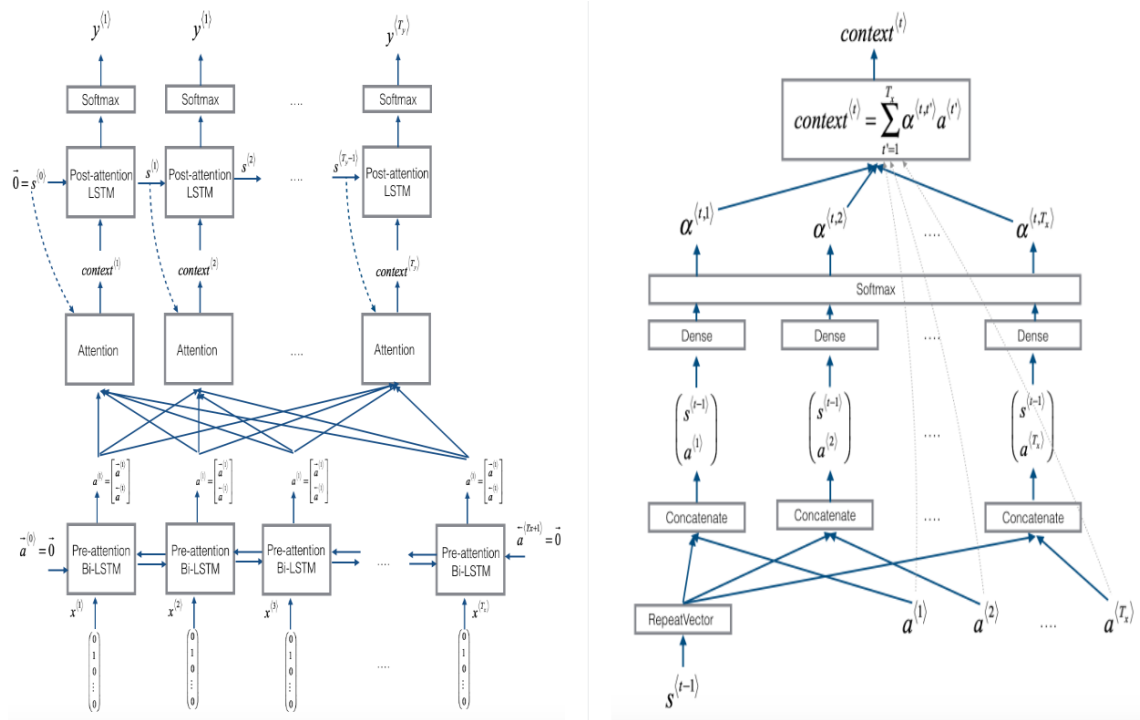


Figure 106: Example of an attention model (left) and calculation of the attention weights in a single time step (right)

Symbol	description	calculation
$a^{<t'>}$	feature for the decoder network at chunk-timestep t'	$a^{<t'>} = (\vec{a}^{<t'>}, \overleftarrow{a}^{<t'>})$
$\alpha^{<t,t'>}$	amount of attention $\hat{y}^{<t>}$ should pay to chunk-feature $a^{<t'>}$ at time step t ($\sum_{t'} \alpha^{<t,t'>} = 1$)	$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} \exp(e^{<t,t'>})}$
$c^{<t>}$	context for the decoder network at time step t	$c^{<t>} = \sum_{t'} \alpha^{<t,t'>} a^{<t'>}$
$\hat{y}^{<t>}$	prediction of decoder network at time step t	

Figure 107: Attention Summary

- With deep learning phonemes are no longer needed!
- Commercial systems are now trained from 10k to 100k hours of audio!
- The attention model is applicable.

CTC cost for speech recognition

- Paper: “Connectionist Temporal Classification: Labeling unsegmented sequence data with recurrent neural networks”
- The first part of an RNN for speech recognition (normally a BRNN), uses a large number of input features and also possibly has a relatively large resolution in time, meaning that many data points are generated per time unit. Notice that the number of time steps here is very

Attention model for speech recognition

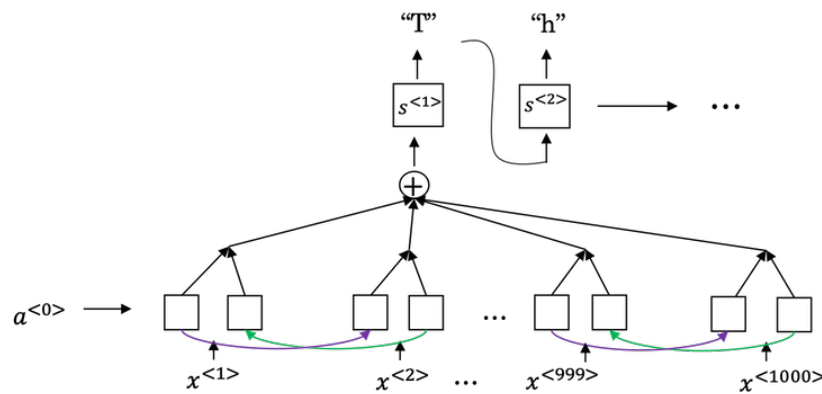


Figure 108: Attention for speech recognition

large and in speech recognition, usually, the number of input time steps is much bigger than the number of output time steps. For example, if you have 10 seconds of audio and your features come at 100 hertz so 100 samples per second, then a 10-second audio clip would end up with a thousand inputs. But your output might not have a thousand alphabets, might not have a thousand characters. The CTC cost function allows the RNN to generate a valid output like:

ttt_h_eee_ _ _qqq_ _ii...

- Underscores are “blank” characters, not “spaces”.
- The basic rule for the CTC cost function is to collapse repeated characters not separated by “blank”.

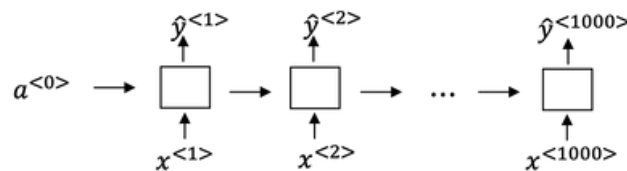


Figure 109: Speech recognition

Trigger Word Detection

- Examples: “Okay Google”, “Alexa”, “xaudunihao” (Baidu DuerOS), “Hey Siri”

One possible solution:

- Create an RNN that inputs sound spectral features, and always outputs 0 except when it has detected the end of a trigger word in which case it outputs 1.
 - Problem: it creates an imbalanced training set, with a lot more 0s than 1s. Thus, “accuracy” is not a good performance metric in this case because a classifier that always returns 0 can have more than 90% accuracy. F-score, precision, and recall are more appropriate.
 - One possible hack is to make it output a series of 1s after the trigger word, not just one 1, though that doesn’t solve the problem definitively.

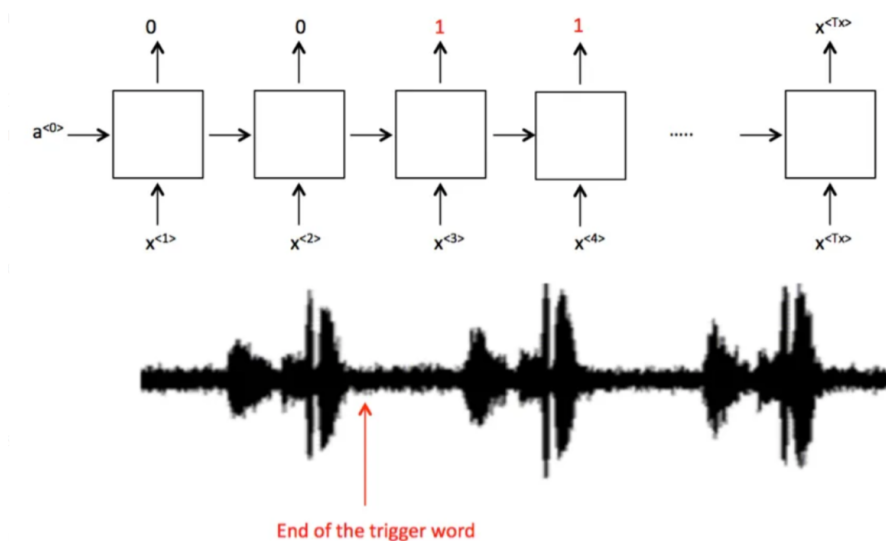


Figure 110: Trigger word detection labeling

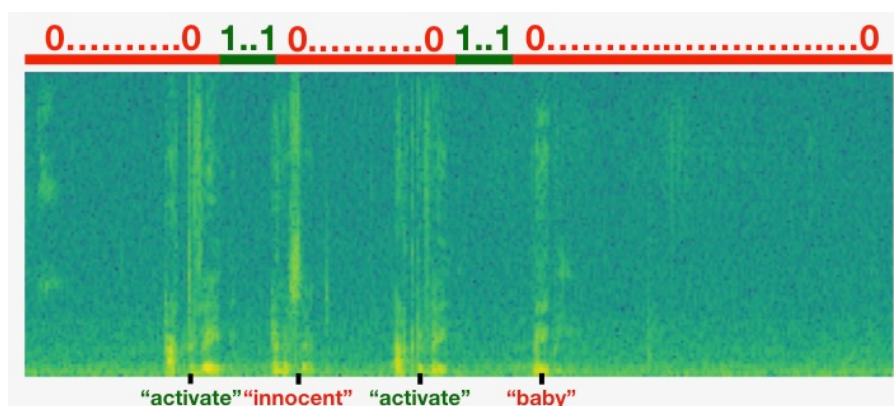


Figure 111: Trigger word detection spectrogram

References

- [1] P. Grover. Evolution of Object Detection and Localization Algorithms. <https://towardsdatascience.com/evolution-of-object-detection-and-localization-algorithms-e241021d8bad>.
- [2] IndoML. Student Notes: Convolutional Neural Networks (CNN) Introduction. <https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/>.
- [3] S. Zivkovic. Language Modelling and Sampling Novel Sequences. <https://datahacker.rs/004-rnn-language-modelling-and-sampling-novel-sequences/>.