ARTIFICIAL NEURAL NETWORKS AND DEEP LEARNING

Lectures from 2022-2023 Course

Notes on Neural Networks by the Neural Network ChatGPT and Human Beatrice Insalata

Introduction to Deep Learning

Machine Learning is a category of research and algorithms focused on finding patterns in data and using those patterns to make predictions. Machine learning falls under the AI umbrella, which in turn intersects the broader field of knowledge discovery and data mining.

Machine learning algorithms can be broadly categorized as unsupervised or supervised by what kind of experience they are allowed to have during the learning process.
Most of the learning algorithms can be understood as being allowed to experience an entire dataset. A dataset is a collection of many examples, collections of features that have been measure from some object we want the system to process.

- **UNSUPERVISED LEARNING ALGORITHMS**: experience a dataset containing many features, then learn useful properties of the structure of this dataset. They exploit dataset regularities to build a meaningful and compact representation. In the context of deep learning, we usually want to learn the entire probability distribution that generated a dataset. It includes algorithms like **clustering**, which divides the D in cluster of similar examples, and **anomaly detection**, through which the computer flags some element as unusual. One traditional example can be K-Means clustering.
- **SUPERVISED LEARNING ALGORITHMS:** given a training set of pairs (input, desired output)
  $x_1, t_1, \ldots, (x_N, t_N)$, learn to produce the correct output of new inputs. Each example is associated to a label. The most famous example is **classification**, in which the computer program is asked to specify which of k categories some input belongs to. To solve this task, the learning algorithm is usually asked to produce a function $f : \mathbb{R}^n \rightarrow 1, ..., k$. When
  $y = f(x)$, the model assigns an input described by vector x to a category identified by numeric code y. Also, **regression** is used to predict a numerical value given some input. To solve this task, the learning algorithm is asked to output a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. This type of task is similar to classification, except that the format of output is different. One example can be logistic regression, which uses maximum likelihood estimation to predict the parameters of a logistic function.

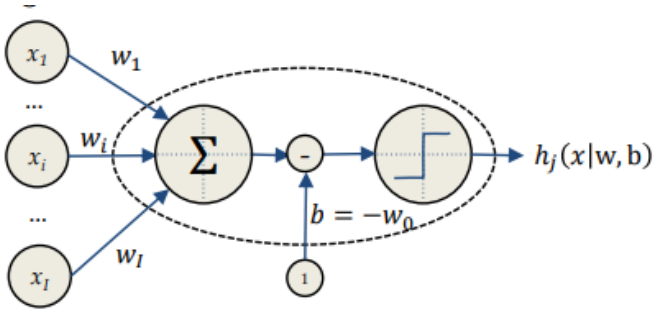
- **REINFORCEMENT LEARNING:** producing actions which affect the environment, and receiving rewards learn to act in order to maximize rewards in the long term, so there is a feedback loop between the learning system and the experience.

It can be very difficult to extract such high-level, abstract features from raw data. Many of these factors of variation can be identified only using sophisticated, nearly human-level understanding of the data. Deep learning solves this central problem by introducing representations that are expressed in terms of other, simpler representations.
Deep learning allows the computer to build complex concepts out of simpler ones. Deep Learning exploits a set of nonlinear cascaded units to extract characteristics and identify multiple levels of abstractions. These features are extracted manually in machine learning while they're found automatically in deep learning.

NEURAL NETWORKS - THE PERCEPTRON

The basic structure behind modern neural networks for deep learning is the perceptron.

A perceptron is a type of artificial neural network that was developed in the 1950s. It is a simple model that is used for binary classification tasks, meaning it can classify input data as either belonging to one class or another. A perceptron consists of a single layer of artificial neurons, which are simple processing units that take in input, process it, and produce an output. The input to a perceptron is a vector of numerical values, and the output is a scalar. The perceptron processes the input by multiplying each input value by a weight and then summing the results, adding a bias factor. This computation is then sent to an output function, which maps the output to binary.



$$h_j(x|\mathrm{w},\mathrm{b}) = \mathrm{h_j}\left(\Sigma_{i=1}^{I} w_i \cdot x_i - b\right) = \mathrm{h_j}\left(\Sigma_{i=0}^{I} w_i \cdot x_i\right) = h_j(w^T x)$$

The basic structure of a perceptron can be seen as:

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i + b > 0 \\ -1 & \text{otherwise} \end{cases}$$

In this equation, $x_1, x_2, ..., x_n$ are the input features, $w_1, w_2, ..., w_n$ are the weights assigned to each input feature, b is the bias term, and y is the output of the perceptron. The perceptron calculates the dot product of the weights and inputs, adds the bias term, and then applies the activation function (which in this case is a step function). If the result is greater than zero, the output is 1, and if it is less than or equal to zero, the output is -1.

Perceptrons are very useful when executing simple tasks that regard finding linear relationships among data.

How are the weighs found and initialized? **Hebbian Learning** introduced some useful insights:

“The strength of a synapse increases according to the simultaneous activation of the relative input and the desired target”

Starting from a random initialization, the weights are fixed one sample at the time (online) and only if the sample is not correctly predicted. The weight of the connection between A and B neurons is calculated using:

$$\begin{cases} w_i^{k+1} = w_i^k + \Delta w_i^k \\ \Delta w_i^k = \eta \cdot x_i^k \cdot t^k \end{cases}$$

where:

- η is the learning rate;
- *x* is the input of neuron A at time *k;*
- *t* is the desired output of neuron B at time *k.*

The reason this is effective is because it allows the neural network to learn through experience. When two neurons are active at the same time, it is likely that they are both contributing to the output of the network. By strengthening the connection between

them, the network is able to more effectively pass information between these neurons and make better predictions.

There could be multiple but correct solutions. This algorithm can not converge because the soliution does not exist, or η is too large and the computational steps too broad. Unfortunately, Hebbian learning does not work in finding nonlinear boundaries. To examine more complex data relationships, we need a different model.

MULTI-LAYER PERCEPTRONS - FFNN

Deep feedforward networks, also often called feedforward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models.
The goal of a feedforward network is to approximate some function f^*, which represents some kind of pattern in data, to make decisions from it. These models are called feedforward because information flows through the layers without forming cycles and in only one direction, without keeping any memory of the previous inputs.

Feedforward neural networks y are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together. For example, we might have three functions $f(1)$, $f(2)$, and $f(3)$ connected in a chain, to form $f(x) = f(3)(f(2)(f(1)(x)))$. These chain structures are the most commonly used structures of neural networks. In this case, $f(1)$ is called the first layer of the network, $f(2)$ is called the second layer, and so on. The overall length of the chain gives the depth of the model. Because the training data does not show the desired output for each of these layers, these layers are called hidden layers.

The final layer of a feedforward network is called the output layer. During neural network training, we drive $f(x)$ to match $f^*(x)$.

Every node, except for the input ones, are neurons, that elaborate data and use an activation function. It uses backpropagation to train the net.

Layers are connected through weighs, and the output of a neuron only depends on those of the previous layers. The activation functions of this new structure must be differentiable in order to be trained.

The structure is as it follows:

$$z_i = f(\sum_{j=1}^{n} w_{ij} x_j + b_i)$$

In a feedforward neural network, the computation of each layer is performed using this equation, where $x_1, x_2, ..., x_n$ are the input features, w_{ij} is the weight of the connection between the input j and the output i, bi is the bias term for the output i, and z_i is the output of the layer for the output i.

Activation Functions

The output of a node in an ANN is a linear combination of the input data and the weights of the connections between the nodes. Without an activation function, the output of the node would be a linear function of the input, which would limit the capabilities of the network to learn and model complex patterns and relationships.

By introducing an activation function, the output of the node becomes a nonlinear function of the input, which allows the network to learn and model informations like hierarchical representations. The specific type of activation function used can have a significant impact on the network's ability to learn and model different types of patterns and relationships.

- **ReLu**, the positive part of its argument, defined as:

$$f(x) = x+ = ReLU(x) = max(0, x)$$

where x is a neuron’s input. Applying this function to the output of a linear transformation yields a nonlinear one, but very close to a linear. Useful for regression tasks. It has the advantage of:

1. Faster SGD convergence;+
2. Sparse activation;
3. Efficient gradient propagation (no vanishing or exploding gradient issues);
4. Scale invariant;

But it can cause some issues:

1. Non differentiable at 0;

2. Non zero centered output;
3. Unbounded, could blow up;
4. Dying neurons, inactive because the derivative is always 0 and weights are not updated anymore;

Leaky ReLu was designed to avoid some of these problems (dying ReLu).

- **Sigmoid**, maps the input values to the range between 0 and 1, which is useful for binary classification tasks. If the output of the sigmoid function is greater than 0.5, it is considered "activated," and the node produces an output of 1. If the output is less than 0.5, it is considered "deactivated," and the node produces an output of 0.

$$f(x) = \frac{1}{1 + e^{-x}}$$

- **Tanh**, for the same binary mapping but between -1 and 1.

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- **Softmax**, used commonly in classification tasks, can map the input to a vector of values between 0 and 1, which sum adds up to 1. It is useful to output probability distribution.

$$p_i = \frac{e^{a_i}}{\sum_{j=1}^{n} e^{a_j}}$$

In this formula, $a_1, a_2, ..., a_n$ are the raw activations or scores, and $p_1, p_2, ..., p_n$ are the resulting probabilities. The exponentiation of each activation (e^{a_i}) ensures that all the probabilities are positive, and the normalization factor ($\sum e^{a_i}$) ensures that the probabilities sum to 1.

Universal Approximation Theorem

The universal approximation theorem states that a feedforward neural network (FFNN) with a single hidden layer and a sufficient number of nodes can approximate any continuous function to an arbitrary degree of accuracy. This means that, in theory, an

FFNN with a single hidden layer is capable of learning and modeling any continuous function, given enough data and computing power.

Regardless the function we are learning, a single layer can represent it:

- Doesn’t mean a learning algorithm can find the necessary weights!
- In the worse case, an exponential number of hidden units may be required.
- The layer may have to be unfeasibly large and may fail to learn and generalize.

Optimization and Learning

Optimization is an important process in the training of feedforward neural networks (FFNNs). It involves adjusting the weights of the connections between the nodes in the network to minimize the error between the predicted output of the network and the actual output. Every network has to be evaluated through an error function, which determines its goodness. This error function has to be differentiable by its weights; in fact, we can use the gradient to find the maximum direction of function growth and, with that, reduce the error.

MAXIMUM LIKELIHOOD ESTIMATION

Maximum likelihood estimation (MLE) is a method of estimating the parameters of a statistical model by maximizing the likelihood function. In the context of feedforward neural networks (FFNNs), MLE can be used to estimate the weights and biases of the network based on a set of training data.

To use MLE with an FFNN, we can define the likelihood function as the probability of the training data given the weights and biases of the network. The likelihood function is then maximized by adjusting the weights and biases in a way that increases the probability of the training data.

For example, suppose we have a neural network with a single weight, w, and we are using the mean squared error (MSE) as the error function. The MSE is calculated as the average of the squared differences between the predicted and actual values for a set of data points.

We can define the likelihood function as follows:

$$L(w) = \prod_{i=1}^{N} p(y_i|x_i, w)$$

where N is the number of data points, y_i is the true output for the ith data point, and x_i is the input for the ith data point. The function $p(y_i|x_i, w)$ represents the probability of

the true output given the input and the weights.

To maximize the likelihood function, we can take the derivative of the likelihood function with respect to the weight w and set it equal to 0:

$$\frac{\partial L(w)}{\partial w} = 0$$

Solving this equation for w gives us the maximum likelihood estimate of the weight.

MLE for Regression

Regression is used to analyze a series of data to learn the input-output relationship between a dependent variable and one or more independent variables. The learned relationship is the output's (a casual variable) value distribution given the input set.

In the context of regression, MLE can be used to estimate the parameters of a model that predicts a continuous output variable based on one or more input variables.

To use MLE for regression, we need to specify a model with a set of parameters that we want to estimate. The model should be a function that describes the relationship between the input and output variables. The likelihood function is then defined as the probability of the observed output values given the model and its parameters.

In general, if we think of the distribution of the output probability as a gaussian with known variance σ^2 and mean $g(x_n|w)$, where this is the function of the model, the likelihood for a regression model can be defined as follows:

$$L(w) = \prod_{i=1}^{N} p(t_i|g(x_i|w), \sigma^2)$$

where w is the set of model parameters and $p(t_i|g(x_i|w), \sigma^2)$ is the probability of the true output given the input and the parameters. This formula can be maximized using the **sum of squared errors**:

$$argmin_w(\sum_{i=1}^{N}(t_i - g(x_i|w))^2)$$

MLE for Classification

Classification is used to find a representation of some characteristics of an entity and assign a label to it. This is done using statistics, outputting a probability for each class. In this context, using the MSE we can start from a Bernoulli distribution for binary problems, then find the likelihood of the model and obtain **binary cross-entropy** as a useful error function:

$$argmin_x(-\sum_{n=1}^{N} t_n log(g(x_n|w)) + (i - t_n) log(1 - g(x_n|w)))$$

ERROR FUNCTIONS

An error function, also known as a “loss function,” is a measure of how well a feedforward neural network (FFNN) is able to predict the desired output given a set of input data. The error function is used to evaluate the performance of the FFNN and guide the optimization process during training.

To use an error function in an FFNN, the following steps are typically followed:

1. Define the error function: The error function is chosen based on the specific task that the FFNN is trying to perform. For example, mean squared error (MSE) is often used for regression tasks, and binary cross-entropy loss is often used for binary classification tasks.
2. Calculate the error: The error function is applied to the predicted outputs of the FFNN and the true outputs of the data to calculate the error.
3. Backpropagate the error: The error is then backpropagated through the FFNN using an optimization algorithm, such as stochastic gradient descent (SGD), to update the weights and biases of the network. This process is repeated until the error is minimized.
4. Evaluate the error: The error function is used to evaluate the performance of the FFNN on the training data and possibly on a separate validation or test set. If the error is not decreasing sufficiently or if the error on the validation set is significantly higher than the error on the training set, it may indicate that the FFNN is overfitting or underfitting the data.

By minimizing the error function during training, the FFNN is able to learn to make more accurate predictions on unseen data.

Some common error functions used in FFNNs include:

- **Mean squared error** (MSE): This error function is often used for regression tasks, where the FFNN is trying to predict a continuous value. The MSE is calculated as the average of the squared differences between the predicted values and the true values. This is the sum of squared errors that can be found using the MLE to maximize the likelihood of a gaussian distribution of fitting a dataset for regression.
- **Binary cross-entropy loss**: This error function is often used for binary classification tasks, where the FFNN is trying to predict a binary outcome (e.g., 0 or 1). The binary cross-entropy loss is calculated as the negative log likelihood of the predicted probabilities. This can be found maximizing the likelihood of the data to fit a Bernoulli probability distribution for classification.
- **Categorical cross-entropy loss**: This error function is often used for multi-class classification tasks, where the FFNN is trying to predict one of several possible outcomes. The categorical cross-entropy loss is calculated as the negative log likelihood of the predicted probabilities.
- **Hinge loss**: This error function is often used for binary classification tasks, and it is commonly used in support vector machines (SVMs). The hinge loss is calculated as the maximum of 0 and the difference between the predicted value and the true value.

There are other error functions that can be used in FFNNs as well, depending on the specific requirements of the task.

CHAIN RULE

The chain rule is a mathematical concept that allows us to calculate the derivative of a composite function, which is a function made up of multiple functions combined together. The chain rule is often used in feedforward neural networks (FFNNs) to calculate the gradient of the error function with respect to the weights and biases of the network.

The basic idea behind the chain rule is that the derivative of a composite function can be expressed as the product of the derivative of the outer function and the derivative of

the inner function. This is expressed mathematically as:

In other words, the derivative of a composite function is equal to the derivative of the outer function evaluated at the output of the inner function, multiplied by the derivative of the inner function.

$$f(g(x))' = f'(g(x)) \cdot g'(x)$$

For example, suppose we have a neural network with a single weight and we are using the mean squared error (MSE) as the error function. The MSE is calculated as the average of the squared differences between the predicted and actual values for a set of data points.

The predicted values in the MSE are produced by the output layer of the neural network, which is a function of the inputs and the weights. The output layer can be represented by the following function:

$$f(x) = g(h(x, w))$$

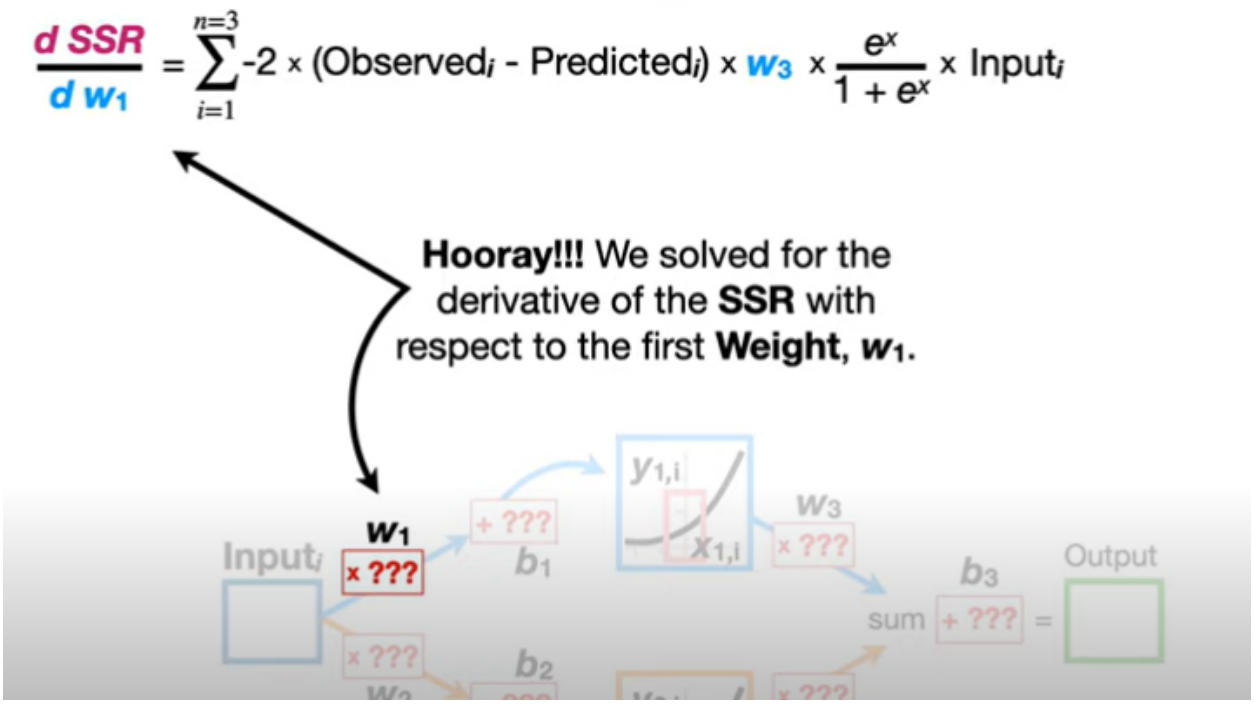
where $h(x, w)$ represents the output of the hidden layer, and $g(h(x, w))$ represents the output of the output layer.

To calculate the gradient of the MSE with respect to the weight w, we can use the chain rule as follows:

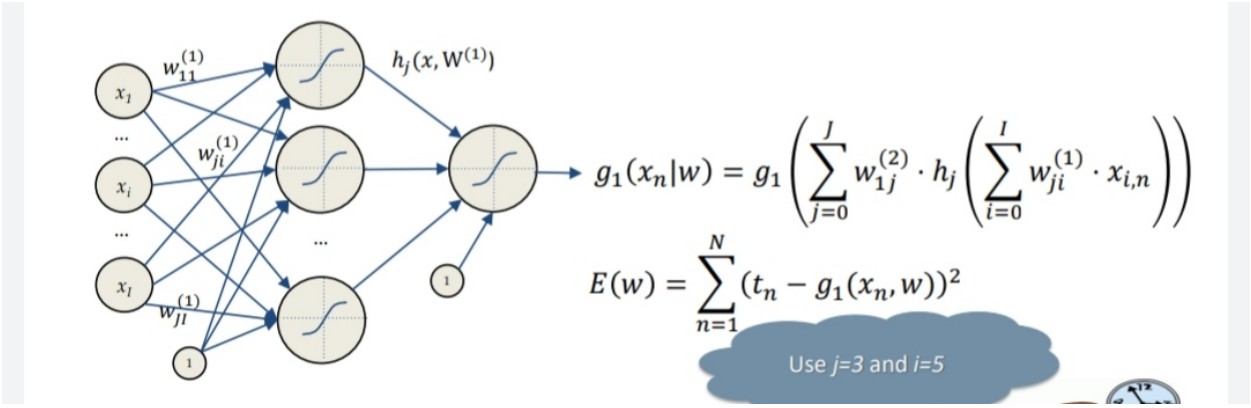
$$\frac{\partial MSE}{\partial w} = \frac{\partial MSE}{\partial f} \cdot \frac{\partial f}{\partial h} \cdot \frac{\partial h}{\partial w}$$

The first term, $\frac{\partial MSE}{\partial f}$, represents the partial derivative of the MSE with respect to the predicted values $f(x)$. The second term, $\frac{\partial f}{\partial h}$, represents the partial derivative of the output layer with respect to the hidden layer. The third term, $\frac{\partial h}{\partial w}$, represents the partial derivative of the hidden layer with respect to the weight.

By applying the chain rule, we can calculate the gradient of the MSE with respect to the weight, which can be used in an optimization algorithm, such as stochastic gradient descent (SGD), to update the weight and minimize the error. This is useful because each weight can this way be updated in parallel with a single backward pass.



GRADIENT DESCENT AND BACKPROPAGATION



Gradient Descent is a popular optimization algorithm that is commonly used to train feedforward neural networks (FFNNs). It is an iterative algorithm that adjusts the weights of the connections between the nodes in the network based on the gradient of the error function, which measures the difference between the predicted output of the network and the actual output. The basic idea behind gradient descent is to take steps in the direction that minimizes a loss function.

Suppose we have a neural network with a single hidden layer and an output layer, and we want to use the MSE as the loss function to train the network to perform a certain task. The input to the network is denoted by x, the output of the network is denoted by $f(x)$, and the desired output is denoted by y.

During the training process, we feed the input data through the network and calculate the output of the network. We then compare the output with the desired output, and calculate the error between the two using the MSE:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(f(x_i) - y_i)^2$$

where N is the number of examples in the training set.

To update the weights of the network so that the error is minimized, we need to calculate the gradient of the MSE with respect to the weights. To do this, we use the chain rule to propagate the error backwards through the network, starting from the output layer and working our way back to the input layer.

For example, suppose we have a hidden layer with three neurons, each with a set of weightsw_1, w_2, w_3. The output of this layer is given by the function g(x), where x is the input to the layer.

To calculate the gradient of the MSE with respect to the weights of this layer, we would use the chain rule as follows:

$$MSE_{gradient} = \frac{2}{N}\sum_{i=1}^{N}(f(x_i) - y_i) \cdot f'(g(x_i)) \cdot g'(x_i) \cdot w_1 + \frac{2}{N}\sum_{i=1}^{N}(f(x_i) - y_i)$$

Once we have the gradient of the MSE with respect to the weights of the hidden layer, we can use gradient descent to update the weights. The basic idea behind gradient descent is to take a step in the direction of the negative gradient, which will move us downhill towards a minimum in the error surface, since the negative of the gradient gives the direction in which the function has the steepest decrease.

To update the weights using gradient descent, we would use the following update rule:

$w_1^{k+1} = w_1^k - \eta * \nabla_{w_1}$
$w_2^{k+1} = w_2^k - \eta * \nabla_{w_2}$

$$w_3^{k+1} = w_3^k - \eta^* \nabla_{w_3}$$

The learning rate determines the size of the step taken to update the weights. A smaller learning rate will result in slower convergence, but may be more stable, while a larger learning rate can make the computation faster but is imprecise and not recommended.

This technique is called backpropagation because the error is sent back through the whole network: we can compute the derivatives autonomously and then obtain the desired result by performing a product. Each neuron finds the derivative of the output with respect to the input values and propagates it back to the previous one, which already has its own derivatives that will be multiplied by it. The update rule of gradient descent works because it uses the gradient of the error function to adjust the weights and biases of a neural network in the direction that minimizes the error: even if the weights are updated in respect to each one of them, every computation keeps the information about the error function’s gradient in respect to the outputs, which gives the direction of steep decrease.

If the gradient is positive, it means that the error is increasing as the weight increases. In this case, the update rule of gradient descent would adjust the weight in the negative direction in order to decrease the error. On the other hand, if the gradient is negative, it means that the error is decreasing as the weight increases. In this case, the update rule of gradient descent would adjust the weight in the positive direction in order to continue decreasing the error.

By repeatedly adjusting the weights and biases in this way, the neural network is able to learn to make more accurate predictions on unseen data.

Batch Gradient Descent

Batch gradient descent (BGD) computes the gradient with respect to the parameters of the cost function on the entire training set:

$$\theta_{k+1} = \theta_k - \eta \cdot \nabla_\theta J(\theta)$$

This means that the algorithm is slow and requires a lor of memory to be used. It converges to the global minimum if the function to optimize is convex, otherwise to a local minimum or a saddle point.

Stochastic Gradient Descent

Stochastic gradient descent (SGD) is an optimization algorithm that is used to minimize the cost function of a machine learning model. It works by iteratively updating the model parameters based on the gradient of the cost function with respect to the parameters. Unlike batch gradient descent SGD computes the gradient using a single training example at a time:

1. Initialize the model parameters with some initial values.
2. For each epoch:
    - Shuffle the training dataset.
    - For each training example:
        - Compute the gradient of the cost function with respect to the parameters using the current training example.
        - Update the parameters using the gradient and a learning rate.
3. Repeat steps 2 and 3 until the cost function is minimized or a specified number of epochs has been reached.

This will produce the following output:

$$\theta_{k+1} = \theta_k - \eta \cdot \nabla_\theta J(x_i, y_i | \theta)$$

To reduce the fluctuations around local minima, tools like Momentum or NAG can be used, which add a new value to the vector update in order to be less focused on variations and give more strength to the overall decreasing direction. This is performed by keeping a sort of “memory” of the previous gradient values.

$v_t = \gamma v_{t-1} + \eta \cdot \nabla_\theta J(\theta)$;

$\theta_t = \theta_{t-1} - v_t$;

Mini-Batch Gradient Descent

Optimal compromise between the two previous techniques, updates the gradient after a batch of n training examples:

$$\theta_{k+1} = \theta_k - \eta \cdot \nabla_\theta J(x_{i:i+n}, y_{i:i+n} | \theta)$$

This means that there’s overall less variance than in SGD, so more convergence stability, but it still reduces the computational load.

Learning Rate Optimization

Even if the gradient descent algorithms allow to reach a convergence, they still can be adapted using the learning rate hyperparameter for better performance. This value can be fixed or adaptive. Popular methods for adaptive learning rates include:

- **Adagard**, which updates the learning rates in a way that less frequent weights receive more updates than the more frequent ones, making it fitting for sparse data. However, it may induce vanishing gradients, so other versions are preferred (Adadelta, which uses means to also reduce the memory load, and RMSprop);
- **Adam**, which adapts the rate to each parameter using the squared mean of past gradients and a decadent mean of past gradients. One term is added for fast convergence and another to prevent oscillations. Other versions are AdaMax and Nadam.

Vanishing Gradient

The vanishing gradient problem is a phenomenon that can occur when training deep neural networks, in which the gradients of the parameters with respect to the loss function become very small, and the network is unable to learn effectively. This can happen when the activation function used in the network has a derivative that is close to zero for a large range of input values.

Suppose we have a deep neural network with N layers, and we're using the sigmoid activation function in each layer. The sigmoid function is defined as:

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

The derivative of the sigmoid function is:

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Now, let's consider the case where we have a weight in the first layer of the network, and we're trying to compute the gradient of the loss function with respect to the weight.

We can use the chain rule to write this as:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y_1} \frac{\partial y_1}{\partial w}$$

where y_i is the output of the first layer, and w is a weight in that layer.

Now, suppose that the output of the first layer is very close to 1. In this case, the derivative of the sigmoid function will be close to zero, because $\frac{d\sigma(x)}{dx}$ approaches zero as x approaches infinity. This means that $\frac{\partial y_1}{\partial w}$ will be very small, and the gradient $\frac{\partial L}{\partial w}$ will also be very small.

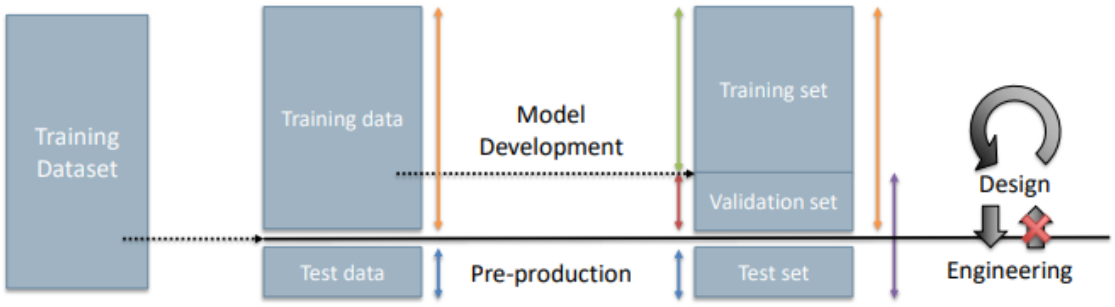
As a result, the network will be unable to update the weight effectively, and the learning process will be slowed down or stopped entirely. This is an example of the vanishing gradient problem. A similar issue can come up when activation functions gradient can give high values (exploding gradient).

Training and Overfitting

A solution approximating the target function over a sufficiently large set of training examples will also approximate it over unobserved examples. Training error/loss is not a good indicator of performance on future data, because the classifier has been learned from the same training data, and estimates will be optimistic. Also, the new data will probably not be the same as the training one.

For these reasons, we need an independent test set; either there's another one, or we can perform some splitting or subsampling, if class distribution is maintained. The training dataset is the available data, which is split into a training set (used for the model's parameter learning) and validation set (data to perform model selection). Both of these are used for model development.

The test set, on the other hand, is used for final model assessment. So training data (yellow) is used for model training (fitting and selection), while validation data (purple) is used for selection and assessment.



Usually, it's assumed that an algorithm will be trained and learn to predict the output for all the other unknown data samples. However, especially when the training phase was too extensive, the number of parameters in the model was too high compared to the size of the training set, or the input data was scarce, the model could adapt to characteristics that were limited to the training set.

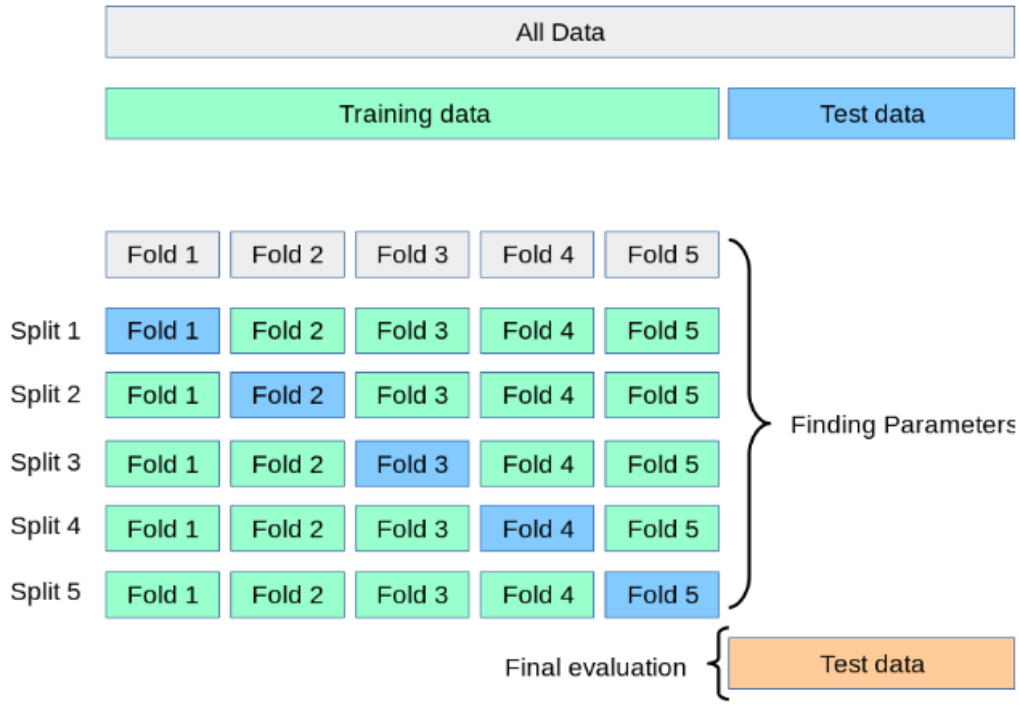
This has an impact on the model's generalization capabilities, which are the ability to perform optimally on unseen data.

Cross-Validation

Cross-Validation is the use of a training dataset to both train the model (parameter fitting + model selection) and estimate its error on new data. It can highlight when overfitting is starting to happen. It consists in removing a section from the training set and using it for validation. The model is trained with its data, and then the loss function is evaluated on both the training set and the validation set. This can show an overall decreasing trend for the first error, but if the second one reaches a minimum and then starts to rise again, it is a sign of overfitting.

Some variants of this practice include:

- **Holdout Cross-Validation**, simplest one, uses a section of the dataset for training and one for validation, preferrable with lots of available data;
- **K-Fold**: Dataset divided in k subsets. The training is repeated k times, using a different split for validation, to have less variance and be less dependant from the selection of samples. However, it requires k training phases, and it's computationally heavy;
- **Leave-One-Out**: the net is trained over N-1 examples, only one is left out for each training. Preferred when having few data available.



Hyperparameters

To evaluate the goodness of a net we can use cross-validation and test sets. However, they do not give indications on what hyperparameters to choose:

- Number of neurons and their disposition;
- Activation functions;
- Error functions;
- Gradient descent algorithm;
- Batch dimension;
- Validation set dimension;

There's no best choice a-prioril it's a matter of experience and design. Strategies that can be useful include monitoring the trend of cross-validation and adapting the choice of

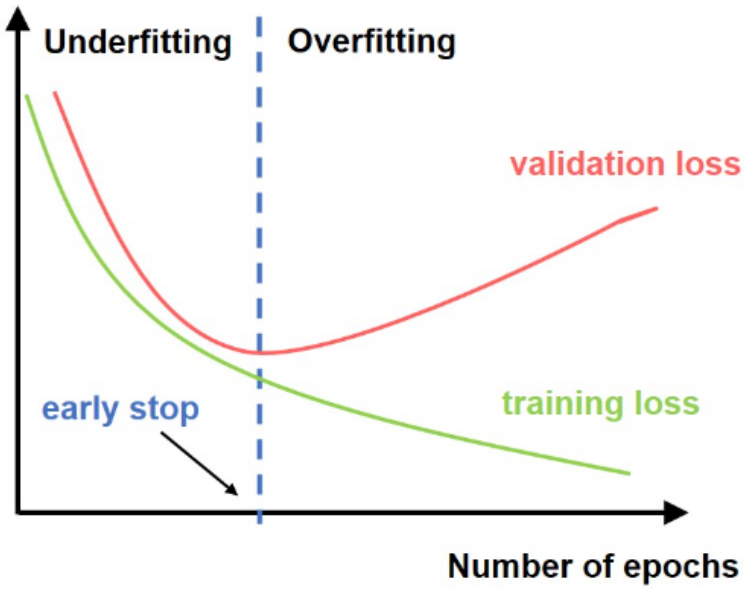
hyperparameters to the best performance. The changes done considering the validation set need to be reflected on test set performance, otherwise it’s still a form of overfitting.

Preventing Overfitting

Different approaches can be used to prevent overfitting.

Early Stopping

Overfitting networks show a decreasing trend in the training error, but a growing validation error. To prevent this, the training can be stopped according to a set of hyperparameters, which are the monitoring quantity, the absolute change, and the number of epochs after which the iterations are stopped if there’s no improvement in the selected metric (patience).



Weight Decay

This regularization puts a constraint on the weights to reduce the model’s freedom. It’s based on an a-priori assumption on the distribution of initialized weights in the model to keep them smaller. This is because smaller weights can allow for better generalization capabilities.

Doing this, we are practically adding to the loss function a penalty term that is proportional to the sum of the squares of the weights in the network. This way, the gradient is forced to reduce the overall norm of the weigts.

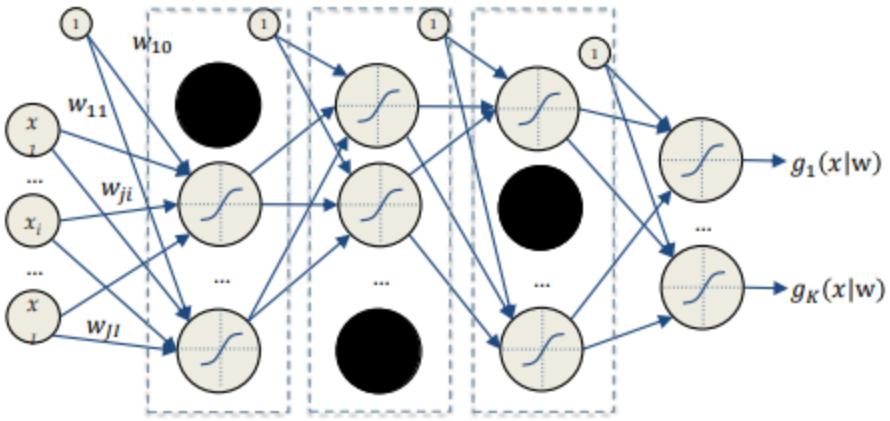
$$argmin_w(\sum_{i=1}^{N}(t_i - g(x_i|w))^2) + \gamma \sum_{q=1}^{Q}(w_q)^2$$

To choose the best gamma, we can use cross-validation, training the model over different values and choosing the one with the best outcomes.

Dropout

This method selects some neurons using a random probability and turns them off during one epoch of the training phase. This way, the network is forced to learn independent features, preventing co-adaptation. By randomly dropping out neurons, the network is forced to learn redundant representations of the features, which helps reduce overfitting.

Each epoch, a different set of neurons is turned off. The final net is an ensable of the different “subnets” used in training, scaling the weights to average the output. During evaluation, we need to scale the activations of the remaining neurons by a factor of $\frac{1}{1-p}$ to compensate for the dropped-out neurons. This is because the expected value of the activations is $(1 - p)$ times their original value when dropout is applied. The choice of p is done as a hyperparameter.



Better Activation Functions

Activation functions like sigmoid or tanh may lead to vanishing gradient issues when training deep NN. In fact, the inputs of the function have to be in a safe range, otherwise, the local gradient may assume values close to 0, which, because of the chain rule, will be multiplied for the others, leading to a nullification. Also, sigmoid's outputs are not centered in 0, therefore it may lead to an oscillating trend in gradient direction. Lastly, the exponential function is a costly operation.

This is why the ReLu was introduced: it removes the saturation problem and converges a lot faster, and it's also scale invariant. However, if the input is negative, it can go to 0, causing vanishing gradient issues in deeper layers and even a neuron's death. It's also not differentiable at 0 and has a non-zero-centered output. [Its unbound nature can lead to an exploding gradient because a derivative of 1 always passes the gradient forward. APPARENTLY MATTEUCCI THINKS THIS IS NOT TRUE (HO SBAGLIATO LA CROCETTA BASTARDA)]

$$\frac{df(x)}{dx} = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

The LeakyReLu avoids the vanishing gradient issue with a pendency where the ReLu was 0 (fixes dying neurons). Other variations include the ELU, which is parametric, and the Randomized ReLu, con pendenza scelta casualmente.

Weight Initialization

Final gradient descent is affected by the initialization of weights. There are different approaches:

- All Zeros: bad choice, every neuron will compute the same gradient and the updates will be the same for each one, leaving us with a net that cannot extract any feature.
- Big Numbers: bad choice, big weights lower generalization capabilities and can lead to an early gradient saturation with some activation functions.
- Small Values: with values around 0, from a 0-centered gaussian distribution with low variance $w \sim N(0, \sigma^2 = 0.01)$, a small network can produce good outcomes, but

bigger nets could suffer from vanishing gradient.

In deeper networks, small gradients shrink through each layer, while bigger ones grow too much.

Xavier initialization

Initialization follows a gaussian distribution with a 0 mean and a standard deviation, which makes the total variance of activations equal to 1. This means making the input variance unitary, under the hypothesis of linear and symmetric activations, so it's necessary to do some preprocessing on it before performing the initializations. The weights will then be sampled from this distribution:

$$w_{i,j} \sim \mathcal{N}(0, \frac{1}{n_i})$$

Where n_i s the input dimension.

Batch Normalization

Usually, the input is manipulated for better performance: operations include normalization (to have all values in a [0,1] range) and standardization (to have a 0 mean and 1 standard deviation). This helps with training because it strengthens the feature extraction process, but deep nets can have a much reduced impact. This is the reason a BatchNorm layer is introduced: to standardize not only the input but also the output of each layer. Each BatchNorm layer has 2 weights per batch, a scale factor and a bias, that can be adjusted during training. Batch refers to a particular subset of data taken into account by SGD training.

In practice:
• Each unit's pre-activation is normalized to $x_{i,j} \sim \mathcal{N}(0, 1)$, also the input values;
• During training, mean and stddev are computed for each minibatch (output of a subset of the net);
• Backpropagation takes into account normalization;
• At test time, the global mean / stddev are used (global statistics are estimated using training running averages);

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

The error factor is used to augment variance and stabilize the denominator, avoiding 0 divisions. The parameters gamma and beta are learned during training, so the error that's back-propagated has to take these into account. The y_i value is the normalized value that will go through the activation function. This process has the aim of minimizing covariate shifts, which are the changes in the distribution of activations that occur during training.

We select a mini batch of n input values and m neurons in the layer, so that the output of batch normalization will be a matrix B of size $n \times m$, where each column represents the transformed activations of the neurons for a single training example. In RNN the values are found for each time step, while in CNN for each filter.

This technique has been shown to:

- Improve gradient flow;
- Allow for higher learning rates and thus faster learning;
- Reduce dependence on weight initialization, and the necessity of dropout;
- Use of binary classification functions in deeper nets;

Another great benefit of it is that it helps reduce the vanishing gradient issue by stabilizing the distribution of activations in the network, which can improve the stability of the optimization process and reduce the risk of vanishing gradients.

Regularization

Regularization consists in adding another factor after the error function, one that’s related to weights, usually their norm-1 or norm-2, to reduce the overall norm and obtain smaller weights. When the weights in a model are large, the model is able to fit the training data more closely, which can lead to a lower training error. However, large weights can also make the model more sensitive to the noise and random variations in the training data, which can cause the model to learn patterns that are not representative of the underlying relationship between the input and the output. As a result, the model may perform poorly on unseen data that does not contain these patterns. There has to be balance between proper learning and overfitting, so a γ factor is introduced as a hyperparameter to be tuned. This operation has the purpose of restricting the value range in which the weights can be selected and, thus, their variance.

$$E(w) = E_p(w) + \gamma||w||$$

Image Classification

Computer vision is an interdisciplinary scientific field that deals with how computers can be made to gain high-level understanding from digital images or videos. It starts from a 2D image and then creates a model of the world, understanding its features. It's used commonly in tasks like;

- Recognition: one or more pre-specified objects can be brought back to their classes together with their position;
- Identification: a specific class instance is found;
- Detection: image is processed until a certain characteristic is spotted;

Images have a size, the number of pixels in the horizontal and vertical dimensions, and a depth, the number of bits used to represent each pixel. For example, an image with 8-bit pixel depth has 8 bits (1 byte) per pixel, which can represent 256 different values. Color information spans 256 different values, and usually images are represented using red, green, and blue (RGB): $I \in \mathbb{R}$^(R × C × 3)

R = I[:, :, 0]
G = I[:, :, 1]
B = I[:, :, 2]

Videos on the other hand have frames: $V \in \mathbb{R}$^(R × C × 3 × T). A T=5 frame video with R = 144 and C = 180 contains 144*180*5*3 values ranging from 0 to 255. Visual data are very redundant, so they're easy to compress, but we do not use compression during a NN training.

Local (Spatial) Transformation

These transformations can be written as:

$$G(r, c) = T_U[I](r, c)$$

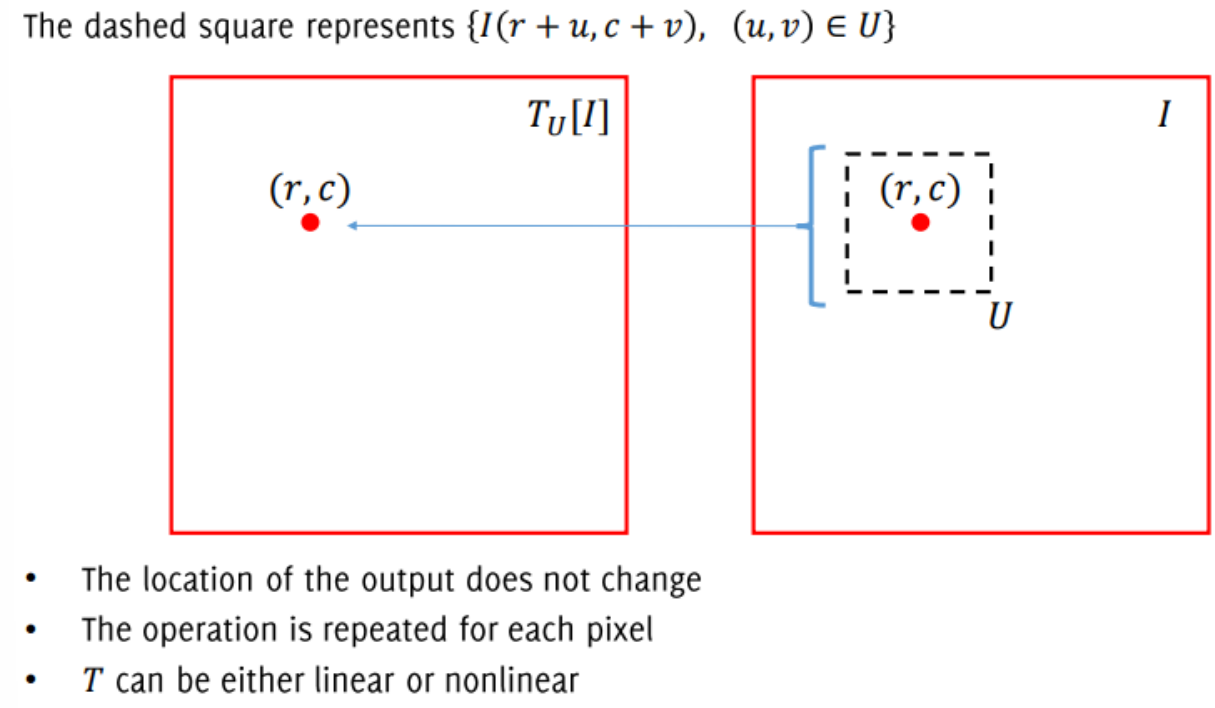
Where:

- I is the input image to be transformed;

- G is the output;
- T_u is a function transforming the image;
- U is a neighborhood, identifies a region that will concurr

T operates on I around U, so that the output at pixel (r,c) is defined by all pixels in U in the input image, such that U is represented by a set of displacements.

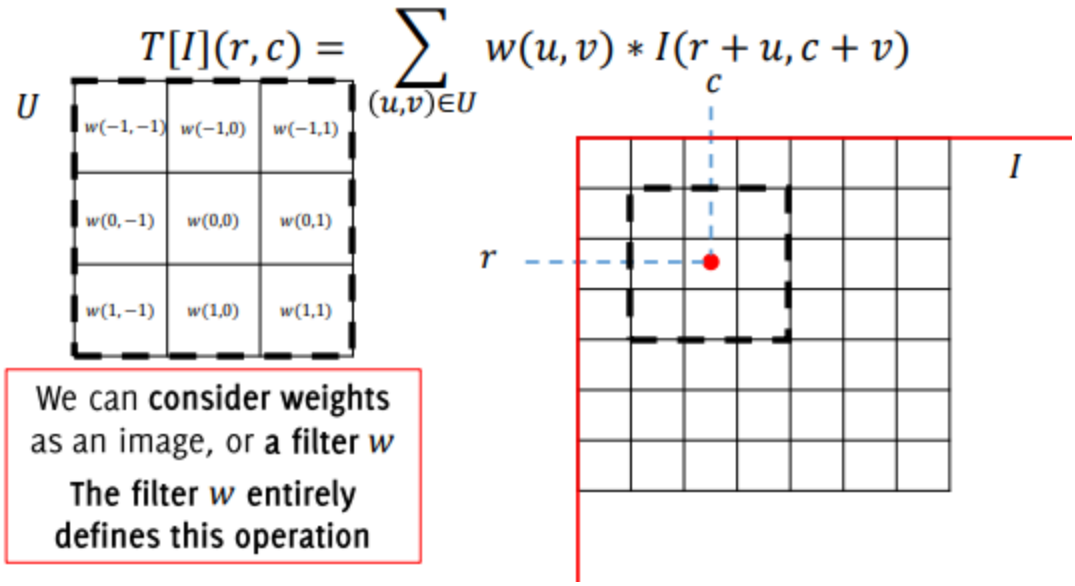
$$\{I(r+u, c+v), (u,v) \in U\}$$



Linearity implies that:

$$T[I](r,c) = \sum_{(u,v)\in U} w(u,v) * I(r+u, c+v)$$

So that every pixel in the U area of the original I image is multiplied by a weight associated to its position, and the sum of this operation over all the U area gives the final scalar result in the output image. We can consider *w* weights as an image, or a **filter** *h,* , that entirely defines this operation.



Correlation

The correlation among a filter and an image is defined as:

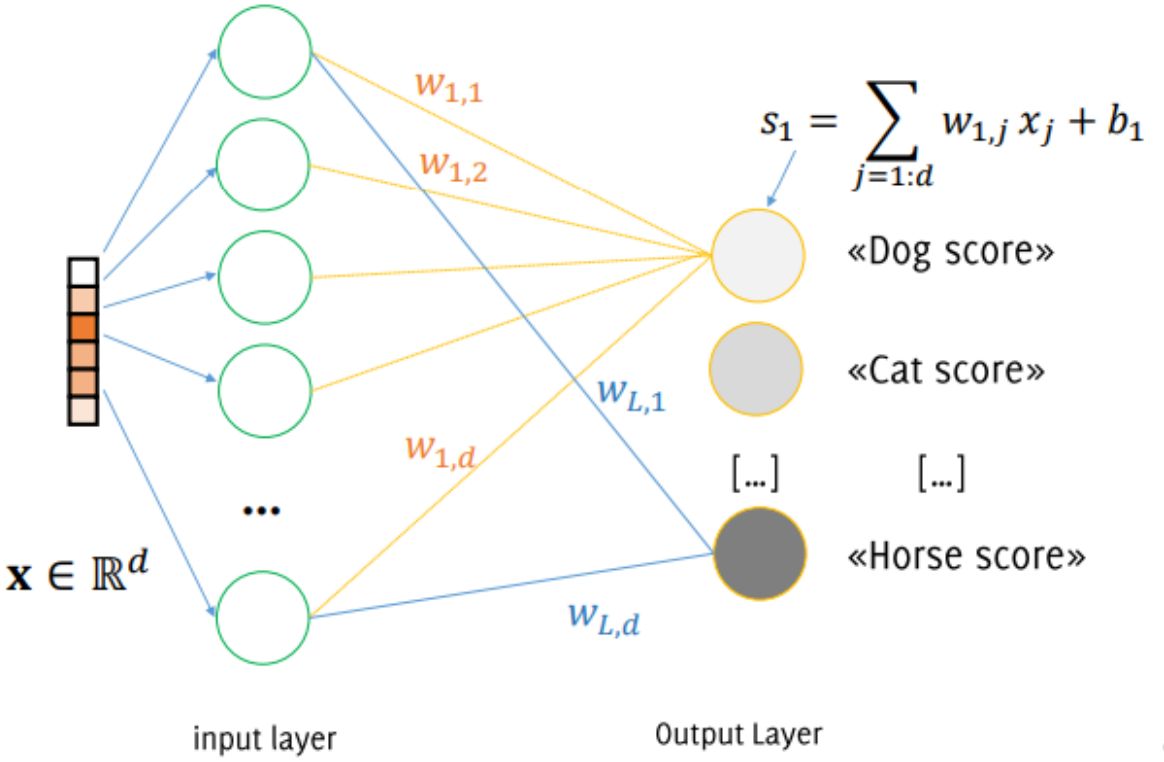
$$(I \otimes w) = \sum_{u=-L}^{L} \sum_{v=-L}^{L} w(u,v) * I(r+u, c+v)$$

where the filter w is of size (2L + 1) x (2L + 1). Normalization helps to scale the pixel values in the image and the values in the kernel to a similar range, which can make the correlation more interpretable and easier to compare to other correlations. Normalizing the pixel values in the image and the values in the kernel by subtracting the mean and dividing by the standard deviation scales the values to a standard range, typically between -1 and 1. This allows us to compare the correlation between the image and the kernel to the correlations between other pairs of images and kernels that have also been normalized.

Linear Classifier

How to feed images to a NN? We use column-wise unfolding to unroll the image into a vector shape, of length $d = R \times C \times 3$, in which every element represents the intensity of a specific color in a pixel of the image (this means that each of the 3 colors is represented by $R \times C$ pixels that span a rangEe of values from 0 to 255). The output of a classifier is usually a vector with the same length as the number of classes, whose

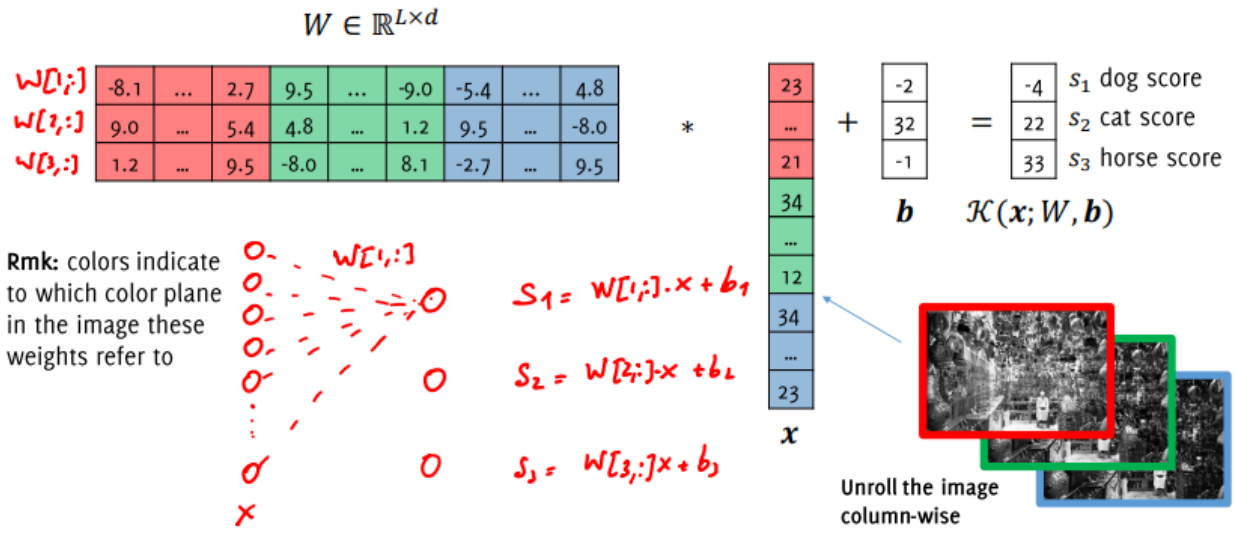
contents are the scores associated with each class in the input image. Suppose we have a 1-layer NN for classification:



High dimensionality prevents a straightforward approach because the number of needed parameters is very high. Nonlinearity is not needed since there are no following layers, also the softmax because it would not change the scores, only normalize them.

If we consider L as the number of classes and d as the unrolled image size, we can rearrange the weights in a matrix $W \in \mathbb{R}^{(L \times d)}$ so that the scores are given by a product:

$$s_i = W[i, :] * x + b_i$$



In linear classification, $K(x)$ is a linear function:

$$K(x) = Wx + b$$

where $W \in \mathbb{R}^{L \times d}$ and $b \in \mathbb{R}^{L}$

The classifier assigns to an input image the class corresponding to the largest score:

$$y_i = argmax_{i=1,\dots,L} [s_j]_i$$

Each layer in a NN can be seen as matrix multiplication + bias; stacking layers becomes equivalent to obtaining a new linear combination of the input, and this is why to decouple the factors non-linear activations are needed. Each weight can be interpreted as the importance of a corresponding feature (such as a pixel in an image) for a specific class. The weights reflect the contribution of the individual features to the final prediction.

Training a Linear Classifier

To find the parameters that minimize the loss function over the whole TR (training set), we have to minimize a loss function:

$$[W, b] = \underset{W \in \mathbb{R}^{L \times d},\ b \in \mathbb{R}^{L}}{\operatorname{argmin}} \sum_{(x_i, y_i) \in TR} \mathcal{L}(x, y_i)$$

The **loss function** measures the unhappiness with the scores assigned to training images. The loss will be higher for a training image that's incorrectly classified. Losses can be minimized through gradient descent, and need proper regularization.

The training dataset is used to learn W and b, and, once they're learned, we can discard it.

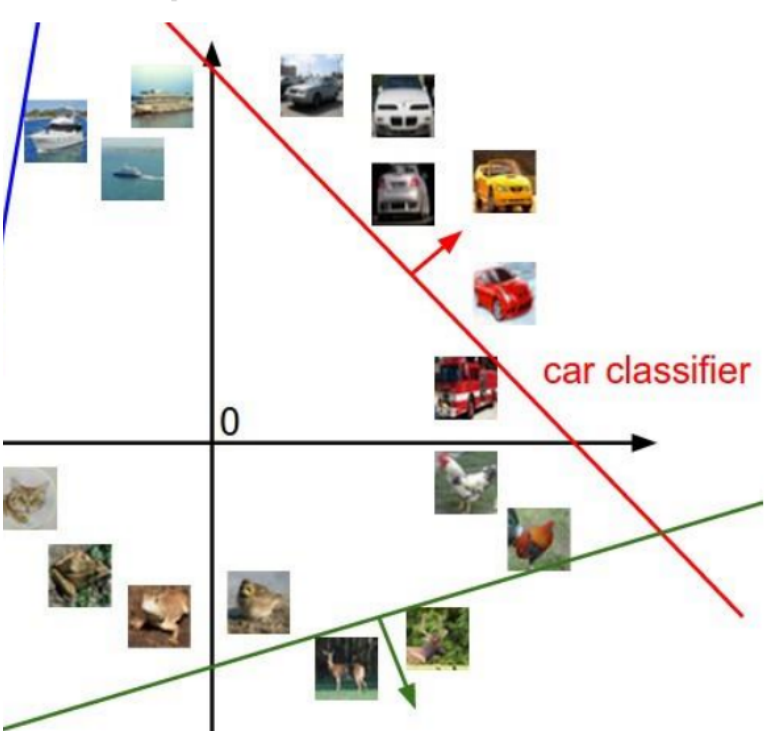
Geometric Interpretation

W[i:] is a d-dimensional vector containing the weights of the score function for the i-th class. Computing the score is equal to do an inner product as shown to find s_i. Thus, the NN computes the product against L different sets of weights and selects the one with the largest score, considering also the bias. No row is influenced by the weights of another, because there's no hidden layer to mix the outputs of different layers.

Each image is interpreted as a point in $\mathbb{R}^d$, Each classifier is a weighted sum of pixel, which coresponds to a linear function in the same space. In $\mathbb{R}^2$, these would be:

$$f([x_1, x_2]) = w_1 x_1 + w_2 x_2 + b$$

Then points $[x_1, x_2]$ with $f([x_1, x_2]) = 0$ would be lines. In $\mathbb{R}^2$, the region that separates negative and positive class scores is a line, in other dimensions it would be a hyperplane. The magnitude of the score indicates the confidence of the classification.



Then, $W[i, :]$ can be seen as a template used in matching the output of the correlation in the central pixel. The template identifies the most relevant features of a class and uses them to find out where other images belong. The template is then a mathematical representation of a class' characteristics through a vector of learned weights. In detail, it approves or disapproves the presence of a certain color in a certain position, assigning a weight to it, which was learned through training. The weights of a vector of weights can modify the boundaries of the class space.

This is an example of the correlation between an image and a learned class template.

Linear Classification Issues

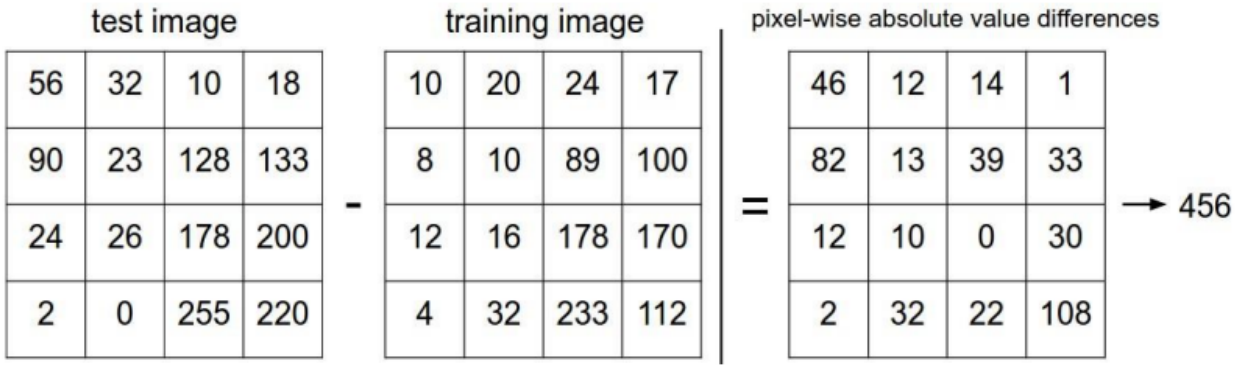
In this way however, the classifier has learned naive features that also take into account backgrounds and characteristics that are too specific considering the dataset. Linear classifier are used because of their relevance and interpretability, but they are not enough to handle complex images.

Dimensionality has to be taken into account when building a model, because high resolution images gave to be stored in memory when training, and a lot of layers require a lot of memory. Transformations can be applied to images and make the classification harder, such as deformation, view point change, scale variations, and others. Classes can vary internally since object are not always standardized.

Nearest Neighbor Classifiers

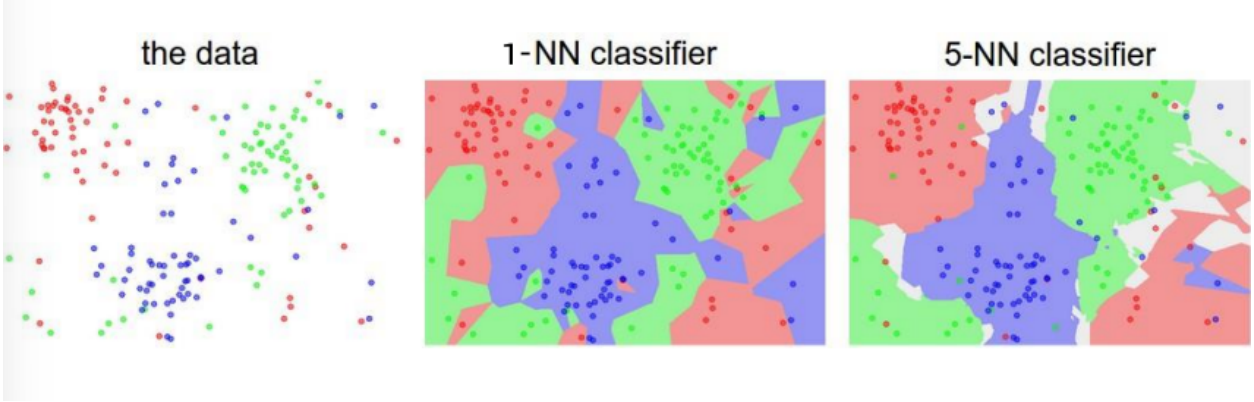
They assign to every image the label of the closest one in the training set. Distance is measured pixel-wise as the difference in the norm-2 of images, or just the modulo. This approach is not very used because it’s not very performing.

$$d(x_j, x_i) = \|x_j, x_i\|_2 = \sqrt{\sum_k ([x_j]_k - [x_i]_k)^2}$$



KNN Classifier

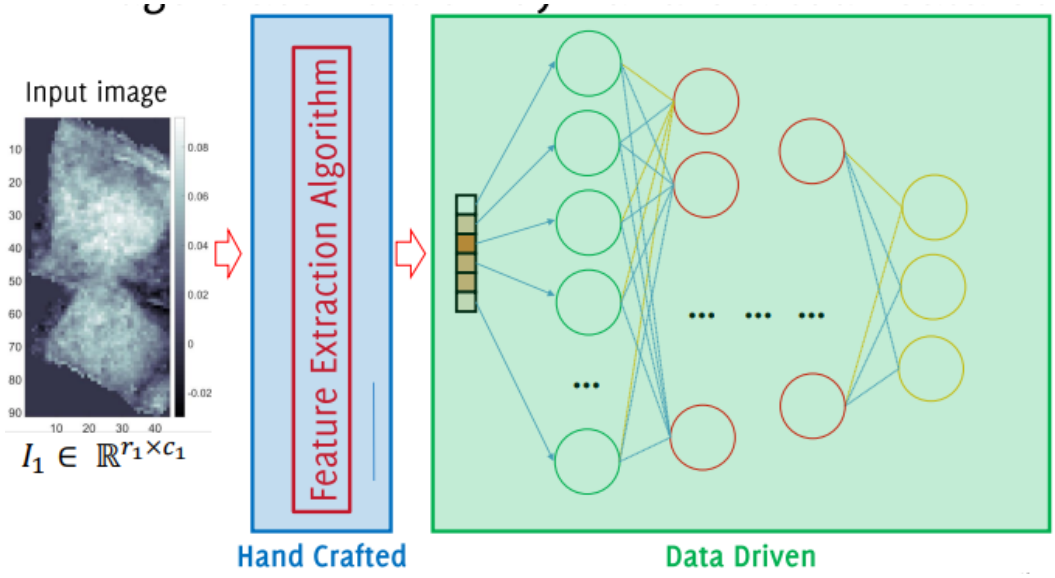
Variation of the previous approach that considers the first k-nearest images and assigns to the group the label that appears the most in their classifications. It reduces irregularities, and it is easy to understand, implement, and train (no training time). But it’s computationally demanding at test time and stores large sets of data in memory, and cannot interpret high dimensional objects correctly.



Convolutional Neural Networks

Image classification can use deep neural networks to better perform classification tasks. Features could be found by hand, considering relevant aspects of the input, to favor interpretability and operability, but it requires a lot of design and programming effort, and risks overfitting by not being a general and portable model.

Also, the classical FC structure is not adequate for processing images since their high dimensionality would require the network to have a large number of parameters per layer, leading to overfitting.



Convolutional NNs make not only the classification data driven, but also the feature extraction. They are networks that are specifically optimized to elaborate image data. A **convolution** is a linear transformation that uses filters to operate on the input. This happens by performing a sum between a set of parameters (the filter/kernel) and a segment of the input of the same dimension, outputting a scalar value. This operation is being performed in each pixel of the input image. Filters are learned from data, and it's important to have them be consistent.

The output size of a Convolutional layer is called **feature map**, and it depends on 3 hyperparameters:

- **Depth**: corresponds to the number of filters in the layer (filters represented as the weights over the connection to another neuron), each of which looking for different characteristics in the input. The set of neurons belonging to different filters and connected to the same input region is the depth column.
- **Stride**: number of pixel of which perform translations over the input image. This parameter allows to set the domain of the depth columns, because each depth column is linked to only one fraction of the input.
- **Zero-Padding**, often used to match input and output shapes.

The CNN has an architecture that's composed by many layers. Once the image gets to a vector shape, it's fed to a traditional NN to perform classification.

The CNN structure is particularly fitting for image processing because of 3 main characteristics:

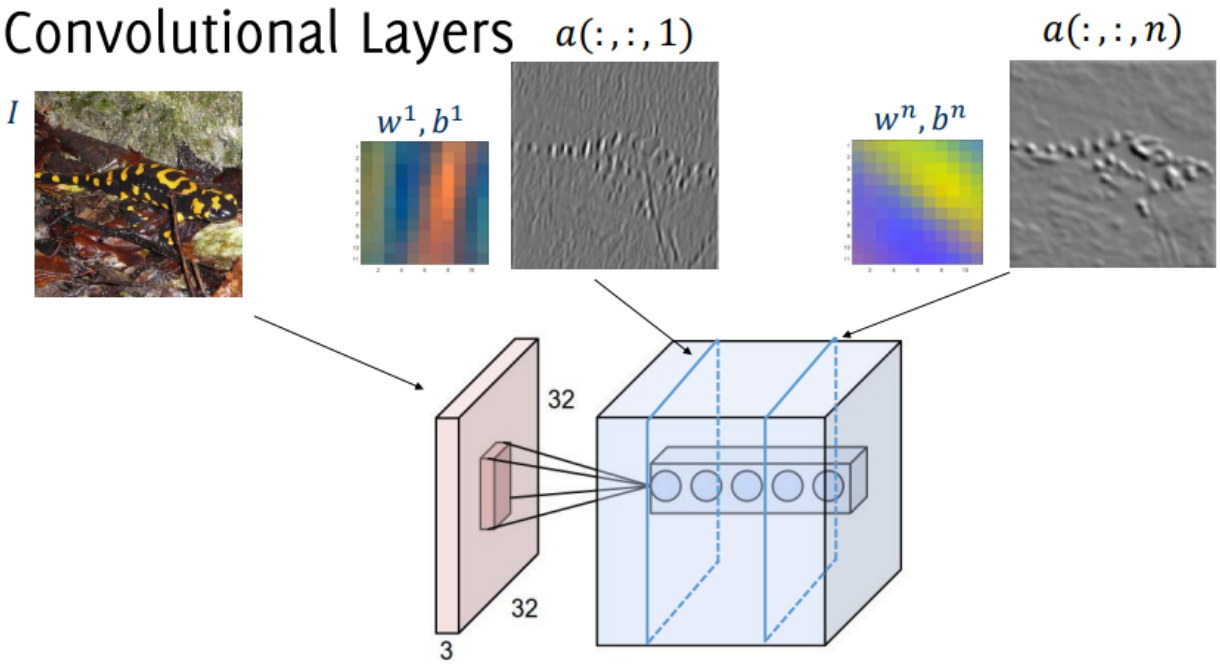
1. **Sparse interaction:** not every neuron is connected to the others. Each one is only connected to a region of the input, which has an extension that's called receptive field of the neuron;
2. **Weight sharing:** if the detection of a certain feature is relevant in one position, it would be relevant also in every section of the input. This means that kernels keep the same parameters while scanning the image, and this implies having the same weights during a convolution of a filter over the whole image. Sometimes though this assumption does not make sense, e.g. when trying to spot localized features;
3. **Translation invariance:** by using multiple filters, a CNN can detect multiple features and construct a hierarchical representation of the input. This hierarchical representation is largely insensitive to small translations of the input, making CNNs translation invariant.

Convolutional Layers

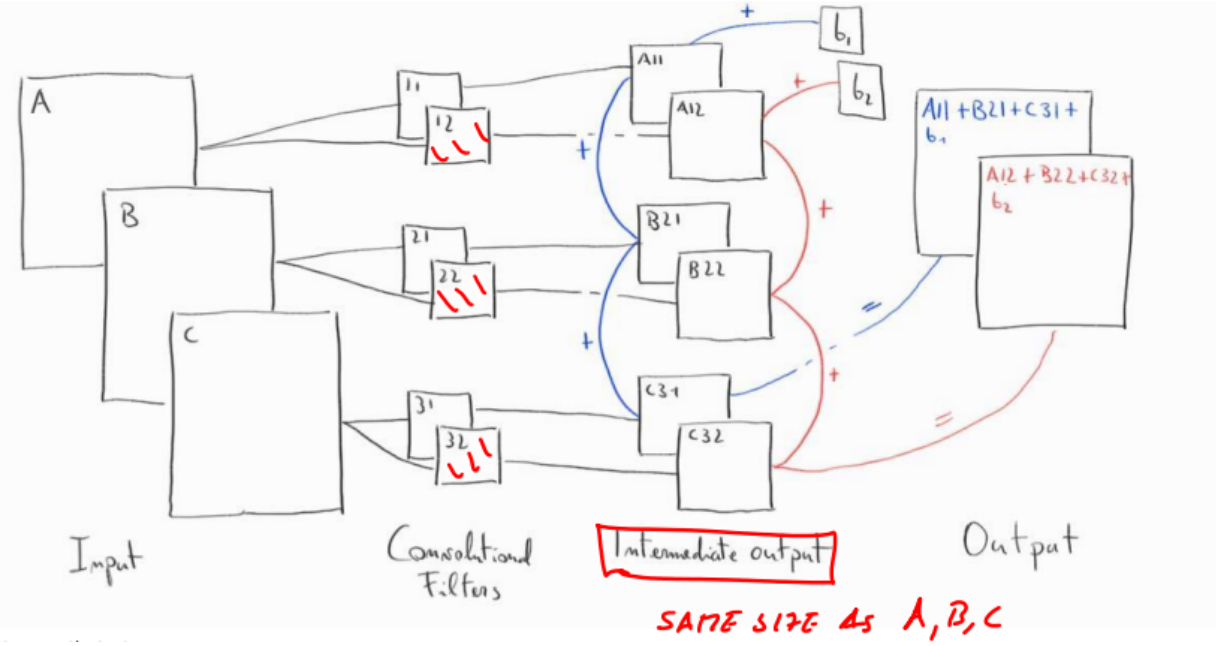
These layers mix all the input components, and their output is a linear combination of all the values in a region of the input, considering all channels. This means that the activation map produced is a result of the outputs of the neurons which are locally connected to the input through weights that correspond to their belonging filter's parameters.

$$a(r,c,1) = \sum_{i,j} w^1(i,j,k)x(r+i,c+j,k) + b^1$$

The $w^1$and b^1 are layer parameters, the filters. The same filters are used for all the spatial extent of the input. Each filter determines a slice in the depth of the output, and each filter has a depth that's the same as the input volume's.



Feature maps and depth column showed in the image. Filters represent the weights of these linear combinations of the input, and are small in spatial extrent but large in depth, corresponding to the one of the input volume. The output of one convoluton against a filter becomes a slice in the volume of the next layer. The number of filters is repeated for each layer of the input, e.g., an input of depth 3 and 2 filters requires the 2 filters to have depth 3 as well. However, the parameters of the filters are the same even if applied to different depths of the input. This means that all the neurons in a depth slice of the feature map use the same weights and biases, corresponding to the same filter's.



The output size of a convolutional layer in a neural network depends on the size of the input, the size of the filter, and the stride used in the convolution operation.

Suppose we have an input tensor of size H x W x D, a filter of size FH x FW x FD, and a stride of S. The output size of the convolutional layer will be:

$$\text{output size} = \left\lfloor \frac{H - FH}{S} + 1 \right\rfloor \times \left\lfloor \frac{W - FW}{S} + 1 \right\rfloor \times FD$$

where $\lfloor \cdot \rfloor$ is the floor function, which rounds down to the nearest integer.

For example, if the input size is H = 5, W = 5, D = 3, the filter size is FH = 3, FW = 3, FD = 3, and the stride is S = 1, the output size will be:

$$\text{output size} = \left\lfloor \frac{5 - 3}{1} + 1 \right\rfloor \times \left\lfloor \frac{5 - 3}{1} + 1 \right\rfloor \times 3 = 3 \times 3 \times 3 = 27$$

The output size can also be calculated using the following formula:

$$\text{output size} = \frac{H - FH + 2P}{S} + 1 \times \frac{W - FW + 2P}{S} + 1 \times FD$$

where P is the padding applied to the input. Padding is an optional operation that adds extra elements to the input around the border, which can be useful in certain situations

(such as preserving the spatial dimensions of the output).

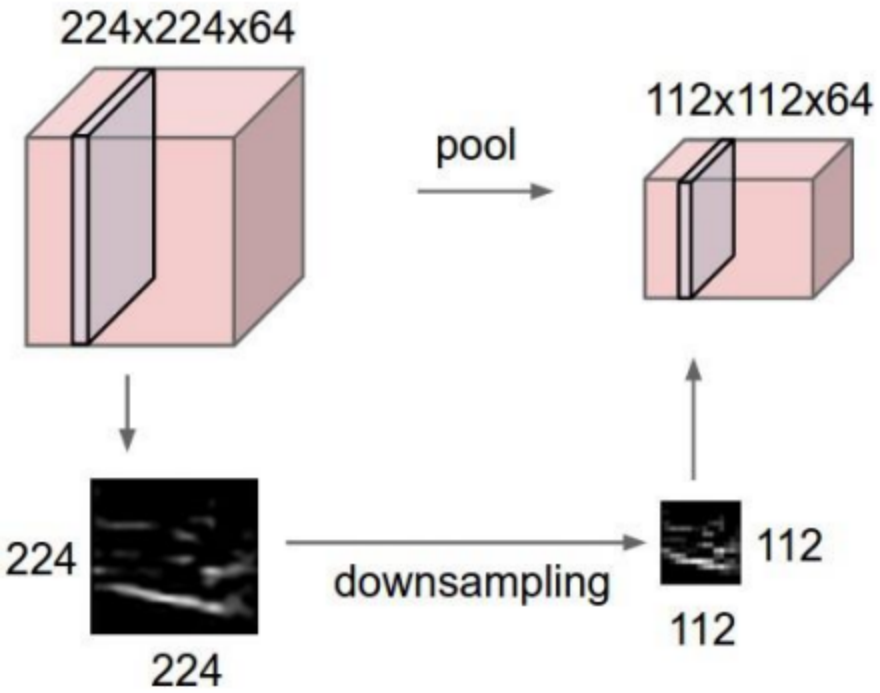
Activation Layers

Activation layers introduce non-linearities in the network to differentiate CNN from linear classifiers. They are scalar functions that operate on each value of the volume, and do not change its size. The ReLu performs a tresholding on the feature maps but may suffer from vanishing gradient and dying neurons, so the Leaky variant is preferred. It acts separately on each layer. Other variants can be the Tanh and Sigmoid, but are preferred in MLP architectures.

Pooling Layers

These layers reduce the spatial size of the volume, leaving the depth untouched. They operate on each layer of the input volume and resizes it using the max operation. Backpropagation considers the derivative 1 at the location corresponding to the maximum and 0 otherwise, so the layer only propagates the error back without retouching. They’re mostly used to reduce the computational load and the overfitting.

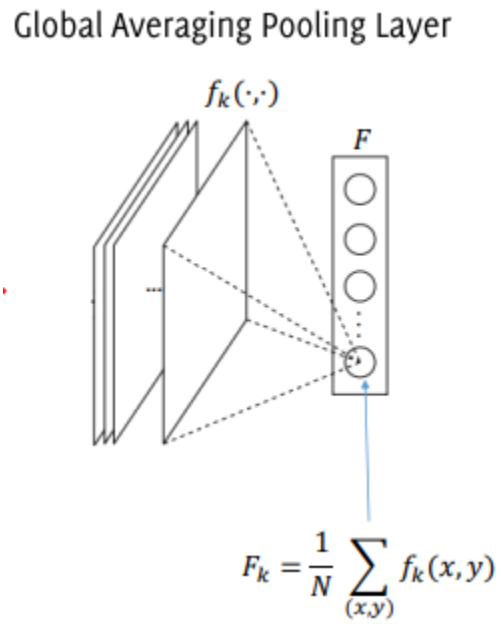
$$\text{output size} = \frac{W_1 - F}{S} + 1 \times \frac{H_1 - F}{S} + 1$$



GAP Layers

In a convolutional neural network (CNN), a global average pooling (GAP) layer is a type of pooling layer that is used to reduce the spatial dimensions of a feature map. It works by taking the average of all the values in the feature map and producing a single output value for each feature map.

GAP layers are often used in CNNs as a way to reduce the number of parameters and computational complexity of the network, as well as to improve the generalization of the model. They can help capture the global structure of the image and reduce the sensitivity of the model to small translations and deformations. The GAP is used as a substitute for the fully connected output layer, and a soft max is introduced just after it. The number of feature maps has to correspond to the number of output classes, and this is why it's used when there are few classes, providing a hidden layer to adjust feature dimension.



Advantages of GAP:

- No parameters to optimize, less overfitting;
- Classification performed by a softmax at the end of the GAP;
- Increased robustness to spatial transformations, like shifts, because flattened MLP are not invariant to shifts (different neurons connected to different weights), while

GAP reduce this problem;

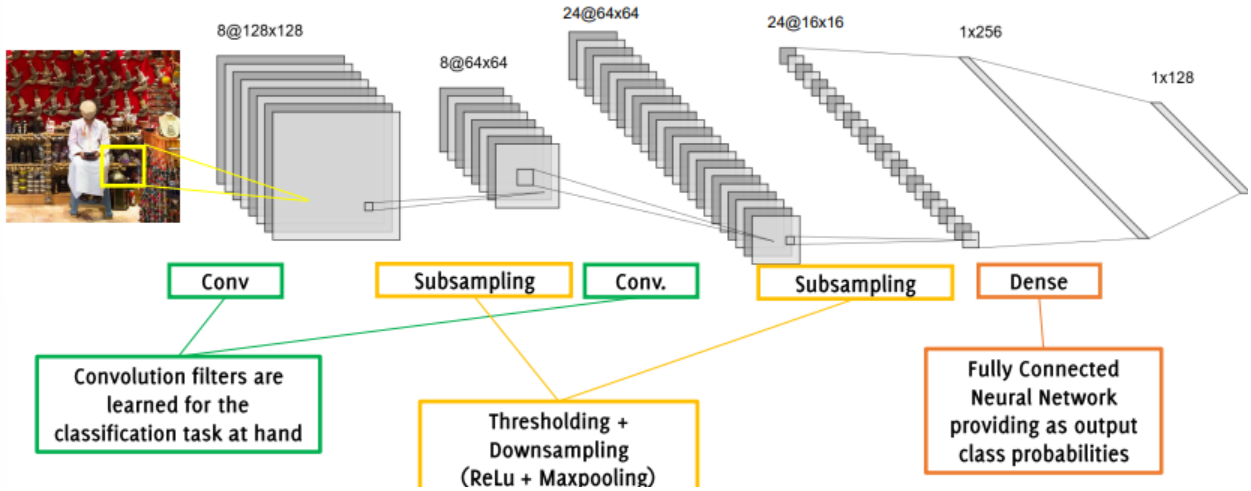
- Network can be used over images of different sizes;

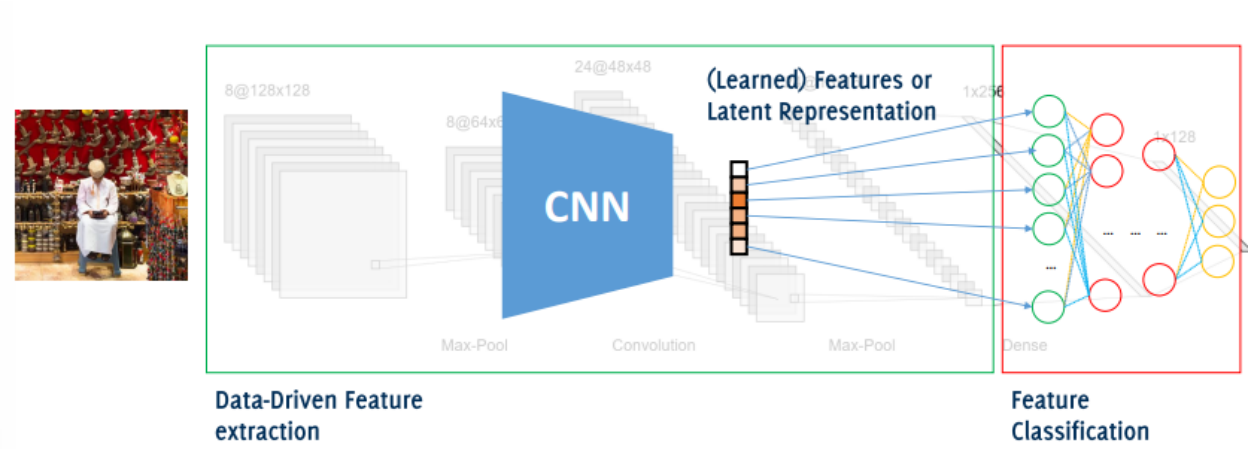
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Input (InputLayer)           [(None, 64, 64)]          0
reshape_3 (Reshape)          (None, 64, 64, 1)         0
conv1 (Conv2D)               (None, 62, 62, 32)        320
pool1 (MaxPooling2D)         (None, 31, 31, 32)        0
conv2 (Conv2D)               (None, 29, 29, 64)        18496
pool2 (MaxPooling2D)         (None, 14, 14, 64)        0
conv3 (Conv2D)               (None, 12, 12, 128)       73856
gpooling (GlobalAveragePooli (None, 128)               0
dropout1 (Dropout)           (None, 128)               0
classifier (Dense)           (None, 64)                8256
dropout2 (Dropout)           (None, 64)                0
Output (Dense)               (None, 10)                650

=================================================================
Total params: 101,578
Trainable params: 101,578
Non-trainable params: 0
_________________________________________________________________
```

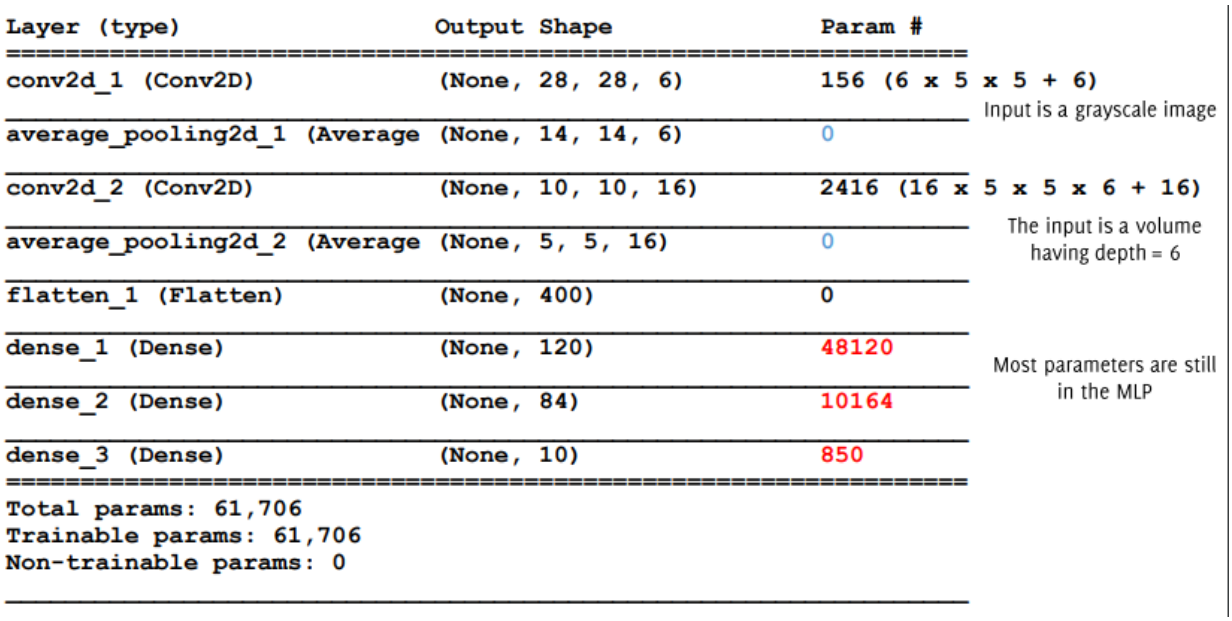
Dense Layers

The spatial dimension is lost and the CNN stacks the hidden layers. It’s called dense because each output neuron is connected to each input neuron. This contains the CNN network found features through which we perform the final classification. In fact, the last layer (FC layer) has the same size as the number of classes, and provides a score for the input image to belong to each class. They’re useful because they can summarize the feature extraction results using single numeric values that will be used by successive layers.





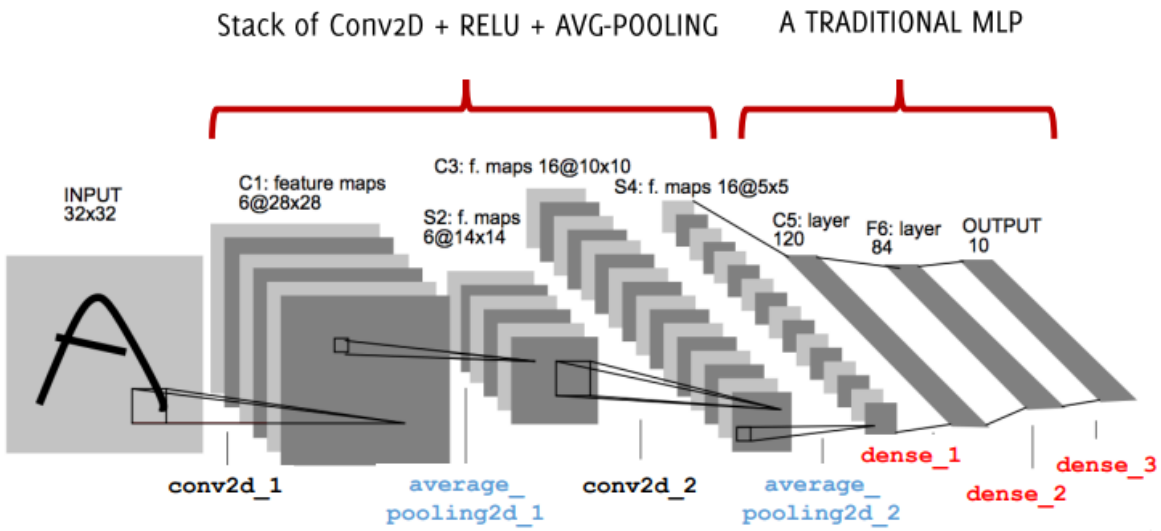
Using each pixel as input for a large MLP is not a good idea because images are highly spatially correlated and using each one as separate input features would not take advantage of this.



Relevant CNN Architectures

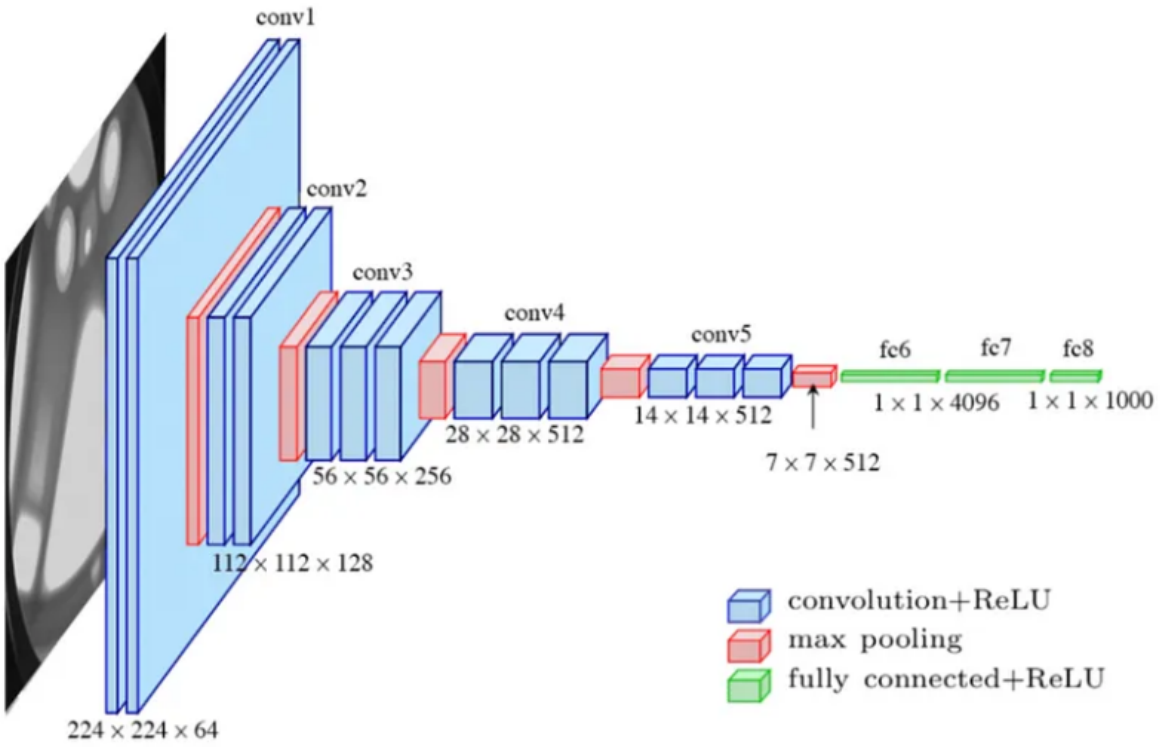
LeNet-5

Created for identifying handwritten numbers, it uses sequences of convolutional, pooling and nonlinear activation layers. Convolution extracts features and subsampling is done through average pooling. Non-linear functions like sigmoid and tanh lead to a last layer which is a MLP classifier.



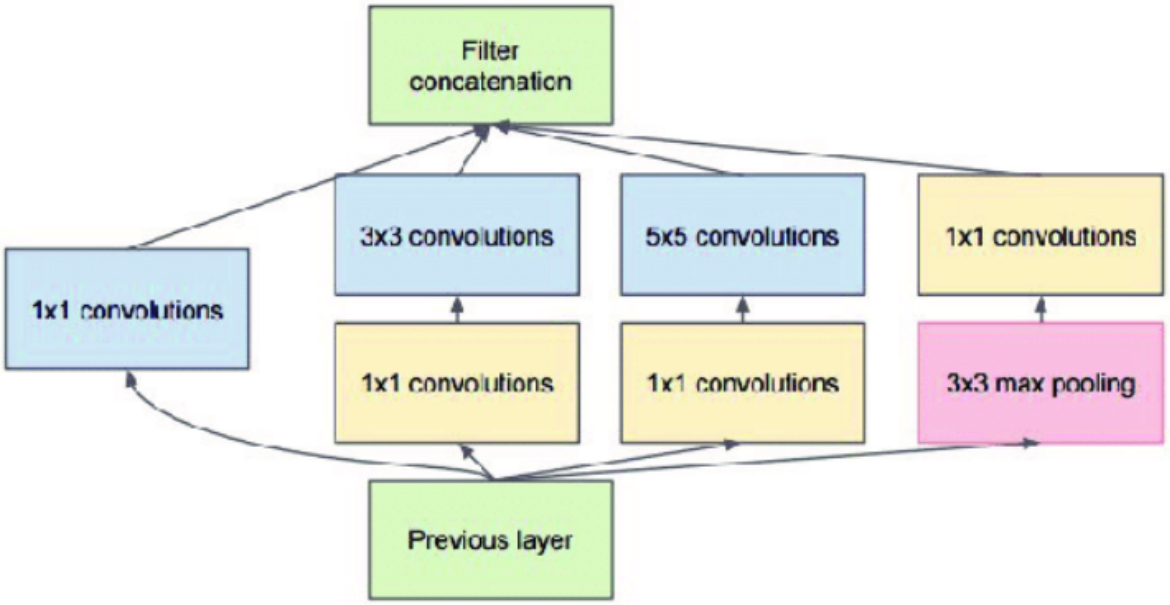
VGG16

Variant of Alexnet, which reduces overfitting by introducing ReLu, Dropout, weight decay and maxpooling. Input dimension is 244x244x3, uses multiple 3x3 conv layers to obtain larger receptive fields with less parameters and more non linearity compared to larger filters in a single layer.



Inception Module

To reduce computational load, since the net works by increasing the depth size, a 1x1 conv layer is introduced before 3x3 and 5x5, to increase non-linearities and reduce the number of parameter. Concatenation preserves spatial resolution, using zero padding, but depth is much expanded, and this is why using 1x1 conv layers is needed for dimensionality reduction.

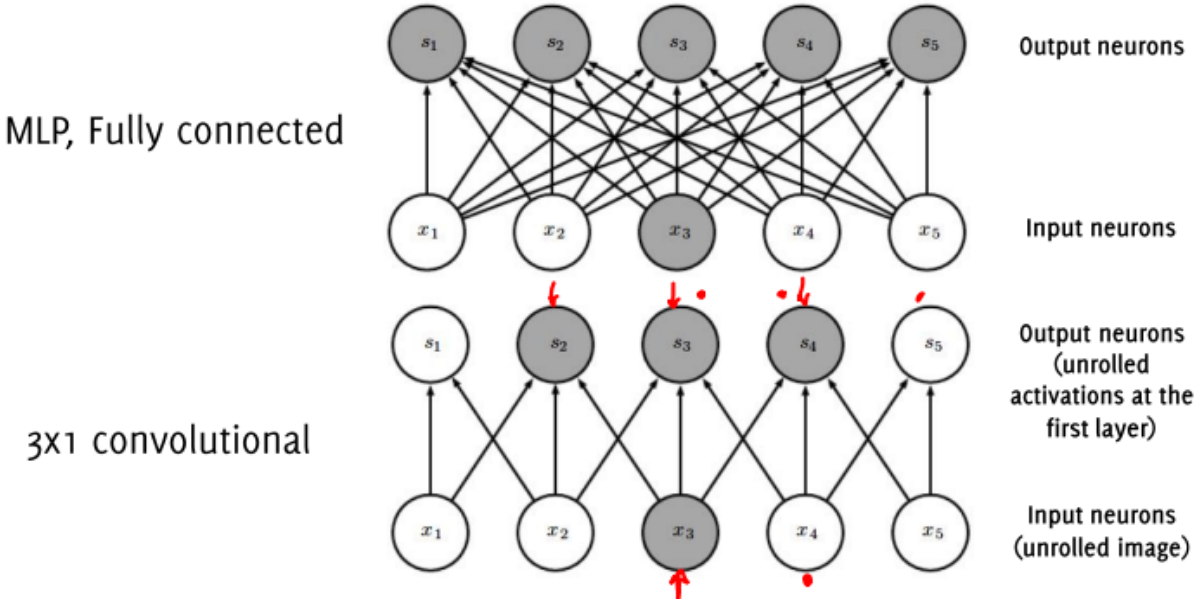

Filter concatenation
3x3 convolutions
5x5 convolutions
1x1 convolutions
1x1 convolutions
1x1 convolutions
1x1 convolutions
3x3 max pooling
Previous layer

CNN Training

CNN operate similarly to FCNN, given that the training process considers an error function through which we want to estimate weights and biases of the model. However, there are some key differences:

- Input: FCNNs can process any type of input, while CNNs are specifically designed to process data with a grid-like topology such as images.
- Architecture: FCNNs are composed of fully connected layers, while CNNs are composed of a combination of convolutional, pooling, and fully connected layers.
- Parameters: FCNNs typically have a large number of parameters (weights and biases), while CNNs have a much smaller number of parameters due to the use of shared weights and pooling layers. All the neurons in the same slice of a feature map (same filter) use the same weights and biases, under the assumption that if a feature is useful to compute in a spatial position, it would also be relevant at a different position.
- Training: FCNNs are typically trained using gradient descent optimization algorithms, while CNNs can be trained using a variety of optimization algorithms such as stochastic gradient descent, Adam, and RMSprop.

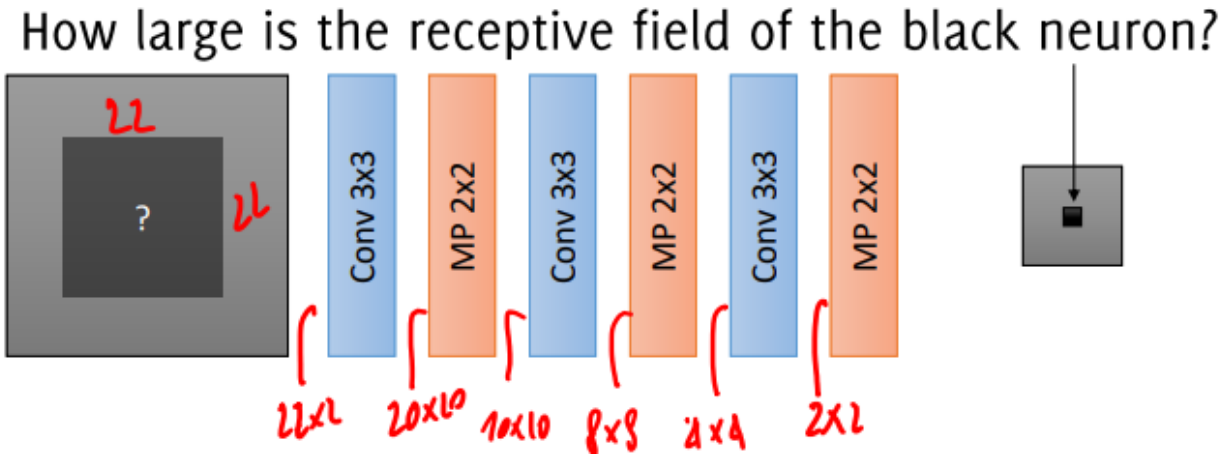
Connectivity among CNN layers is also sparse: in fact, not all neurons in one layer are connected to all neurons in the next layer, as each neuron in the output layer is only connected to a subset of the neurons in the input layer (namely, the ones that are multiplied by the same filter “instance”).



If the CNN was a FC, the W matrix would be a large matrix (#rows = number of output neurons, #cols = number of input neurons), which would be mostly 0 except for where local connectivity takes place. Also, a lot of data would be the same because of weight sharing.

Receptive Field

Due to sparse connectivity, unlike in FCNN where each output depends on the entire input, in CNN each output only considers a specific region of it (sparse connectivity). This region is the receptive field for the output. The deeper the layer, the wider the neuron's receptive field. As we move to different layers, spatial resolution is reduced, the number of maps increases, and we search for high-level patterns, not caring about their exact location. Having more filters in the last layers means that we're able to combine high-level patterns, since in a layer features can be considered to have the same importance.



Training a CNN

Each CNN can be seen as a MLP. This means that classic gradient descent and backpropagation algorithms can be used to minimize a loss function. Weight sharing has to be considered, as well as maxpooling characteristics and ReLu activation.

Data Augmentation

Deep learning models are data-hungry, they need a lot of data to perform efficiently. This is necessary to define millions of parameters characterizing these networks: a complex net with a small dataset will perform poorly and overfit, while a simple net will not be useful in learning complex features. To deal with this issue, there can be 2 different approaches, and data augmentation is the first one.

It can be performed using geometric transformations such as:

- Shift/Rotation/Affine/perspective distortion
- Shear
- Flip
- Scaling

or photometric transformations:

- Adding noise

- Modifying average intensity
- Superimposing other images
- Modifying image contrast

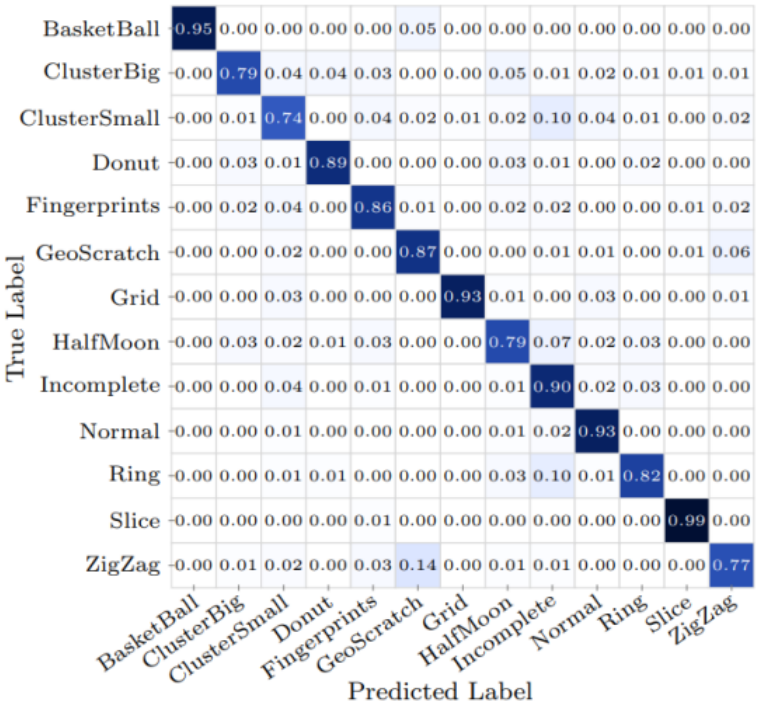
The input label should be preserved, so choosing transformations that do not impact key features is important. This type of transformation however may not be enough to capture interclass variability: even if CNN is trained using augmentation, perfect invariance won't be achieved. **TTA** (test time augmentation) can also be performed at test time to improve the prediction accuracy:

- Perform random augmentation of each test image $A_l(I)_l$;
- Average the predictions of each augmented image $p_i = CNN(A_l(I))$;
- Take the average vector of posterior for defining the final guess $p = Avg(\{p_l\}_l)$.

Augmentation reduces the risk of overfitting and increases the training set size. It can also be used for class imbalance by creating more realistic examples from the minority class. Transformations can also be class specific to preserve the image label. This way, the net can become invariant to selected transformations.

Confusion Matrix

It's a tool to visualize the predictions of the net. The i-th row and j-th column represent the elements of class i classified as elements of class j. The ideal confusion matrix has all 1s on the diagonal.



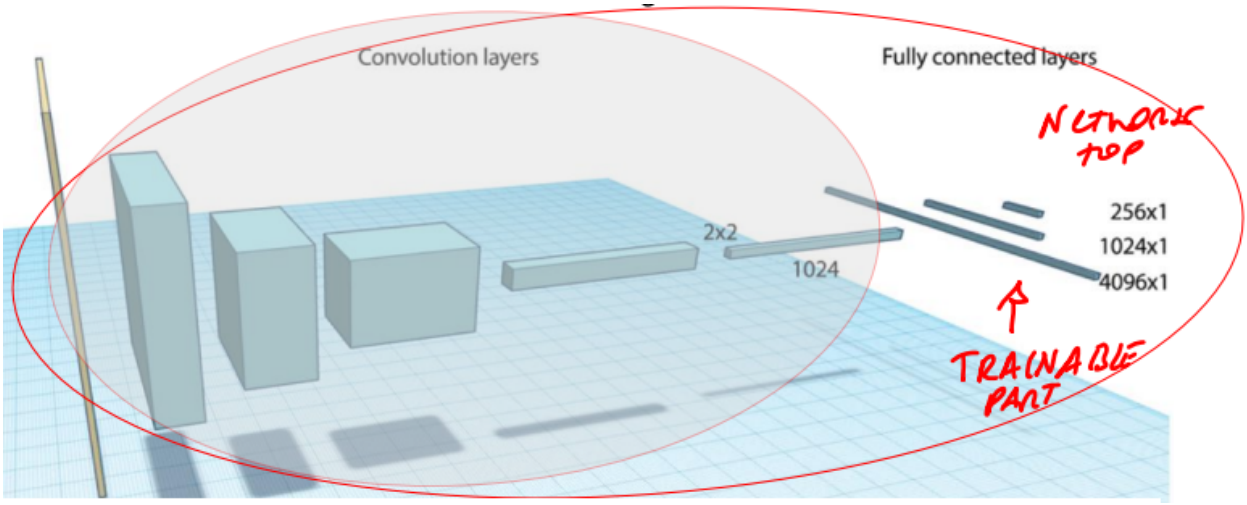
Transfer Learning

Distance in the latent representation of a CNN is meaningful for image semantics, in contrast with the Euclidean Distance. Features can be used for different classification tasks. The output of the FC layer has the same size as the number of classes and is very task-specific. The latent representation, however, is a data-driven description of the images, a feature vector. These features are defined to maximize the classification performance and trained via backpropagation. So a CNN is composed by a feature extraction section, which is powerful and general purpose, and a classifier, which is FC and solves the specific classification task.

We can use a **feature extraction network** to take the convolutional layers of a network trained to solve large classification problems, because it should be an effective and general feature extractor. The FC layers on the other hand, are very specific and meant to solve a specific purpose.

1. Take a successful pre-trained model like VGG;
2. Remove the FC layers and design new ones to match a new problem;

3. Freeze the weights in convolutional layers;
4. Train the whole net on new training data;



In transfer learning, only the FC layers are trained. A good option when little training data is provided and the pre-trained model is expected to match the problem at hand. Another choice is fine tuning, retraining the whole CNN while the convolutional layers are initialized to the pre-trained model. This is useful when enough training data is provided or when the task at hand differs. Lower learning rates are used.

FCNN for Image Segmentation

A fully convolutional neural network (FCNN) is a type of neural network that is designed to process data that has a grid-like topology, such as an image. It is called "fully convolutional" because it consists entirely of convolutional layers, which are a type of neural network layer that is designed to process data with a grid-like topology.

FCNs have several key properties that make them well-suited for image processing tasks:

1. They are translation invariant: the output of an FCN is the same regardless of the position of the input in the image;
2. They can process images of any size: an FCN can be used to process images of any size, because the convolutional layers can handle inputs of any size. The FCL are the ones that demand for the input to have a fixed size, removing them removes also this constraint;
3. They can learn spatial hierarchies: FCNs can learn complex patterns in images by using multiple layers with different receptive field sizes. This allows them to learn both local patterns (e.g., edges) and global patterns (e.g., objects);

FCNs are widely used for image segmentation tasks, where the goal is to classify each pixel in an image as belonging to a particular object or background. They are also used for other image processing tasks, such as object detection and image generation.

Although the operations used in FC and convolutional layers are different, it is possible to view an FC layer as a special case of a convolutional layer. To do this, we can think of the input to the FC layer as a 1x1 image and the weights of the FC layer as a 1x1 convolutional kernel. The output of the FC layer is then computed by applying this 1x1 kernel to the input image. So we can say:

$$1 \times 1 \times N \times L$$

Where L is the number of filters (thus the output size) and N is the number of neurons in the previous layer.

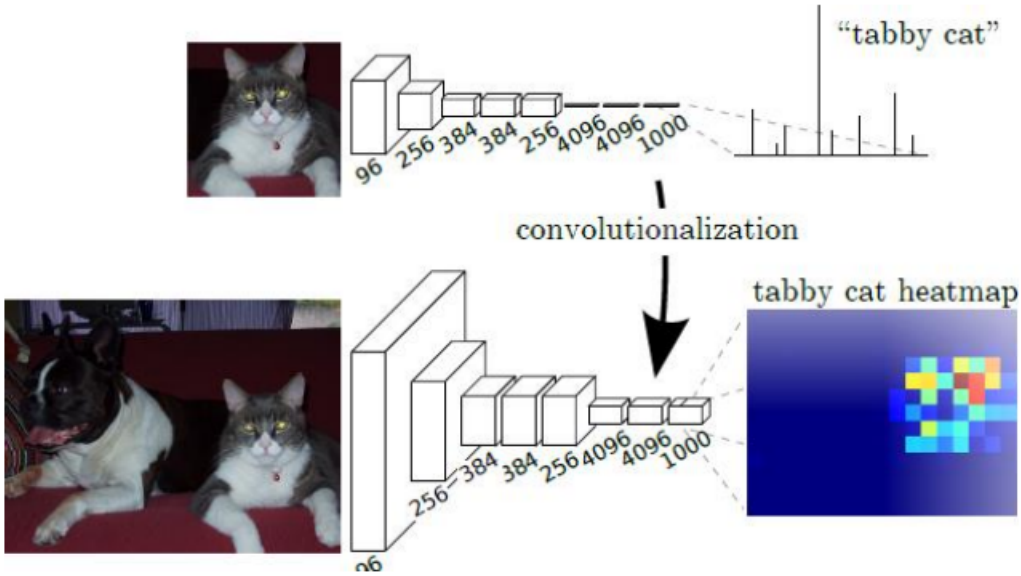
By removing the fully connected (FC) layers from a neural network and using only convolutional layers, it is possible to create an FCN that is able to process images of any size and retain the spatial information in the feature maps. This makes FCNs well-suited for image segmentation tasks, because they are able to classify each pixel in the image based on its relationships with surrounding pixels.

In contrast, traditional neural networks that use FC layers are not able to process images of variable size and do not retain the spatial relationships between pixels in the image. This can make them less effective for image segmentation tasks, because they are not able to use the spatial relationships between pixels to make accurate predictions.

Heatmaps

Heatmaps are graphical representations of data where the values are represented as colors, with higher values being indicated by warmer colors (e.g., red, yellow) and lower values being indicated by cooler colors (e.g., blue, green). Heatmaps are often used to visualize the outputs of a neural network, particularly in tasks such as image classification or object detection, where the goal is to identify specific objects or features in an image.

Heatmaps are typically produced by convolutional neural networks (CNNs), which are a type of neural network that is designed to process data with a grid-like topology (e.g., an image). CNNs use convolutional layers, which apply a series of filters to an input image to extract features from the image. The outputs of these filters can be visualized as heatmaps, which show the locations in the image where the filters are most activated. Each pixel of the heatmap corresponds to a receptive field of the input image.



The stack of convolutions operates on the whole image as a filter, and provides with more efficiency in respect to patch extraction and classification, avoiding multiple repeated computations on overlapping patches.

In a convolutional neural network (CNN) for image segmentation, the final layer is typically a convolutional layer that generates a heatmap with one channel for each class. The heatmap slices encode the probability of each pixel belonging to each class, with values between 0 and 1, and have a lower resolution than the original image.

To generate the final label map, you can threshold the probabilities in the probability distribution generated by the softmax function to assign each pixel to the class with the highest probability. To do this, you can select the class with the highest probability for each pixel. This would correspond to the class with the highest value in the heatmap for that pixel.

Because of this, it is important to have a heatmap with one channel for each class. This is because each channel in the heatmap encodes the probability of each pixel belonging to a particular class, and all of them have to be considered when making the final prediction.

Image Segmentation

Semantic segmentation assigns to every image pixel (row, column) a label from a determined set.

$$\Lambda = \{\text{"wheel"}, \text{"cars"}, ..., \text{"castle"}\}$$
$$I \rightarrow S \in \Lambda^{R \times C}$$

Given an image I, associate to each pixel (r, c) a label from Λ. The result of a segmentation is a map of labels containing, in each pixel, the estimated class. Segmentation does not recognize different instances belonging to the same class. The training set is made of pairs (I, GT), where GT is a pixel-wise annotated image over the categories in Λ. Training set is very difficult to annotate.

To use a CNN for image segmentation, we can use different approaches.

Direct Heatmap Prediction

In this approach, the CNN directly predicts the probability of each pixel belonging to each class, and the final label map is generated by applying an activation function (such as the softmax function) to the heatmap and selecting the class with the highest probability for each pixel. The label predicted is assigned to the whole receptive field of the pixel, since the heatmap has a lower resolution than the original image.

Shift and Stitch

Assume there's a downsampling ratio f between the input size and the output heatmap. You can take the same input and shift it, then get the output and repeat the process multiple times. For every output there's a corresponding value in the shift vector. These can be used to get a better resolution final image, by combining them together.

1. Compute the heatmaps for all f^2 possible shifts of the input $(0 \leq r, c < f)$;
2. Map the predictions from the f^2 heatmaps to the image: each pixel in the heatmap provides prediction of the central pixel of the receptive field;
3. Interleave the heatmaps to form an image as large as the input.

The upsampling process, however, is very rigid. The same process can be implemented by removing the strides in pooling layers (this is equivalent as computing the output of all the shifted versions at once) and rarefying the filters of the following convolution layer by upsampling and 0 padding, repeating the same operation over each dimension and for every subsampling layer.

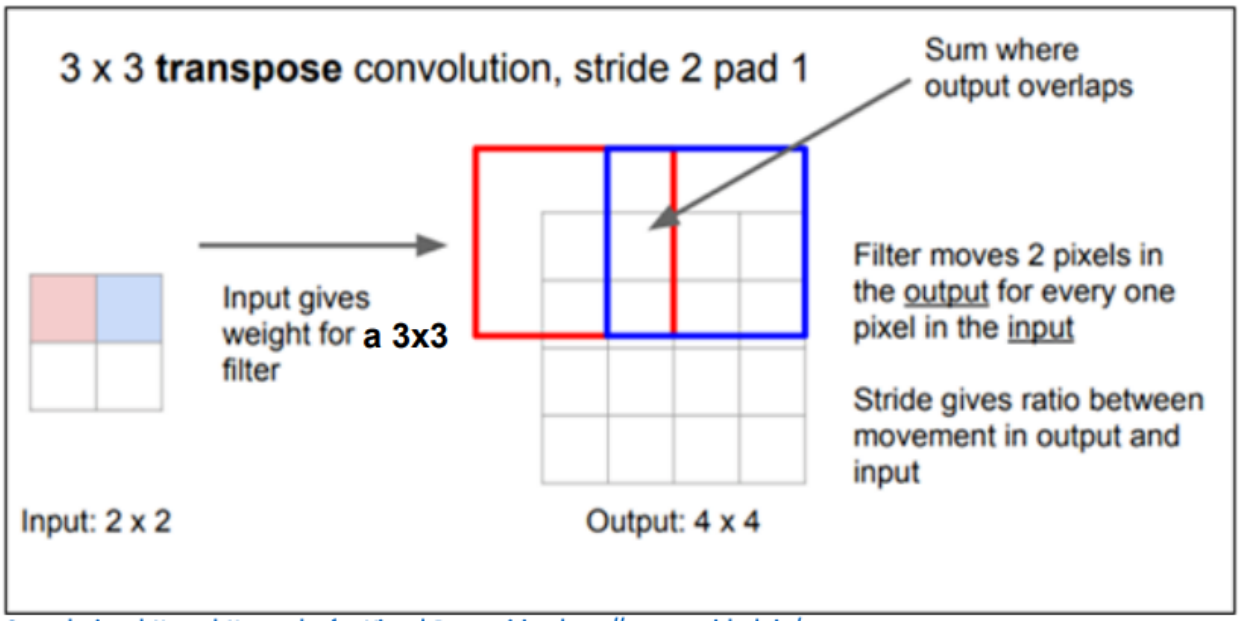
Only Convolutions

Having only convolutional layers can be an advantage because the net can process images of any size. However, they reduce a lot of the spatial resolution of the input, so going deeper may be necessary to extract high level information, but it can make the result lose too much input scale. Combining the 2 types of layers keeps both the global information (the what) and the local information (the where).

Upsampling

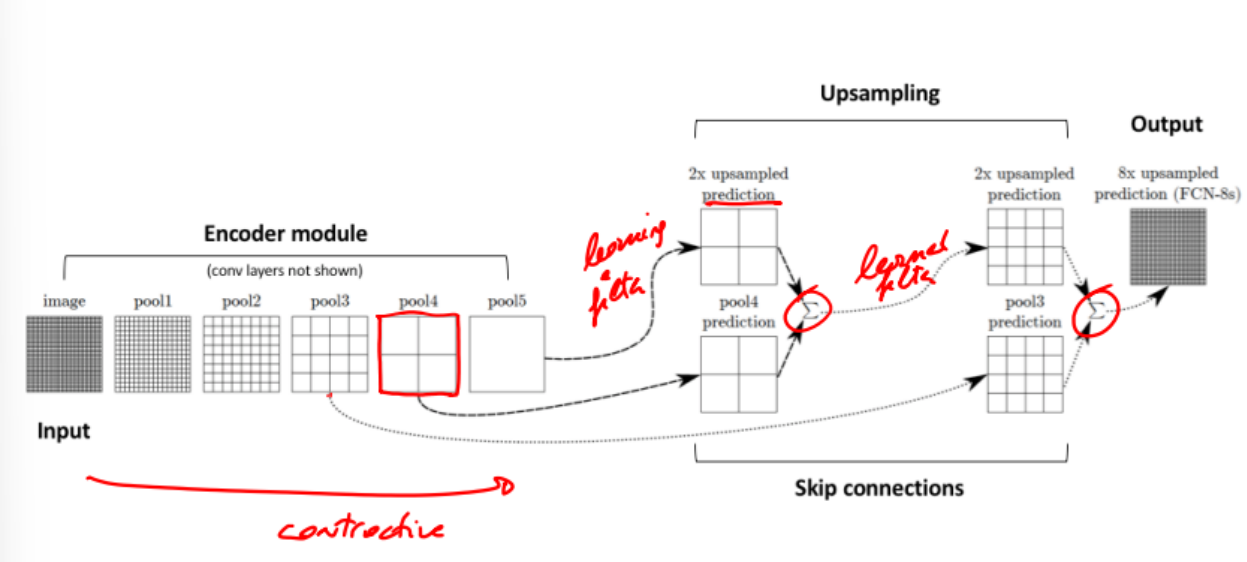
Upsampling is a process that is used in convolutional neural networks (CNNs) to increase the size of the feature maps produced by the network. It is often used in conjunction with downsampling, which is a process that reduces the size of the feature maps. Using a CNN that performs heatmap prediction and then upsamples the output heatmaps to produce a high-resolution output map may be a good strategy. This can be useful in tasks such as image segmentation, where the goal is to classify each pixel in the image and produce a detailed output map.

- **Nearest Neighbor**: a method of increasing the size of an image or feature map by replicating the values of the original pixels. It is a simple and fast method of upsampling that can be used in a convolutional neural network (CNN).
  To perform nearest neighbor upsampling, the original image or feature map is first resized to the desired size by inserting zero-valued pixels between the original pixels. The value of each new pixel is then set to the value of the closest original pixel;
- **Bed of Nails**: pixels are filled with 0 values instead of the closest number;
- **Max Unpooling**: often used in combination with max pooling, it also keeps the spatial information about the original reduced feature map in the upsampled one;
- **Transpose Convolution**; opposite operation of convolution, the input feature map is first upsampled by inserting zero-valued pixels between the original pixels. This increases the size of the feature map and creates a larger grid of pixels that can be convolved with a set of filters. The filters are then applied to the upsampled feature map by performing a standard convolution operation, which involves sliding the filters over the feature map and computing the dot product between the values in the filters and the values in the feature map at each position.
  The resulting output of the transposed convolution is a larger feature map that has the same size as the original feature map, but with a higher resolution. Upsampling filters can be learned, and, depending on the stride and filter dimension, some data in the feature map may be the result of more than 1 of them, through a weighted sum.



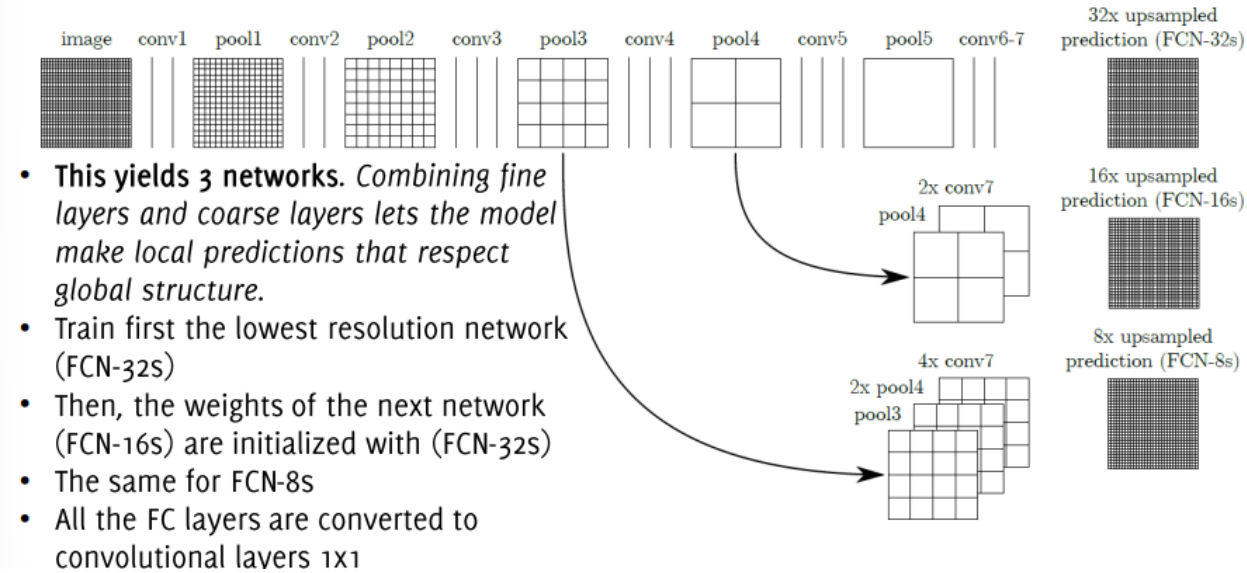
Skip Connections

Another way to perform more precise upsampling is to use a traditional convolutional neural network (CNN) for feature extraction and a separate network with only transposed convolution layers for upsampling, and then combine the two networks by adding the upsampled feature maps from the second network to the output of the pooling layers in the first network. This approach can be useful for tasks such as image segmentation, where the goal is to classify each pixel in the input image and produce a high-resolution output map.



By progressively adding the upsampled feature maps from the transposed convolution network to the output of the pooling layers in the traditional CNN, it is possible to preserve the spatial relationships between pixels in the input data and produce a more accurate output map. Also, the filters of the upsampling section can learn better fitting representations of the input and extract more relevant information.

The learning happens in the same way as with traditional CNN. The deeper the expanding part, the better the results.



Training a CNN

To train a FCNN for segmentation, you will need a training set, and crop as many patches x_i from annotated images and assign to each one a label corriesponding to each center.

Patch Based

1. Prepare a training set for a classification net;
2. Crop as many patches x_i from annotated images and assign to each one the label corresponding to the center;
3. Train a CNN for classification from scratches, or fine-tune a pre-trained model over the segmentation classes;
4. Convolutionalization, once trained, move the FC layers to 1x1 convolutions. To "move" an FC layer into 1x1 convolutions, you can replace the FC layer with a series of 1x1 convolutional layers that perform the same function. 1x1 convolutions are convolutions that use a kernel size of 1x1 and are used to reduce the number of channels in a feature map or to combine multiple feature maps.
5. Desing the upsampling part of the net.

The classification network is trained to minimize the classification loss over a mini-batch, where batches are randomly assembled during training. It is very inefficient since convolutions on overlapping patches are repeated.

$$\theta = argmin_\theta \sum_{x_j \in B} l(x_j, \theta)$$

Patch based is used when having a shift and stitch approach or other technologies based on heatmap predictions.

Full Image

It’s possible to train the net over the whole image for a net with upsampling layers.

$$\theta = argmin_{x_j \in I} \sum_{x_j \in R} l(x_j, \theta)$$

where x_j are the pixels in a region R of the input image, and the loss is evaluated over the corresponding labels in the annotation. So, each region provides a mini batch estimate for computing gradient, all the receptive fields of the units below the loss for an image.

1. FC-CNN are trained in an end to end manner to predict the segmented output S;
2. This loss is the sum of losses over different pixels, and backpropagation is used to fix it;
3. More efficiency because there's no need for a first classification network and does not have to recompute on overlaps;

However, the minibatches are more stochastic and have more variance, and it's not possible to perform patch resampling for correcting imbalance. One should go for weighting the loss over different labels.

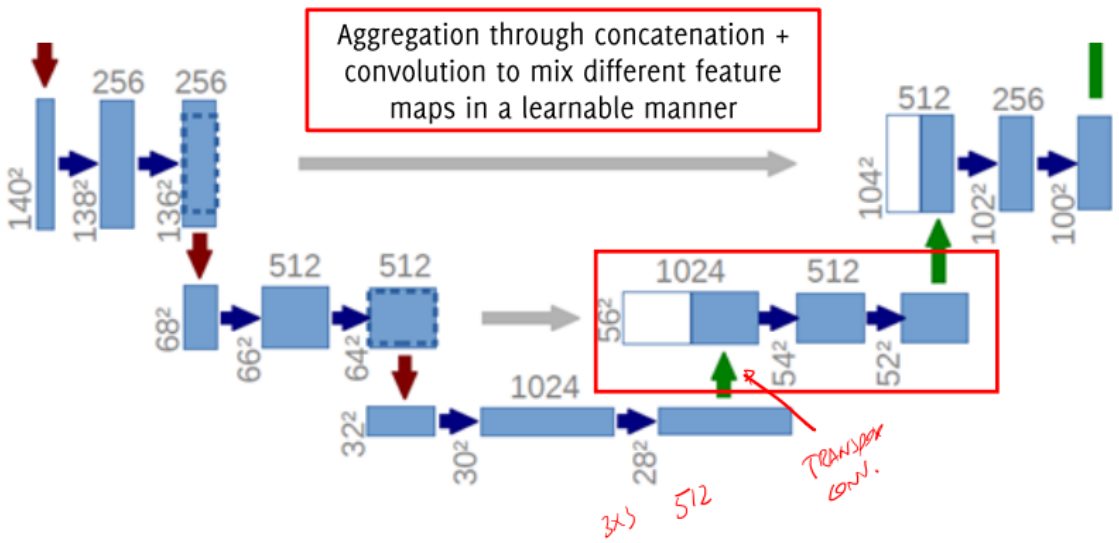
$$\theta = argmin_{x_j \in I} \sum_{x_j \in R} w(x_j) l(x_j, \theta)$$

U-NET

Has an expansion part and a contraction part, without using FC layers (symmetric structure). It uses a lot of feature channels and data augmentation.

1. **Contracting part:** repeats blocks of :
   - 3 x 3 convolution + ReLu
   - 3 x 3 convolution + ReLu
   - Maxpooling 2 x 2

   At each downsampling the number of feature maps is doubled. It uses skip connections to aggregate through concatenation the last conv layer with the first conv layer of the respective expansion block.



It uses not only skip-connections but also learnable upsampling filters to upscale the spatial resolution.

2. **Expanding Part:** Repeats blocks of:
    - 2 x 2 transpose convolution, halving the number of feature maps but doubling the spatial resolution
    - Concatenation of corresponding cropped features
    - 3 x 3 conv + ReLu
    - 3 x 3 conv + ReLu

Last layer is a conv $1 \times 1 \times N$ to yield predictions out of the feature maps. The training is done using

$$\theta = min_{x_j \in I} \sum_{x_j} w(x_j) l(x_j, \theta)$$

where the weight

$$w(\boldsymbol{x}) = w_c(\boldsymbol{x}) + w_0\, e^{-\frac{\left(d_1(\boldsymbol{x})+d_2(\boldsymbol{x})\right)^2}{2\sigma^2}}$$

- w_c is used to balance class proportions (remember no patch resampling in full-image training)
- d_1 is the distance to the border of the closest cell
- d_2 is the distance to the border of the second closest cell

Weights are large when the distance to the first two closest small cells is small. The first factor takes into account class unbalance in the training set, while the second one enhances classification performance at borders.

GAP, Localization and CNN Explanations

Localization

The input image contains a single relevant object to be classified into a fixed set of categories. The task is to:

1. Assign the object class to the image;
2. Locate the object in the image by its bounding box

A training set of annotated images with labels and a bounding box around each object is required. Extended localization problems involve regression over more complicated geometries. We have to give an input image $I \in \mathbb{R}^{R \times C \times 3}$:

- A label l from a fixed set of categories Λ;
- The coordinates (x, y, h, w) of the bounding box enclosing that object, $I \rightarrow (x, y, h, w, l)$

So, there are substantially 2 predictions to make, the class label and the bounding box. To find the target class, softmax loss δ can be used, through categorical crossentropy, while BB coordinates are found through regression loss $||[x', y', w', h'] - [x, y, w, h]||_2$

The training loss has to be a single scalar since we compute gradient of a scalar function with respect to network parameters, so we minimize a multitask loss to merge 2 losses:

$$L(x) = \alpha\delta(x) + (1 - \alpha)R(x)$$

where α is a hyperparameter of the network, which directly influences the loss definition and might be difficult to tune. It's also possible to adopt pre-trained models and train the two FC separately but fine-tuning is still needed.

Weakly Supervised Localization

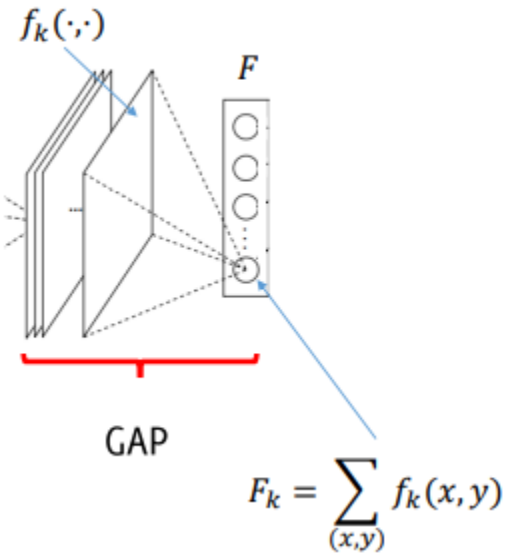
Weakly supervised localization is a type of image localization task in which the training data consists of only image-level labels, rather than precise bounding box annotations for each object in the image. This can make it more challenging to localize objects in the image, as the network must learn to identify the objects based on their appearance and context in the image, rather than relying on precise bounding box annotations.

The GAP revisited - CAM

The advantages of a GAP layer extend beyond simply acting as structural regularizer that prevents overfitting. In fact, CNNs retain a remarkable localization ability until the final layer. A CNN trained on object localization is successfully able to localize the discriminative regions for action classification as the objects that the humans are interacting with rather than the humans themselves. A CAM, or class activation map, identifies exactly which regions of an image are being used for discrimination. CAMs are very easy to compute, the only requirement is a FC layer after the GAP and a minor tweak. A class activation map is a heatmap that shows the regions in an input image that are most relevant for predicting a particular class. It is typically generated by taking the output of a convolutional neural network (CNN) and overlaying it onto the input image, highlighting the areas that had the most influence on the network's prediction.

A very simple architecture made only of convolutions and activation functions leads to a final layer having:

- n feature maps $f_k(.,.)$, having resolution similar to the input image
- a vector after GAP made of n averages F_k



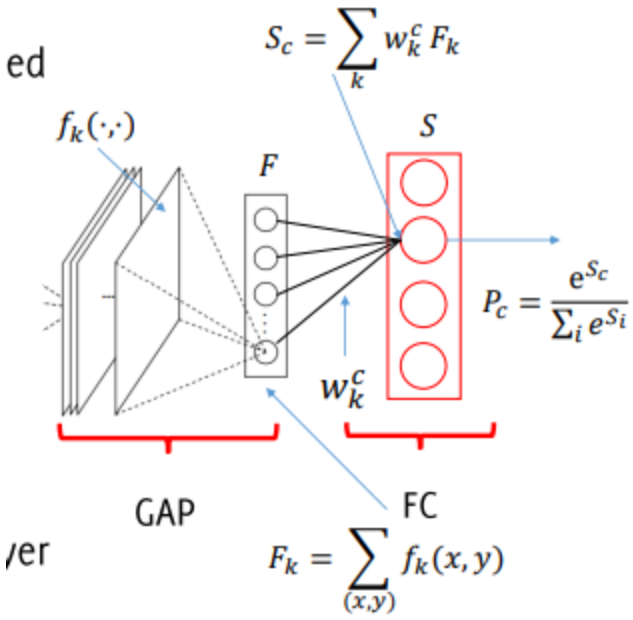
The Global Average Pooling (GAP) layer produces a vector by taking the average of all the values in the input tensor along the spatial dimensions (e.g., height and width). The resulting vector has the same depth as the input tensor (i.e., the same number of channels).

To use the GAP layer for weakly supervised localization, you would typically apply it to the output of the final convolutional layer in your CNN, which is typically a tensor with multiple channels corresponding to different features or class scores. The GAP layer then produces a vector with one element per channel, which can be fed into an fully connected (FC) layer for further processing.

The FC layer can then be used to learn the spatial importance of each class by looking at the correlations between the elements of the GAP vector and the spatial positions of the feature activations in the input image. For example, if certain elements of the GAP vector are highly correlated with certain spatial positions in the input image, it indicates that those positions are important for the corresponding class.

In practice, you would typically train the CNN with a loss function that encourages the network to produce class scores that are highly correlated with the ground truth spatial positions of the objects of interest. This would allow the FC layer to learn the spatial importance of each class in an end-to-end manner, without the need for explicit

supervision at the pixel level. Class probability is then found via softmax. It produces a probability distribution over the possible classes. The idea is that by using the softmax function, we can identify which regions of the input image are most important for predicting each class. This can be useful for localization tasks, as it allows us to identify which parts of the image are most relevant for predicting each class.



$S_c = \sum_k w_k^c F_k$, where w_k^c encodes the importance of F_k for the class c, and $\{w_k^c\}_{k,c}$ are the parameters of the last FC layer.

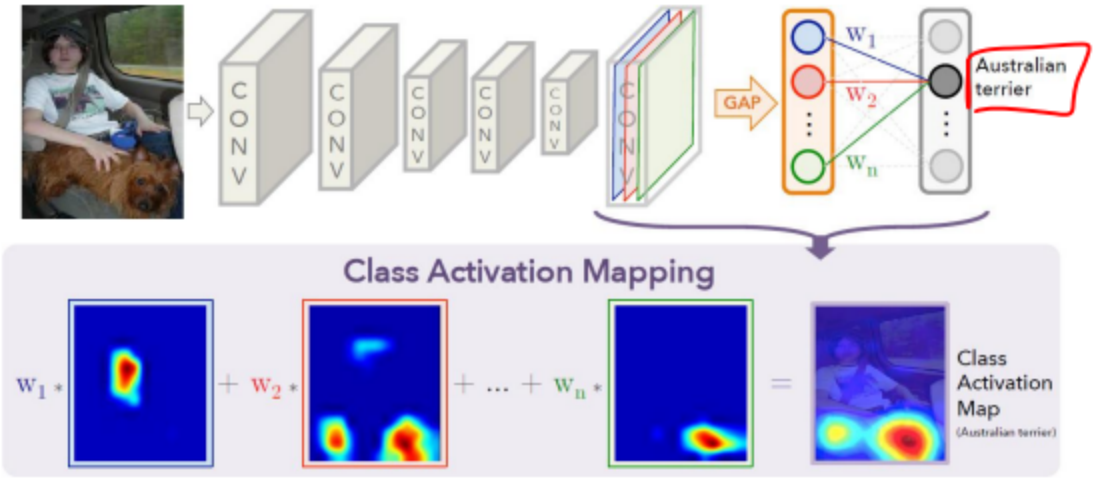
However, $S_c = \sum_k w_k^c \sum_{x,y} f_k(x,y) = \sum_{x,y} \sum_k w_k^c f_k(x,y)$ and CAM is defined as

$M_c(x,y) = \sum_k w_k^c f_k(x,y)$, where it indicates the importance of the activations at (x,y) for predicting the class c. Thanks to this, we can consider the spatial information by learning the weight values considering the location.

The weights of the FC layer represent the importance of each feature map to yield the final prediction. Upsampling may be necessary to match the input image.

Since the CAM output gives both the class activation mapping and the image classification, a softmax is necessary to generate the class distribution. The spatial information is kept by removing the FC layers and using a GAP + FC + softmax. For this reason, classification performance may drop, but so do the number of parameters.

The depth of the last convolutional activations can differ from the number of classes because the activations are passed through the softmax function before being used to produce the class activation map. The softmax function converts the activations into a probability distribution over the classes, so the depth of the activations does not need to match the number of classes. In the class activation map, the probability of each class is multiplied by the activations at each location in the feature map, and the resulting maps for each class are summed to produce the final class activation map.

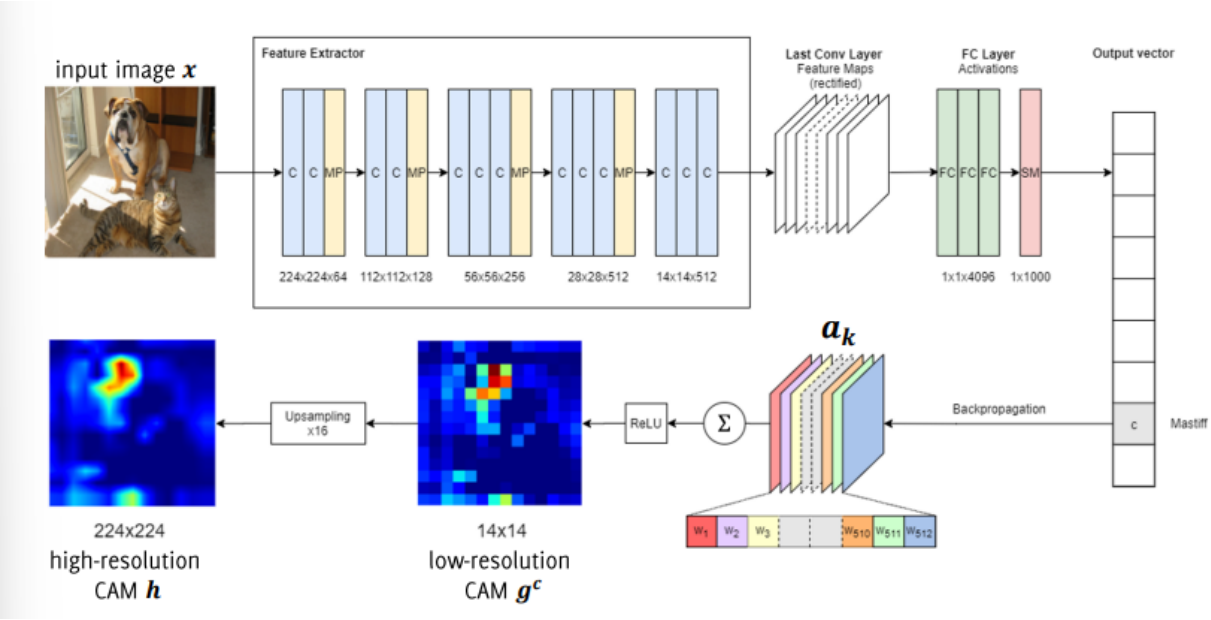


Grad-CAM

One main difference between CAM and Grad-CAM is that CAM uses the weights of the FC layer to generate the map, while Grad-CAM uses the gradients of the target class with respect to the feature maps of the last convolutional layer. This means that CAM focuses on the overall importance of the regions in the input image for the prediction made by the CNN, while Grad-CAM specifically focuses on the contribution of the regions to the prediction of the target class.

It combines the gradient information with the spatial information from the last convolutional layer to produce a class activation map that highlights the regions of the input image that are most important for the final prediction.
The FC layer is still needed in Grad-CAM to find weights that highlight the classification. The FC layer provides the weights that are used to weigh the importance of each feature map in the final class activation map. The gradient of the last layer heatmaps is then weighted by these weights to produce the final class activation map.

In Grad-CAM, the weights of the computation on the heatmaps are the gradient of the predicted class with respect to the activations of the last convolutional layer. These weights are used to weight the importance of each activation map in the final Grad-CAM output, which is a class activation map that shows the regions in the input image that are most important for the predicted class.



$$g^c = \text{RELU}\left(\sum_k w_k^c a_k\right), \qquad w_k^c = \frac{1}{nm}\sum_i \sum_j \frac{\partial y_c}{\partial a_k(i,j)}$$

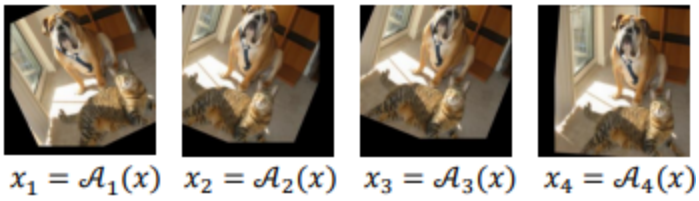
$$a_k = f_k(x, y)$$

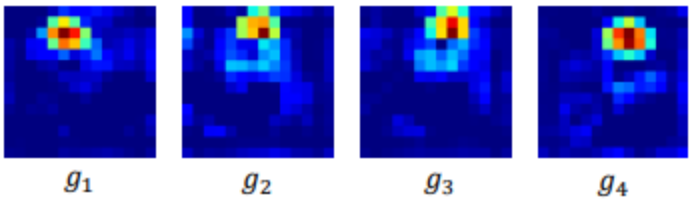
No need to modify the network top when adding or removing classes. In CAM, the network topology needs to be modified because the output of the last convolutional layer needs to be passed through a global average pooling (GAP) layer and then through a fully-connected (FC) layer to produce the class activation map. In contrast, in Grad-CAM, the gradient of the last convolutional layer is used to produce the class activation map, so no additional layers are needed.

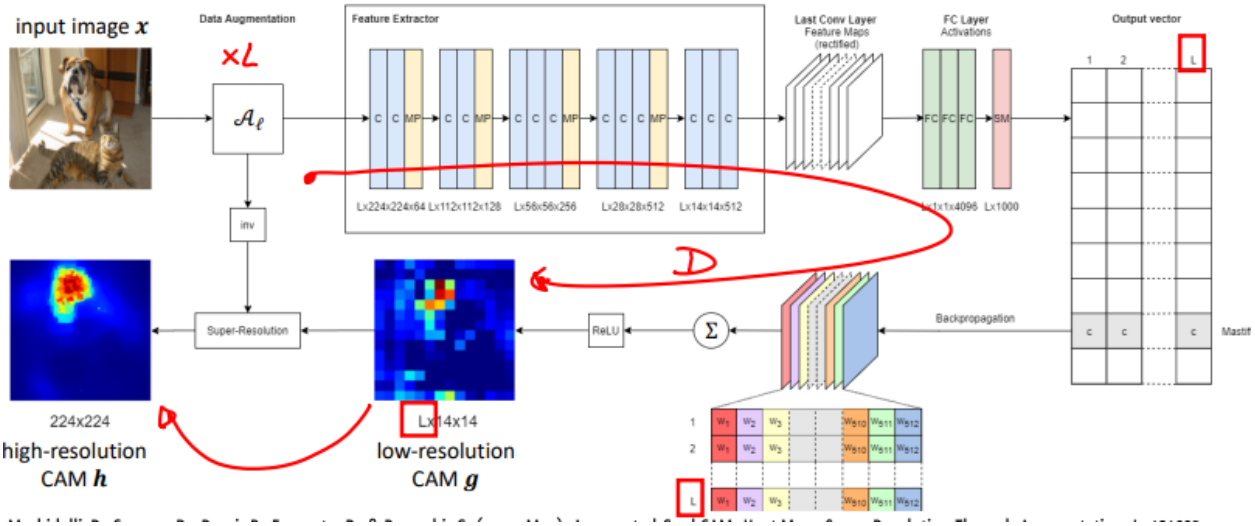
Augmented Grad-CAM

We consider the augmentation operator $A_l : \mathbb{R}^{N \times M} \rightarrow \mathbb{R}^{N \times M}$, including random rotations and translations of the input image x. Augmented Grad-CAM: increase heatmap resolution through image augmentation. All the responses that the CNN generates to the multiple augmented versions of the same input image are very instructive for reconstructing a high-resolution heatmap.

We perform heapmap super resolution by taking advantage of the information shared in multiple low-resolution heatmaps computed from the same input under different-but known-transformations. CNNs are in general invariant to roto-translations, in terms of predictions, but each g_l actually contains different information.







We model heatmaps computed by Grad-CAM as the result of an unknown downsampling operator $D : \mathbb{R}^{NxM} \rightarrow \mathbb{R}^{nxm}$. The High resolution heatmap h is recovered by solving an inverse problem.S

$$\underset{h}{\operatorname{argmin}} \frac{1}{2} \sum_{l=1}^{L} \|\mathcal{D}\mathcal{A}_\ell h - g_\ell\|_2^2 + \lambda TV_{\ell_1}(h) + \frac{\mu}{2}\|h\|_2^2 \qquad (1)$$

TV_{ℓ_1}: Anistropic Total Variation regularization is used to preserve the edges in the target heat-map (high-resolution)

$$TV_{\ell_1}(\boldsymbol{h}) = \Sigma_{i,j} \|\partial_x \boldsymbol{h}(i,j)\| + \|\partial_y \boldsymbol{h}(i,j)\| \qquad (2)$$

This is solved through Subgradient Descent since the function is convex and non-smooth

Other approaches like GradCam ++ and Perception Visualization can be used for even better results.

CNN for Object Detection

Assign to an input image $I \in \mathbb{R}^{R \times C \times 3}$:

- Multiple labels $\{l_i\}$ from a fixed set of categories Λ, each corresponding to an instance of that object.
- The coordinates $\{(x, y, h, w)_i\}$ of the bounding box enclosing each object

$$I \rightarrow \{(x, y, h, w, l)_1, ..., (x, y, h, w, l)_N\}$$

Given a fixed set of categories and an input image that contains an unknown and varying number of instances, draw a bounding box on each object instance. A training set of annotated images and bounding boxes for each object is required, each image requires a varying number of outputs.

Assign to an input image I:

- **multiple** labels $\{l_i\}$ from a fixed set of categories $\Lambda = \{$"wheel", "cars", ..., "castle", "baboon"$\}$, each corresponding to **an instance of that object**
- the coordinates $\{(x, y, h, w)_i\}$ of the **bounding box enclosing each object**
- the **set of pixels** S in each bounding box corresponding to that label

$$I \rightarrow \{(x, y, h, w, l, S)_1, \ldots, (x, y, h, w, l, S)_N\}$$

Sliding Window

The sliding window is a technique used for object detection in images. It involves scanning the image with a window of a fixed size, and applying a classifier to each window to predict whether or not the window contains an object of interest.

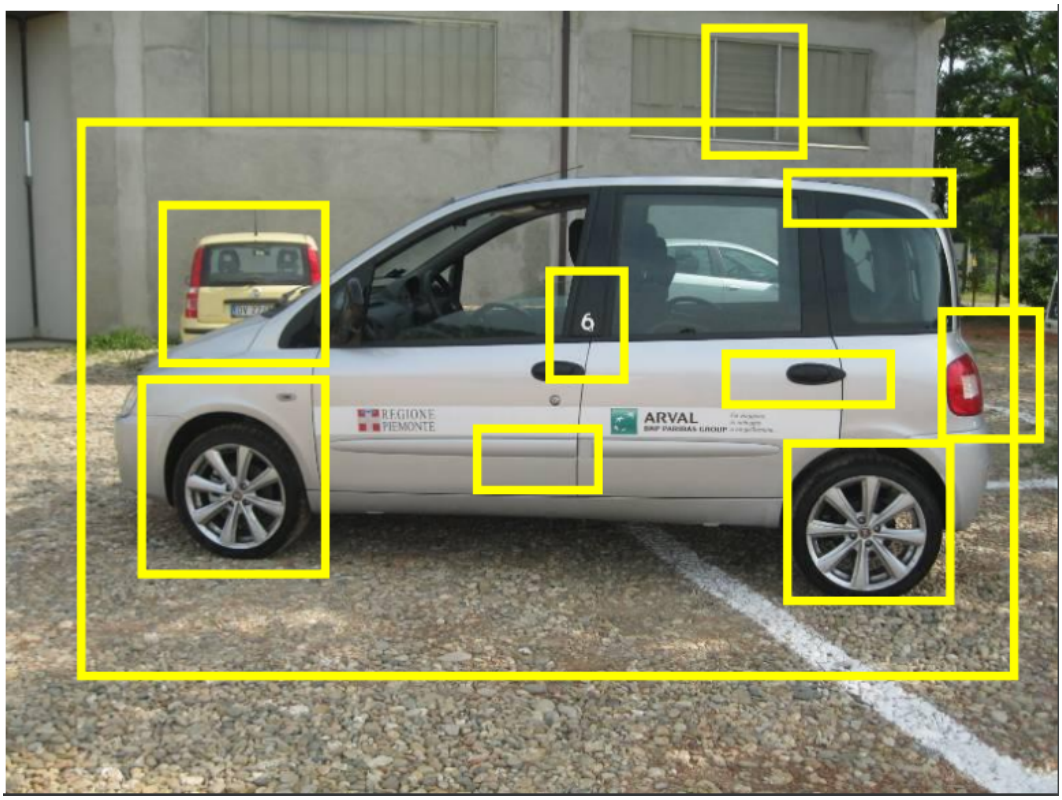
The window is then moved by a certain amount, called the stride, and the process is repeated until the entire image has been covered. The classifier's output for each

window is then used to create a heatmap of the image, with higher values indicating a higher probability of the object being present in that location.

However, this is very inefficient, and choosing the crop size is difficult: the computation should be repeated over a lot of sizes. Also, a background class has to be included.

Region Proposal

Region proposal algorithms are meant to identify bounding boxes that correspond to a candidate object in the image. Algorithms with very high recall (but low precison) were there before the deep learning advent. The idea is to apply a region proposal algorithm and then classify by a CNN the image inside every proposed region.

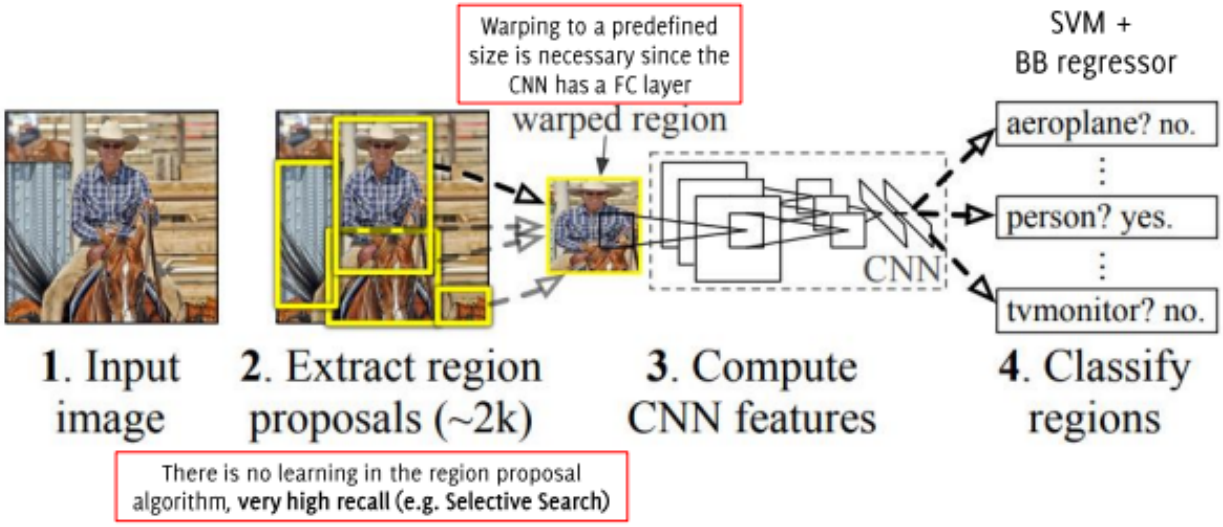


R-CNN

Region-based Convolutional Neural Network (R-CNN) is a method for object detection in images. It is a two-stage process that first uses a region proposal algorithm to generate a set of potential bounding boxes (regions) that may contain objects, and then uses a CNN to classify the objects in these regions and refine the bounding boxes.

In the first stage, a region proposal algorithm generates a set of candidate bounding boxes that may contain objects. These bounding boxes are then passed through a CNN to classify the objects within them and refine the bounding boxes. This process is repeated until the desired number of bounding boxes is generated.

In the second stage, the CNN is trained to classify the objects within the bounding boxes and refine the bounding boxes. The CNN is trained on a large dataset of labeled images, and it learns to recognize the characteristics of different objects. Once the CNN has been trained, it can be used to classify objects in new images.



The SVM is trained to minimize the classification error over the extracted ROI. The regions are refined by a regression network to correct the bounding box estimate from the ROI algorithm (they're also rescaled before the CNN, so dimensional changes are needed). The pretrained CNN is fine-tuned by placing a FC layer after feature extraction. No end-to-end training of the SVM. A background class is included to get rid of regions that do not match objects.

However, there are some limitations:

- Ad-hoc training objectives and not end-to-end training
  - Fine-tuning network with softmax classifier before training SVM;
  - Train post-hoc linear SVM;

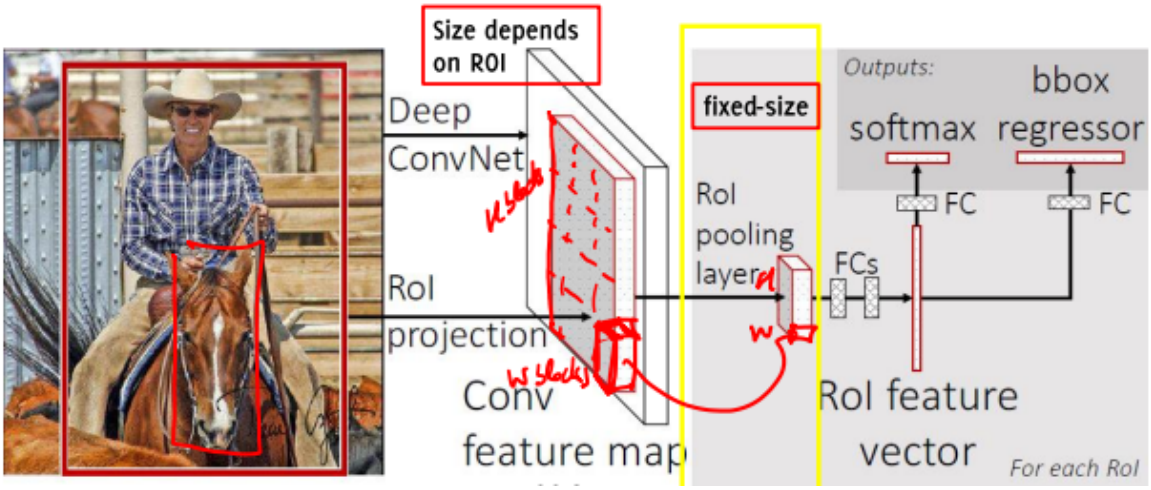
- Train post-hoc bounding box regressions;

The main limitation of the R-CNN approach is that it is not an end-to-end trainable system. The region proposal and feature extraction stages are separate from the classification stage, and the parameters of each stage must be optimized independently. This makes the R-CNN approach more complex and slower to train compared to end-to-end trainable object detection methods.

- Region proposals are from different algorithms and that part has not been optimized for detection;
- Training is slow and takes space;
- No feature re-use since CNN has to be executed on every ROI;

Fast R-CNN

1. The whole image is fed to a CNN that extracts feature maps;
2. The region proposals are identified from the image and projected into the feature maps. Regions are directly cropped from the feature maps, instead from the image, to reuse convolutional computation.
3. Fixed size is still required to feed data to a fully connected layer: ROI pooling layers extract a feature vector of fixed size H x W from each region proposal. Each ROI in the feature maps is divided in a H x W grid and then maxpooling over this provides the feature vector.
4. The FC layers estimate both classes and BB location (BB regressor). A convex combination of the 2 is used as a multitask loss to be optimized, as in R-CNN, but with no SVM.
5. Training performed in end-to-end manner, only 1 execution of convolution. In this new architecture, it’s possible to backpropagate through the whole network, thus train the whole structure and becoming faster. Now most of the time is on ROI extraction.



Faster R-CNN

It uses a convolutional neural network (CNN) to produce a feature map from the input image, and then generates region proposals using a region proposal network (RPN). The RPN is trained to predict the likelihood of an object being present in each region, as well as the bounding box coordinates for that object. These region proposals are then fed through the CNN to extract features, and these features are used to classify the objects and refine the bounding box coordinates using bounding box regression.

The region proposal network (RPN) is implemented as a fully convolutional layer that takes an input image and generates a set of region proposals, each with a class label (object or background) and a bounding box location. The RPN operates on a set of sliding windows at different scales and aspect ratios and uses a set of anchor boxes as reference to predict the class and location for each window.

The RPN uses a set of convolutional filters to compute features for each anchor box and then applies two sibling fully connected (FC) layers to predict the objectness score and the bounding box coordinates. The objectness score indicates the likelihood of the anchor box containing an object, while the bounding box coordinates are used to refine the location of the anchor box to better fit the object. The RPN also includes non-maximum suppression (NMS) to remove overlapping proposals and keep only the top-scoring ones.

The output of the RPN is a set of region proposals, which are then fed into the Fast R-CNN network to classify the proposals and refine the bounding box coordinates using the same convolutional features used by the RPN. The Fast R-CNN network includes a ROI pooling layer that resizes the region proposals to a fixed size, allowing the use of fully connected layers for classification and bounding box regression. The output of the Fast R-CNN network is a set of class-specific bounding boxes and associated class probabilities for each object in the image.

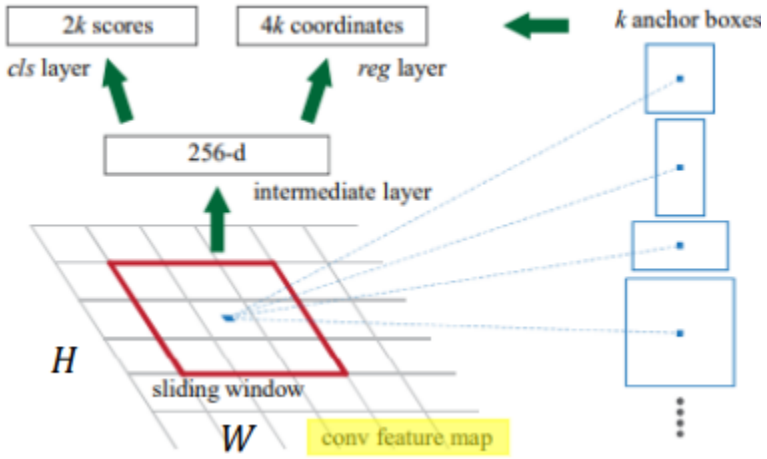
In a Faster R-CNN network, the backbone is a convolutional neural network (CNN) that is responsible for extracting features from the input image. This is typically a pre-trained CNN such as VGG or ResNet, which has already been trained on a large dataset and can be used as a starting point for the Faster R-CNN network. The backbone CNN is used to generate feature maps from the input image, which are then passed to the region proposal network (RPN) to generate proposals for object bounding boxes. The feature maps are also used by the detection network to classify and refine the object bounding boxes. The use of a pre-trained backbone helps to reduce the amount of training data and computation needed for the Faster R-CNN network, and can also improve the performance of the network.

The key difference between R-CNN and Faster R-CNN is that in the latter, the region proposal and feature extraction steps are done in a single pass through the CNN, rather than as separate steps as in R-CNN. This makes Faster R-CNN more efficient and faster to train and deploy.

Instead of the ROI extraction algorithm, train a region proposal network (RPN), which is a F-CNN (3x3 filter size). RPN operates on the same feature maps used for classification, thus at the least convolutional layer. RPN can be seen as an additional module that improves efficiency and focuses Fast R-CNN over the most promising regions for object detection. This is because the feature maps can keep some spatial and presence information.

Goal: associate to each spatial location k anchor boxes, i.e. ROI having different scales and ratios. The network outputs $H \times W \times k$ candidates anchors (H x W is the sliding

window size, it's used to make the initial prediction and has fixed dimensions) and estimate objectiveness scores for each one.



Intermediate layer: standard CNN layer that takes as input the last layer of the feature-ectraction network and uses 256 filters of size 3 x 3. It reduces the dimensionality of the feature maps and maps each region to a lower dimensional vector of size H x W x 256 (necessary for processing by FC layers).

The *cls* network is trained to predict the object probability (i.e the probability for an anchor to contain an object $1/0 \rightarrow 2k$ probability estimates . It's made of a stack of conv layers of 1 x 1 size (equivalent to a FC). Each of these k probability pair correspond to a specific anchor having a certain dimension and express the probability for that anchor in that spatial location to contain any object.

The regression network is trained to adjust each of the k predicted anchors to better match gorund truth $\rightarrow 4k$ estimates for the 4k bounding box coordinates. Each of these 4k tuples expresses the refinements for a specific anchor. Using different anchors does not requre a redesign, just defining different labels for each anchor. This sections makes the ROI have different sizes.

RPN returns H x W x k region proposals, thus replaces the RP algorithms. After the RPN, there is a non-maximum suppression based on the objectiveness score. Remaining proposals are then fed to the ROI pooling and classified by the standard Fast-CNN architecture. So, the ones remaining are non-overlapping BB that are used for final classification and bb definition through a ROI.

The sequence for object detection in the RPN of a Faster R-CNN network is as follows:

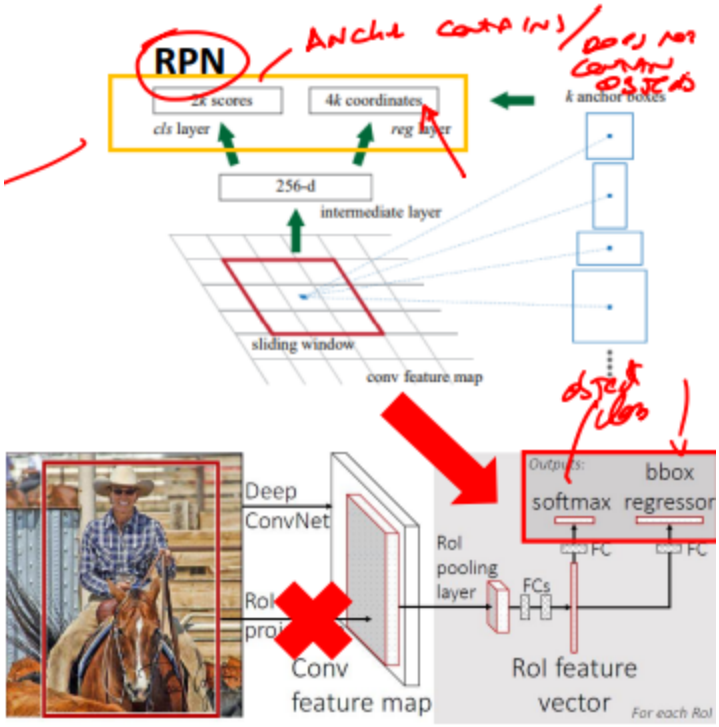
1. The input image is processed by the convolutional layers of the network, which extract features from the image.
2. The output of the convolutional layers is used to generate a set of anchor boxes, which are bounding boxes of different shapes and sizes that are placed over the image at different locations.
3. Each anchor box is passed through a convolutional layer to standardize its size, so that it can be fed to the fully-connected (FC) layers for classification.
4. The FC layers are used to estimate the probability of an object being present in each anchor box, and to refine the bounding box to better fit the object.
5. The best anchor box is selected based on the highest probability of an object being present and the most accurate bounding box refinement.
6. The region of the image within the selected bounding box is extracted and passed through the ROI (Region of Interest) pooling layer, which resizes the region to a fixed size so that it can be fed to the final FC layers for object classification.
7. The final FC layers classify the object within the region of interest and output the class label and a bounding box for the object.

N.B: After the convolutional layer, the bounding boxes (BBs) are standardized to a fixed size and fed to the fully connected (FC) layers for object classification and BB refinement. However, after the refinement step, the BBs may have different shapes and sizes, and so they need to be processed by the region of interest (ROI) pooling layer in order to be of the same size and be passed to the final FC layers for object detection. The ROI pooling layer resizes and aligns the BBs so that they can be fed to the final FC layers.

In Faster R-CNN, the anchor sizes follow the following evolution:

1. Before being passed through the Convolutional layer: the anchor sizes can vary, depending on the specific configuration of the network.
2. After being passed through the Convolutional layer: the anchor sizes are standardized, so that they all have the same dimensions.
3. After being passed through the region proposal network (RPN): the anchor sizes are modified according to the output of the RPN, which predicts the refinement of the bounding boxes.

4. After being passed through the Region of Interest (ROI) pooling layer: the anchor sizes are again standardized, so that they all have the same dimensions.
5. After being passed through the final region regression layer: the anchor sizes are modified again according to the output of the region regression layer, which further refines the bounding boxes.



Training now involves 4 losses:

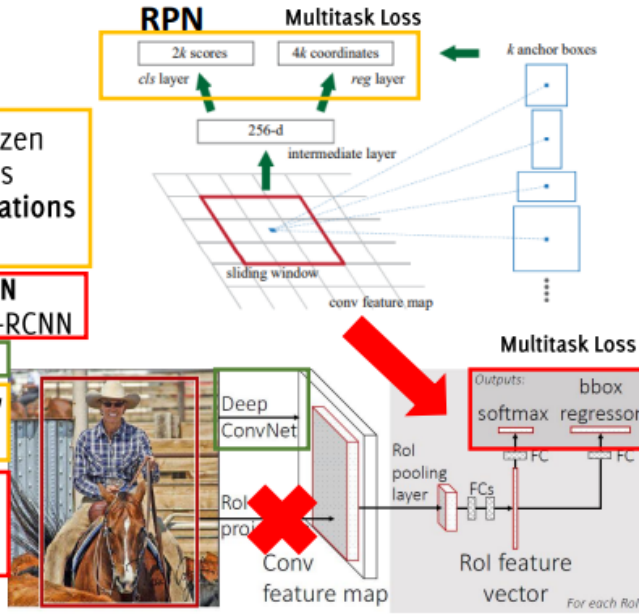
- RPN classification object presence;
- RPN regression coordinates;
- Final classification score;
- Final BB coordinates;

During training, the probability is defineed by measuring the overlap with annotated BB; the loss include a term of the final BB coordinates, as these are defined over the image.

Faster R-CNN Training

Training procedure

1. **Train RPN** keeping backbone network frozen and training only RPN layers. This ignores object classes but **just bounding box locations** (Multi-task loss cls + reg)
2. **Train Fast-RCNN** using proposals from **RPN trained before.** Fine tune the whole Fast-RCNN **including the backbone**
3. **Fine tune the RPN in cascade of the new backbone**
4. **Freeze backbone and RPN and fine tune only the last layers of the Faster R-CNN**



Faster R-CNN

At test time,

- Take the top ~ 300 anchors according to their object scores
- Consider the refined bounding box location of these 300 anchors
- These are the ROI to be fed to a Fast R-CNN
- Classify each ROI and provide the non-background ones as output

Faster R-CNN provides as output to each image a **set of BB** with their **classifier posterior**

The network becomes much faster (0.2s test time per image)

Faster R-CNN

It's still a **two stage detector**

First stage:

- run a **backbone network** (e.g. VGG16) to **extract featues**
- run the **Region Proposal Network** to **estimate** ~ 300 **ROI**

Second stage (the same as in Fast R-CNN):

- **Crop Features** through ROI pooling (with alignment)
- **Predict object class** using FC + softmax
- **Predict bounding box offset** to improve localization using FC + softmax

YOLO

The two-step approach makes region-based methods difficult to optimize. In YOLO, the object detection is framed as a single regression proble, from the image pixels to the BB coordinates and class probability, solving this regression all at once with a large CNN.

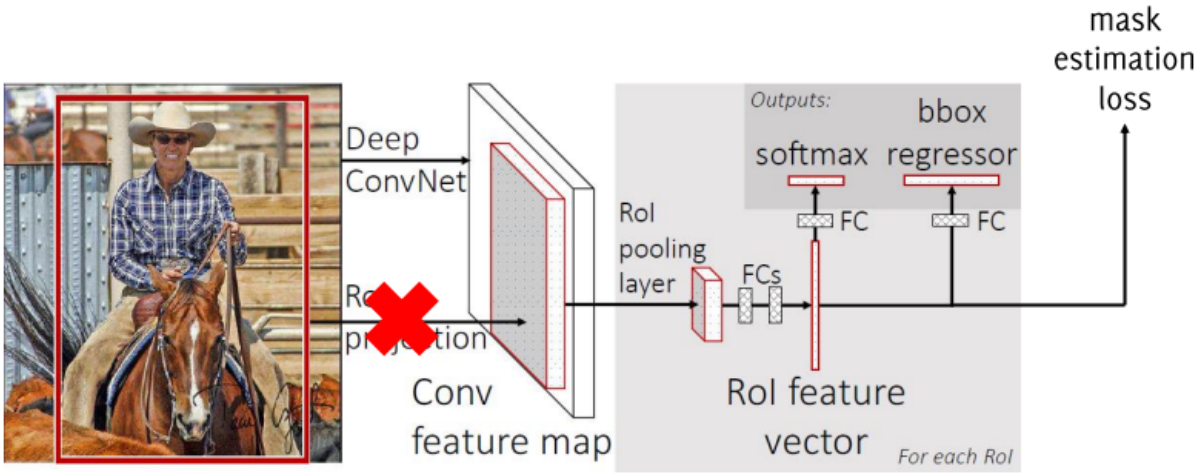
1. Divide each image in a grid (dimension H x L);
2. Each cell contains B anchors (BB) associated;
3. For each cell, we predict the offset of the base BB $(dx, dy, dh, dw, objectness\ score)$ and the classification score of the BB over the C considered categories + background.

$$output\ dimension: H \times L \times B \times (5 + C)$$

the whole prediction is performed in a single forward pass over the image, by a single CNN, which has difficult training and less accuracy.

Instance Segmentation

Combines object detection and semantic segmentation to separate each object instance. The ROI is extracted, classified and BB estimated, and inside each one, semantic segmentation is performed. This extension adds a branch for predicting an object mask in parallel with the existing BB recognition task.



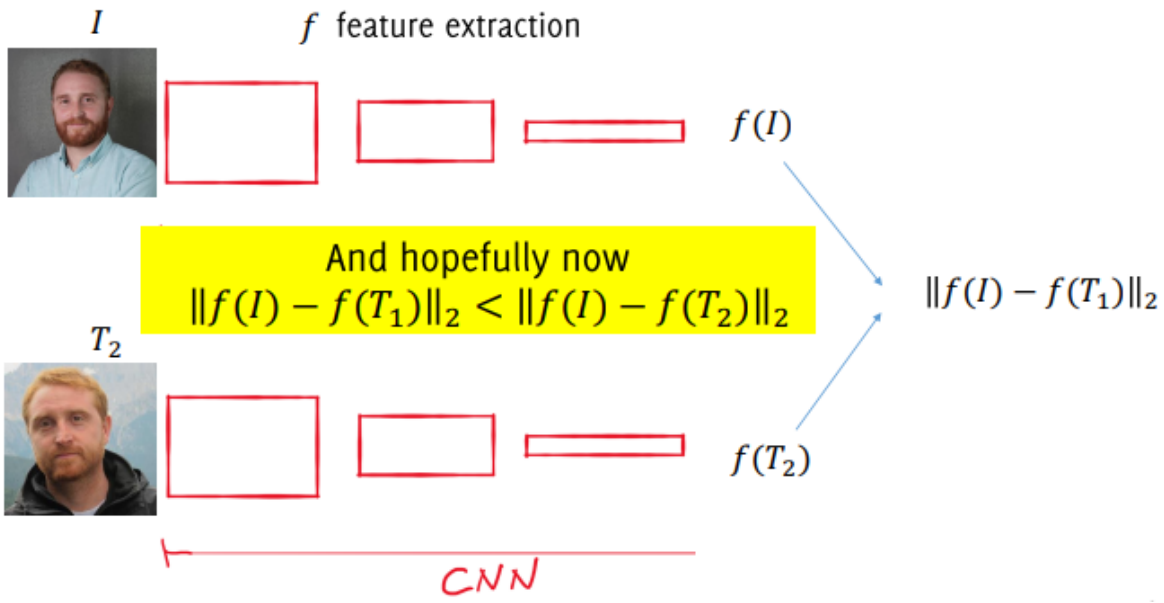
Object Detection with Distances

Siamese Networks

We need a trainig set of some images per class, images in different conditions, py snippets and a GPU. However adding a new class requires the whole net to be trained again. Image comparison by distance doesn't give good results on manipulated images $i = argmin_{j=1,\dots,N} \| I - T_j \|_2$.

The feature extractor is more general purpose than classification: $f(I)$: latent representation of Iprovides a meaningful description of the image in terms of patterns that are useful for solving the classification problem. We can calculate the distance on latent representations and compare:

1. Extract features from the first image $f(I)$;
2. Perform identification as $i = argmin_{j=1,\dots,N} \| f(I) - f(T_j) \|_2$



To optimise a network for this task, it needs to be trained over image distance, to obtain a desired comparison of 2 distances:

We would like to **train the weights** W of our CNN such that

$$\|f_W(I) - f_W(T_i)\|_2 < \|f_W(I) - f_W(T_j)\|_2 \quad \forall j \neq i$$

When I belongs to class i

The 2 nets have to perform the same operations, using the same weights. Hence the term siamese. During training, the net is feed with pairs of images (I_i, I_j) that might or not be referred to the same individual. Each image is annotated.

The loss can be computed as:

1. **Contrastive Loss:**

$$W = argmin_w \sum_{i,j} L_\omega(I_i, I_j, y_{i,j})$$

Where:

$$\mathcal{L}_\omega(I_i, I_j, y_{i,j}) = \frac{(1 - y_{i,j})}{2} \|f_\omega(I_i) - f_\omega(I_j)\|_2 + \frac{y_{i,j}}{2} \max\left(0, m - \|f_\omega(I_i) - f_\omega(I_j)\|_2\right)$$

- $y_{i,j} \in \{0,1\}$ is the label associate with the input pair (I_i, I_j):
  - 0 when (I_i, I_j) refers to the same person
  - 1 otherwise
- m is a hyperparameter indicating the margin we want (like in Hinge Loss)
- $\|f_\omega(I_i) - f_\omega(I_j)\|_2$ is the distance in the latent space.

2. **Triplet Loss**:

This loss function is compared on a Positive input referring to the same class and on a negative input, to another class, and we have to maximize the distance between these 2 comparisons.

$$\mathcal{L}_\omega(I, P, N) = \max(0, m + (\|f_\omega(I) - f_\omega(N)\|_2 - \|f_\omega(I) - f_\omega(P)\|_2))$$

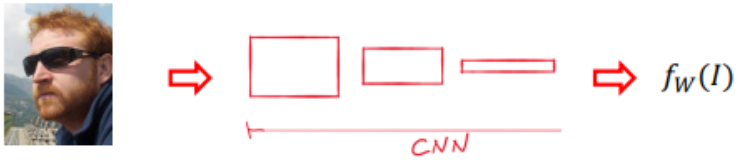
Triplet loss forces that a pair of samples from the same individual are smaller in distance than those with different ones.

m always play the role of the margin.

The selection of triplets for training is a matter of study

When a new image has to be verified

1. We feed the image I to the trained network, thus compute $f_W(I)$



2. Identify the person having average minimum distance from templates (in case there are many associated to the same individual)

$$\hat{\imath} = \underset{u}{\operatorname{argmin}} \frac{\sum_{T_{u,j}} \left\| f_W(I) - f_W(T_{u,j}) \right\|_2}{\#\{T_u\}}$$

3. Assess whether

$$\frac{\sum_{T_{i,j}} \left\| f_W(I) - f_W(T_{i,j}) \right\|_2}{\#\{T_{i,j}\}} < \gamma$$

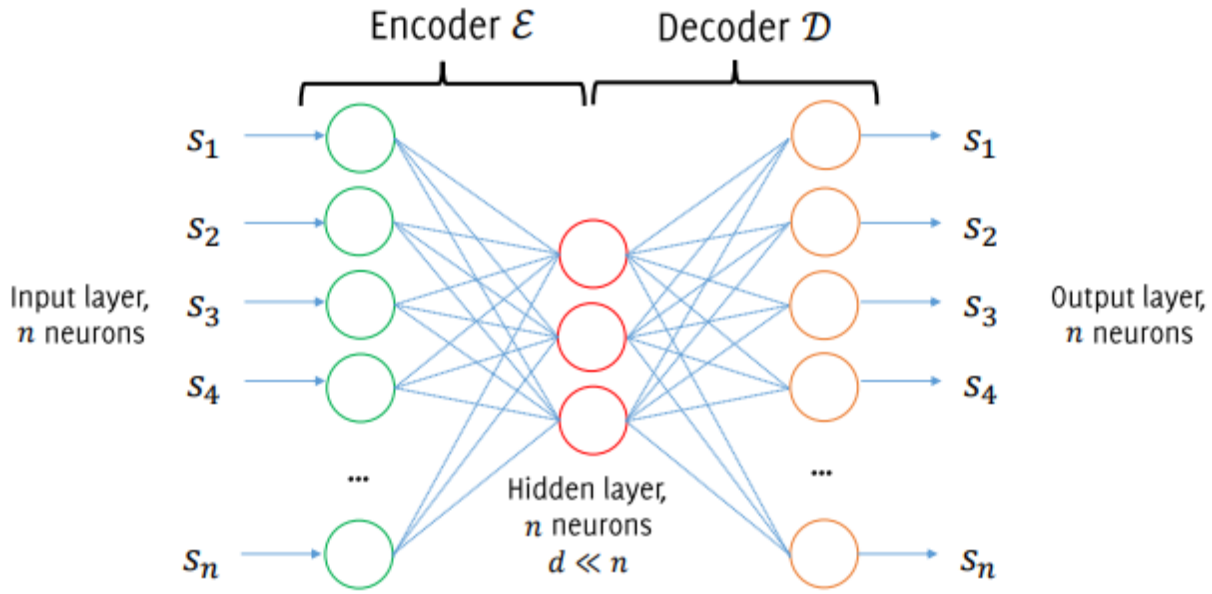
is sufficiently small, otherwise **no identification**

Other decision rules can be adopted (e.g. searching for the person giving the a template with a minimum distance)

Autoencoders and GAN

Autoencoders

Networks which are used for data reconstruction (unsupervised learning). The typical structure of an autoencoder is:



They can be trained to reconstruct all the data in a training set, so their objective is usually to learn an identity function. The reconstruction loss over batch S is:

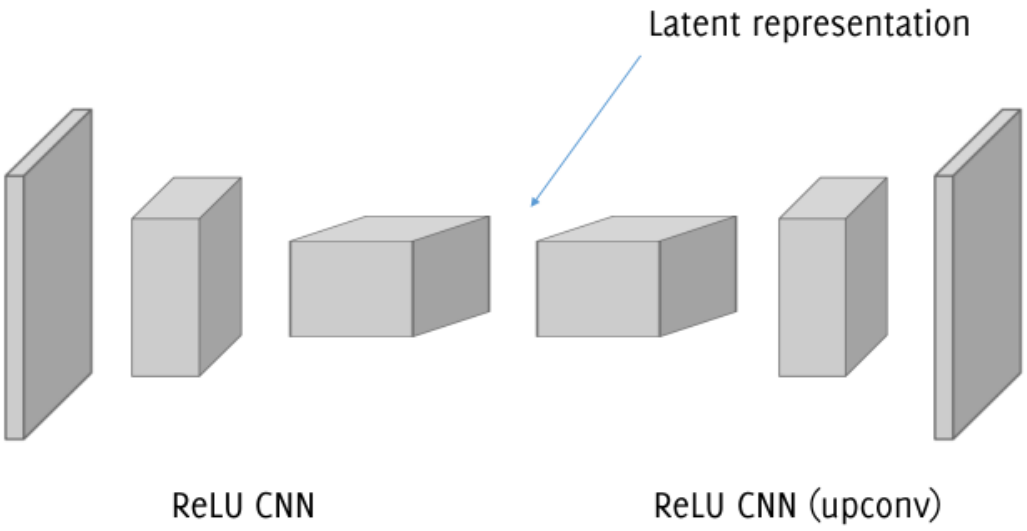
$$l(S) = \sum_{s \in S} \|s - D(E(s))\|_2$$

And training of $D(E())$ is performed through a standard backpropagation algorithm. The autoencoder learns the mapping. There are no external labels involved in training the autoencoder, as it performs input reconstruction.

Featrues $z = E(s)$ are referred to as latent representation. Autoencoders typically do not provide exact reconstruction since n << d, so we expect the latent representation to be a meaningful and compact description of the input. It is possible to add a regularization term $+\lambda R(s)$ to steer latent representation $E(s)$ to satisfy desired

properties or the reconstruction $D(E(s))$. More powerful and nonlinear representations can be learned by stacking multiple hidden layers.

It is possible to use convolutional layers and transpose convolution to implement a deep net.

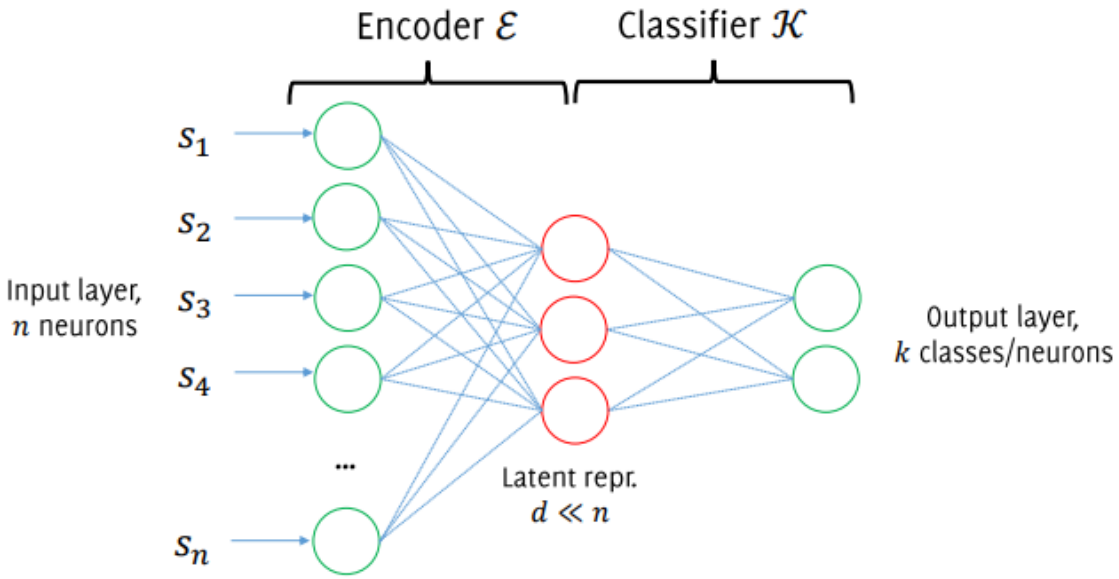


Weight Initialization

Autoencoders can be used to initialize the classifier when the training set includes:

- Few annotated data
- Many unlabeled ones

1. Train the AE in a fully unsupervised way, using unlabeled data S;
2. Get rid of decoder and keep the encoder weights;
3. Plug in a FC layer for classifying samples from latent representation;
4. Fine-tune the autoencoder using the few supervised samples provided L. If L is large enough, the encoder weights can also be fine-tuned.



Autoencoders provide a good initialization because the latent vector is a good latent representation of the inputs used for training. They can learn a compact representation of the data, which can be used as a good starting point for training a model to perform a specific task. This is especially useful when the data is high-dimensional or has a complex structure, as the autoencoder can learn to capture the most important features of the data and reduce the dimensionality. Using the weights learned by the autoencoder as initialization for a model can often lead to faster convergence and better performance, especially if the task is related to the data the autoencoder was trained on.

Noise Reduction

These models can learn to remove noise from data: by adding some noise to the training set, the net will try to learn and reconstruct the original input. This can also produce a more robust latent representation, because imprecisions will be ignored.

Generative Models

Goal: given a training set of images $TR = \{x_i\}$, generate images that are similar to these. They can be used for augmentation, simulation, and planning. The training can enable inference of latent representations to get general features.

Autoencoders

1. Train an autoencoder on S;
2. Discard the encoder;
3. Draw random vectors $z \sim \phi_x$ to mimic a new latent representation and feed this to the decoder input.

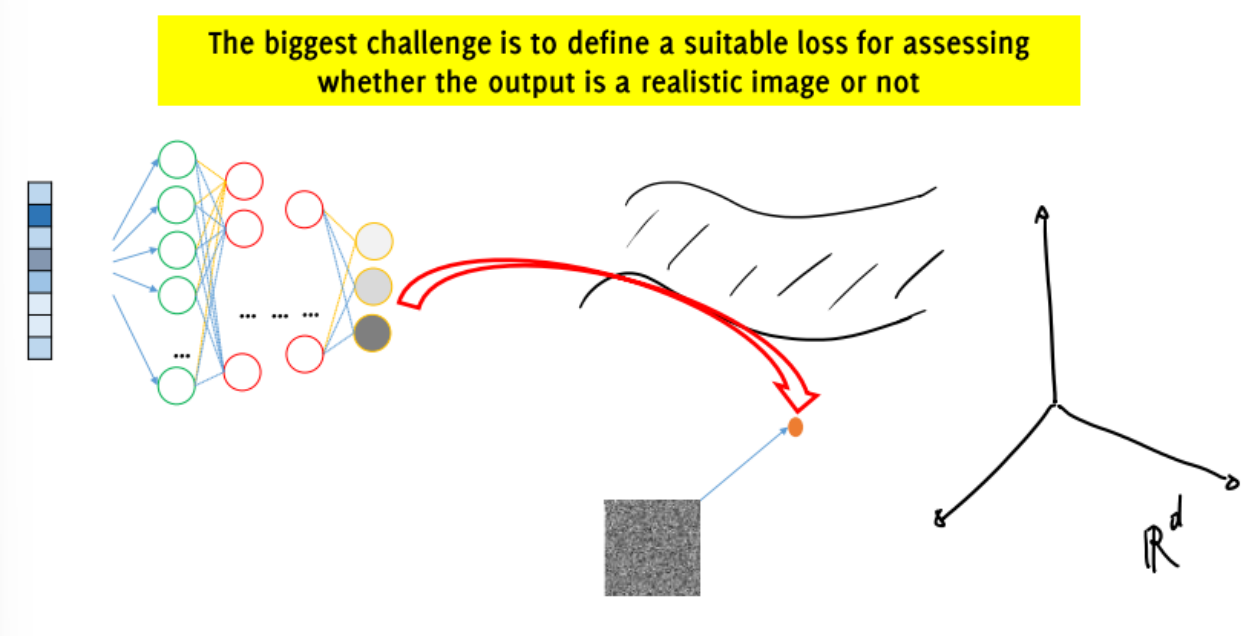
This approach is difficult since we do not know the distribution of proper latent representation to give appropriate input vectors. Variational autoencoders leverage a prior over Z to generate images using a similar architecture.

In the context of autoencoders, the latent space refers to the internal representation of the input data that is learned by the autoencoder during training. The autoencoder consists of an encoder and a decoder. The encoder maps the input data to the latent space, and the decoder maps the latent space back to the original input data space. The latent space is typically lower-dimensional than the input data space, and it is thought to capture the most important features of the input data.

GAN

Do not look for an explicit density model ϕ_s describing the manifold of natural images, just find a model able to generate samples that look like training ones $S \subset \mathbb{R}^n$. Instead of sampling the ϕ_s, sample a seed from a known distribution, ϕ_z, defined a priori as **noise**. Feed this seed to a learned transformation that generates realistic samples, as if they were drawn from ϕ_s. Use a NN to learn this transformation, in an unsupervised manner.

So you’re basically using a decoder over random noise to see if the output is in the desired latent space or not. Both the generator and discriminator are first trained on correct data, and then they are trained together in an adversarial manner to improve the performance of the GAN. The first training of the discriminator is used to teach it the characteristics of real samples, so that it can then use this knowledge to distinguish real samples from fake ones produced by the generator. The second training of the discriminator is needed because the generator is constantly trying to improve its ability to produce realistic samples, and the discriminator needs to continue learning and adapting in order to maintain its ability to distinguish real samples from fake ones. This process continues until the generator is able to produce samples that are indistinguishable from real ones, at which point the GAN has succeeded in learning to generate realistic samples.

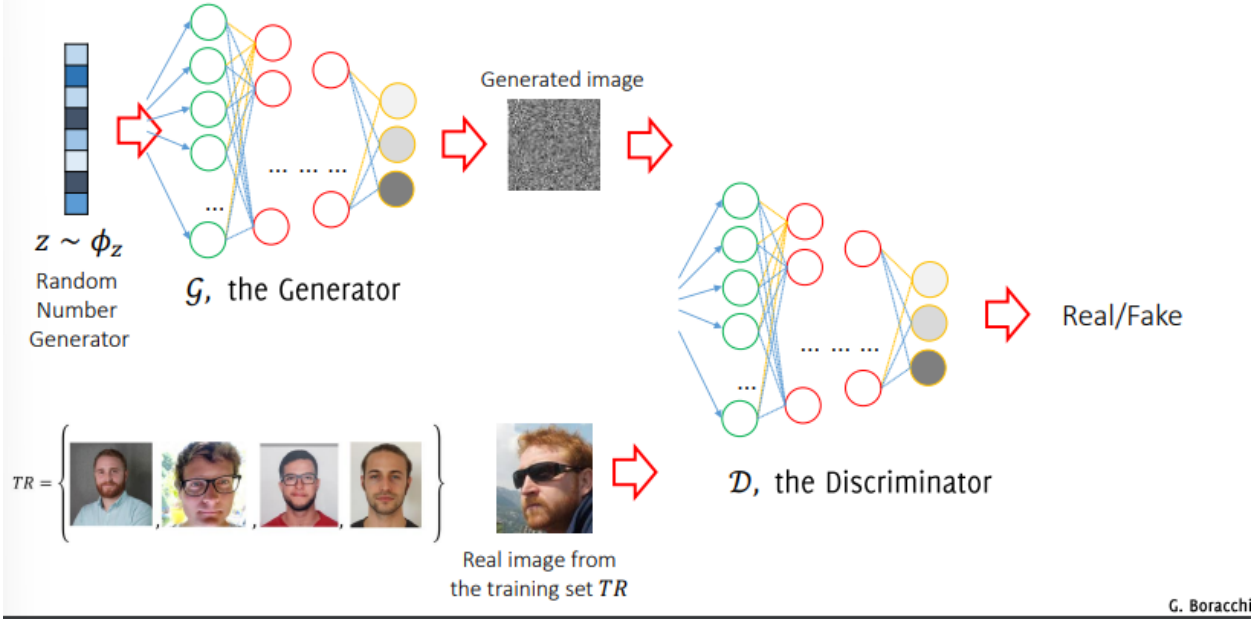


GAN - Adversary Nets

An adequate loss function for a GAN needs to measure how realistic an output is. This can be done with another NN. Train a pair of NN addressing 2 different tasks, competing in an adversarial manner:

- **Generator G** that produces realistic samples taking as input some noise. It has a first training done to understand and initialize the model in a way that's closed to the desired one. G is trained to generate images that can fool D, namely can be classified as real by D. The loss of G is therefore given by D. It can be seen as a decoder.
- **Discriminator D** that takes as input an image and assess whether it's real or generated by G. It can be seen as a binary classifier.

The two are trained together, and at the end, only G is kept. They are adversary networks. We hope that at the end of the training, G will be able to fool D consistently. We discard D and keep G as generator.



GAN Training

D and G are chosen as MLP or CNN, with the following inputs (in the final net, the input is only one however):

- $D = D(s, \theta_d)$
- $G = G(z, \theta_g)$

where θ_g, θ_d are network parameters , $s \in \mathbb{R}^n$ is an input image (either real or generated by G), and $z \in \mathbb{R}^d$ is some random noise to be fed to the generator. Our network will output:

- $D(\cdot, \theta_d) : \mathbb{R}^n \to [0, 1]$ gives as output the posterior for the input to be a true image;
  $G(\cdot, \theta_d) : \mathbb{R}^d \to \mathbb{R}^n$ gives as output the generated image;

A good discriminator is such:

- $D(s, \theta_d)$ is maximim when $s \in S$, true image from the training set;
- $1 - D(s, \theta_d)$ is maximum when s was generated from G;
- $1 - D(G(z, \theta_g), \theta_d)$ is maximum when $z \sim \phi_z$

Training D consists in maximizing the binary crossentropy

$$max_{\theta_d}(E_{s\sim\phi_s}[logD(s,\theta_d)] + E_{z\sim\phi_z}[log(1-D(G(z,\theta_g).\theta_d))])$$

A good generator G makes D to fail, minimizing the above

$$min_{\theta_g}max_{\theta_d}(E_{s\sim\phi_s}[logD(s,\theta_d)] + E_{z\sim\phi_z}[log(1-D(G(z,\theta_g).\theta_d))])$$

where:

- s is a sample from the real data distribution, ϕ_s is the distribution of real data, and $D(s,\theta_d)$is the output of the discriminator for the real sample s and its parameters θ_d;
- z is a sample from the noise distribution, ϕ_z is the noise distribution, and $G(z,\theta_g)$ is the output of the generator for the noise sample z and its parameters θ_g;

Computing the gradients of the loss function with respect to the weights of each network, and using these gradients to update the weights in the opposite direction for the generator and the same direction for the discriminator trains the net.

The process can be repeated until the loss function has converged to a minimum or maximum, depending on which network we are training. This process can be implemented using backpropagation and an optimizer such as SGD or Adam.

Solve by an iterative numerical approach

$$\min_{\theta_g} \max_{\theta_d} \left(\mathrm{E}_{s\sim\phi_S}[\log \mathcal{D}(\boldsymbol{s}, \theta_d)] + \mathrm{E}_{z\sim\phi_Z}[\log(1 - \mathcal{D}(\mathcal{G}(\boldsymbol{z}, \theta_g), \theta_d))] \right)$$

Alternate:

- k-steps of Stochastic Gradient Ascent w.r.t. θ_d , keep θ_g fixed and solve

$$\max_{\theta_d} \left(\mathrm{E}_{s\sim\phi_S}[\log \mathcal{D}(\boldsymbol{s}, \theta_d)] + \mathrm{E}_{z\sim\phi_Z} \left[\log \left(1 - \mathcal{D}(\mathcal{G}(\boldsymbol{z}, \theta_g), \theta_d) \right) \right] \right)$$

- 1-step of Stochastic Grandient Descent w.r.t. θ_g being θ_d fixed

$$\min_{\theta_g} \left(\mathrm{E}_{s\sim\phi_S}[\log \mathcal{D}(\boldsymbol{s}, \theta_d)] + \mathrm{E}_{z\sim\phi_Z} \left[\log \left(1 - \mathcal{D}(\mathcal{G}(\boldsymbol{z}, \theta_g), \theta_d) \right) \right] \right)$$

and since the first term does not depend on θ_g, this consists in minimizing

$$\min_{\theta_g} \left(\mathrm{E}_{z\sim\phi_Z} \left[\log \left(1 - \mathcal{D}(\mathcal{G}(\boldsymbol{z}, \theta_g), \theta_d) \right) \right] \right)$$

for $i = 1 \ldots$ #number of epochs

for k −times # gradient ascent steps for θ_d

- Draw a minibatch $\{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_m\}$ of noise realization
- Sample a minibatch of images $\{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m\}$
- Update θ_d by stochastic gradient ascend:

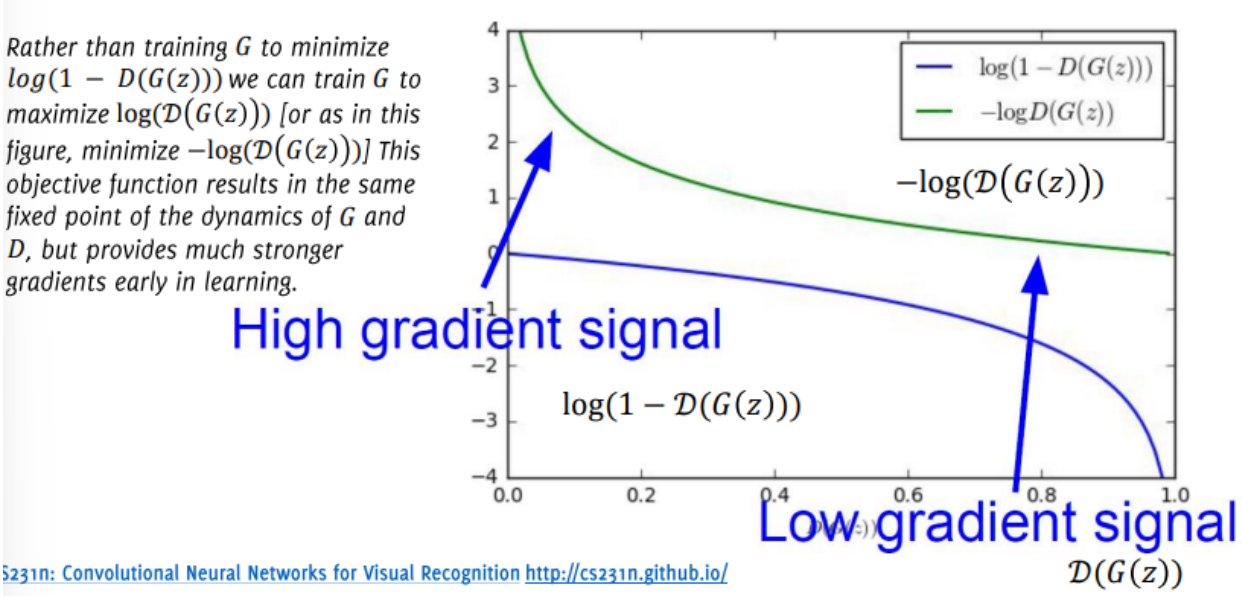
$$\nabla_{\theta_d} \left[\sum_i \log \mathcal{D}(\boldsymbol{s_i}, \theta_d) + \log \left(1 - \mathcal{D}(\mathcal{G}(\boldsymbol{z}_i, \theta_g), \theta_d) \right) \right]$$

Draw a minibatch $\{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_m\}$ of noise realizations # gradient descent steps for θ_g

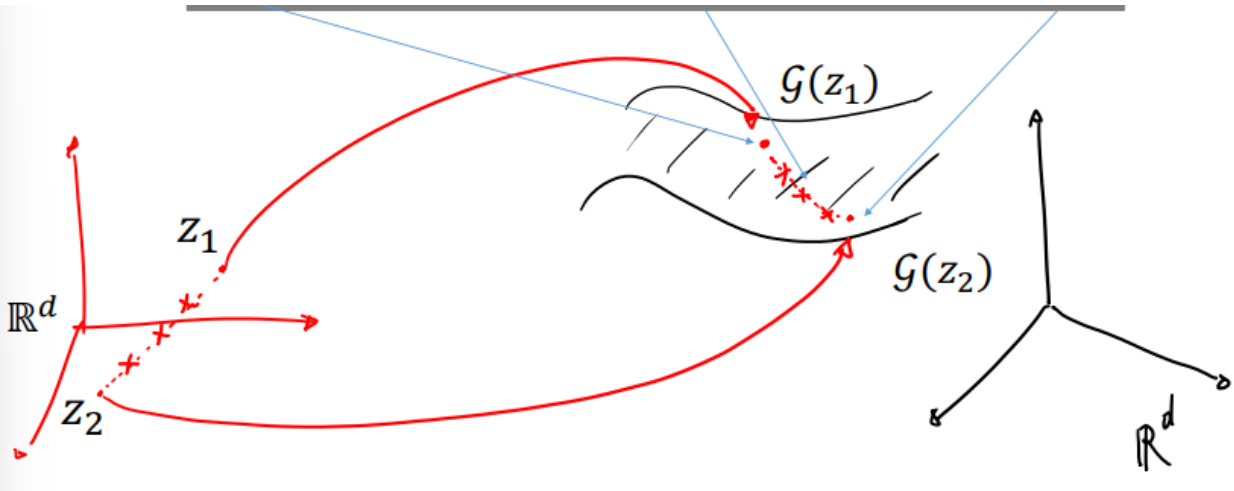
Update $\mathcal{G}$ by stochastic gradient descent:

$$\nabla_{\theta_g} \left[\sum_i \log \left(1 - \mathcal{D}(\mathcal{G}(\boldsymbol{z}_i, \theta_g), \theta_d) \right) \right]$$

However, this is the textbook best practice of training, in real life the trick consists in maximizing the second part without the 1-. That's because at the beginning the D is good at rejecting samples that are very different, and the gradient of the G is relatively flat, while it's steep when the sample is already good. This is not good for G learning. So, instead of minimizing the function, we maximize its manipulation, which provides a steeper gradient in its early stages.



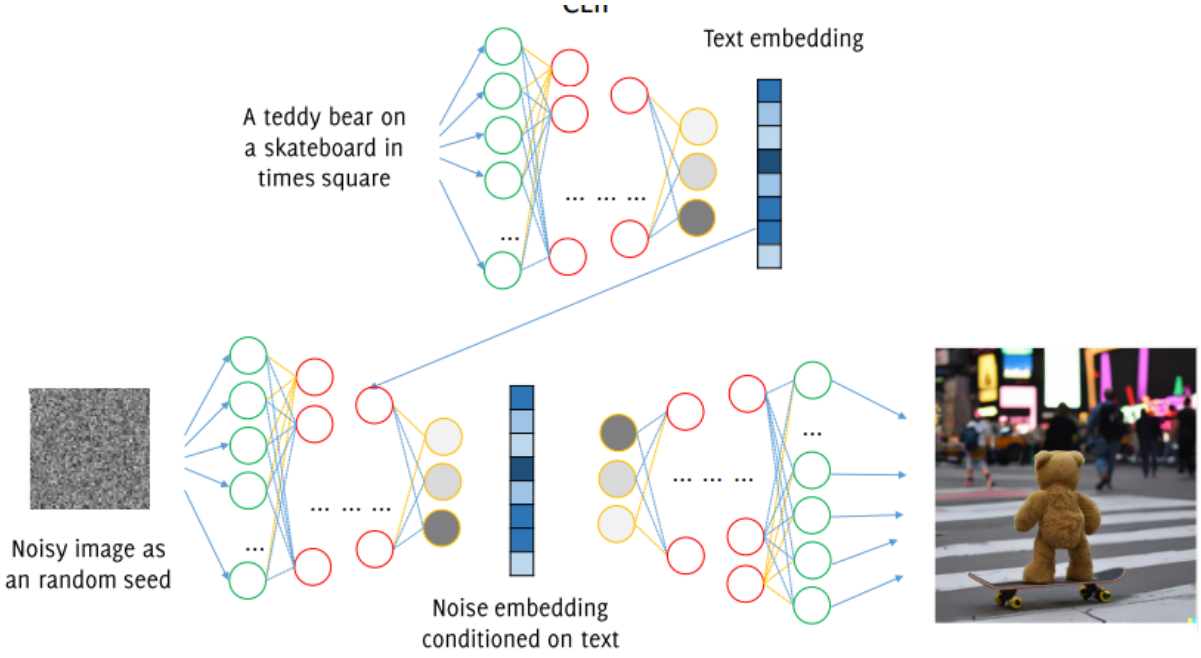
At the end, the discriminator is discarded and the generator and ϕ_z are preserved as generative model. The training is rather unstable and the steps have to be synchronized. The generator never uses S during training, and its performance is difficult to assess quantitatively. There is no explicit expression for the generator, so a likelihood for a single sample cannot be computed. This means that the generator in a GAN is a function that takes in a noise sample and produces a synthetic output, such as an image. It does not have a fixed distribution like the data distribution of real samples. Therefore, it is not possible to compute the likelihood of a single sample from the generator. Instead, the generator is trained by minimizing the loss function, which is a measure of the difference between the generated samples and the real samples. In each iteration of training, the generator produces a batch of synthetic samples, and the discriminator compares them to a batch of real samples. The loss is then computed based on the difference between the two batches of samples, and the gradients are computed to update the weights of the generator and the discriminator.



The generator is a neural network that takes in noise as input and tries to map it to the latent space of the desired output. The generator is trained by trying to minimize the binary cross-entropy loss function, which is a measure of the difference between the generator's output and the desired output in the latent space. The generator is not given explicit examples of the desired output, so it needs to learn how to move through the latent space in order to produce output that is similar to the desired output. This process is often done iteratively, with the generator and discriminator being trained alternately until the generator produces output that is similar enough to the desired output.

DALL-E 2

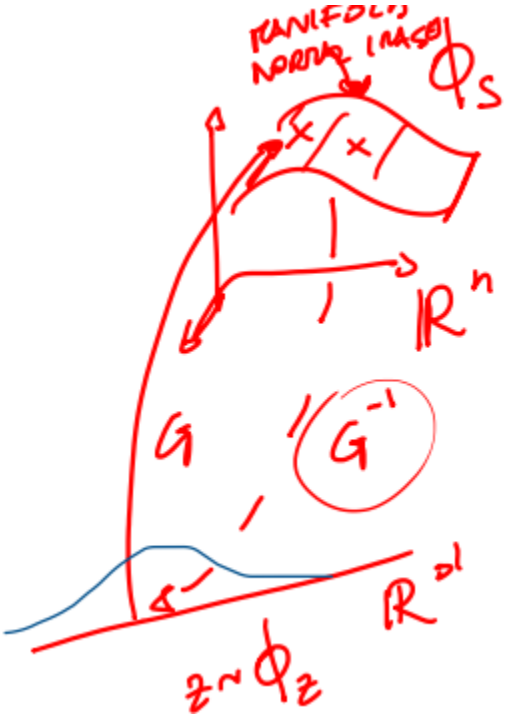
DALL-E 2 is a neural network that is trained to generate images from text descriptions, using a combination of a transformer-based text encoding model and a transformer-based image generation model. The text encoding model takes a text description as input and produces a fixed-length text embedding vector, which is then passed to the image generation model. The image generation model takes the text embedding vector and produces an image as output. During training, the network is presented with pairs of text descriptions and corresponding images, and the network learns to generate images that match the given text descriptions. The weights of the network are updated based on the error between the generated images and the ground truth images, using a combination of reconstruction loss and adversarial loss. The network is able to generate a wide variety of images, including ones that have never been seen before, by combining the information from the text embedding with the learned patterns in the image generation model.



GAN for Anomaly Detections

GAN can map random variables to the manifold of images. An anomlay detector can be implemented if we train a G to generate normal images and then invert the mapping and get G^{-1}.

In the context of using an inverse GAN for anomaly detection, the goal is to use the trained generator to transform a given input image from the desired manifold into the latent space of noise samples. If the resulting noise sample is far from the distribution of noise samples that the generator was trained on, then it is likely that the input image is an anomaly. This is because the generator was trained to transform noise samples into images that belong to the desired manifold, so if it is unable to transform a given input image back into a noise sample that is similar to the ones it was trained on, it is likely that the input image is not a typical member of the desired manifold.

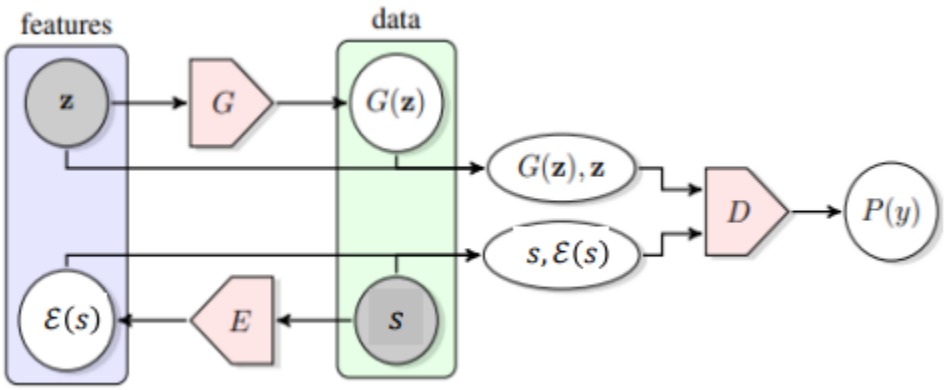


Bi-GANs

A BiGAN adds an encoder E to the adversarial net which:

- Brings an image back to the space of noise vectors;
- Can be used to reconstruct an input image s as in autoencoders $G(E(s))$;
- The discriminator D takes as input as in conditional GAN;

A Bi-GAN can be used to learn the normal data distribution, and then any data point that is not well-represented by the learned distribution can be considered an anomaly. For example, if the Bi-GAN is trained on a dataset of normal images, it can learn a mapping between the latent space of noise samples and the image space. If an image is then presented to the Bi-GAN that is not well-represented by the learned distribution, it may be considered an anomaly.

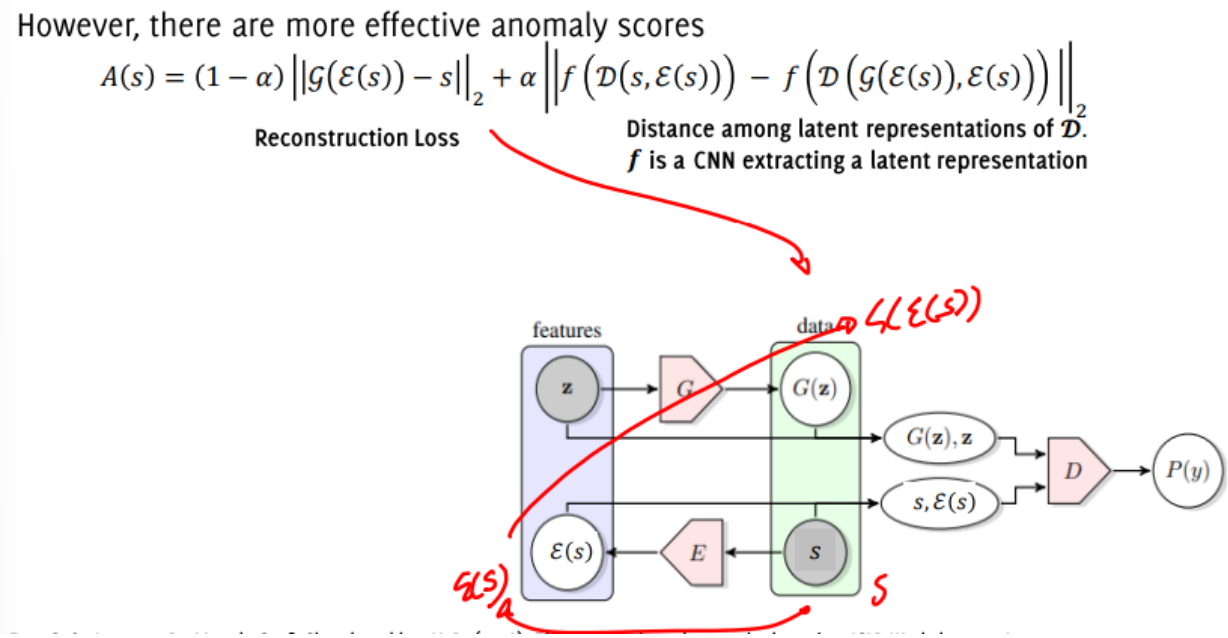


$$\min_{G,E} \max_{\mathcal{D}} V(\mathcal{D}, E, G)$$

$$V(\mathcal{D}, E, G) = \mathrm{E}_{s\sim\phi_S}[\log \mathcal{D}(s, E(s))] + \mathrm{E}_{z\sim\phi_z}[1 - \log \mathcal{D}(G(z), z))]$$

The encoder $E(\cdot)$ can be used for detection by computing the likelihood of $\phi_z(E(s))$ and consider as anomalous all the images s corresponding to a low likelihood. This is based on the assumption that the bi-GAN has been trained to model the distribution of normal images, and therefore images that are significantly different from the normal images should have a low likelihood under the model. By comparing the likelihood of an input image to a threshold, it is possible to determine whether the image is anomalous or not.

Another option is to use the posterior of the discriminator as anomaly score $D(s, E(s))$ since the discriminator will consider the anomalous sample as fake.



Bi-GANs are limited image-wise and have little training stability, also reconstructed images have low quality. We can use fully convolutional nets for better performance.

Recurrent Neural Networks

So far, we have only considered static datasets, which are those that do not change over time. Dynamic datasets can be handled using memoryless models and models with memory. Memoryless models are models that do not retain any information about previous input or output. They only consider the current input and produce an output based on that. On the other hand, models with memory retain some information about past input and use it to influence the current output.

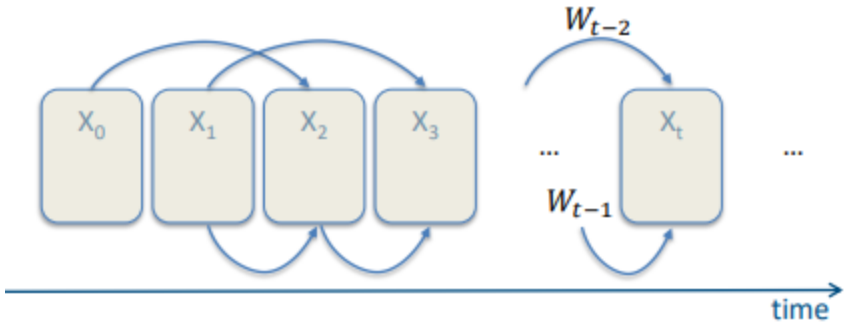
Memoryless Models for Sequences

Autoregressive Models

They predict the next input from the previous ones using **delay traps**.

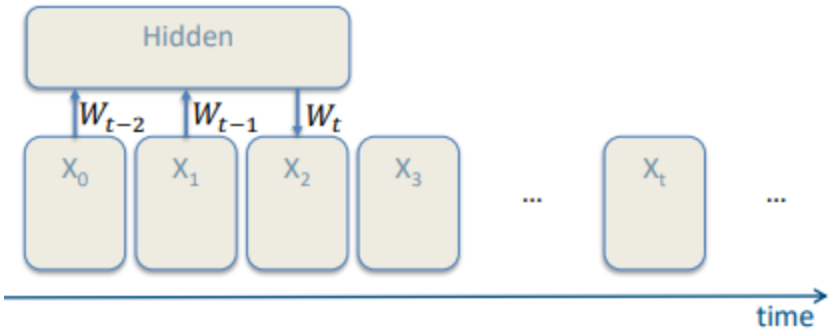
To build an autoregressive model, you first need to specify the number of previous time steps to include as predictors (also known as the lag order). For example, if you want to use the past three values to predict the current value, the lag order would be 3. You can then estimate the model parameters by fitting the model to a training dataset using a statistical technique such as least squares.

Once the model is trained, you can use it to make predictions by inputting the previous time steps as predictors and using the model to estimate the current value.



Feed Forward NN

FFNNs can be used to model dynamic data by including nonlinear hidden layers in the network. These hidden layers can learn complex relationships between the input and output data, which can help the network to better predict future values based on past values. The use of nonlinear hidden layers allows the FFNN to capture more complex relationships in the data, which can improve the accuracy of the predictions made by the model.



Dynamic Systems

Generative models with a hidden state cannot be observed directly.

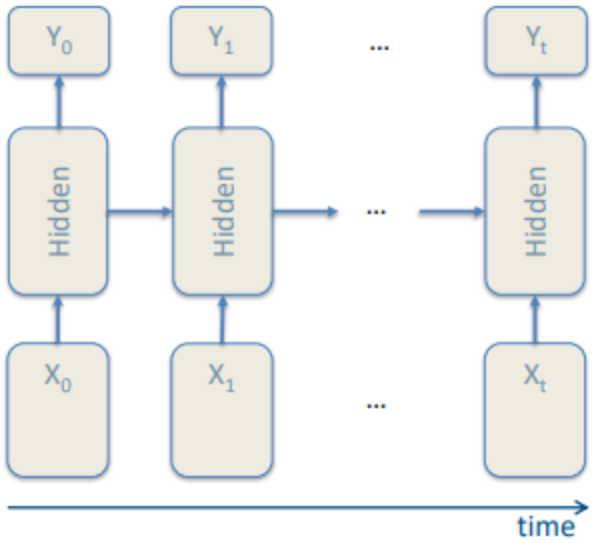
- The hidden state has some dynamics possiblyaffected by noise and produces the output;
- To compute the output need to infer hidden state;
- Input are treated as driving inputs;

Linear Dynamic Systems

Linear dynamic systems are mathematical models that describe the evolution of a system over time. They are characterized by a set of linear differential equations that relate the current state of the system to its past states.

In linear dynamical systems this becomes:
• State continuous with Gaussian uncertainty;

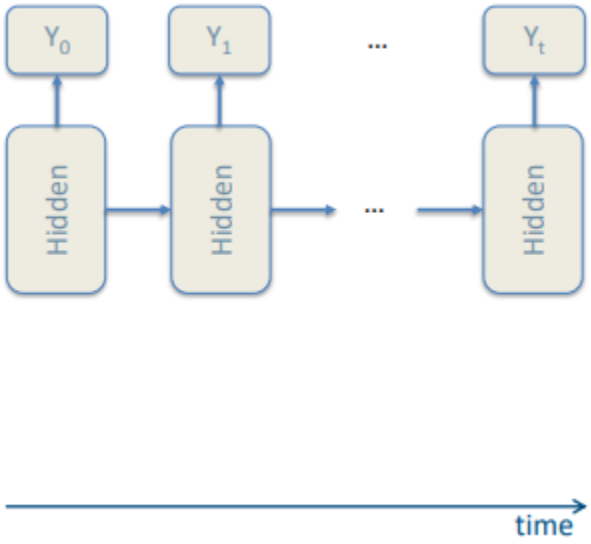
• Transformations are assumed to be linear;
• State can be estimated using Kalman filtering;



Hidden Markov Models

Hidden Markov models (HMMs) are probabilistic models that describe the evolution of a system over time. They are characterized by a set of hidden states that the system can occupy at any given time, and a set of observations that can be made about the system. The state transitions and observations are governed by probabilistic rules.

- State assumed to be discrete, state transitions are stochastic (transition matrix);
- Output is a stochastic function of hidden states;
- State can be estimated via Viterbi algorithm;



Recurrent NN

Recurrent neural networks (RNNs) are a type of neural network that are designed specifically to process sequential data, such as time series, natural language, and audio. They are called "recurrent" because they perform the same operation for each element of the input sequence, with the output of one element serving as the input for the next.

RNNs are composed of three main types of layers: input, hidden, and output. The input layer receives the input sequence, and the output layer produces the final output. The hidden layer is where the computation occurs, and it contains units called "recurrent units" that are responsible for maintaining a memory of past inputs.

The basic structure of an RNN is as follows: for each element in the input sequence, the input layer receives the element and passes it through the hidden layer, which processes it using the recurrent units. The output of the hidden layer is then passed through the output layer, which produces the final output. The output of the hidden layer is also passed back into the hidden layer as input for the next element in the sequence, allowing the RNN to maintain a memory of past inputs.

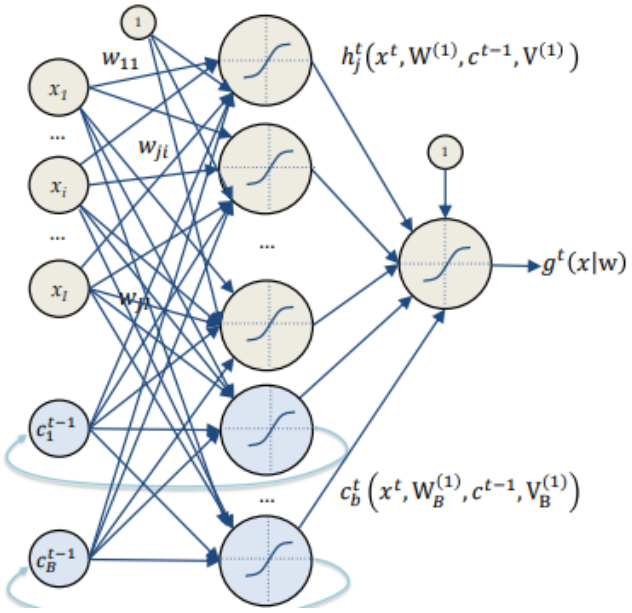
Memory via recurrent connections:

- Distributed hidden state allows to store information efficiently
- Non-linear dynamics allows complex hidden state updates

$$g^t(x_n|w) = g\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j^t(\cdot) + \sum_{b=0}^{B} v_{1b}^{(2)} \cdot c_b^t(\cdot)\right)$$

$$h_j^t(\cdot) = h_j\left(\sum_{j=0}^{J} w_{ji}^{(1)} \cdot x_{i,n}^t + \sum_{b=0}^{B} v_{jb}^{(1)} \cdot c_b^{t-1}\right)$$

$$c_b^t(\cdot) = c_b\left(\sum_{j=0}^{J} w_{bi}^{(1)} \cdot x_{i,n}^t + \sum_{b'=0}^{B} v_{bb'}^{(1)} \cdot c_{b'}^{t-1}\right)$$



Unrolling a recurrent neural network (RNN) means creating a feedforward neural network with a fixed number of layers, where each layer corresponds to a time step in the original RNN. The weights in the unrolled network are shared across all time steps, meaning that the same set of weights is used at each time step. Unrolling an RNN is typically done as a way to make it easier to train the network using techniques like backpropagation through time. By unrolling the network, the gradients can be computed using the standard backpropagation algorithm, rather than having to compute them for each time step separately. This can make training the network more efficient, especially for long sequences of data.

Each layer typically contains both standard neurons and neurons that are specifically designed to capture temporal relationships. The neurons that capture temporal relationships are referred to as "recurrent" neurons, and they are typically organized into "hidden states" that capture the current state of the input sequence at a given time step.

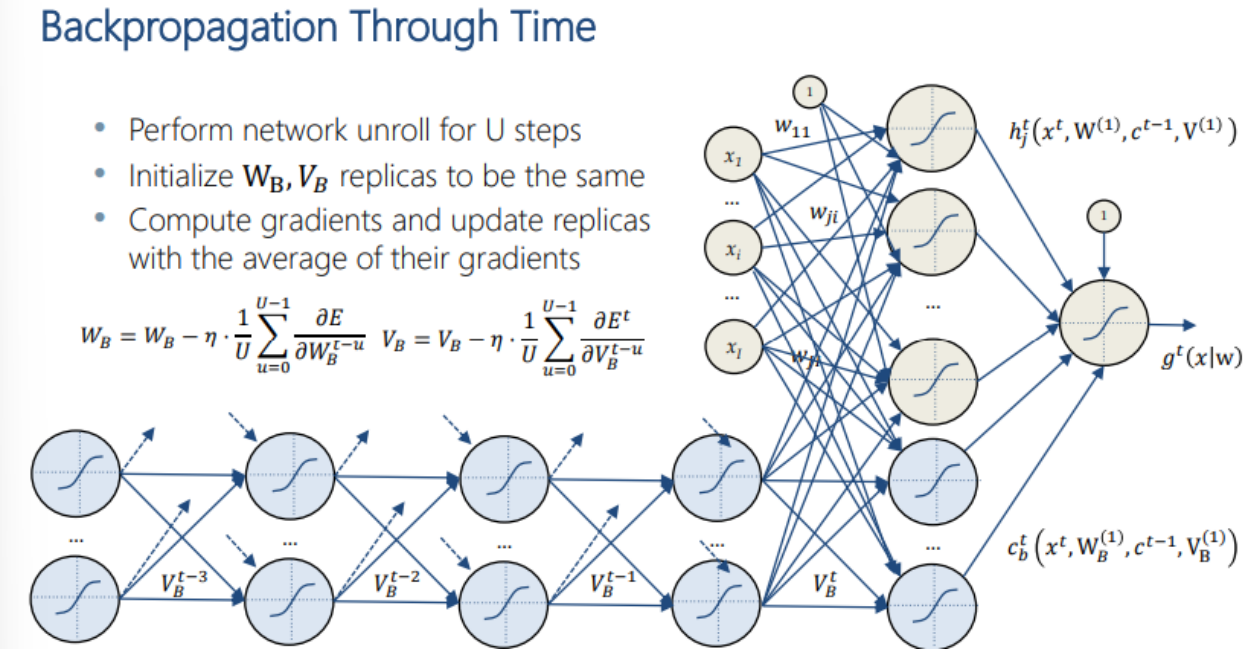
When training an RNN, the network is unrolled through time, meaning that it is trained on a sequence of inputs one time step at a time. The hidden state of the network at each time step is updated based on the input at that time step and the previous hidden state, allowing the network to capture temporal dependencies between the input elements.

Backpropagation Through Time

Backpropagation through time (BPTT) is an algorithm used to train an RNN by calculating the gradient of the loss function with respect to the network's parameters at each time step, and then updating the parameters using gradient descent. BPTT involves unrolling the network through time and treating it as a standard feedforward neural network, allowing the gradients to be calculated using standard backpropagation algorithms.

Steps:

1. Unfold the RNN for U timesteps, obtaining a FFNN. Given N a RNN that has to learn a temporal task starting from time *t-u* from time *t*, and N^* the FFNN resulting from the unfold of N.
    - For every timestep in the interval (*t-u, t*], the net N^* has a layer containing k neurons, where k is the number of neurons in N;
    - For every layer of N^*, there's a copy of each neuron in N;
    - For every timestep $\tau \in (t-u, t]$ the synaptic weight from neuron i in layer τ to neuron j in layer $\tau + 1$ of N^* is a copy of the connection between i and j in N.
2. Initialize replicas of W_B and V_B (which are the weights between the grey and blue neurons and blue neurons respectively) so that they're the same;
3. Backpropagate on the new network. All the weights are trained using gradient descent, considering a timestep τ. This means that we'll have diferent weights for dsifferent timesteps on the same connections, and to avoid this the weights are updated using a mean (the one written in the picture);



The Vanishing Gradient

BPTT (backpropagation through time) is an algorithm used to train recurrent neural networks (RNNs). It involves computing the gradient of the loss function with respect to the RNN's weights at each time step and summing these gradients to update the weights. The problem with BPTT is that, as the number of time steps increases, the gradients can become very small, leading to a phenomenon known as the vanishing gradient problem.

Here is an example of how the vanishing gradient problem can occur mathematically:

Suppose we have an RNN with weights $w_1, w_2, ..., w_n$ at each time step. The output at time step t is given by:

$$y_t = f(w_t \cdot x_t)$$

where $f(\cdot)$ is an activation function and x_t is the input at time step t. The loss at time step t is given by:

$$L_t = \frac{1}{2}(y_t - y_{target})^2$$

The gradient of the loss with respect to the weights at time step t is given by:

$$\frac{\partial L_t}{\partial w_t} = (y_t - y_{target}) \cdot \frac{\partial f(w_t \cdot x_t)}{\partial w_t}$$

To update the weights, we need to compute the gradients at all time steps and sum them:

$$w_t = w_t - \alpha \sum_{t=1}^{T} \frac{\partial L_t}{\partial w_t}$$

where α is the learning rate.

Now suppose that the activation function $f(\cdot)$ is a sigmoid function:

$$f(x) = \frac{1}{1 + e^{-x}}$$

The derivative of the sigmoid function is given by:

$$\frac{\partial f(x)}{\partial x} = f(x) \cdot (1 - f(x))$$

Substituting this into the expression for the gradient of the loss, we get:

$$\frac{\partial L_t}{\partial w_t} = (y_t - y_{target}) \cdot f(w_t \cdot x_t) \cdot (1 - f(w_t \cdot x_t))$$

Since f(x) is always between 0 and 1, the term $f(w_t \cdot x_t) \cdot (1 - f(w_t \cdot x_t))$ is always between 0 and $\frac{1}{4}$. This means that the gradients at each time step are always scaled down by a factor of $\frac{1}{4}$. As the number of time steps increases, the gradients become increasingly small, leading to the vanishing gradient problem. The gradients are multiplied together as they are backpropagated through the network, and if the gradients are small, they will be multiplied many times, leading to an exponentially small

gradient. This can cause the gradients to become so small that they are effectively zero, which can prevent the network from learning.

Tanh and Sigmoid activations saturate easily and are not the best choices for RNN. ReLu is more suited for this task, even if it could lead to an exploding gradient if the weight initialization is incorrect.

LSTM

LSTM are better suited for the task because they treat long-term and short-term dependencies differently, thus being easier to train and avoid gradient problems.

The structure of an LSTM unit consists of:

- An input gate: This gate controls which information from the input should be passed through to the memory cell.
- A forget gate: This gate controls which information from the previous memory cell state should be "forgotten" or discarded.
- A memory cell: This is where the information is stored.
- An output gate: This gate controls which information from the memory cell should be passed through to the output.

Each LSTM unit also has a hidden state, which is used to store information from the previous timestep. At each timestep, the input, forget, and output gates are all controlled by a set of weights and biases.

The input, forget, and output gates are all activated by sigmoid functions, which means that their output is a value between 0 and 1. The memory cell is activated by a tanh function, which means that its output is a value between -1 and 1.

The input, forget, and output gates all take in information from the previous hidden state and the current input, and use these to decide how much of the previous hidden state and current input should be passed through to the memory cell and output.

Constant Error Carousel

LSTM use the Constant Error Carousel for learning in a way that does not cause vanishing gradient.

The CEC method keeps the derivative of the error with respect to the weights constant and equal to 1 by adjusting the gradient of the error during the backpropagation process. This means that the loop has its weight fixed to 1.

In traditional training methods, the gradient of the error is calculated using the chain rule and it depends on the error and the weights of the network. When the error is small, the gradient will also be small and this can cause the vanishing gradient problem in deep networks, such as LSTMs.

The CEC method, instead of minimizing the error, keeps it constant during the training process. To achieve this, the gradient of the error is adjusted such that it is always equal to 1. This ensures that the gradients are not too small and that the weights are updated consistently during training. This can help alleviate the vanishing gradient problem as the gradients are not small, making it easier for the network to learn.

It's important to note that CEC does not solve the vanishing gradient problem completely, but it can mitigate the effects of it.

The LSTM network still learns even if the weights are not updated following the gradient, due to its memory cell structure. The memory cell in an LSTM network is designed to retain information over time, which allows the network to learn long-term dependencies in the input sequences.

In CEC, the network is still able to learn by updating the values of the memory cells through linear activations, even if the weights are not updated following the gradient. The memory cells in the LSTM network are updated during the training process to retain information that is useful for predicting the target output.

It's important to note that CEC is not a traditional training method where the goal is to minimize the error, but rather to maintain a constant error level during training. Even though the weights may not be updated following the gradient, the network is still able to learn by updating the values of the memory cells.

The weights of the LSTM network are still modified during the training process in CEC, but the way they are updated is different from traditional training methods.

In CEC, the weights are updated using backpropagation with a fixed learning rate. However, the learning rate is not a function of the error, as it is in traditional training

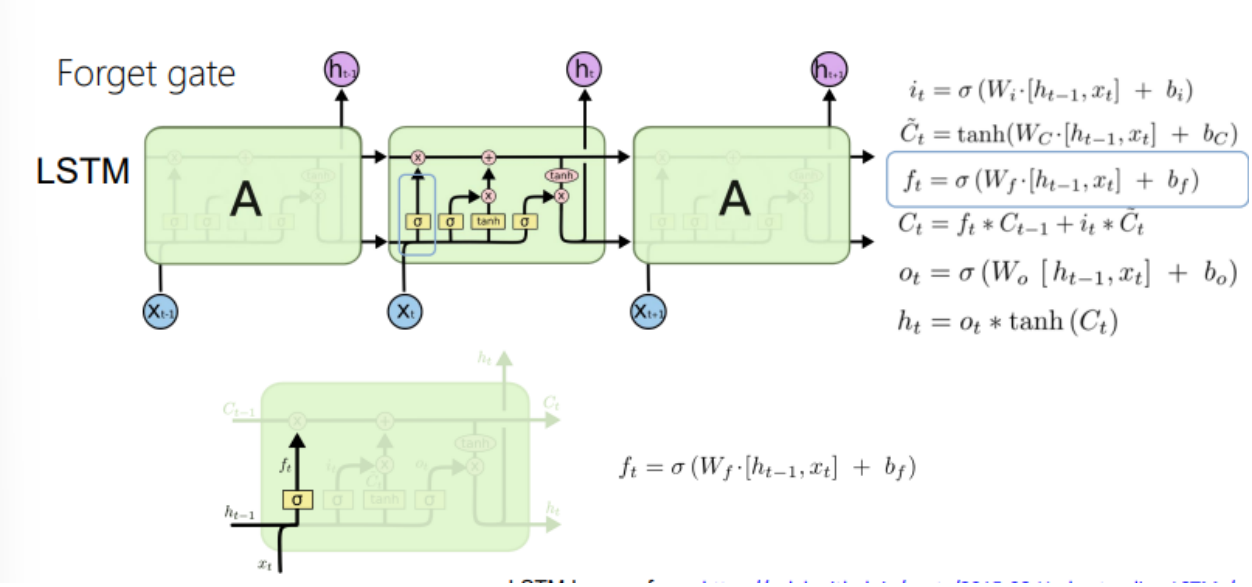
methods. Instead, the gradient of the error is adjusted such that it is always equal to 1. This ensures that the error remains constant during the training process.

When updating the weights, the gradient of the error is multiplied by the fixed learning rate, this is the same as in traditional methods. The difference is that in CEC, the gradient is adjusted such that it is always equal to 1, and this is what makes the error to be constant during training.

So, the weights are still modified according to the learning rate multiplied by 1, but the goal is not to minimize the error, but to maintain a constant error level during training.

Forget Gate

The forget gate in a Long Short-Term Memory (LSTM) unit is a type of gate that determines what information should be discarded or kept in the memory cell of the LSTM unit from the long-term memory. It does this by taking in a sigmoid function of the input and previous hidden state (sum), and using the output to decide which values in the memory cell should be kept or discarded.

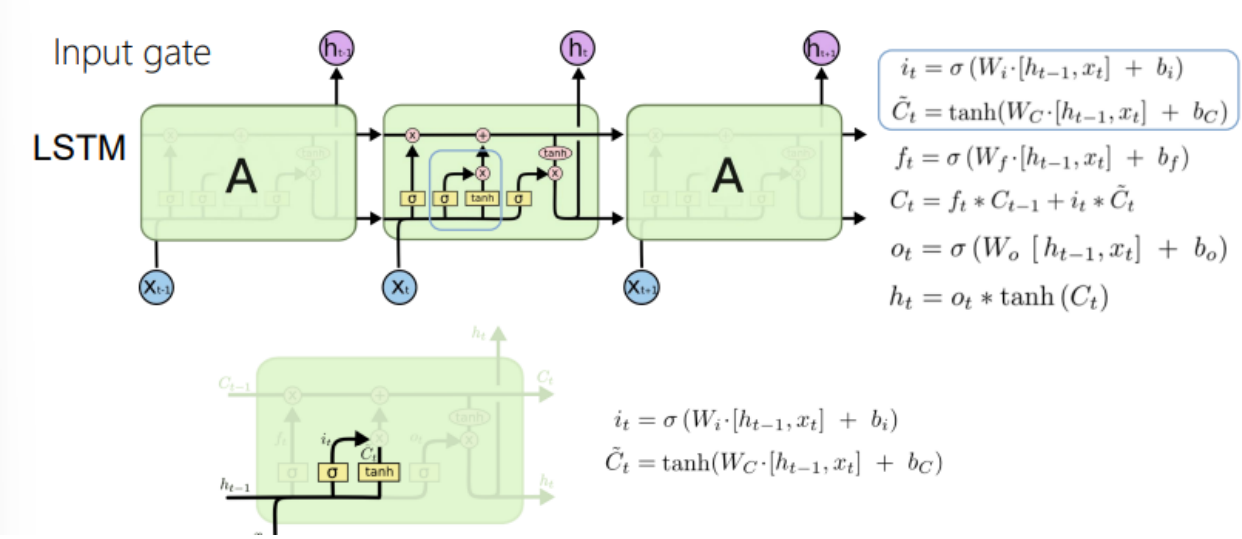


Input Gate

The input gate in an LSTM unit serves to determine which information from the current input should be passed on to the cell state. It can be seen as composed by 2 blocks, each with its own weights:

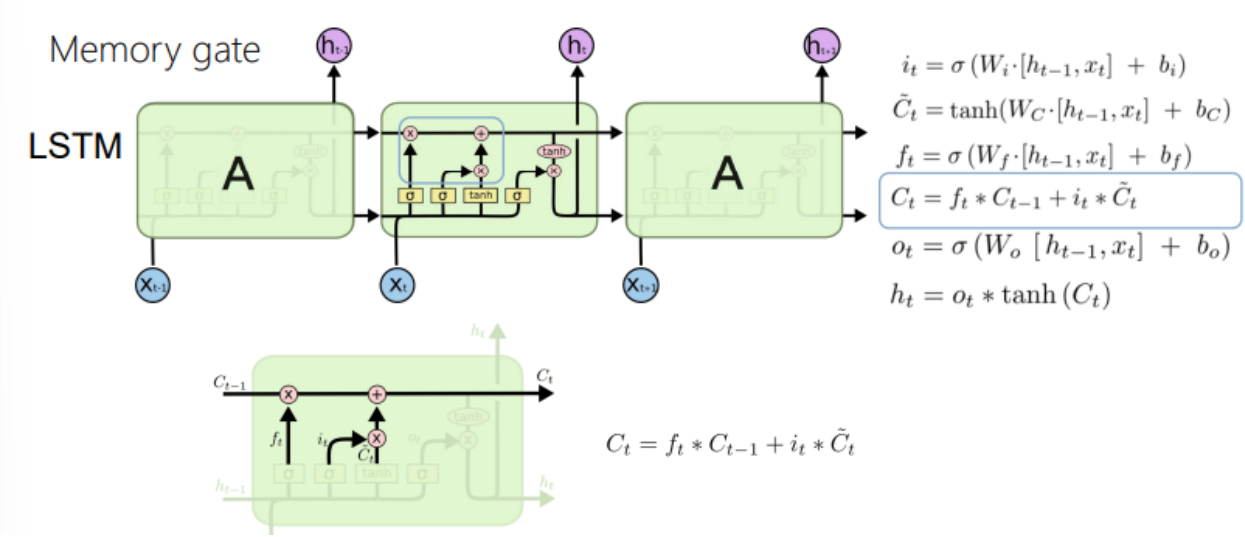
1. Combination (sum) of the input and the short-term memory passed to a tanh to create a potential long-term memory;
2. Combination (sum) of input and short-term memory passed to sigmoid to determine what percentage of that potential memory add to the long-term.

Multiplying these two outputs we obtain the value to add to the long-term memory.



Memory Gate

Updates the hidden state of the cell by combining the forget gate's output (which gives the percentage of long-term memory to be remembered) and the long-term memory, so as to obtain a fraction of it to be propagated. This value is then summed by the input gate's output to generate the new long-term memory that will be propagated to the next cell.

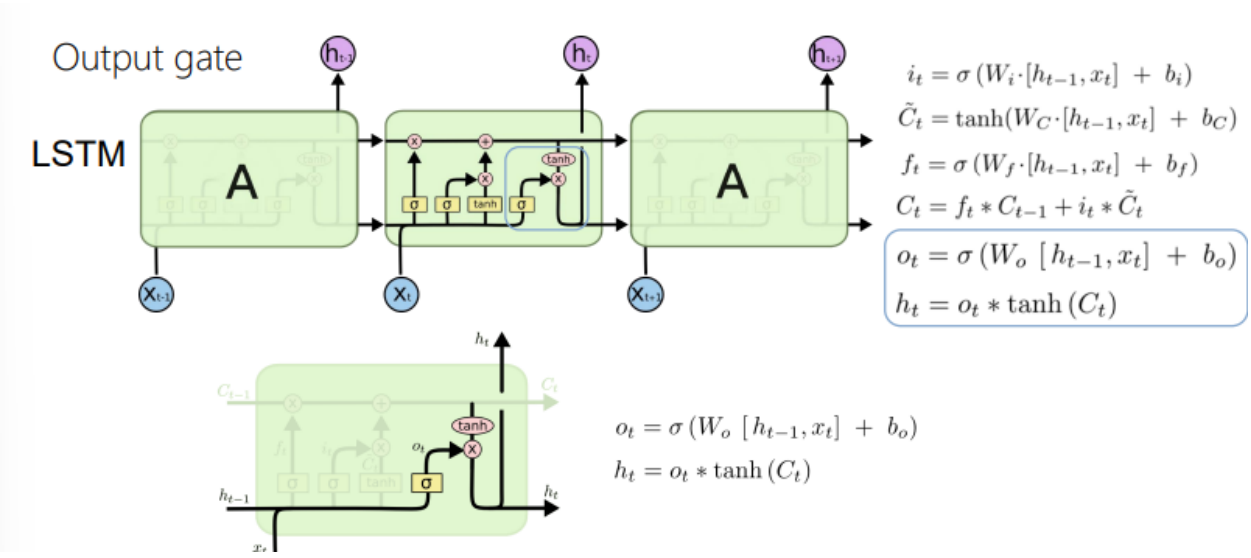


Output Gate

Updates the short-term memory, also using 2 blocks in a progressive way:

1. Sigmoid activated block that considers a sum of the input and short-term memory (potential short-term memory);
2. Tanh that takes as input the long-term memory (% potential memory to remember);

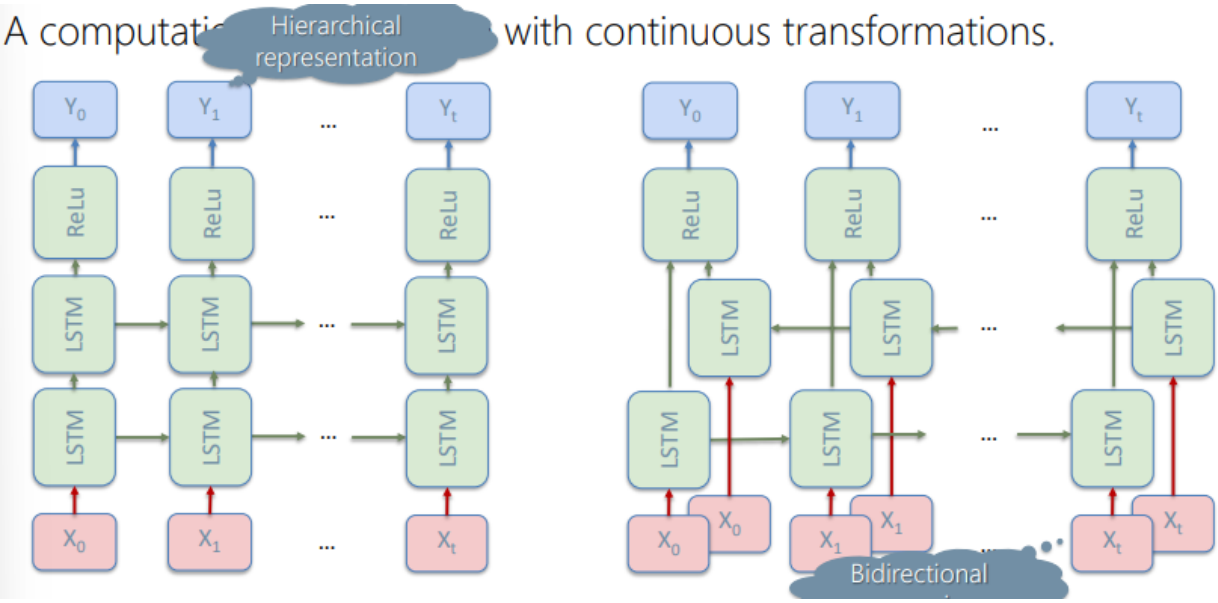
The 2 outputs are then multiplied to have the new short-term memory. The weights of the net are reused for every unroll, and that is the memory cell at every time step.



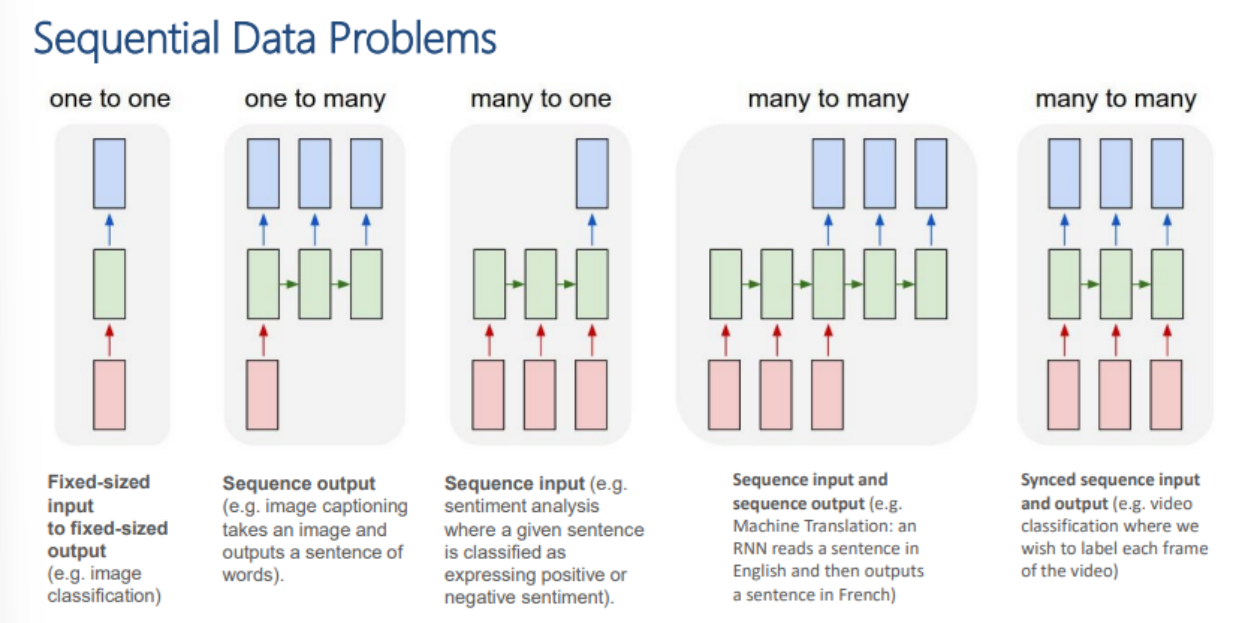
Multiple Layer and Bidirectional LSTM

Multiple layer LSTM is simply a stack of LSTM layers, where the output of one layer becomes the input of the next layer. This allows the LSTM to learn more complex patterns in the data by learning higher level features in the layers closer to the output and lower level features in the layers closer to the input.

Bidirectional LSTM is a variant of LSTM where there are two separate LSTM layers, one for processing the data in the forward direction and another for processing the data in the backward direction. The output of these two LSTM layers are then concatenated and passed to the next layer. This allows the LSTM to capture both past and future context in the data, which can be useful for tasks such as language modeling where the meaning of a word can depend on the words that come before and after it.



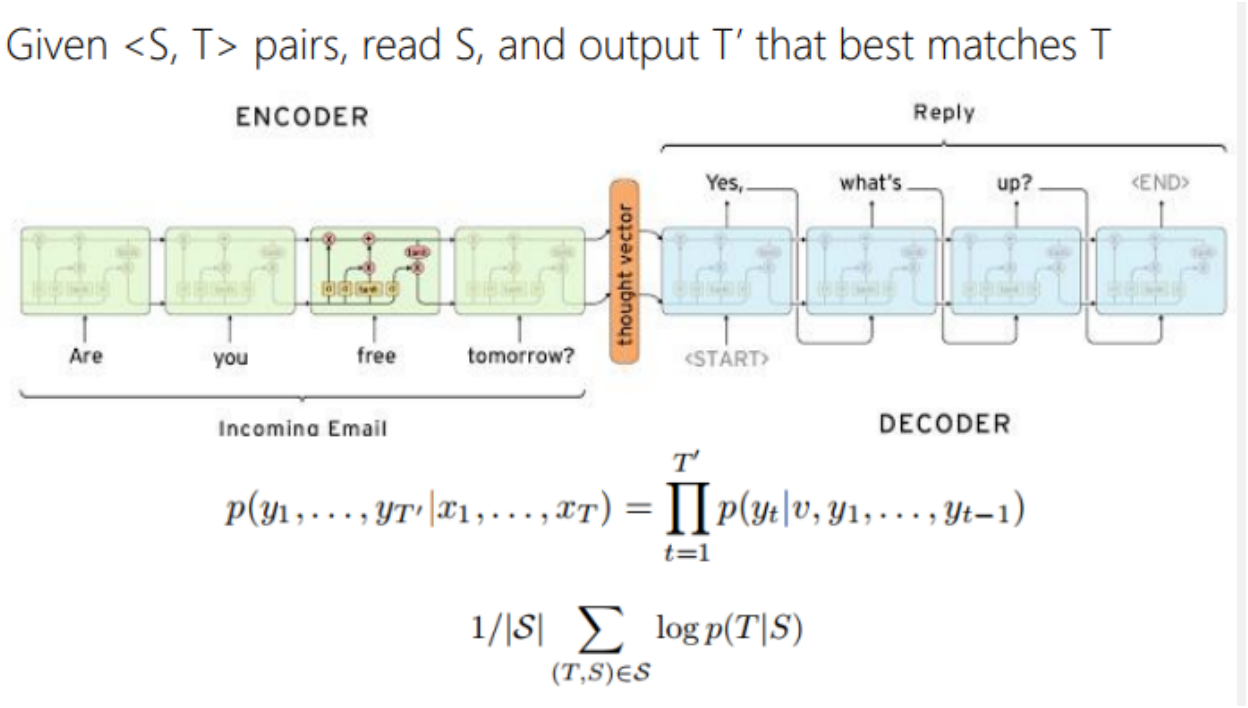
Seq2Seq Learning



Sequence-to-sequence (seq2seq) learning is a type of machine learning model that is used to map a sequence of inputs to a sequence of outputs. It is commonly used for tasks such as machine translation, where the input sequence is a sentence in one language and the output sequence is the corresponding translation in another language. Seq2seq learning involves training an encoder-decoder model, where the encoder processes the input sequence and produces a fixed-length representation of it, called the "context," and the decoder processes the context and produces the output sequence. The encoder and decoder are typically implemented using recurrent neural networks (RNNs) or long short-term memory (LSTM) networks, which are able to capture temporal dependencies in the input and output sequences.

The Seq2Seq model follows the classical encoder decoder architecture:

- During training, the decoder is provided with the correct output sequence as input at each time step, not using the outputs of the previous time step. This allows the decoder to learn to produce the correct output sequence given the context vector.
- During inference, the decoder does not have access to the correct output sequence. Instead, it uses the output from the previous time step as input for the current time step. This allows the decoder to generate the output sequence one time step at a time, using the context vector and its own output from the previous time step as input.



The model's outcome predicts the probability of obtaining an output sequence given an input sequence, and the prediction is trained by maximizing the crossentropy.

Word Embedding

Word embedding is a way to represent words in a continuous vector space, such that similar words are represented by similar vectors. This is useful for many natural language processing tasks, because it allows the model to operate on words in a more meaningful way, rather than simply treating each word as a unique, discrete token. There are many different ways to create word embeddings, but a common approach is to use a neural network to learn the embeddings from large amounts of text data. In this case, the network takes a one-hot encoded word as input and produces a dense, continuous vector as output. This vector is then used to represent the word in downstream tasks.

The performance of real-world applications depends on input encoding:

1. Local representations:
    - N-Grams;
    - Bag-of-words;
    - 1-of-N-coding;
2. Continuous representations:
    - Latent semantic analysis;
    - Latent Dirichet allocation;
    - Distributed representations

N-Gram Language Model

In n-gram models, one-hot vectors can be used to represent the words in a corpus. In this representation, each unique word in the corpus is represented by a vector with a single element set to 1, and all other elements set to 0. For example, if the corpus contains the words "cat", "dog", and "bird", the one-hot vectors for these words might be [1, 0, 0], [0, 1, 0], and [0, 0, 1], respectively. These vectors can be

used as inputs to an n-gram model, which uses the context of the surrounding words to predict the next word in a sequence.

In N-gram language models, the goal is predict the probability of a sentence given the previous words, called “context”.

Determine $P(s = w_1, \dots, w_k)$ in some domain of interest

$$P(s_k) = \prod_i^k P(w_i | w_1, \dots, w_{i-1})$$

In traditional n-gram language models “the probability of a word depends only on the context of n−1 previous words”

$$\hat{P}(s_k) = \prod_i^k P(w_i | w_{i-n+1}, \dots, w_{i-1})$$

Typical ML-smoothing learning process (e.g., Katz 1987):

- compute $\hat{P}(w_i | w_{i-n+1}, \dots, w_{i-1}) = \frac{\#w_{i-n+1}, \dots, w_{i-1}, w_i}{\#w_{i-n+1}, \dots, w_{i-1}}$
- smooth to avoid zero probabilities

Curse of Dimensionality

Let’s assume a 10-gram LM on a corpus of 100.000 unique words (each dimension is a word position in the group that can take 100.000 different values).

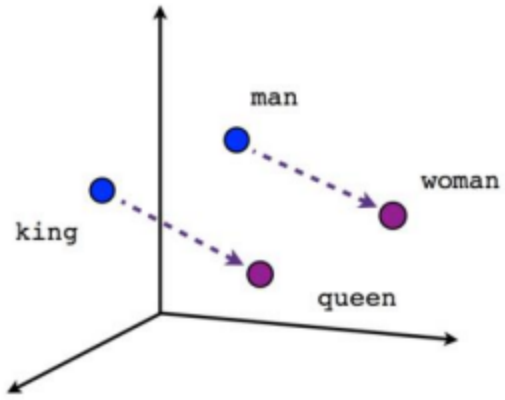
- The model lives in a 10D hypercube where each dimension has 100.000 slots (1o groups of 100.000 words.
- Training the model means assigning a probability to each of the 100.000^{10} slots.
- Probability mass vanishes, so more data is needed to fill the space.
- However, as the number of unique words increases, it becomes increasingly unlikely that the model will have enough data to accurately model the language.

In practice, corpuses can have 10^6 unique words and context is often limited to size 2 (trigram model), but with short context, not a lot of information is captured.

Also, one-hot vector space representation does not capture word similarity, so generalization is needed.

Embedding

Any technique mapping a word (or phrase) from it's original high-dimensional input space (the body of all words) to a lower-dimensional numerical vector space -so one embeds the word in a different space. Closer points are closer in meaning, forming clusters.



Male-Female

The main idea behind word embedding is that similar words should have similar representations in this space. For example, the words "cat", "dog", and "pet" might be represented by vectors that are close together in the space, while the words "car", "bus", and "vehicle" might be represented by vectors that are farther away. This is because these words are semantically similar to each other, while the latter group of words is more related to transportation.

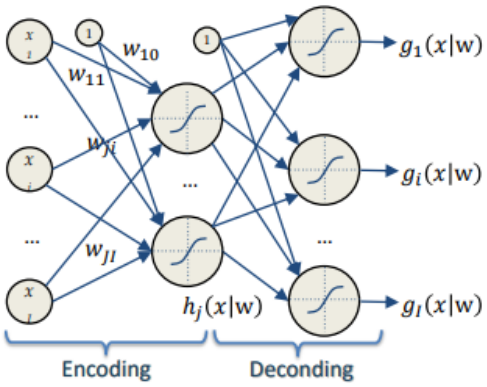
Word embedding is typically achieved through the use of neural networks, which are trained on a large dataset of text to learn the relationships between words. During training, the neural network is presented with pairs of words and their contexts, and it tries to predict the second word based on the first. For example,

given the context "The cat is a" the model might try to predict the word "pet". By training on many such examples, the model is able to learn the relationships between words and their meanings, and it can then use this knowledge to generate numerical representations of words that capture these relationships.

Neural Autoencoder Recall

Network trained to output the input (so to learn the identity function)

- Limited number of units in hidden layers for compressed representation;
- Constrain the representation to be sparse (sparse representation);



$$x \in \Re^I \xrightarrow{enc} h \in \Re^J \xrightarrow{dec} g \in \Re^I$$
$$J \ll I$$

$$E = \|g_i(x_i|w) - x_i\|^2 + \lambda \sum_j \left| h_j \left(\sum_i w_{ji}^{(1)} x_i \right) \right|$$

Reconstruction error: $g_i(x_i|w) \sim x_i$

Sparsity term: $h_j(x_i|w) \sim 0$

Distributed Representation

Each unique word w in a vocabulary V ($\|V\| > 10^6$) is mapped to a continuous m-dimensional space (typically 100 < m < 500). Similar words should be close to each other in feature space.

$$w \in V \xrightarrow{mapping\ C} \Re^m$$

obama = [0 0 ... 0 1 0 ... 0 0] (w_1 obama w_V) → obama = [0.12 ... -0.25] (f_1 f_m)

«one-hot» encoding — feature vector

Fighting the curse of dimensionality with:
• Compression (dimensionality reduction)
• Smoothing (discrete to continuous)
• Densification (sparse to dense)

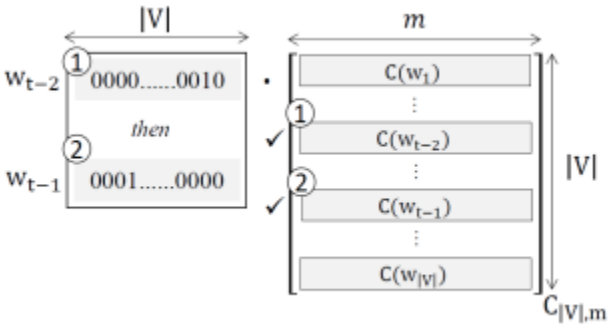
Neural Net Language Model

The Bengio neural network language model is a type of language model that uses a neural network to predict the likelihood of a sequence of words. It was proposed by Y. Bengio et al. in the early 2000s as a way to improve the performance of language models by capturing the underlying structure of language.

The basic idea behind the Bengio model is to use a neural network to learn a continuous, low-dimensional representation of words, called word embeddings. These embeddings capture the semantic relationships between words and can be used to predict the likelihood of a given word given the words that precede it.

Input Layer

The architecture of the Bengio model consists of three main components: an input layer, a hidden layer, and an output layer. The input layer takes a sequence of words as input and converts them into their corresponding word embeddings using a lookup table to store and retrieve them. Word embeddings are dense, continuous-valued vectors that represent words or phrases in a high-dimensional space, and the lookup table is used to map words or phrases to their corresponding embeddings. The lookup table is typically implemented as a matrix, where each row corresponds to a word or phrase and each column corresponds to a dimension of the embedding space. When a word or phrase is input to the model, the lookup table is used to retrieve the corresponding embedding from the matrix and pass it on to the next layer of the network for further processing. During training, the lookup table is updated along with the rest of the model's parameters, in order to learn the best word embeddings for the task at hand.



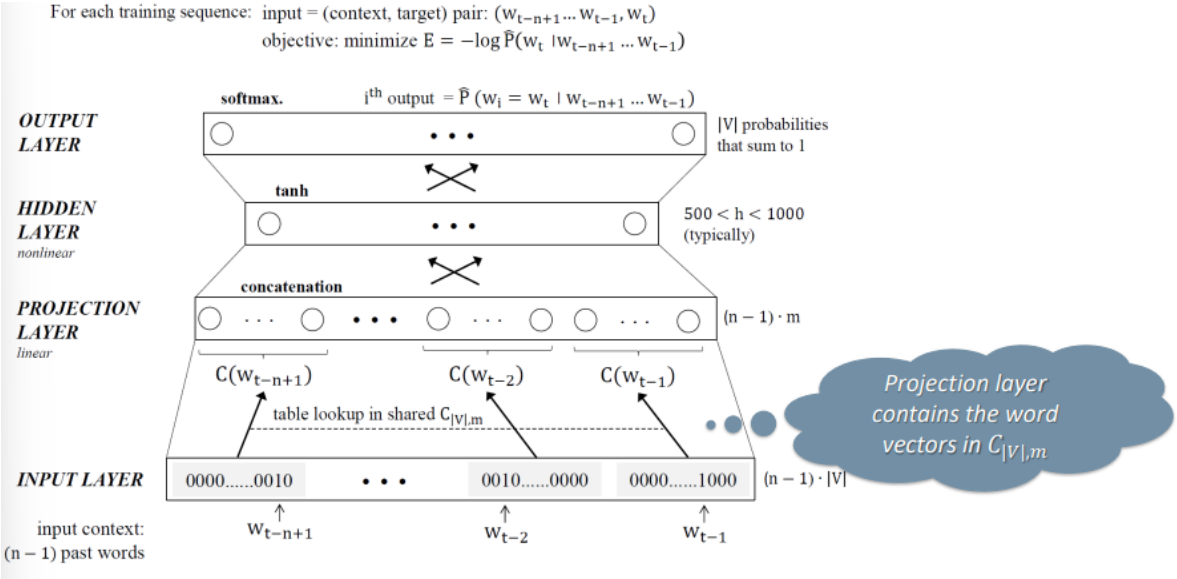
Projection Layer

The projection layer in a neural language model is a linear layer that takes the output of the hidden state of the RNN at each time step and projects it to the size of the vocabulary. This allows the model to predict the next word in the sequence by choosing the word from the vocabulary that has the highest probability according to the projection of the hidden state. The weights of the projection layer are learned during training.

Hidden Layer

The hidden layer then processes these embeddings using one or more layers of neurons, which may include various types of recurrent neural network (RNN) cells such as long short-term memory (LSTM) cells or gated recurrent units (GRUs). The output layer takes the hidden layer's output and uses it to predict the likelihood of the next word in the sequence.

To train the Bengio model, the network is fed a large dataset of text, using couples of <context, target>, and the weights of the network are adjusted to minimize the difference between the predicted likelihood of the next word and the actual next word in the sequence. The trained model can then be used to generate new text by feeding it a seed sequence of words and using the predicted likelihoods of the next word to sample the next word in the sequence.



The complexity of training the neural network language model using stochastic gradient descent is given by the formula:

$$n \times m + n \times m \times h + h \times |V|$$

where:

- n is the number of training examples
- m is the number of words in each training example
- h is the number of hidden units
- V is the vocabulary size

The first term, n * m, represents the complexity of computing the forward pass for each training example. The second term, n * m * h, represents the complexity of computing the gradients for each training example. The third term, h * |V|, represents the complexity of updating the weights during the weight update step.

Output Layer

The output of a language model is typically a probability distribution over a fixed vocabulary of words. Given an input sequence of words, the model estimates the likelihood of each word in the vocabulary given the context of the words that come before it in the sequence. For example, given the input sequence "the cat sat on

the", the model would output a probability distribution over all possible words that could come next in the sequence, such as "mat", "chair", or "roof". The word with the highest probability according to the model's output is considered the most likely to be the next word in the sequence.

Softmax is used to output a multinomial distribution

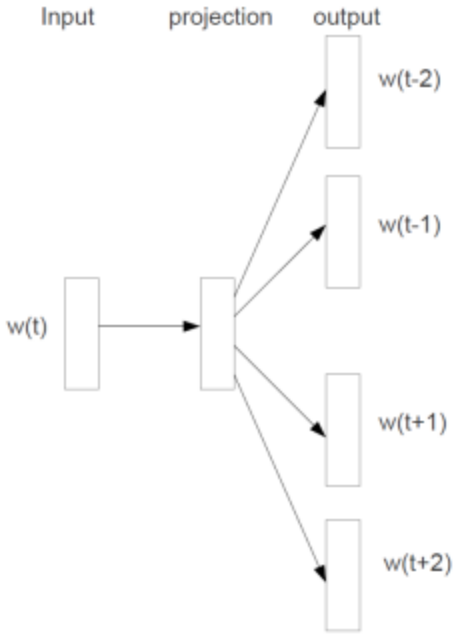
$$\hat{P}(w_i = w_t | w_{t-n+1}, \ldots, w_{t-1}) = \frac{e^{y_{w_i}}}{\sum_{i'}^{|V|} e^{y_{w_{i'}}}}$$

- $y = b + U \cdot \tanh(d + H \cdot x)$
- x is the concatenation $C(w)$ of the context weight vectors
- d and b are biases (respectively h and $|V|$ elements)
- U is the $|V| \times h$ matrix with hidden-to-output weights
- H is the $(h \times (n-1) \cdot m)$ projection-to-hidden weights matrix

One potential drawback of the Bengio neural language model is that it relies on a large number of parameters, which can make it prone to overfitting if not properly regularized. Additionally, the model may be computationally expensive to train, especially when using a large vocabulary size or a large number of hidden units.

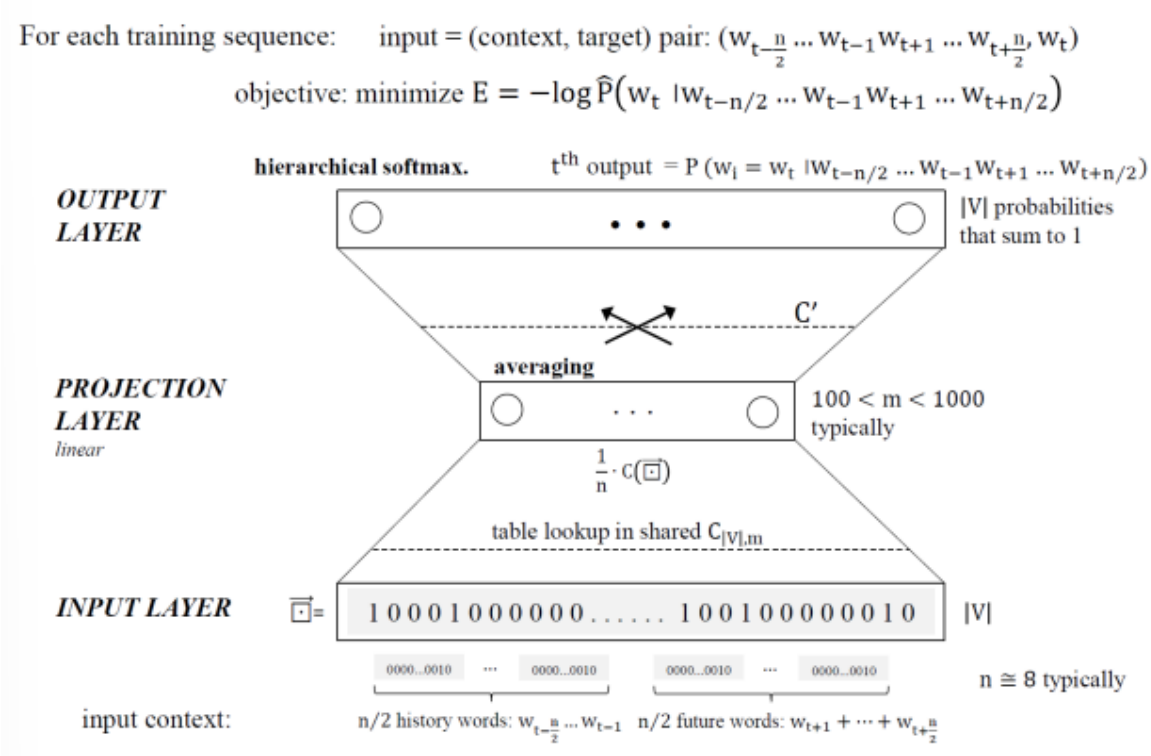
Word2Vec

The word2vec model is a neural network language model that is designed to be simpler and more efficient than other language models, such as the Bengio model. One key difference is that the word2vec model does not have a hidden layer, which allows it to be trained much faster. Another difference is that the projection layer, which maps the input words to the output prediction, is shared among all input words. This means that the projection layer weights are the same for all input words, rather than having separate weights for each input word. Finally, the context in the word2vec model contains words from both the history (previous words) and the future (following words), rather than just the history as in the Bengio model. This allows the model to capture more contextual information about the input words.



Skip-gram architecture

Training works by maximizing the probability of the target word given the context words. This is done using stochastic gradient descent. For each <context, target> pair, the model adjusts the word vectors of the context words in order to maximize the probability of the target word. If the probability of the target word is overestimated, some portion of the context is subtracted from the context word vectors in order to decrease the probability. If the probability of the target word is underestimated, some portion of the context is added to the context word vectors in order to increase the probability. The output of the model is a set of word vectors for each word in the vocabulary, which can be used for various NLP tasks such as language modeling, word similarity, and text classification.



The output is a probability distribution over the vocabulary, which is used to predict the target word given the context words. The probability distribution is generated using the softmax function, which takes as input the dot product of the context and target word vectors, and outputs a value between 0 and 1 for each word in the vocabulary. The goal of training is to adjust the word vectors in a way that maximizes the probability of the correct target word given the context words.

In the word2vec model, the complexity is

$$n \times m + m \times log|V|$$

That's because for each context-target pair, the model updates only the context words. The complexity is determined by the number of context-target pairs (n × m) and the number of context words (m) that need to be updated in each pair. The log|V| term represents the complexity of updating the weights of the projection layer, which is shared among all context words.

Summary

The word2vec model uses word representations (called embeddings) that are learned through the context in which the words appear. These embeddings are learned in such a way that semantically similar words are mapped to nearby points in the embedding space. The dimensions of the space learned by the word2vec model can be thought of as representing different semantic meanings or concepts. Through the use of context, the model can learn to map words to specific points in this space so that semantically similar words are located close to each other. The model can then use this space to make predictions about the probability of a given word occurring in a given context, based on its location in the space relative to the context words.

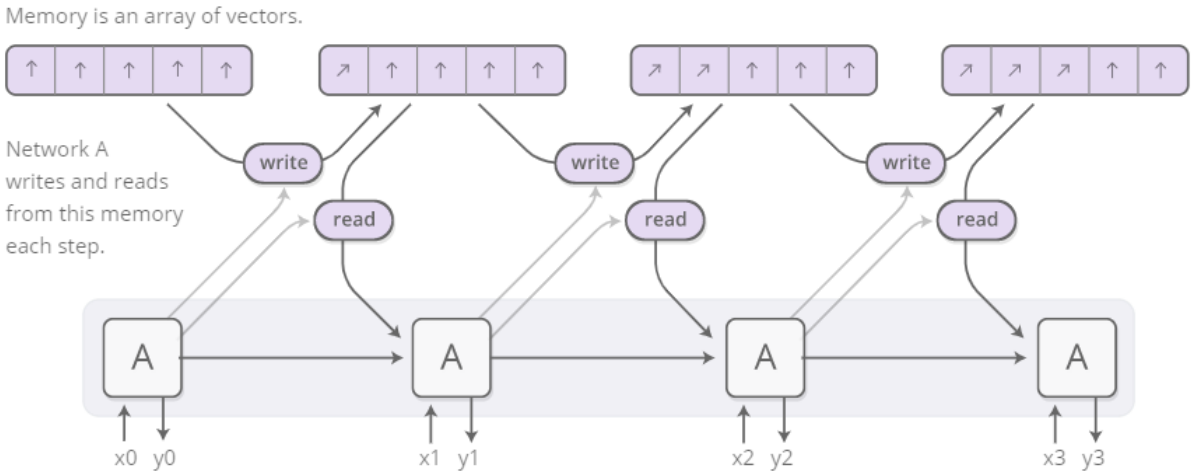
The model is trained using a log-likelihood loss function, which measures how well the model's predictions match the actual data. The complexity of the model is lower than that of the Bengio model because it does not have a hidden layer and because the projection layer is shared across all context-target pairs. This allows the model to be trained on larger amounts of data more efficiently. No need for classifiers, just use cosine distances.

Beyond Seq2Seq Models

Neural Turing Machines

VERY GOOD EXPLANATION AT https://distill.pub/2016/augmented-rnns/

Recurrent Neural Networks have been extended with memory to cope with very long sequences and the encoding bottleneck. A Neural Turing Machine (NTM) is a type of neural network that is designed to be able to read and write to an external memory. The NTM consists of a controller network, which is a neural network that processes input and produces output, and an external memory matrix, which is used to store and retrieve information. The controller network can interact with the memory matrix through read and write operations, allowing it to perform tasks such as sorting, copying, and translation. The NTM has been used in a variety of applications, including machine translation, language modeling, and image captioning.



The memory matrix is typically implemented as a 2-dimensional array, with one dimension representing the rows of the matrix and the other dimension representing the columns. The rows of the matrix correspond to the different memory locations and the columns correspond to the different memory slots. The values in the memory matrix are typically real numbers.

The NTM can use the memory matrix to store and retrieve information that it needs to perform its task. For example, in a sequence learning task, the NTM can use the memory matrix to store information about the previous elements in the sequence. The NTM can use its read and write heads to access this information and use it to make predictions about the next element in the sequence.

The read and write heads are implemented as a set of weight matrices that act as soft attention mechanisms. Each head has a set of addressing parameters that are learned by the neural network, and used to generate a probability distribution over the memory matrix. These probability distributions are then used to determine which elements in the memory matrix the head should read from or write to.

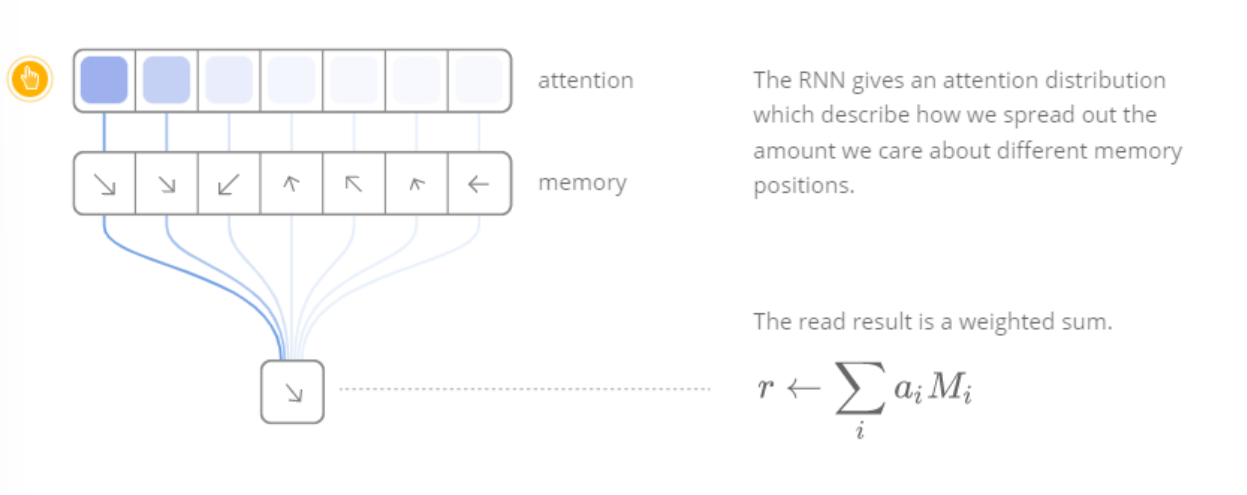
$$output(t) = s(t) \times W + b$$

- Current state of the network (s(t));
- Inputs to the network (x(t));
- Weights of the network (W) and the biases (b);

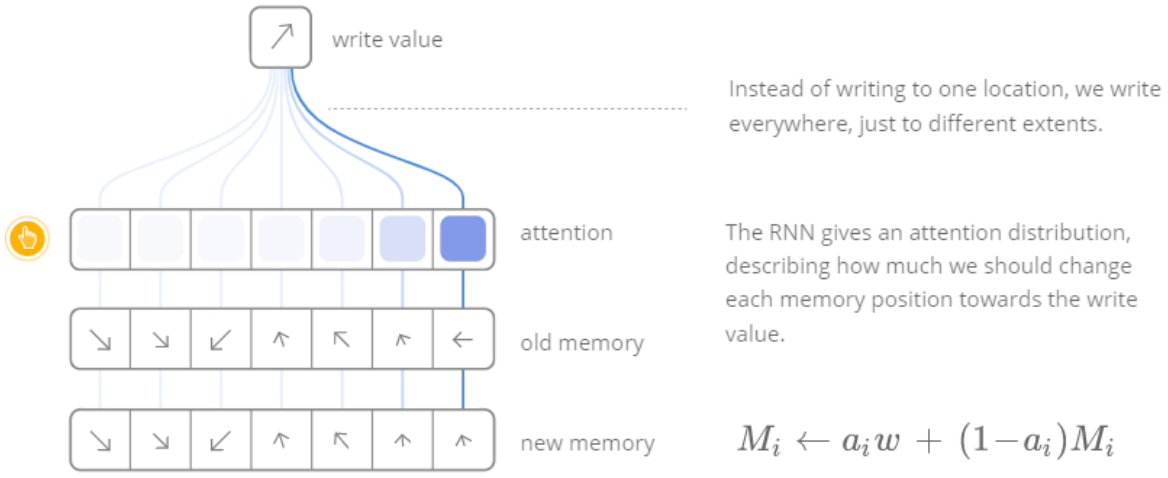
To generate a probability distribution from the output of an RNN, you can use a softmax function, which scales the output values between 0 and 1 and ensures that the values sum to 1:

$$probability\ distribution = softmax(output(t))$$

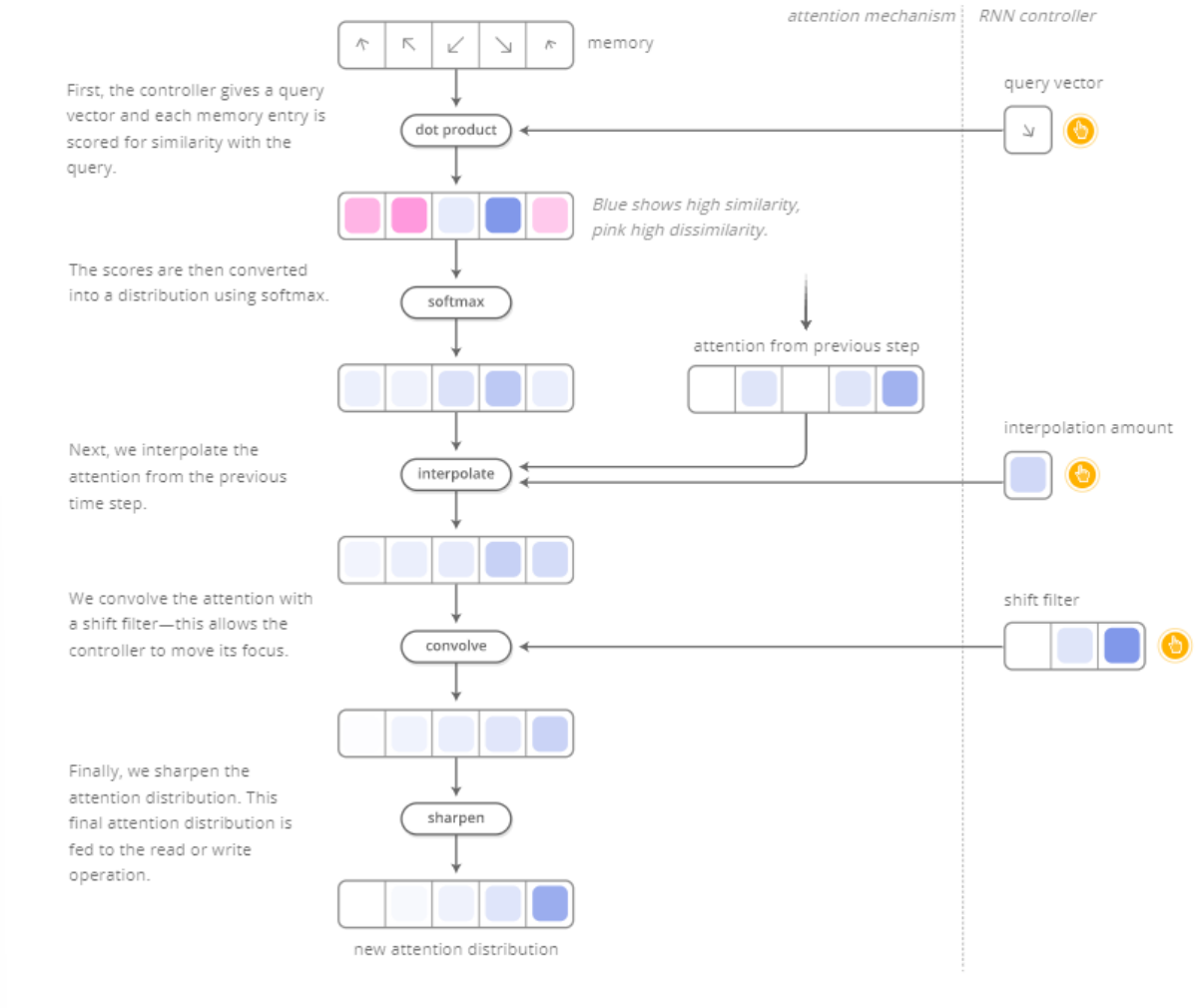
When reading from the memory matrix, the NTM takes the weighted sum of the memory matrix rows according to the read head's probability distribution. This allows the NTM to selectively focus on specific parts of the memory matrix, and read out the relevant information.



When writing to the memory matrix, the NTM uses a similar process. The write head's probability distribution is used to determine which elements in the memory matrix the NTM should write to (in the sense that we write the most, because every position is written at every step), and the weights of the write head are used to determine the amount of information that should be written to each memory location. The NTM can also erase previously written information by erasing the corresponding memory cells, by decreasing their values, or by making them zero by using a erase vector. We do this by having the new value of a position in memory be a convex combination of the old memory content and the write value, with the position between the two decided by the attention weight.



The key feature of NTM is the ability to attend different parts of the memory matrix selectively, which allows the NTM to perform different kind of computations, like copying, sorting and even solving differential equations, by using different neural network architectures and different sets of addressing mechanisms that make use of the memory matrix efficiently, which makes it quite powerful than other sequence models like LSTM, RNN or even GRU.



This capability to read and write allows NTMs to perform many simple algorithms, previously beyond neural networks. For example, they can learn to store a long sequence in memory, and then loop over it, repeating it back repeatedly. They can also mimic a lookup table and perform sorting operations.

Attention Mechanism in Seq2Seq Models

In sequence-to-sequence (Seq2Seq) models with attention, the decoder has an attention mechanism that allows it to focus on different parts of the input sequence while generating the output sequence. This can be particularly useful in situations where the input and output sequences are of different lengths, or when the input sequence is long and the decoder needs to selectively attend to certain parts of it in order to generate the correct output.

The attention mechanism works by first computing a set of attention weights, which are used to weight the different parts of the input sequence when generating the output. These weights are computed using a function that takes as input the current hidden state of the decoder, h_t, and the hidden states of the encoder,

$h_1, h_2, ..., h_n$. The attention weights, $a_1, a_2, ..., a_n$, are then computed applying the softmax function on the scores, as:

$$a_i = \frac{\exp(e_i)}{\sum_{j=1}^{n} \exp(e_j)}$$

where e_i is an attention score computed as:

$$e_i = score(h_t, h_i)$$

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \boldsymbol{W} \bar{\boldsymbol{h}}_s & \text{[Luong's multiplicative style]} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_1} \boldsymbol{h}_t + \boldsymbol{W_2} \bar{\boldsymbol{h}}_s\right) & \text{[Bahdanau's additive style]} \end{cases}$$

The idea is to give the decoder more information about the source sequence (the input to the encoder) when generating each target word. The attention mechanism allows the decoder to "pay attention" to different parts of the source sequence at each time step, rather than only considering the final hidden state of the encoder.

The function $score(\cdot, \cdot)$ can be any function that takes two hidden states as input and outputs a scalar attention score. A common choice is to use a dot product or a multi-layer perceptron (MLP) to compute the attention score.

Once the attention weights have been computed, they are used to weight the hidden states of the encoder when generating the output. This is done by computing the context vector as a weighted sum of the hidden states of the encoder, c_t:

$$c_t = \sum_{i=1}^{n} a_i h_i$$

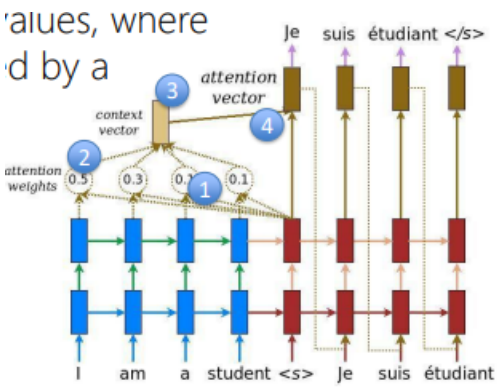
The weighted sum is then concatenated with the current hidden state of the decoder, h_t, and passed through an MLP to generate the output at time step t. The output at time step t is computed as:

$$o_t = MLP(h_t, c_t)$$

The attention mechanism can be thought of as a way to selectively weight different parts of the input sequence when generating the output, allowing the decoder to focus on the most relevant parts of the input when generating each output time step.

The output of a decoder layer at timestep t is influenced not only by its own hidden state ht but also by an "attention context vector" ct, which is a weighted combination of the hidden states of the encoder hs. The attention mechanism works by first computing an "attention score" or "alignment" between each encoder hidden state and the current target hidden state, using a similarity function such as dot product or a feed-forward neural network. These scores are then used to compute a "soft alignment" or "attention weights" alpha, which are used to weight the encoder hidden states when computing the attention context vector ct.

The attention context vector ct is then concatenated with the current target hidden state ht and used as input to the next decoder layer or to compute the final output of the decoder



Chatbots

Chatbots can be defined along at least two dimensions, core algorithm and context handling:

- Generative: encode the question into a context vector and generate the answer word by word using conditioned probability distribution over answer's vocabulary. E.g., an encoder-decoder model.
- Retrieval: rely on knowledge base of question-answer pairs. When a new question comes in, inference phase encodes it in a context vector and by using similarity measure retrieves the top-k neighbor knowledge base items.

Chatbots can be defined along at least two dimensions, core algorithm and context handling:

- Single-turn: build the input vector by considering the incoming question. They may lose important information about the history of the conversation and generate irrelevant responses.

$$\{(q_i, a_i)\}$$

- Multi-turn: the input vector is built by considering a multi-turn conversational context, containing also incoming question.

$$\{([q_{i-2}; a_{i-2}; q_{i-1}; a_{i-1}, q_i], a_i)\}$$

  A hierarchical generative model is a type of machine learning model that is organized in a hierarchy, typically with multiple levels of latent variables or hidden states, where each level models a different abstraction or feature of the data. It is different from the transformer architecture which is a way to deal with sequential data by attention mechanism and a special type of feedforward NN.

  One example of a hierarchical generative model is the Hierarchical Variational Autoencoder (HVAE). In this model, a set of encoder networks are trained to encode the input data at different levels of abstraction, creating a hierarchy of latent variables. A set of decoder networks are then trained to generate data at each level of the hierarchy, using the latent variables as inputs.

  At the lowest level, the encoder network is trained to encode the raw input data (e.g. an image) into a set

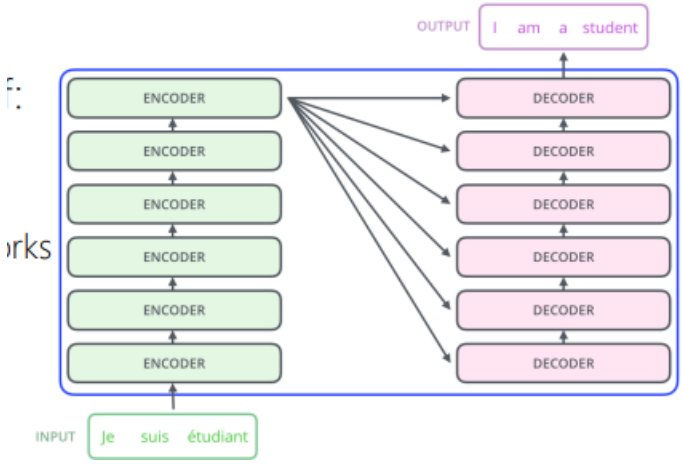
of latent variables that capture the lower level features such as edges and shapes. The decoder network is trained to generate images that are similar to the input data by using these latent variables as inputs. At the next level, the encoder network is trained to encode the output of the first level encoder network into a set of latent variables that capture higher level features such as object parts and poses. The decoder network is trained to generate images that are similar to the input data by using these latent variables as inputs. This process is repeated at higher levels of the hierarchy, where the encoder network encodes the output of the previous level encoder network into latent variables that capture higher level features.

Transformers

A transformer model is a type of neural network architecture that is particularly well-suited for tasks involving sequences of data, such as natural language processing (NLP) and computer vision. The key innovation of the transformer architecture is the use of self-attention mechanisms, which allow the model to weigh the importance of different elements in the input sequence when making predictions.

The transformer architecture was first introduced in a 2017 paper by Google researchers Vaswani et al. called "Attention is All You Need". In this paper, the authors proposed the Transformer model, an architecture that uses self-attention mechanisms to process sequences of data while bypassing the sequential structure. In contrast to sequential architectures such as RNNs and LSTMs, the transformer model allows for parallel processing of the entire sequence, greatly reducing the computational cost.

A transformer model consists of an encoder and a decoder. The encoder takes in a sequence of input data and applies a series of self-attention mechanisms to it, generating a set of hidden representations for each element in the input sequence. The decoder then takes these hidden representations as input and generates an output sequence. The attention mechanism is the core of the transformer and it's the one which makes it so powerful. The mechanism allows the model to attend selectively to different positions in the input sequence, based on the current task or output.



For example, in an NLP task, the input to a transformer model might be a sentence or a document, and the model's task would be to generate a summary or a translation of the input text. In this case, the

attention mechanism would allow the model to weigh the importance of different words in the input text when making its predictions, allowing it to focus on the most relevant information and generate a more accurate summary or translation.

Transformers are made out of different core components, and an important aspect of it is self-attention.

In the encoder, the self-attention mechanism is used to create a representation of the input sequence that captures the dependencies between the different positions within the sequence. This representation is then passed to the decoder, where the decoder self-attention mechanism is used to improve the representation further by considering the dependencies between the different positions within the decoder input sequence and the encoder representation.

The decoder self-attention mechanism is also called masked self-attention because it is designed to prevent the decoder from "peeking" at future tokens in the target sequence by masking out certain positions in the input sequence. This ensures that the decoder can only make predictions based on the information that is available up to the current position in the input sequence.

So in summary, both the encoder and the decoder use self-attention mechanisms, but the purpose is slightly different for each one. In encoder, self-attention is used to build a representation of the input sequence and in decoder, it is used to generate the output sequence.

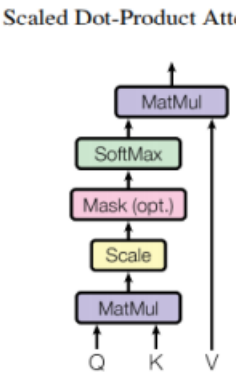
Scaled Dot-Product Attention

In the transformer model, the attention mechanism is based on the scaled dot-product attention. The attention mechanism takes in three inputs: queries, keys, and values.

- Queries: Queries are the inputs that are used to compute the attention. In the transformer model, they are the hidden states of the decoder. The queries are used to compute the attention weights, which indicate how much attention should be given to different parts of the input.
- Keys: Keys are the inputs that are used as references for computing the attention. In the transformer model, they are the hidden states of the encoder. The keys are compared to the queries to compute the attention weights.
- Values: Values are the inputs that are used to compute the output of the attention mechanism. They can be any data that you want to compute the attention on. In the transformer model, they are also the hidden states of the encoder. The values are used to compute the final output of the attention mechanism, which is a weighted sum of the values, where the weights are determined by the queries and keys.

The attention is calculated as a dot product of the query with all the keys, and then the result is scaled by the square root of the dimension of keys. After this, a softmax is applied to the result to obtain the attention weights. The attention mechanism is used to weight the values, and the final output is computed as a weighted sum of the values.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$



In the transformer model, this attention mechanism is used to compute self-attention in the encoder and decoder layers. The encoder layers use self-attention to compute a representation of the input sequence that captures both local and global dependencies, while the decoder layers use self-attention to compute a representation of the output sequence that takes into account both the input sequence and the previous output tokens.

Multi-Head Attention

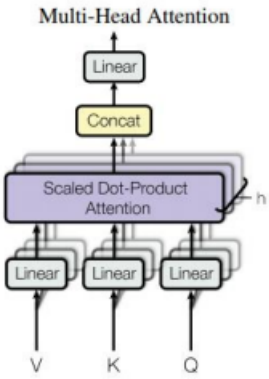
The basic idea behind multi-head attention is to apply attention mechanism multiple times (i.e., multiple "heads"), where each head is focused on different parts of the input sequence. In practice, this is done by applying the attention mechanism to different linear projections of the input, called queries, keys, and values. The attention mechanism is then computed as a dot product between the queries and keys, followed by a softmax to obtain the attention scores. The values are then multiplied by the attention scores to obtain the final attention output.

In the transformer model, the inputs are first linearly transformed into multiple sets of queries, keys, and values, known as multi-heads. Then, these multi-heads are used to calculate multiple attention scores, and finally, all the attention scores are concatenated and transformed again. This architecture allows the model to capture various different dependencies, by using the different attention heads.

Formally, the multi-head attention can be formulated as follows:
Given the queries Q, keys K, and values V, multi-head attention is defined as:

$$MultiHead(Q, K, V) = Concat(head_1, ..head_h)W_o$$

where $head_i = Attention(QW_i^q, KW_i^k, VW_i^v)$ is the output of attention on the i-th head, and W_o, W_i^q, W_i^k, W_i^v are trainable matrices.



Position-wise FFN

In transformer models, the position-wise feed-forward network (FFN) is a component that is applied to each individual position of the input sequence. The purpose of the position-wise FFN is to add additional information to the input that captures the relationships between the different positions in the sequence.

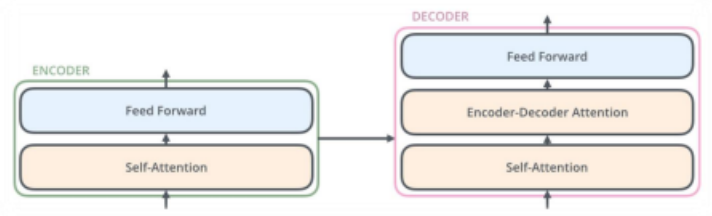
The position-wise FFN is typically a simple feed-forward neural network with two linear layers, each followed by a non-linear activation function such as ReLU. The first linear layer projects the input vector to a higher-dimensional space, and the second linear layer then projects it back to the original dimension.

The position-wise FFN can be represented mathematically as:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

where x is the input vector, W_1 and W_2 are the weight matrices for the two linear layers, b_1 and b_2 are the bias terms, and max(0, x) is the ReLU activation function.

The position-wise FFN allows the transformer model to capture information that is not just based on the relationships between the input words, but also on the relationships between different positions in the input sequence. This improves the model's ability to understand the meaning of the input and make more accurate predictions. The main function of the positionwise ffnn is to add a non-linearity to the overall model, this is useful to make the model learn more complex representations of the input data, it also helps to make the model learn relations between different parts of the input.



Embedding and Softmax

In a transformer model, the input is first passed through an embedding layer, where each token is mapped to a dense vector representation called an "embedding". These embeddings are then used as the input to the transformer model.

The output of the transformer model is passed through a softmax activation function, which converts the output into a probability distribution over the vocabulary. The softmax function essentially outputs a probability for each word in the vocabulary, given the input sequence. The highest probability corresponds to the predicted next word in the sequence.

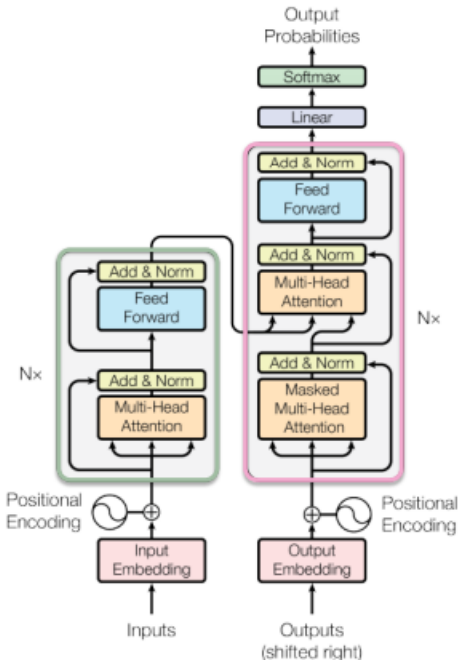
More formally, the input to the transformer model is a sequence of token indices (e.g. integers), and the embedding layer maps these indices to a sequence of embeddings, which are passed through the transformer model. The transformer model then produces a sequence of logits, which are passed through a softmax activation function to produce a probability distribution over the vocabulary. The word with the highest probability is chosen as the predicted word.

The softmax function maps the logits of the transformer model to a probability distribution over the vocabulary:

$$Softmax(logits) = exp(logits) / \sum exp(logits)$$

where logits is the output from the transformer model, and the denominator term is the normalization constant which ensures the sum of the probabilities across the vocabulary is 1.

One important thing to note is that in large scale models the use of 'hierarchical softmax' is used that replaces softmax with a binary tree-like structure that reduce the computational cost of computing the normalization constant by reducing the number of exponentiation operations and also reduce the memory requirement.



Positional Encoding

In transformer models, the attention mechanism works by comparing the relationships between words based on their position in the input sequence. However, this position information is not explicitly provided to the model, as the input is passed through an embedding layer before being processed by the self-attention mechanism. To ensure that the model takes into account the relative position of words in the input sequence, a technique called positional encoding is used.

Positional encoding is a technique used in transformer models to provide information about the position of the input elements in the sequence to the model. It is usually added to the input embeddings before they are fed into the transformer layers. The positional encoding is designed to add a unique vector to each position of the input sequence, which encodes information about the position of that element in the sequence. This allows the transformer model to distinguish between elements that have similar features but appear at different positions in the input.

The basic idea is to add a fixed vector (the positional encoding) to the word embeddings, where the value of the vector depends on the position of the word in the input sequence. This is typically done by using a mathematical function that maps the position of a word to a unique vector.

One common method of positional encoding is to use sine and cosine functions of different frequencies to generate a unique vector for each position in the input sequence. These functions are applied to the position of each word, and the results are concatenated to form the positional encoding vectors. For example, for a sequence of length L and a dimension D, the positional encoding for the i-th position can be represented as:

$$PE_i = [sin(i/10000^{(2j/D)}) for\ j\ in\ range(D/2)] + [cos(i/10000^{(2j/D)}) for\ j\ in\ range(D/2)]$$

where PE_i is the position encoding vector for the i-th position, and j ranges over the dimensions of the position encoding vector.

This way, the final embeddings of a word, is just the sum of the word's embeddings and position embeddings. this way the transformer is aware of the relative position of each word and can attend correctly.